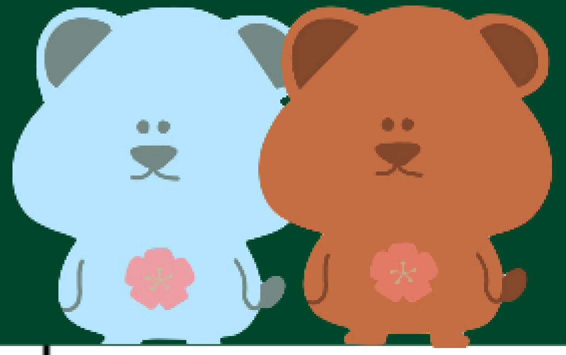
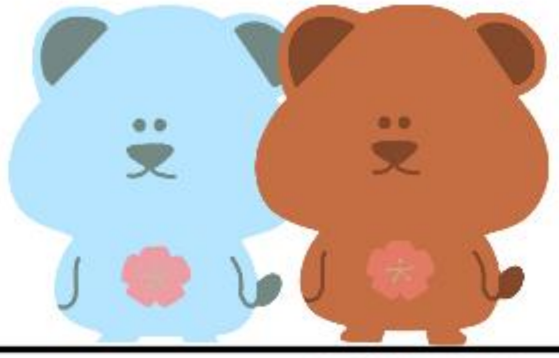


Recurrent Neural Networks

민소연, 최지우

EWHA EURON

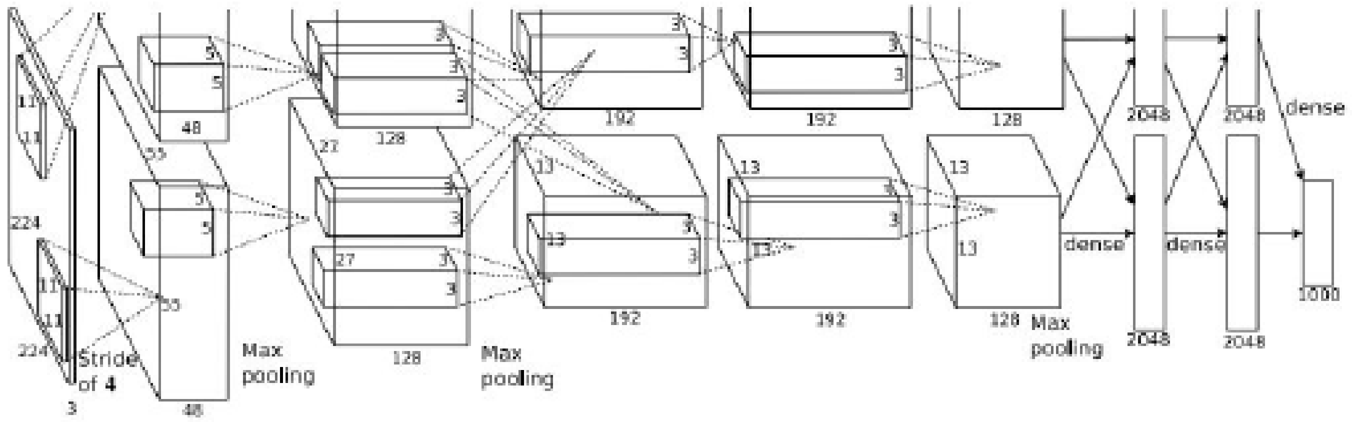
Index

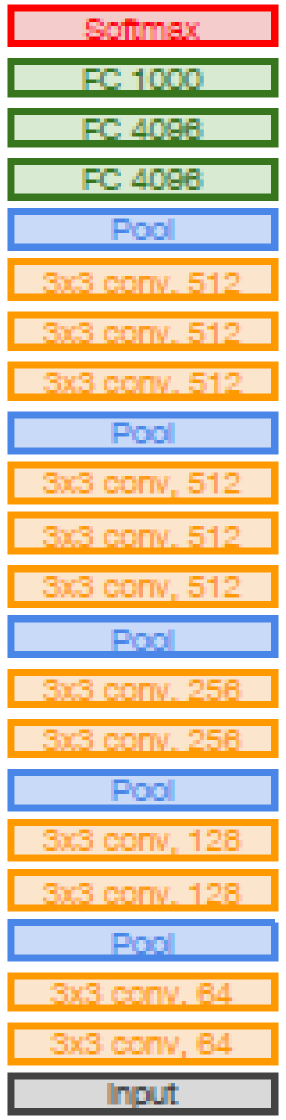


review

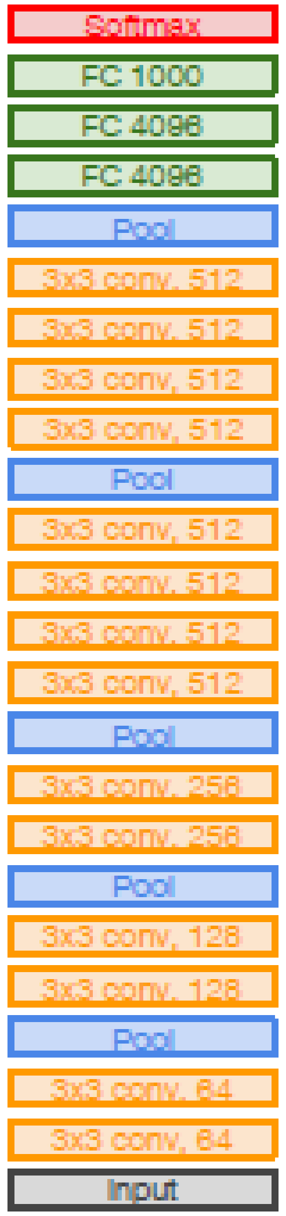
#01 CNN Architectures

AlexNet



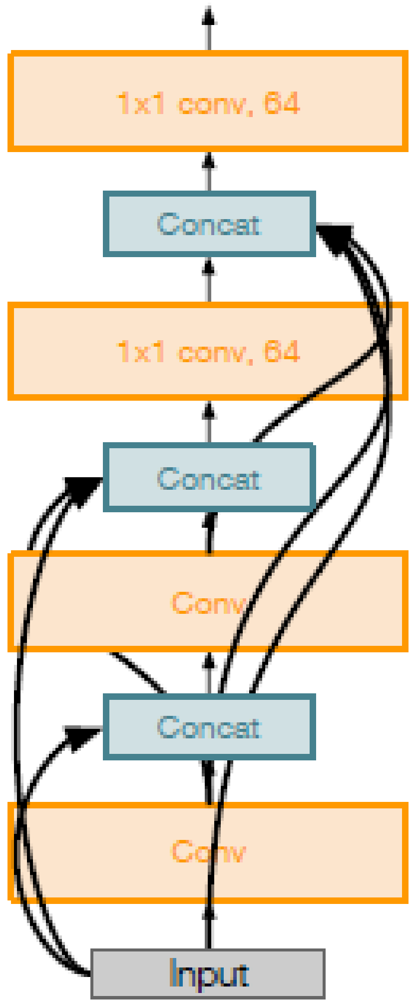


VGG16

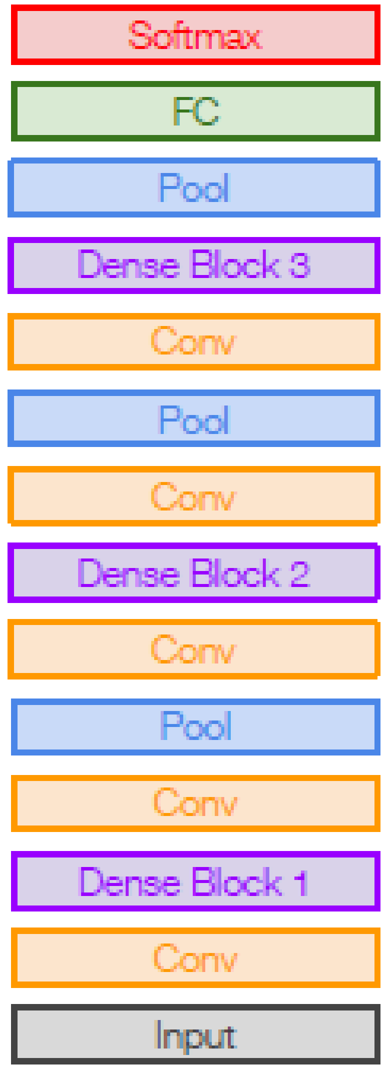


VGG19

DenseNet

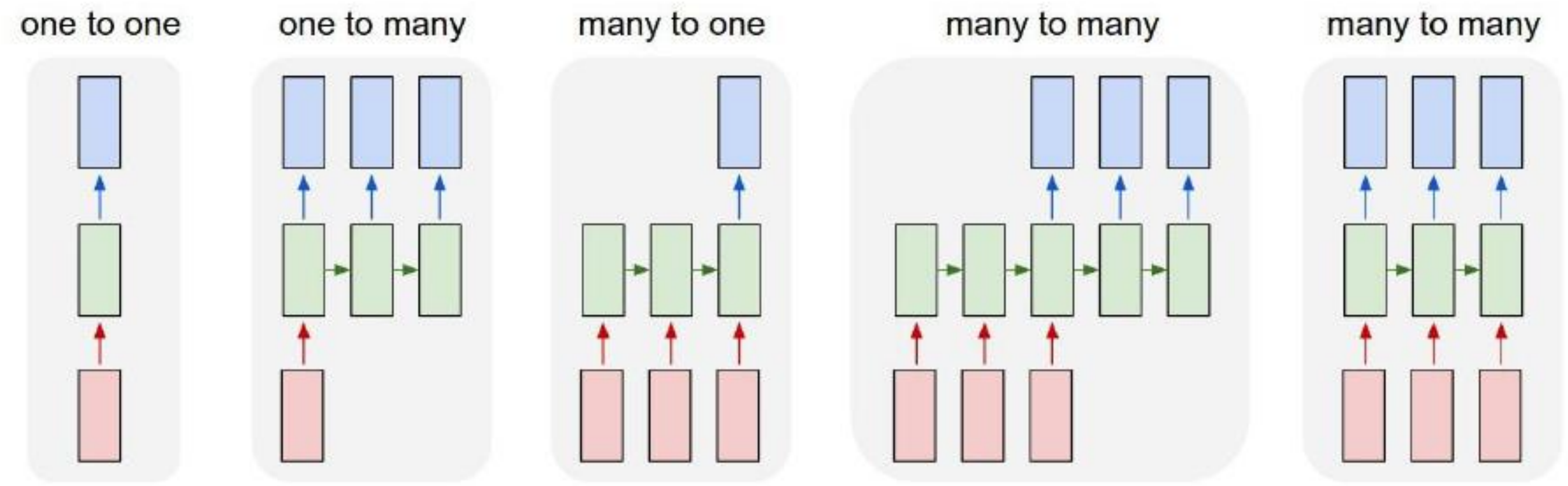


Dense Block



RNN

#01 RNN



Vanilla Neural Networks

Ex. Image Captioning

Image -> sequence of words

Ex. Sentiment Classification

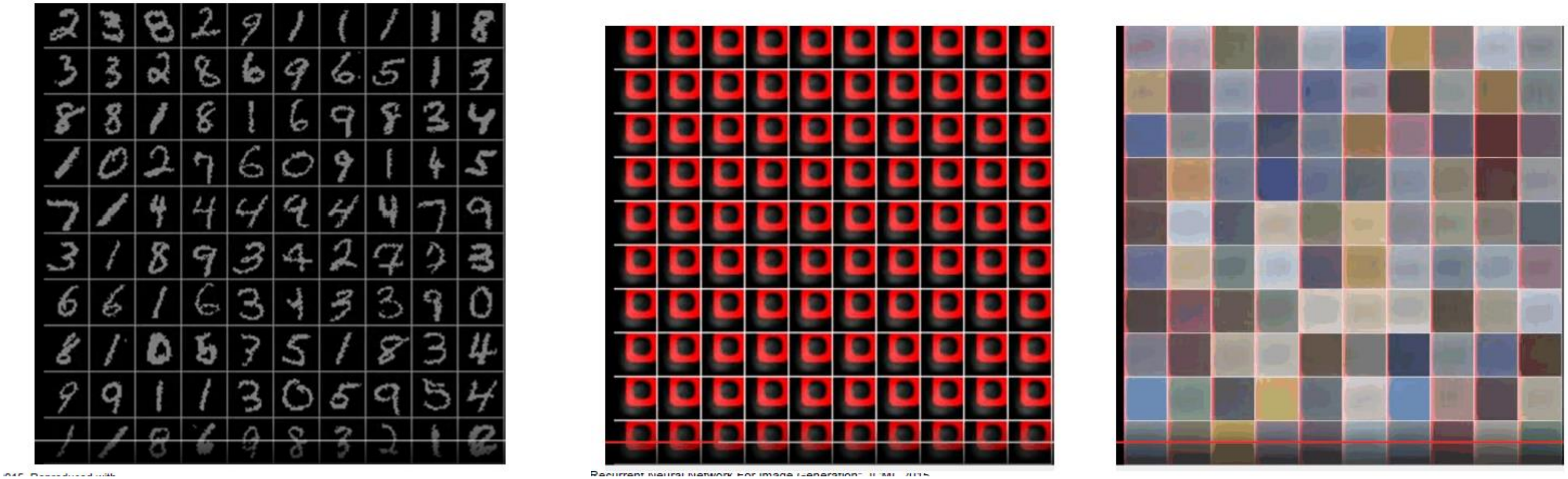
Sequence of words -> sentiment

Ex. Machine Translation

Seq of words -> seq of words

Ex. Video classification on frame level

#01 RNN



Fixd input image 를 RNN sequential processing 한 결과

#01 RNN

We can process a sequence of vectors **x** by
applying a **recurrence formula** at every time step:

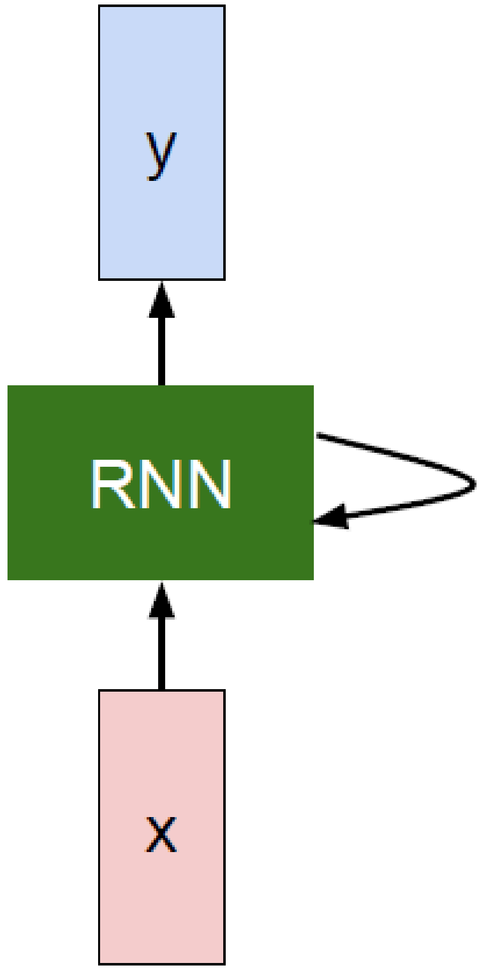
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

old state

input vector at
some time step

some function
with parameters W



전 단계의 hidden state와 현재 input xt로부터 다음 hidden state가 생성된다.
Input이 들어올 때마다 hidden state는 update가 된다.

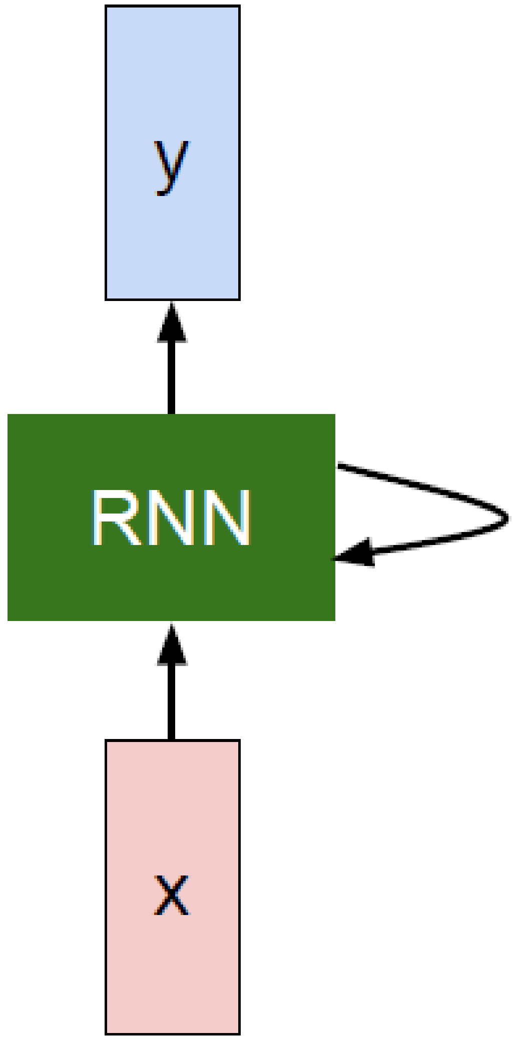
the same function and the same set of parameters are used at every time step.

즉, 자신의 영역에서 parameter와 function은 매 step 마다 동일하게 사용한다.

#01 RNN

(Vanilla) Recurrent Neural Network

The state consists of a single *"hidden"* vector **h**:



$$h_t = f_W(h_{t-1}, x_t)$$

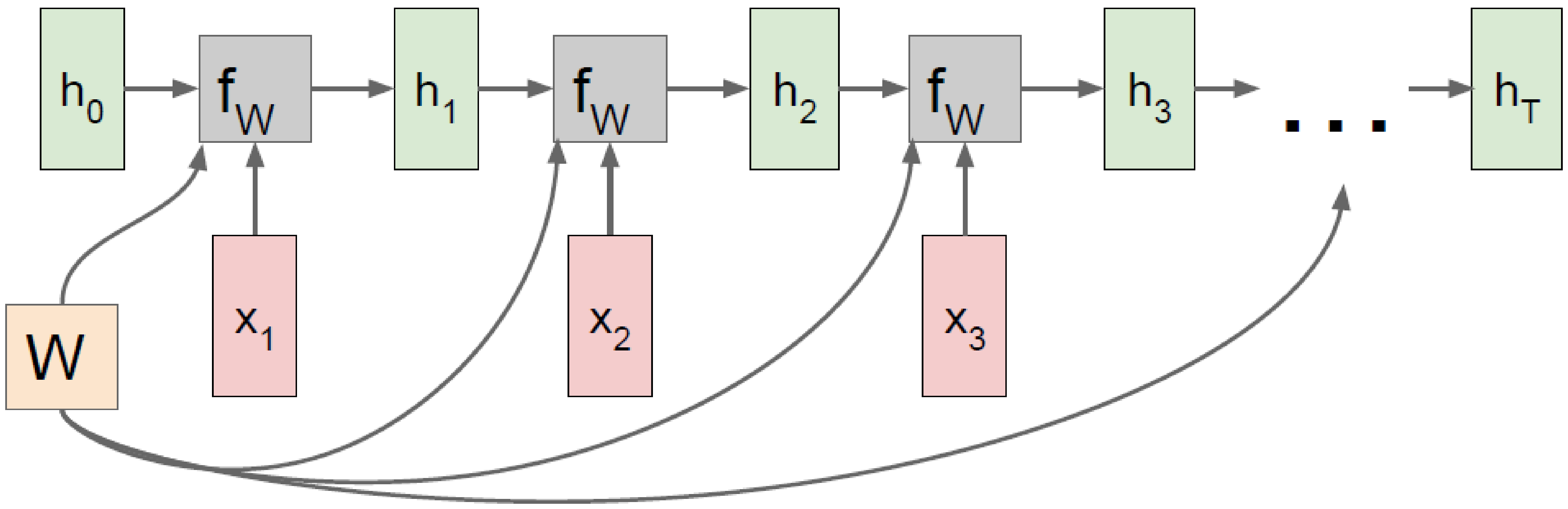
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

#01 RNN

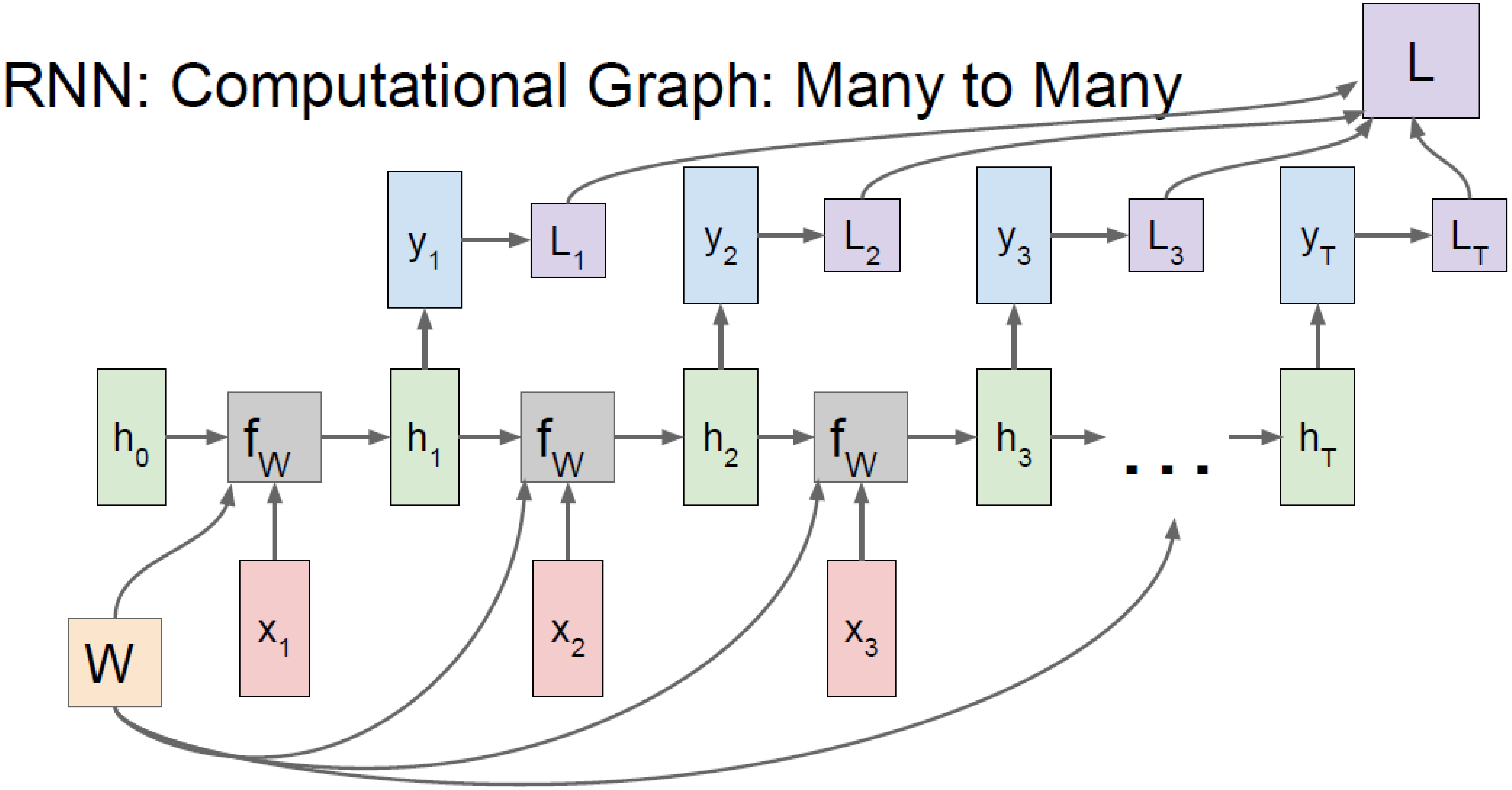
RNN: Computational Graph

Re-use the same weight matrix at every time-step



H,x값이 연속적으로 처리되고 있으므로 달라지지만 가중치 w값은 동일한 것을 확인할 수 있다

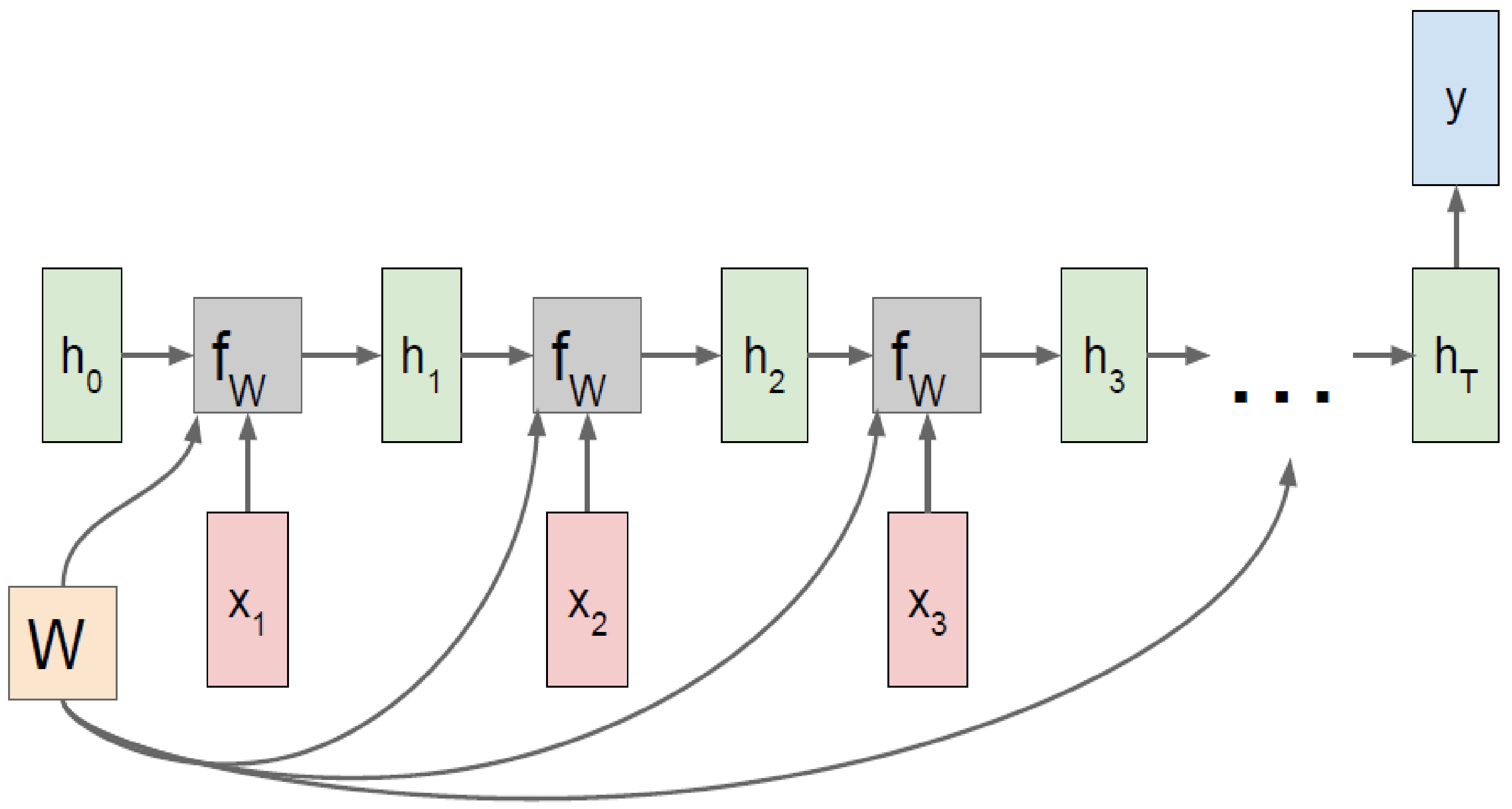
#01 RNN



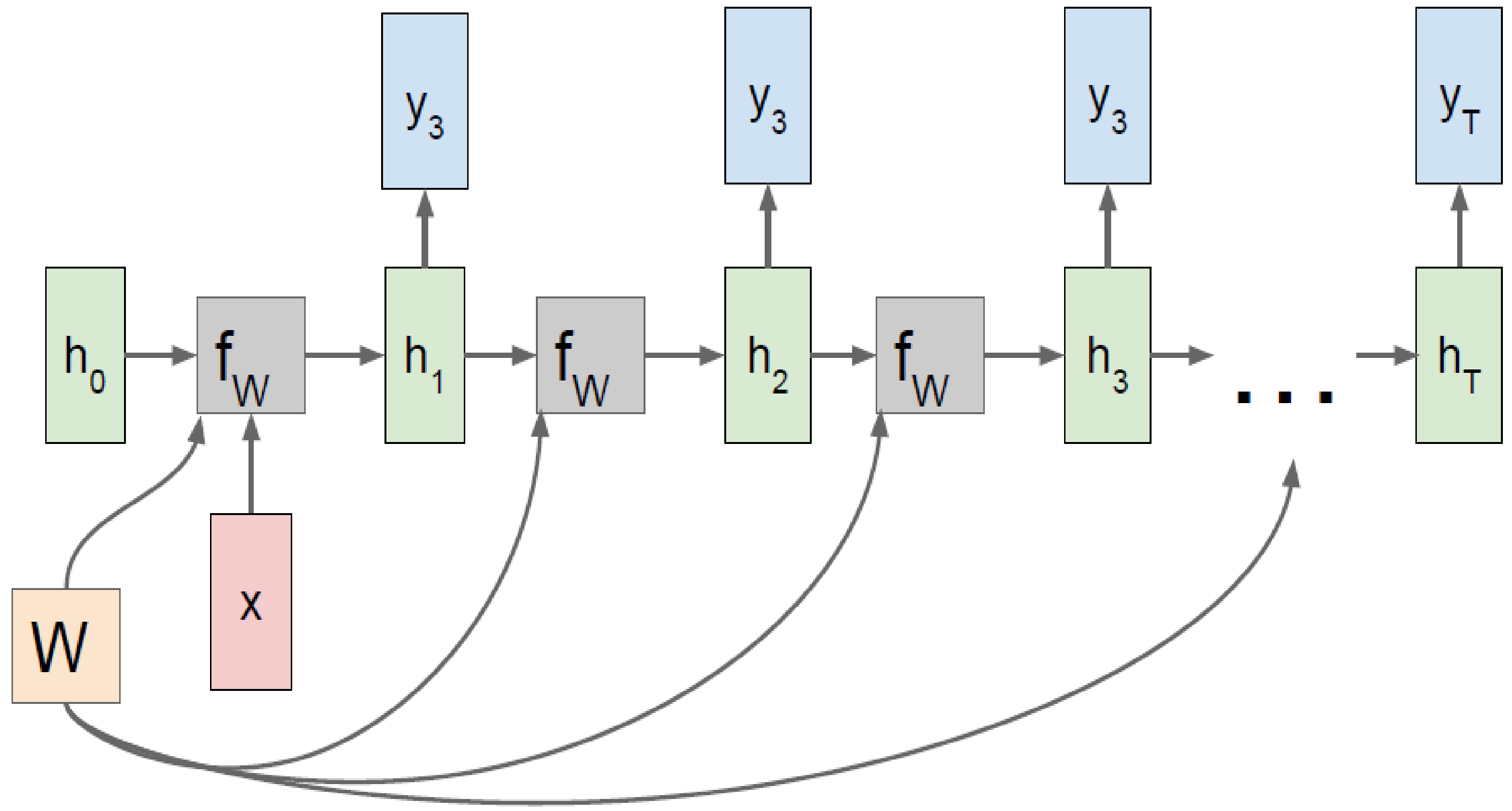
출력값마다 나오는 loss의 총합이 최종 loss가 된다. W값이 동일하기 때문에 역전파할 때는 gradient를 합산하면 된다.

#01 RNN

RNN: Computational Graph: Many to One

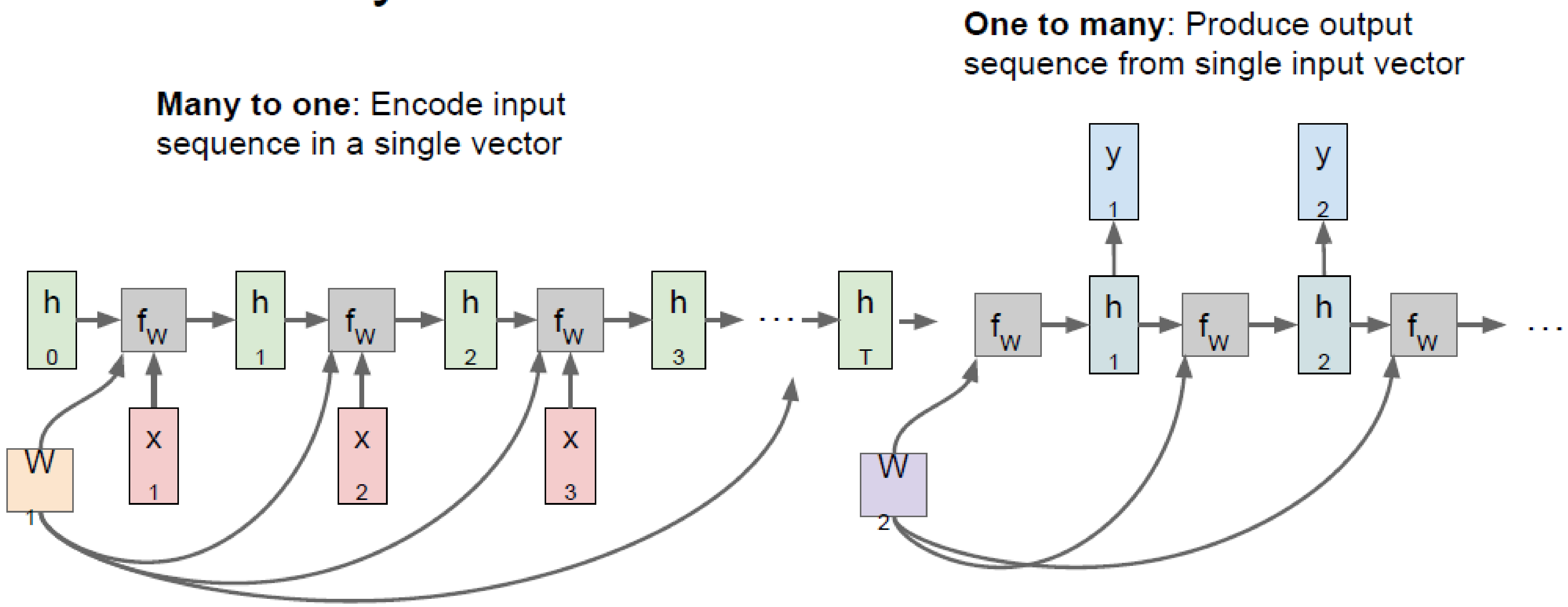


RNN: Computational Graph: One to Many

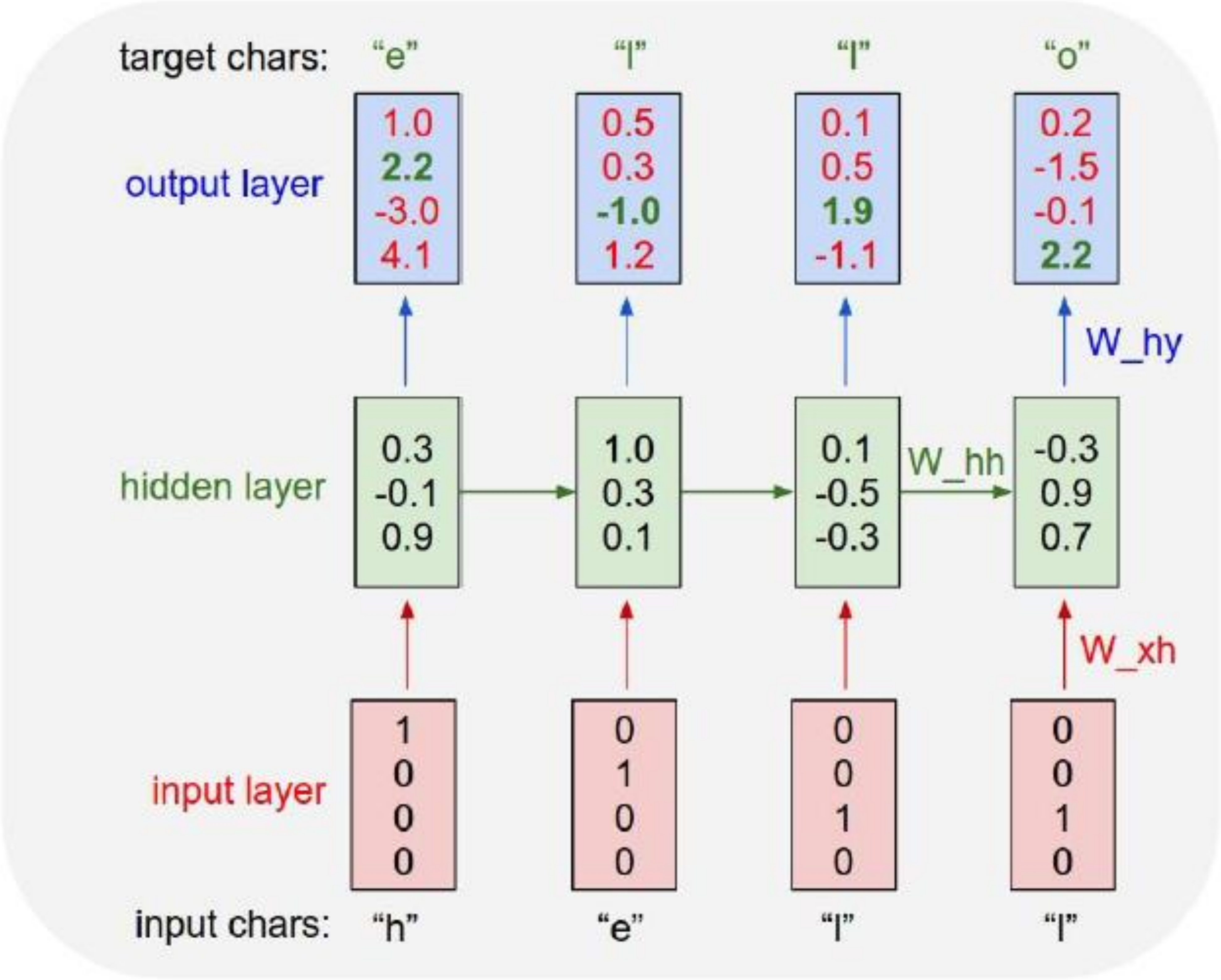


#01 RNN

Sequence to Sequence: Many-to-one + one-to-many



#01 RNN : train

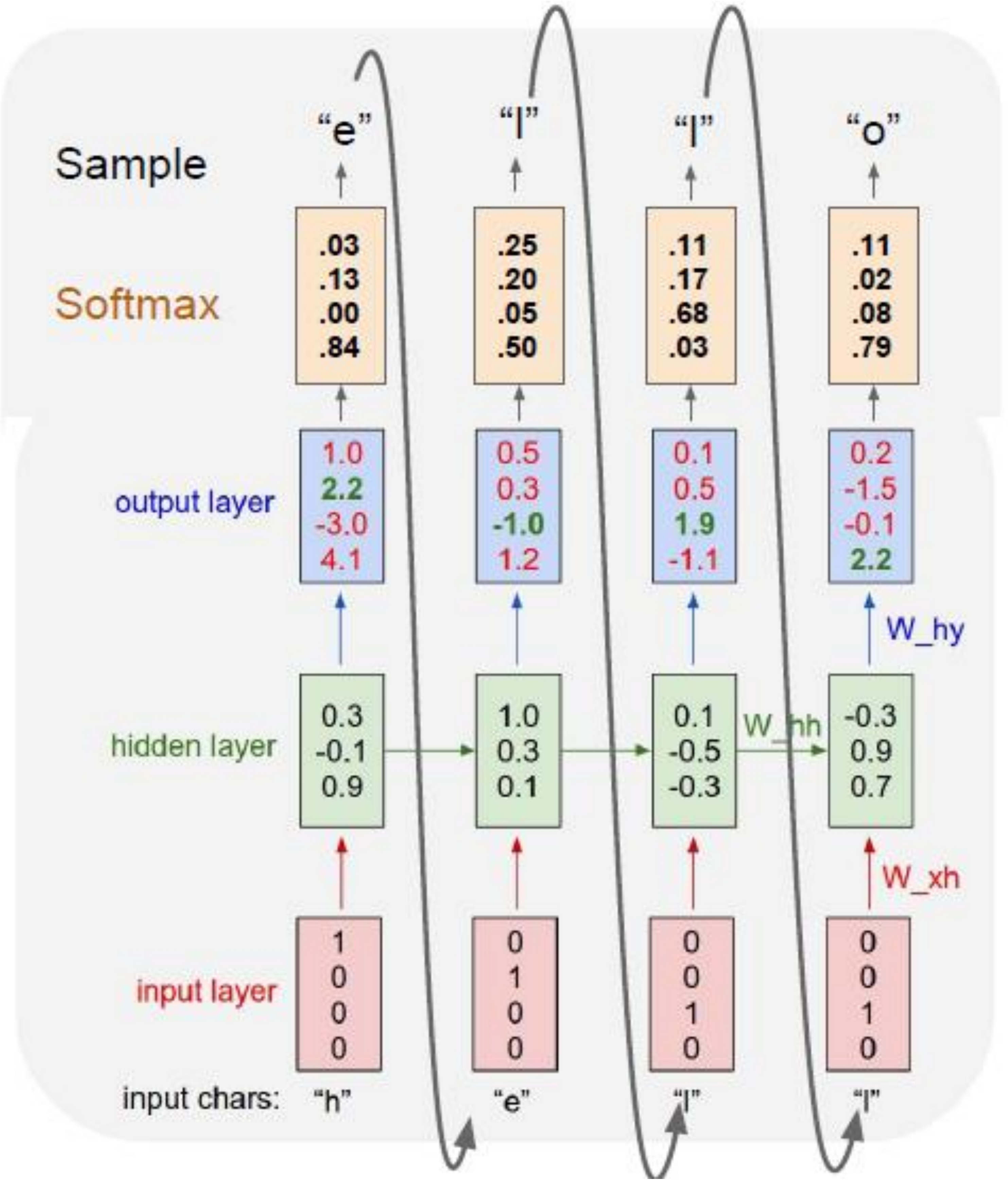


정답값과 오차를 통해 loss를 구한 후 역전파를 하면서 학습

Hidden layer
다음 hidden state에 영향을 줌

활성화함수 계산의 성능을 개선하기 위해 one-hot encoding

#01 RNN: Sampling at test time



출력을 다시 다음 입력으로 넣어줌 = Sampling

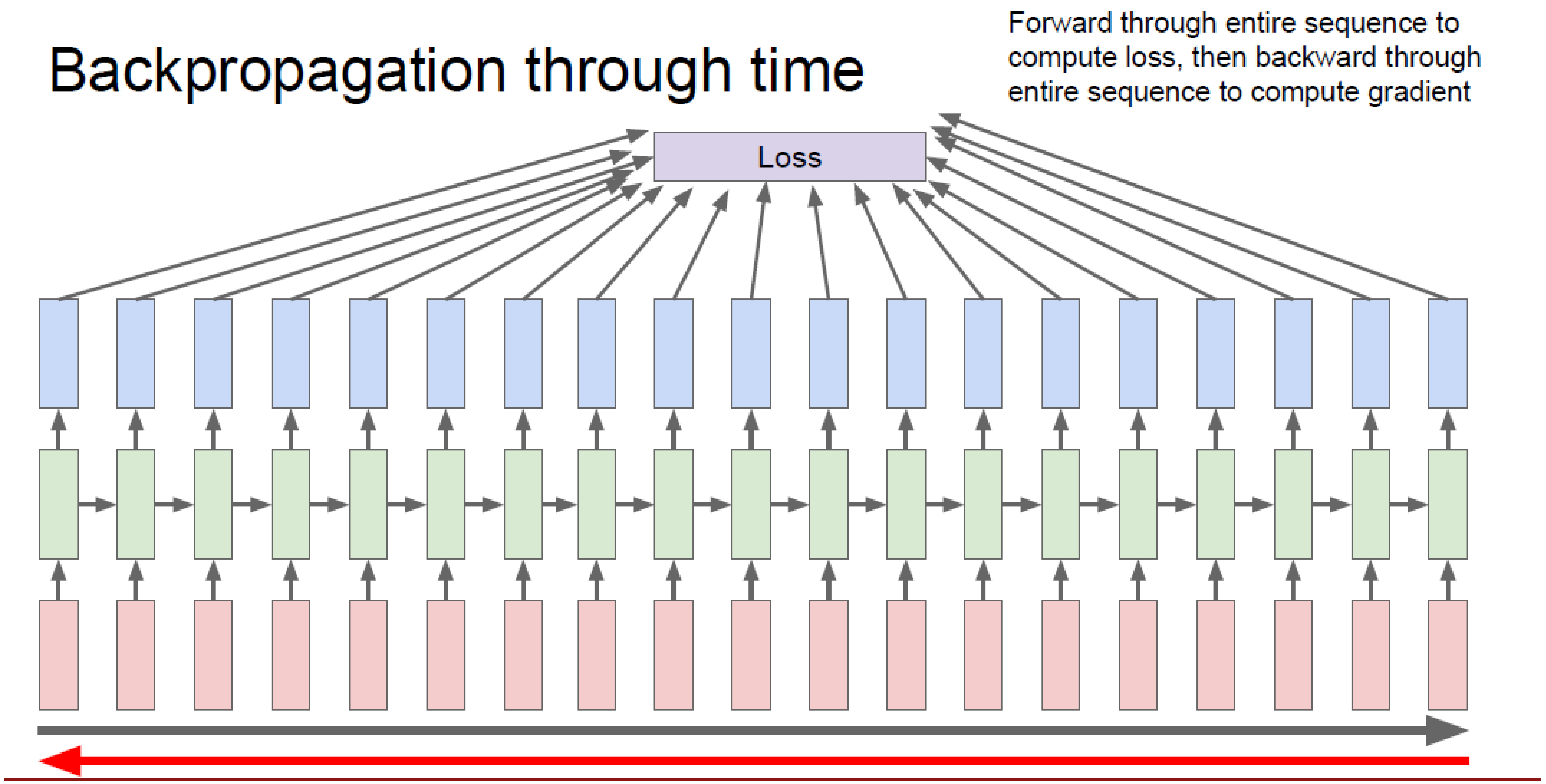
점수 분포를 확률 분포로 수정. 값끼리의 비교

문자에 대해 점수 분포를 산출

Hidden layer
다음 hidden state에 영향을 줌

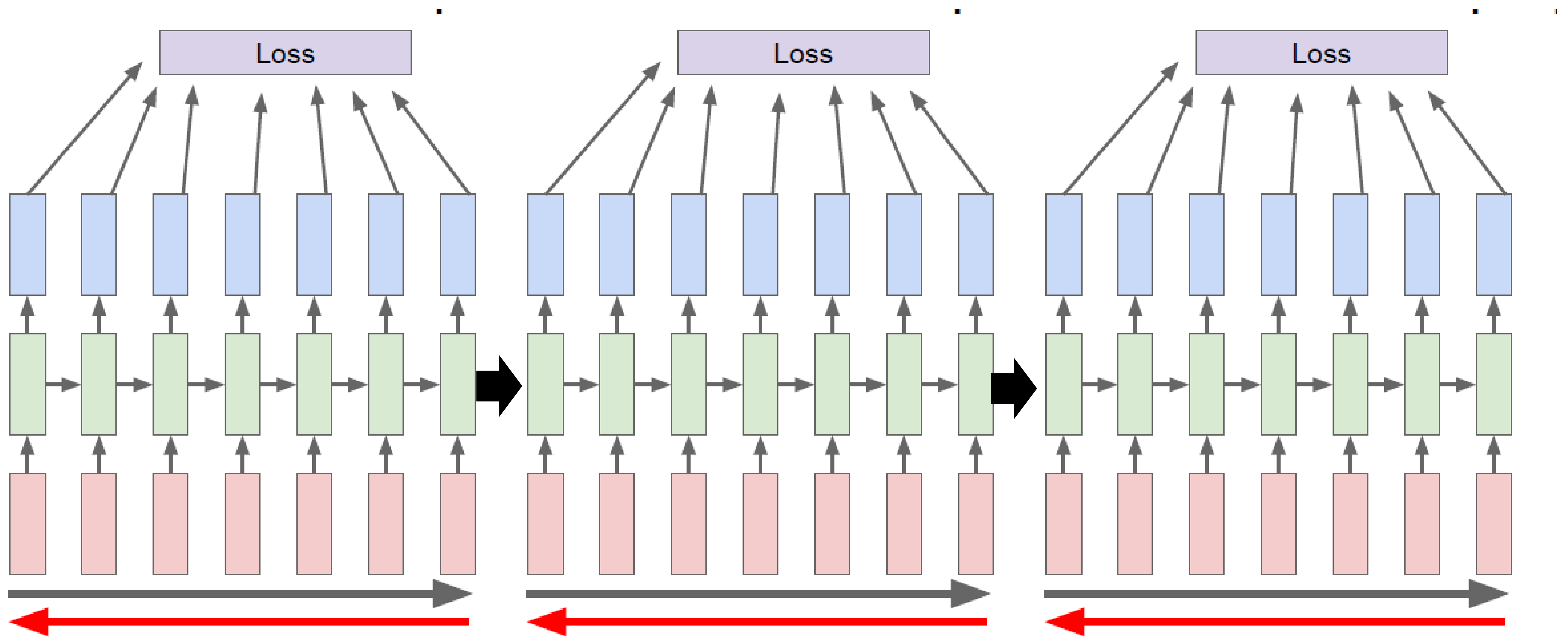
활성화함수 계산의 성능을 개선하기 위해 one-hot encoding

#01 RNN



모든 출력을 구하면서 역전파를 진행하면 계산량이 증가하고 속도가 느려지는 문제점 발생.

#01 RNN



Mini batch 를 나눠서 일정 부분만큼 forward 후 loss 계산한다.
Gradient를 통해 update를 하는 과정

#01 RNN

```python
"""
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
```

가중치 초기화

#01 RNN

```python
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
    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]
```

Loss function의 forward pass
Compute loss

Loss function의 backward pass
Compute parameter gradient

#01 RNN

```python
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
```

RNN에서 학습된 것을 확률분포로
만든 후 출력을 다시 입력으로 넣음
Sampling

#01 RNN

```python
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
```

Main loop

$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

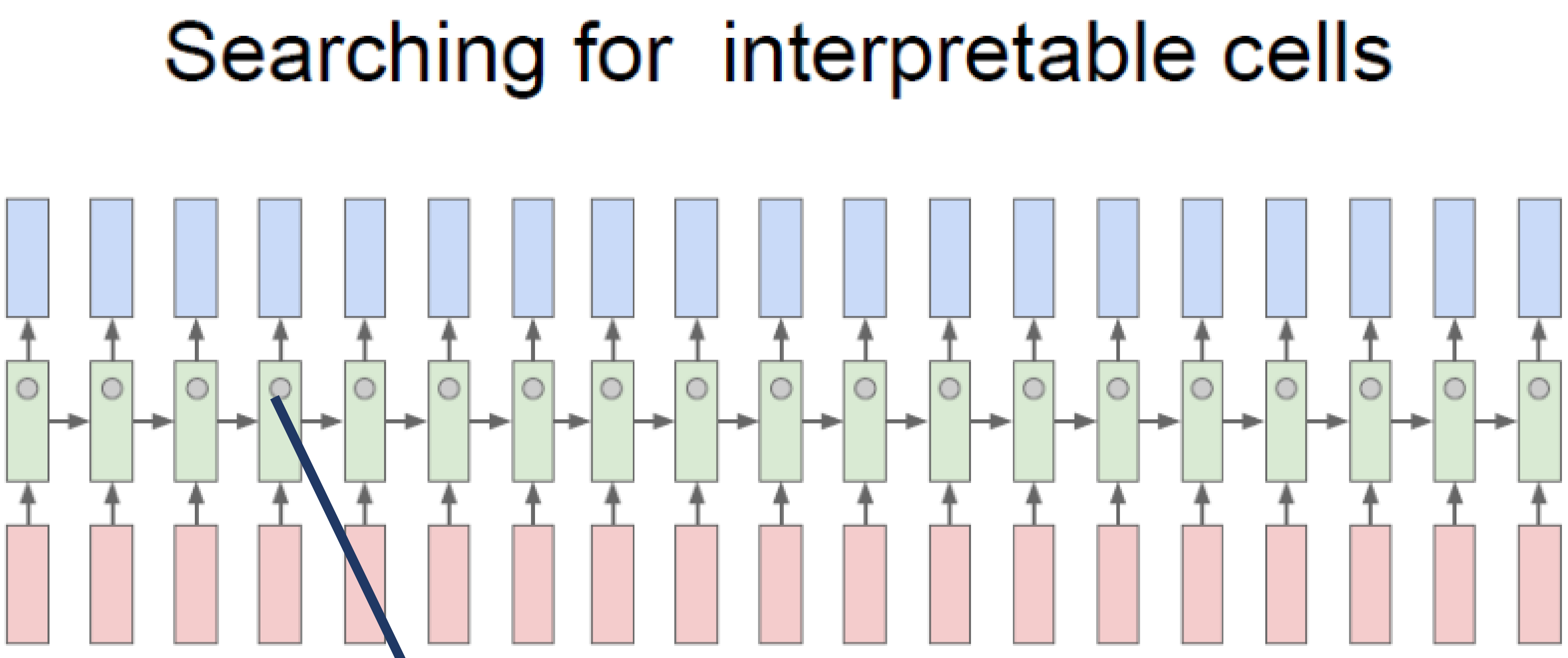
some function
with parameters W

old state

input vector at
some time step

RNN Practical Use

01 Searching for interpretable cells



Hidden vector! -> Updating

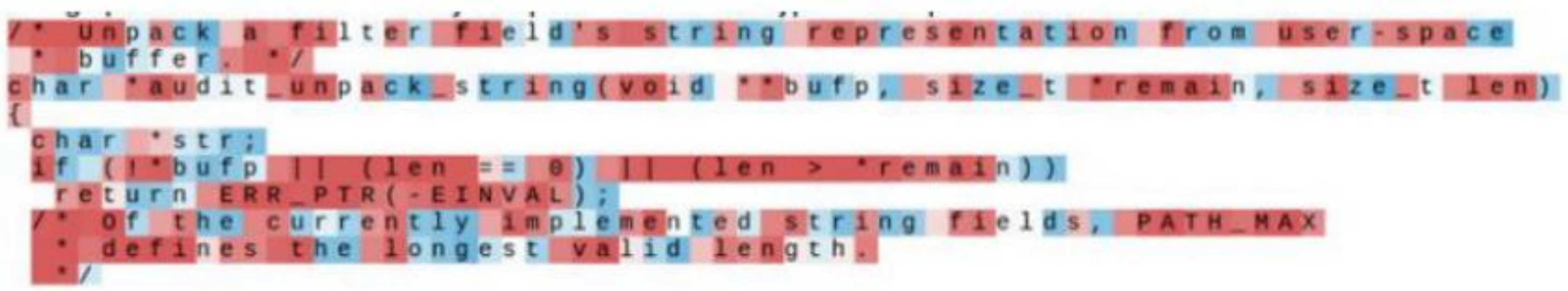
= RNN의 학습 과정

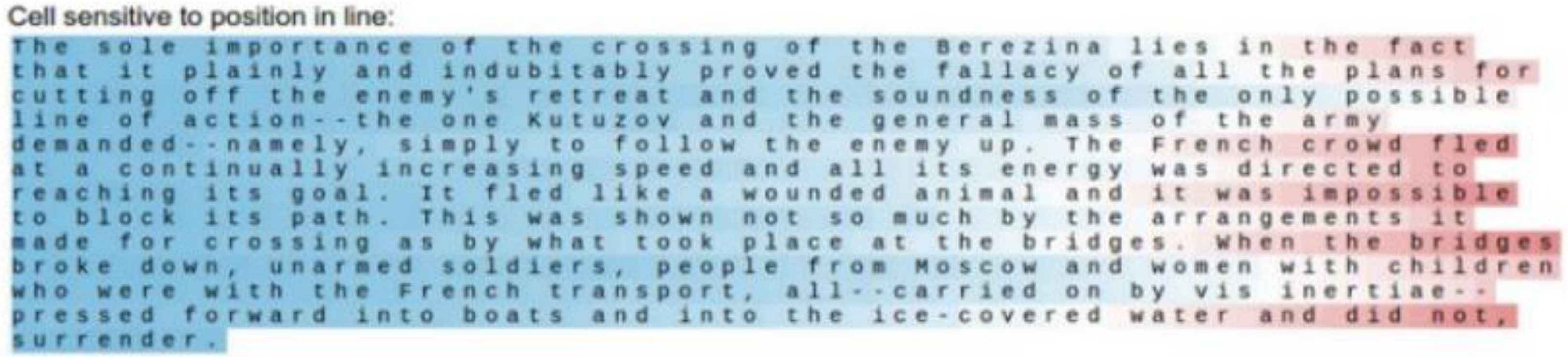
각각의 vector들이 의미하는 바를 직접 확인함으로써 해석 가능한 의미 있는 vector들을 볼 수 있지 않을까?

01 Searching for interpretable cells

하나의 hidden states가 인식하고 있는 것 = 그 states에서 출력하는 hidden vector

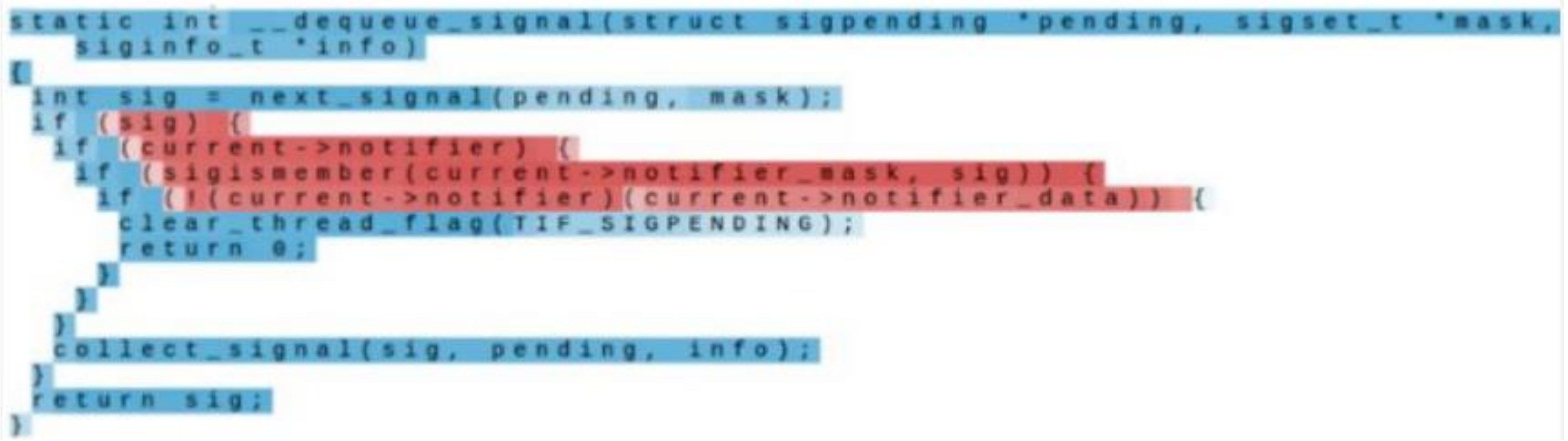
→ hidden vector가 의미하는 바 = RNN이 학습하고 있는 것들





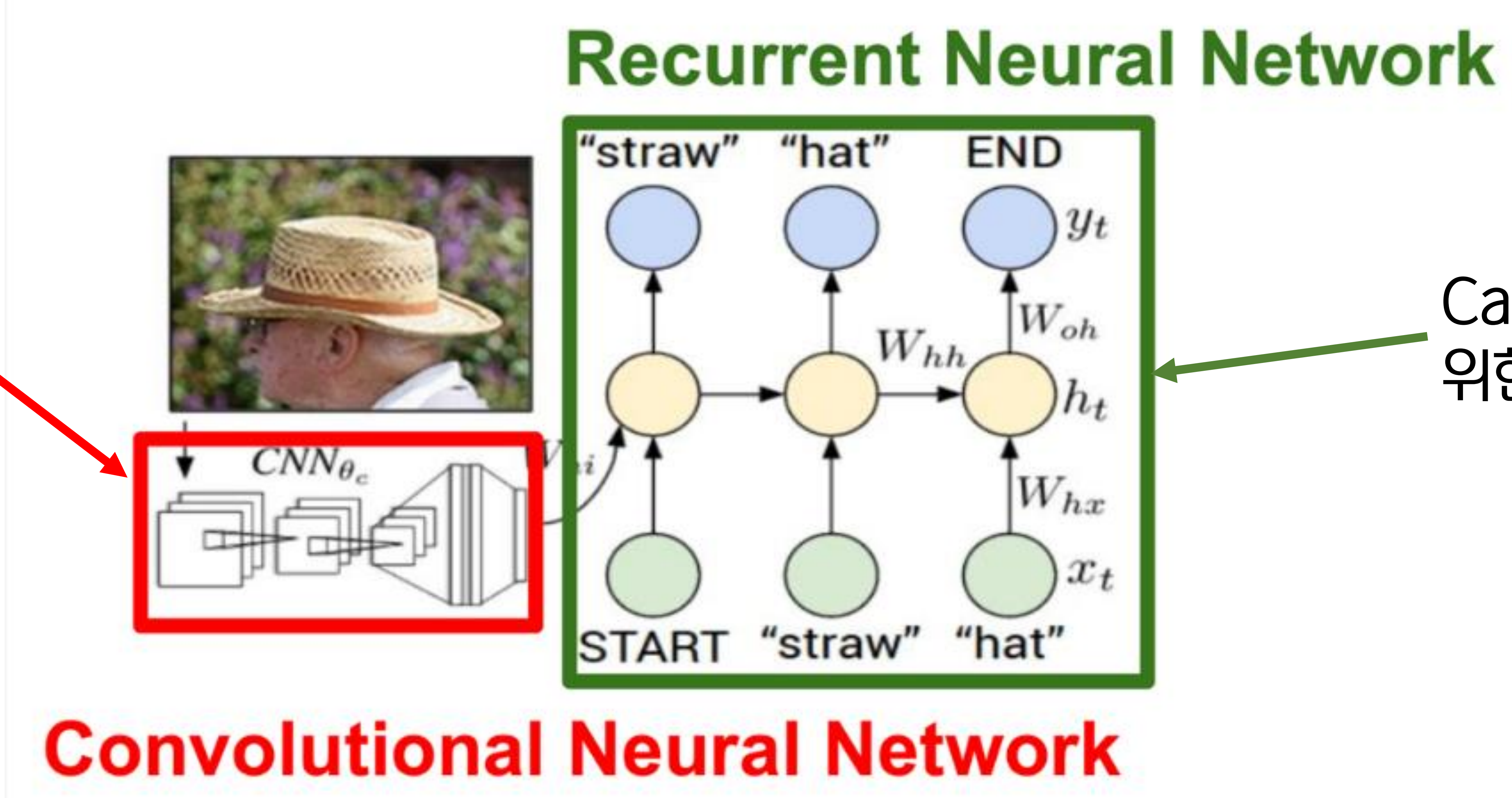
line length tracking cell

quote detection cell



02 Image Captioning

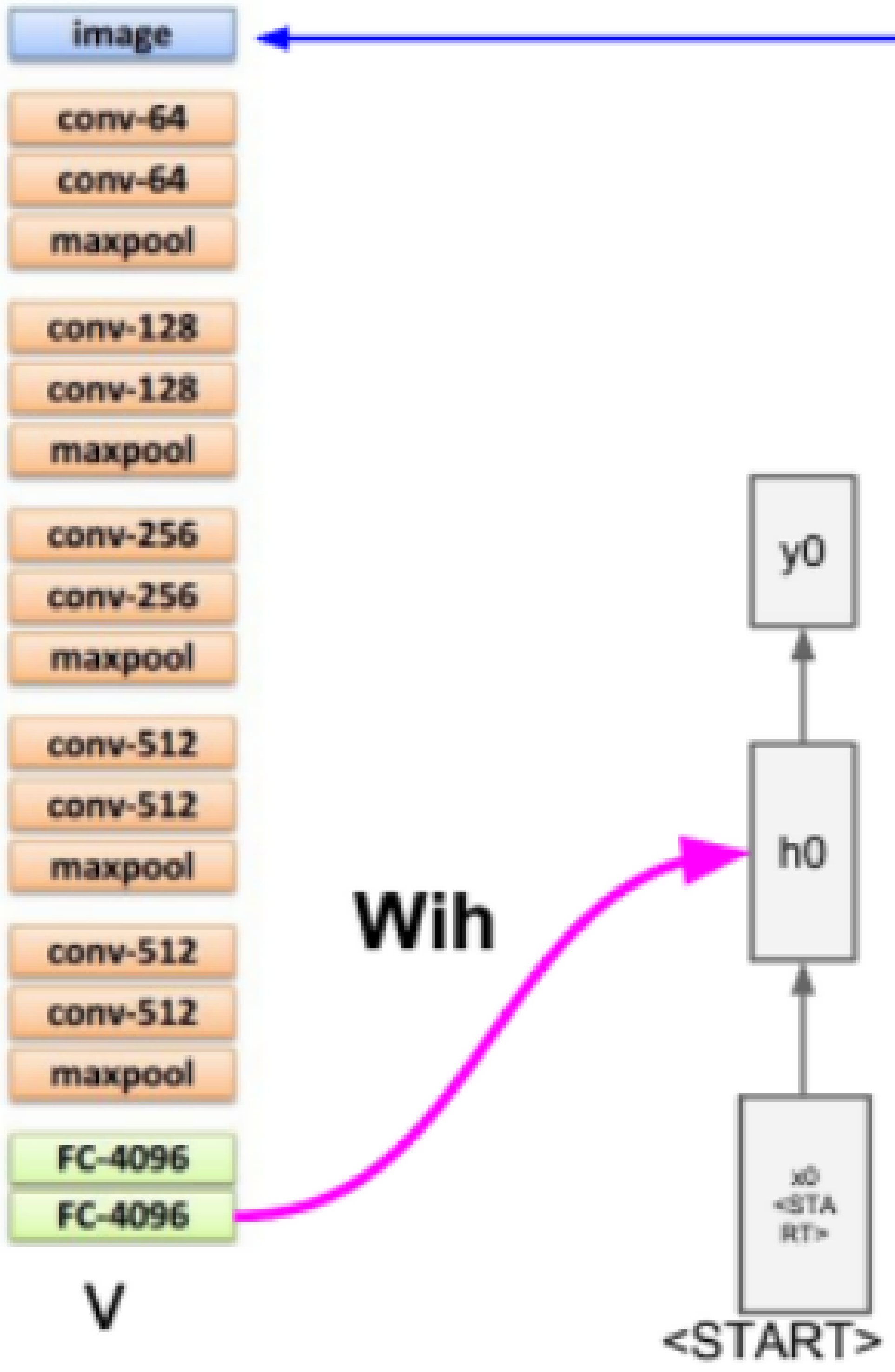
Image input을
위한 CNN



Caption output을
위한 RNN

02 Image Captioning

02 Image Captioning



test image

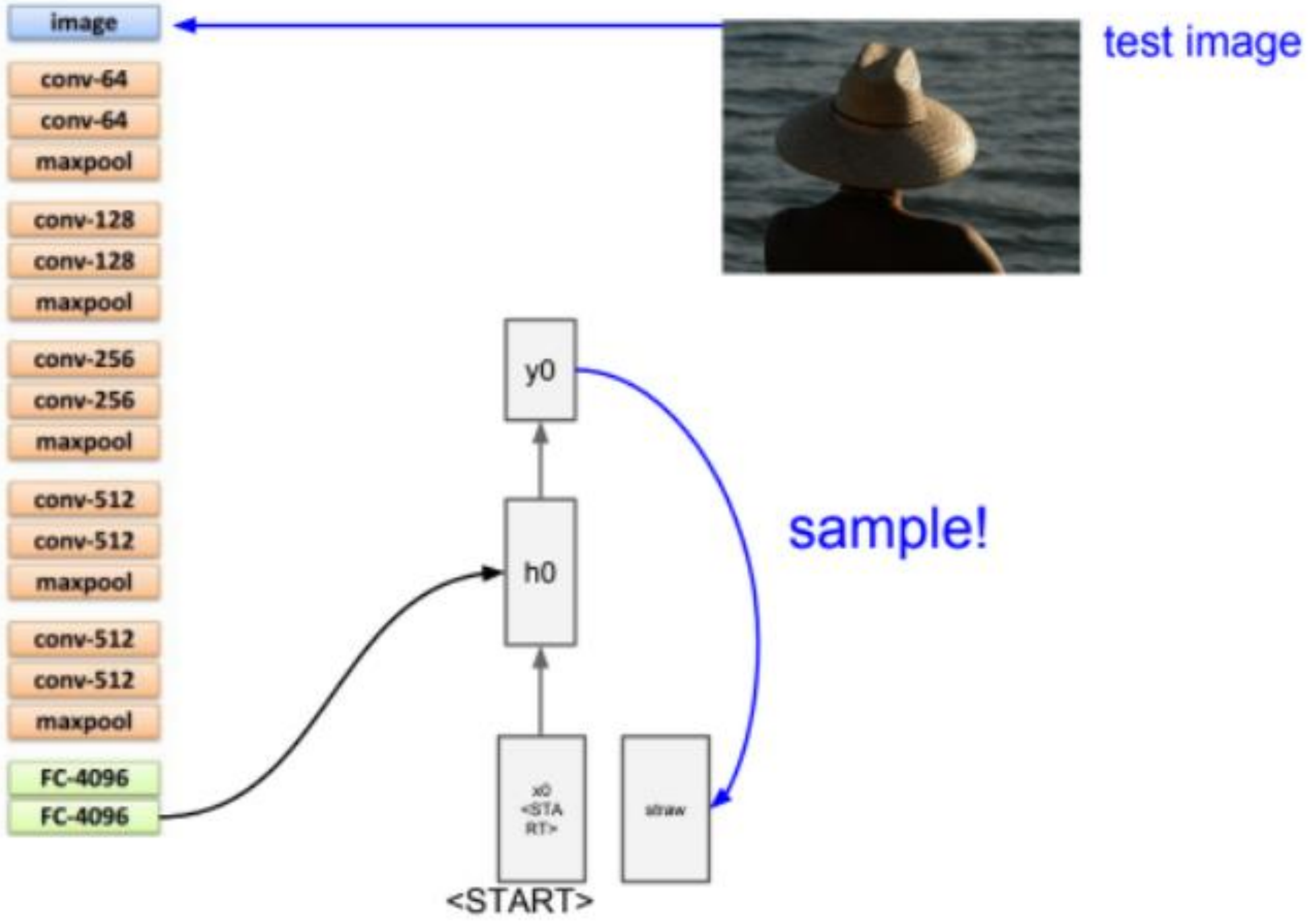
before:

h = tanh(Wxh * x + Whh * h)

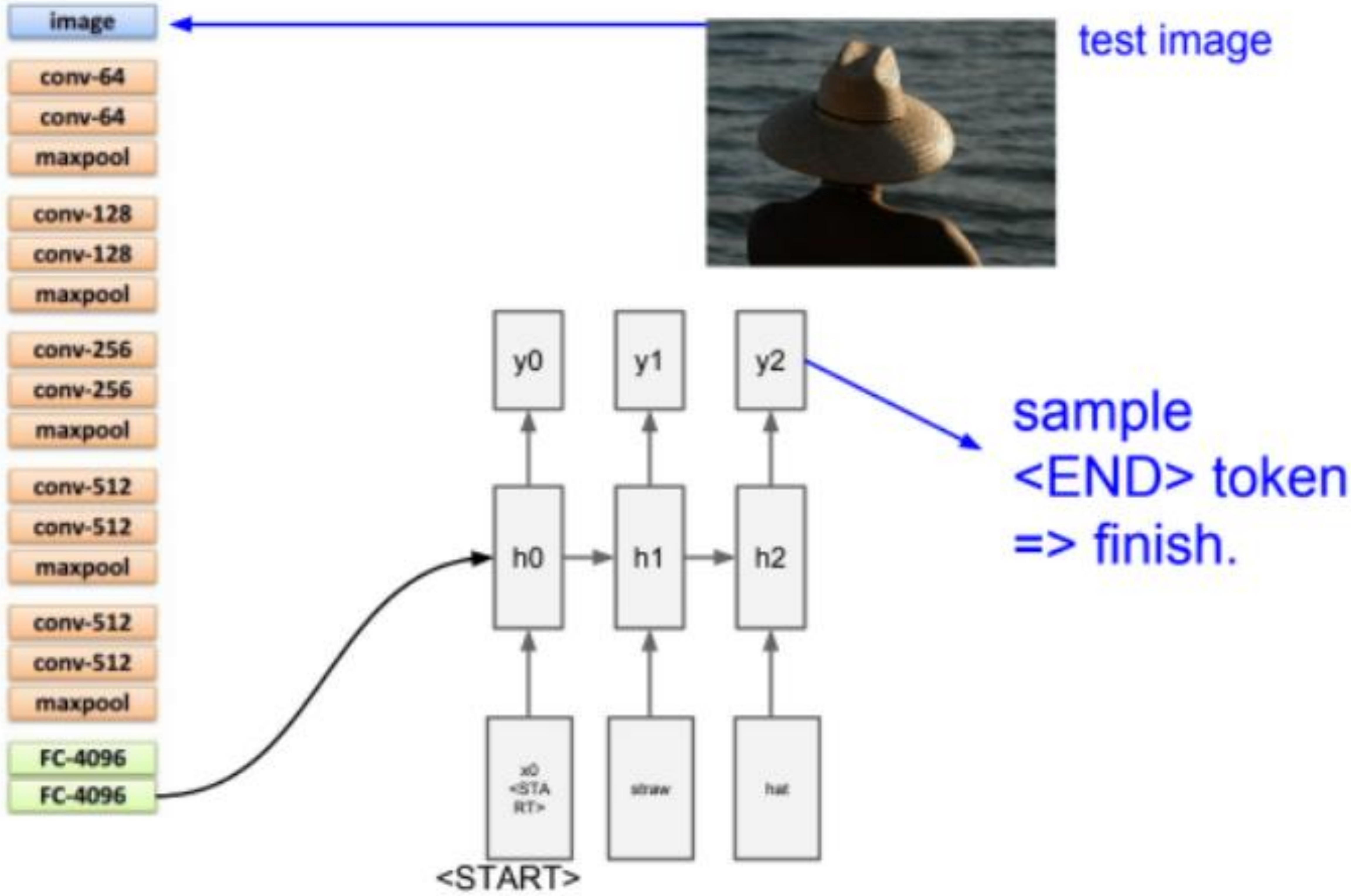
now:

h = tanh(Wxh * x + Whh * h + Wih * v)

02 Image Captioning



02 Image Captioning

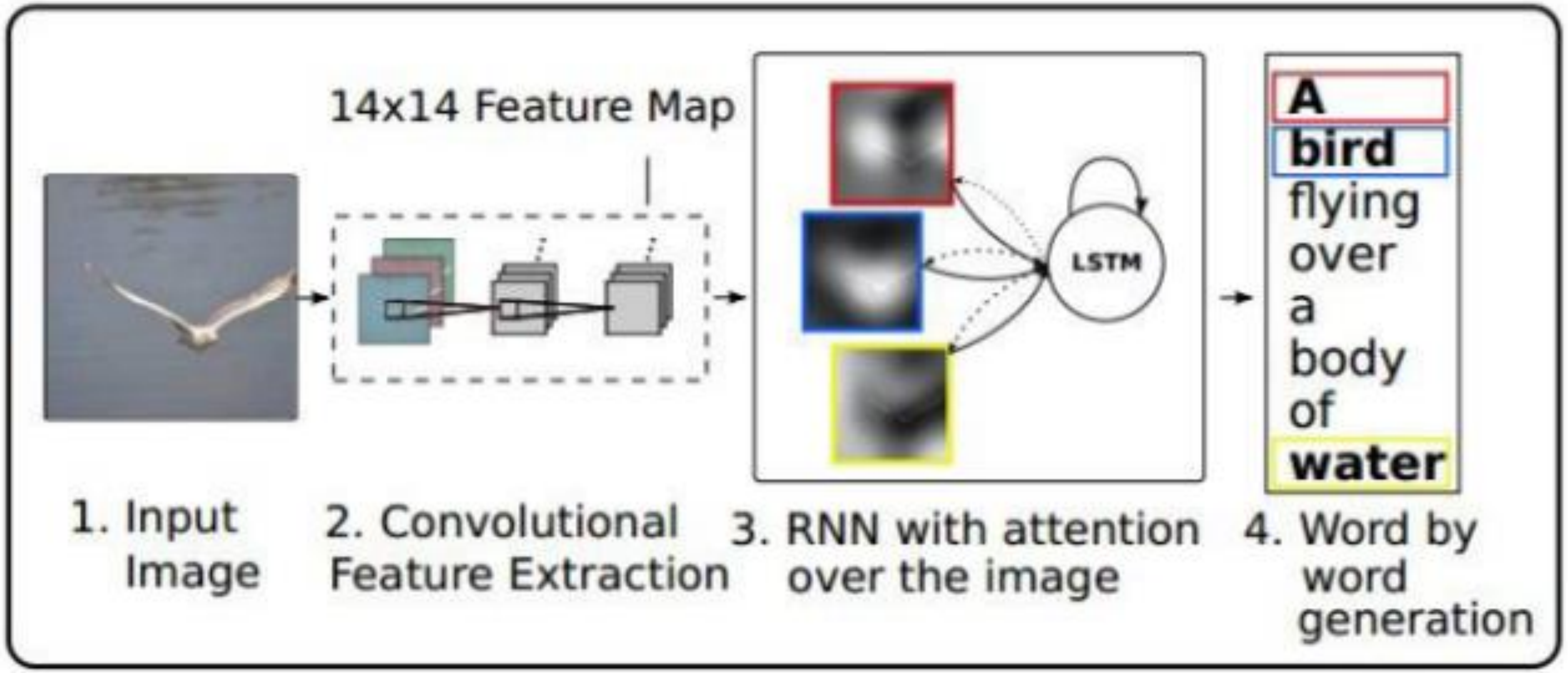


02 Image Captioning

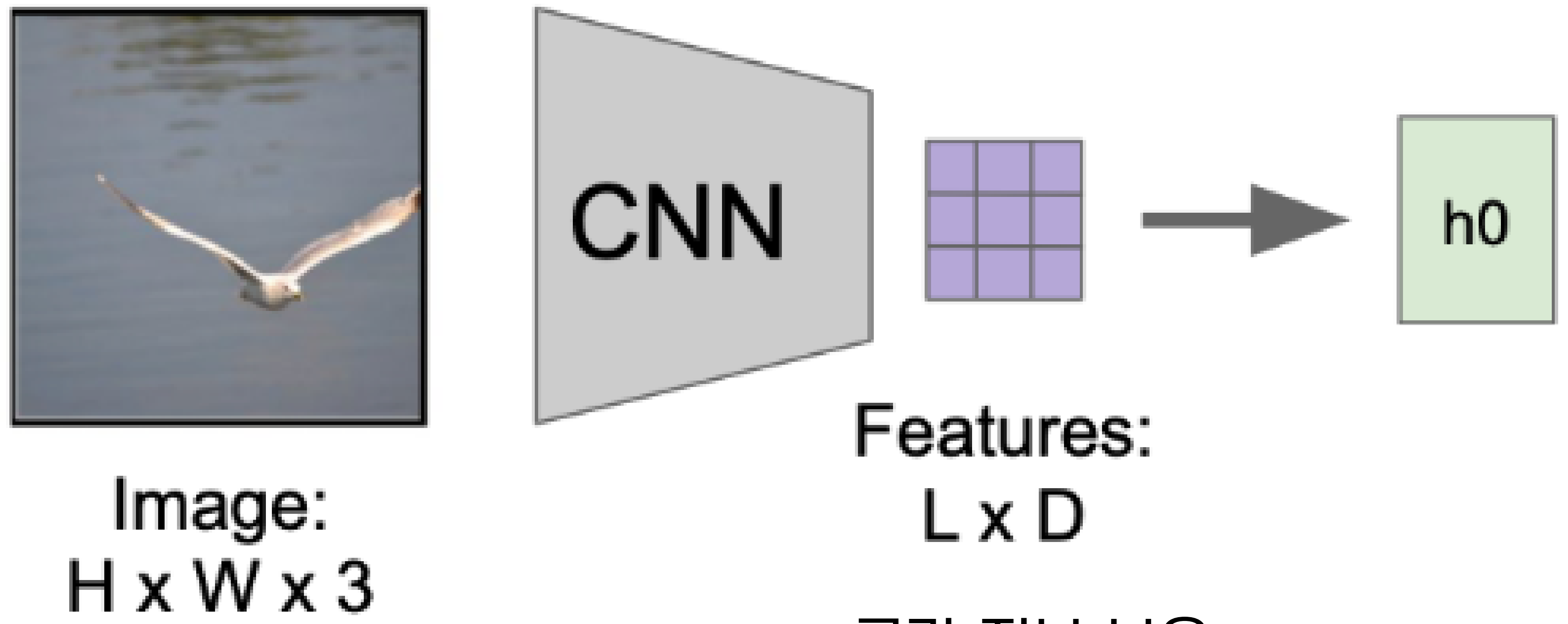
02 Image Captioning

03 Image Captioning with Attention

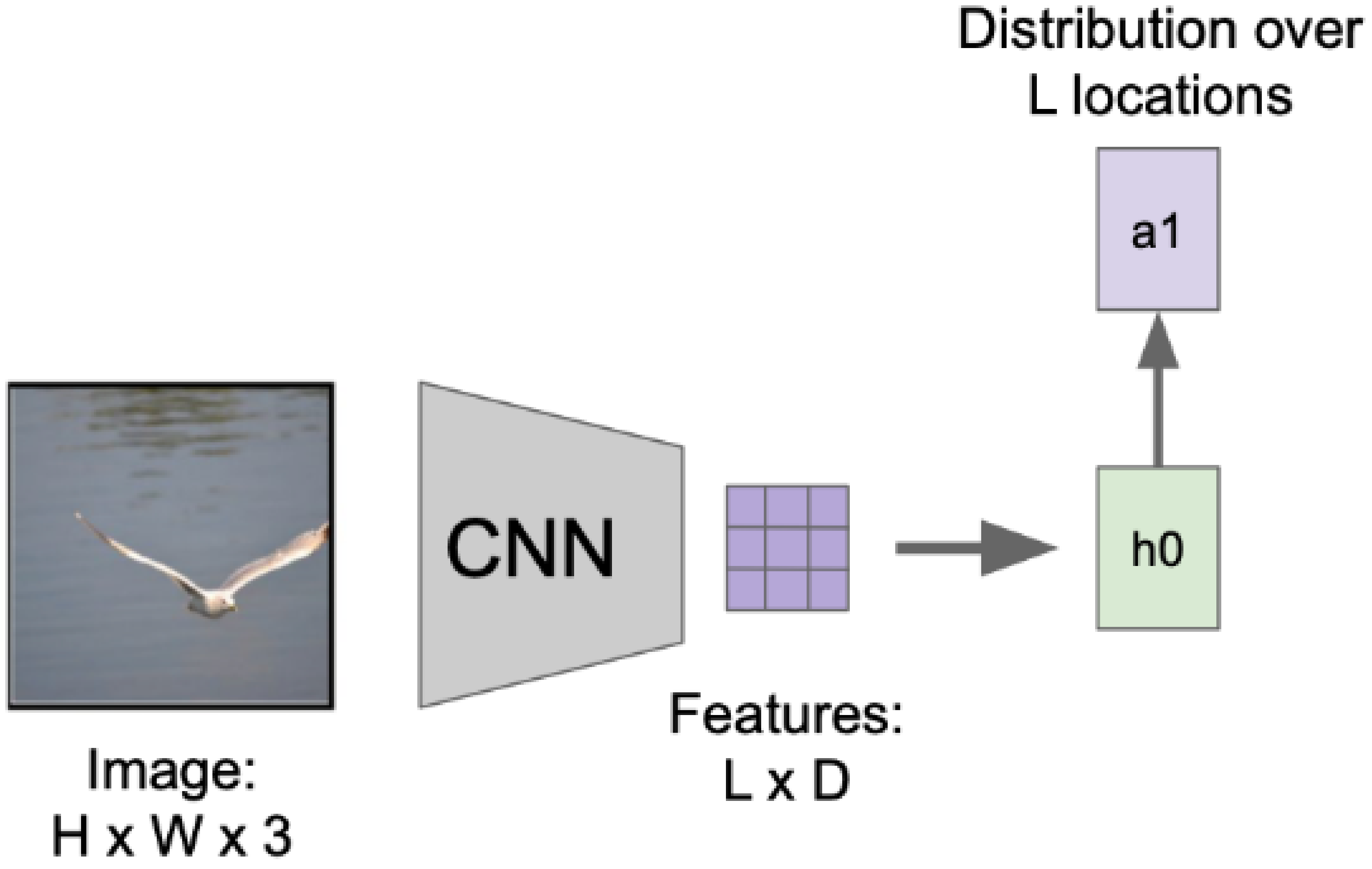
RNN focuses its attention at a different spatial location
when generating each word



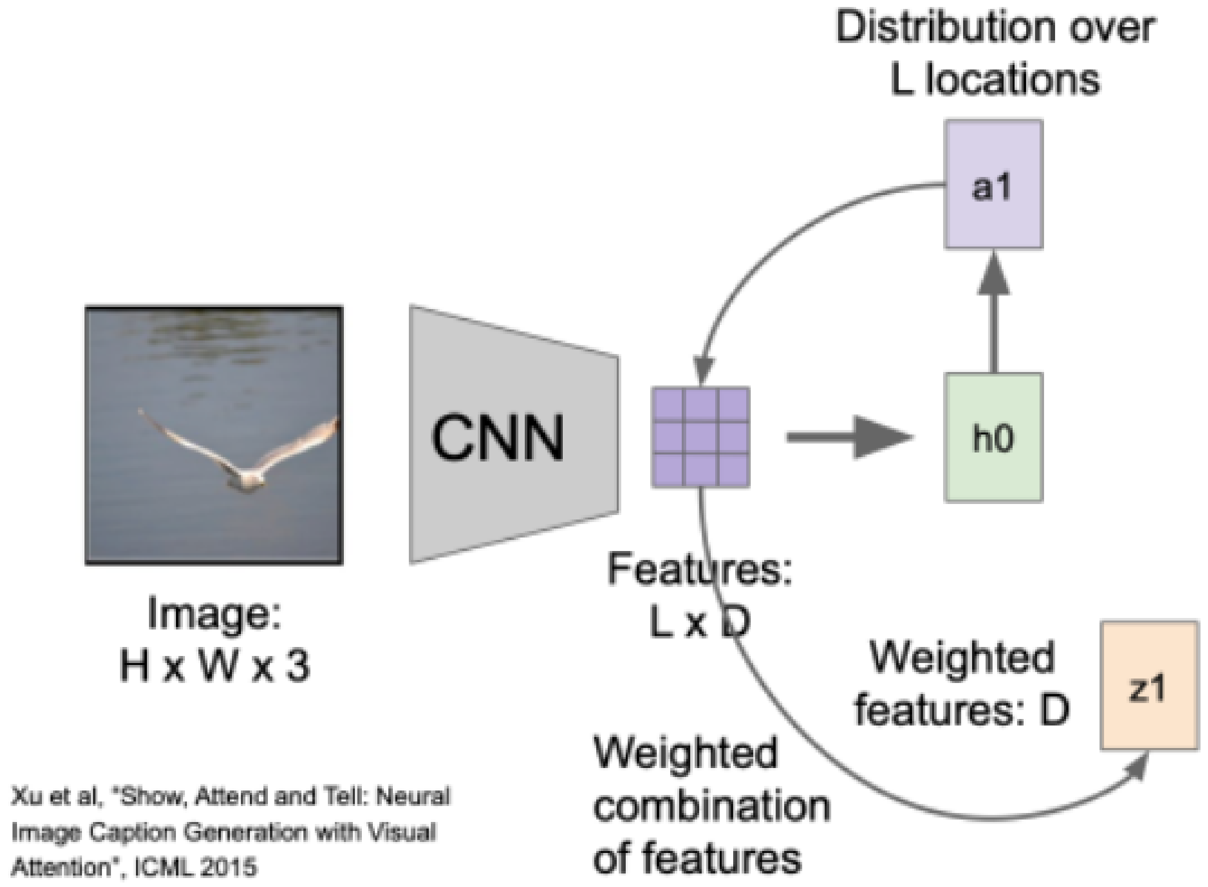
03 Image Captioning with Attention



= 공간 정보 보유
(grid of sector)



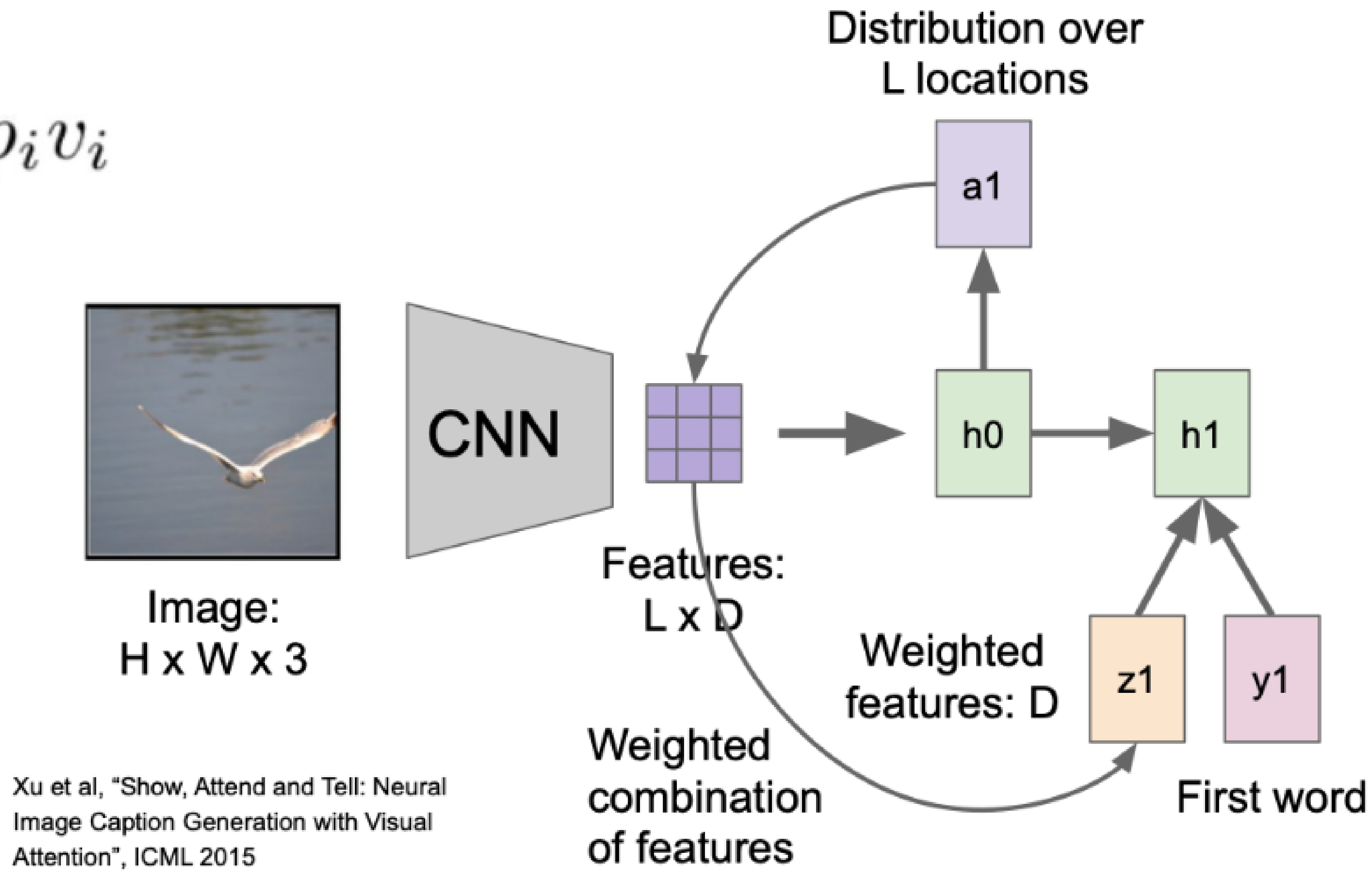
03 Image Captioning with Attention



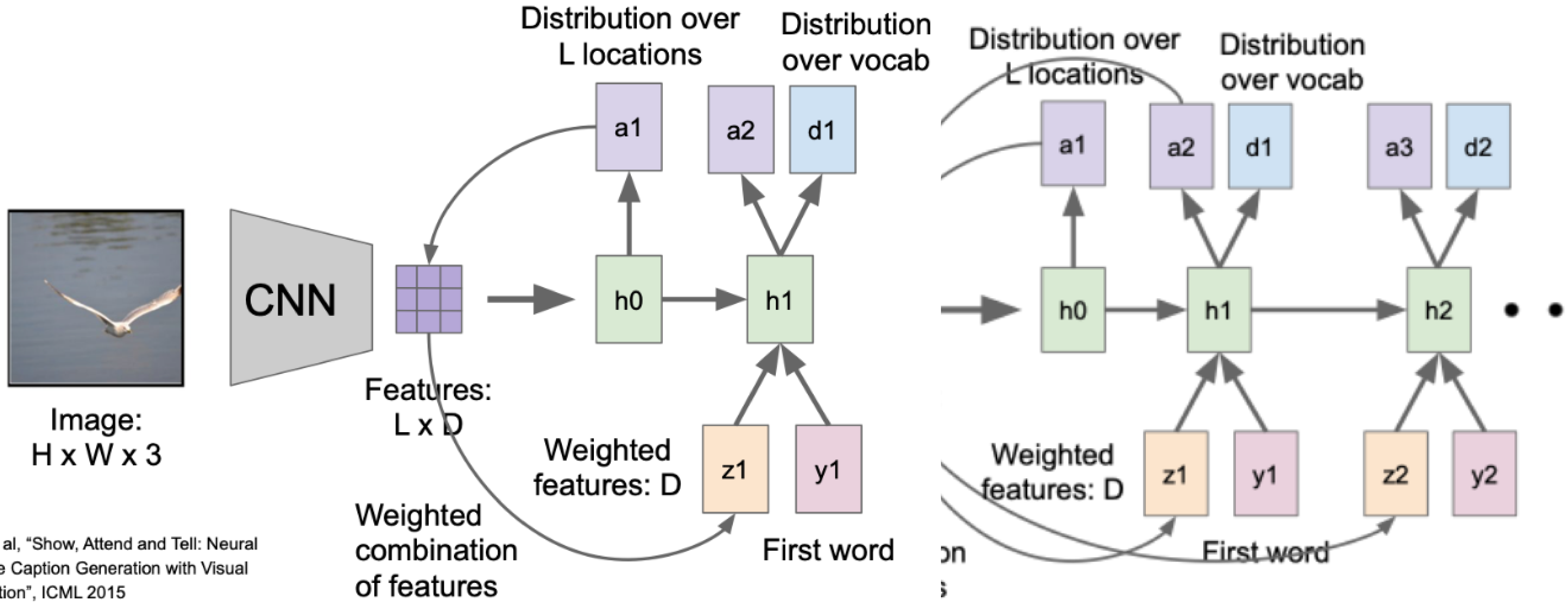
$$z = \sum_{i=1}^{L} p_i v_i$$

이미지에서 보고 싶은
위치에 대한 분포를 생성

→ Train시 모델이
집중해서 봐야하는 위치
지정

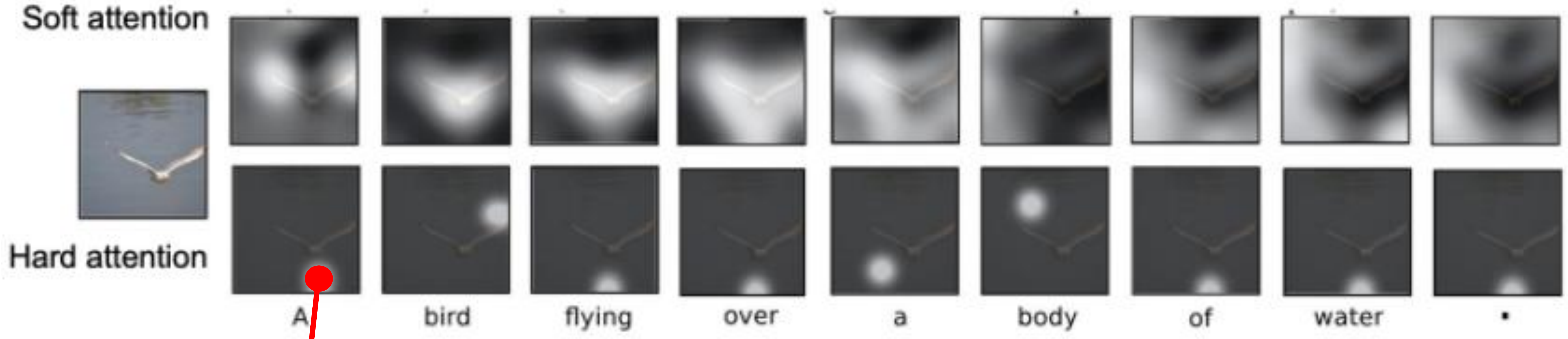


03 Image Captioning with Attention

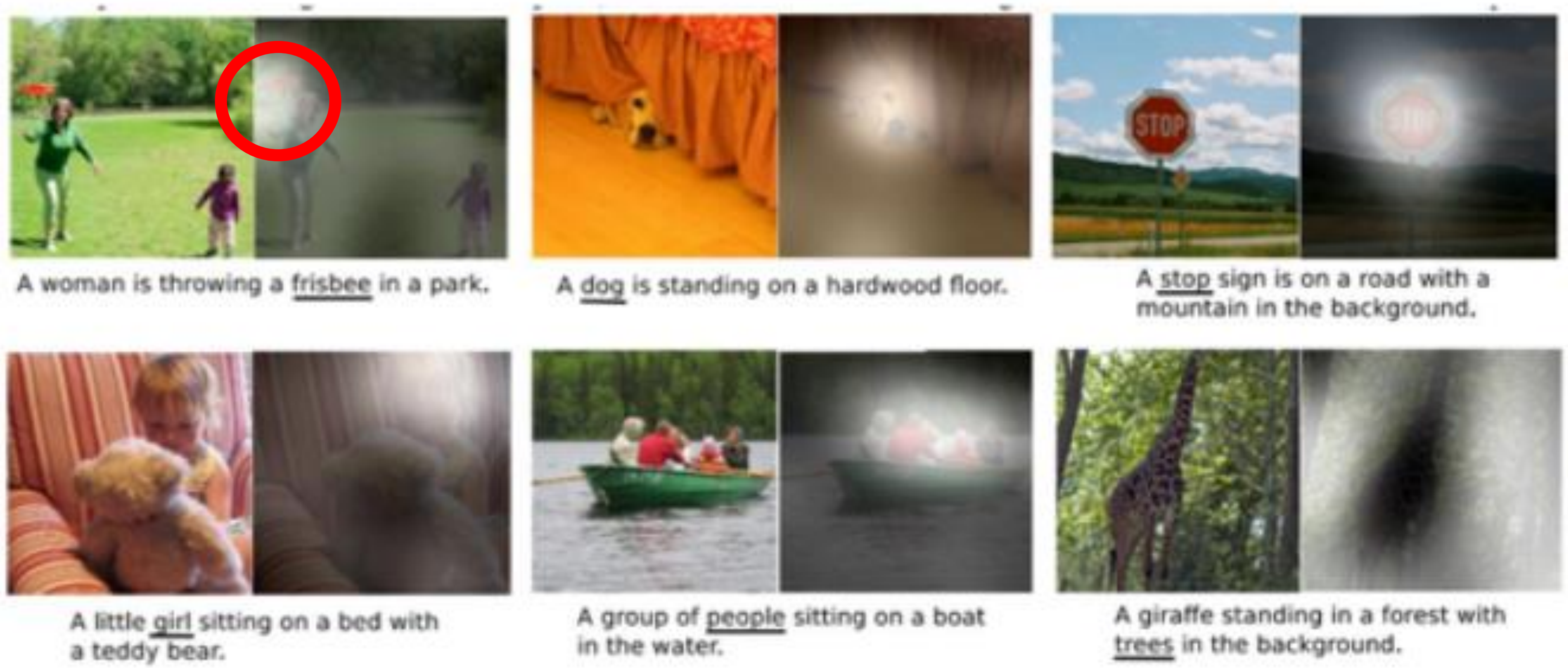


Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

03 Image Captioning with Attention



Attention Point



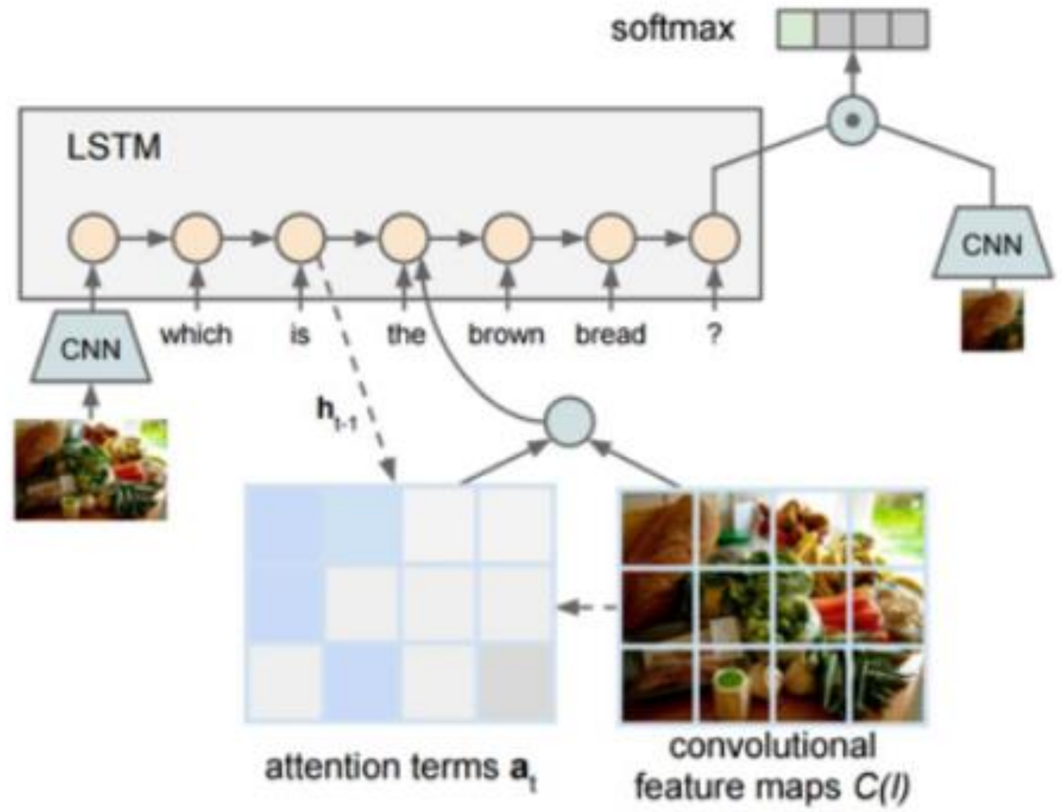
Soft attention
→모든 특징과 이미지 위치 간의 weight

Hard attention
→ 조금 더 국소 부분에 치우쳐서 attention

04 Visual Question Answering

Visual Question Answering: RNNs with Attention

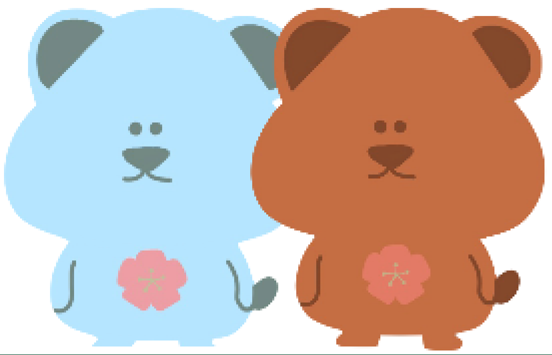
Many of One →



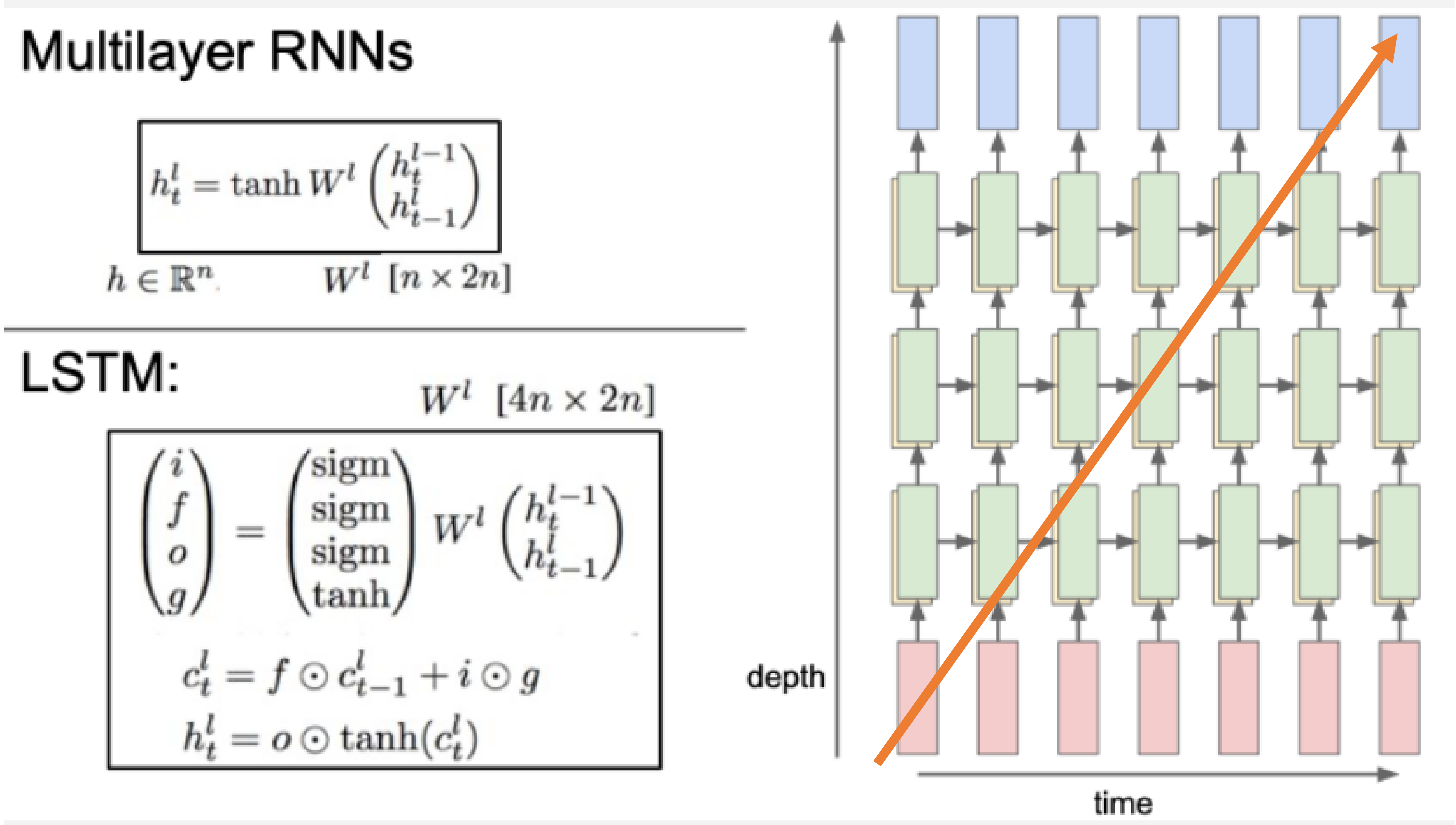
Zhu et al, "Visual 7W: Grounded Question Answering in Images", CVPR 2016
Figures from Zhu et al, copyright IEEE 2016. Reproduced for educational purposes.

EWHA EURON

LSTM Long Short-Term Memory



01 Vanilla RNN Gradient Flow



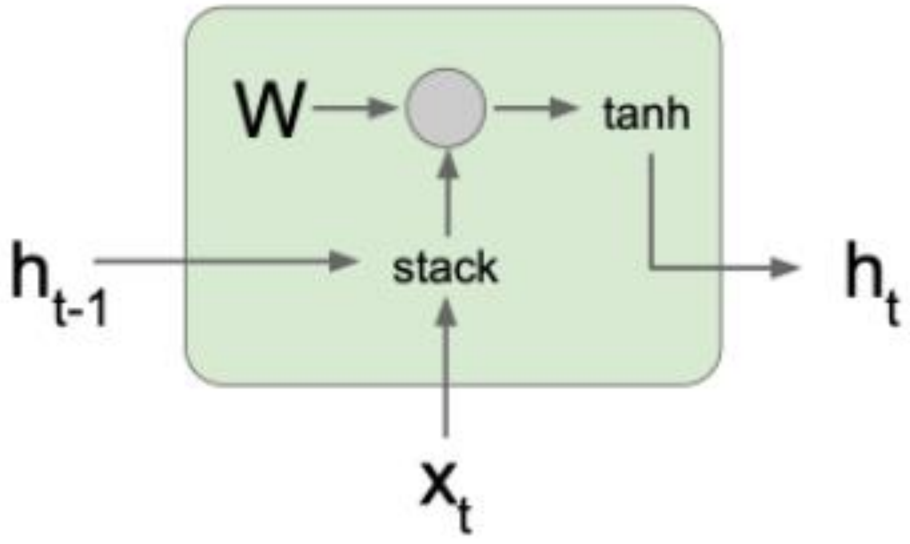
성능 향상

01 Vanilla RNN Gradient Flow

Vanilla RNN Gradient Flow

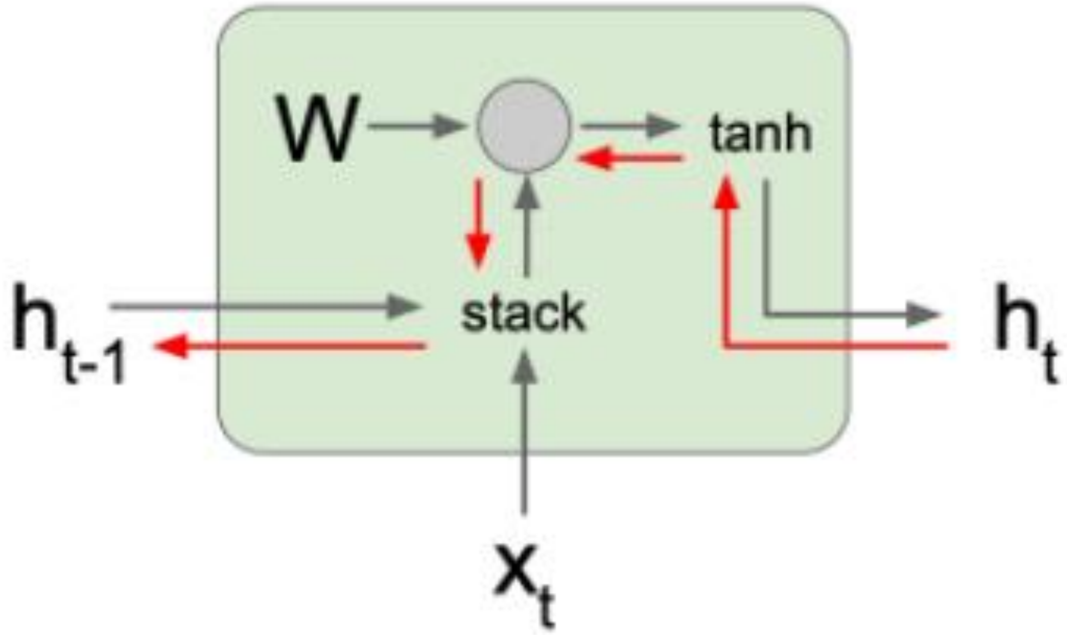
Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

일반적인 RNN을 잘 사용하지 않음
→ 학습 과정에서의 문제점

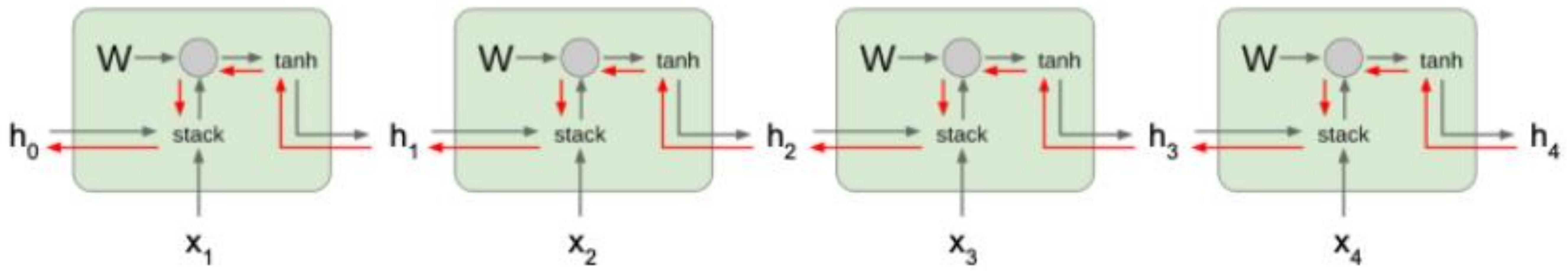


$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

Backpropagation from h_t to h_{t-1} multiplies by W (actually W_{hh}^T)



01 Vanilla RNN Gradient Flow



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1: Exploding gradients → **Gradient clipping**: Scale gradient if its norm is too big

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
  grad *= (threshold / grad_norm)
```

Largest singular value < 1:
Vanishing gradients

Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

Largest singular value < 1: Vanishing gradients → Change RNN architecture

02 LSTM

일반적인 RNN을 잘 사용하지 않음
→ 학습 과정에서의 문제점

→ 장기 의존성의 문제를 해결하기 위한 방책 → LSTM

Long Short Term Memory (LSTM)

Vanilla RNN

$$h_t = \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

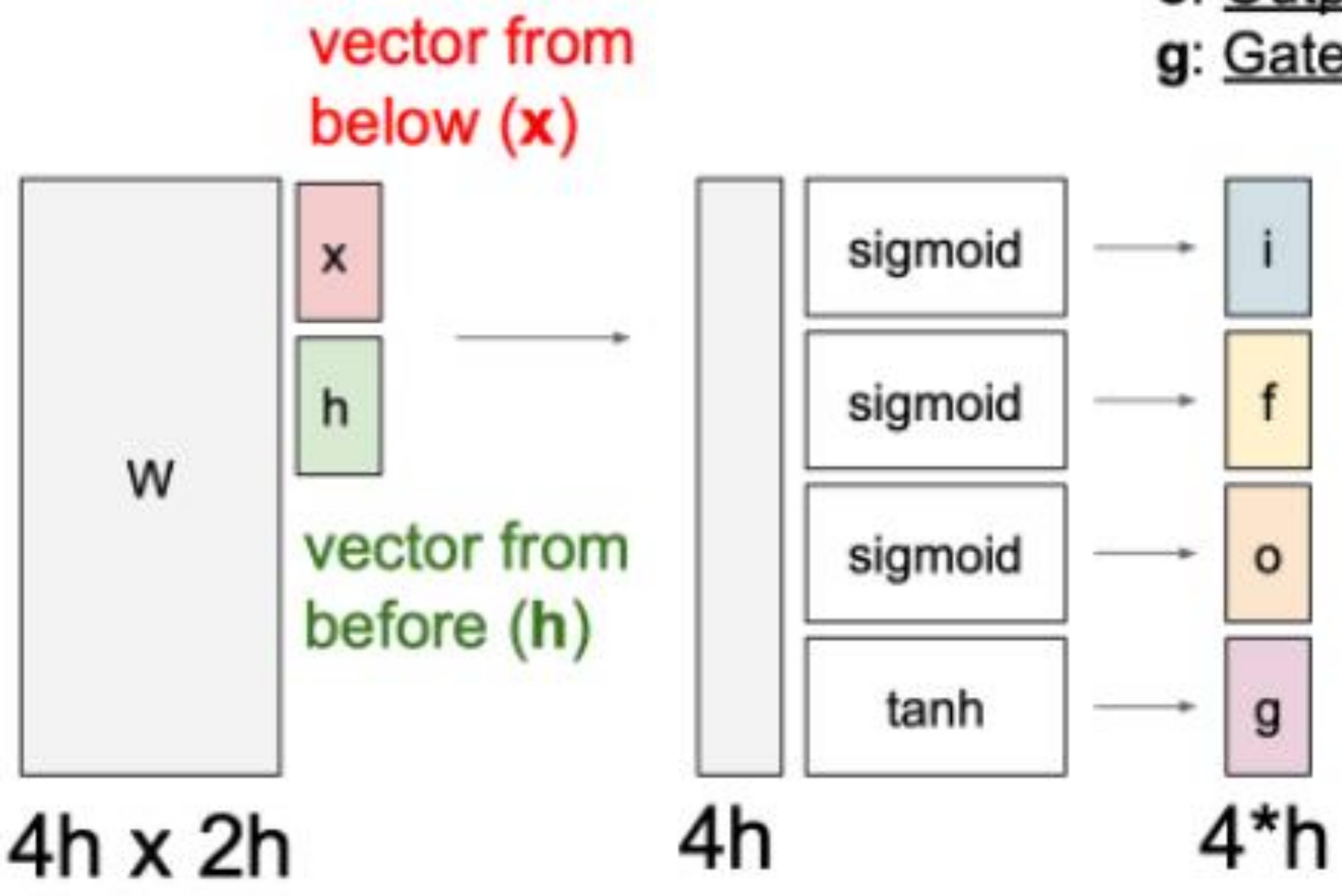
$$h_t = o \odot \tanh(c_t)$$

02 LSTM

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

i: Input gate, whether to write to cell
f: Forget gate, Whether to erase cell
o: Output gate, How much to reveal cell
g: Gate gate (?), How much to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

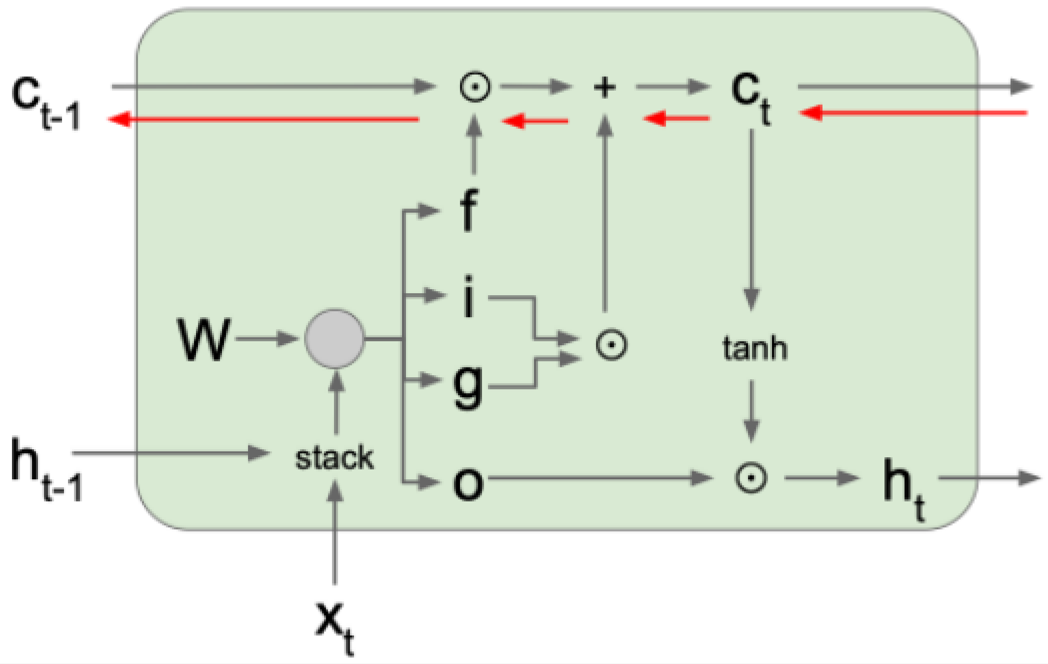
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

02 LSTM

Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]



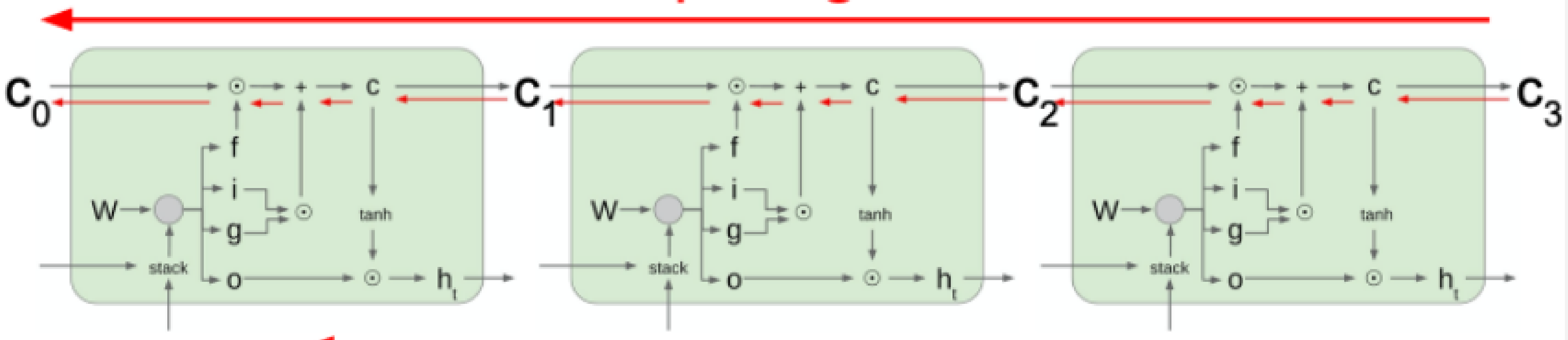
Backpropagation from c_t to c_{t-1} only elementwise multiplication by f, no matrix multiply by W

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

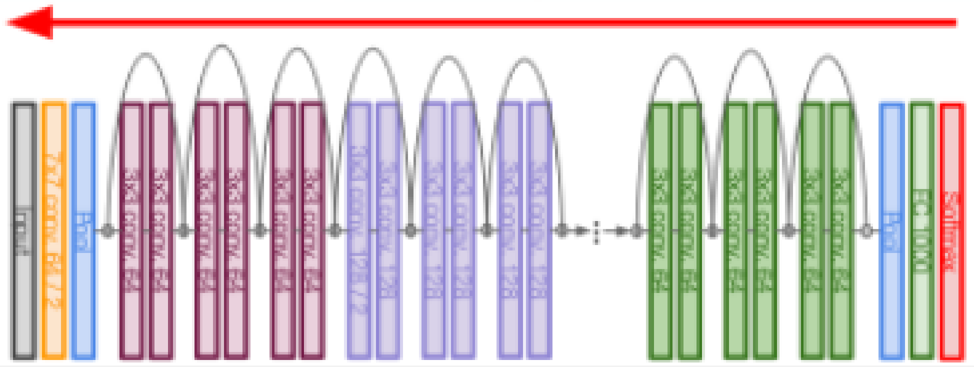
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Uninterrupted gradient flow!



Similar to ResNet!



In between:

Highway Networks

$$g = T(x, W_T)$$

$$y = g \odot H(x, W_H) + (1 - g) \odot x$$

Srivastava et al, "Highway Networks", ICML DL Workshop 2015

Other RNN Variants

GRU [*Learning phrase representations using rnn encoder-decoder for statistical machine translation,* Cho et al. 2014]

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$$
$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$$
$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h)$$
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

[*LSTM: A Search Space Odyssey,* Greff et al., 2015]

[*An Empirical Exploration of Recurrent Network Architectures,* Jozefowicz et al., 2015]

MUT1:

$$\begin{aligned} z &= \mathrm{sigm}(W_{\mathrm{xz}}x_t + b_{\mathrm{z}}) \\ r &= \mathrm{sigm}(W_{\mathrm{xr}}x_t + W_{\mathrm{hr}}h_t + b_{\mathrm{r}}) \\ h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + \tanh(x_t) + b_{\mathrm{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT2:

$$\begin{aligned} z &= \mathrm{sigm}(W_{\mathrm{xz}}x_t + W_{\mathrm{hz}}h_t + b_{\mathrm{z}}) \\ r &= \mathrm{sigm}(x_t + W_{\mathrm{hr}}h_t + b_{\mathrm{r}}) \\ h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh}x_t + b_{\mathrm{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT3:

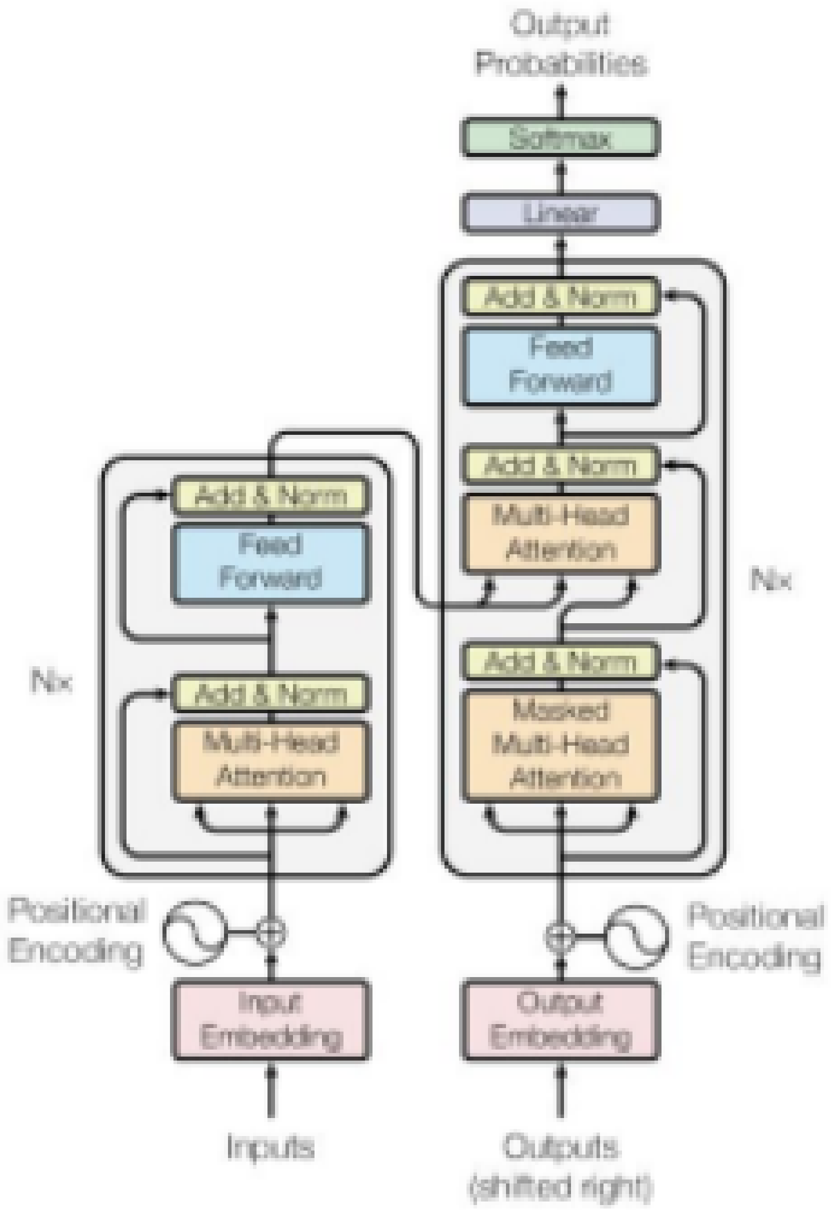
$$\begin{aligned} z &= \mathrm{sigm}(W_{\mathrm{xz}}x_t + W_{\mathrm{hz}}\tanh(h_t) + b_{\mathrm{z}}) \\ r &= \mathrm{sigm}(W_{\mathrm{xr}}x_t + W_{\mathrm{hr}}h_t + b_{\mathrm{r}}) \\ h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh}x_t + b_{\mathrm{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

Recently in Natural Language Processing… New paradigms for reasoning over sequences

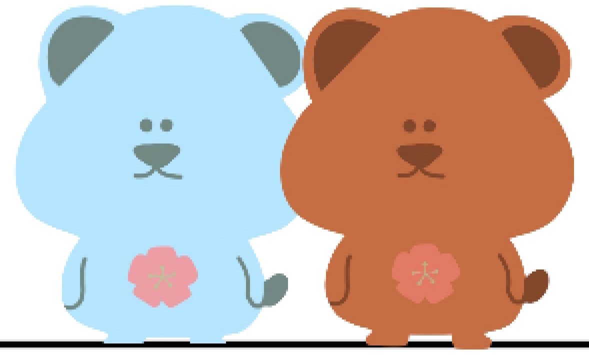
["Attention is all you need", Vaswani et al., 2018]

- New "Transformer" architecture no longer processes inputs sequentially; instead it can operate over inputs in a sequence in parallel through an attention mechanism

- Has led to many state-of-the-art results and pre-training in NLP, for more interest see e.g.
  - "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", Devlin et al., 2018
  - OpenAI GPT-2, Radford et al., 2018



THANK YOU



EWHA
EURON