



# M2177.003100

## Deep Learning

### [9: Recurrent Neural Nets (Part 1)]

Electrical and Computer Engineering  
Seoul National University

© 2019 Sungroh Yoon. this material is for educational uses only. some contents are based on the material provided by other paper/book authors and may be copyrighted by them.

(last compiled at 23:40:00 on 2019/10/20)

# Outline

Introduction

Various RNN Architectures

Fundamental RNN Architectures

Summary

Example: Language Modeling

# 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](#)
  - ▶ *Dive into Deep Learning* [▶ Link](#)

# Outline

## Introduction

Various RNN Architectures

Fundamental RNN Architectures

Summary

Example: Language Modeling

# Recurrent neural networks (RNNs)

- a family of neural networks for processing \_\_\_\_\_ data
- comparison:

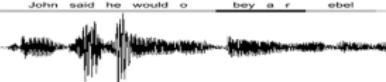
	CNN	RNN
specialized for	grid of values $\mathbf{X}$ (e.g. image)	sequence of values <sup>1</sup> $x^{(1)}, \dots, x^{(\tau)}$
scale to	wide and tall images	long sequences <sup>2</sup>
can process	images of variable size	sequences of variable length

---

<sup>1</sup>RNNs usually operate on minibatches of sequences with a different sequence length  $\tau$  for each member of the minibatch, but we omit minibatch indices to simplify notation

<sup>2</sup>much longer sequences than would be practical for networks without sequence-based specialization

# Sequential data examples

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	AAGTCTAACG <b>TAAACG</b> TCCCT
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, <b>Harry</b> Potter met <b>Hermione</b> Granger.

(source: coursera)

# RNN modeling capability

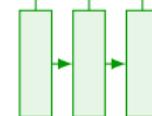
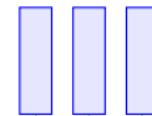
e.g. **image captioning**  
image → sentence

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

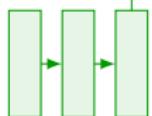
one to one



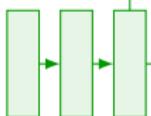
one to many



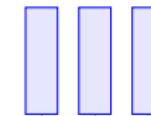
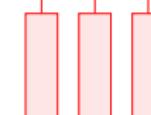
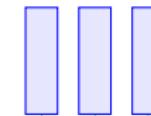
many to one



many to many



many to many



**vanilla  
neural net**

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

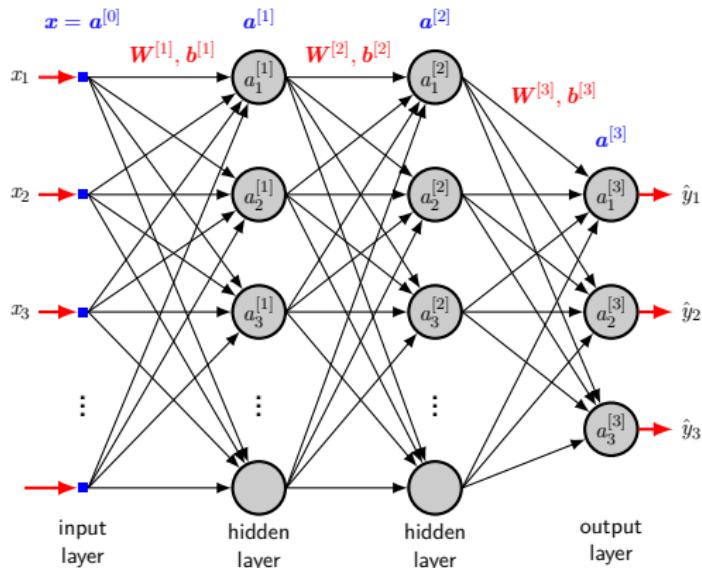
e.g. **video frame classification**  
seq of frame → seq of class

(source: cs231n)

# Deep learning RNNaissance



# Why not a vanilla neural net?



- problems
  - ▶ input/output: can be of \_\_\_\_\_ 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 \_\_\_\_\_ across
  - ▶ examples of different forms (*e.g.* different lengths in RNN)
- if we use separate parameters for each time index
  - ▶ 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 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

Introduction

Example: Language Modeling

## Fundamental RNN Architectures

Representation

Various RNN Architectures

Training

Summary

# Dynamical systems

- classical form of a dynamical system:

$$\begin{aligned}s^{(t)} &= f(s^{(t-1)}; \theta) \\ &= f_{\theta}(s^{(t-1)})\end{aligned}\tag{1}$$

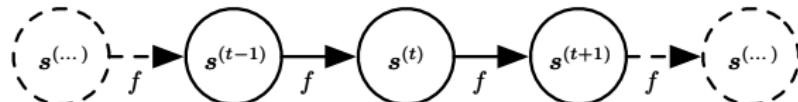
(simplified notation)

- ▶  $s^{(t)}$  : **state** of the system
- ▶ function  $f$  maps state at  $t - 1$  to state at  $t$  (parameterized by  $\theta$ )
- ▶ “      ”
  - ▷ definition of  $s$  at time  $t$  refers back to same definition at time  $t - 1$

- \_\_\_\_\_ (1) for  $\tau = 3$  time steps gives

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

- ▶ does not involve recurrence
- ▶ can be represented by a **directed acyclic computational graph**



- note: **same parameter  $\theta$**  used for all time steps
- another example: consider a dynamical system

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

- ▶ driven by external signal  $\mathbf{x}^{(t)}$
- ⇒ state now contains information about the whole past sequence

# Outline

Introduction

Example: Language Modeling

**Fundamental RNN Architectures**  
Representation  
Training

Various RNN Architectures

Summary

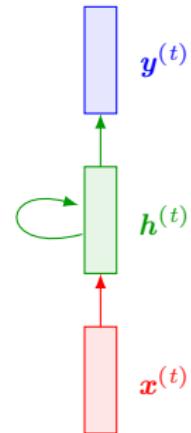
# Recurrent neural net (RNN)

- rewrite (2) using variable  $h$  to represent the state:

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

- ▶ the state consists of \_\_\_\_\_ units
- ▶ same  $f$ , same  $\theta$  at every time step
- typical RNNs add extra architectural features
  - e.g. output layers: read info out of  $h$  to make predictions

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

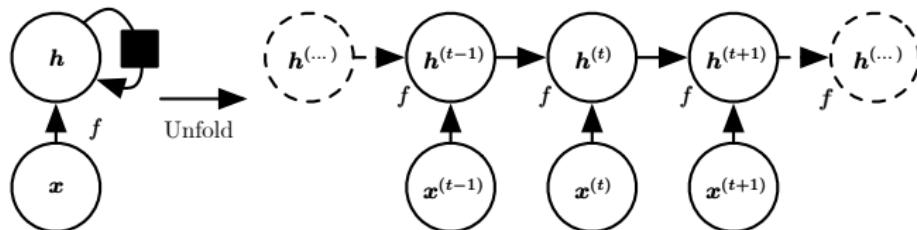


- what is the role of hidden units?

# Hidden states as a lossy summary

- assume a task: predicting future from past using an RNN
- the net learns to use  $h^{(t)}$  as a “**lossy summary**”
  - of task-relevant aspects of the past sequence of inputs up to  $t$
- this summary
  - maps  $\underbrace{x^{(t)}, x^{(t-1)}, \dots, x^{(2)}, x^{(1)}}_{\text{arbitrary length sequence}}$  to  $\underbrace{h^{(t)}}_{\text{fixed length vector}}$   $\Rightarrow$  necessarily lossy
  - may selectively keep some aspects of past sequence (more precisely)
    - e.g. language modeling (predicting next word given previous words)  
→ store only enough info for prediction (not entire input sentence)
- most demanding situation (e.g. autoencoders in ch 14)
  - $h^{(t)}$  should be \_\_\_\_\_ 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    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  $g^{(t)}$ :

$$\begin{aligned} \mathbf{h}^{(t)} &= g^{(t)}(\overbrace{\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}}^{\text{the whole past sequence as input}}) \\ &= f_{\theta}(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) \end{aligned}$$

- 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<sup>3</sup>
  - ▶  $g^{(t)}$ : a separate model for all possible time steps
- advantages of learning the same transition function  $f$ 
  - ▶ \_\_\_\_\_ (to sequence lengths not seen in training)
  - ▶ requires far fewer training examples (thanks to **parameter sharing**)

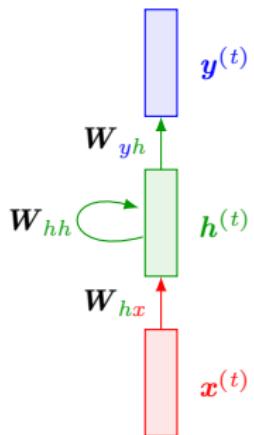
---

<sup>3</sup> $f$  is specified in terms of transition from one state to another rather than a variable-length history of states

# Simple/vanilla/Elman RNN

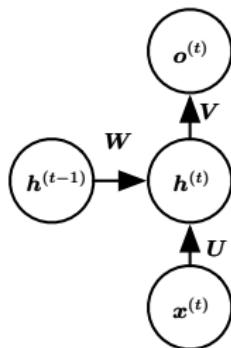
$$\begin{aligned}\underbrace{\mathbf{h}^{(t)}}_{\text{new state}} &= f_{\theta}(\underbrace{\mathbf{h}^{(t-1)}}_{\text{old state}}, \underbrace{\mathbf{x}^{(t)}}_{\text{input}}) \\ &= \tanh(\mathbf{W}_{hh} \mathbf{h}^{(t-1)} + \mathbf{W}_{hx} \mathbf{x}^{(t)} + \mathbf{b}_h)\end{aligned}$$

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



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

## Alternative notation (textbook)



$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)})$$

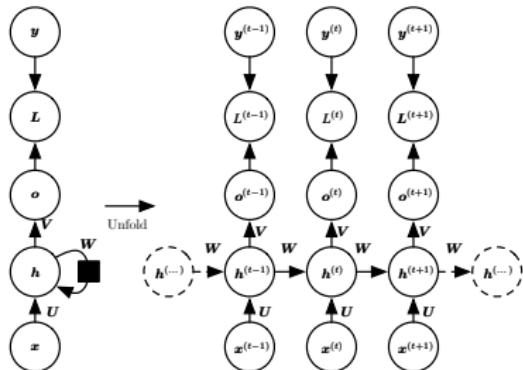
$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)})$$

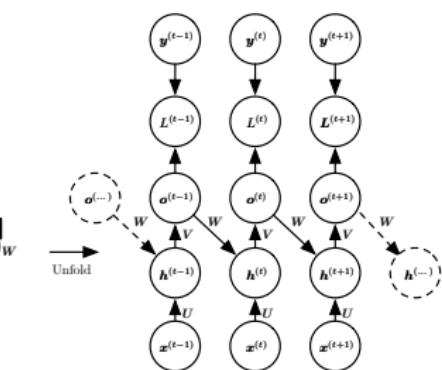
- ▶  $\mathbf{b}, \mathbf{c}$  : bias vectors
- ▶  $\underbrace{\mathbf{U}}$  ,  $\underbrace{\mathbf{V}}$  ,  $\underbrace{\mathbf{W}}$  : weight matrices
  - input-to-hidden
  - hidden-to-output
  - hidden-to-hidden

# Two types of recurrence structures

- hidden-to-hidden

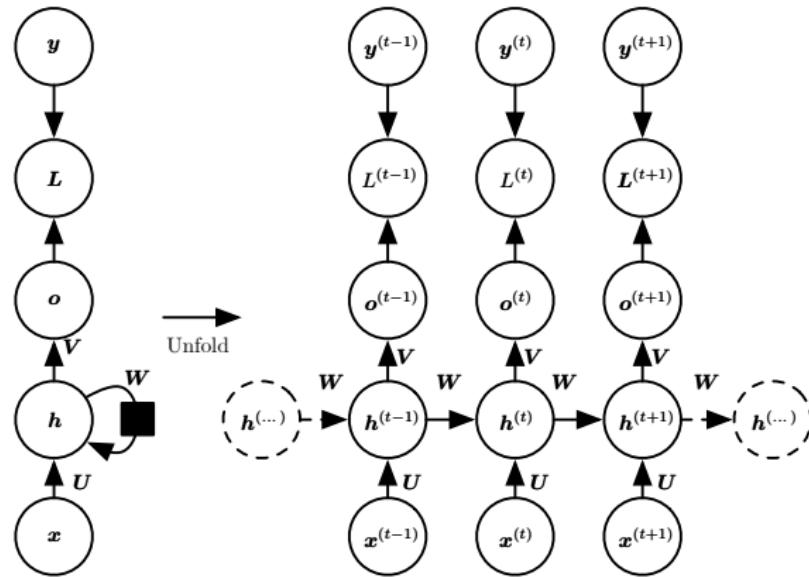


- output-to-hidden



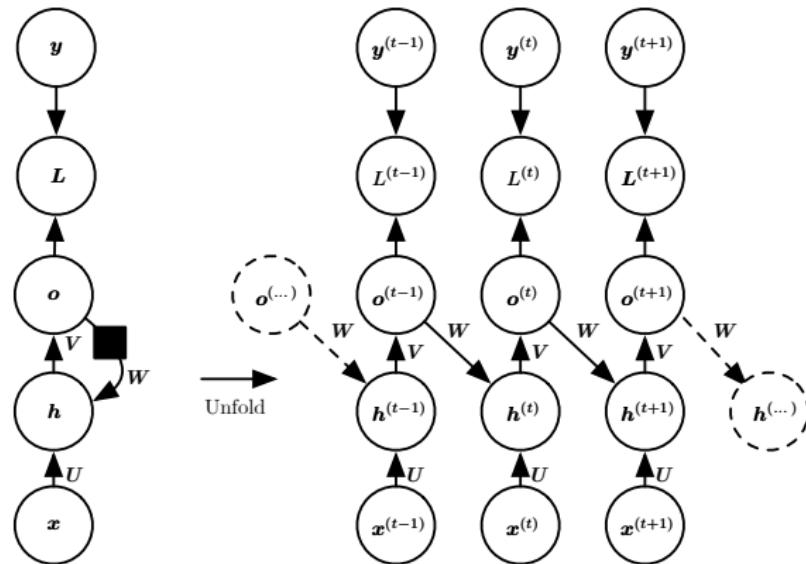
- ▶ it is possible to have \_\_\_\_ types simultaneously

## hidden-to-hidden recurrence



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

## output-to-hidden recurrence



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

# Outline

Introduction

Example: Language Modeling

## Fundamental RNN Architectures

Representation

Various RNN Architectures

Training

Summary

# RNN training

- three types
  - ▶ teacher forcing
  - ▶ backprop through time (BPTT)
  - ▶ real-time recurrent learning (RTRL)
- teacher forcing<sup>4</sup>
  - ▶ 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 \_\_\_\_\_ backprop on unfolded graph
- RTRL: training while forward prop
  - ▶ no backprop needed but computationally expensive (not used in practice)

---

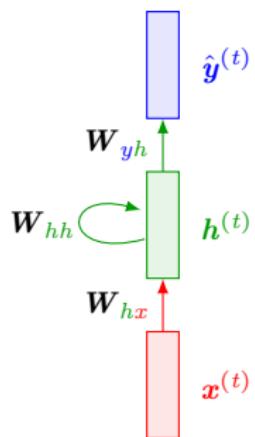
<sup>4</sup>c.f. knowledge distillation (for model compression) [Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network". In: *arXiv preprint arXiv:1503.02531* (2015)]

# Forward propagation

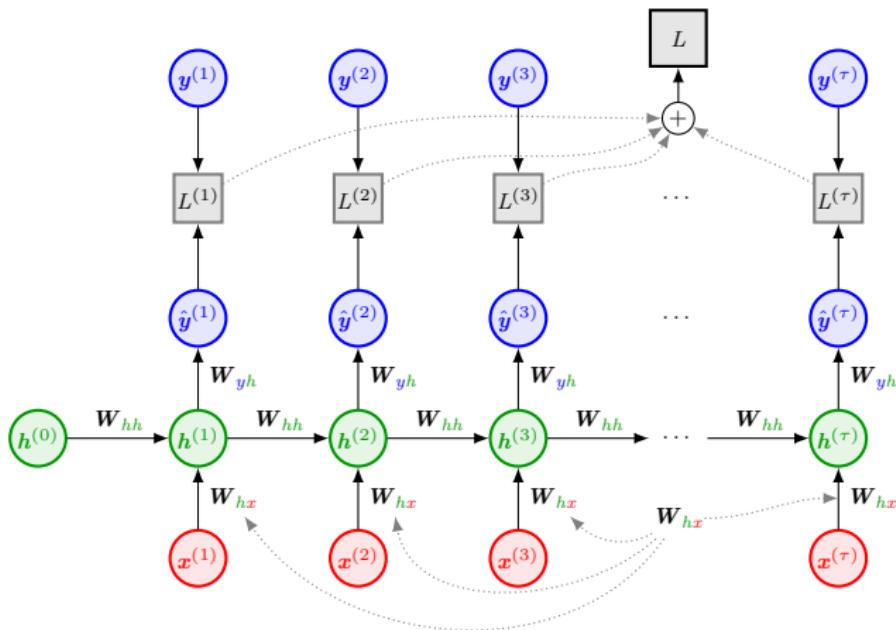
- assume a vanilla RNN
  - ▶ for  $t = 0$ : specify initial state  $\mathbf{h}^{(0)}$
  - ▶ for each time step (from  $t = 1$  to  $t = \tau$ ): apply updates

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

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



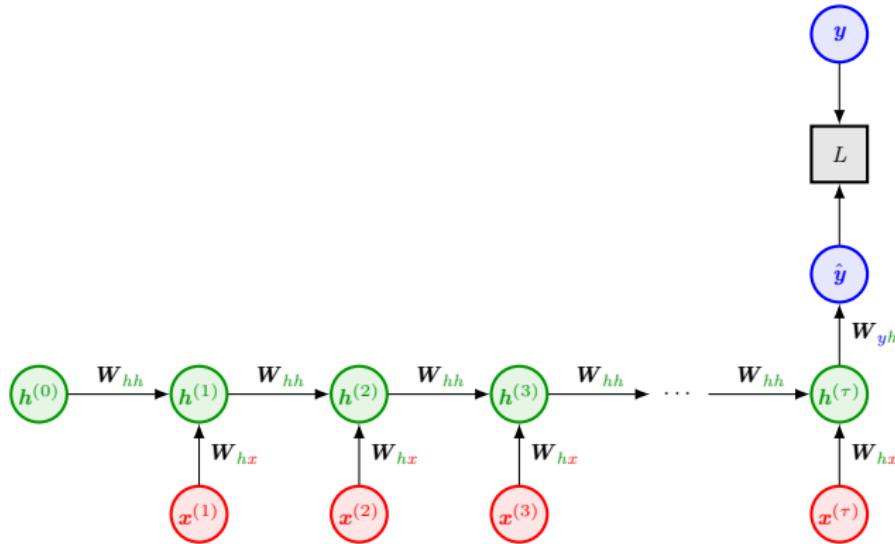
## many-to-many



- total loss for a sequence of  $x$  (paired with same-length sequence  $y$ )

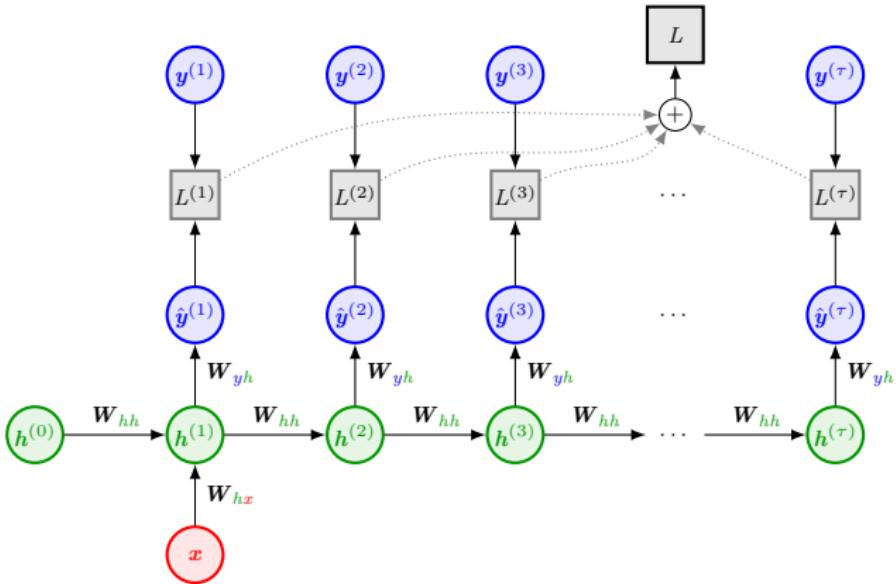
► sum of losses over    time steps:  $L = \sum_{t=1}^{\tau} L^{(t)} = \sum_{t=1}^{\tau} L(\hat{y}^{(t)}, y^{(t)})$

## many-to-one



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

one-to-many



- total loss for a vector  $x$  (label: a sequence of  $y$ )

- sum of losses over all time steps: 
$$L(\hat{y}, y) = \sum_{t=1}^{\tau} L^{(t)} = \sum_{t=1}^{\tau} L(\hat{y}^{(t)}, y^{(t)})$$

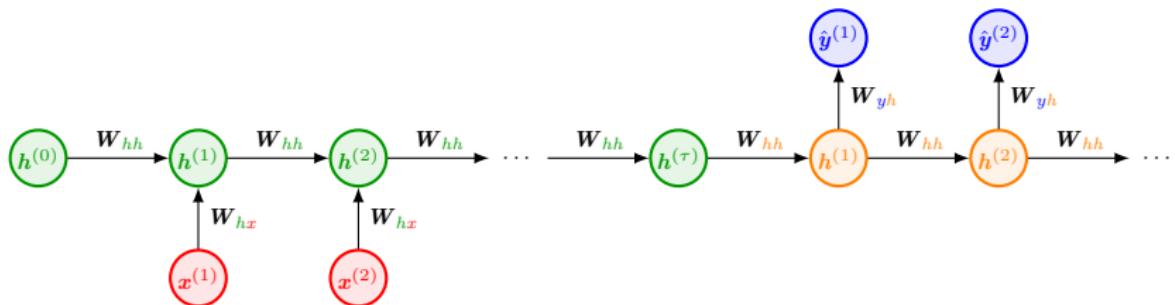
## sequence-to-sequence (= many-to-one + one-to-many)

- many-to-one

- ▶ encode input sequence into a single \_\_\_\_\_

- one-to-many

- ▶ produce output sequence from the single input vector



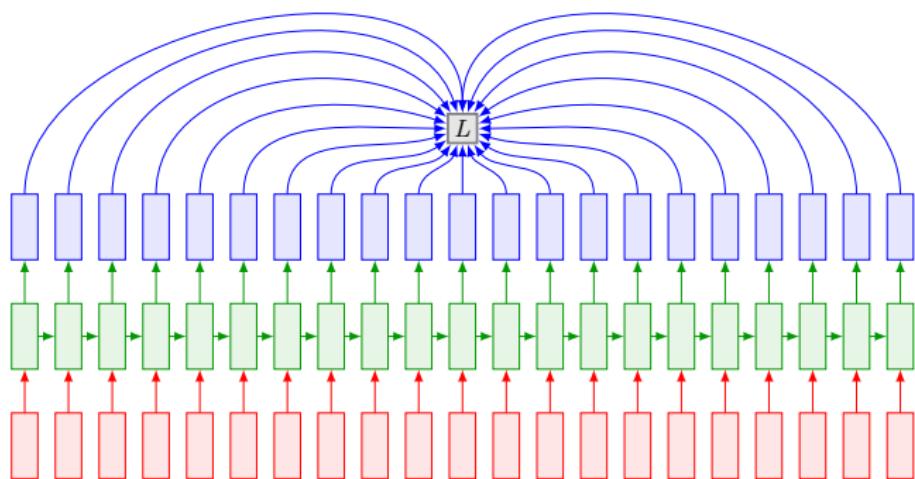
- two RNNs: trained jointly (more on this later)

# BPTT: computing gradient through RNN

- use of backprop on unrolled graph: called BPTT (backprop through time)
  - forward prop through entire sequence → compute loss
  - backprop through entire sequence → compute



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

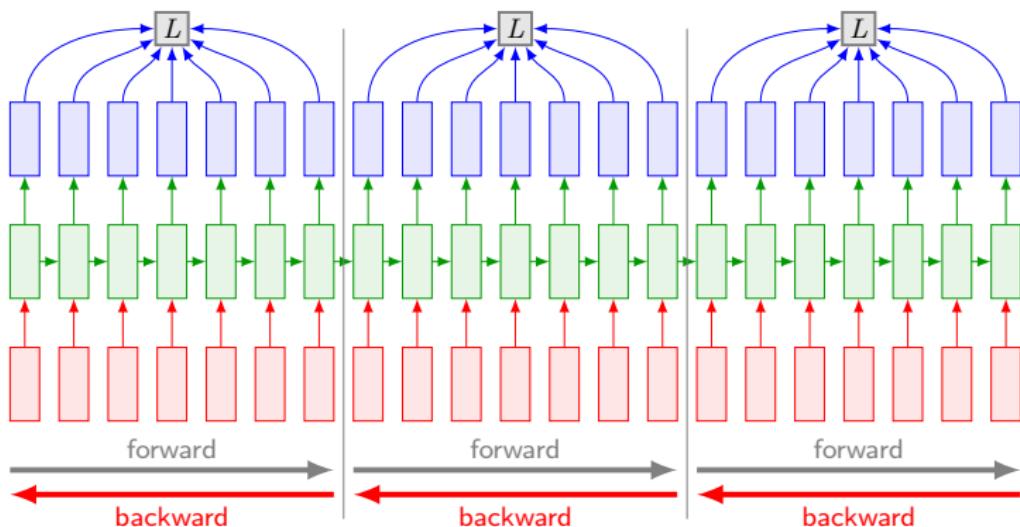


# Computational cost

- computing  $\nabla_{\theta} L$ 
  - ▶ an expensive operation
  - ▶ involves both forward/backward propagation through graph
- runtime:  $O(\tau)$  and cannot be reduced by parallelization
  - ▶ forward prop graph: inherently \_\_\_\_\_  
*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

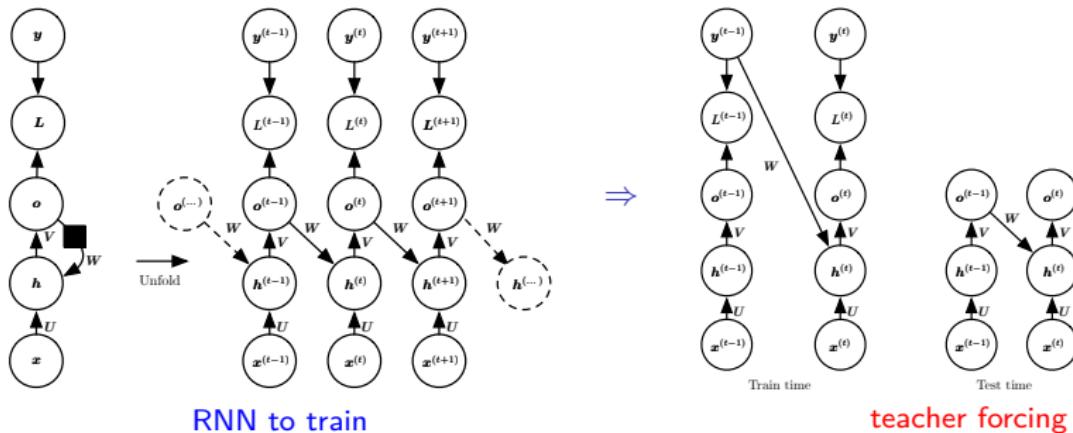
# Truncated BPTT

- 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



# Teacher forcing

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



# Outline

Introduction

Various RNN Architectures

Fundamental RNN Architectures

Summary

**Example: Language Modeling**

# Language modeling

- central to many important natural language processing (NLP)
  - ▶ used to generate text in various NLP tasks
- e.g. speech recognition, machine translation, image captioning, spelling correction, text summarization, ...
- major tasks (← inherently probabilistic)
  - ▶ assign **probability to sentences** in a language

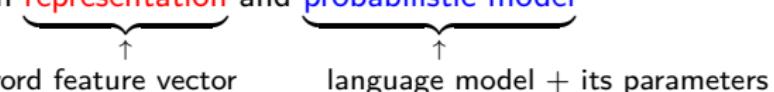
$$P(\text{apple and pair salad}) = 3.2 \times 10^{-13}$$

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

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

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

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

- 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  $\underbrace{\text{representation}}$  and  $\underbrace{\text{probabilistic model}}$   


↑  
word feature vector      ↑  
language model + its parameters
- NLM addresses the  $n$ -gram data sparsity issue through
  - ▶ parameterizing words as vectors (i.e. word \_\_\_\_\_)
  - ▶ using them as inputs to a neural net

# Character-level $n$ -gram examples

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

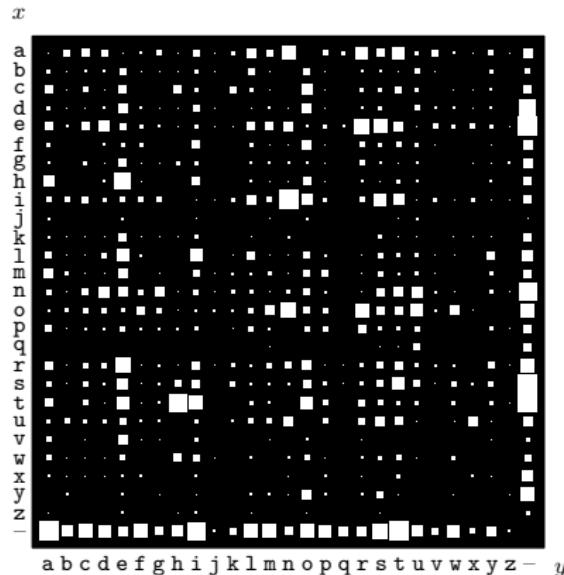
- monogram

►  $p(x)$

$i$	$a_i$	$p_i$
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

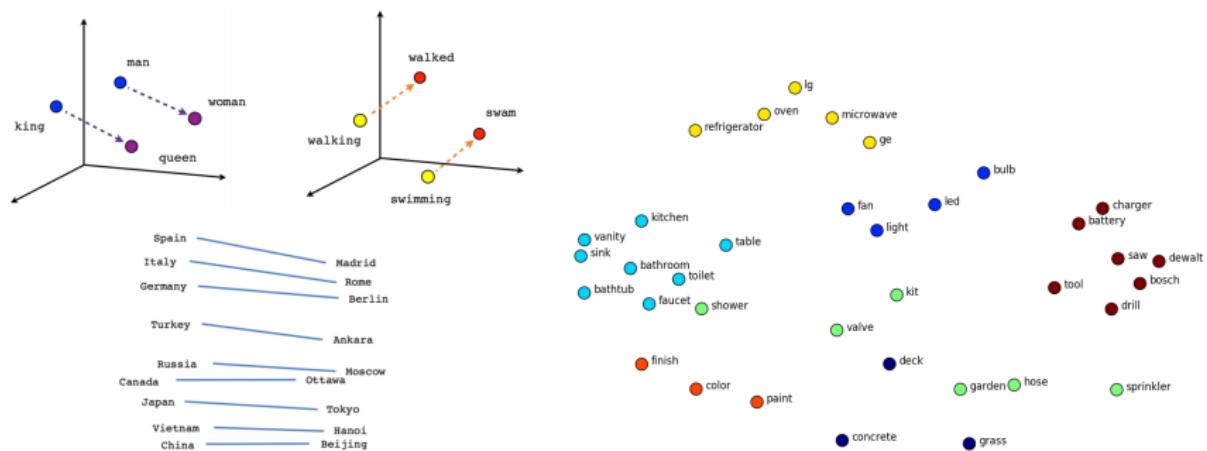
►  $p(xy)$



(source: MacKay)

# Word embedding (GloVe, ELMo)

- learned representation of words based on usage
  - uses a \_\_\_\_\_ 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 \_\_\_\_\_ in your vocabulary

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

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

- special cases

- ▶ start of sentence: <SOS>
- ▶ end of sentence: <EOS>
- ▶ unknown word: <UNK>

## step 2: supervised training

- use RNN to model the probability of different token sequences
- architecture

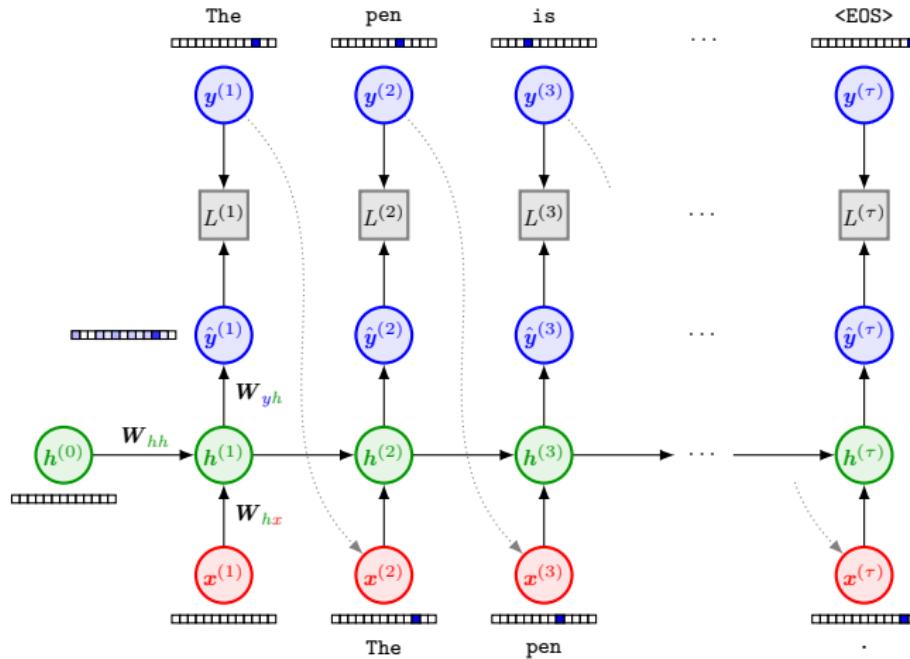
- ▶ real output  $y^{(t)}$ : vocabulary-size \_\_\_\_\_ vector
- ▶ model output  $\hat{y}^{(t)}$ : vocabulary-size \_\_\_\_\_ vector
- ▶ if we use a vanilla RNN

$$\mathbf{x}^{(t)} = \mathbf{y}^{(t-1)}$$

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

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

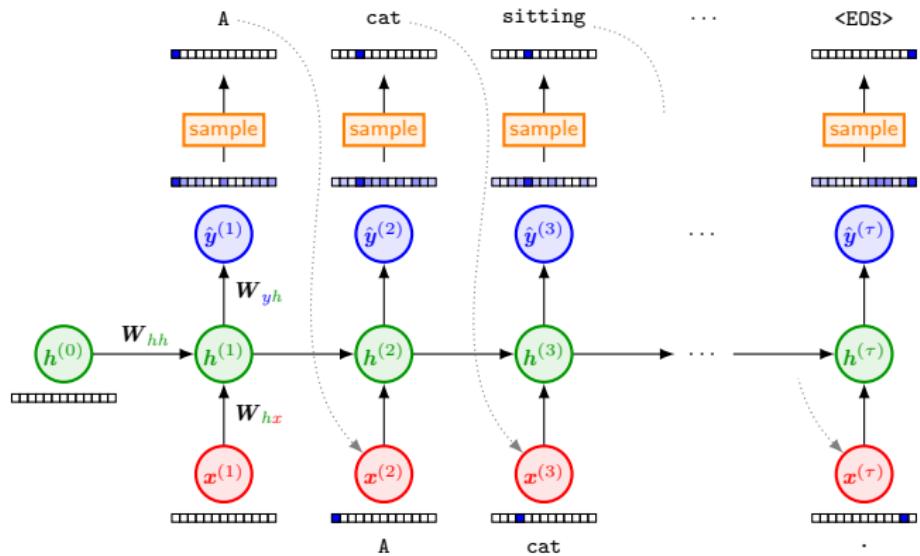
- example: The pen is mightier than the sword.<EOS>



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

# Sampling novel sentences

- use your \_\_\_\_\_ model ( $\rightarrow$  you can get a sense of what it has learned)
    - training:  $\mathbf{x}^{(t)} = \hat{\mathbf{y}}^{(t-1)}$
    - sampling:  $\mathbf{x}^{(t)} \sim \hat{\mathbf{y}}^{(t-1)}$
- same initialization:  $\mathbf{x}^{(1)} = \mathbf{0}, \mathbf{h}^{(0)} = \mathbf{0}$



- one additional complication:
  - ▶ must determine the **sequence length**  $\tau$
  - ▶ 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  $\tau$  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 \_\_\_\_\_
- in training set
  - ▶ insert this symbol as an extra member of the sequence  
(immediately after  $x^{(\tau)}$  in each training example)

## approach #2

- introduce an **extra output unit** (Bernoulli)
  - ▶ decides (at each time step) either continue generation or \_\_\_\_ generation
  - ▶ usually a sigmoid unit trained with cross-entropy loss (end or continue)
- more general than approach #1<sup>5</sup>
  - ▶ may be applied to any RNN (*e.g.* that emits a sequence of real numbers)

## approach #3

- add an **extra output** that predicts integer  $\tau$ 
  - ▶ based on the decomposition

$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}) = P(\tau)P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)} | \tau) \quad (34)$$

⇒ model can sample a value of  $\tau$  and then sample  $\tau$  steps worth of data

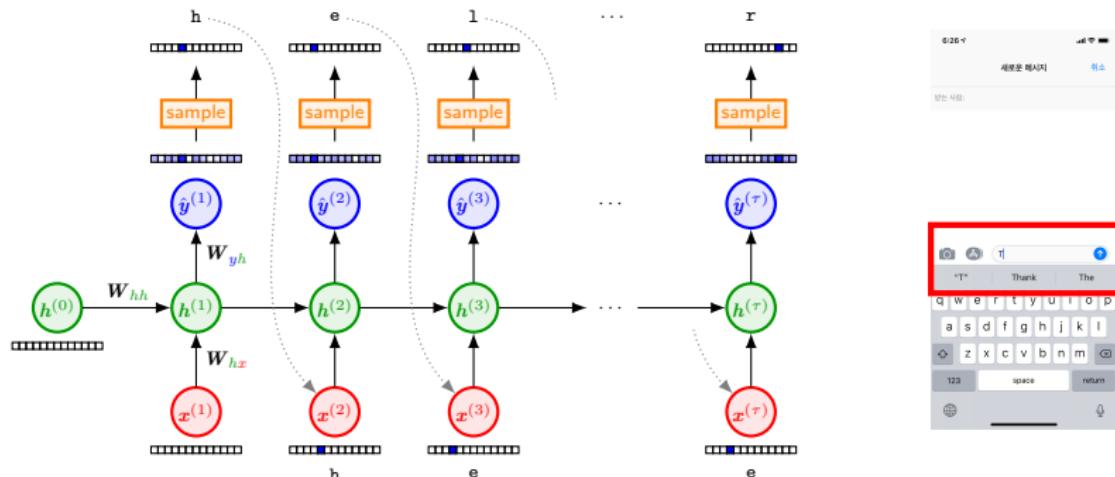
---

<sup>5</sup>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: \_\_\_\_\_-level language modeling
  - vocabulary = [a, b, c, ..., z, Ȑ, ., , ;, ..., 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.

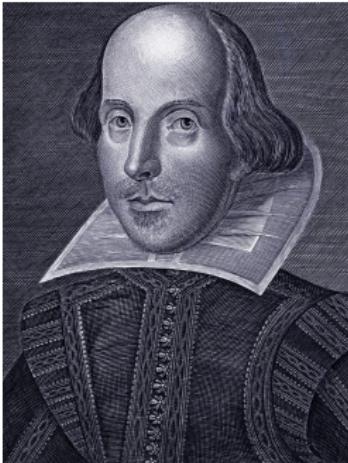
 **karpathy / min-char-rnn.py**  
Last active 22 hours ago

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

```
1  ***
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD license
4  ***
5  Import numpy as np
6
7  # data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(set(data))
10 vocab_size = len(chars)
11 embed_size = 8
12 hidden_size = 100
13 seq_length = 25 # number of steps to unroll the RNN for
14 learning_rate = 1e-1
15
16 # hyperparameters
17 hidden_size = 100 # size of hidden layer of neurons
18 seq_length = 25 # number of steps to unroll the RNN for
19 learning_rate = 1e-1
20
21 # model parameters
22 Wih = np.zeros((hidden_size, vocab_size))*.01 # input to hidden
23 Whh = np.random.rand(hidden_size, hidden_size)*.01 # hidden to hidden
24 Why = np.random.rand(vocab_size, hidden_size)*.01 # hidden to output
25 bh = np.zeros((hidden_size, 1)) # hidden bias
26 by = np.zeros((vocab_size, 1)) # output bias
27
28 def lossFun(inputs, targets, hprev):
29     """ Compute loss and gradients over all time steps
30     inputs,targets are both list of integers.
31     hprev is last array of hidden state
32     returns the loss, gradients on model parameters, and last hidden state
33     """
34
35     x, hx, ys, ps = [], {}, {}, {}
36     hs[-1] = np.copy(hprev)
37     loss = 0
38
39     # forward pass
40     for t in range(len(inputs)):
41         x[t] = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
42         x[t][inputs[t]] = 1
43
44         hprev = np.tanh(np.dot(Wih, x[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
45         ps[t] = np.tanh(np.dot(Why, hprev[t]) + by) # unnormalize log probabilities for next char
46         ps[t] = ps[t]/np.sum(ps[t]) # softmax (cross-entropy loss)
47         loss += -np.log(ps[t][targets[t]]) # softmax (cross-entropy loss)
48
49         # backward pass: compute gradients going backwards
50         dprev, dhb, dyb = np.zeros_like(hprev), np.zeros_like(why), np.zeros_like(by)
51         dby = np.zeros_like(by)
52         dnewhs = np.zeros_like(hs[0])
53
54         for t1 in reversed(range(len(inputs))):
55             dy = np.copy(ps[t1])
56             dy[targets[t1]] += 1 # backprop into y, see http://cs231n.github.io/neural-networks-case-study/#grad
```

```
51         dyb += np.dot(dy, hs[t].T)
52         dby += dy
53         dhb = np.tanh(dy, T, dy) + dnewhs # backprop into h
54         dnewhs = (1 - hs[t]**2) * dhb**2 # backprop through tanh nonlinearity
55         dnewhs *= dhb
56         dnewhs = np.dot(dnewhs, xs[t].T)
57         dnewhs += np.dot(dnewhs, hs[t-1].T)
58         dnewhs = np.tanh(dnewhs, T, dnewhs)
59         for t2 in range(len(inputs)-1, t, -1):
60             dx = np.zeros_like(xs[t2])
61             dx[t2] = dnewhs
62             dnewhs = np.clip(dxparam, -5, 5, out=dnewhs) # clip to mitigate exploding gradients
63             return loss, dnewhs, dnewh, dhb, dyb, hs[len(inputs)-1]
64
65 def sample(h, seed_ix, n):
66     """sample a sequence of integers from the model
67     h is memory state, seed_ix is seed letter for first time step
68     """
69     x = np.zeros((vocab_size, 1))
70     x[seed_ix] += 1
71     ix = seed_ix
72     for t in range(n):
73         h = np.tanh(np.dot(Whh, x) + np.dot(Wih, h) + bh)
74         y = np.dot(Why, h) + by
75         p = np.exp(y) / np.sum(np.exp(y))
76         ix = np.random.choice(range(vocab_size), p=p.ravel())
77         x[ix] = 1
78     return ix
79
80
81 x, h, y, p, mhy, mbh, myb = np.zeros_like(h), np.zeros_like(h), np.zeros_like(why)
82 mbh, myb = np.zeros_like(h), np.zeros_like(h) # memory variables for Adagrad
83 smooth_loss, loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
84 while True:
85     # prepare inputs (we're sweeping from left to right in steps seq_length long)
86     if seq_length > len(inputs):
87         inputs = inputs[-seq_length:]
88     hprev = np.zeros((hidden_size, 1)) # reset RNN memory
89     p = # go from start of data
90     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93     # sample from the model now and then
94     if n % 100 == 0:
95         sample_ix = sample(hprev, inputs[0], 200)
96         txt = ''.join(ix_to_char[i] for i in sample_ix)
97         print('...'.join('%.3f' % x for x in hprev))
98
99     # forward seq_length characters through the net and fetch gradient
100     x = np.zeros((vocab_size, 1))
101     x[inputs[0]] = 1
102     hprev = np.tanh(np.dot(Wih, x) + np.dot(Whh, h) + bh) # hidden state
103     ps = np.tanh(np.dot(Why, hprev[0]) + by) # unnormalize log probabilities for next char
104     ps = ps/np.sum(ps) # softmax (cross-entropy loss)
105     loss += -np.log(ps[targets[0]]) # softmax (cross-entropy loss)
106
107     # backward pass: compute gradients going backwards
108     dprev, dhb, dyb, dby, hprev = lossFun(inputs, targets, hprev)
109     smooth_loss = smooth_loss * 0.999 + loss * 0.001
110     if n % 100 == 0: print('Iter %d, loss: %.4f' % (n, smooth_loss)) # print progress
111
112     # perform parameter update with Adagrad
113     for param, dparam, mem in zip([Wih, Whh, why, bh, by],
114                                   [dprev, dhb, dyb, dby, hprev],
115                                   [mhy, mbh, myb, mh, mh]):
116         mem += dparam * dparam
117         param -= learning_rate * dparam / np.sqrt(mem + 1e-8) # adgrad update
118
119         p += seq_length # move data pointer
120
121     n += 1 # iteration counter
```

- generating Shakespeare-style text



William Shakespeare (1564–1616)

(source: wikipedia, cs231n)

tyntd-iafhatawiaoahrdemot 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."

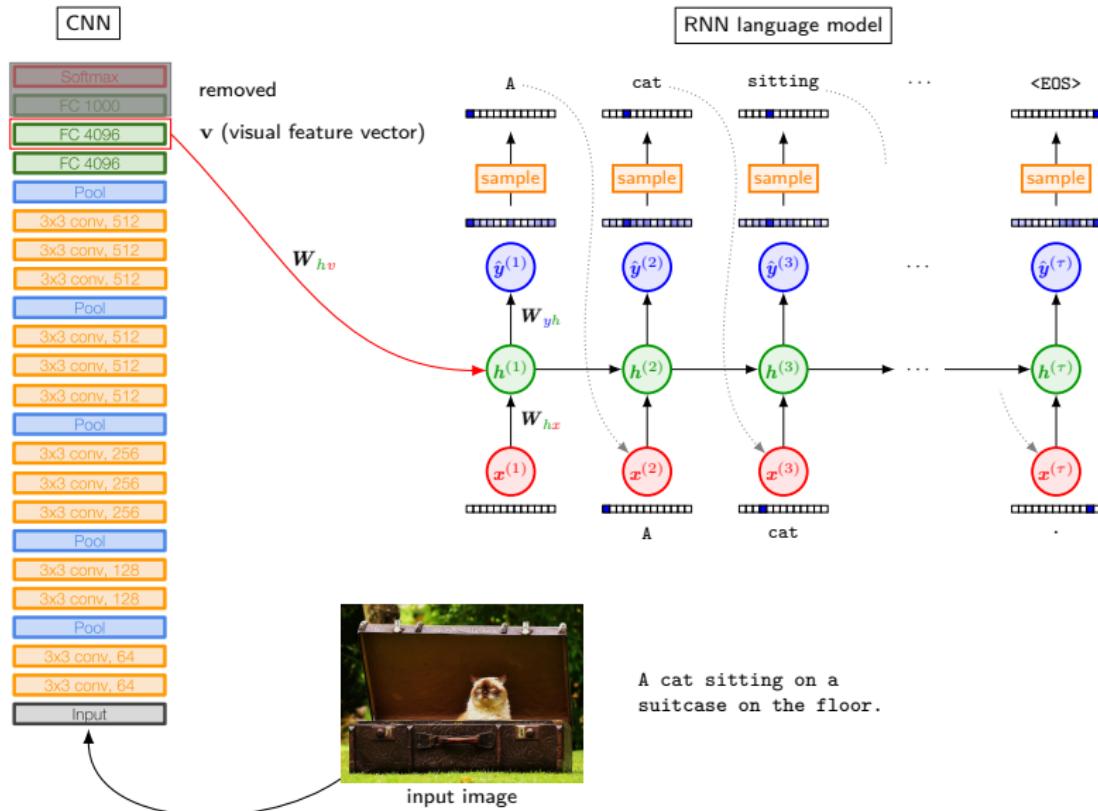
train more

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 = \_\_\_\_\_ 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

Introduction

Fundamental RNN Architectures

Example: Language Modeling

Various RNN Architectures

**Bidirectional RNN**

Encoder-Decoder Architecture

Deep RNN

Recursive RNN

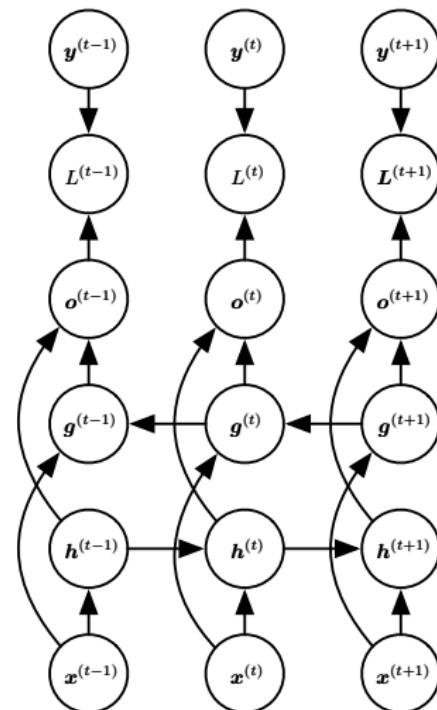
Summary

# Motivation

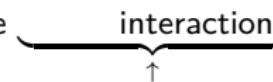
- all RNNs considered up to now: “\_\_\_\_\_” structure
  - i.e. state at time  $t$  only captures information from
    - ▷ past  $(x^{(1)}, \dots, x^{(t-1)})$  and present input  $x^{(t)}$
- but often output  $y^{(t)}$  may depend on the *whole input sequence*
  - e.g. speech recognition, handwriting 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

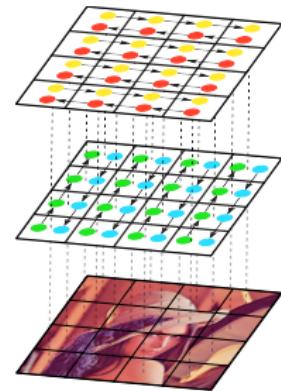
# Bidirectional RNN

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



## 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  interactions between features in the same feature map



(source: Visin et al., 2015)

# Outline

Introduction

Fundamental RNN Architectures

Example: Language Modeling

Various RNN Architectures

Bidirectional RNN

**Encoder-Decoder Architecture**

Deep RNN

Recursive RNN

Summary

# Recall: RNN modeling capability

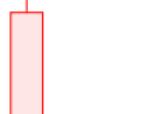
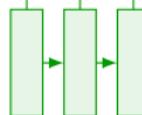
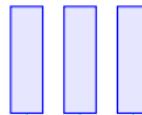
e.g. **image captioning**  
image → sentence

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

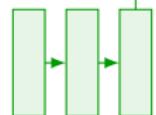
one to one



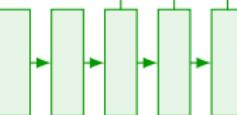
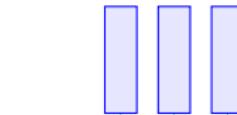
one to many



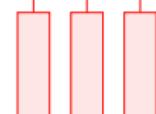
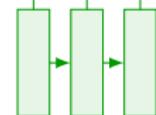
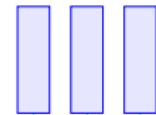
many to one



many to many



many to many



**vanilla  
neural net**

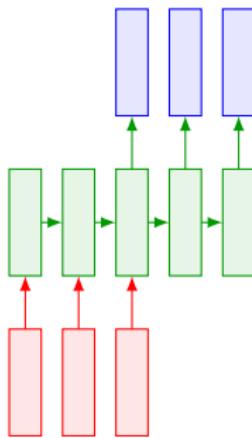
e.g. **sentiment classification**  
sentence → sentiment

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

(source: cs231n)

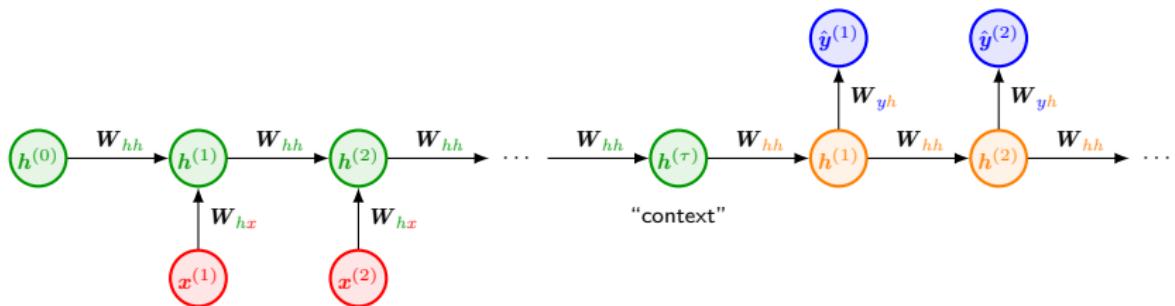
# Mapping sequence to sequence (of different lengths)

- RNN can be trained to map an input sequence to an output sequence
  - ▶ which is \_\_ 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)

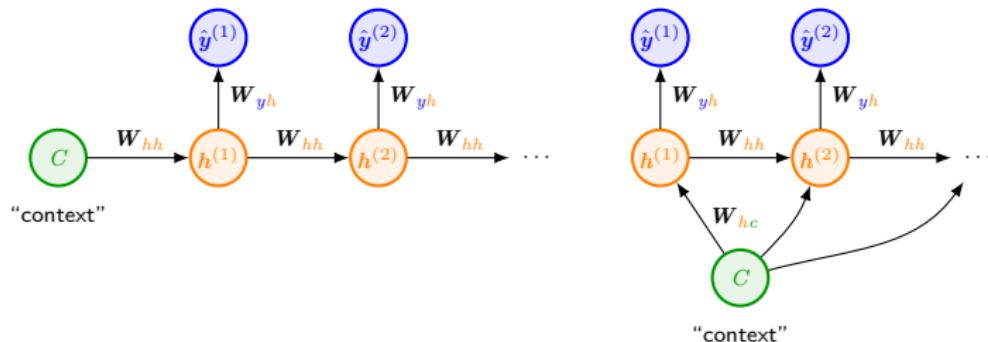
- many-to-one (encoder)
  - ▶ encode input sequence into a single vector
- one-to-many (decoder)
  - ▶ produce output sequence from the single input vector



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

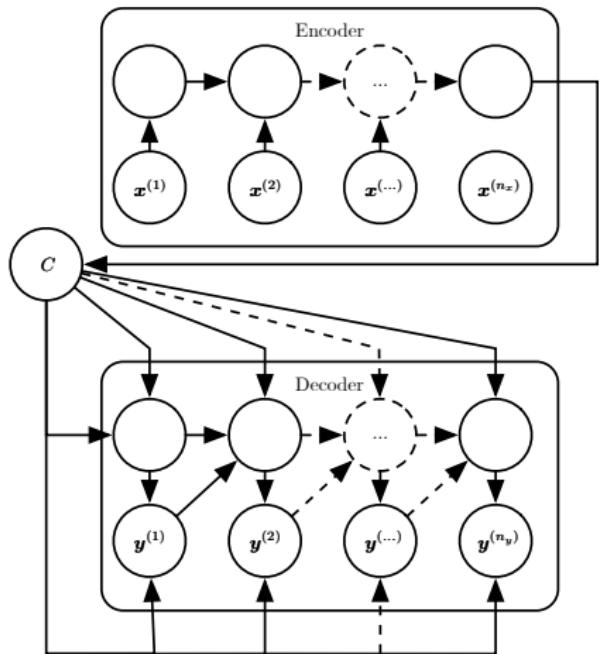
# Receiving inputs in decoder

- a one-to-many (vector-to-sequence) RNN
  - ▶ can receive input in two ways
    1. input can be provided as \_\_\_\_\_ state of RNN, or
    2. input can be connected to hidden units at **each time step**



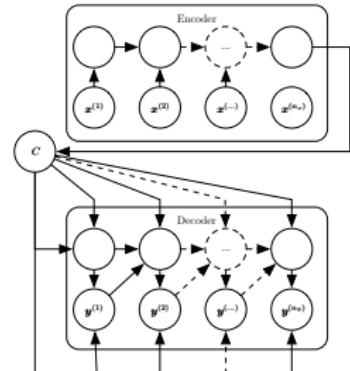
# Encoder-decoder/sequence-to-sequence architecture

- goal: map a variable-length sequence to another variable-length sequence
  - ▶ first application: machine translation
- two names
  - ▶ sequence-to-sequence 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  $\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n_x)})$
- ▶ emits context  $C$  (generally a fixed-size vector)
  - usually a simple function of final hidden state  $h_{n_x}$
  - represents a semantic summary of input seq

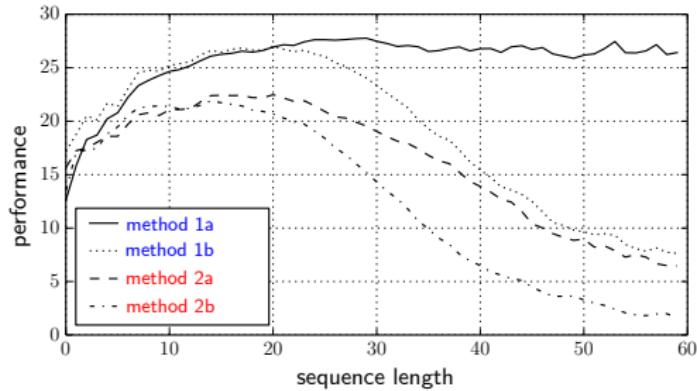


## (2) decoder/writer/output RNN:

- ▶ takes context  $C$  as input
- ▶ generates output sequence  $\mathbf{Y} = (\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)})$
- encoder/decoder RNNs: trained \_\_\_\_\_
  - ▶ to maximize the average of  $\log P(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n_x)})$   
over all the pairs of  $\mathbf{X}$  and  $\mathbf{Y}$  sequences in training set
- innovation of this architecture: not necessarily  $n_x = n_y$

# Attention mechanism

- one clear limitation
  - ▶ dimension of context  $C$  may be too small to summarize a long sequence

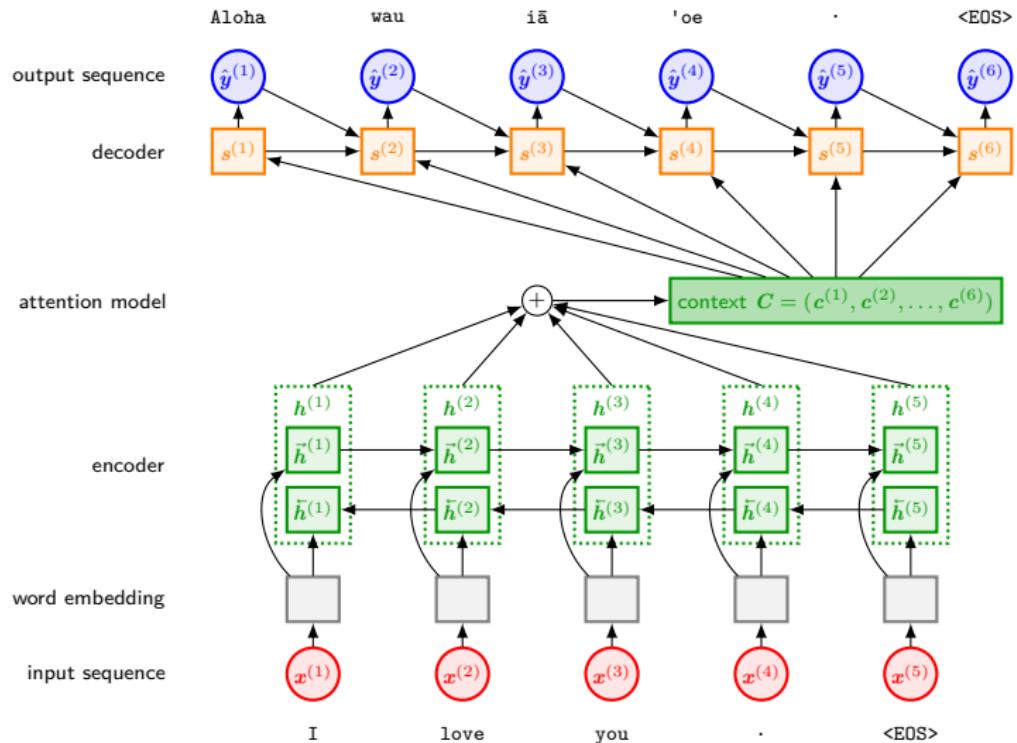


- methods 1 and 2
  - ▶ with and without attention, respectively
- a and b
  - ▶ trained with max 50 and 30 words, respectively

(source: Bahdanau et al., 2015)

- attention mechanism
  - ▶ makes  $C$  a variable-length \_\_\_\_\_ (not a fixed-size vector)
  - ▶ learns to associate
    - element  $c^{(t)}$  of sequence  $C \leftrightarrow$  element  $y^{(t)}$  of output sequence

- example



- more details: [Link](#)

- let  $t \in [1, n_y]$  and  $t' \in [1, n_x]$
- $\underbrace{t}_{\text{output index}}$        $\underbrace{t'}_{\text{input index}}$

- context sequence  $C = (c^{(1)}, c^{(2)}, \dots, c^{(n_y)})$

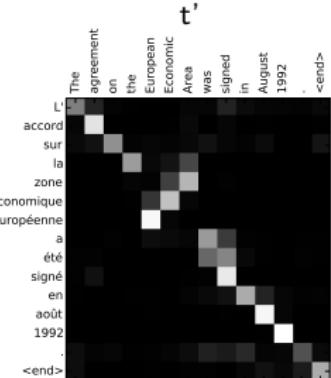
$$c^{(t)} = \sum_{t'=1}^{n_x} \alpha^{(t,t')} h^{(t')}$$

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

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

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

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



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

English → French

- $t'$ : English

- $t$ : French

(source: Bahdanau, 2015)

# Outline

Introduction

Fundamental RNN Architectures

Example: Language Modeling

Various RNN Architectures

Bidirectional RNN

Encoder-Decoder Architecture

**Deep RNN**

Recursive RNN

Summary

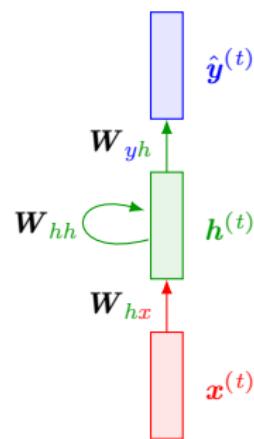
# Computation in most RNNs

- 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

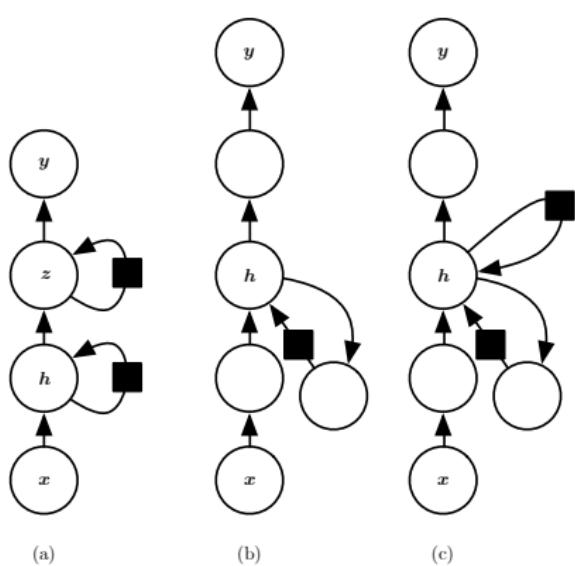
- each of these three blocks
  - ▶ associated with a single weight matrix
  - ⇒ corresponds to a transformation

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



# Deep RNN

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

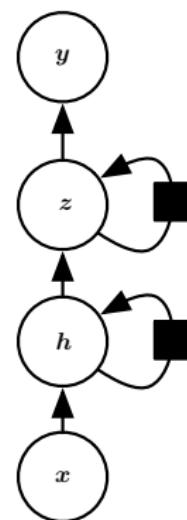


- three examples:

- (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 \_\_\_\_\_
- lower layer  $\rightarrow$  higher layer in hierarchy
  - ▶ raw data  $\rightarrow$  higher-level representation
- in practice
  - ▶ hidden layer depth: not much greater than 2–3



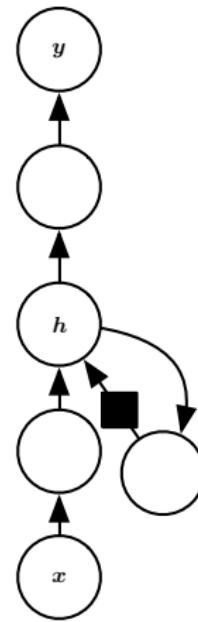
(a)

## 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 \_\_\_\_\_

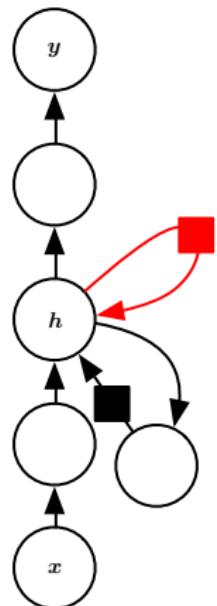
from a variable in step  $t$   
to a variable in step  $t + 1$



(b)

## Example (c)

- path-lengthening can be mitigated by
  - ▶ introducing \_\_\_ connections in hidden-to-hidden path



(c)

# Outline

Introduction

Fundamental RNN Architectures

Example: Language Modeling

Various RNN Architectures

Bidirectional RNN

Encoder-Decoder Architecture

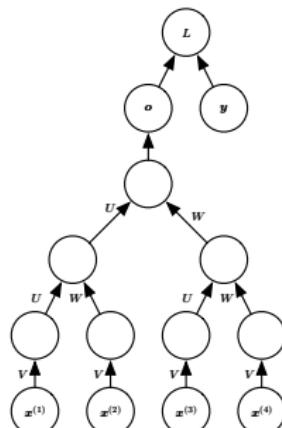
Deep RNN

**Recursive RNN**

Summary

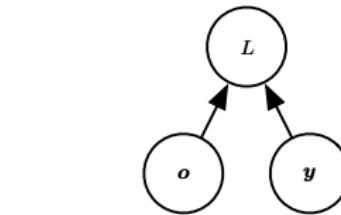
# Recursive neural networks<sup>6</sup>

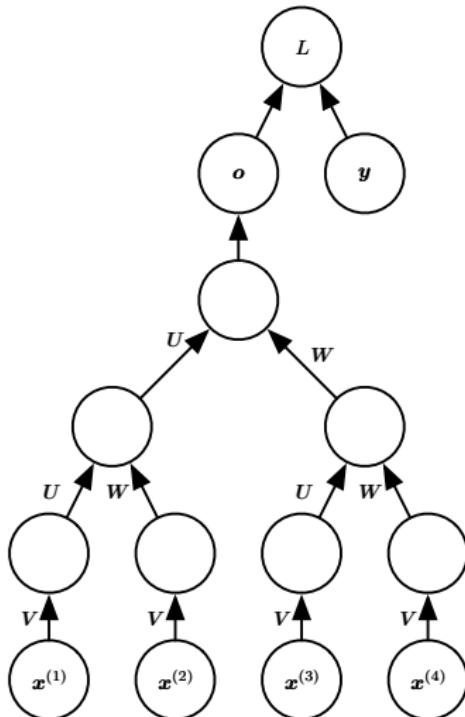
- represent yet another generalization of recurrent nets
  - ▶ has potential use for learning to reason
- comparison: computational graph
  - ▶ recurrent neural nets: **chain-like** structure
  - ▶ recursive neural nets: structured as a **deep** \_\_\_\_\_
- have been successfully applied to
  - ▶ processing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



<sup>6</sup>do not abbreviate “recursive neural network” as “RNN” to avoid confusion with “recurrent neural network”

# Example

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



# Benefits

- one clear advantage of recursive nets over RNNs:
  - ▶ for a sequence of the same length  $\tau$ ,
  - ▶ the  can be drastically reduced from  $\tau$  to  $O(\log \tau)$   
  
(measured as the number of compositions of nonlinear operations)
- ⇒ 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

Introduction

Various RNN Architectures

Fundamental RNN Architectures

**Summary**

Example: Language Modeling

# 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)