CS224n LECTURE 7

RNN / LSTM / GRU

UOS STAT NLP Seminar
Kang Changgu

Vanilla RNN

RNN (= Recurrent Neural Network)

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

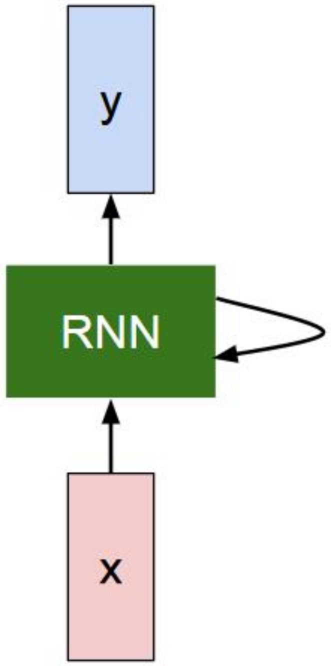
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

old state

input vector at some time step

some function with parameters W

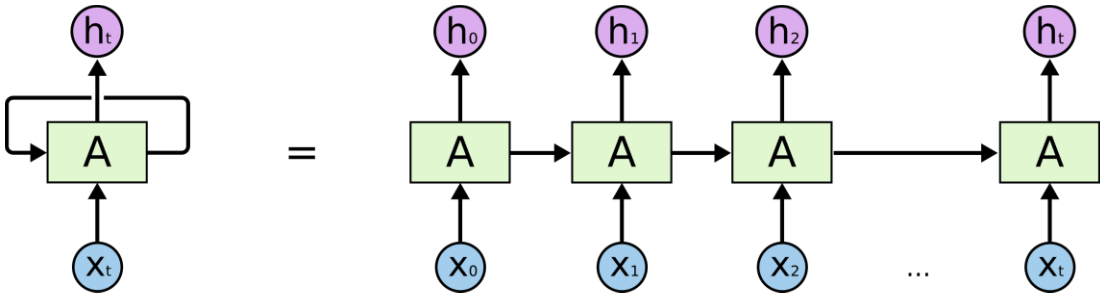


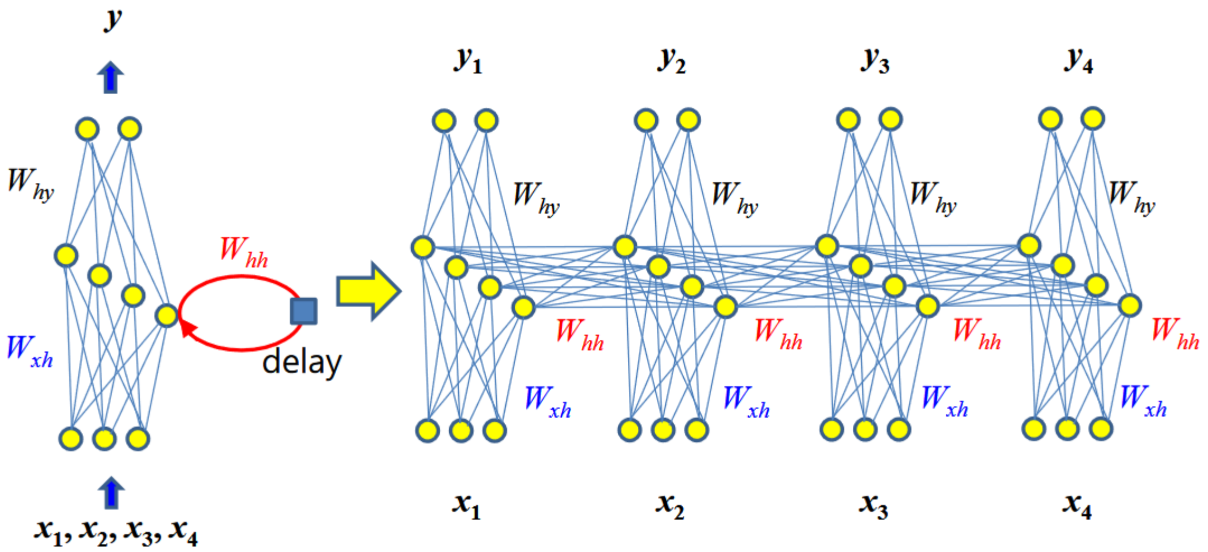
$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$y_t = W_{hy} h_t$$

Unfolding

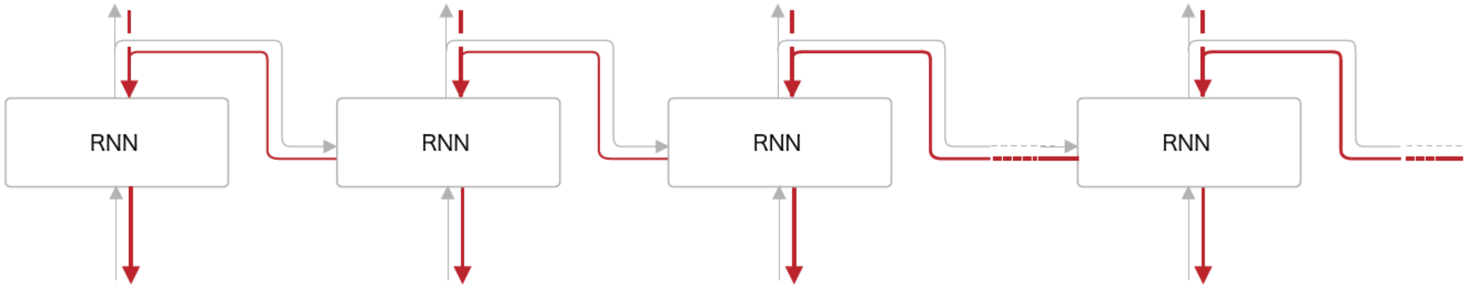
RNN을 시간 축을 따라 unfolding 했을 때, 입력/은닉 상태를 시간 축을 따라 표현 가능

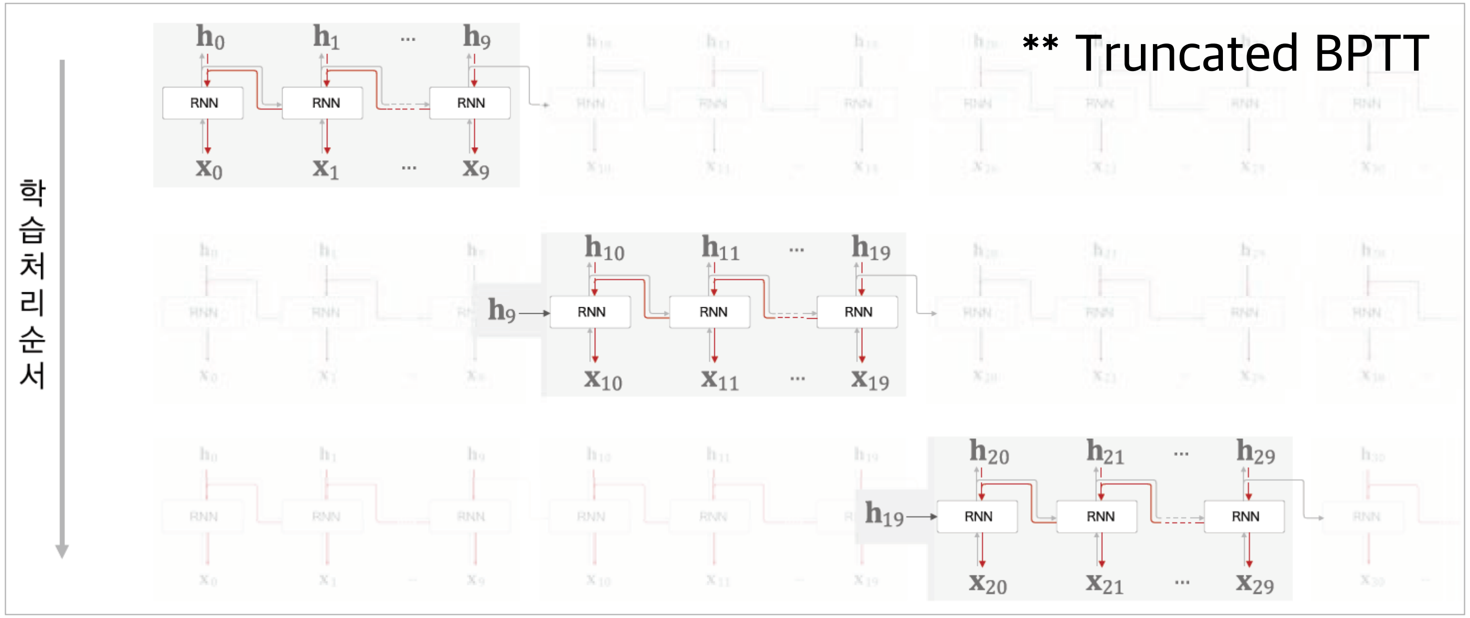




BPTT (= Backpropagation Through Time)

RNN을 시간 축을 따라 unfolding 했을 때, backpropagation 시 시간 역순으로 gradient 전파 표현 가능





End-to-end RNN

$\hat{\boldsymbol{y}}^{(4)} = P(\boldsymbol{x}^{(5)}|\text{the students opened their})$

output distribution

$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$

hidden states

$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$

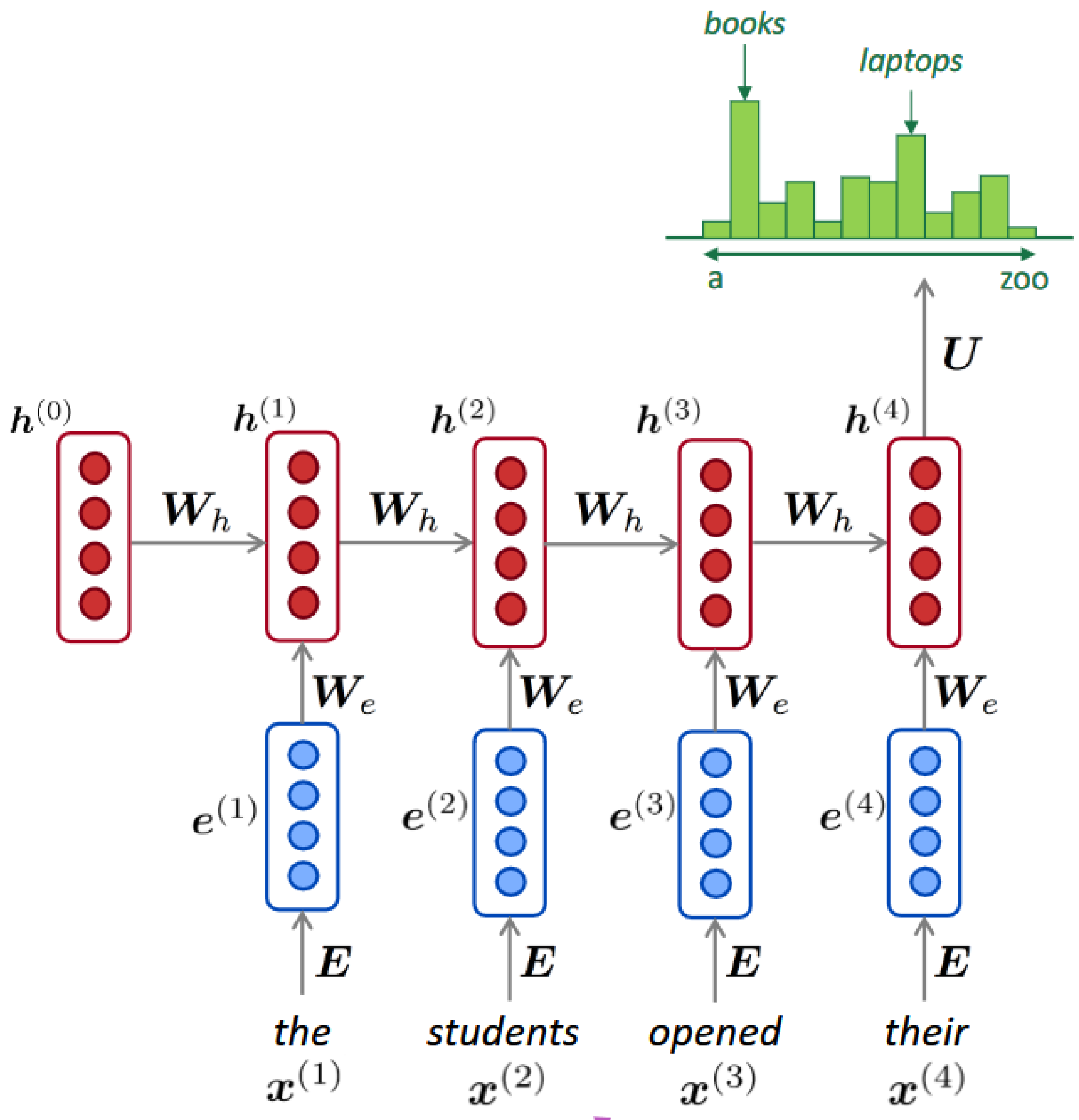
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$

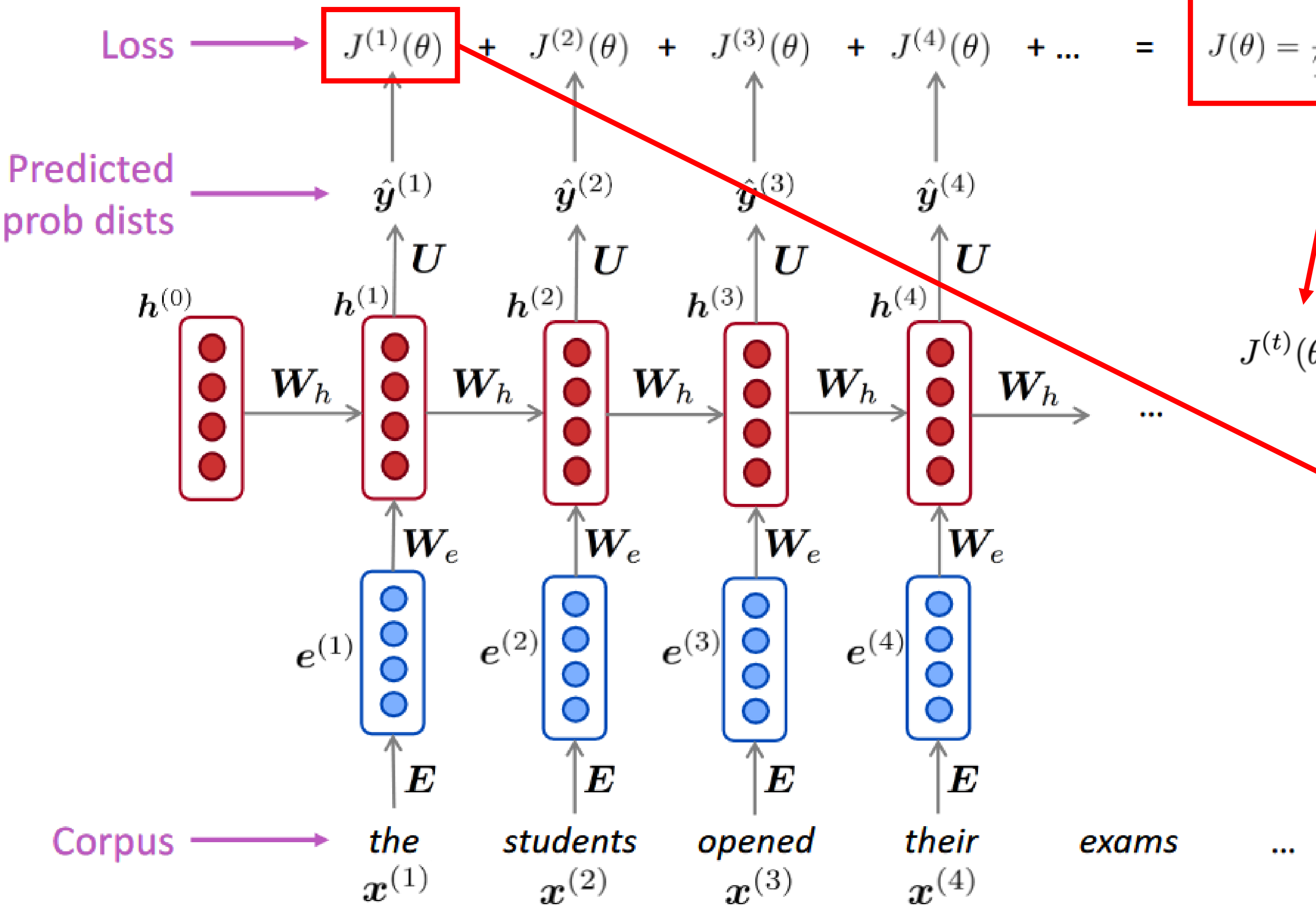
words / one-hot vectors

$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$



***Note**: this input sequence could be much longer, but this slide doesn't have space!*

Training RNN



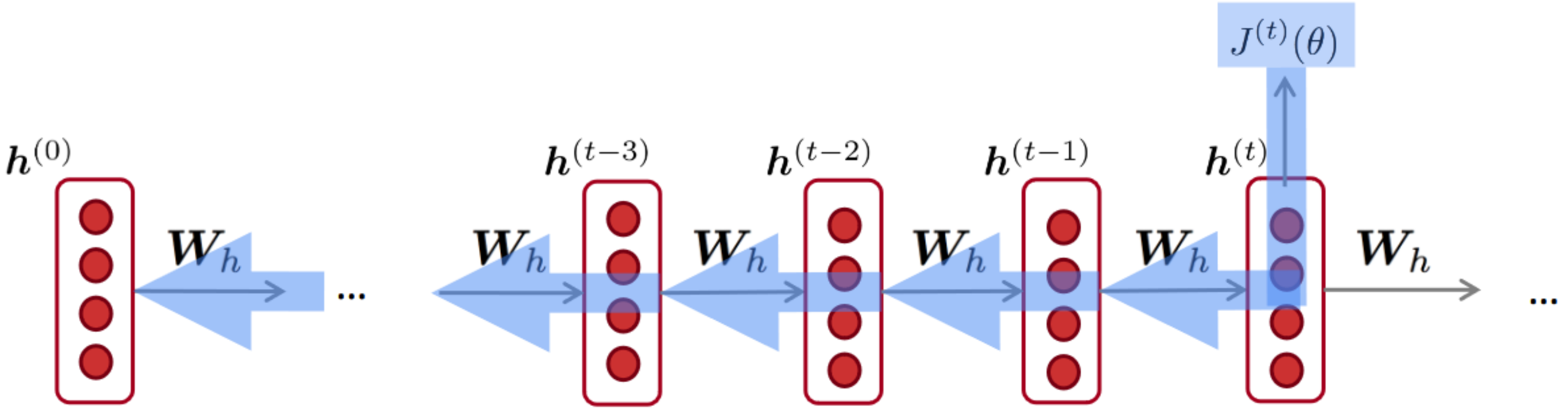
$$\boxed{J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)}$$

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

Backpropagation for RNN

BPTT : 시간 역순으로 각 hidden state를 따라 gradient 전달



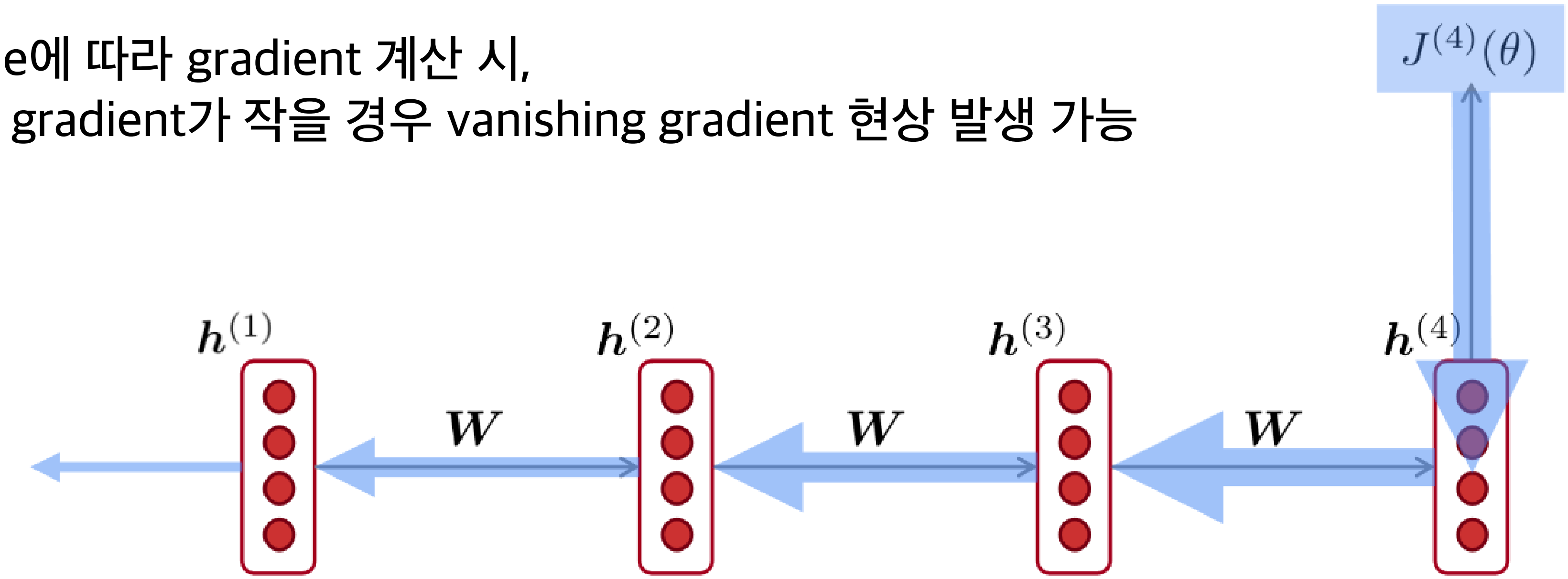
Question: What's the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix $\boldsymbol{W}_h$?

Answer: $$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\bigg|_{(i)}$$

“The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears”

Vanishing gradient

chain rule에 따라 gradient 계산 시,
곱해지는 gradient가 작을 경우 vanishing gradient 현상 발생 가능



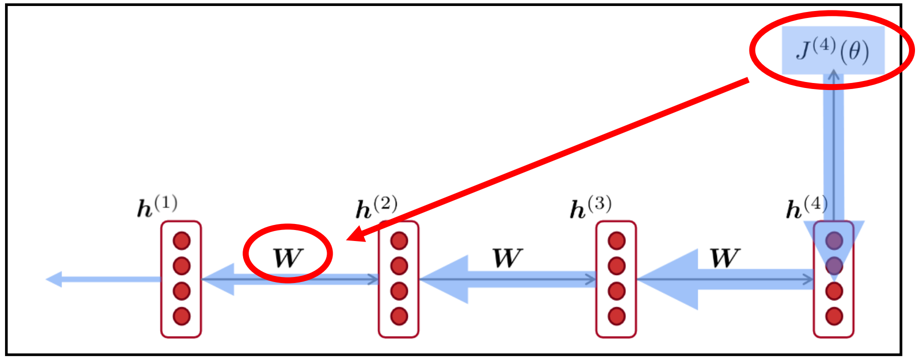
$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \boxed{\frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}}} \times \boxed{\frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}}} \times \boxed{\frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$

What happens if these are small?

Vanishing gradient problem:
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

Vanishing gradient (proof)

https://mmuratarat.github.io/2019-02-07/bptt-of-rnn



$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W^T \times diag[f'(j_{j-1})]$$

h_j을 h_{j-1}에만 partial derivative -> Jacobian Matrix

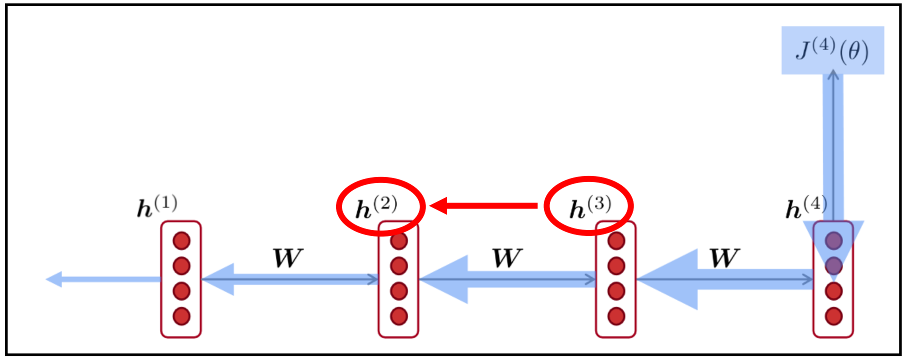
h_j는 $f(X_j, h_{j-1})$이므로 chain rule 적용

$$\frac{\partial h_j}{\partial h_{j-1}} = [\frac{\partial h_j}{\partial h_{j-1,1}} \cdots \frac{\partial h_j}{\partial h_{j-1,D_n}}] = \begin{bmatrix} \frac{\partial h_{j,1}}{\partial h_{j-1,1}} & \cdot & \cdot & \cdot & \frac{\partial h_{j,1}}{\partial h_{j-1,D_n}} \\ \cdot & \cdot & & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & & \cdot & \cdot \\ \frac{\partial h_{j,D_n}}{\partial h_{j-1,1}} & \cdot & \cdot & \cdot & \frac{\partial h_{j,D_n}}{\partial h_{j-1,D_n}} \end{bmatrix}$$

좌측의 Jacobian Matrix 계산 시,

$W^T \times diag[f'(h_{j-1})]$ 로 분해 가능 (= W^T의 각 행에 $f'(h_{j-1,i})$ 곱)

Vanishing gradient (proof)



$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W^T \times diag[f'(j_{j-1})]$$

$$\| \frac{\partial h_j}{\partial h_{j-1}} \| \leq \| W^T \| \| diag[f'(h_{j-1})] \| \leq \beta_W \beta_h$$

L2 norm에 대하여 upper bound β_W, β_h로 둘 경우,

좌측과 같이 partial derivative의 upper bound 계산 가능

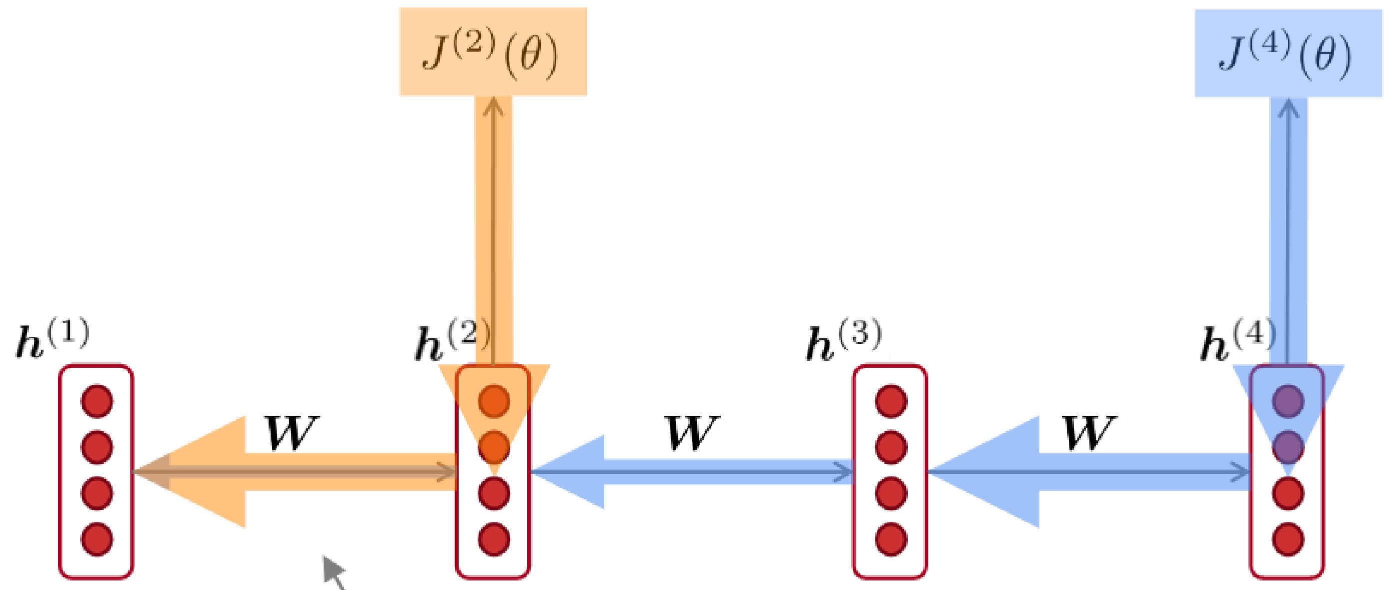
$$\| \frac{\partial h_t}{\partial h_k} \| = \| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \| \leq (\beta_W \beta_h)^{t-k}$$

$\beta_W \beta_h$가 1보다 작고 t-k가 충분히 큰 수일 경우, $(\beta_W \beta_h)^{t-k} \rightarrow 0$

$\beta_W \beta_h$가 1보다 크고 t-k가 충분히 큰 수일 경우, $(\beta_W \beta_h)^{t-k} \rightarrow large$

=> gradient vanishing/exploding 발생 가능

Effect of vanishing gradient



Gradient signal from faraway is lost because it's much smaller than gradient signal from close-by.

So model weights are updated only with respect to near effects, not long-term effects.

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}}$$

$$\downarrow$$

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W}$$

$$\bigcirc + \underbrace{\bigcirc + \cdots + \bigcirc}$$

gradient signal from close-by
-> influence ↑

gradient signal from faraway
-> influence ↓

Effect of vanishing gradient

If the gradient becomes vanishingly small over longer distances (step t to step $t+n$), then we can't tell whether:

1. There's no dependency between step t and $t+n$ in the data
2. We have wrong parameters to capture the true dependency between t and $t+n$

But if gradient is small, the model can't learn this dependency

- So the model is unable to predict similar long-distance dependencies at test time

model the dependency
= learn the connection between layers

Way to fix vanishing gradient

In a vanilla RNN, the hidden state is constantly being rewritten

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}\right)$$

-> not easy to preserve information one hidden state to another,
particularly putting it through the non-linear function

How about a RNN with separate memory?

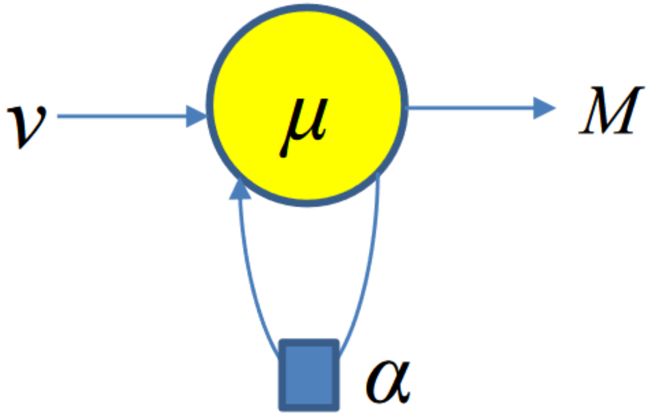
-> separate place to store information to preserve

LSTM

Gating/Leaky Unit

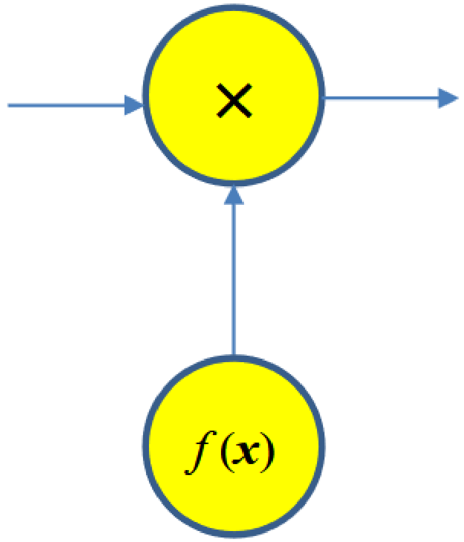
gating unit : controls the flow of information
leaky unit : determine how much information to preserve (including self-loop)

$$M^{(t)} \leftarrow \alpha\mu^{(t)} + (1 - \alpha)v^{(t)}$$



leaky unit

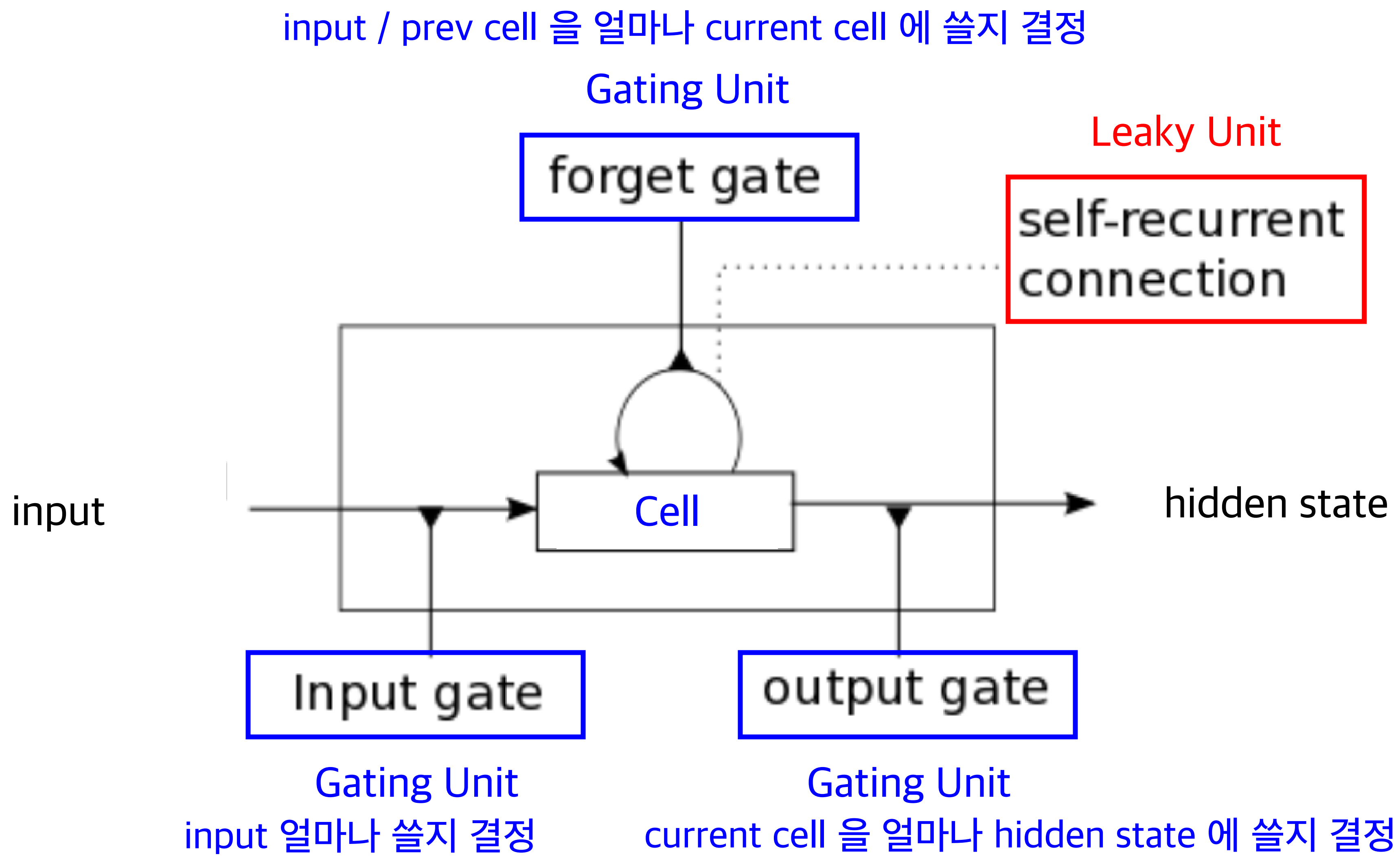
α = 1 일 경우, 정보를 보존
α = 0 일 경우, 정보를 잊음



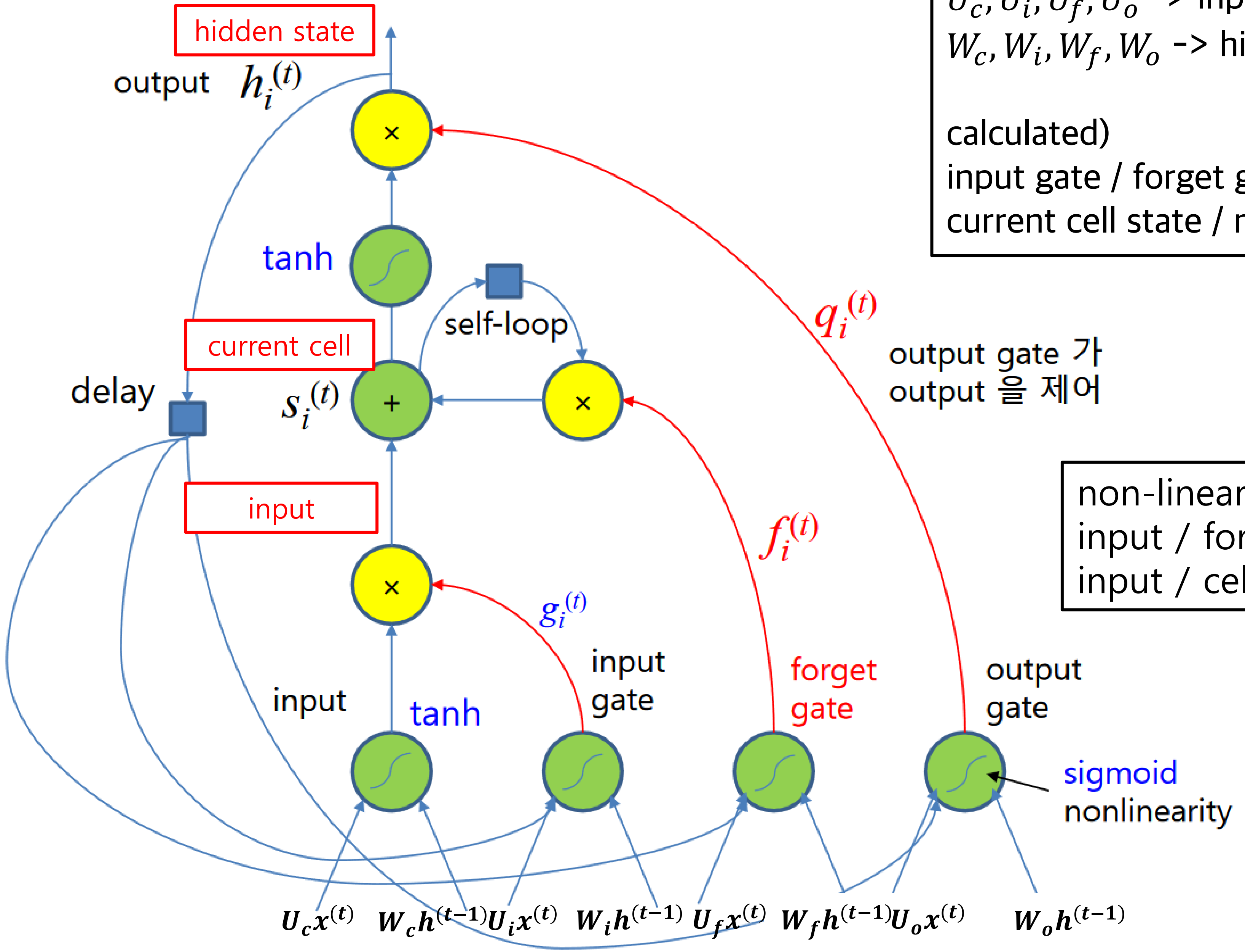
gating unit

f(x) = 1 일 경우, 정보를 보존
f(x) = 0 일 경우, 정보를 잊음

LSTM (= Long Short Term Memory)



LSTM



weight matrix)
U_c, U_i, U_f, U_o -> input $x^{(t)}$ 와 Affine 연산
W_c, W_i, W_f, W_o -> hidden state $h^{(t)}$ 와 Affine 연산

calculated)
input gate / forget gate / output gate (0~1 사이의 값)
current cell state / next cell state / hidden state

non-linearity function)
input / forget / output gate -> sigmoid
input / cell -> tanh

LSTM

We have a sequence of inputs $x^{(t)}$, and we will compute a sequence of hidden states $h^{(t)}$ and cell states $c^{(t)}$. On timestep t:

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

Sigmoid function: all gate values are between 0 and 1

$$\boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{W}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f \boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right)$$

$$\boldsymbol{i}^{(t)} = \sigma\left(\boldsymbol{W}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i \boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right)$$

$$\boldsymbol{o}^{(t)} = \sigma\left(\boldsymbol{W}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o \boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right)$$

New cell content: this is the new content to be written to the cell

Cell state: erase ("forget") some content from last cell state, and write ("input") some new cell content

Hidden state: read ("output") some content from the cell

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right)$$

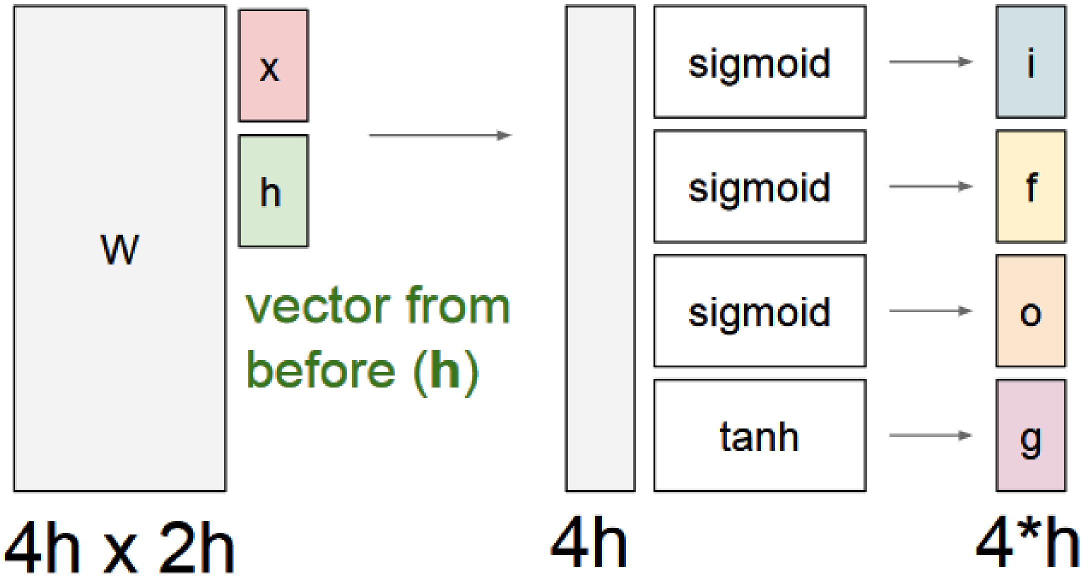
$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

Gates are applied using element-wise product

All these are vectors of same length n

Matrix Affine



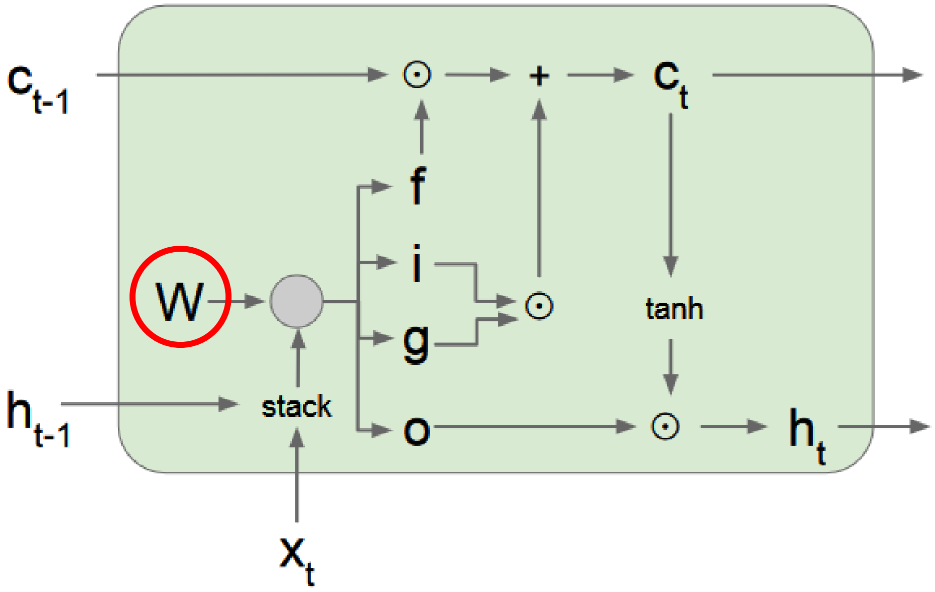
U_c, U_i, U_f, U_o / W_c, W_i, W_f, W_o를
하나의 weight matrix W로 통합

-> matrix size 4h x 2h

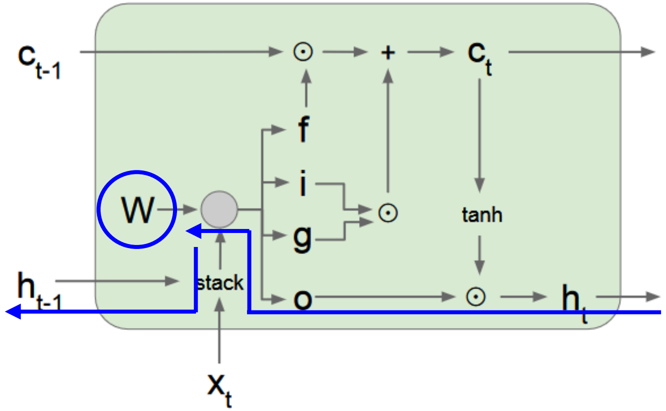
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

weight matrix W와 Affine 계산 후, hadamard product

$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$



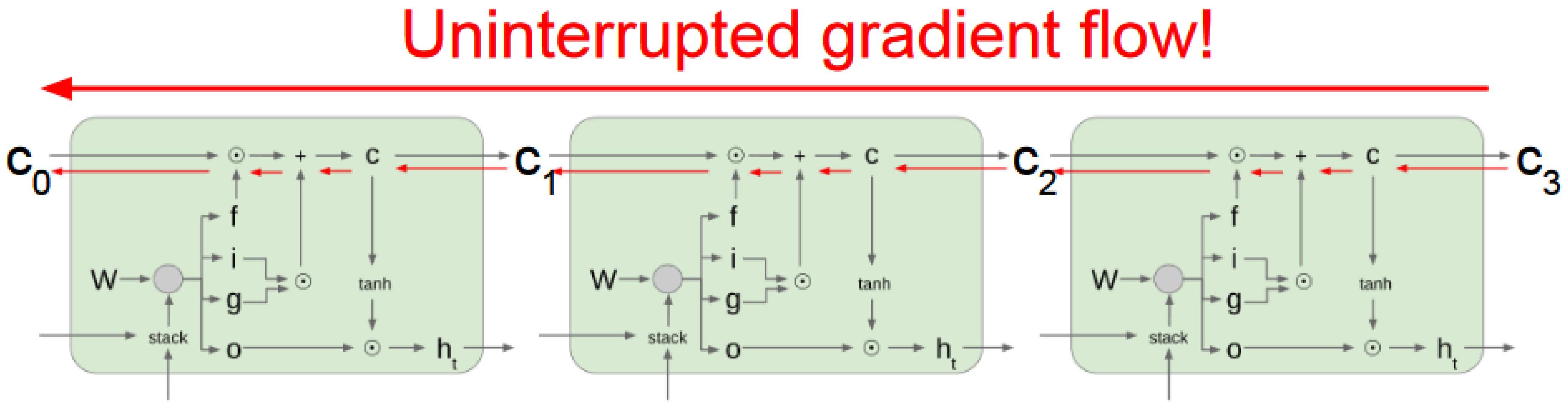
Gradient Flow



hidden state 를 통해 backpropagation 이루어질 경우,

weight matrix W에 대한 partial derivative 가

반복적으로 곱해지면서 vanishing gradient 발생 가능



cell state 를 통해 back propagation 이루어질 경우, weight matrix 의 interrupt 없이

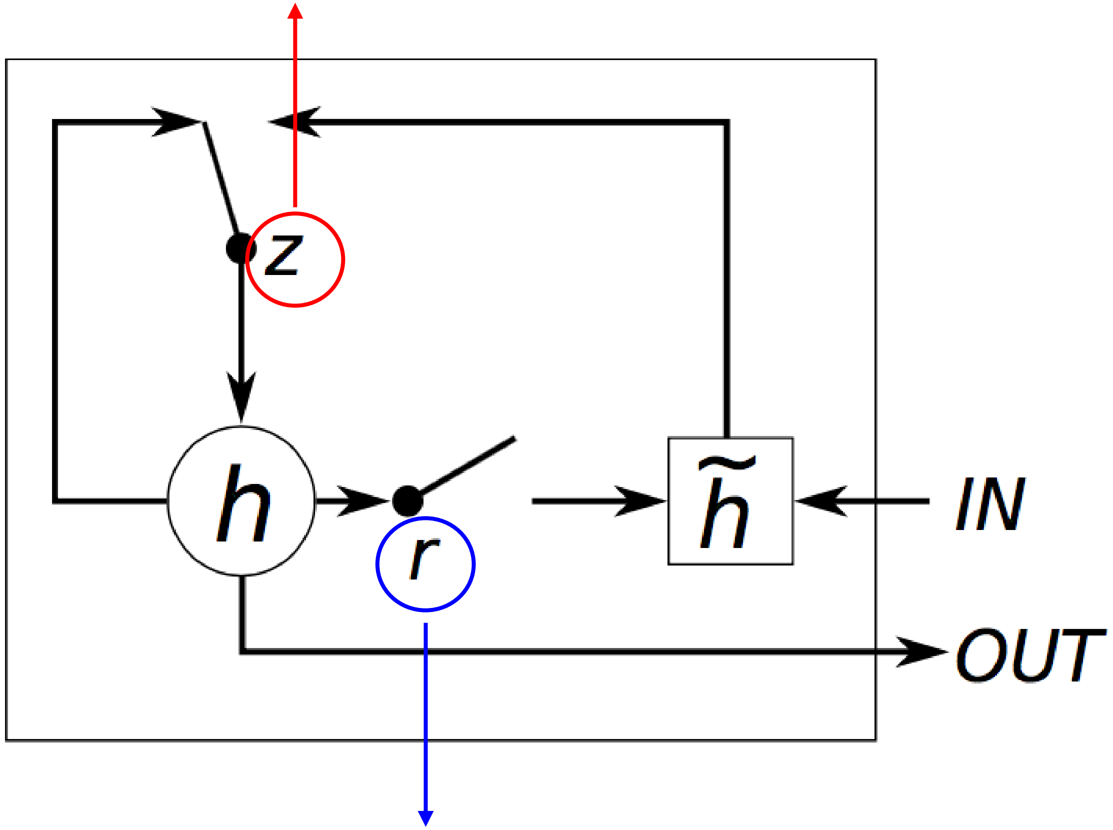
Hadamard product 에 대한 partial derivative 만 고려하며 gradient flow 가능 -> gradient vanishing ↓ ↓

GRU

GRU (= Gated Recurrent Unit)

update gate

current hidden state 로 업데이트 할지 =
prev hidden state 로 업데이트 하지 않을지 결정



reset gate
previous hidden state 를 보존할지 결정

LSTM과 달리, cell memory 가 없으므로 complexity ↓

long term memory 를 보존하고 싶다면 update gate z = 0

GRU

Update gate: controls what parts of hidden state are updated vs preserved

Reset gate: controls what parts of previous hidden state are used to compute new content

$$\boldsymbol{u}^{(t)} = \sigma\left(\boldsymbol{W}_u \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_u \boldsymbol{x}^{(t)} + \boldsymbol{b}_u\right)$$

$$\boldsymbol{r}^{(t)} = \sigma\left(\boldsymbol{W}_r \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_r \boldsymbol{x}^{(t)} + \boldsymbol{b}_r\right)$$

New hidden state content: reset gate selects useful parts of prev hidden state. Use this and current input to compute new hidden content.

Hidden state: update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

$$\tilde{\boldsymbol{h}}^{(t)} = \tanh\left(\boldsymbol{W}_h (\boldsymbol{r}^{(t)} \circ \boldsymbol{h}^{(t-1)}) + \boldsymbol{U}_h \boldsymbol{x}^{(t)} + \boldsymbol{b}_h\right)$$

$$\boldsymbol{h}^{(t)} = (1 - \boldsymbol{u}^{(t)}) \circ \boldsymbol{h}^{(t-1)} + \boldsymbol{u}^{(t)} \circ \tilde{\boldsymbol{h}}^{(t)}$$

How does this solve vanishing gradient?
Like LSTM, GRU makes it easier to retain info long-term (e.g. by setting update gate to 0)

Is vanishing/exploding gradient just a RNN problem?

No! It can be a problem for all neural architectures (including feed-forward and convolutional), especially deep ones.

For example:

- Residual connections aka "ResNet"
- Also known as skip-connections
- The identity connection preserves information by default
- This makes deep networks much easier to train

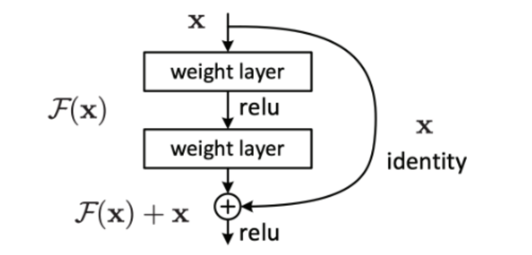


Figure 2. Residual learning: a building block.

For example:

- Dense connections aka "DenseNet"
- Directly connect each layer to all future layers!

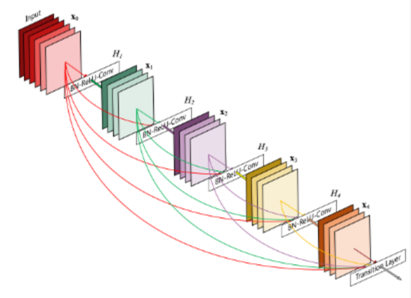
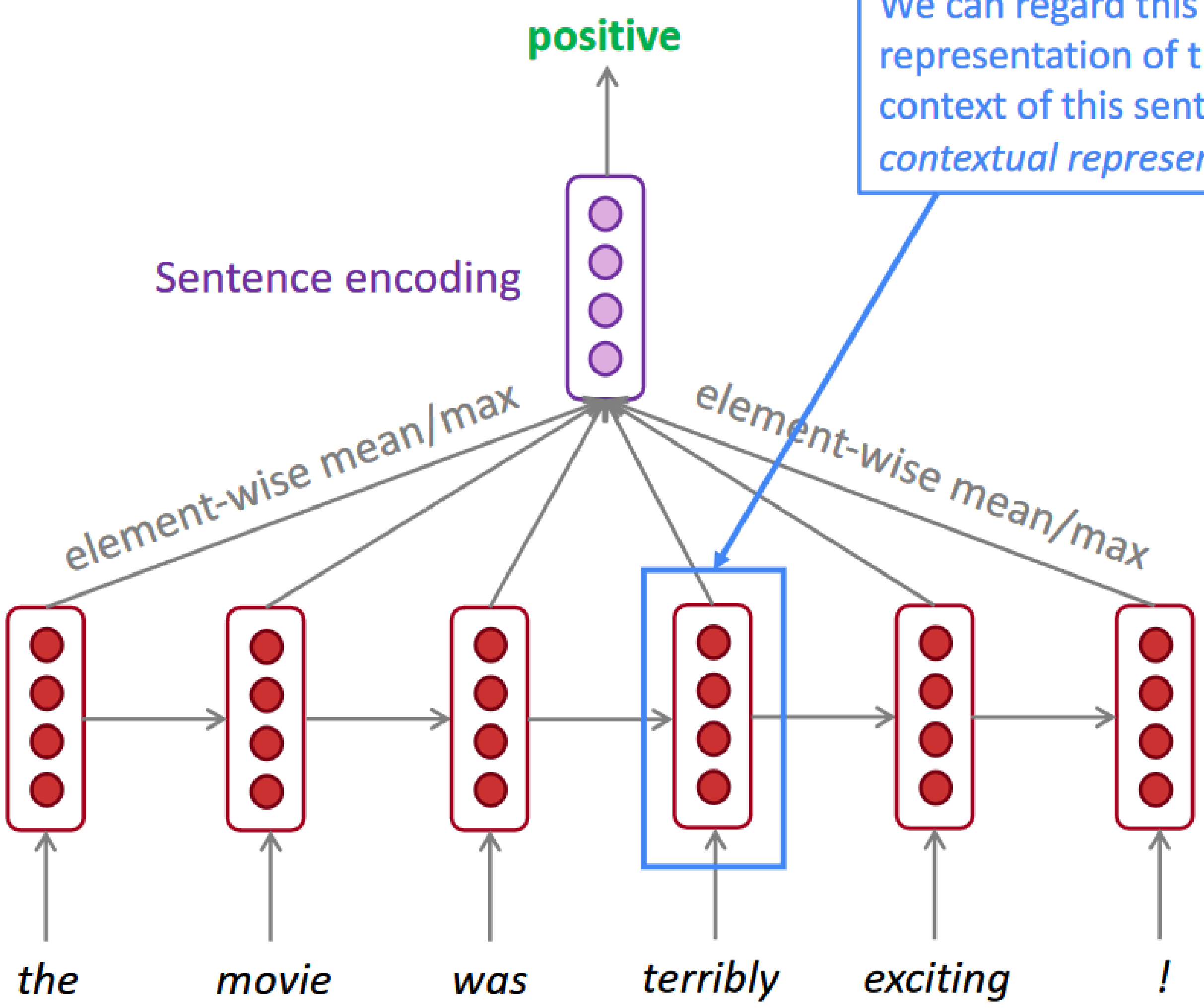


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

For example:

- Highway connections aka "HighwayNet"
- Similar to residual connections, but the identity connection vs the transformation layer is controlled by a dynamic gate
- Inspired by LSTMs, but applied to deep feedforward/convolutional networks

Bidirectional RNN



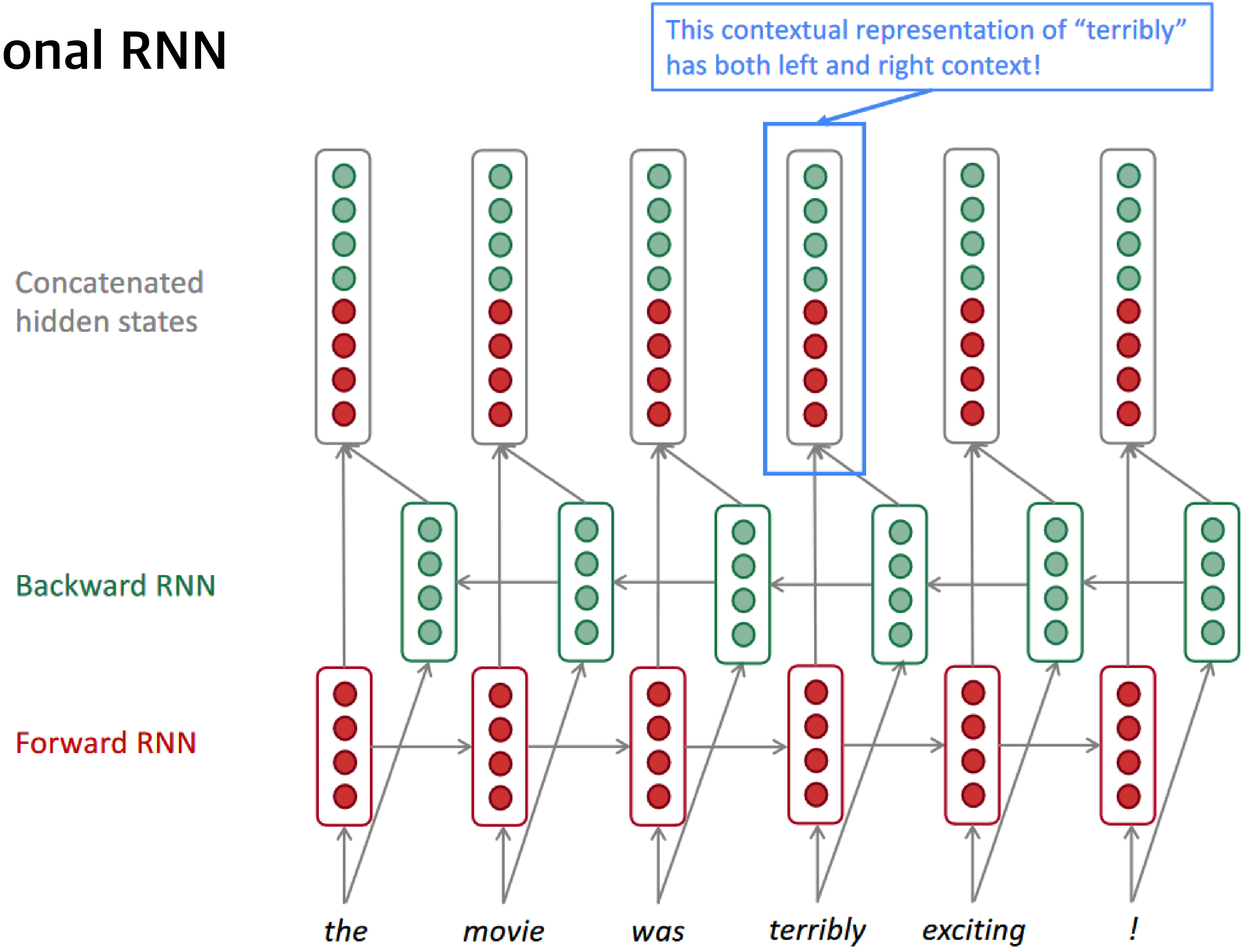
We can regard this hidden state as a representation of the word *"terribly"* in the context of this sentence. We call this a *contextual representation.*

These contextual representations only contain information about the *left* context (e.g. *"the movie was"*).

What about *right* context?

In this example, *"exciting"* is in the right context and this modifies the meaning of *"terribly"* (from negative to positive)

Bidirectional RNN



Multi-layer RNN

The hidden states from RNN layer *i* are the inputs to RNN layer *i+1*