M2177.003100
Deep Learning

[7: Recurrent Neural Nets (Part 1)]

Electrical and Computer Engineering
Seoul National University

(last compiled at 20:37:00 on 2018/10/14)

Outline

References

- *Deep Learning* by Goodfellow, Bengio and Courville ▸ Link
  - Chapter 10 Sequence Modeling: Recurrent and Recursive Nets

- online resources:
  - *Deep Learning Specialization (coursera)* ▸ Link
  - *Stanford CS231n: CNN for Visual Recognition* ▸ Link
  - *Machine Learning Yearning* ▸ Link

Outline

Recurrent neural networks (RNNs)

순차적(sequential) data를 처리하기위한 모델 => data 사이의 dependency가 있음

- a family of neural networks for processing sequential data

cnn => grid 형태의 data에 최적화,
rnn => 시계열 형태의 data에 최적화

- comparison:

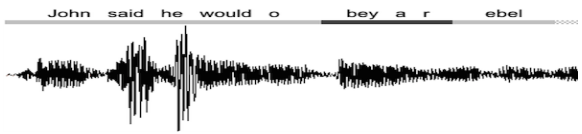
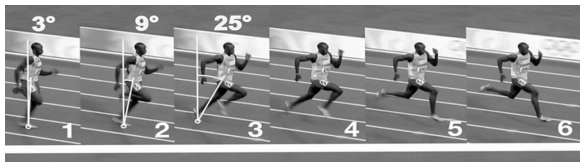
	CNN	RNN
specialized for	grid of values **X** (*e.g.* image)	sequence of values[1] $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$
scale to	wide and tall images	long sequences[2]
can process	images of variable size	sequences of variable length

cnn => minibatch에서의 data size가 같음 but rnn은 다름(ex) 문장)

[1] RNNs usually operate on minibatches of sequences with a different sequence length τ for each member of the minibatch, but we omit minibatch indices to simplify notation

[2] much longer sequences than would be practical for networks without sequence-based specialization

Sequential data examples 시간 관련 => 전부다 rnn과 연관 가능

example	x	y
speech recognition		"John said he would obey a rebel."
music generation		
sentiment classification	"This movie is really, really fantastic."	★★★★★
DNA sequence	AAGTCTAACGTAAACGTCCCT	AAGTCTAACGTAAACGTCCCT
machine translation	¿Estás disfrutando de esta clase?	Are you enjoying this class?
video activity recognition		running
name entity recognition	Yesterday, Harry Potter met Hermione Granger.	Yesterday, Harry Potter met Hermione Granger.

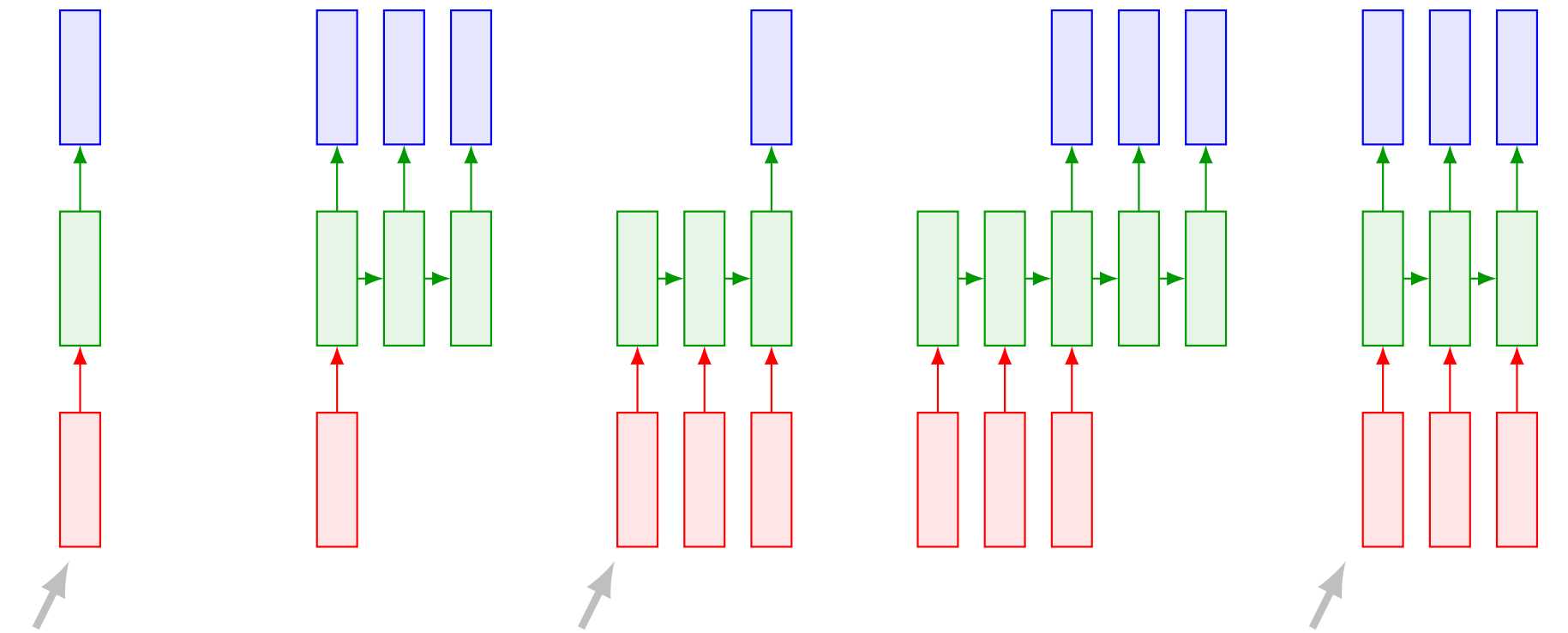
(source: coursera)

RNN modeling capability

입출력 길이가 같음 or 입출력 길이가 다름

e.g. **image captioning**
image → sentence

e.g. **machine translation**
seq of words → seq of words



vanilla neural net
== MLP

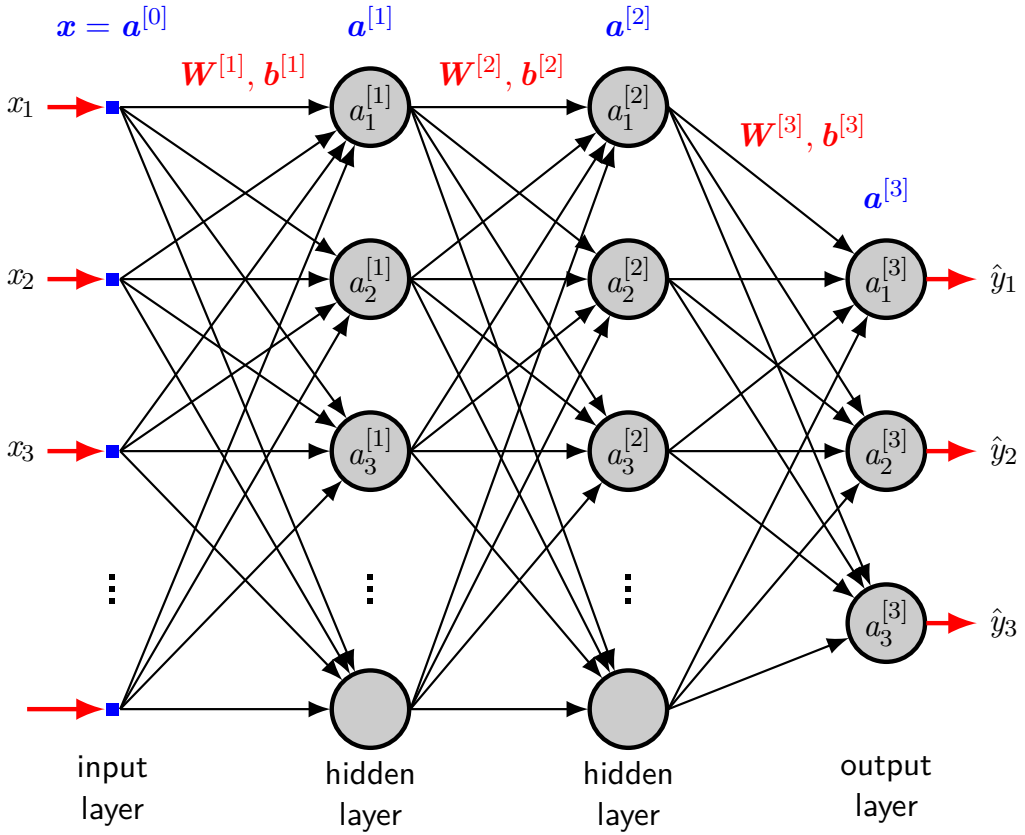
e.g. **sentiment classification**
sentence → sentiment

e.g. **video frame classification**
seq of frame → seq of class

(source: cs231n)

Deep learning RNNaissance

Why not a vanilla neural net?



- problems

cnn => parameter sharing함(param 숫자를 줄임으로써 overfitting 방지 가능)

  - input/output: can be of variable lengths
  - no sharing of features learned across different positions of sequence

Parameter sharing

- means: sharing parameters across different parts of a model
  - CNN: same kernel across image regions
  - RNN: same weights across time steps
- particularly important when
  - a specific piece of info can occur at multiple positions within the sequence
- benefits: the model can handle and generalize across
  - examples of different forms (*e.g.* different lengths in RNN)
- if we use separate parameters for each time index => param숫자가 말도 안되게 많아짐, generality 감소
  - cannot generalize to sequence lengths not seen during training
  - cannot share statistical strength across
    - ▷ different sequence lengths and different positions in time

Comparison

- 1D convolution over temporal sequences vs RNN:

	1D convolution	RNN
output	sequence	sequence
each member of output	a function of a small number of neighboring members of input	a function of the previous members of output
parameter sharing	same kernel at each time step	same update rule applied to outputs
depth	shallow	deep

Outline

Dynamical systems

- classical form of a dynamical system:

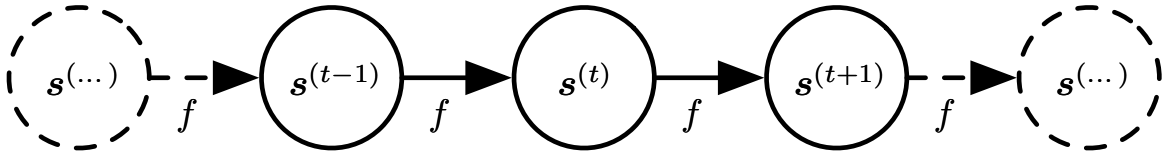
$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}) \tag{1}$$
$$= f_{\boldsymbol{\theta}}(\boldsymbol{s}^{(t-1)}) \quad \text{(simplified notation)}$$

  - $\boldsymbol{s}^{(t)}$: state of the system
  - function f maps state at $t-1$ to state at t (parameterized by $\boldsymbol{\theta}$)
  - "recurrent"
    - definition of s at time t refers back to same definition at time $t-1$

- unfolding(1) for $\tau = 3$ time steps gives unfolding => 풀어주는

$$\boldsymbol{s}^{(3)} = f_{\boldsymbol{\theta}}(\boldsymbol{s}^{(2)}) = f_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\boldsymbol{s}^{(1)}))$$

  - does not involve recurrence f의 parameter => time independent(즉 매 시간마다 sharing)
  - can be represented by a directed acyclic computational graph



  - note: same parameter $\boldsymbol{\theta}$ used for all time steps

- another example: consider a dynamical system

$$\boldsymbol{s}^{(t)} = f_{\boldsymbol{\theta}}(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}) \tag{2}$$

  - driven by external signal $\boldsymbol{x}^{(t)}$

$\Rightarrow$ state now contains information about the whole past sequence

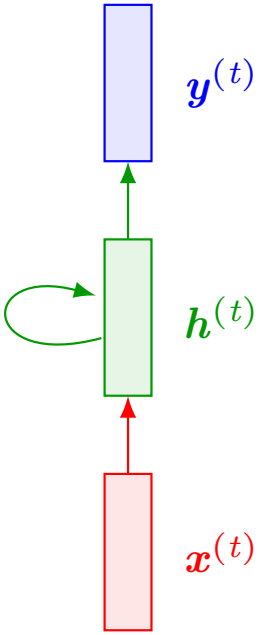
Outline

Recurrent neural net (RNN)

- rewrite (2) using variable $\boldsymbol{h}$ to represent the state:

$$\underbrace{\boldsymbol{h}^{(t)}}_{\text{new state}} = f_{\boldsymbol{\theta}}(\underbrace{\boldsymbol{h}^{(t-1)}}_{\text{old state}}, \underbrace{\boldsymbol{x}^{(t)}}_{\text{input}})$$

  - the state consists of hidden units
  - same f, same $\boldsymbol{\theta}$ at every time step



- typical RNNs add extra architectural features

  e.g. output layers: read info out of $\boldsymbol{h}$ to make predictions

$$\boldsymbol{y}^{(t)} = \sigma(\boldsymbol{W}\boldsymbol{h}^{(t)})$$

- what is the role of hidden units?

Hidden states as a lossy summary

hidden state => 현재까지의 state에 대한 lossy summary

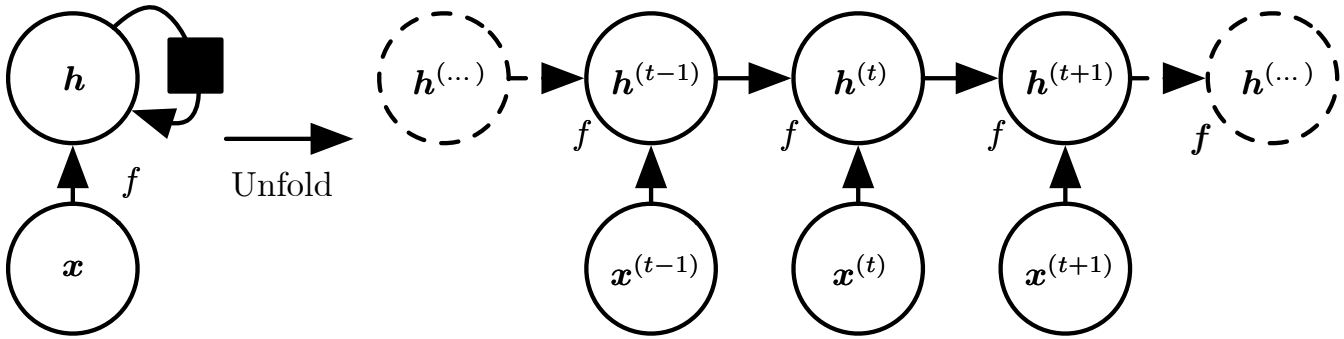
- assume a task: predicting future from past using an RNN
- the net learns to use $\boldsymbol{h}^{(t)}$ as a "lossy summary"
  - of task-relevant aspects of the past sequence of inputs up to t
- this summary (=> 모든 특성을 다 저장하는게 아니라 목적에 필요한 내용만 저장)
  - maps $\underbrace{\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}}_{\text{arbitrary length sequence}}$ to $\underbrace{\boldsymbol{h}^{(t)}}_{\text{fixed length vector}}$ $\Rightarrow$ necessarily lossy
  - may selectively keep some aspects of past sequence (more precisely)

e.g. language modeling (predicting next word given previous words)

$\rightarrow$ store only enough info for prediction (not entire input sentence)

- most demanding situation (*e.g.* autoencoders in ch 14)
  - $\boldsymbol{h}^{(t)}$ should be rich enough to approximately recover the input sequence

Two ways to draw an RNN



1. recurrent graph (circuit diagram): left
   - black square: a delay of a single time step

2. unfolded graph: right
   - each node: now associated with one particular time instance

- this RNN just processes information from input x by
  - incorporating it into state h that is passed forward through time

- we can represent unfolded recurrence after t steps with a function $g^{(t)}$:

문제: g(t) => time 'dependent'

해결: f => time 'independent'

$$\boldsymbol{h}^{(t)} = g^{(t)}(\overbrace{\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}}^{\text{the whole past sequence as input}})$$
$$= f_{\boldsymbol{\theta}}(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)})$$

- we learn f rather than $g^{(t)}$
  - f: a single model that operates on all time steps/all sequence lengths
    - same input size regardless of sequence length[3]
  - $g^{(t)}$: a separate model for all possible time steps

- advantages of learning the same transition function f
  - generalization (to sequence lengths not seen in training)
  - requires far fewer training examples (thanks to parameter sharing)

[3] f is specified in terms of transition from one state to another rather than a variable-length history of states

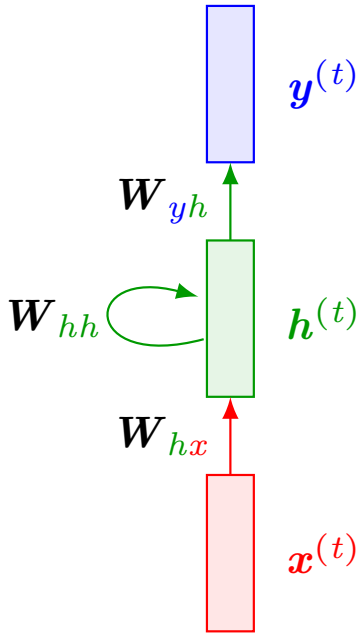
Simple/vanilla/Elman RNN

$$\underbrace{\boldsymbol{h}^{(t)}}_{\text{new state}} = f_{\boldsymbol{\theta}}(\underbrace{\boldsymbol{h}^{(t-1)}}_{\text{old state}}, \underbrace{\boldsymbol{x}^{(t)}}_{\text{input}})$$

Wyx => y: 출력, x: 입력

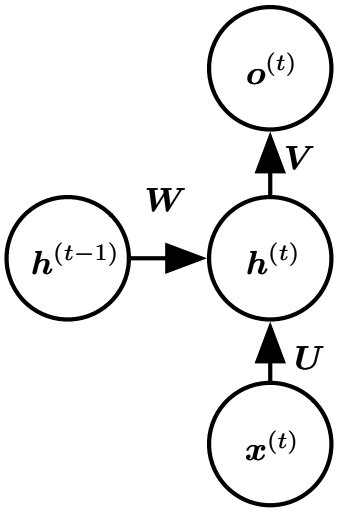
$$= \tanh(\boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{hx}\boldsymbol{x}^{(t)} + \boldsymbol{b}_h)$$

$$\boldsymbol{y}^{(t)} = \text{softmax}(\boldsymbol{W}_{yh}\boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$



- system state
  - consists of a single hidden vector $\boldsymbol{h}^{(t)}$
- parameter sharing
  - same parameters ($\underbrace{\boldsymbol{W}_{hx}, \boldsymbol{W}_{hh}, \boldsymbol{W}_{yh}}_{\text{weight matrices}}$, $\underbrace{\boldsymbol{b}_h, \boldsymbol{b}_y}_{\text{bias vectors}}$) at every time step
- activation: may very from architecture to architecture
  very => vary(오타)

Alternative notation (textbook)



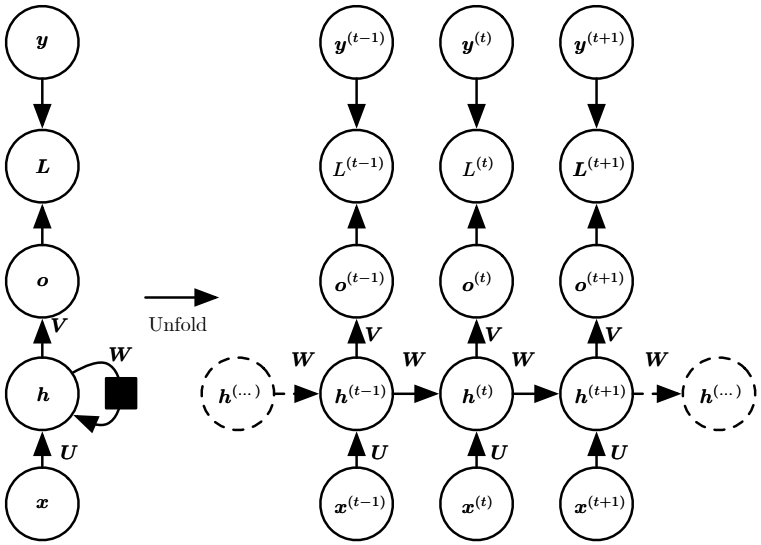
$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}$$
$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)})$$
$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}$$
$$\hat{\boldsymbol{y}}^{(t)} = \mathrm{softmax}(\boldsymbol{o}^{(t)})$$

- $\boldsymbol{b}, \boldsymbol{c}$: bias vectors
- $\underbrace{\boldsymbol{U}}_{\text{input-to-hidden}}$, $\underbrace{\boldsymbol{V}}_{\text{hidden-to-output}}$, $\underbrace{\boldsymbol{W}}_{\text{hidden-to-hidden}}$: weight matrices

Two types of recurrence structures

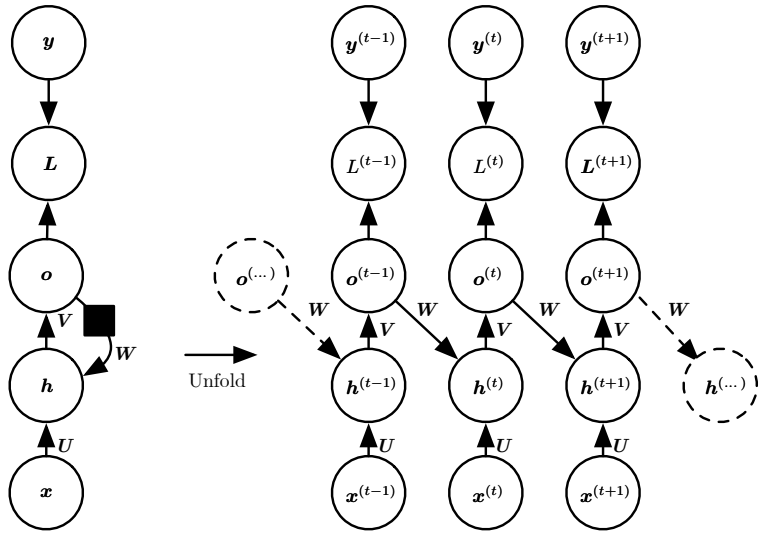
얘는 parallel training 힘듬

- hidden-to-hidden



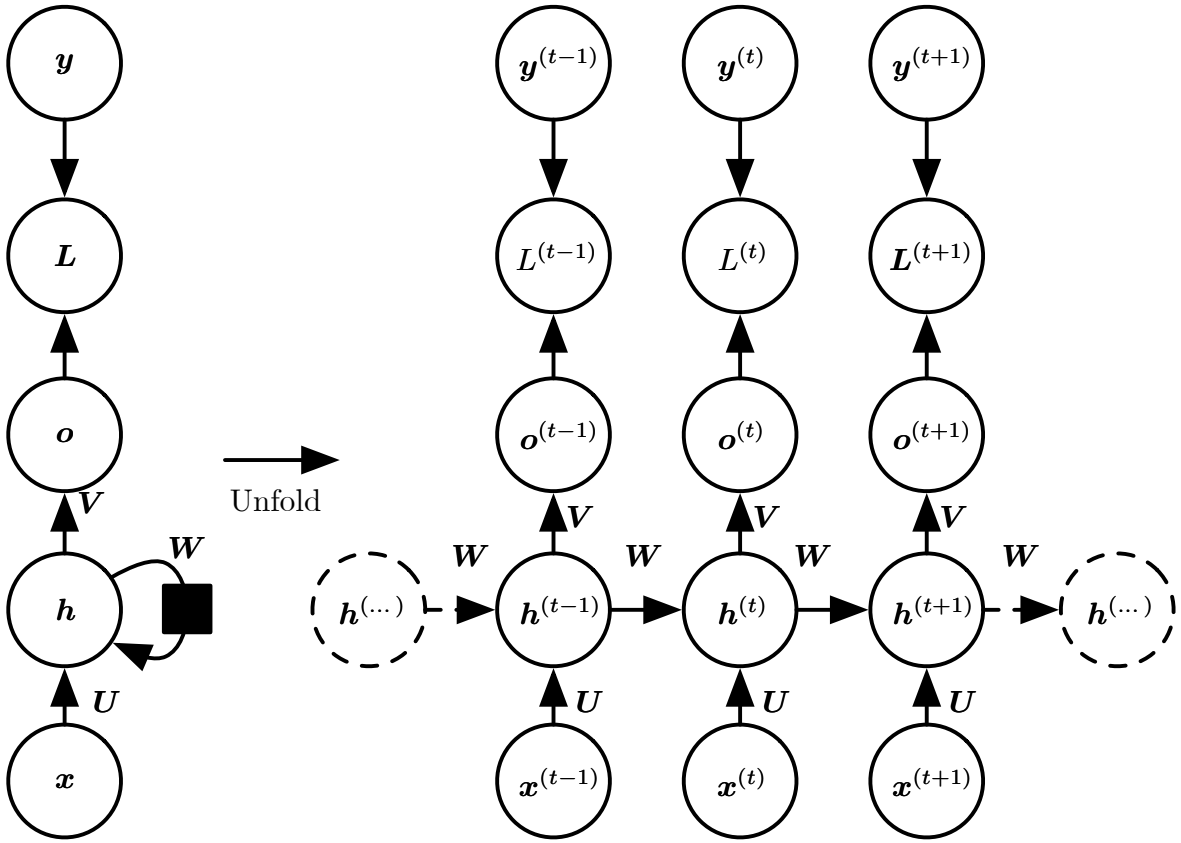
얘는 parallel training 가능

- output-to-hidden



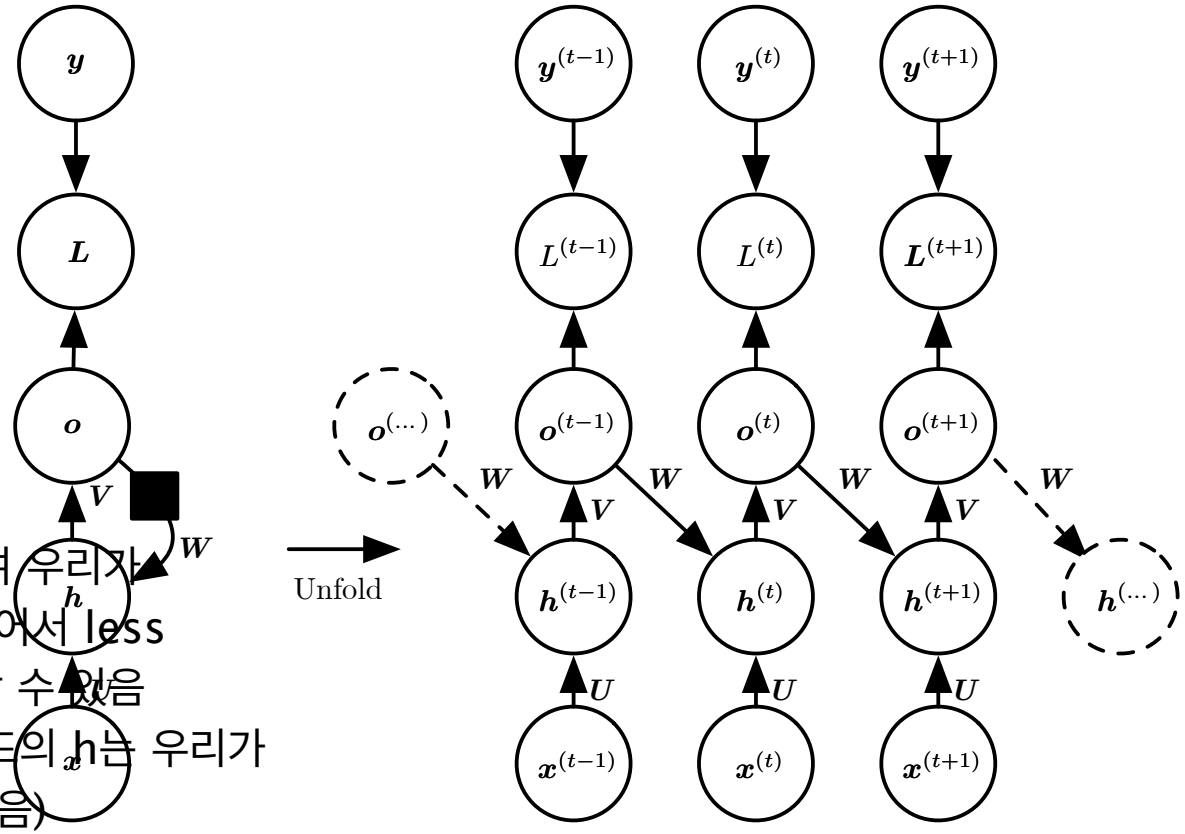
- it is possible to have both types simultaneously

hidden-to-hidden recurrence



- most popular, representative architecture
  - a universal RNN that simulates a turing machine (see textbook)

output-to-hidden recurrence



o(output)에 대하여 우리가 control할 수 가 없어서 less expressive라고 할 수 있음 (반면에 이전슬라이드의 h는 우리가 마음대로 바꿀 수 있음)

- less expressive than the previous type (unless $\boldsymbol{o}$ is very high-dim and rich)
  - but train is easier (each time step trained independently in parallel)

Outline

RNN training

- three types
  - teacher forcing => 답을 따라가는 것
  - backprop through time (BPTT)
  - real-time recurrent learning (RTRL) => 거의 사용하지 않음(계산량이 너무 많음)

- teacher forcing
  - for nets with output-hidden but without hidden-hidden connections
  - motivated by computational burden of BPTT

- BPTT: most widely used (can train hidden-to-hidden architecture)
  - essentially truncated backprop on unfolded graph
    (time이 너무 범위가 넓으므로)
- RTRL: training while forward prop
  - no backprop needed but computationally expensive (not used in practice)

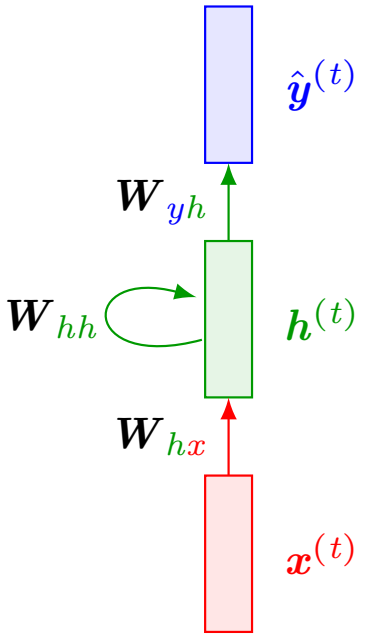
Forward propagation

NN에서의 forward prop => loss에 대한 gradient를 계산할 수 있게함 => 그 이후 back prop 가능하게함

- assume a vanilla RNN
  - for $t = 0$: specify initial state $\boldsymbol{h}^{(0)}$
  - for each time step (from $t = 1$ to $t = \tau$): apply updates

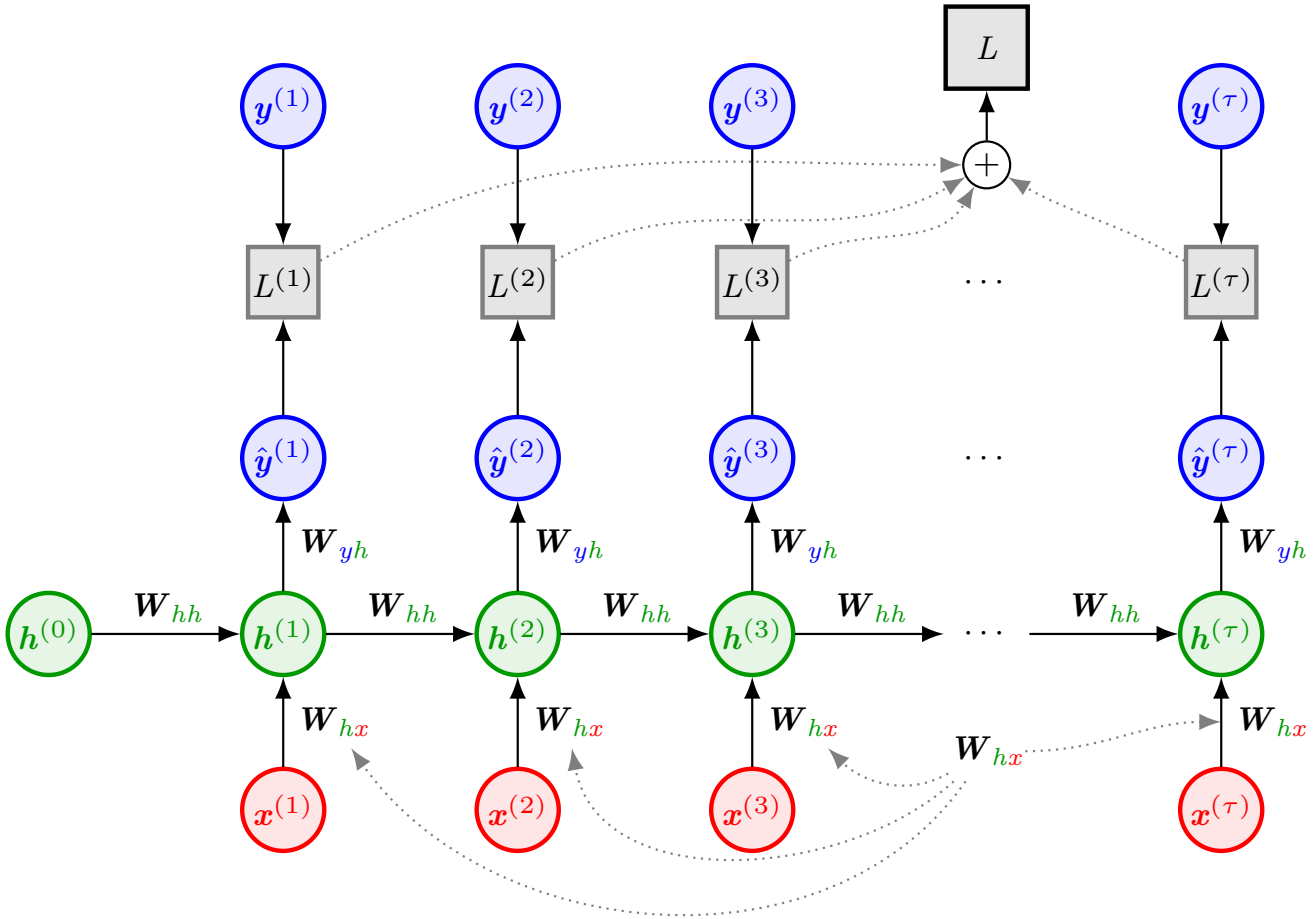
$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{hx}\boldsymbol{x}^{(t)} + \boldsymbol{b}_h)$$

$$\hat{\boldsymbol{y}}^{(t)} = \mathrm{softmax}(\boldsymbol{W}_{yh}\boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$



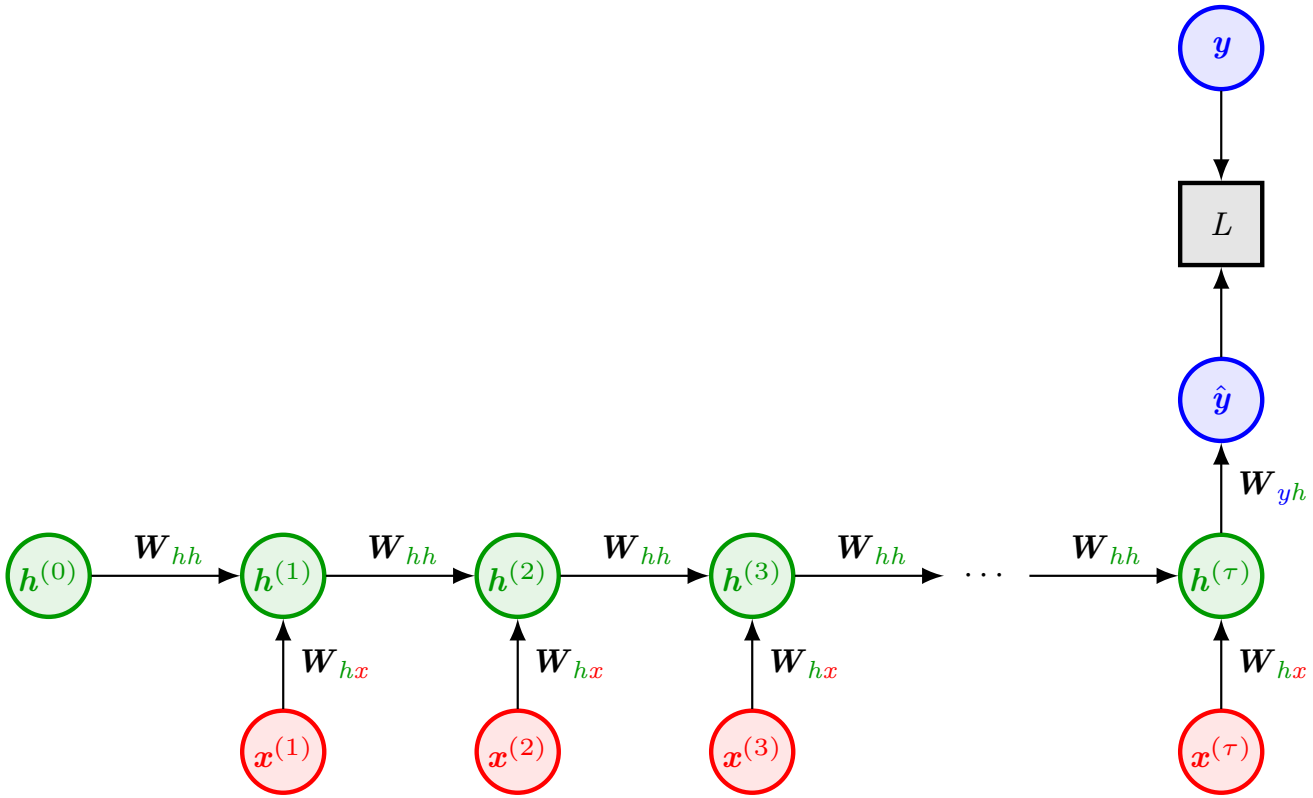
many-to-many

이게 minibatch 1개임!!!



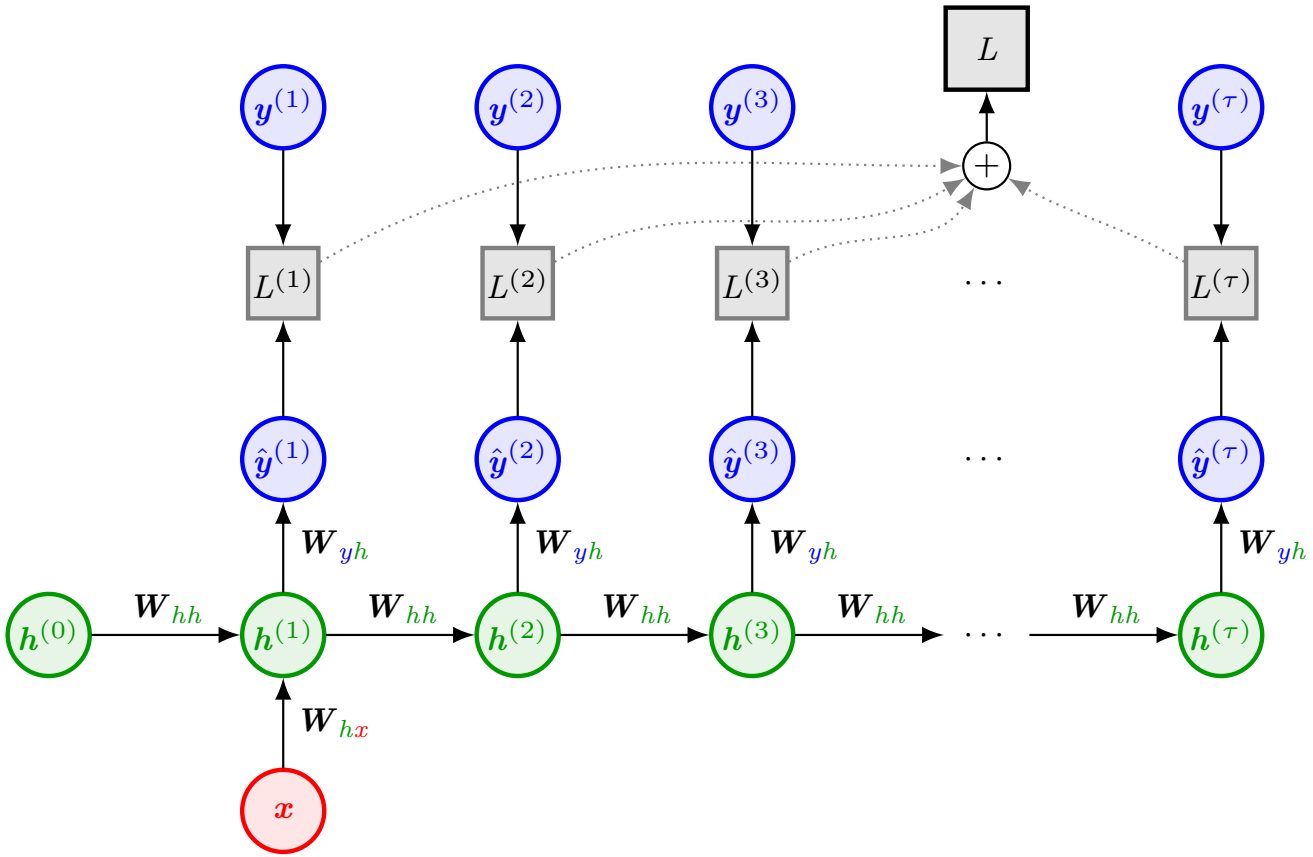
- total loss for a sequence of $\boldsymbol{x}$ (paired with same-length sequence $\boldsymbol{y}$)
  - sum of losses over all time steps: $L = \sum_{t=1}^{\tau} L^{(t)} = \sum_{t=1}^{\tau} L(\hat{\boldsymbol{y}}^{(t)}, \boldsymbol{y}^{(t)})$

many-to-one



- loss for a sequence of $\boldsymbol{x}$ (label: a single vector $\boldsymbol{y}$)
  - $L = L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

one-to-many

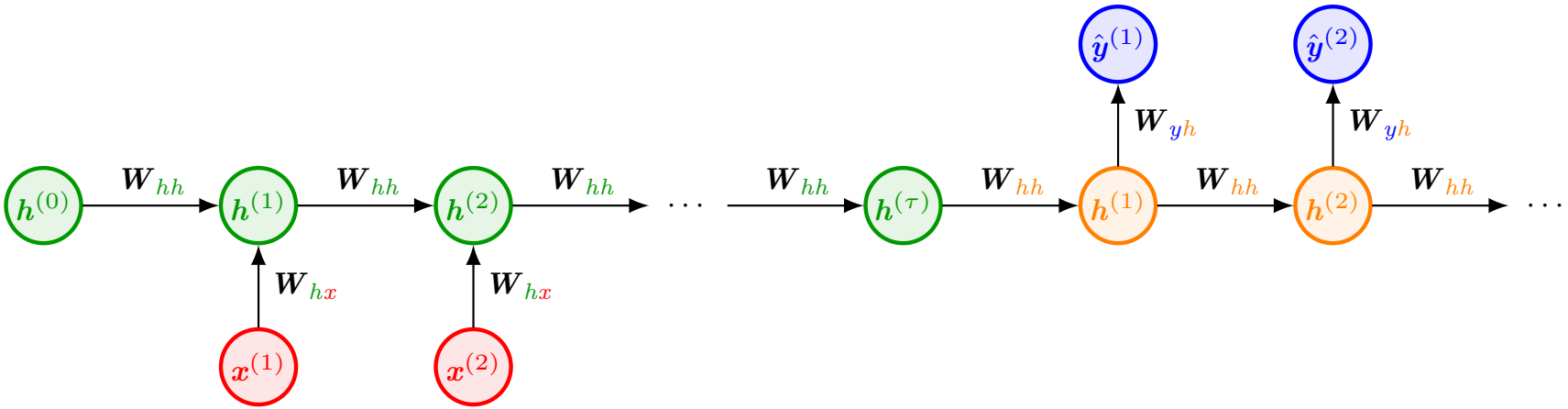


- total loss for a vector $\boldsymbol{x}$ (label: a sequence of $\boldsymbol{y}$)
  - sum of losses over all time steps: $L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{t=1}^{\tau} L^{(t)} = \sum_{t=1}^{\tau} L(\hat{\boldsymbol{y}}^{(t)}, \boldsymbol{y}^{(t)})$

x의 길이와 y의 길이가 independent함

sequence-to-sequence (= many-to-one + one-to-many)

- many-to-one
  - encode input sequence into a single vector
- one-to-many
  - produce output sequence from the single input vector

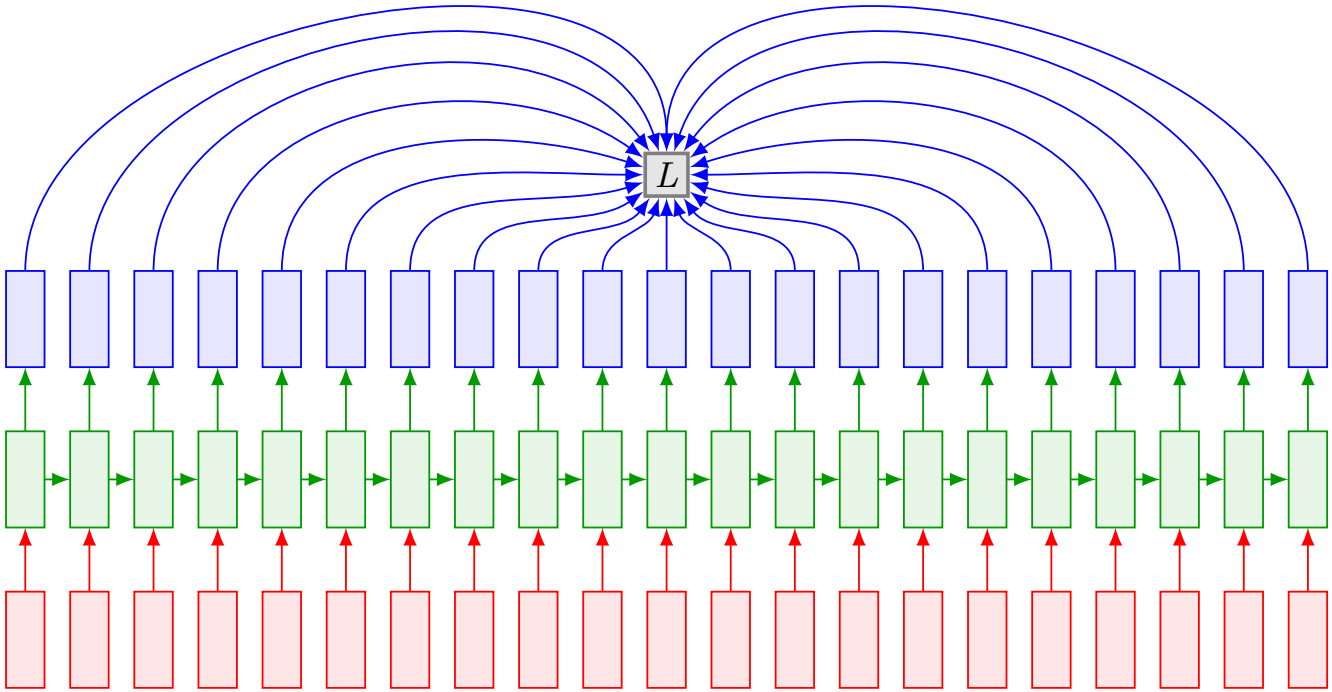


- two RNNs: trained jointly (more on this later)

BPTT: computing gradient through RNN

- use of backprop on unrolled graph: called BPTT (backprop through time)
  - forward prop through entire sequence $\rightarrow$ compute loss
  - backprop through entire sequence $\rightarrow$ compute $\underbrace{\text{gradient}}_{\uparrow}$

may then be used with any general-purpose gradient-based techniques to train an RNN



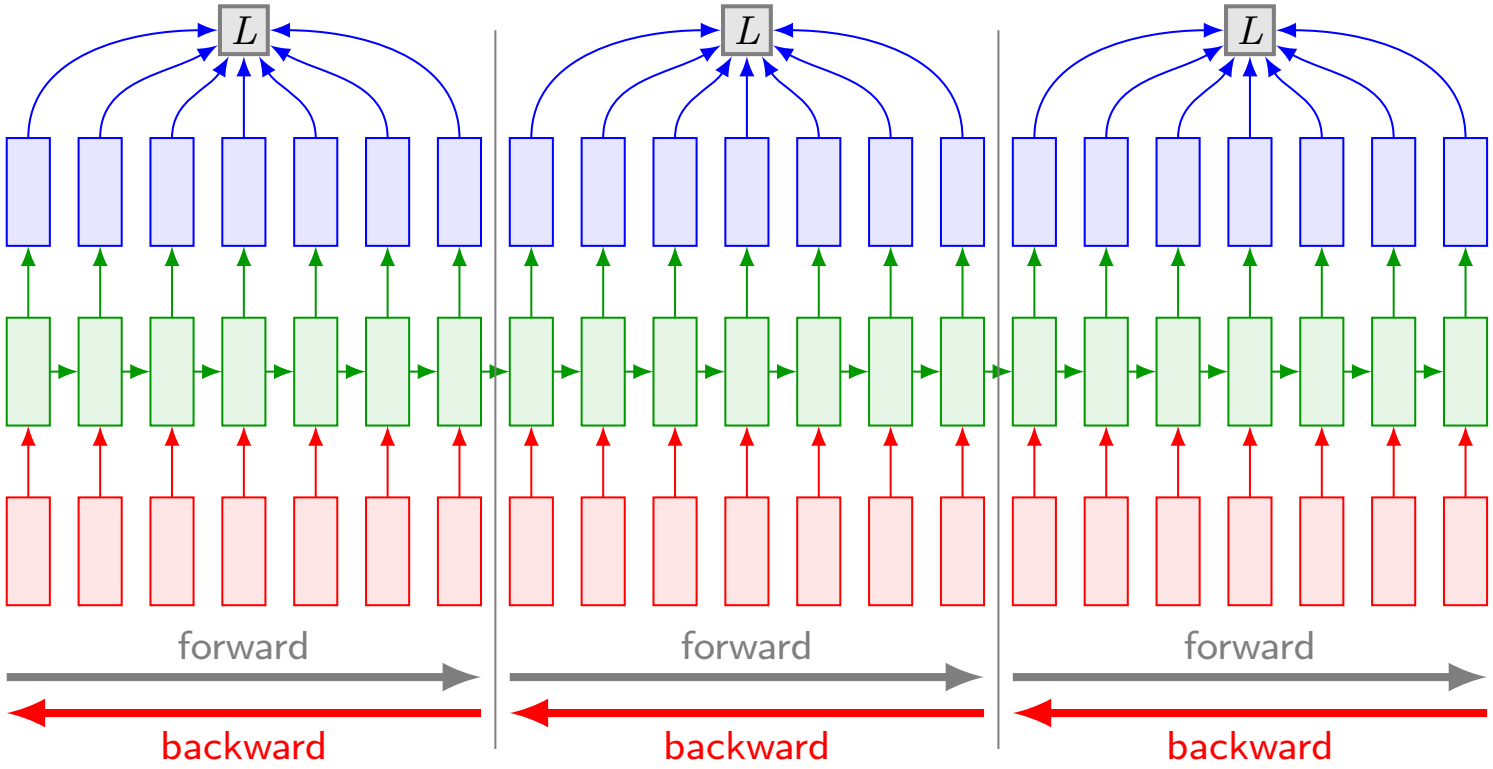
Computational cost

- computing $\nabla_{\boldsymbol{\theta}} L$
  - an expensive operation => forward backword 둘다 길다
  - involves both forward/backward propagation through graph
- runtime: $O(\tau)$ and cannot be reduced by parallelization
  - forward prop graph: inherently sequential(dependency가 있어서 계산이 힘듬)

  i.e. each time step may only be computed after the previous one
- memory: also $O(\tau)$
  - states computed in fwd pass must be stored until reused during bwd pass
- the network with recurrence between hidden units:
  - thus very powerful but also expensive to train

Truncated BPTT

- run forward/backward through chunks of sequence (instead of whole sequence)
  - carry hidden states forward in time forever
  - but only backpropagate for some smaller number of steps
  - in spirit: similar to minibatch sgd

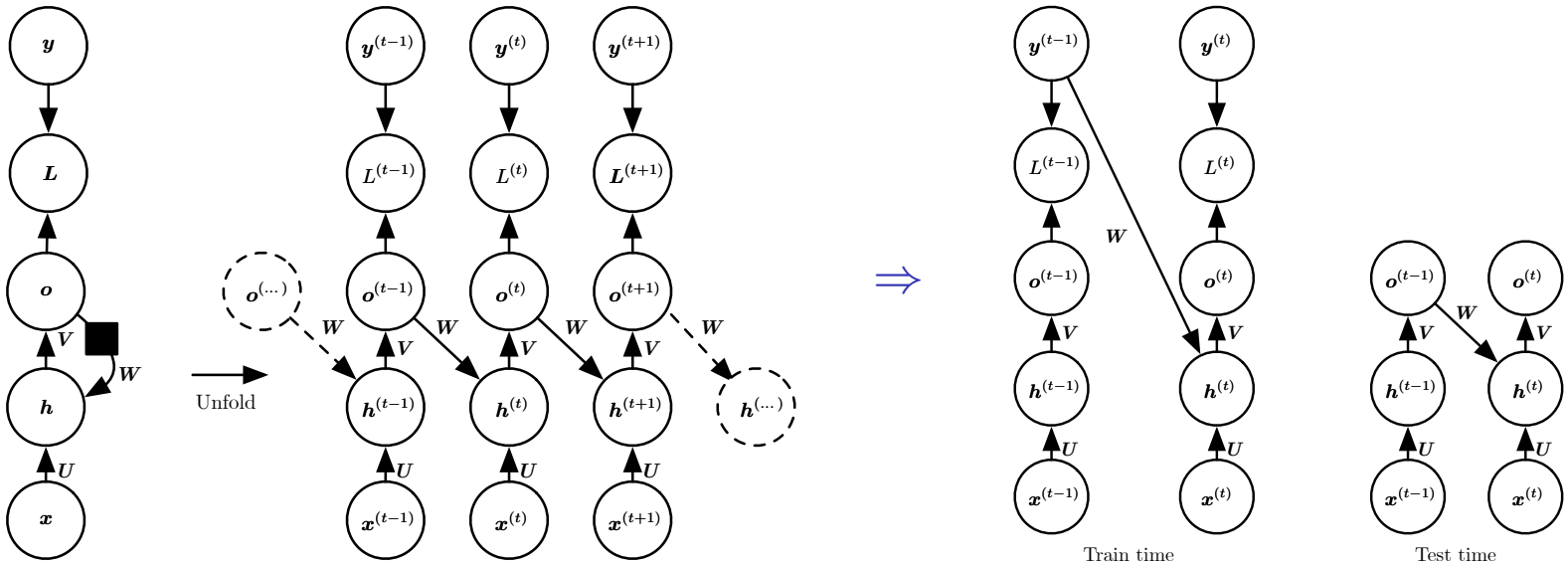


Teacher forcing

hidden to hidden 연결이 없음
=> output을 parallel하게 하여 빠른 training가능

- a training technique for nets with output-hidden recurrence
  - no hidden-to-hidden recurrence ($\rightarrow$ less expressive)
  - each time step can be trained independently in parallel ($\rightarrow$ faster)

- what it does
  - training: ground truth output $\boldsymbol{y}^{(t-1)}$ as input at time t
  - testing: model output $\hat{\boldsymbol{y}}^{(t-1)}$ as input at time t



RNN to train

teacher forcing

Outline

Language modeling

자연어 처리에서 중요한 기본모듈

자연어: 조금 틀려도 큰 문제가 없음

프로그래밍언어: 조금만 틀려도 큰 문제있음

- central to many important natural language processing (NLP)
  - used to generate text in various NLP tasks => text generation

  e.g. speech recognition, machine translation, image captioning, spelling correction, text summarization, ...

- major tasks (← inherently probabilistic)
  - assign probability to sentences in a language

$$P(\text{apple and pair salad}) = 3.2 \times 10^{-13}$$

$$P(\text{apple and pear salad}) = 5.7 \times 10^{-10}$$

  - predict the next word in a sequence given the words that precede it

$$P(\text{pair} \mid \text{apple and}) = 0.00012$$

$$P(\text{pear} \mid \text{apple and}) = 0.03$$

- major types
  - (classical) statistical language model (*e.g.* n-gram)
  - neural language model (NLM) (*e.g.* word embedding)

- key reason for improved performance by NLM
  - ability to generalize

- NLM learns directly from raw text data
  - both representation and probabilistic model

word feature vector — language model + its parameters

- NLM addresses the n-gram data sparsity issue through (n-gram => 몹시 sparse함)
  - parameterizing words as vectors (*i.e.* word embedding)
  - using them as inputs to a neural net

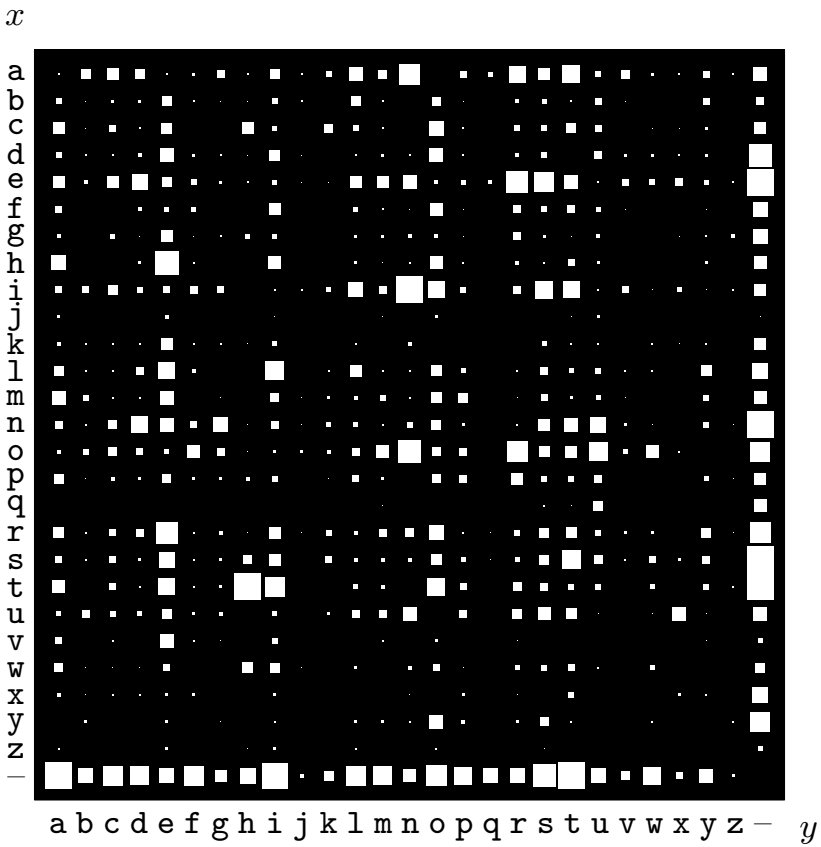
Character-level n-gram examples

- corpus: *The frequently Asked Questions Manual for Linux*

- monogram ->1character
  - $p(x)$

i	a_i	p_i
1	a	0.0575
2	b	0.0128
3	c	0.0263
4	d	0.0285
5	e	0.0913
6	f	0.0173
7	g	0.0133
8	h	0.0313
9	i	0.0599
10	j	0.0006
11	k	0.0084
12	l	0.0335
13	m	0.0235
14	n	0.0596
15	o	0.0689
16	p	0.0192
17	q	0.0008
18	r	0.0508
19	s	0.0567
20	t	0.0706
21	u	0.0334
22	v	0.0069
23	w	0.0119
24	x	0.0073
25	y	0.0164
26	z	0.0007
27	–	0.1928

- bigram ->2character
  - $p(xy)$

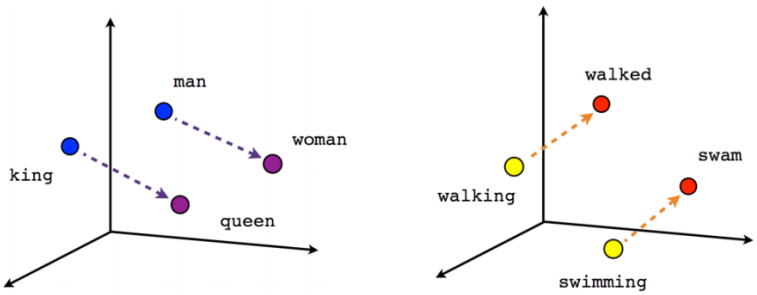


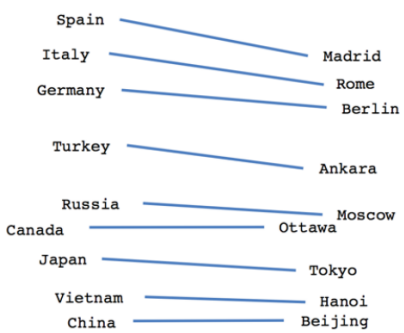
(source: MacKay)

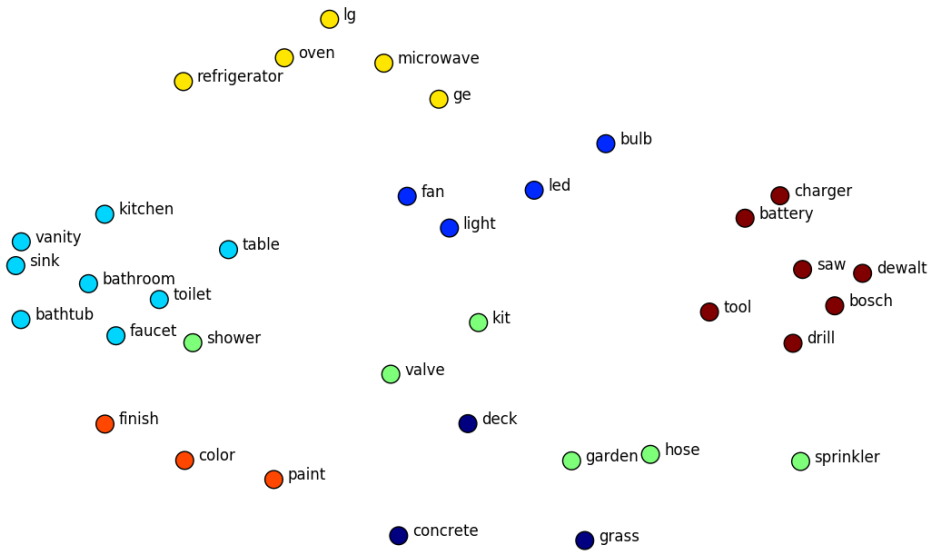
Word embedding

pre-train된 word embedding을 재사용 가능

- learned representation of words based on usage
  - uses a real-valued (ont hot이 아니라) vector to represent each word
  - allows words with a similar meaning to have a similar representation







RNN-based language modeling

- training data: text **corpus** (a large and structured set of texts)

step 1: tokenization

- input sentence $\mapsto$ individual tokens in your vocabulary

  e.g. `The pen is mightier than the sword.`

`The`	`pen`	`is`	`mightier`	`than`	`the`	`sword`	`.`	`<EOS>`
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
$\boldsymbol{y}^{(1)}$	$\boldsymbol{y}^{(2)}$	$\boldsymbol{y}^{(3)}$	$\boldsymbol{y}^{(4)}$	$\boldsymbol{y}^{(5)}$	$\boldsymbol{y}^{(6)}$	$\boldsymbol{y}^{(7)}$	$\boldsymbol{y}^{(8)}$	$\boldsymbol{y}^{(9)}$

- special cases
  - start of sentence: `<SOS>`
  - end of sentence: `<EOS>`
  - unknown word: `<UNK>` => 우리의 voca에 없는 경우

step 2: supervised training

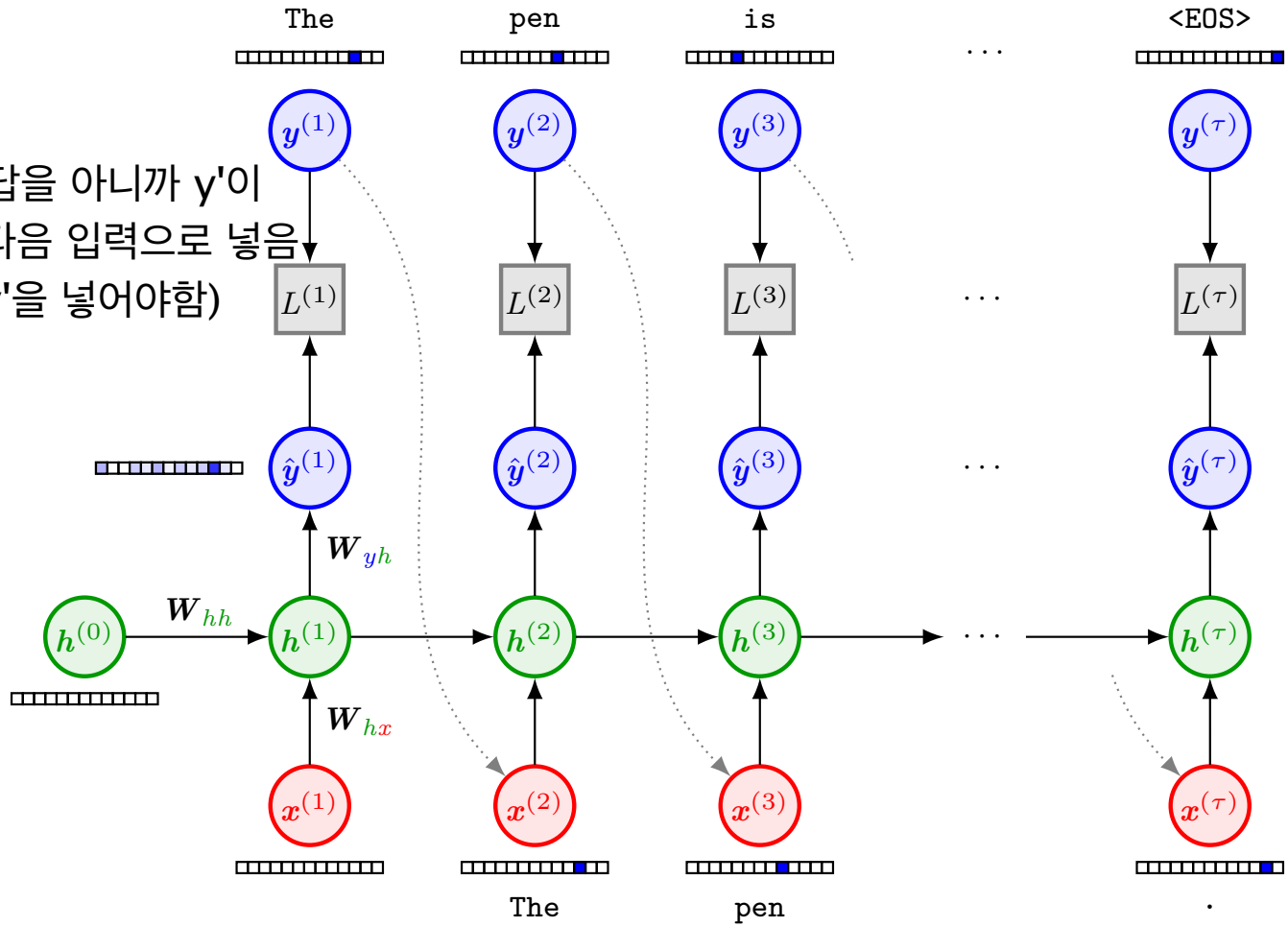
- use RNN to model the probability of different token sequences
- architecture
  - real output $\boldsymbol{y}^{(t)}$: vocabulary-size one-hot vector => 너무 심각하게 sparse함
  - model output $\hat{\boldsymbol{y}}^{(t)}$: vocabulary-size softmax vector
  - if we use a vanilla RNN

$$\begin{aligned}
\boldsymbol{x}^{(t)} &= \boldsymbol{y}^{(t-1)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{hx}\boldsymbol{x}^{(t)} + \boldsymbol{b}_h) \\
&= \tanh(\boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{hx}\boldsymbol{y}^{(t-1)} + \boldsymbol{b}_h) \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{W}_{yh}\boldsymbol{h}^{(t)} + \boldsymbol{b}_y)
\end{aligned}$$

  - initialization: $\boldsymbol{x}^{(1)} = \mathbf{0}$, $\boldsymbol{h}^{(0)} = \mathbf{0}$
  - loss: *e.g.* cross-entropy loss (see next page)

- example: `The pen is mightier than the sword.<EOS>`

train time에는 답을 아니까 y'이 아니라 y를 직접 다음 입력으로 넣음 (test time에는 y'을 넣어야함)



$$L = \sum_{t=1}^{\tau} L^{(t)} = \sum_{t=1}^{\tau} L(\hat{\boldsymbol{y}}^{(t)}, \boldsymbol{y}^{(t)}) \qquad \text{where} \qquad L(\hat{\boldsymbol{y}}^{(t)}, \boldsymbol{y}^{(t)}) = \underbrace{-\sum_{i}^{|\text{vocab}|} y_i^{(t)} \log \hat{y}_i^{(t)}}_{\text{cross-entropy loss}}$$

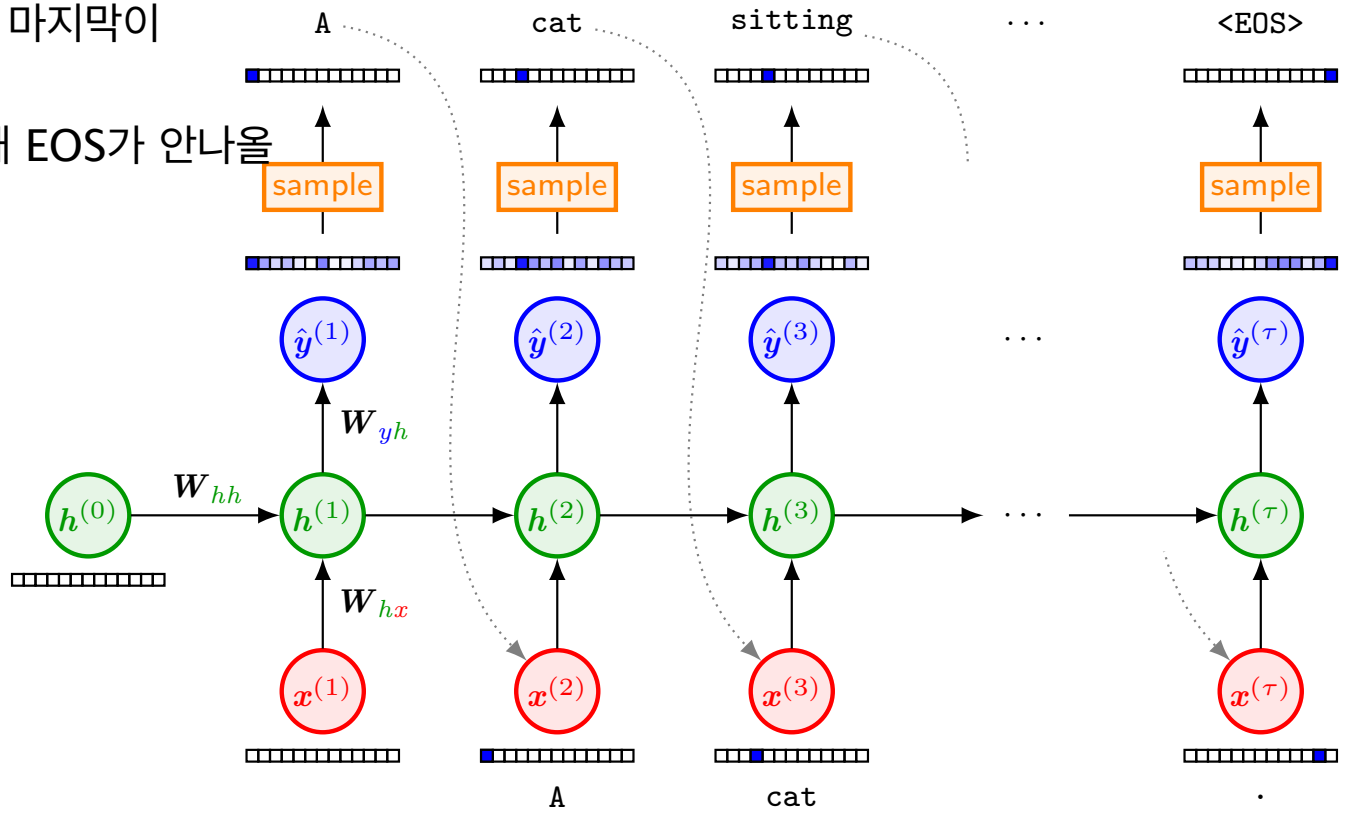
Sampling novel sentences

sampling이 어렵다 => 좋은 sample을 뽑기 어렵다는 의미

- use your trained model ($\rightarrow$ you can get a sense of what it has learned)
  - training: $\boldsymbol{x}^{(t)} = \boldsymbol{y}^{(t-1)}$
  - sampling: $\boldsymbol{x}^{(t)} \sim \hat{\boldsymbol{y}}^{(t-1)}$

  } same initialization: $\boldsymbol{x}^{(1)} = \mathbf{0}, \boldsymbol{h}^{(0)} = \mathbf{0}$

train시에 대부분 마지막이 EOS이므로 generation할 때 EOS가 안나올 확률은 매우 낮음



- one additional complication: =>문장을 어디서 끊을지가 어려운 issue임
  - must determine the sequence length τ
  - this can be achieved in various ways

1. a special input symbol (end marker like <EOS>)
2. extra output unit (predicts whether to continue or end)
3. extra output unit (predicts length τ directly)

approach #1: end marker

- add a special symbol ("end of sequence")
  - when that symbol is generated, the sampling process stops
- applicable when: the output is a symbol taken from a vocabulary(== catogorized)
- in training set
  - insert this symbol as an extra member of the sequence
    (immediately after $\boldsymbol{x}^{(\tau)}$ in each training example)

approach #2

- introduce an extra output unit (Bernoulli)
  - decides (at each time step) either continue generation or halt generation
  - usually a sigmoid unit trained with cross-entropy loss (end or continue)

- more general than approach #1[4] => #1방법은 symbol의 sequence에서만 사용이 용이함
  - may be applied to any RNN (*e.g.* that emits a sequence of real numbers)

approach #3

- add an extra output that predicts integer τ
  - based on the decomposition

$$P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}) = P(\tau) P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)} \mid \tau) \tag{34}$$

  $\Rightarrow$ model can sample a value of τ and then sample τ steps worth of data

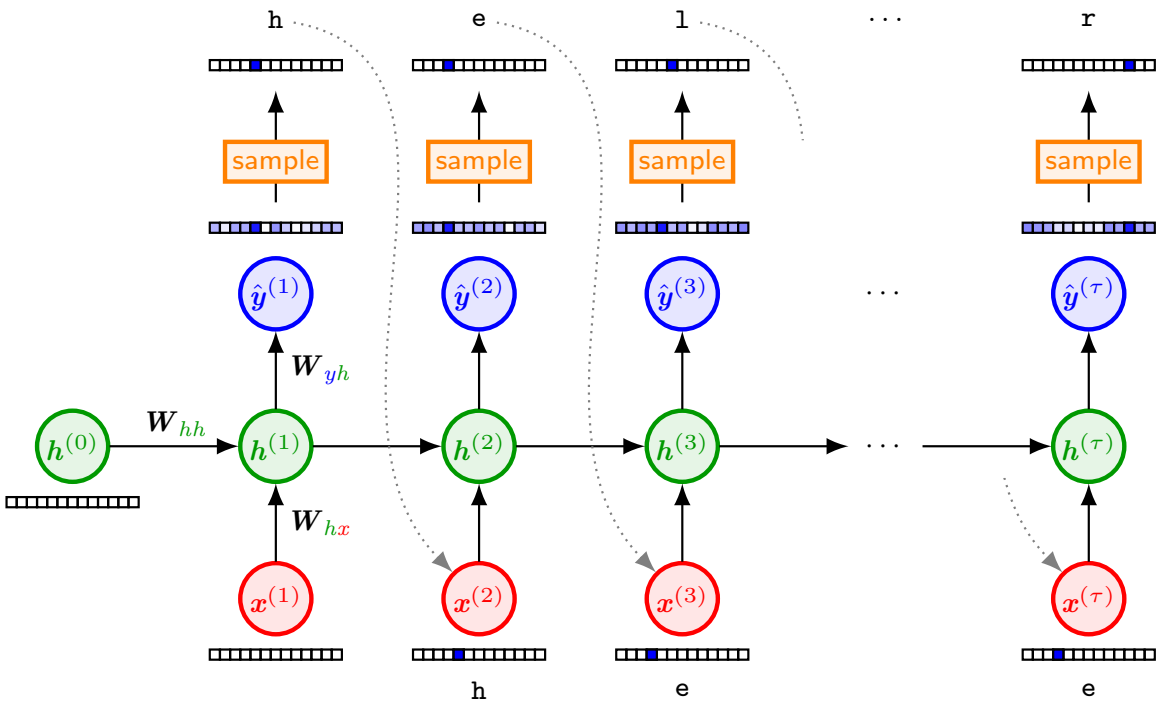
[4] applicable to only RNNs that output a sequence of symbols

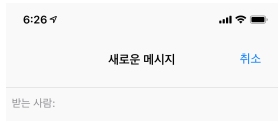
Word-level vs character-level

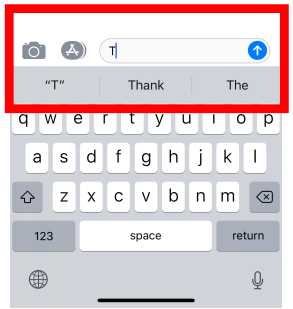
- so far: word-level language modeling
  - vocabulary = [a, aaron, …, zzz, <UNK>, <EOS>]

- also possible: character-level language modeling
  - vocabulary = [a, b, c, …, z, ␣, ., ,, ;, …, 0, 1,…, 9, A, …, Z]

e.g. sampling 'helicopter'







Example code

- minimal char-level language model (only 112 lines of code) ▶ Link

Instantly share code, notes, and snippets.

karpathy / min-char-rnn.py

Last active 22 hours ago

Minimal character-level language model with a Vanilla Recurrent Neural Network, in Python/numpy

min-char-rnn.py

```
"""
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
BSD License
"""
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }

# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward pass
  for t in xrange(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
  for t in reversed(xrange(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1 # backprop into y. see http://cs231n.github.io/neural-networks-case-study/#gra
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]

def sample(h, seed_ix, n):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in xrange(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes

n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    p = 0 # go from start of data
  inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
  targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

  # sample from the model now and then
  if n % 100 == 0:
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print '----\n %s \n----' % (txt, )

  # forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```

- generating Shakespeare-style text

William Shakespeare (1564–1616)

(source: wikipedia, cs231n)

```
tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase
rrranbyne 'nhthnee e plia tklrgd t o idoe ns,smtt h ne etie h,
hregtrs nigtike,aoaenns lng
```

train more

```
"Tmont thithey" fomesscerliund
Keushey. Thom here sheulke,
anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

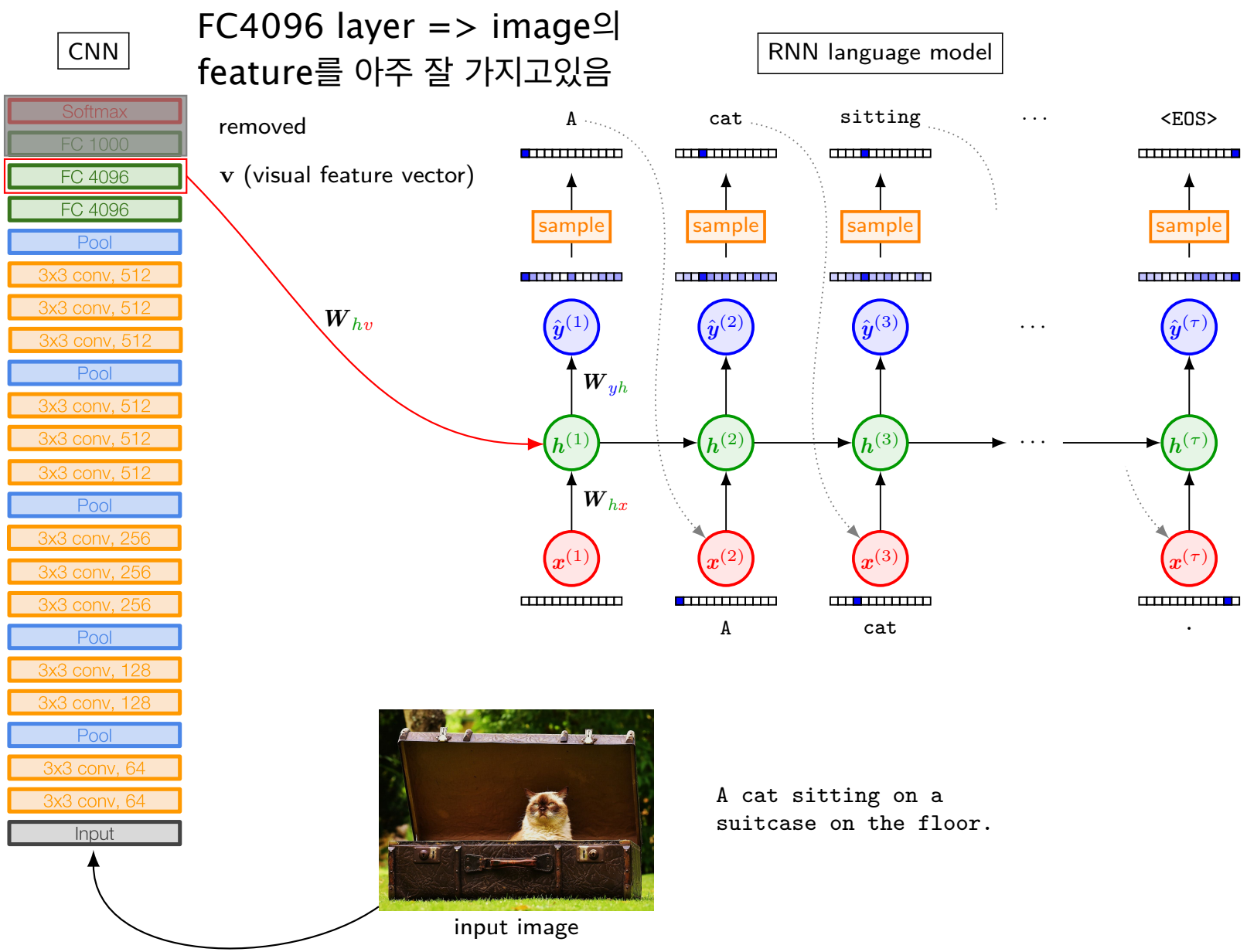
train more

```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say
falling misfort how, and Gogition is so overelical and ofter.
```

train more

```
"Why do what that day," replied Natasha, and wishing to himself
the fact the princess. Princess Mary was easier, fed in had
oftened him. Pierre asking his soul came to the packs and drove up
his father-in-law women.
```

Application of language modeling: image captioning



Known heuristics for neural language modeling

1. size matters
   - best models = largest models (esp. # of memory units)
2. regularization matters

   e.g. dropout on input connections
3. CNN vs embedding
   - character-level CNN models can be used on front-end

     ↑

     instead of word embedding
4. ensembles matter
   - as usual

(source: Jozefowicz et al., 2016)

Outline

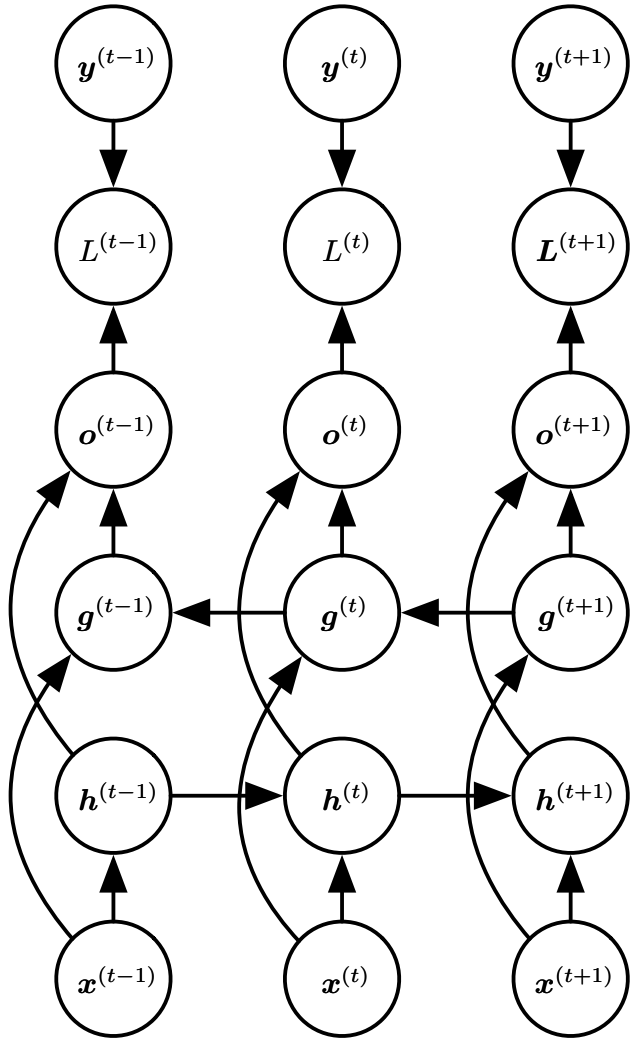
Motivation

real-time이 아닌이상 과거의 것 뿐만아니라 미래의 것을 봐도 됨

- all RNNs considered up to now: "causal" structure
  i.e. state at time t only captures information from
    - ▷ past $(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t-1)})$ and present input $\boldsymbol{x}^{(t)}$
- but often output $\boldsymbol{y}^{(t)}$ may depend on the *whole input sequence*
  e.g. speech recognition, handwriting recognition, seq-to-seq learning
- speech recognition: correct interpretation of the current sound as a phoneme
  - ▶ may depend on next few phonemes (because of co-articulation)
- bidirectional RNN: invented to address that need
  - ▶ has been extremely successful

Bidirectional RNN

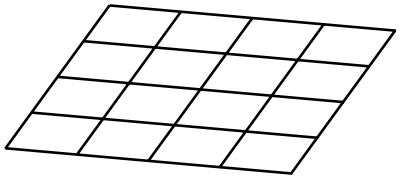
- learns to map
  - input seq $\boldsymbol{x}$ to target seq $\boldsymbol{y}$ with loss $L^{(t)}$ at each step t
- $\boldsymbol{h}$ recurrence propagates info forward
  - forward in time ($\longrightarrow$)
- $\boldsymbol{g}$ recurrence propagates info backward
  - backward in time ($\longleftarrow$)
- output units $\boldsymbol{o}^{(t)}$ can benefit from
  - a summary of the past in $\boldsymbol{h}^{(t)}$
  - a summary of the future in $\boldsymbol{g}^{(t)}$

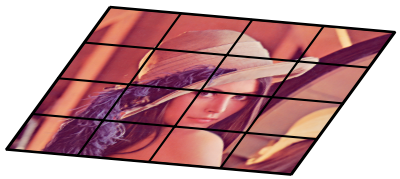


Extension to 2D (*e.g.* images)

- use *four* RNNs: each one goes up, down, left, right
- output $O_{i,j}$ at each point (i, j) can capture
  - mostly local information
  - but also long-range information (if RNN learns to carry that info)
- more expensive than CNN
  - but allow for long-range lateral interactions

between features in the same feature map





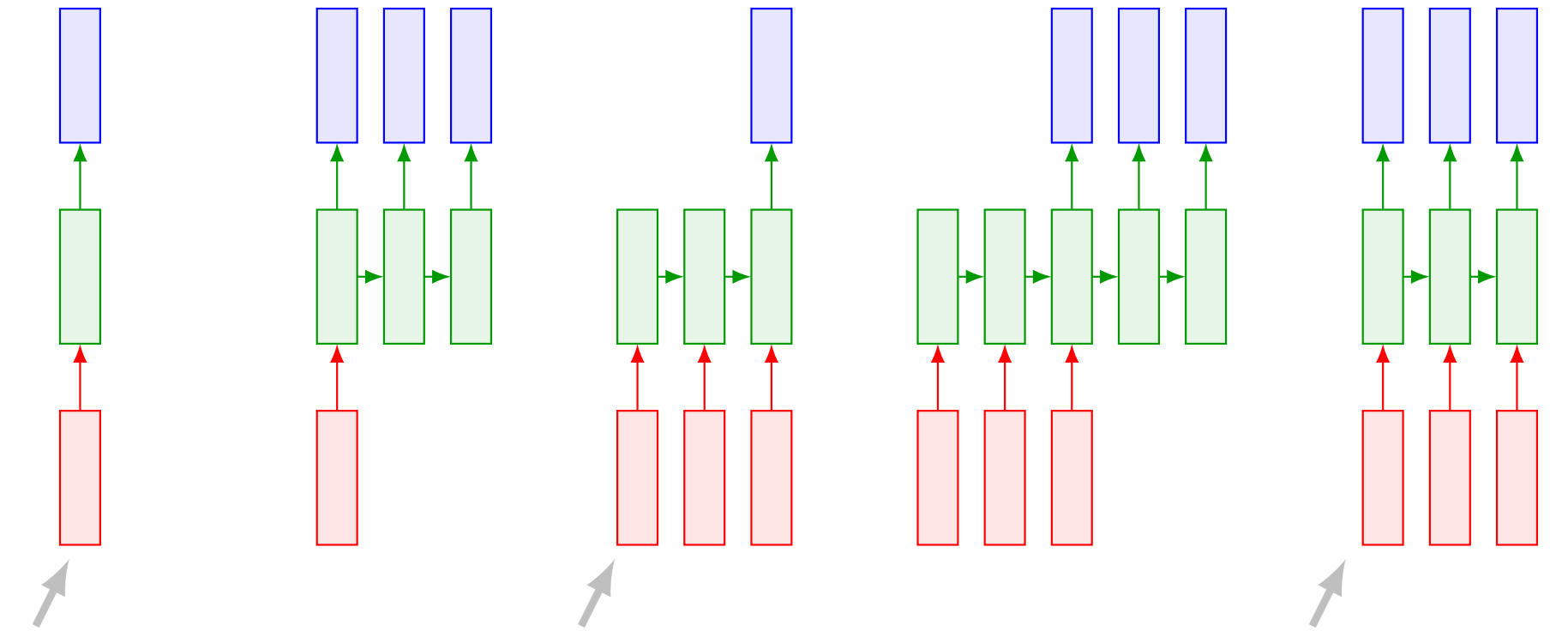
(source: Visin et al., 2015)

Outline

Recall: RNN modeling capability

e.g. **image captioning**
image → sentence

e.g. **machine translation**
seq of words → seq of words



vanilla neural net

e.g. **sentiment classification**
sentence → sentiment

e.g. **video frame classification**
seq of frame → seq of class

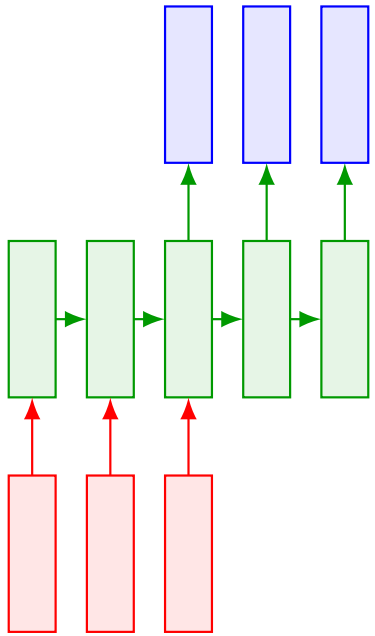
(source: cs231n)

Mapping sequence to sequence (of different lengths)

- RNN can be trained to map an input sequence to an output sequence
  - which is not necessarily of the same length

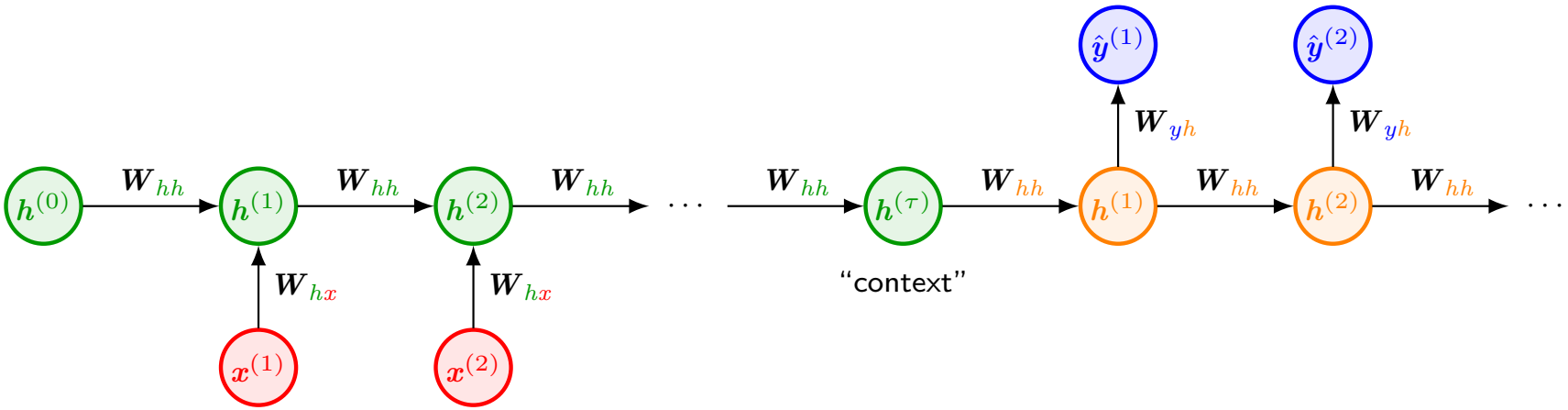
- this comes up in many applications

  e.g. speech recognition, machine translation, question answering



sequence-to-sequence (= many-to-one + one-to-many)

- many-to-one (encoder)
  - encode input sequence into a single vector
- one-to-many (decoder)
  - produce output sequence from the single input vector



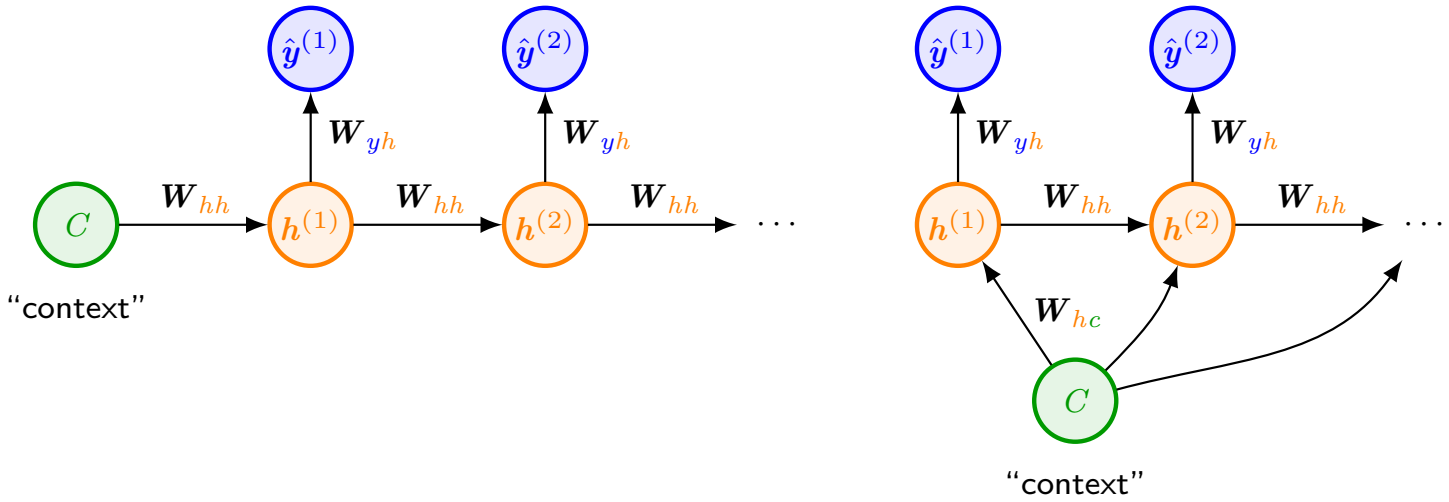
- context C: a vector (or a sequence of vectors in attention model)
  - summarizes the input sequence $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n_x)})$
  - output from the encoder/input to the decoder

context => 입력에 대한 lossy summary

Receiving inputs in decoder

- a one-to-many (vector-to-sequence) RNN
  - can receive input in two ways

1. input can be provided as initial state of RNN, or
2. input can be connected to hidden units at each time step

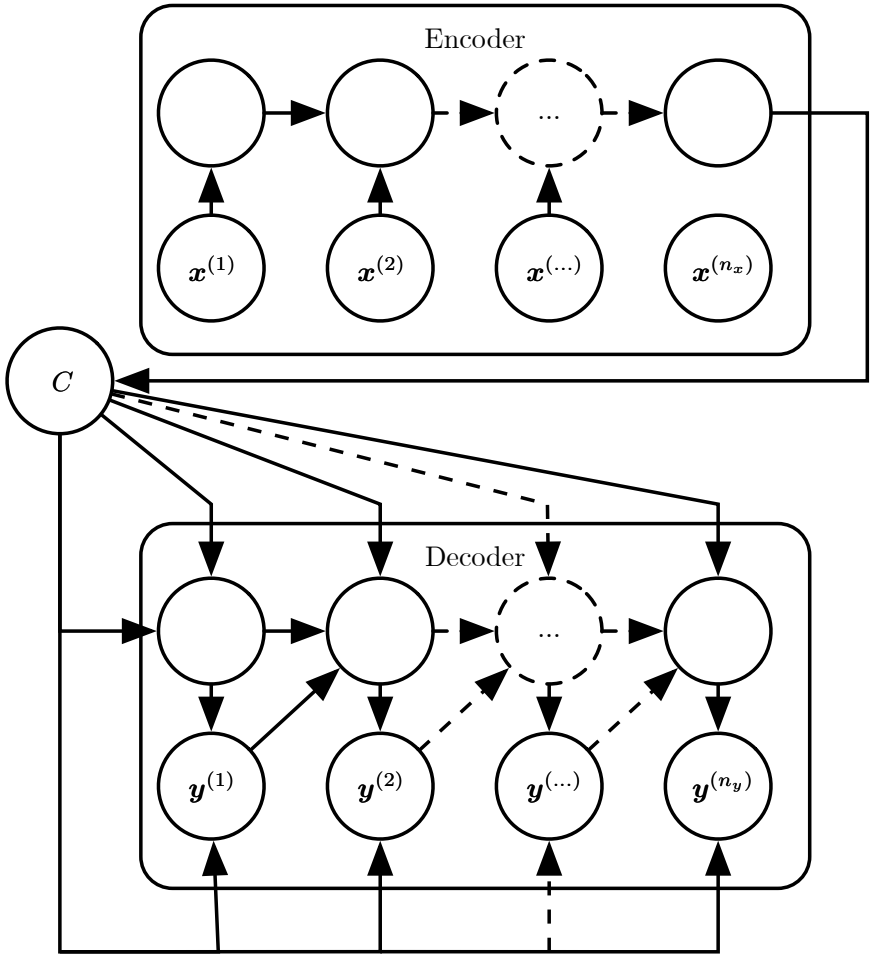


Encoder-decoder/sequence-to-sequence architecture

- goal: map a variable-length sequence to another variable-length sequence
  - first application: machine translation

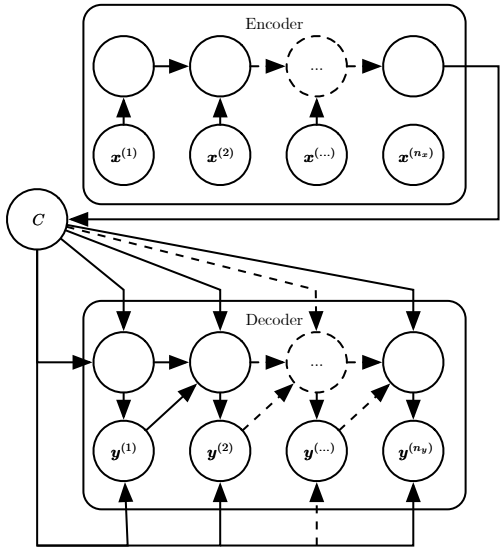
- two names
  - encoder–decoder architecture (Cho *et al.*, 2014)
  - sequence-to-sequence architecture (Sutskever *et al.*, 2014)

- corresponds to
  - many-to-many RNN



(1) encoder/reader/input RNN:

- processes input sequence $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n_x)})$
- emits context C (generally a fixed-size vector)
  - usually a simple function of final hidden state $\boldsymbol{h}_{n_x}$
  - represents a semantic summary of input seq



(2) decoder/writer/output RNN:

- takes context C as input
- generates output sequence $\boldsymbol{Y} = (\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n_y)})$

- encoder/decoder RNNs: trained jointly
  - to maximize the average of $\log P(\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n_y)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n_x)})$

over all the pairs of $\boldsymbol{X}$ and $\boldsymbol{Y}$ sequences in training set

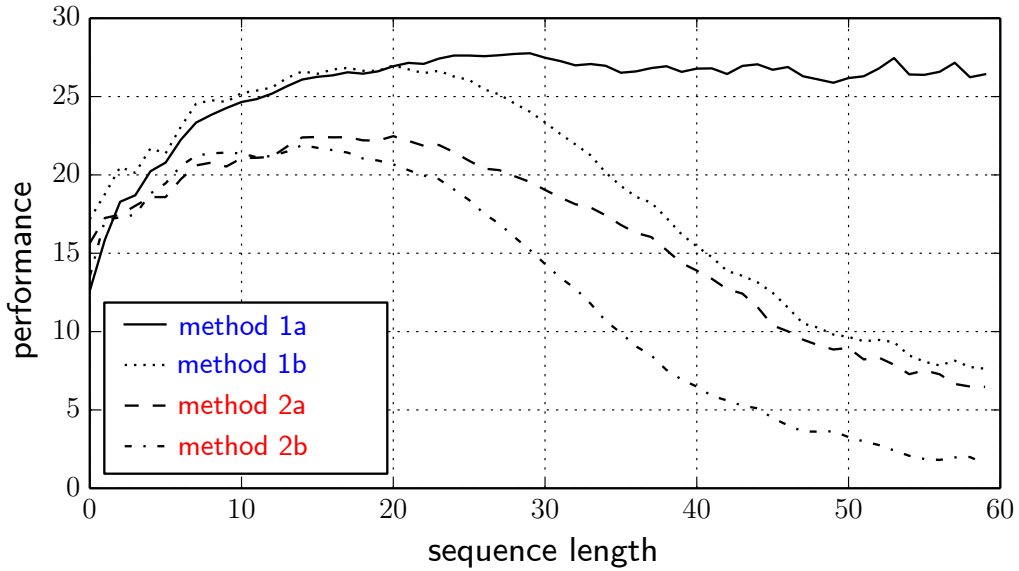
- innovation of this architecture: not necessarily $n_x = n_y$

Attention mechanism

- one clear limitation

  1개의 single context vector로는 long sequence에 대하여 표현력 부족

  - dimension of context C may be too small to summarize a long sequence



- methods 1 and 2
  - with and without attention, respectively
- a and b
  - trained with max 50 and 30 words, respectively

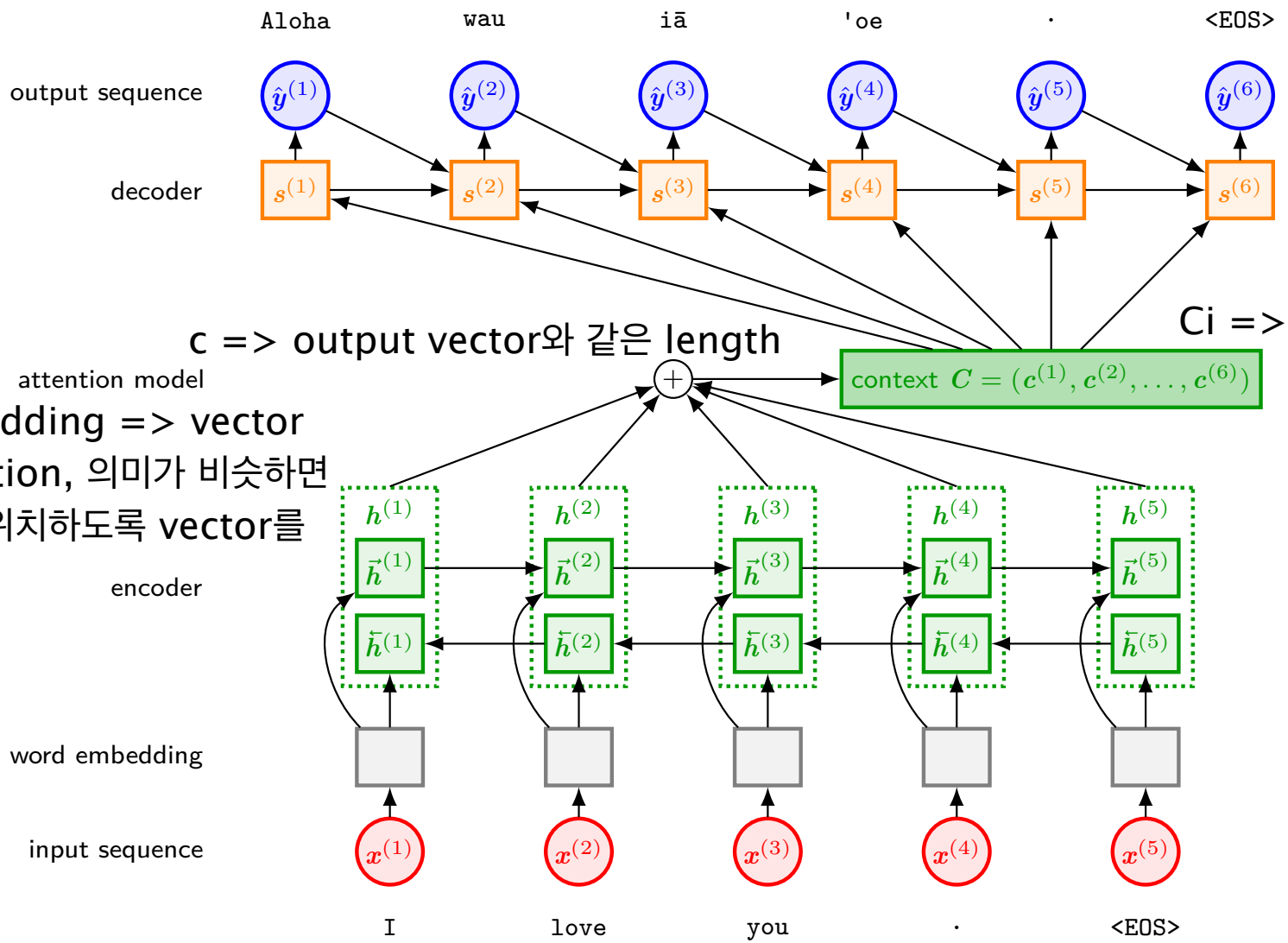
(source: Bahdanau et al., 2015)

- attention mechanism
  - makes $\boldsymbol{C}$ a variable-length sequence (not a fixed-size vector)
  - learns to associate

    element $\boldsymbol{c}^{(t)}$ of sequence $\boldsymbol{C}$ $\leftrightarrow$ element $\boldsymbol{y}^{(t)}$ of output sequence

- example

attention => 표현력이 부족한 것을 보강하기 위해 나온 것



c => output vector와 같은 length

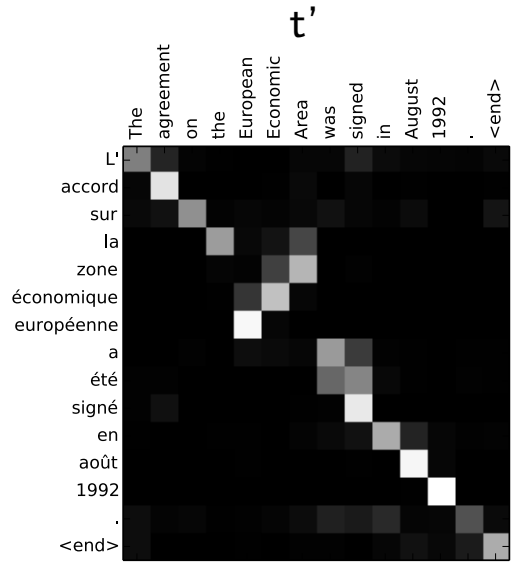
Ci => vector

word embedding => vector representation, 의미가 비슷하면 비슷한 곳에 위치하도록 vector를 만들어줌

- more details: Link
  - let $\underbrace{t \in [1, n_y]}_{\text{1~6 output index}}$ and $\underbrace{t' \in [1, n_x]}_{\text{input index 1~5}}$
  - context sequence $\boldsymbol{C} = (\boldsymbol{c}^{(1)}, \boldsymbol{c}^{(2)}, \ldots, \boldsymbol{c}^{(n_y)})$

$$\boldsymbol{c}^{(t)} = \sum_{t'=1}^{n_x} \alpha^{(t,t')} \boldsymbol{h}^{(t')}$$

가중치 * hidden vector의 weighted sum



$\alpha^{(t,t')}$

  - $\alpha^{(t,t')}$: amount of attention $\boldsymbol{y}^{(t)}$ should pay to $\boldsymbol{h}^{(t')}$

$$\alpha^{(t,t')} = \frac{\exp(e^{(t,t')})}{\sum_{t'=1}^{n_x} \exp(e^{(t,t')})}$$

  - $e^{(t,t')}$: learned by a neural net

$$e^{(t,t')} = f(\boldsymbol{h}^{(t')}, \boldsymbol{s}^{(t-1)})$$

English → French
- t': English
- t: French

(source: Bahdanau, 2015)

t번째 output을 위해 t'번째 input에 쏟는 attention

Outline

Computation in most RNNs

- can be decomposed into
  - three blocks of parameters and associated transformations:

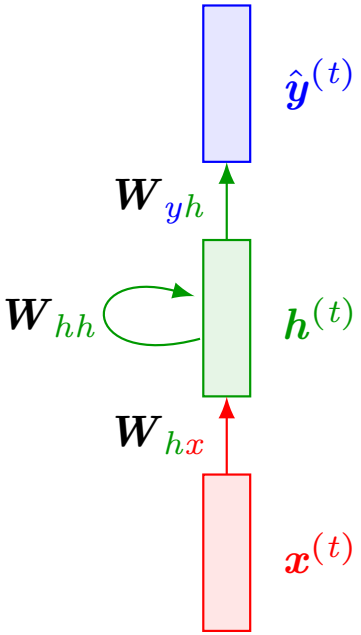
1. from input to hidden state ($i \rightarrow h$)
2. from previous hidden state to next hidden state ($h \rightarrow h$)
3. from hidden state to output ($h \rightarrow o$)

- consider a vanilla RNN
- each of these three blocks
  - associated with a single weight matrix

$\Rightarrow$ corresponds to a shallow transformation

↑

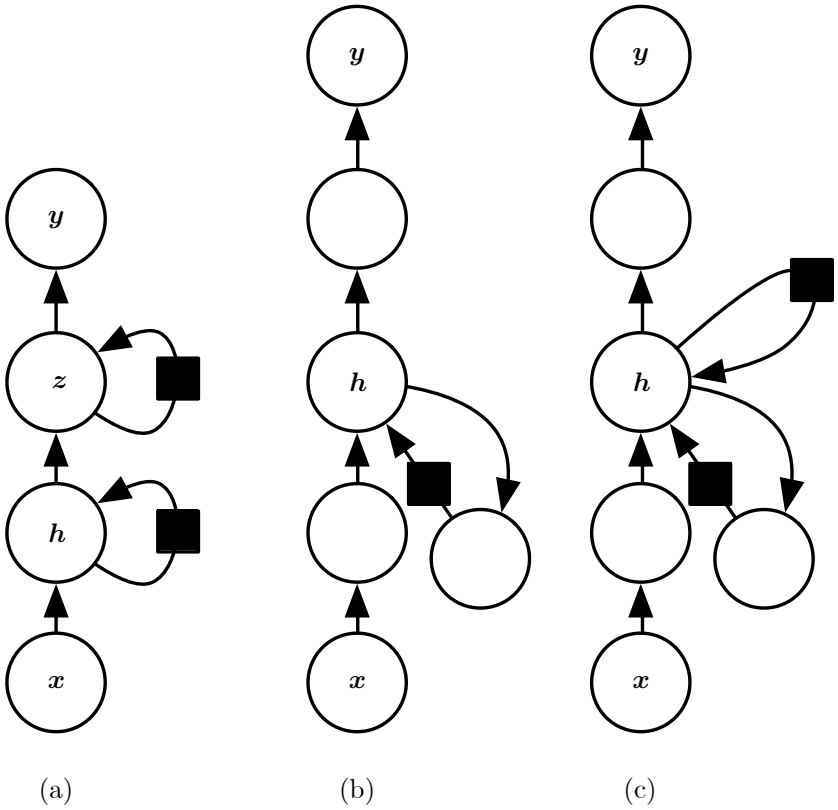
a learned affine transformation
followed by a fixed nonlinearity
= a single layer in MLP



Deep RNN

vertically deep rnn(원래 rnn은 time wise 무한대이므로 horizontally deep함)

- would it be advantageous to introduce depth in each of these operations? yes



(a) (b) (c)

- three examples:

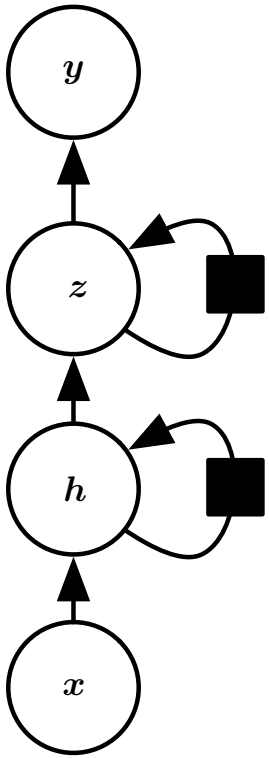
(a) hidden recurrent state can be broken down
  - into groups organized hierarchically

(b) deeper computation (*e.g.* an MLP) can be introduced
  - in input-to-hidden, hidden-to-hidden and hidden-to-output parts
  - this may lengthen shortest path linking different time steps

(c) path-lengthening effect can be mitigated
  - by introducing skip connections

Example (a)

- decompose the hidden state into multiple layers
- lower layer $\rightarrow$ higher layer in hierarchy
  - raw data $\rightarrow$ higher-level representation
- in practice
  - hidden layer depth: not much greater than 2–3



(a)

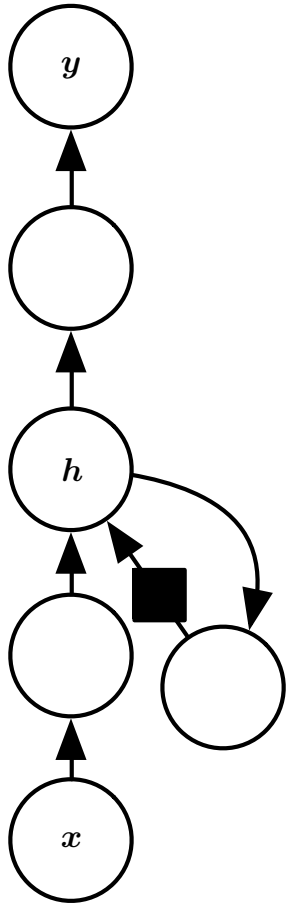
Example (b)

- we can go a step further from example (a)
  - a separate (deep) MLP for each of the three blocks

- but adding depth
  - may hurt learning by making optimization difficult
  - makes the shortest path become longer

↑
from a variable in step t
to a variable in step $t+1$

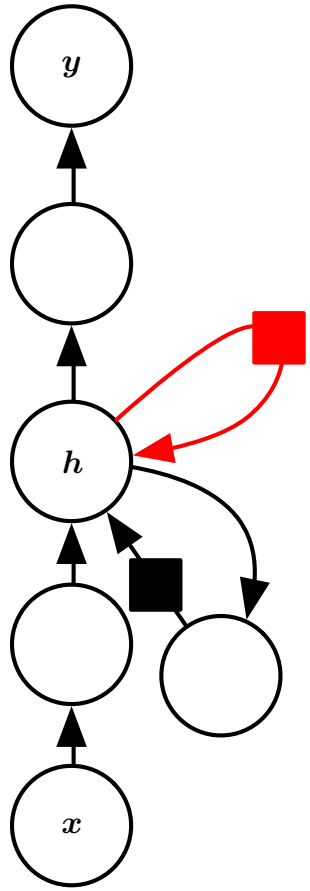
길이지기 때문에 back propagation때 gradient vanishing/exploding 위험성 -> (c)에서 skip connection을 이용해서 어느정도 해결시도



(b)

Example (c)

- path-lengthening can be mitigated by
  - introducing skip connections in hidden-to-hidden path

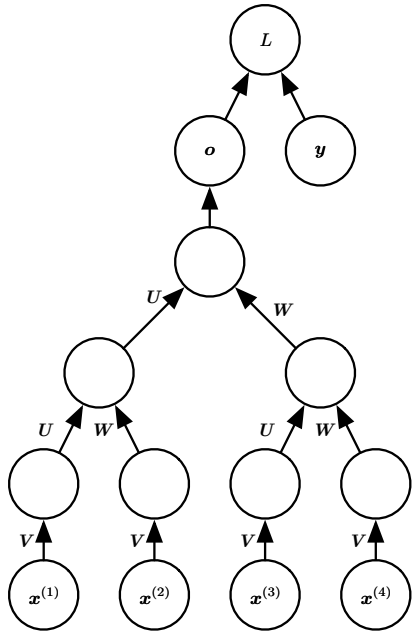


(c)

Outline

Recursive neural networks[5]

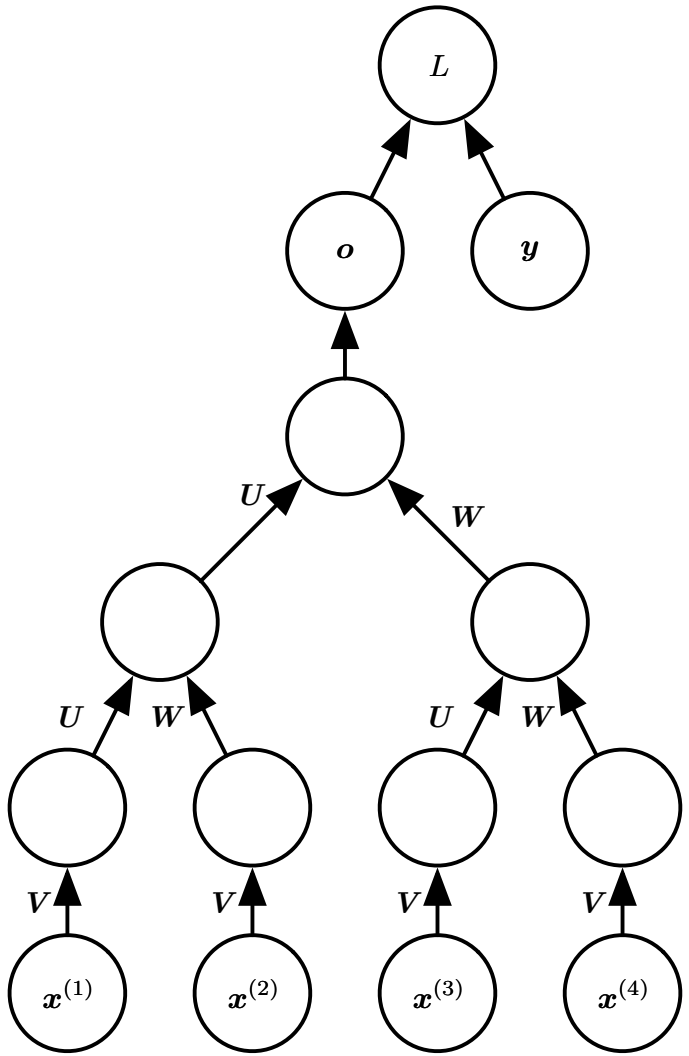
- represent yet another generalization of recurrent nets
  - has potential use for learning to reason
- comparison: computational graph
  - recurrent neural nets: chain-like structure
  - recursive neural nets: structured as a deep ___
- have been successfully applied to
  - processing *data structures* as input to neural nets
  - natural language processing, computer vision



[5] do not abbreviate "recursive neural network" as "RNN" to avoid confusion with "recurrent neural network"

Example

- a variable-size sequence $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}$
  - can be mapped to a fixed-size representation (output $\boldsymbol{o}$)
  - with fixed parameters (weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$)
- this figure: supervised learning case
  - target $\boldsymbol{y}$ is provided
    ↑
    associated with the whole sequence



Benefits

- one clear advantage of recursive nets over RNNs:
  - for a sequence of the same length τ,
  - the $\underbrace{\quad}_{\uparrow}$ can be drastically reduced from τ to $O(\log \tau)$

    (measured as the number of compositions of nonlinear operations)

  $\Rightarrow$ which might help deal with long-term dependencies

- an open question: how to best structure the tree
- one option: have a tree structure independent of data

  e.g. balanced binary tree

- in some application domains
  - external methods can suggest the appropriate tree structure

  e.g. parse tree provided by natural language parser

Outline

Summary

- sequential data: ubiquitous (speech recognition, machine translation, DNA, ...)
  - examples considered: character-level language modeling, image captioning
- recurrent neural net (RNN): for processing sequential data
  - use recurrence to keep track of the state of a dynamical system
    - hidden-to-hidden: more expressive but slower to train
    - output-to-hidden: less expressive but faster to train
  - training techniques: BPTT (most common), RTRL, teacher forcing
- a variety of RNN structures
  - bidirectional, encoder-decoder/sequence-to-sequence, deep, recursive
- key challenge: vanishing/exploding gradient problems
  - motivated gated RNNs (*e.g.* LSTM and GRU; next lecture)