COSCO VII

Recurrent Neural Network (RNN) Software Implementation using *Python*

June 29th, 2017
Presenter: Ryunosuke Murakami
Demonstration: Maiko Arakawa
Supporters: Hiroki Yomogita,
Tomohide Fukuchi, Yohei Shimmyo

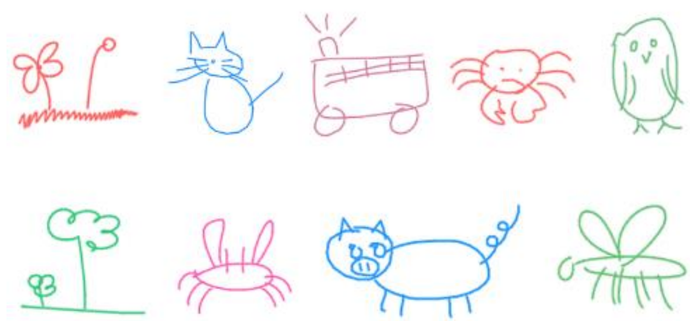
Outline

- Introduction
- Recurrent architecture
- Learning algorithm
  - Backpropagation in RNN.
  - Long-short term memory.
  - Connectionist Temporal Classification.
- Summary
- Demo
  - machine translation model

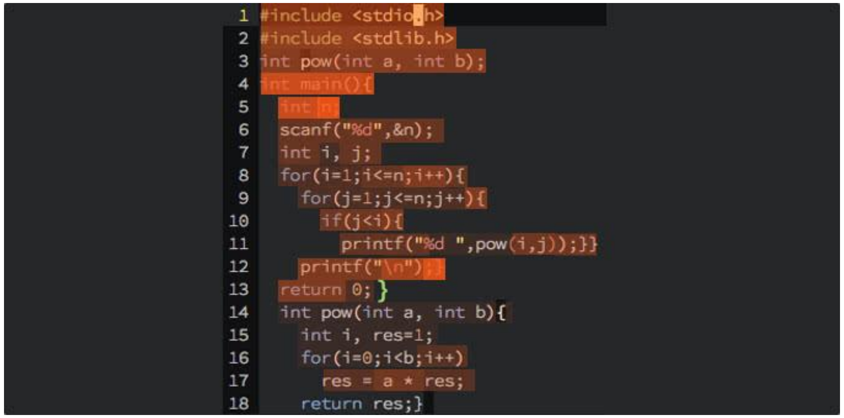
What's RNN?

Recurrent Neural Network can handle

- Voice data
- Language data
- Movie data



Sketch-rnn (Google)



DeepFix (Indian Institute of Science)

Machine translation

We can get an idea of the learned feature vectors by displaying.

word	We	can	get	...	the	learned	?
input	x^1	x^2	x^3		x^{t-1}	x^t	x^{t+1}
output		y^1	y^2		y^{t-2}	y^{t-1}	y^t

Estimation task for next word from given inputs

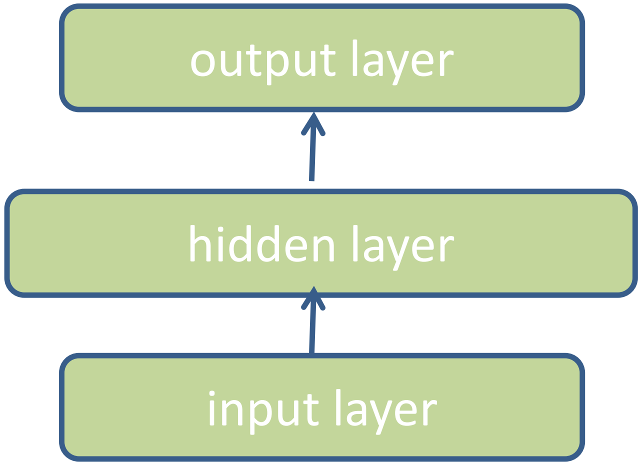
x^t : series of input

y^t : *series of output*

Difference of structure

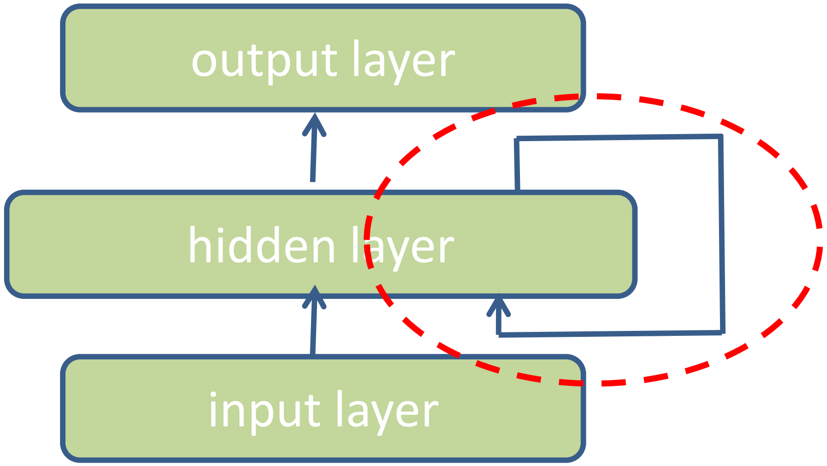
Feed-forward NN

- Directed acyclic graph

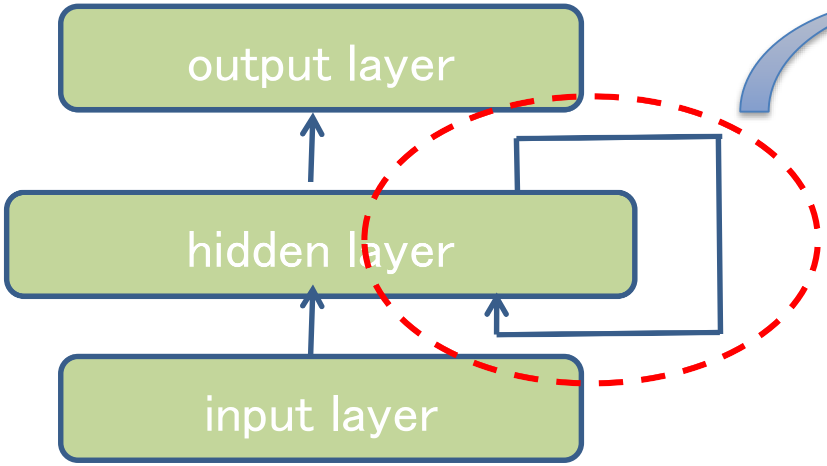


Recurrent NN

- Directed cycle graph



Why use RNN?



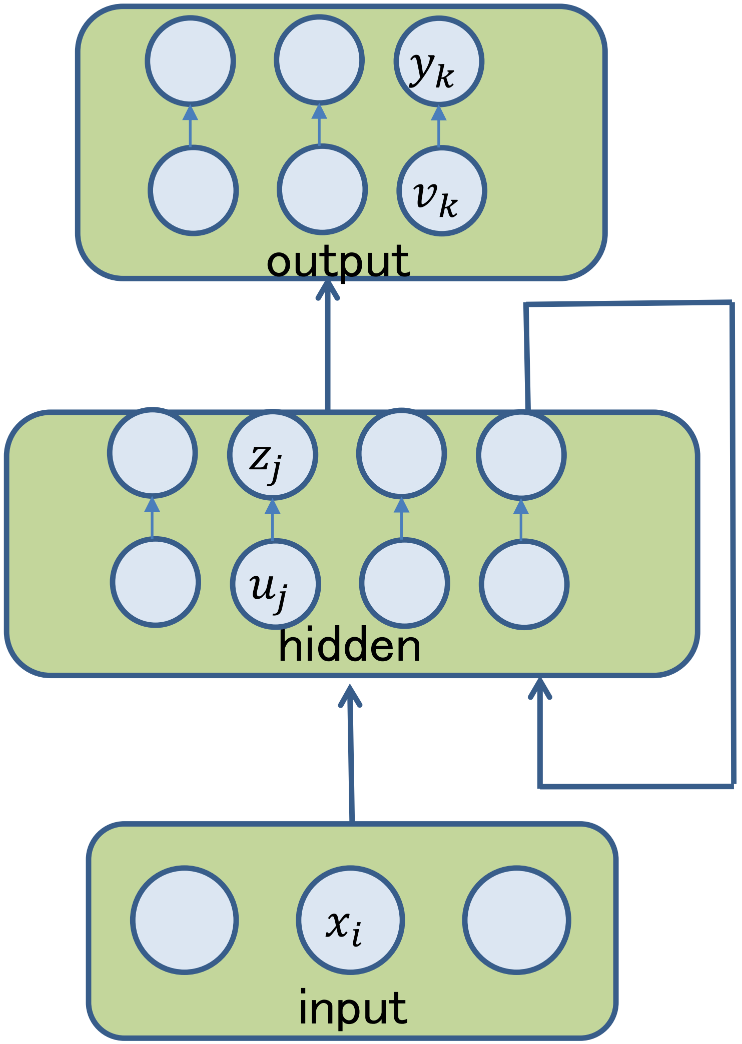
– Use previous hidden state as input

- Learn context and dependence between words
- High accuracy of word expectation

Forward propagation of RNN

- Input of hidden layer
  - $u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$
- Output of hidden layer
  - $z_j^t = f(u_j^t)$
- Input of output layer
  - $v_k^t = \sum_j w_{kj}^{(out)} z_j^t$
- Output of output layer
  - $y_k^t = f(v_k^t)$

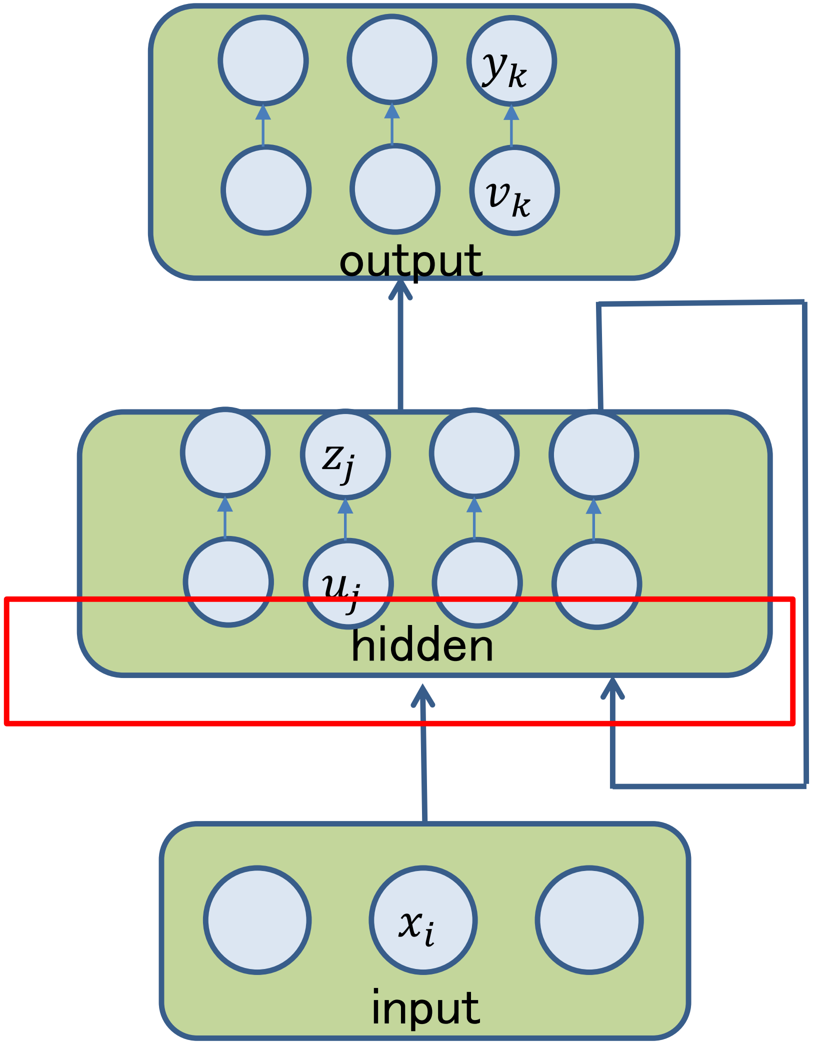
where t is time



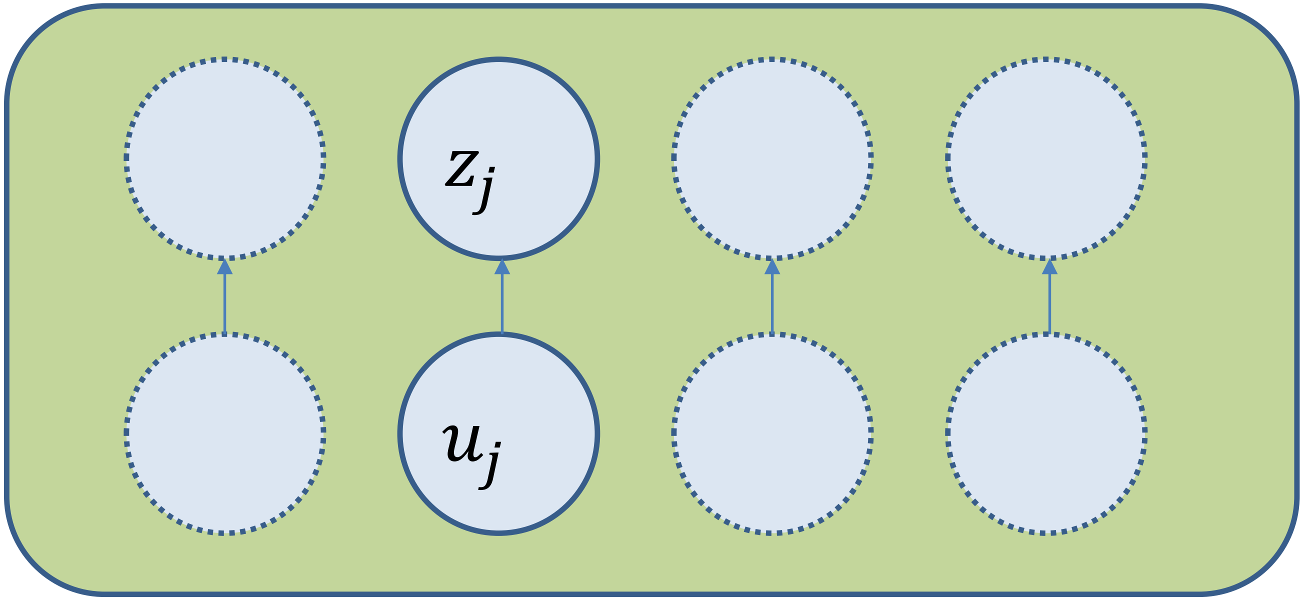
Forward propagation of RNN

- Input of hidden layer
  - $u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$
- Output of hidden layer
  - $z_j^t = f(u_j^t)$
- Input of output layer
  - $v_k^t = \sum_j w_{kj}^{(out)} z_j^t$
- Output of output layer
  - $y_k^t = f(v_k^t)$

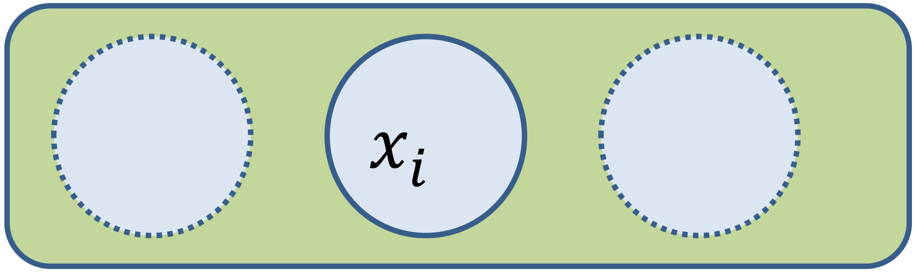
where t is time



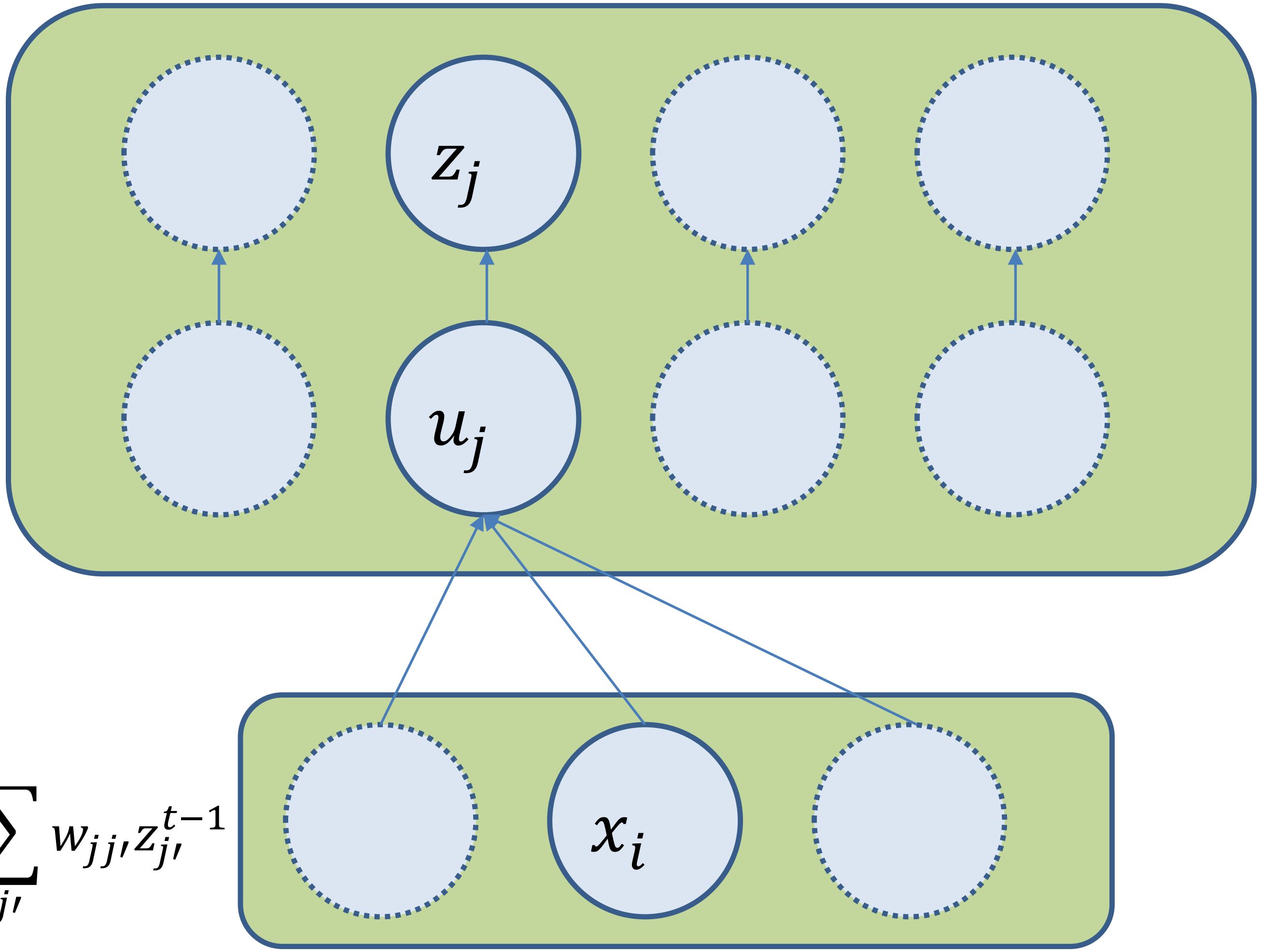
Detail of input of hidden layer



$$u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$$

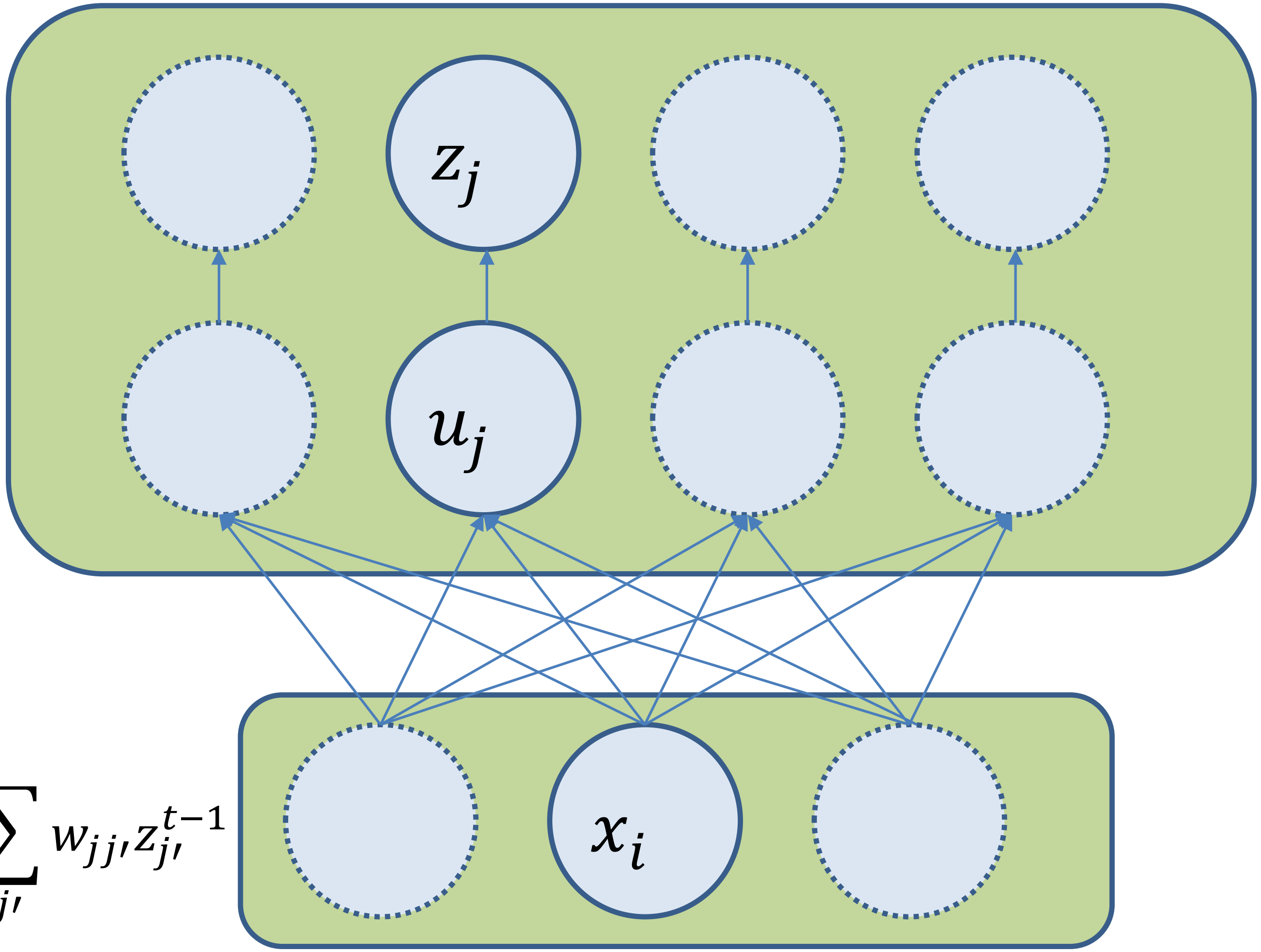


Detail of input of hidden layer



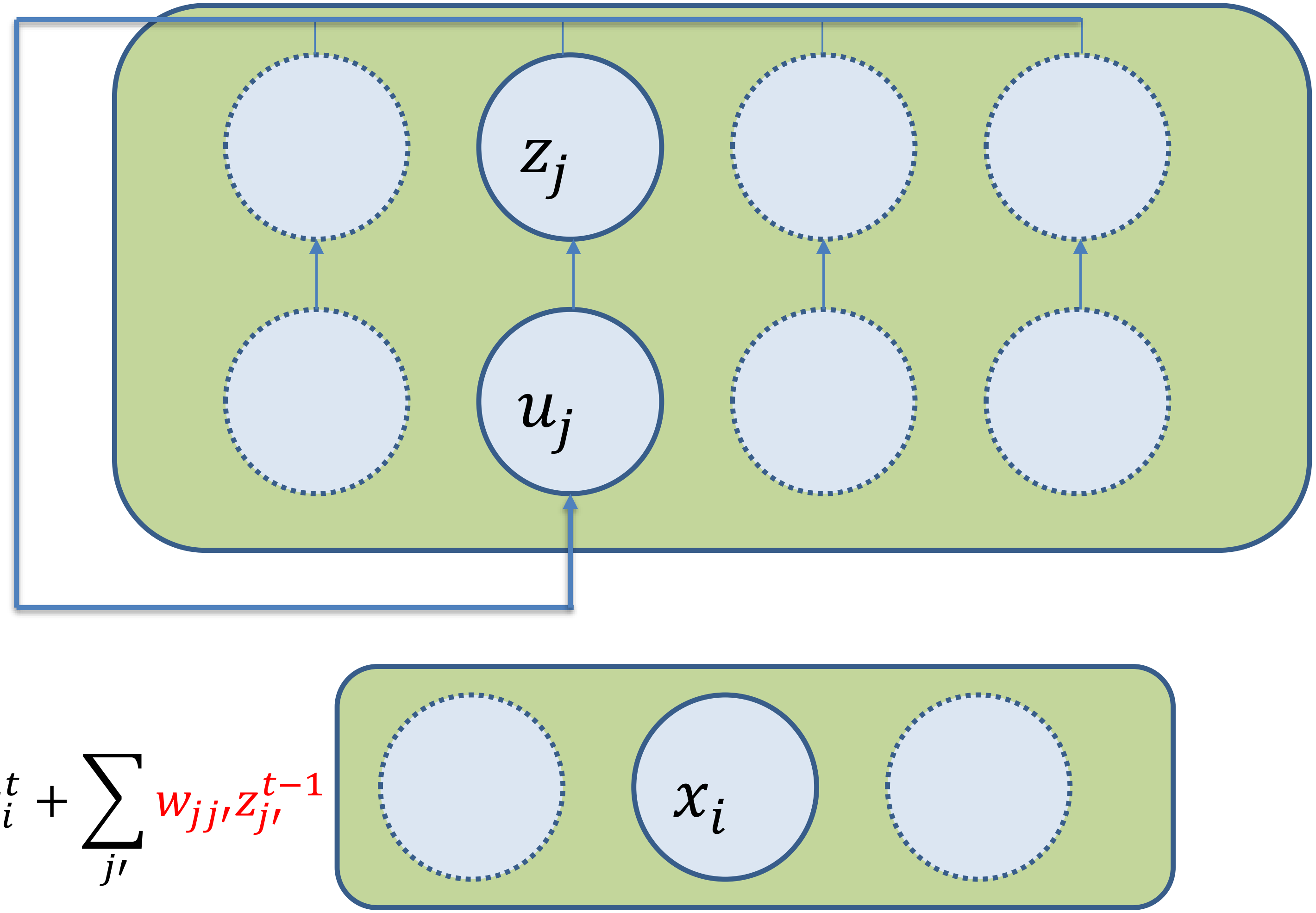
$$u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$$

Detail of input of hidden layer



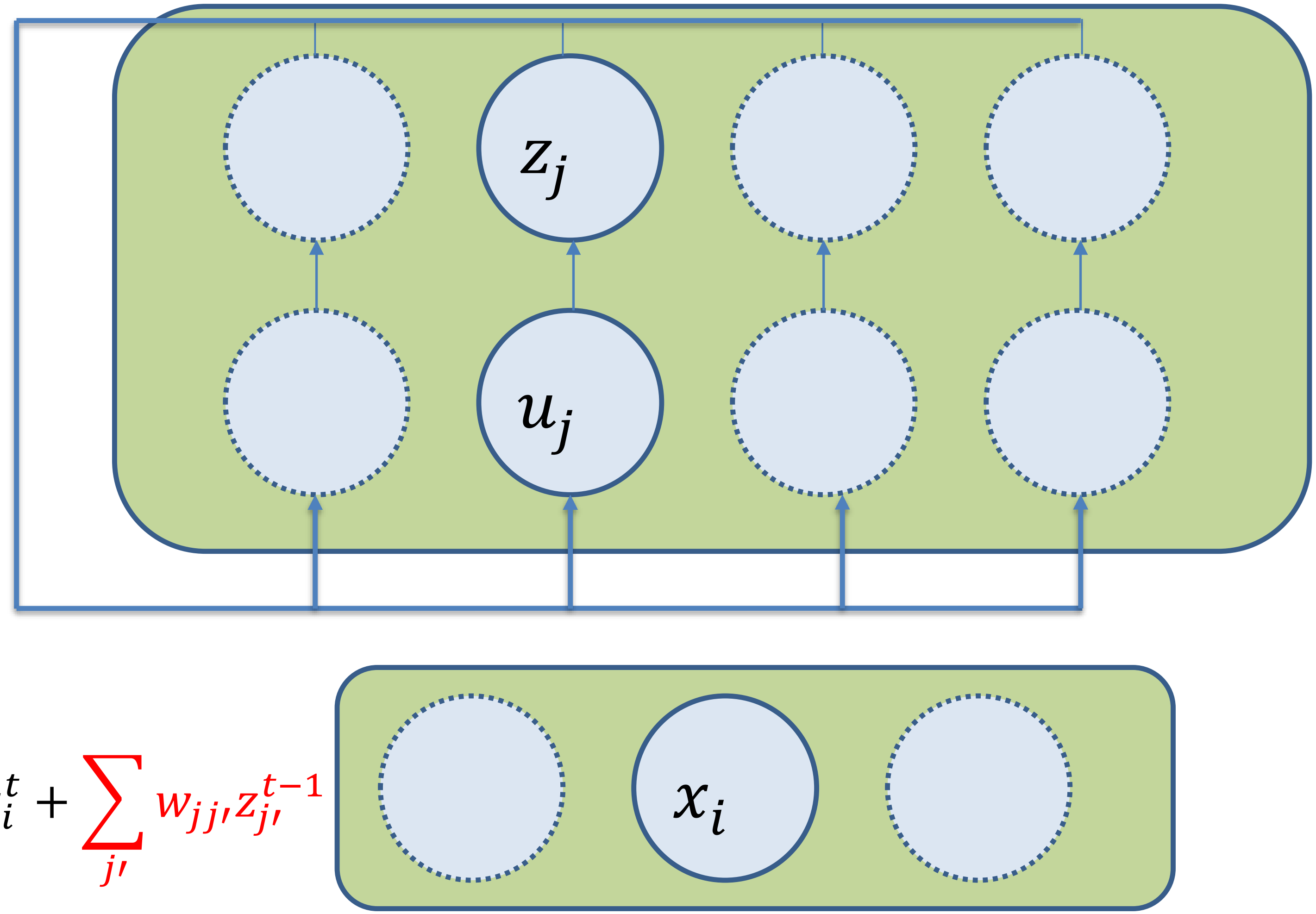
$$u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$$

Detail of input of hidden layer



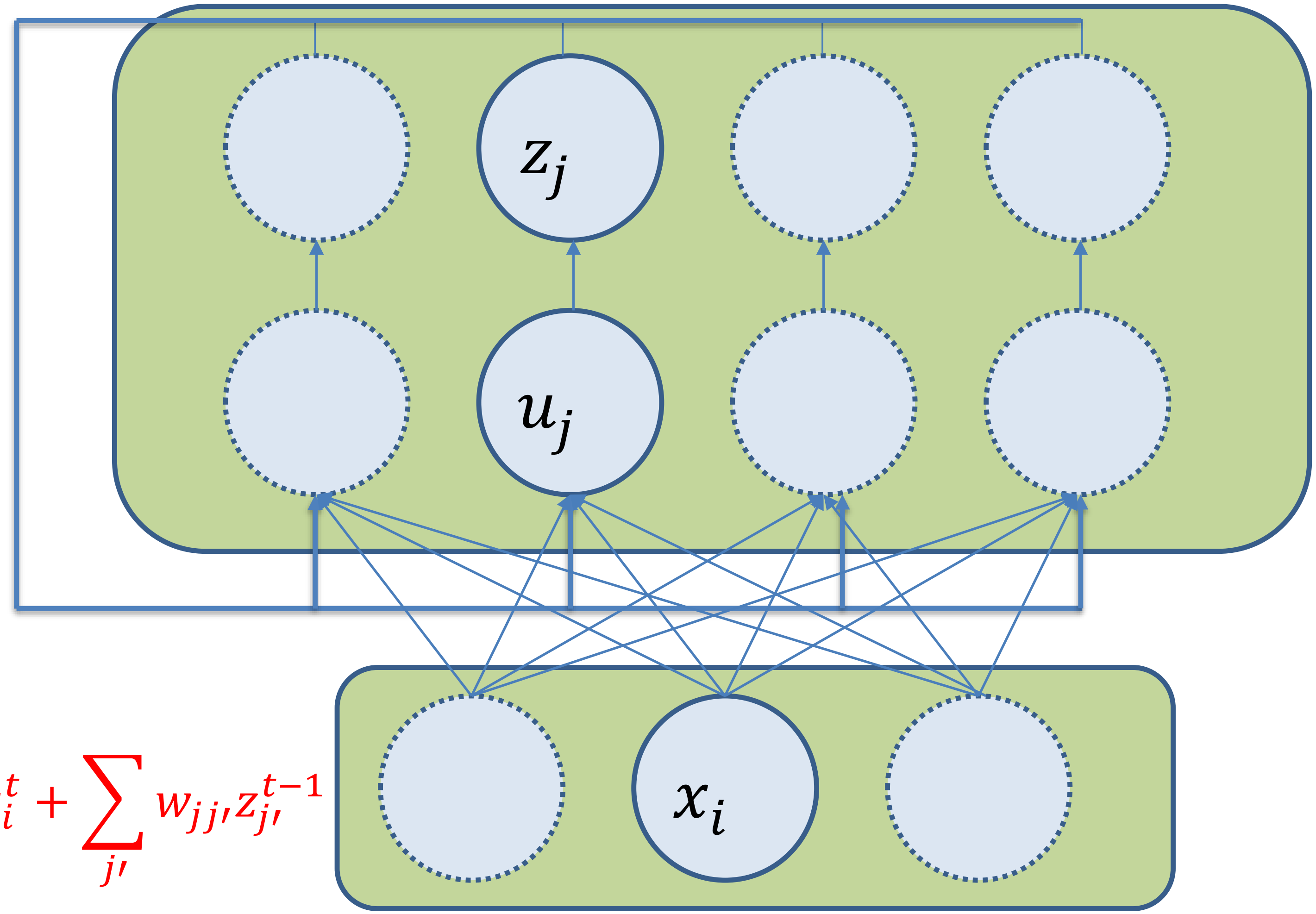
$$u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$$

Detail of input of hidden layer



$$u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$$

Detail of input of hidden layer

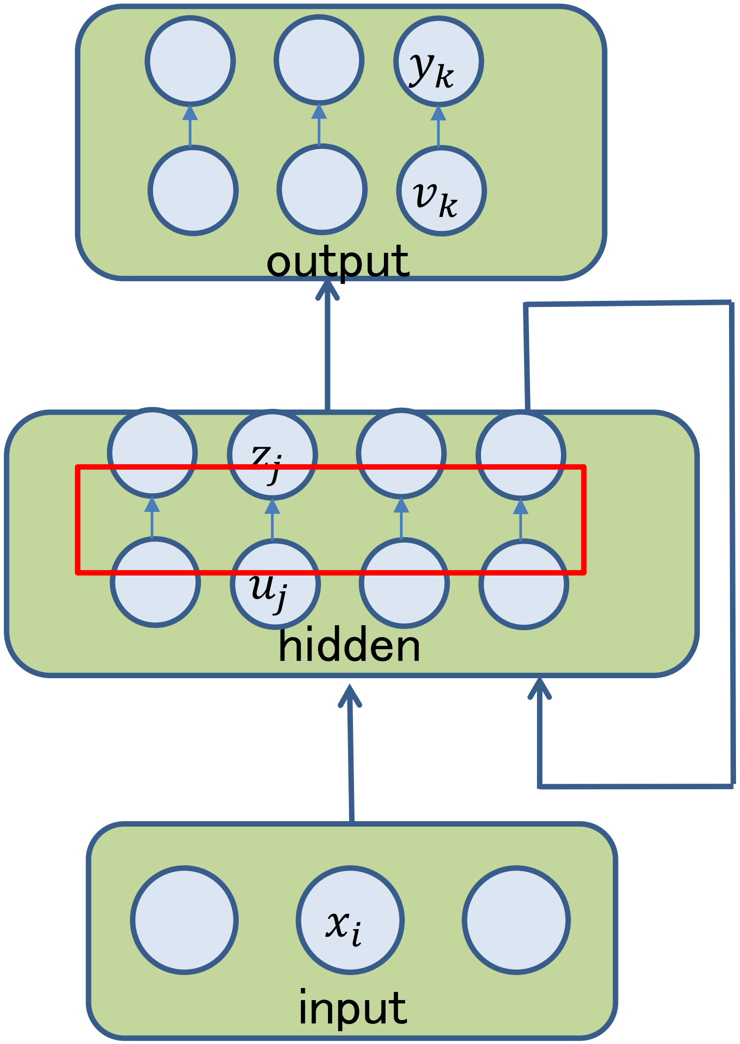


$$u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$$

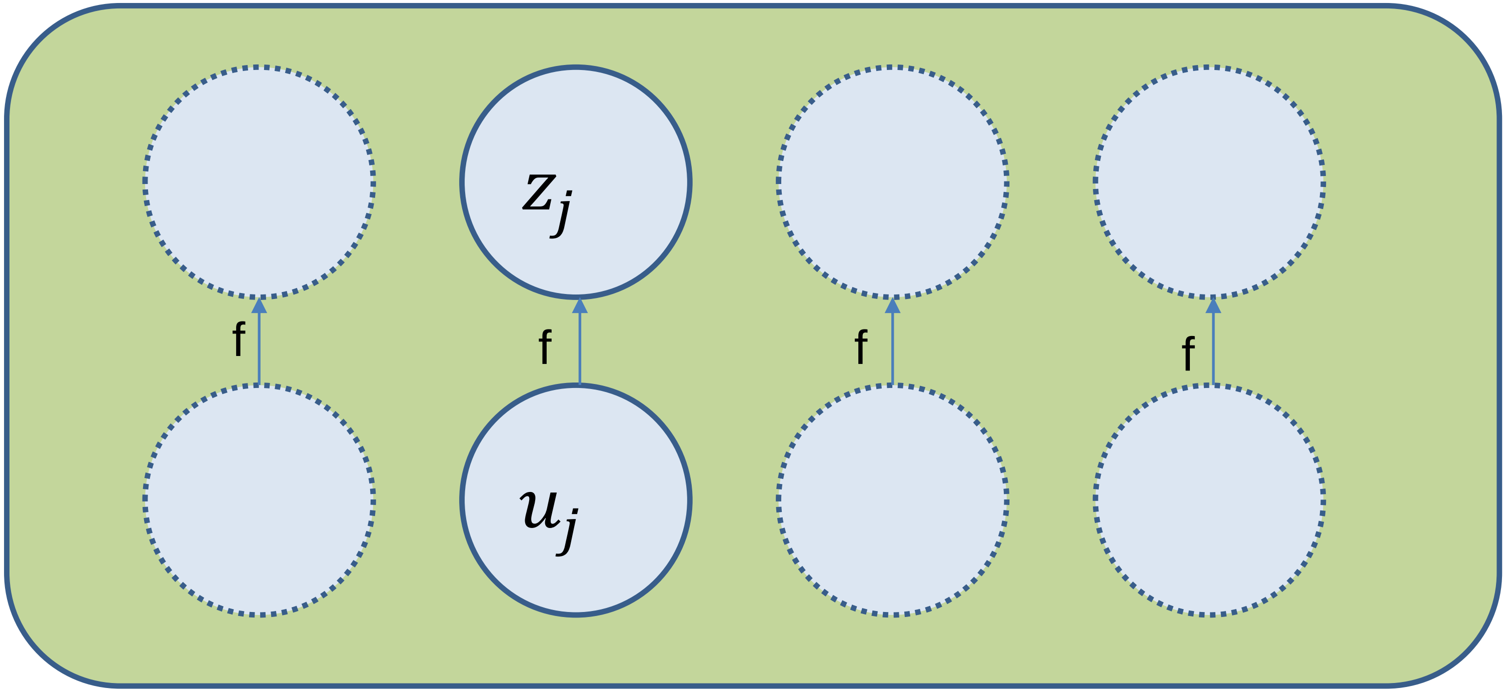
Forward propagation of RNN

- Input of hidden layer
  - $u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$
- Output of hidden layer
  - $z_j^t = f(u_j^t)$
- Input of output layer
  - $v_k^t = \sum_j w_{kj}^{(out)} z_j^t$
- Output of output layer
  - $y_k^t = f(v_k^t)$

where t is time



Detail of output of hidden layer

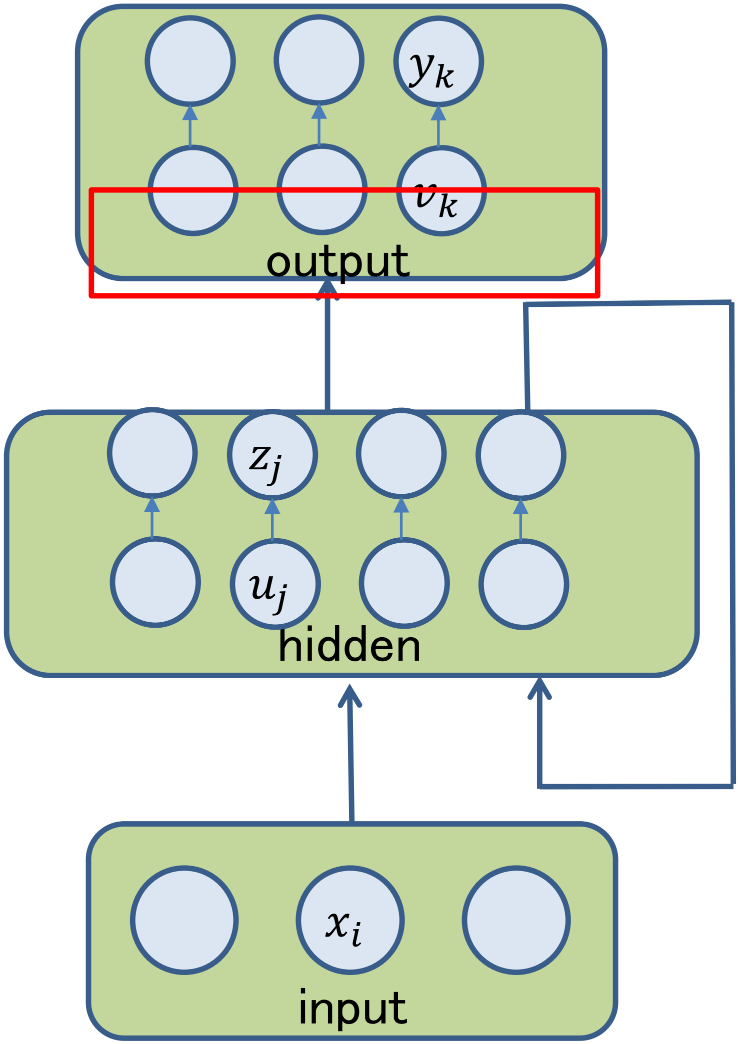


$$z_j^t = f(u_j^t)$$

Forward propagation of RNN

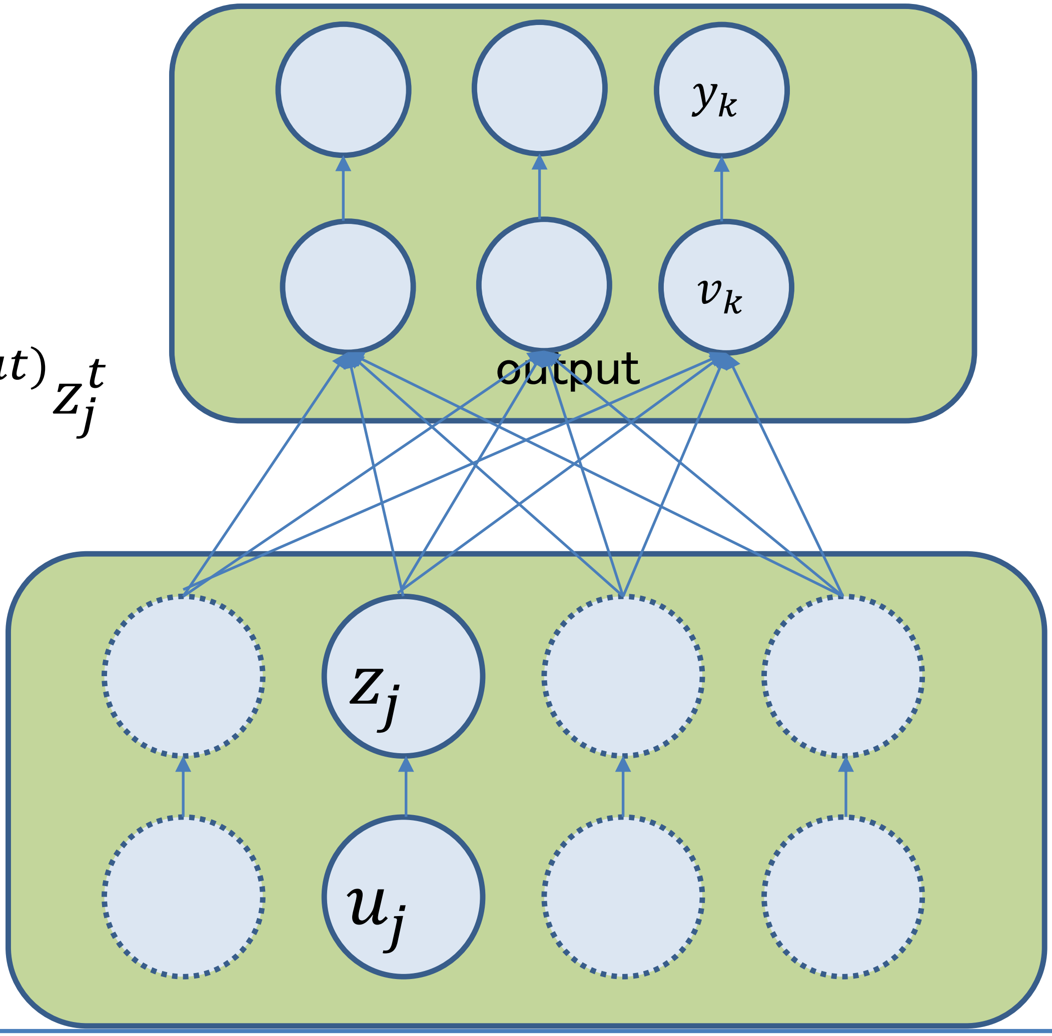
- Input of hidden layer
  - $u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$
- Output of hidden layer
  - $z_j^t = f(u_j^t)$
- Input of output layer
  - $v_k^t = \sum_j w_{kj}^{(out)} z_j^t$
- Output of output layer
  - $y_k^t = f(v_k^t)$

where t is time



Detail of input of output layer

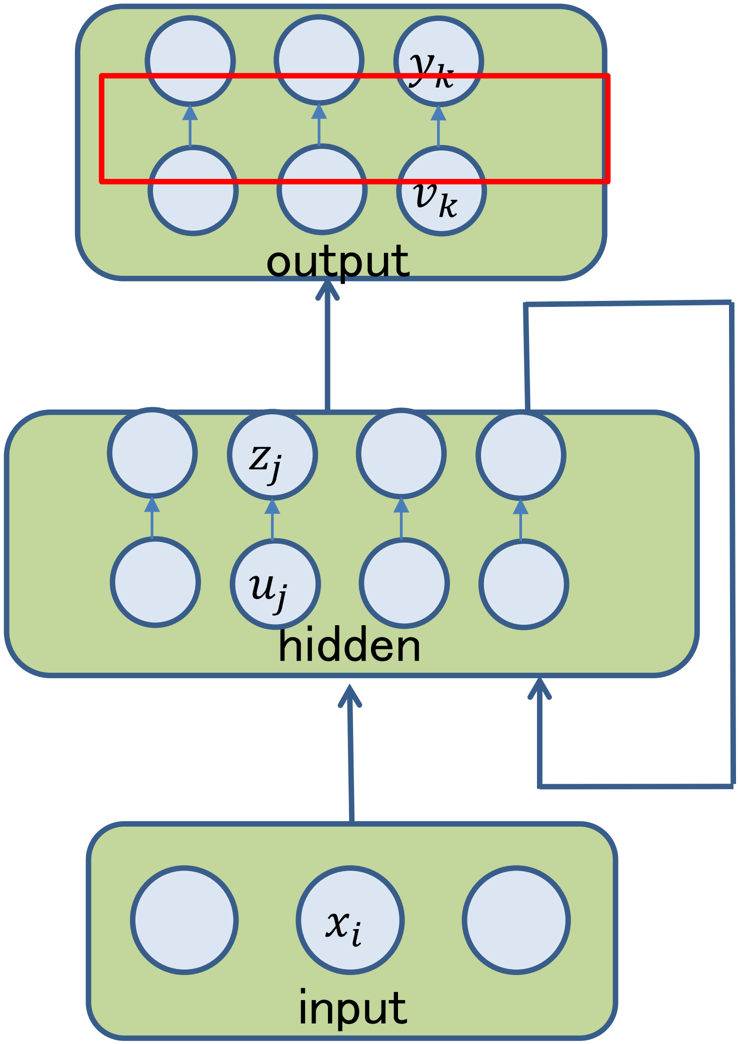
$$v_k^t = \sum_j w_{kj}^{(out)} z_j^t$$



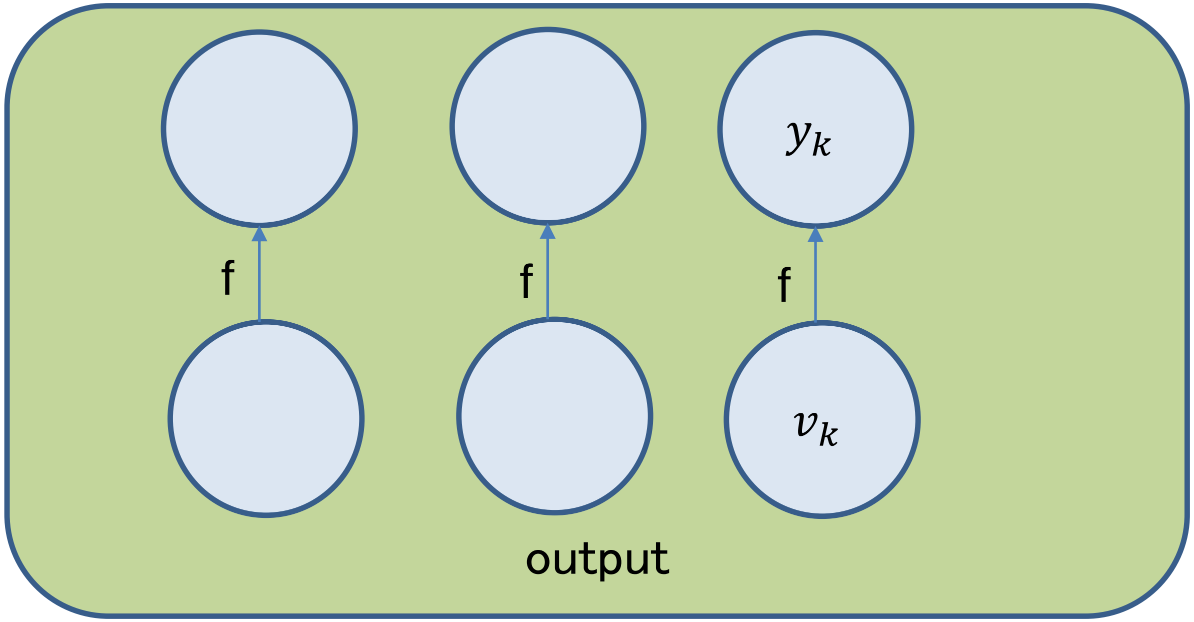
Forward propagation of RNN

- Input of hidden layer
  - $u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$
- Output of hidden layer
  - $z_j^t = f(u_j^t)$
- Input of output layer
  - $v_k^t = \sum_j w_{kj}^{(out)} z_j^t$
- Output of output layer
  - $y_k^t = f(v_k^t)$

where t is time



Detail of output on output layer

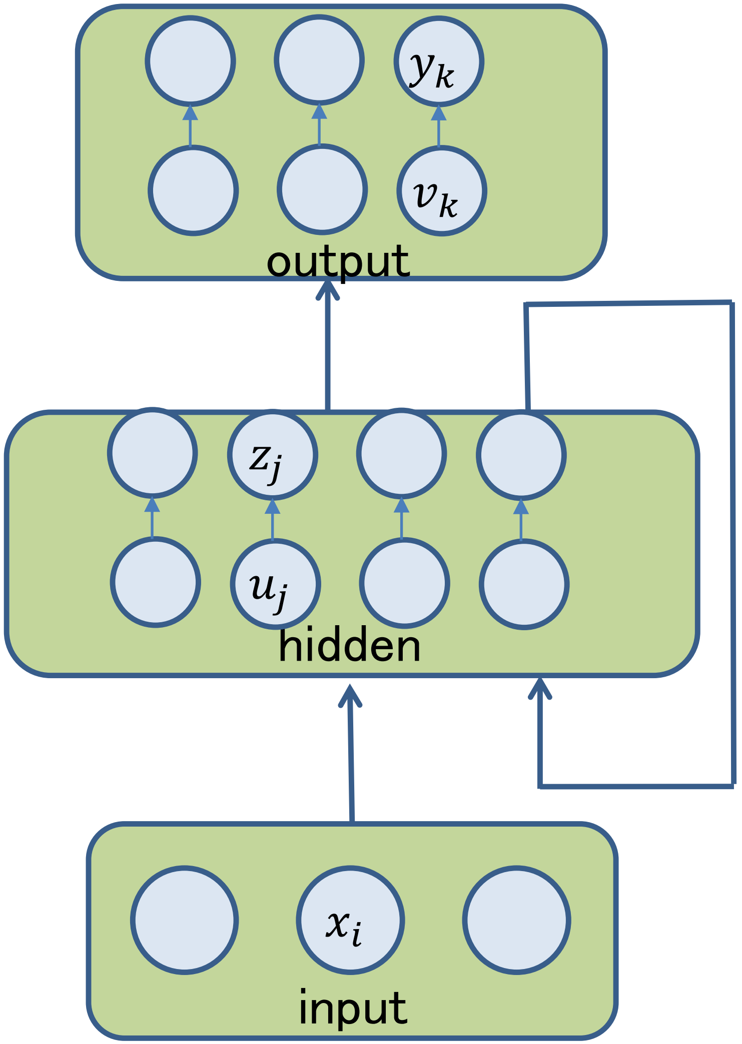


$$y_k^t = f(v_k^t)$$

Forward propagation of RNN

- Input of hidden layer
  - $u_j^t = \sum_i w_{ji}^{(in)} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$
- Output of hidden layer
  - $z_j^t = f(u_j^t)$
- Input of output layer
  - $v_k^t = \sum_j w_{kj}^{(out)} z_j^t$
- Output of output layer
  - $y_k^t = f(v_k^t)$

where t is time



Backpropagation of RNN

Backpropagation of RNN has 2 methods:

- Real time recurrent learning(RTRL)
- Backpropagation through time(BPTT)

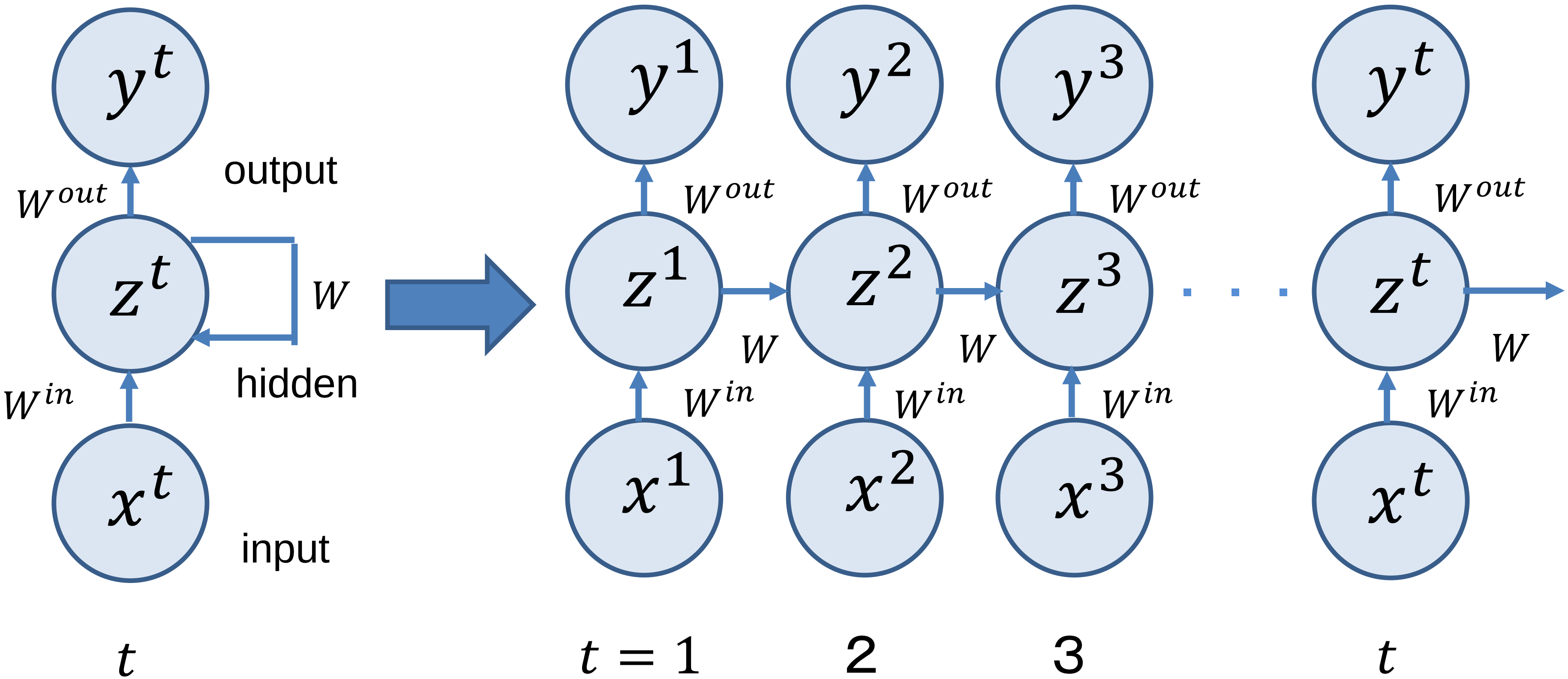
RTRL

- memory efficient
- possible to learn real-time
- limited learning capability

BPTT

- fast to calculate
- high performance of learning capability
- Limited learning cycle

Expand network through time direction



RNN network

Expanded network through time

BPTT

- Find δ of output layer.

$$\delta_j^{(l)} = \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)} f'\left(u_j^{(l)}\right)$$

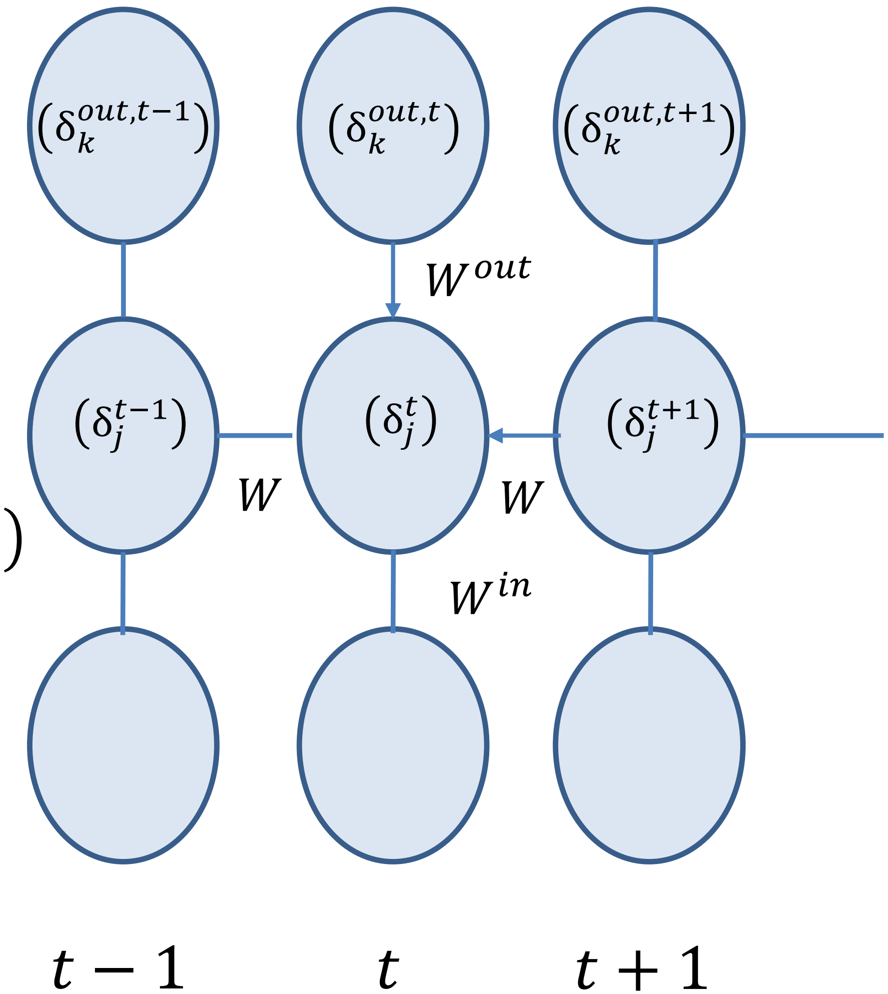
l: number of layers

- Find δ of hidden layer.

$$\delta_j^t = \left(\sum_k w_{kj}^{out} \delta_k^{out,t} + \sum_{j'} w_{j'j} \delta_{j'}^{t+1}\right) f'(u_j^t)$$

- Calculate gradient of error

$$\varDelta w_{ij}(t) = -\eta \sum_{\tau=t_0} \delta_i(r+1) y_j(t)$$



Problem on *RNN* learning

Length of the series data
determines the performance of RNN

The longer data length become,
the longer networks become

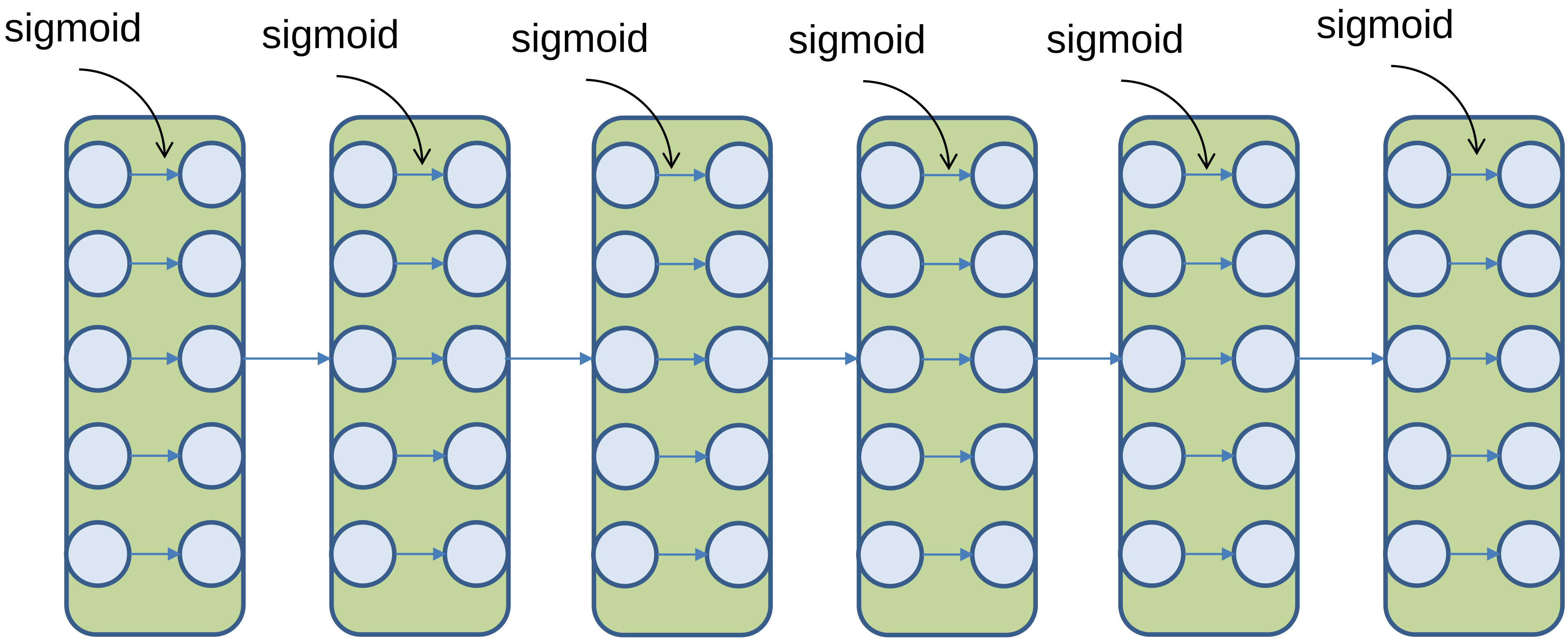
Numerous layers network causes
vanishing gradient problem

What is *vanishing gradient*?

- vanishing gradient
  - Assume Sigmoid for activation functions
  - The more layers, the slower learning speed
  - The reason of slowing is "*gradient vanishment*"
- exploding gradient
  - *Gradient explode*

Why gradient vanishes?

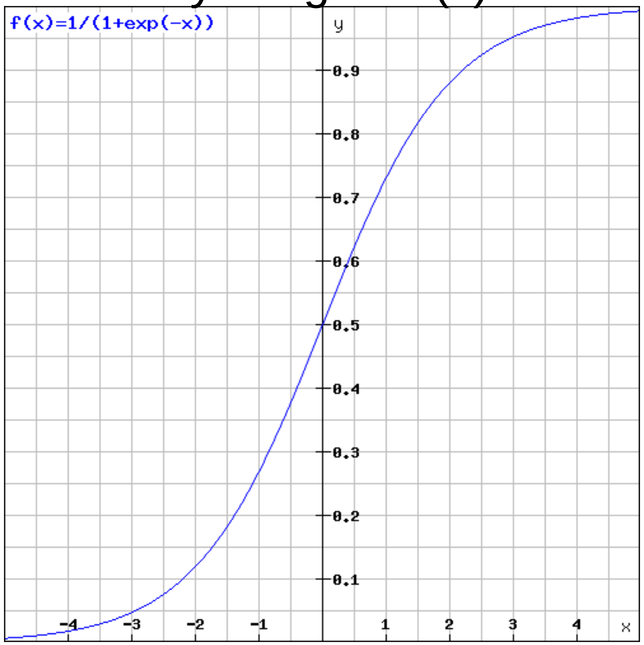
- The more layer, more number of sigmoid is applied



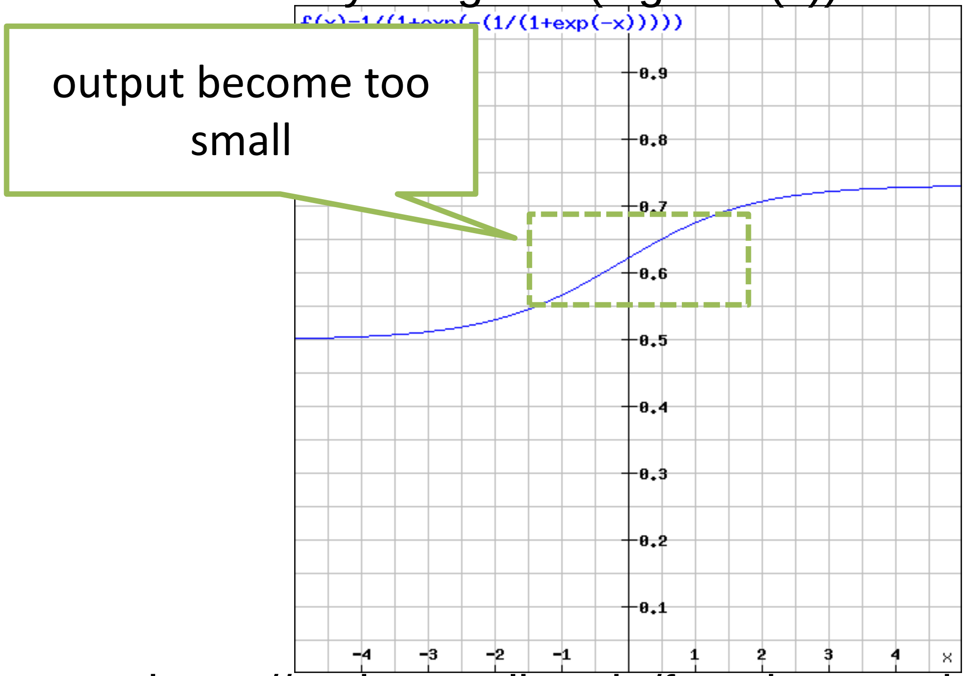
Why gradient vanishes?

- The more sigmoid is applied, the more its shape becomes flat -> gradient vanishes

y = sigmoid(x)



y = sigmoid(sigmoid(x))



https://rechneronline.de/function-graphs/

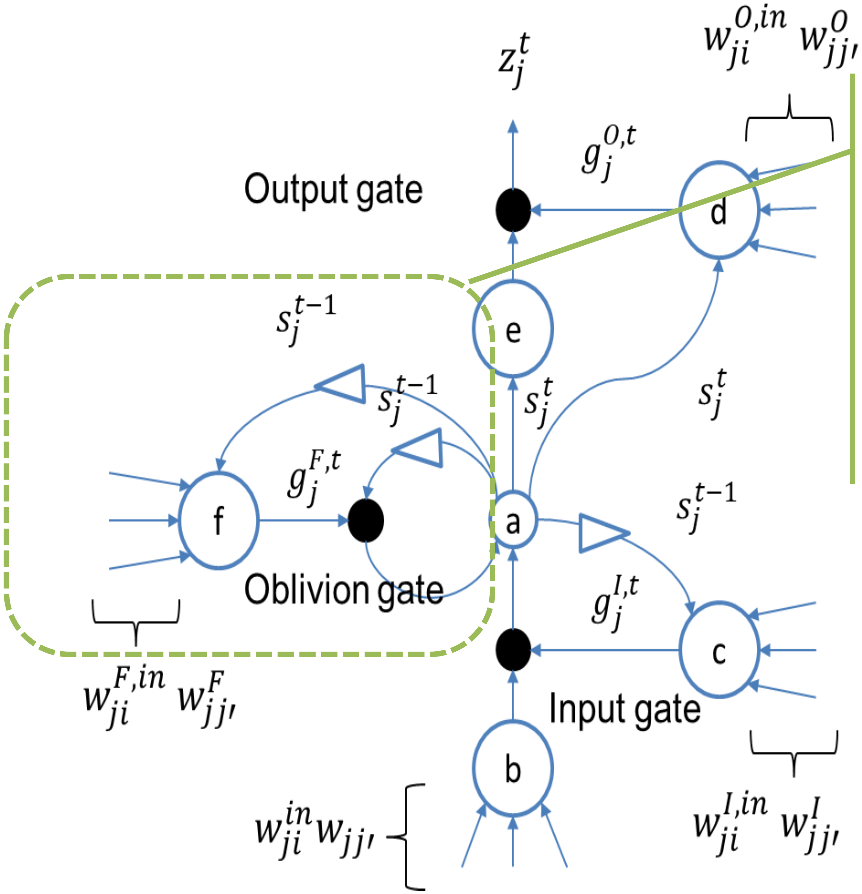
Long-short term memory(LSTM)

- *Deeper layer may cause vanishing gradient*
  - *RNN has deep network when it expand through time*
  - *It makes difficult to learn for long series data*
- *LSTM* is a solution for *vanishing gradient problem*
  - *memorize series data for long time*
- LSTM has only difference unit in hidden layer
  - Just replace the conventional unit with *memory-unit*

Oblivion gate

Output layer

Input layer



- *unit f outputs* $g_j^{F,t}$
- s_j^{t-1} is multiplied by $g_j^{F,t}$
- When the output is
  - Close to 0 > reset (oblivion)
  - Close to 1 > keep the state

$$g_j^{F,t} = f(u_j^{F,t})$$
$$= f\left(\sum_i w_{ji}^{F,in} x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1} + w_j^F s_j^{t-1}\right)$$

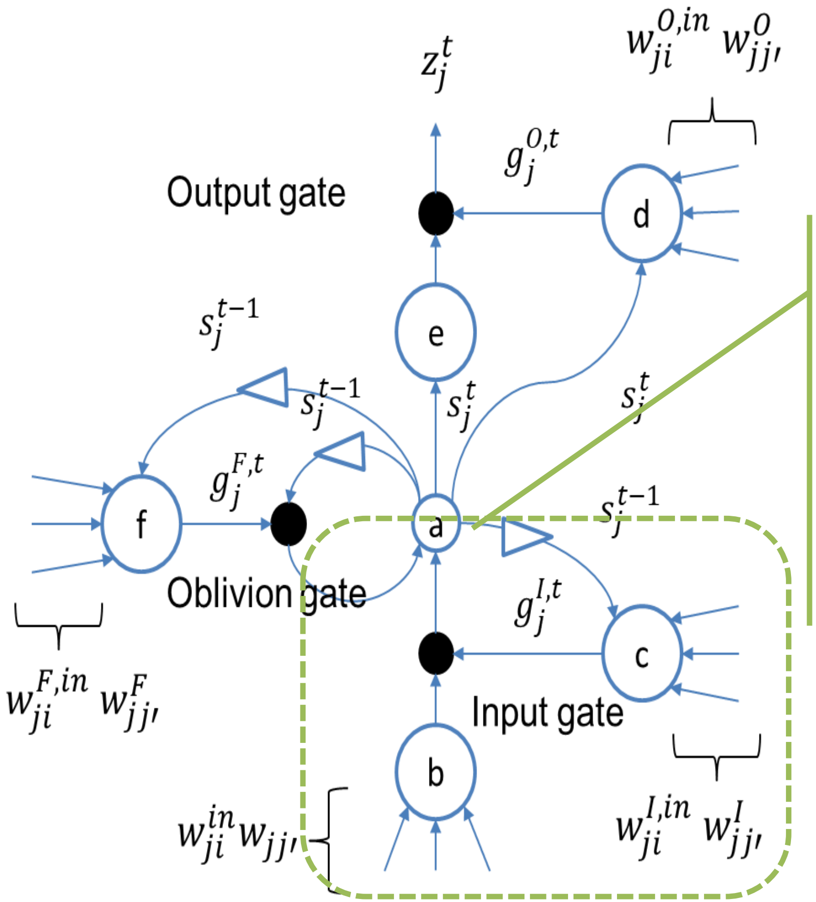
Input layer

hidden layer

Input gate

Output layer

Input layer



- Unit c outputs $g_j^{I,t}$
- unit b receives input from outside

and then multiplied by $g_j^{I,t}$

- The value is transmitted into memory cell

$$g_j^{I,t} = f(u_j^{I,t})$$
$$= f\left(\sum_i w_{ji}^{I,in} x_i^t + \sum_{j'} w_{jj'}^{I} z_{j'}^{t-1} + w_j^I s_j^{t-1}\right)$$

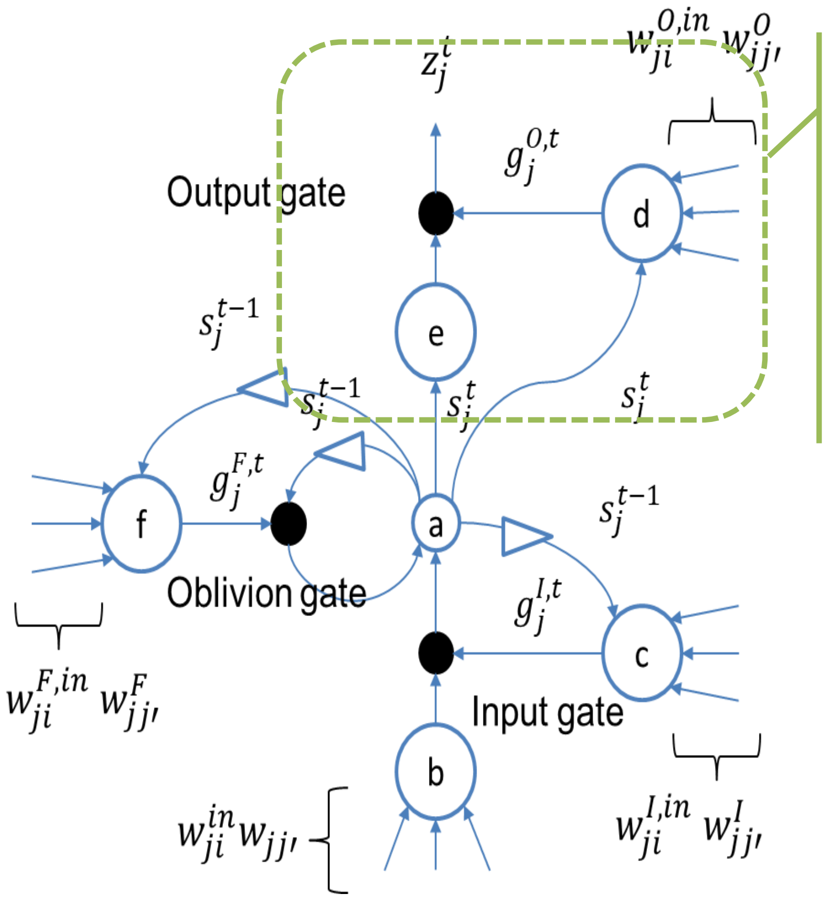
Input layer

hidden layer

Output gate

Output layer

Input layer



- Unit d outputs $g_j^{O,t}$
- When the $g_j^{O,t}$
  - Close to 1 > outputs transmitted into outsides
  - Close to 0 > block

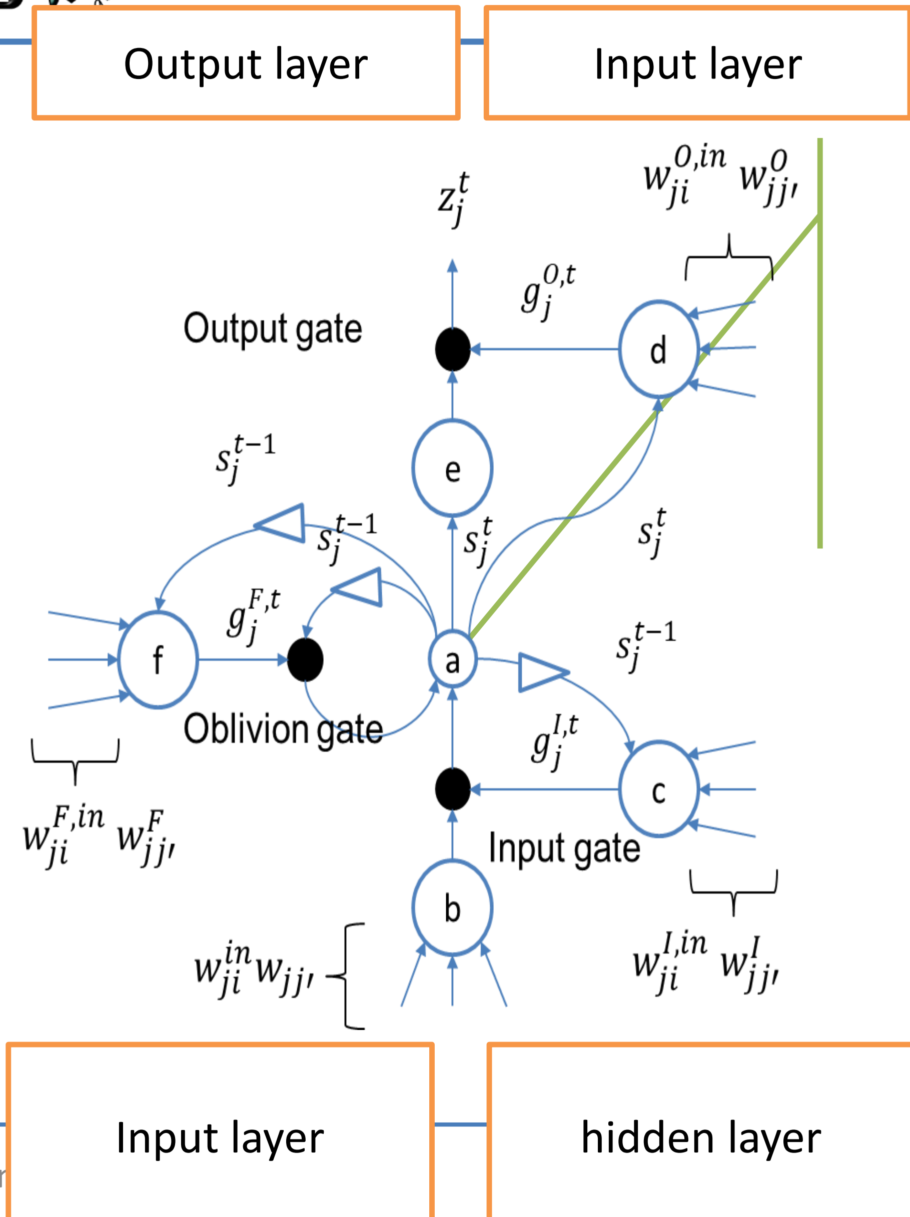
$$z_j^t = g_j^{O,t} f(s_j^t)$$

$$g_j^{O,t} = f(u_j^{O,t})$$
$$= f\left(\sum_i w_{ji}^{O,in} x_i^t + \sum_{j'} w_{jj'}^{O} z_{j'}^{t-1} + w_j^{O} s_j^t\right)$$

Input layer

hidden layer

Memory cell a



- Memory cell (a) contains the state
- Memorization is realized by make its output return 1 time after
- In this time, the value is multiplied by output of unit (f)

$$s_j^t = g_j^{F,t} s_j^{t-1} + g_j^{I,t} f(u_j^t)$$

$$u_j^t = \sum_i w_{ji}^t x_i^t + \sum_{j'} w_{jj'} z_{j'}^{t-1}$$

Weakness of RNN

- RNN is unsuitable in the case: length of sequence data differs between input and output
- Input-length corresponds with output-length
  - RNN has to output y_t from input $x_t$in each time *t*

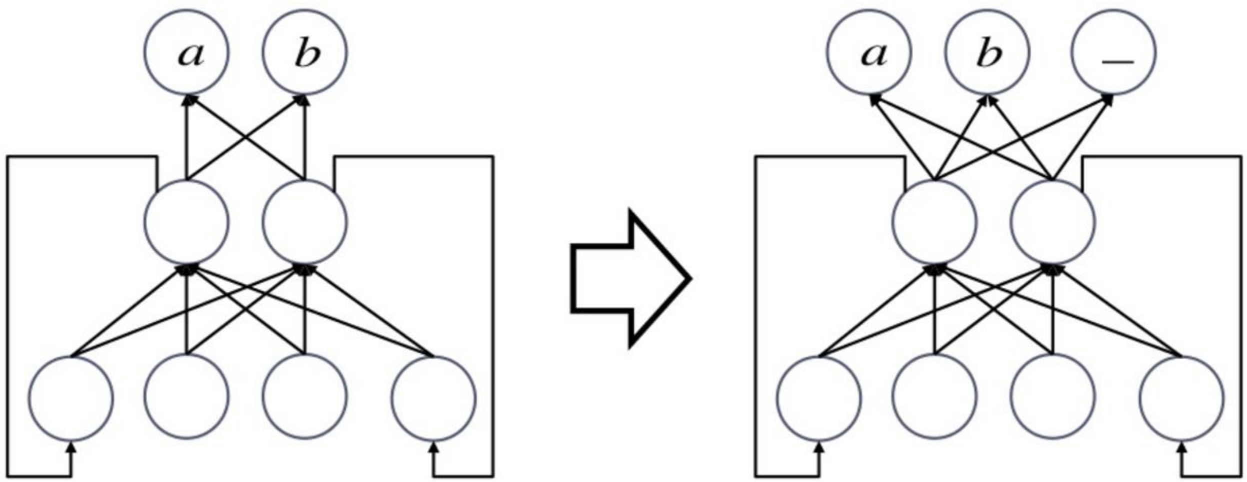
  Ex) input: "abac", ouput: "xxyy"   but true answer = "xy"
- Solution for this problem
  - Connectionist Temporal Classification

Connectionist temporal classification

- CTC can solve classification tasks

  How to solve?

- Add output unit with ‘_’ unit(vacant unit)
  - Expand the label by adding vacant value



Ex. phoneme estimation

- $L = \{a, b\}$ … *The set of labels to be recognized*
- $L' = \{a, b, \_\}$ … *Label:* L *with vacant label '_'*

There are countless numbers of redundant expression

ex. {a,_,_,b_,_} … {a,a,a,_,_,_,b} … {a,a,b,b,_,_,_,_} …

l*(Series data without redundancy) and* π*(with redundancy) have following "many-to-one" relation*

$$l = B(\pi)$$

l = B(a__b__) = B(aaa___b)

Ex. phoneme estimation

The case: input length is 6

Adjust the length of the data

– Assume vacant label _

– Complement label with sequential vacant or same label

- $l = \text{“}ab\text{”}$

True label

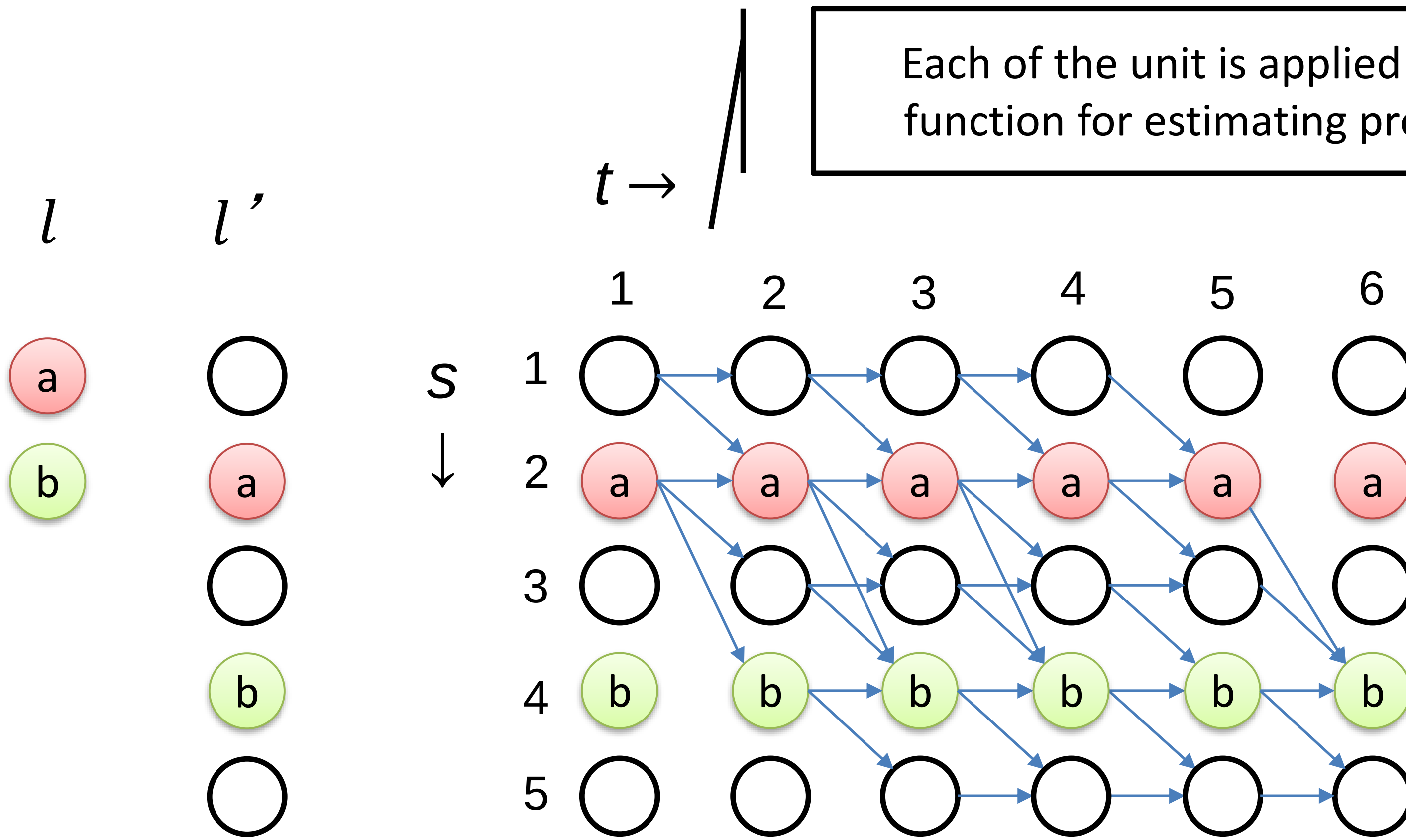
Length of input = 6

a,b,_,_,_,_
a,a,b,_,_,_
a,_,_,_,_,b
⋮
a,a,a,b,b,b

Combination

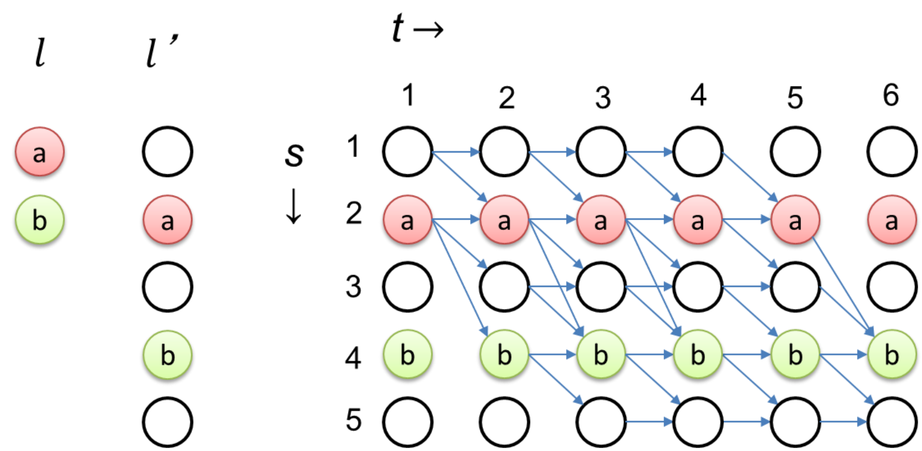
Combination of estimates π

Each of the unit is applied *softmax* function for estimating probability



$$B^{-1} = \{\pi | B(\pi) = l\}$$

Estimation



$$\mathrm{B}^{-1} = \{\pi | B(\pi) = l\}$$

$$p(l|X) = \sum_{\pi \in B^{-1}(l)} p(\pi|X)$$ sum of probability for total path

$$p(\pi|X) = \prod_{t=1}^{T} y_{\pi t}^{t}$$ Probability that the path π is true (input is X)

$p(l|X)$ computation

- $p_l(ab|\,X) = \begin{array}{c} p_l(a, b, \_, \_, \_, \_|X) \\ + \\ p_l(\text{a,a,b,\_,\_,\_}\,|X) \\ + \\ p_l(\text{a,\_,\_,\_,\_,b}\,|X) \\ \vdots \\ p_l(\text{a,a,a,b,b,b}\,|X) \end{array} = \begin{array}{c} y_a^1 \cdot y_b^2 \cdot y_\_^3 \cdot y_\_^4 \cdot y_\_^5 \cdot y_\_^6 \\ + \\ y_a^1 \cdot y_a^2 \cdot y_b^3 \cdot y_\_^4 \cdot y_\_^5 \cdot y_\_^6 \\ + \\ y_a^1 \cdot y_\_^2 \cdot y_\_^3 \cdot y_\_^4 \cdot y_\_^5 \cdot y_b^6 \\ \vdots \\ y_a^1 \cdot y_a^2 \cdot y_a^3 \cdot y_b^4 \cdot y_b^5 \cdot y_b^6 \end{array}$

$p(\pi|X)$: each line of probability

Highest value is estimated as correct l

$p(l|X)$ computation

- $p(l|X) = \sum_{\pi \in B^{-1}(l)} p(\pi|X)\,,\ ex.\, l = ab$

$p_l(a, b, \_, \_, \_, \_|X)$

$y_a^1 \cdot y_b^2 \cdot y_\_^3 \cdot y_\_^4 \cdot y_\_^5 \cdot y_\_^6$

+

$p_l(ab|\, X)\ needs\ huge\ amount\ of\ computation!$

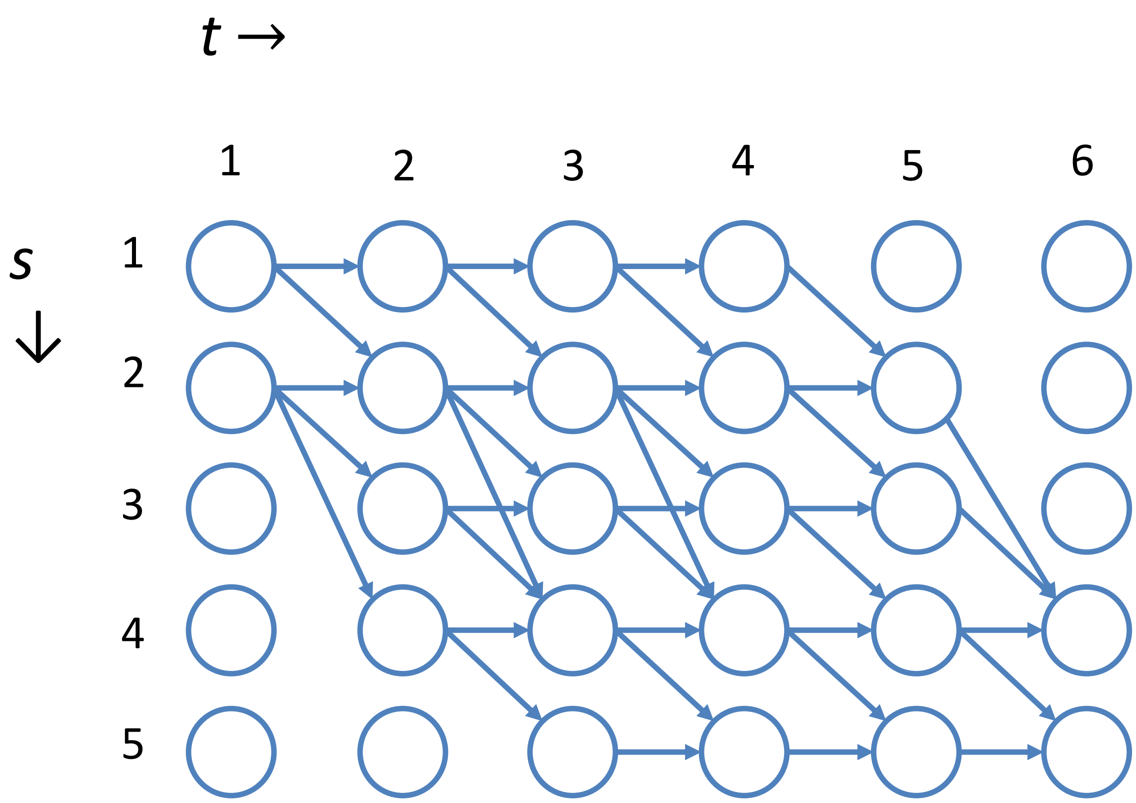
- Highest value is the estimation of correct l

Forward backward method

- Assume the set of path in time=*t* and s^{th} label
  - Former part of path to *t* $\pi_{1:t} = (\pi_1, \dots, \pi_t)$
  - Latter part of path from *t* $\pi_{t:T} = (\pi_t, \dots, \pi_T)$

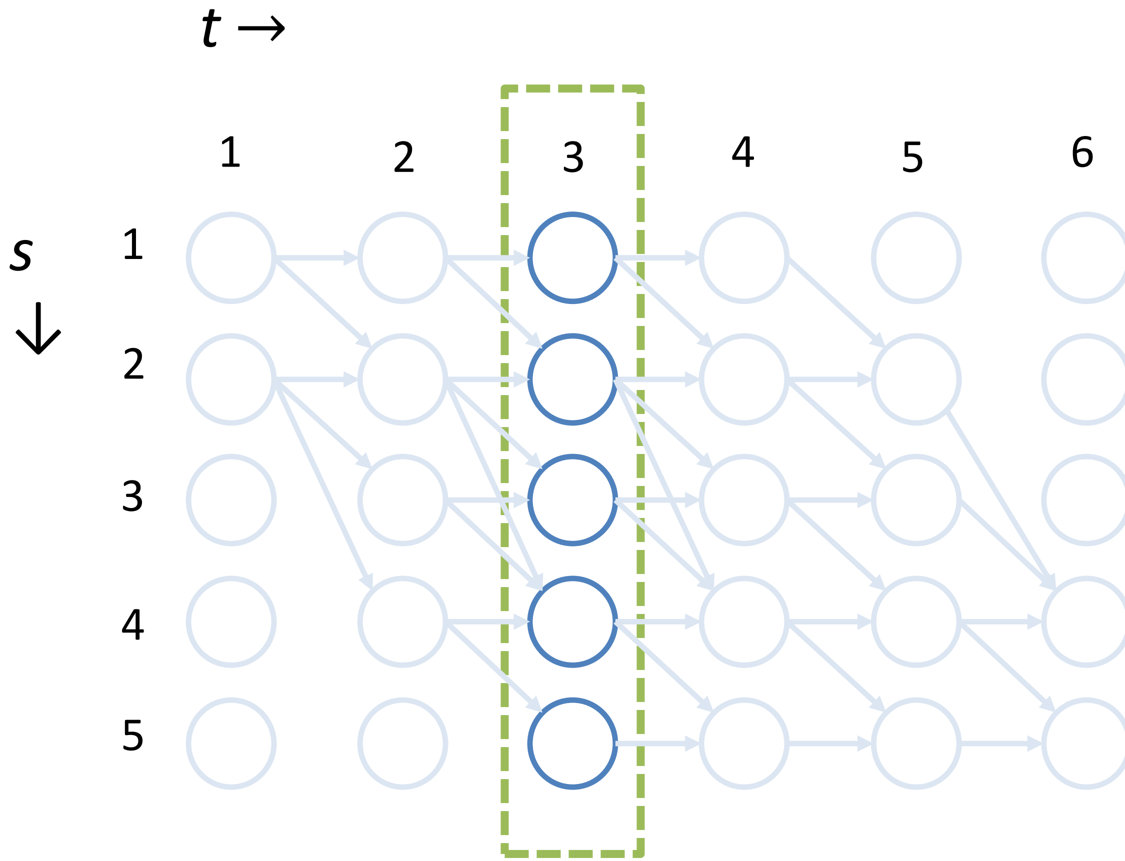
- *Sum of probabilities*
  - *Former part of path* $p(\pi_{1:t})$ $\alpha_{s,t}$
  - *Latter part of path* $p(\pi_{t:T})$ $\beta_{s,t}$
- α *and* β *could be calculated recurrently*

To calculate $p(l|X)$

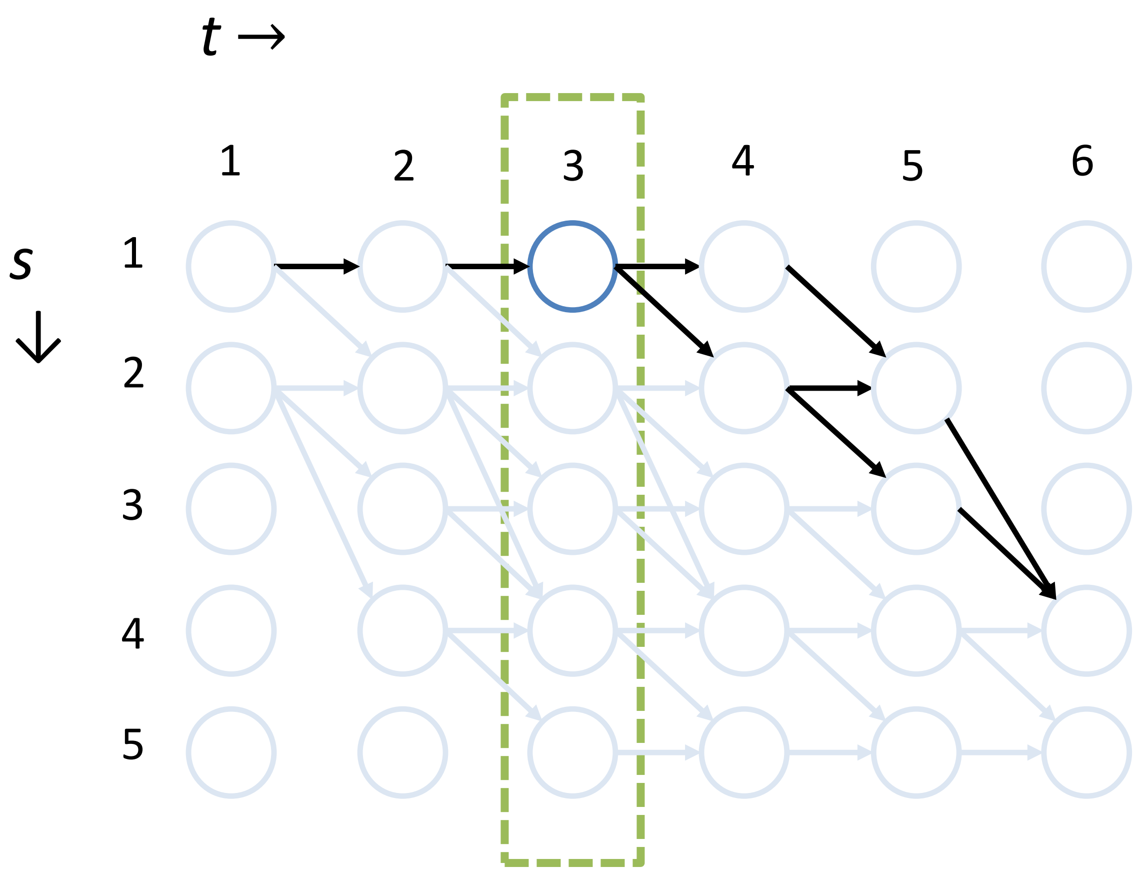


$p(l|X) =$ sum of total path probability

Focus on the certain time

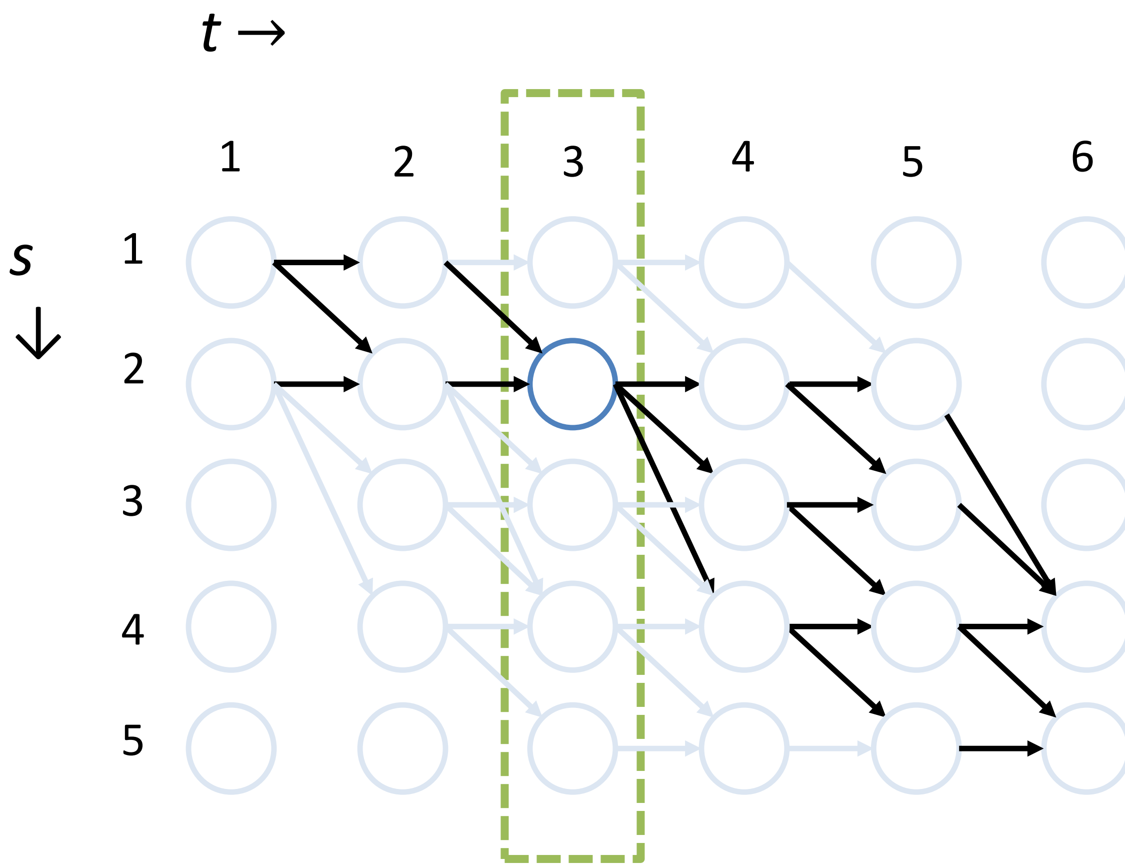


Focus on the certain time



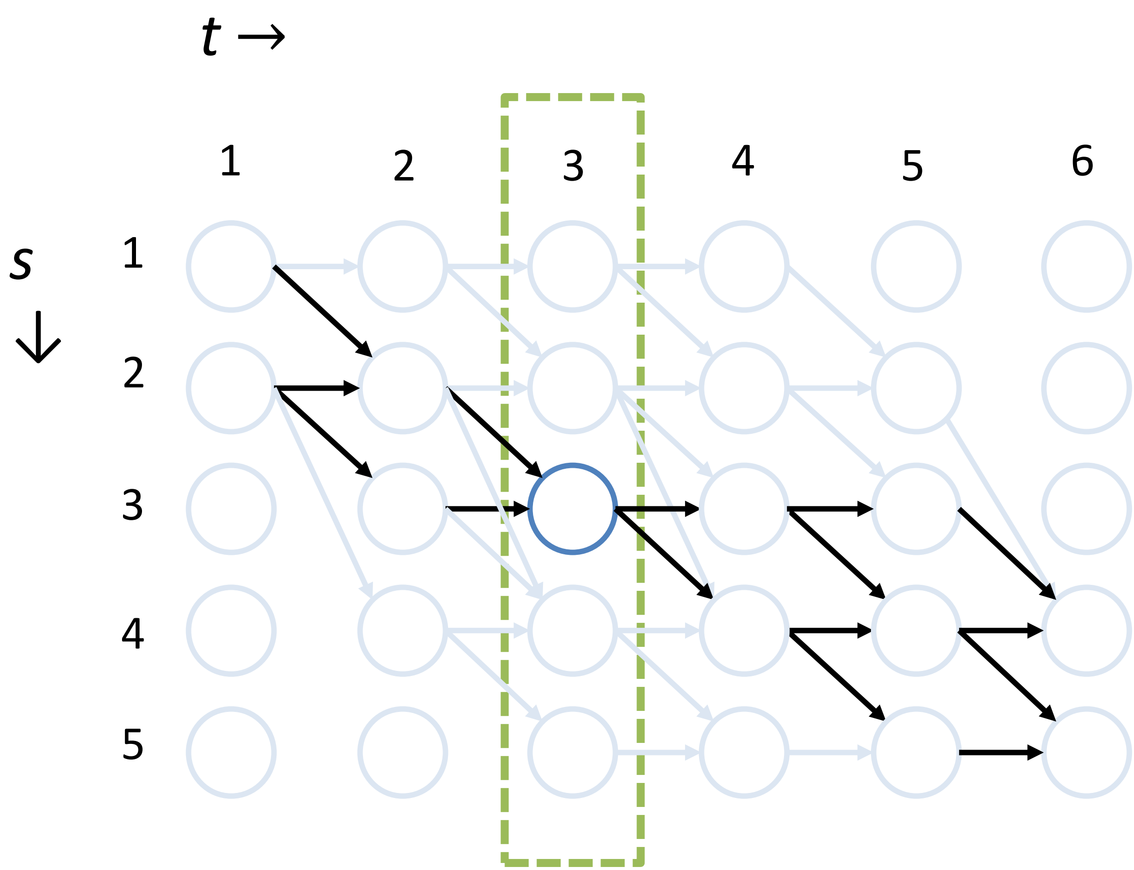
path a

Focus on the certain time



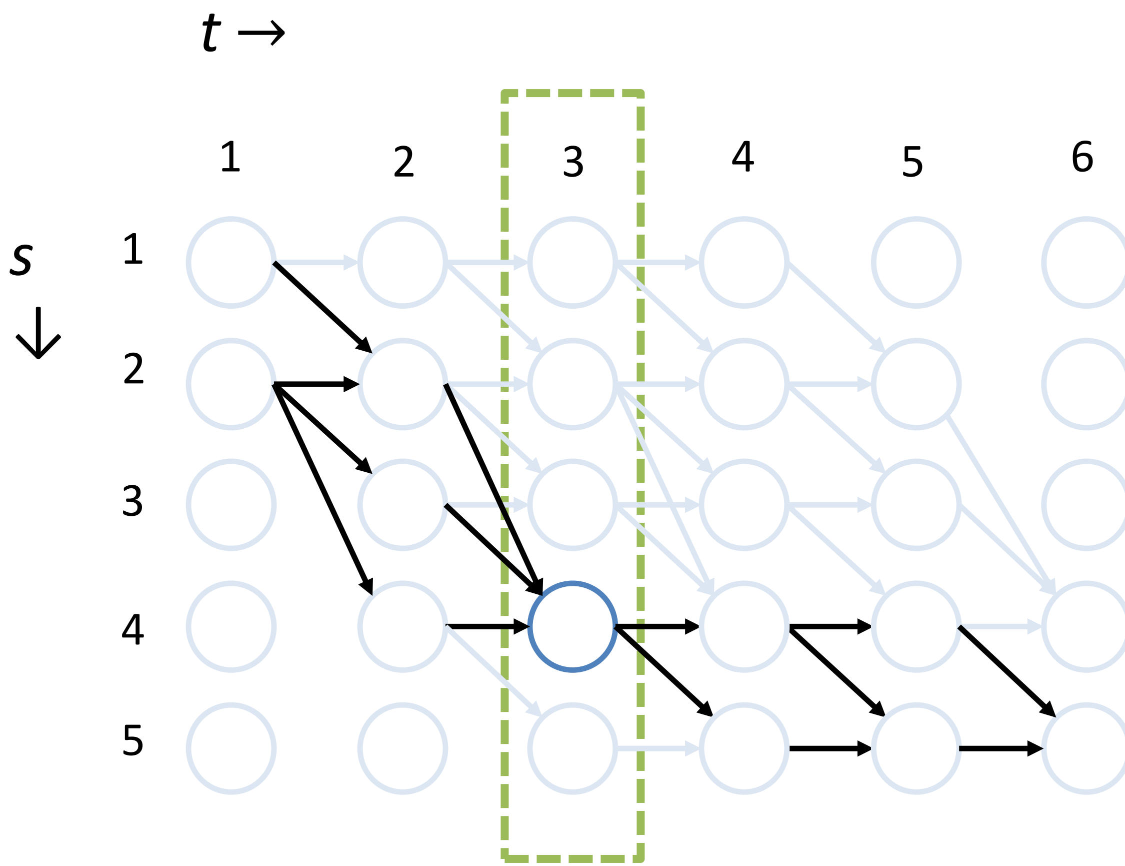
path b

Focus on the certain time



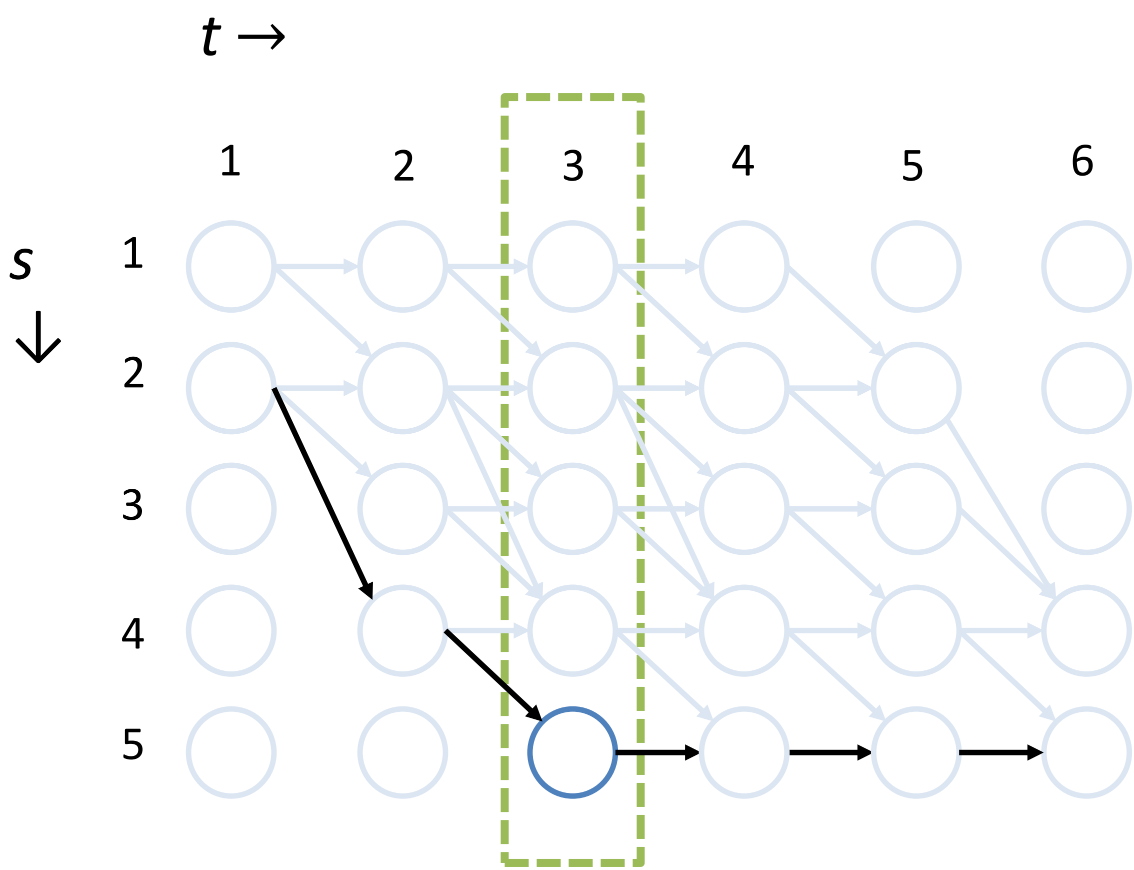
path c

Focus on the certain time



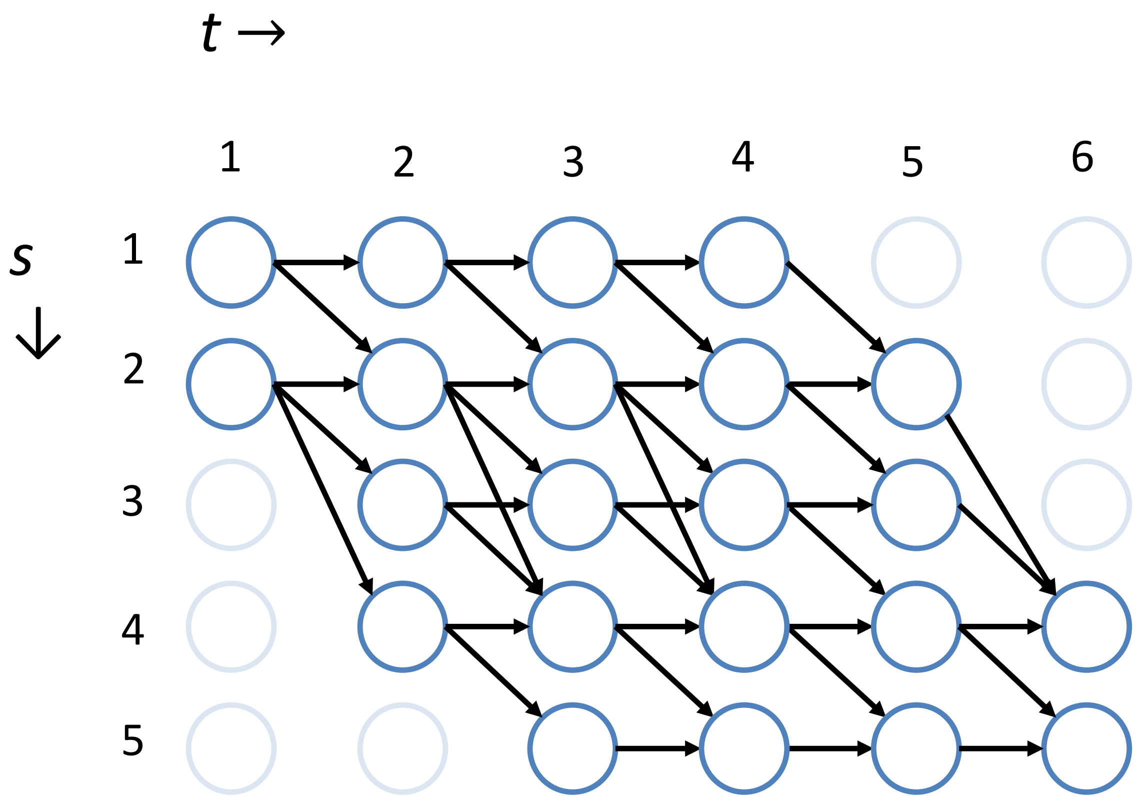
path d

Focus on the certain time



path e

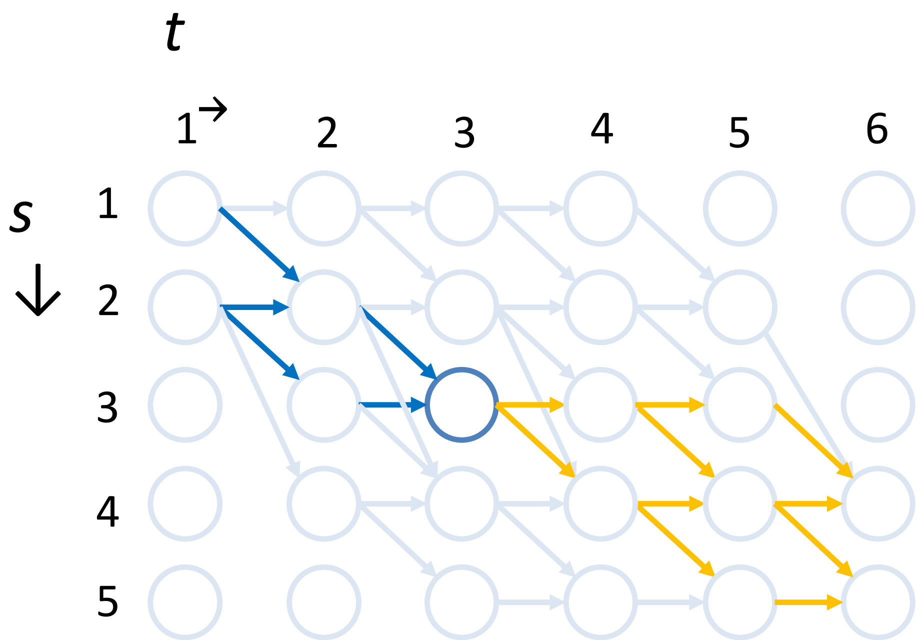
Forward backward method



$$p(l|X) = \text{path a + path b + … path d}$$

Forward backward method

Sum of former part of path: $\alpha_{s,t}$

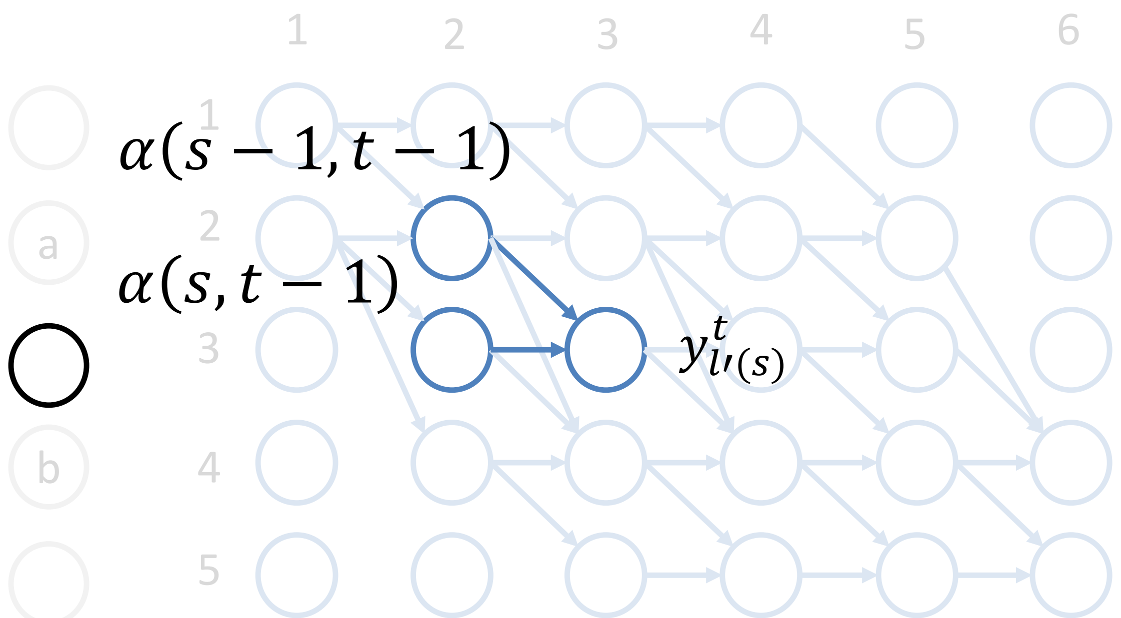


Latter part of path $p(\pi_{t:T})$: $\beta_{s,t}$

$$p(l|X) = \sum_{s} \alpha_{s,t} + \beta_{s,t}$$

Why forward backward is fast?

$\alpha_{s,t}$ is calculated by recurrence relation

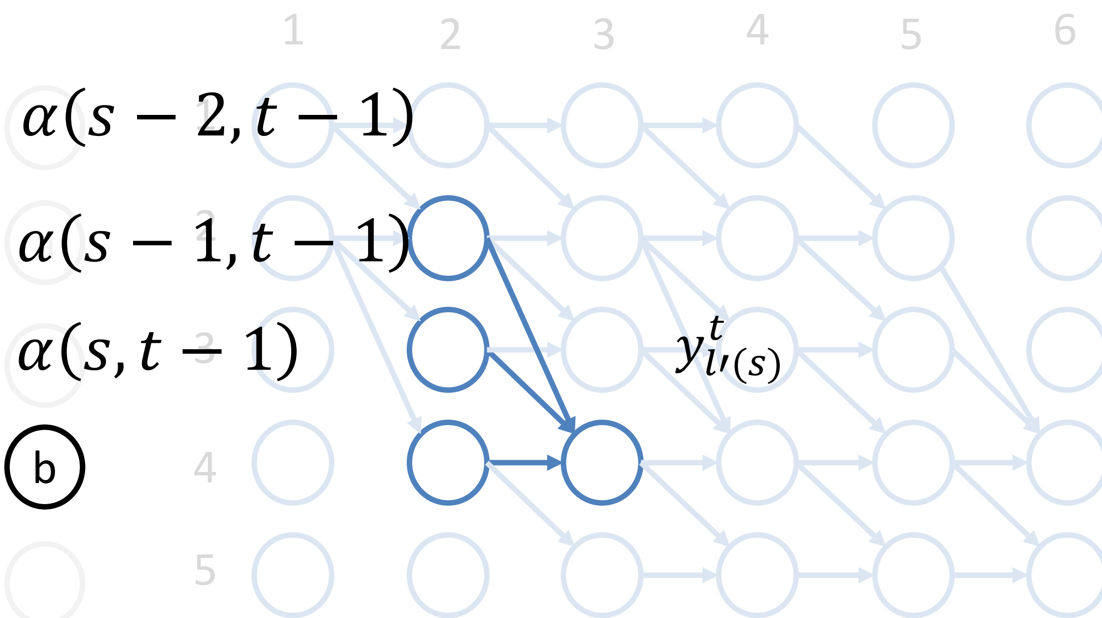


$$\alpha(s,t) = \; y^t_{l'(s)}\alpha(s,t) + y^t_{l'(s)}\alpha(s,t)$$

(pattern: s = '_')

Why forward backward is fast?

$\alpha_{s,t}$ is calculated by recurrence relation

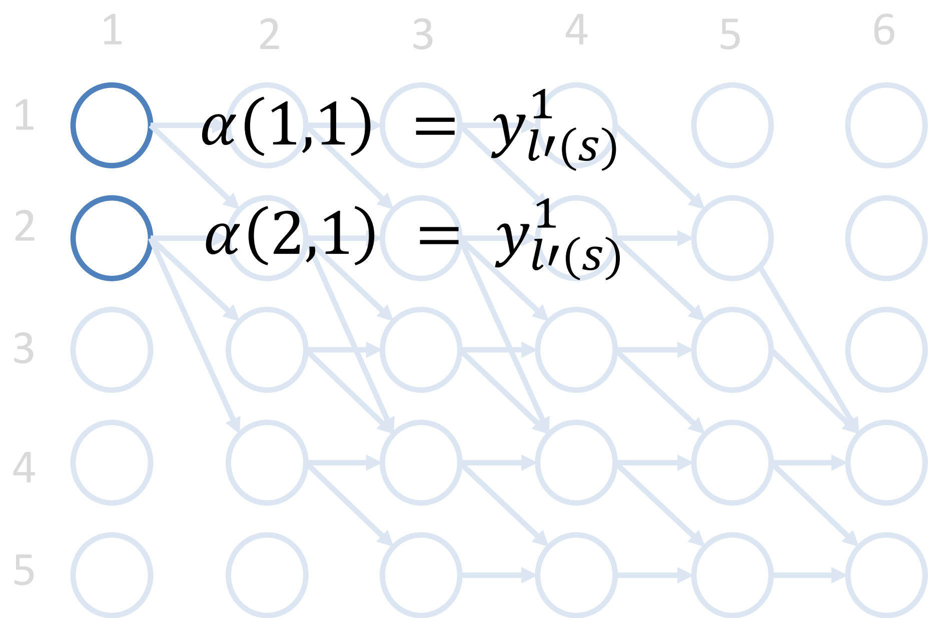


$$\alpha(s,t) = \ y^t_{l'(s)}\alpha(s-2,t-1) + \ y^t_{l'(s)}\alpha(s-1,t-1) + \ y^t_{l'(s)}\alpha(s,t-1)$$

(pattern: s != ‘_’)

Why forward backward is fast?

t = 1



$$\alpha(1,1) = y^1_{l'(s)}$$

$$\alpha(2,1) = y^1_{l'(s)}$$

Accelerate the calculation of α with dynamic programming (as well as β)

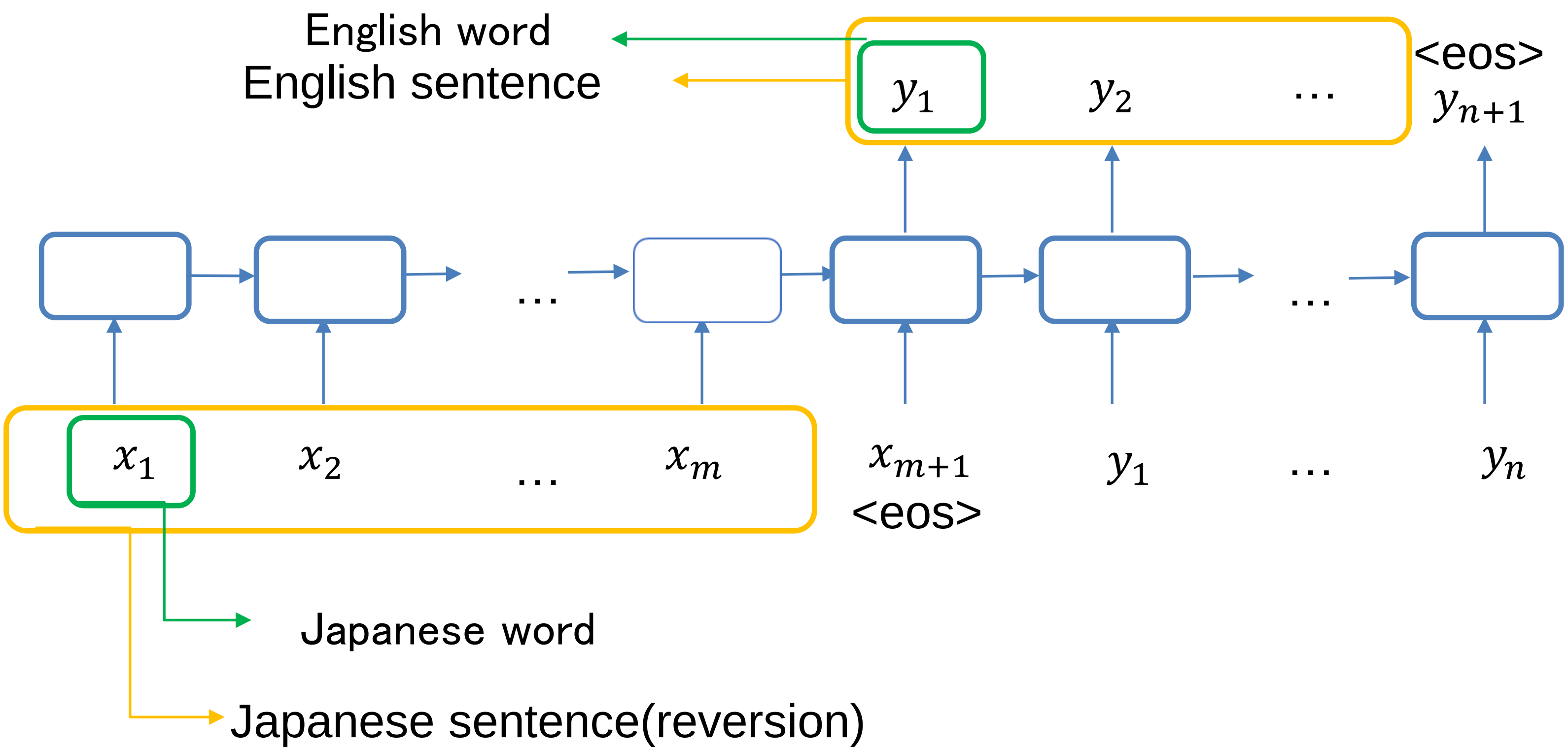
Summary

- RNN can memorize past state
- Length of series data determine performance
- Learning is conducted by expanding network through time-direction
- LSTM is solution for longer term memorization
- CTC can estimates likelihood label
  - Forward and backward method is efficient

Demo

- Machine translation
  - Japanese to English
- Encoder decoder model

Encoder–Decoder model



Tips:input sequence of sentence is reversion for better result

Bilingual data

jp.txt

誰 が 一番 に 着く か 私 に は 分かり ませ ん 。
十 中八九 彼 は 成功 する だろ う 。
あなた の 銀行 口座 を 教え て いただけ ます か 。

eng.txt

i can 't tell who will arrive first .
ten to one , he will succeed .
may we know your bank account ?

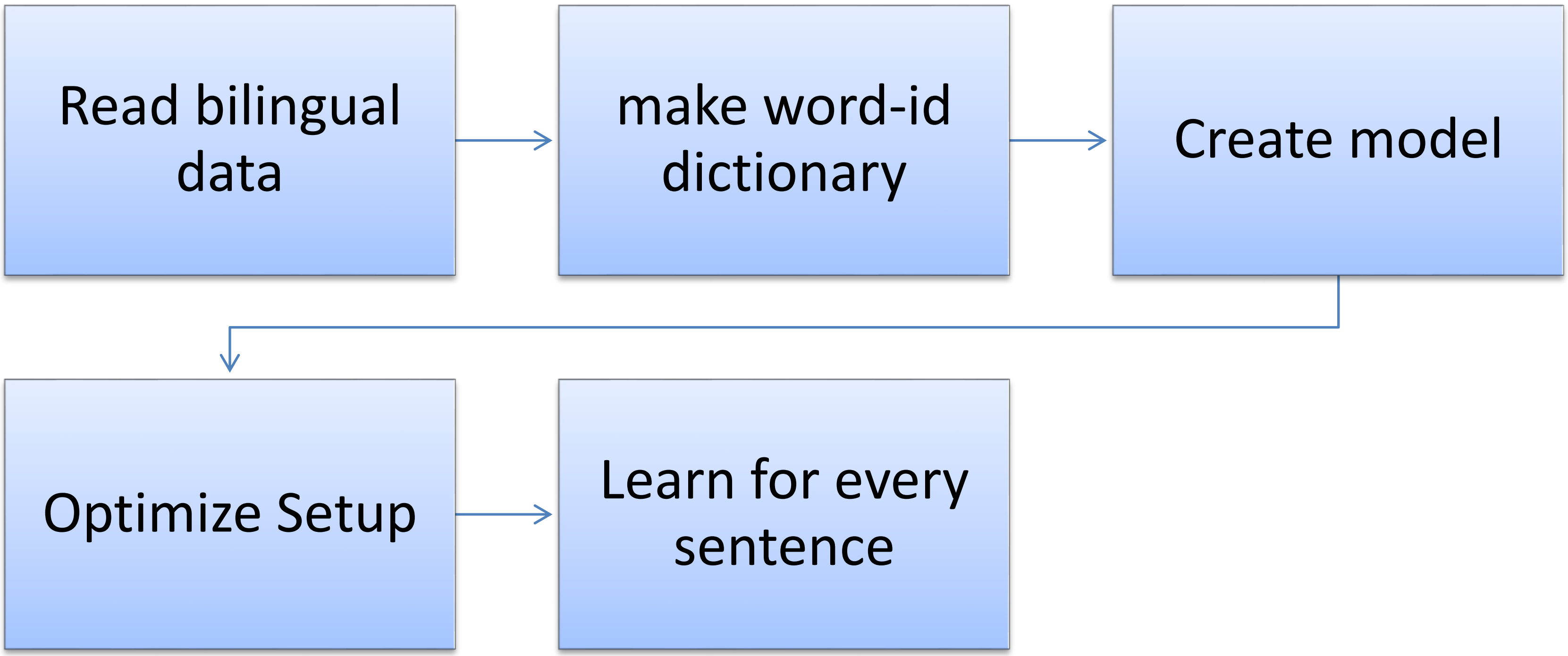
We use 10000 sentences for training

Word is separated by space

Encoder–Decoder translation model

TRAINING

Implementing Encoder-Decoder model



Read data and make dictionary

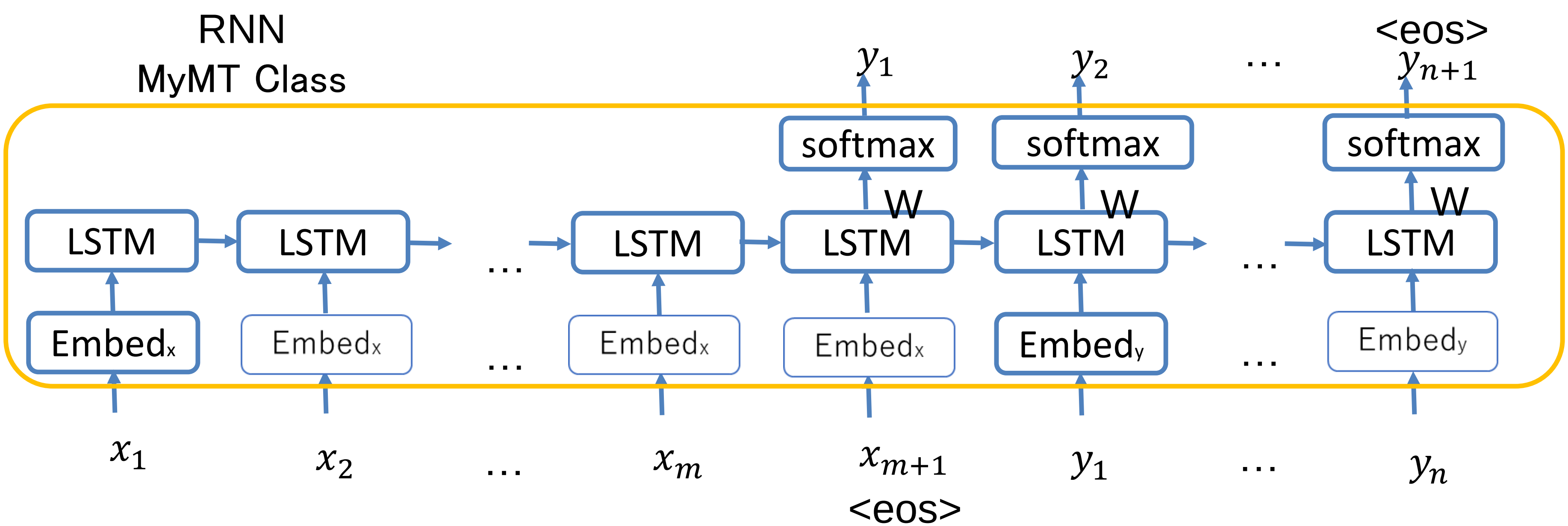
jp.txt

```
jvocab = {}
jlines = open('jp.txt').read().split('¥n')
for i in range(len(jlines)):
    lt = jlines[i].split()
    for w in lt:
        if w not in jvocab:
            jvocab[w] = len(jvocab)

jvocab['<eos>'] = len(jvocab)
jv = len(jvocab)
```

eng.txt

```
evocab = {}
    elines =
    open('eng.txt').read().split('¥n')
    for i in range(len(elines)):
        lt = elines[i].split()
        for w in lt:
            if w not in evocab:
                evocab[w] = len(evocab)

    evocab['<eos>'] = len(evocab)
    ev = len(evocab)
```

Encoder-Decoder translation model graph



MyMt Class – inithialized–

```
class MyMT(chainer.Chain):
    def __init__(self, jv, ev, k):
        super(MyMT, self).__init__(
            embedx = L.EmbedID(jv, k),
            embedy = L.EmbedID(ev, k),
            H = L.LSTM(k, k),
            W = L.Linear(k, ev),
```

MyMT Class － Forward －

```
def __call__(self, jline, eline):
        for i in range(len(jline)):
          wid = jvocab[jline[i]]
          x_k = self.embedx(Variable(np.array([wid], dtype=np.int32)))
          h = self.H(x_k)

        x_k = self.embedx(Variable(np.array([jvocab['<eos>']], dtype=np.int32)))
        tx = Variable(np.array([evocab[eline[0]]], dtype=np.int32))
        h = self.H(x_k)
        accum_loss = F.softmax_cross_entropy(self.W(h), tx)

        for i in range(len(eline)):
          wid = evocab[eline[i]]
          x_k = self.embedy(Variable(np.array([wid], dtype=np.int32)))
          next_wid = evocab['<eos>']  if (i == len(eline) - 1) else evocab[eline[i+1]]
          tx = Variable(np.array([next_wid], dtype=np.int32))
          h = self.H(x_k)
          loss = F.softmax_cross_entropy(self.W(h), tx)
          accum_loss += loss
        return accum_loss
```

Create model and Setup optimizer

```
demb = 100
model = MyMT(jv, ev, demb)
optimizer = optimizers.Adam()
optimizer.setup(model)
```

Learning

```
for epoch in range(100):
    for i in range(len(jlines)-1):
        jln = jlines[i].split()
        jlnr = jln[::-1]
        eln = elines[i].split()
        model.H.reset_state()
        model.zerograds()
        loss = model(jlnr, eln)
        loss.backward()
        loss.unchain_backward()  # truncate
        optimizer.update()
        print i, " finished"
    outfile = "mt-" + str(epoch) + ".model"
    serializers.save_npz(outfile, model)
```

Encoder-Decoder model

TESTING

Test Data

- Test data(100) + Training data(2000) = 2100 sentences

Uh...
Some sentences include unknown word.
Skip it.

Result

- Accuracy is 0.077

because

- The number of training data is too small
- The low frequency word is forgotten.

References

References

- 岡谷貴之. (2015)　「深層学習」

Demo

-「ChainerとRNNと機械翻訳」
 <http://qiita.com/odashi_t/items/a1be7c4964fbea6a116e>

Thank you for your listening!