RNN & LSTM & GRU

AILAB Seminar #11

김유리

Index

RNN

RNN
등장배경

기존 신경망의 한계

→ Sequence data를 처리하기 어려움

RNN
등장배경

① 여기서 말하는 기존 신경망이란?

기존 신경망의 한계

→ Sequence data를 처리하기 어려움

② Sequence data?

RNN
등장배경

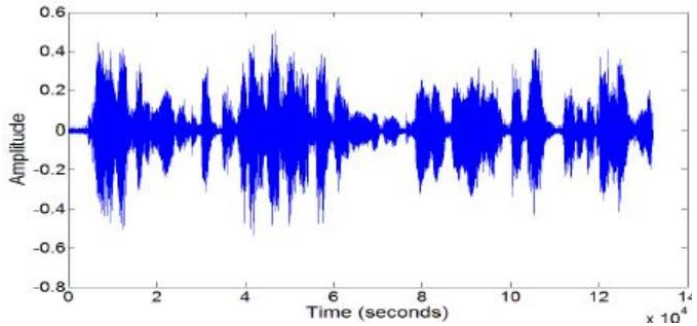
① 여기서 말하는 기존 신경망이란?

→ feed-forward neural network (입력 층 → 출력 층 한 방향으로만 흐르는)

RNN
등장배경

② Sequence data

→ 순서가 있는 데이터



음성 데이터

주식 데이터

This sentence is a sequence of words…

t=1 t=2 t=3

자연어 데이터

RNN
등장배경

기존 신경망의 한계

→ Sequence data를 처리하기 어려움

→ 연속적인(Sequential) 시계열(time series) 데이터에 적합한 모델인 RNN에 대해 알아보자!

RNN
RNN 알고리즘

RNN?

→ **Recurrent Neural Network**

RNN
RNN 알고리즘

RNN?

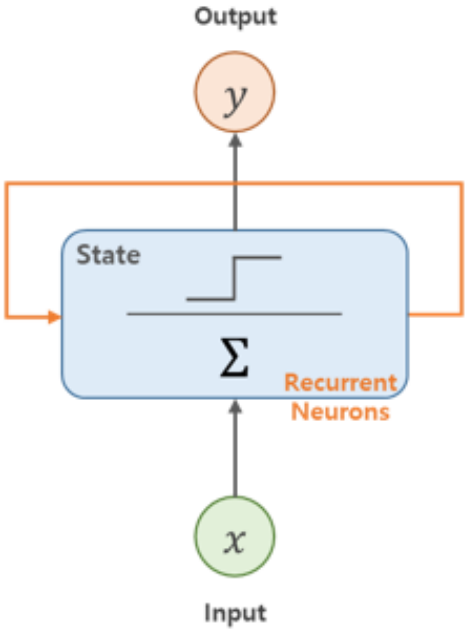
→ **Recurrent Neural Network**

RNN

RNN 알고리즘

Recurrent Neurons?

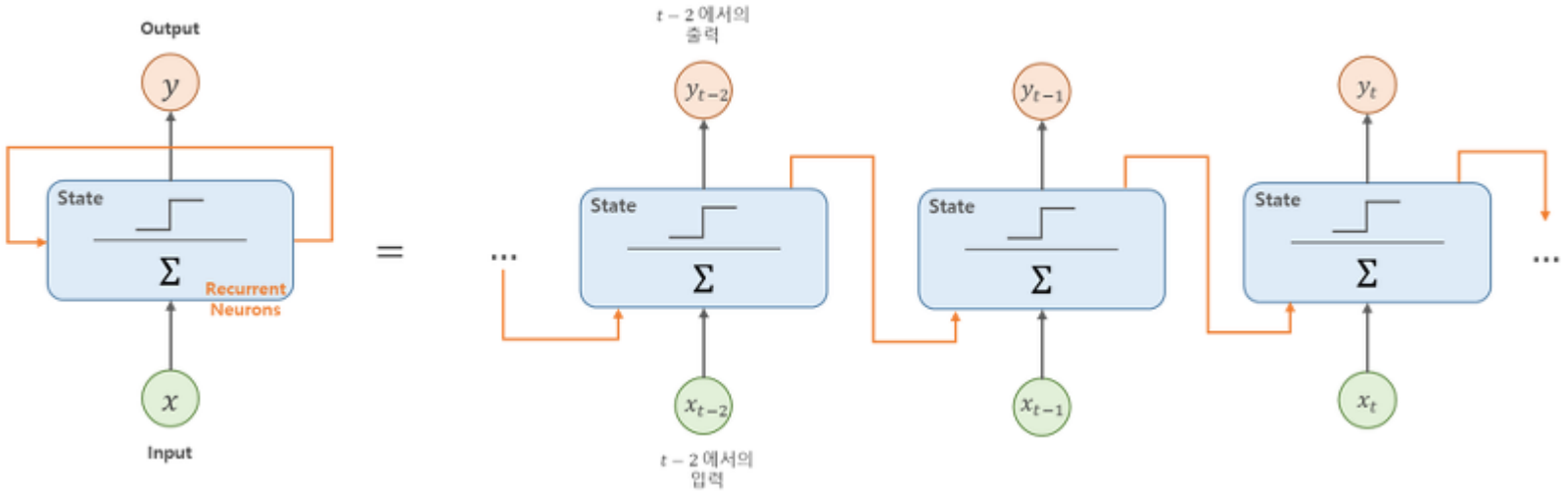
→ 입력(x)를 받아 출력(y)를 만들고, 이 출력을 다시 입력으로 받음



RNN
RNN 알고리즘

Recurrent **N**eural **N**etwork?

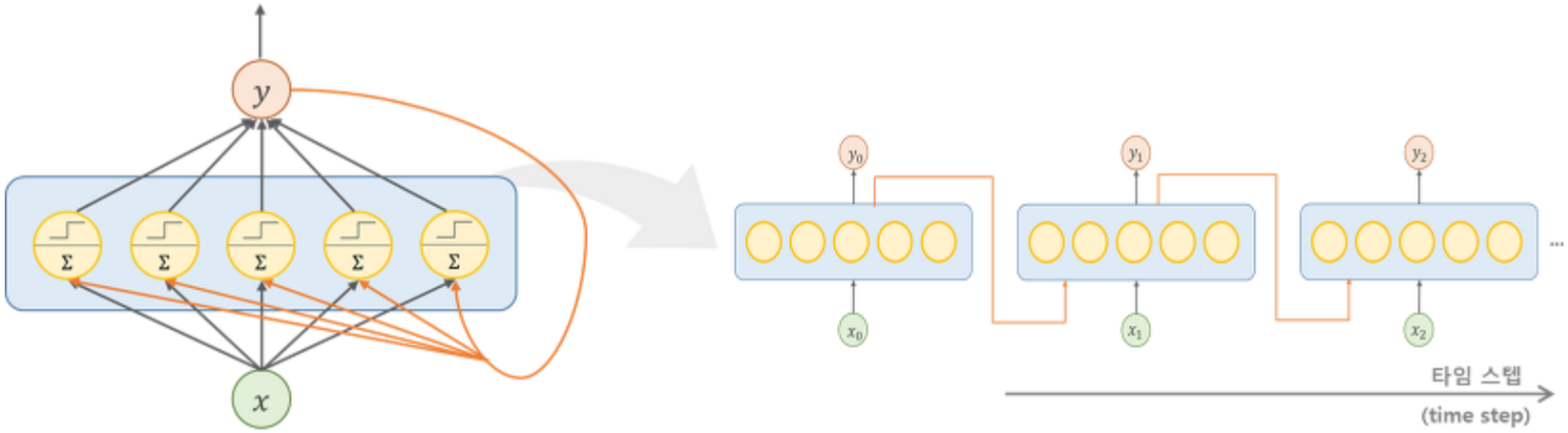
→ 타임 스텝(time step) t마다 Recurrent Neuron을 펼쳐서 time step 별 입력(x_t)과 출력(y_t)을 나타냄



RNN
RNN 알고리즘

Recurrent **N**eural **N**etwork의 **Layer**

→ 타임 스텝(time step) t마다 모든 뉴런은 입력 벡터(x_t)와 이전 타임 스텝의 출력 벡터 (y_{t-1})을 입력 받음



RNN
RNN 알고리즘

Recurrent **N**eural **N**etwork의 **Layer**

→ 각 뉴런은 두개의 가중치를 가짐 (입력 벡터 x_t를 위한 w_x, 이전 타임 스텝의 출력 벡터 y_{t-1}를 위한 w_y)

Layer 전체로 생각하면, 가중치 벡터 w_x, w_y를 행렬 W_x, W_y로 나타낼 수 있음

$$\mathbf{y}_t = \phi\left(\mathbf{W}_x^T \cdot \mathbf{x}_t + \mathbf{W}_y^T \cdot \mathbf{y}_{t-1} + \mathbf{b}\right)$$

타임 스텝 t에서의 미니 배치 입력을 X_t로 나타내면, 아래와 같이 Recurrent layer의 출력을 한번에 계산 가능

$$\begin{aligned}\mathbf{Y}_t &= \phi\left(\mathbf{X}_t \cdot \mathbf{W}_x + \mathbf{Y}_{t-1} \cdot \mathbf{W}_y + \mathbf{b}\right) \\ &= \phi\left(\begin{bmatrix}\mathbf{X}_t & \mathbf{Y}_{t-1}\end{bmatrix}\begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix} + \mathbf{b}\right)\end{aligned}$$

- $\mathbf{Y}_t$: 타임 스텝 t에서 미니배치에 있는 각 샘플(미니배치)에 대한 순환 층의 출력이며, $m \times n_{\text{neurons}}$ 행렬(m은 미니배치, n_{neurons}은 뉴런 수)
- $\mathbf{X}_t$: 모든 샘플의 입력값을 담고 있는 $m \times n_{\text{inputs}}$ 행렬 (n_{inputs}은 입력 특성 수)
- $\mathbf{W}_x$: 현재 타임 스텝 t의 입력에 대한 가중치를 담고 있는 $n_{\text{inputs}} \times n_{\text{neurons}}$ 행렬
- $\mathbf{W}_y$: 이전 타임 스텝 $t-1$ 의 출력에 대한 가중치를 담고 있는 $n_{\text{neurons}} \times n_{\text{neurons}}$ 행렬
- $\mathbf{b}$: 각 뉴런의 편향(bias)을 담고 있는 n_{neurons} 크기의 벡터

RNN
RNN 알고리즘

Recurrent Neural Network의 Layer

→ 아래 식에서 Y_t는 X_t와 Y_{t-1}의 함수 이므로 타임 스텝 $t = 0$에서부터 모든 입력에 대한 함수가 됨

(첫번째 타임 스텝인 $t = 0$에서는 이전 출력이 없기 때문에 일반적으로 0으로 초기화)

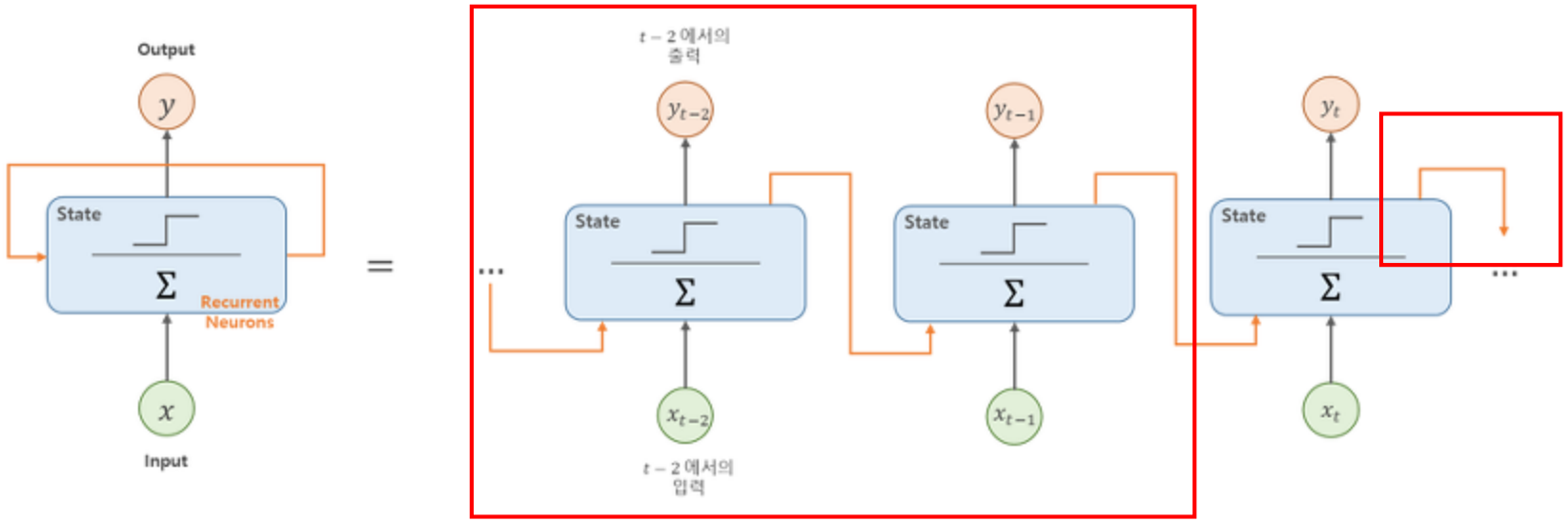
$$\mathbf{Y}_t = \phi\left(\mathbf{X}_t \cdot \mathbf{W}_x + \mathbf{Y}_{t-1} \cdot \mathbf{W}_y + \mathbf{b}\right)$$
$$= \phi\left(\begin{bmatrix}\mathbf{X}_t & \mathbf{Y}_{t-1}\end{bmatrix}\begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix} + \mathbf{b}\right)$$

- $\mathbf{Y}_t$: 타임 스텝 t에서 미니배치에 있는 각 샘플(미니배치)에 대한 순환 층의 출력이며, $m \times n_{\text{neurons}}$ 행렬(m은 미니배치, n_{neurons}은 뉴런 수)
- $\mathbf{X}_t$: 모든 샘플의 입력값을 담고 있는 $m \times n_{\text{inputs}}$ 행렬 (n_{inputs}은 입력 특성 수)
- $\mathbf{W}_x$: 현재 타임 스텝 t의 입력에 대한 가중치를 담고 있는 $n_{\text{inputs}} \times n_{\text{neurons}}$ 행렬
- $\mathbf{W}_y$: 이전 타임 스텝 $t-1$ 의 출력에 대한 가중치를 담고 있는 $n_{\text{neurons}} \times n_{\text{neurons}}$ 행렬
- $\mathbf{b}$: 각 뉴런의 편향(bias)을 담고 있는 n_{neurons} 크기의 벡터

RNN
RNN 알고리즘

Memory Cell

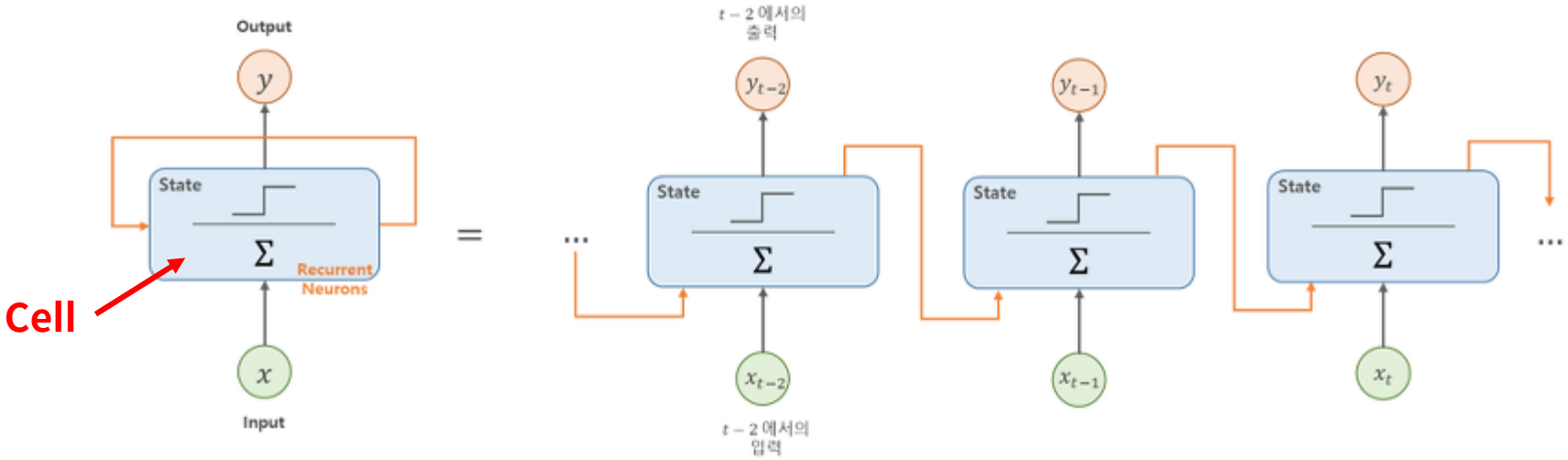
→ 타임 스텝 t에서 뉴런의 출력은 이전 타임 스텝의 모든 입력에 대한 함수 : **Memory**라고 부름



RNN
RNN 알고리즘

Memory Cell

→ 타임 스텝에 걸쳐 어떠한 상태를 보존하는 신경망의 구성 요소 : **Memory Cell** 또는 **Cell**이라고 부름



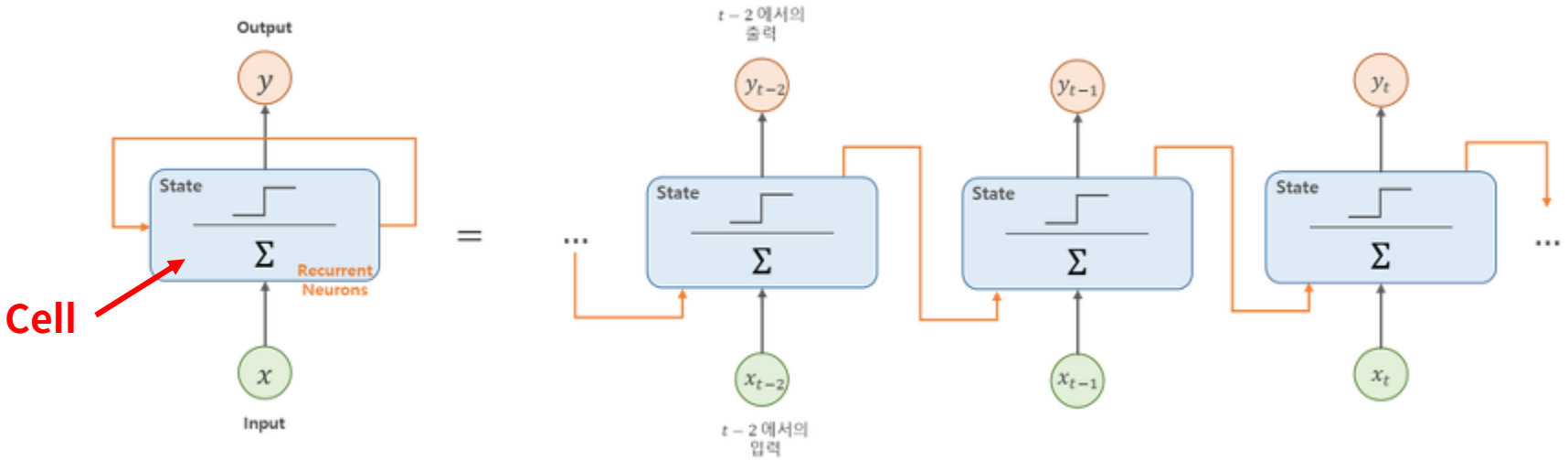
RNN
RNN 알고리즘

Memory Cell

→ 타임 스텝 t에서의 Cell의 상태 : h_t

(h = hidden, 타임스텝에서의 입력과 이전 타입 스텝의 상태에 대한 함수)

$$\mathbf{h}_t = f\left(\mathbf{h}_{t-1}, \mathbf{x}_t\right)$$

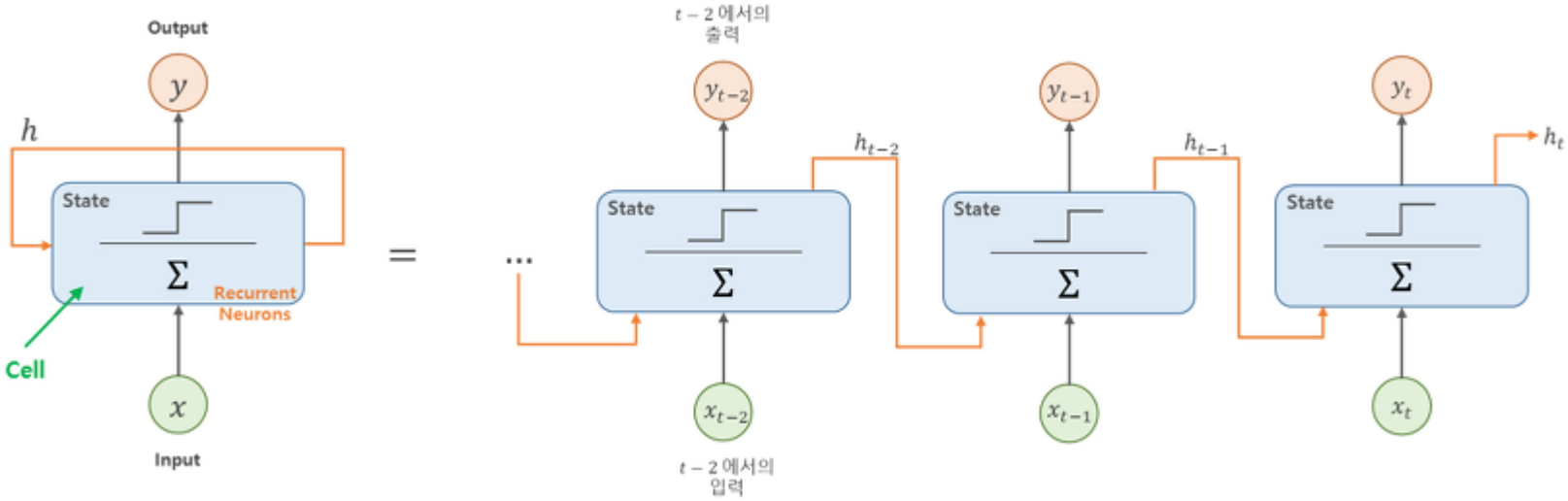


RNN
RNN 알고리즘

Memory Cell

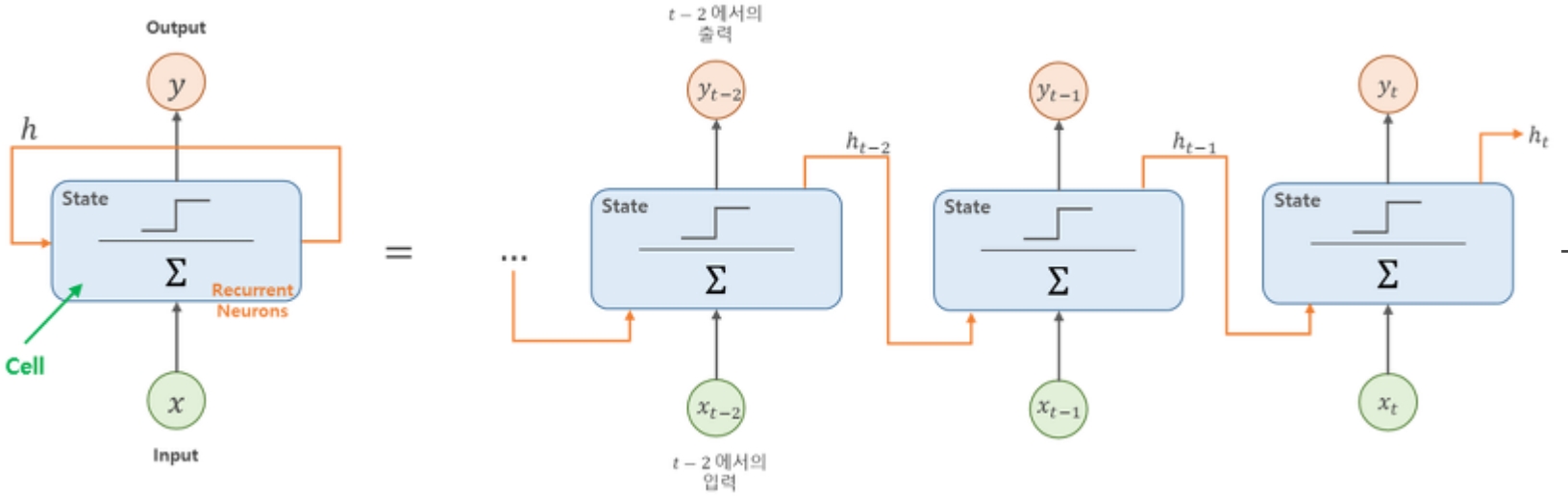
→ 타임 스텝 t에서의 hidden state(h_t)와 출력(y_t)가 구분됨

(입력으로는 (h_t)가 들어감)



RNN

RNN 알고리즘



수식

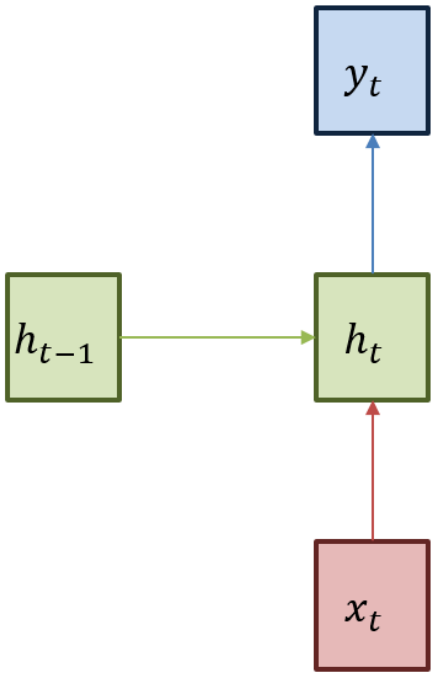
$$\mathbf{h}_t = \tanh(\mathbf{X}_t \cdot \mathbf{W}_x + \mathbf{h}_{t-1} \cdot \mathbf{W}_h + \mathbf{b})$$

$$= \tanh\left(\begin{bmatrix} \mathbf{X}_t & \mathbf{h}_{t-1} \end{bmatrix} \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_h \end{bmatrix} + \mathbf{b} \right)$$

$$\mathbf{Y}_t = \mathbf{W}_y^T \cdot \mathbf{h}_t$$

RNN

RNN 알고리즘



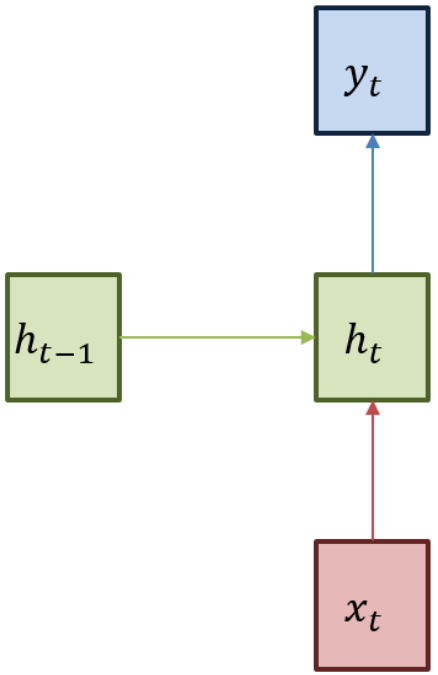
$$y_t = W_{hy} h_t + b_y$$

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

→ 각 타임 스텝 별로 W_{hh}, W_{xh}, W_{hy}는 모두 공유

RNN

RNN 알고리즘



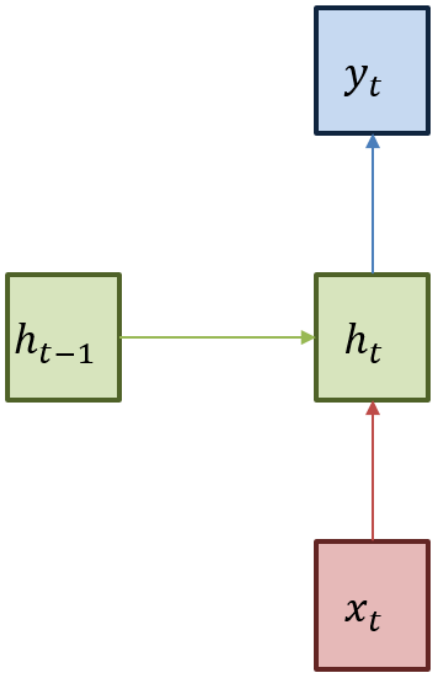
$$y_t = W_{hy} h_t + b_y$$

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

왜 activation func으로 tanh를 쓰나요?

RNN

RNN 알고리즘



$$y_t = W_{hy} h_t + b_y$$

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

왜 activation func으로 tanh를 쓰나요?

→ sigmoid대신 사용해서 & 문장이 길어질 수록

Gradient Vanishing 문제를 조금이라도 해결하기 위해

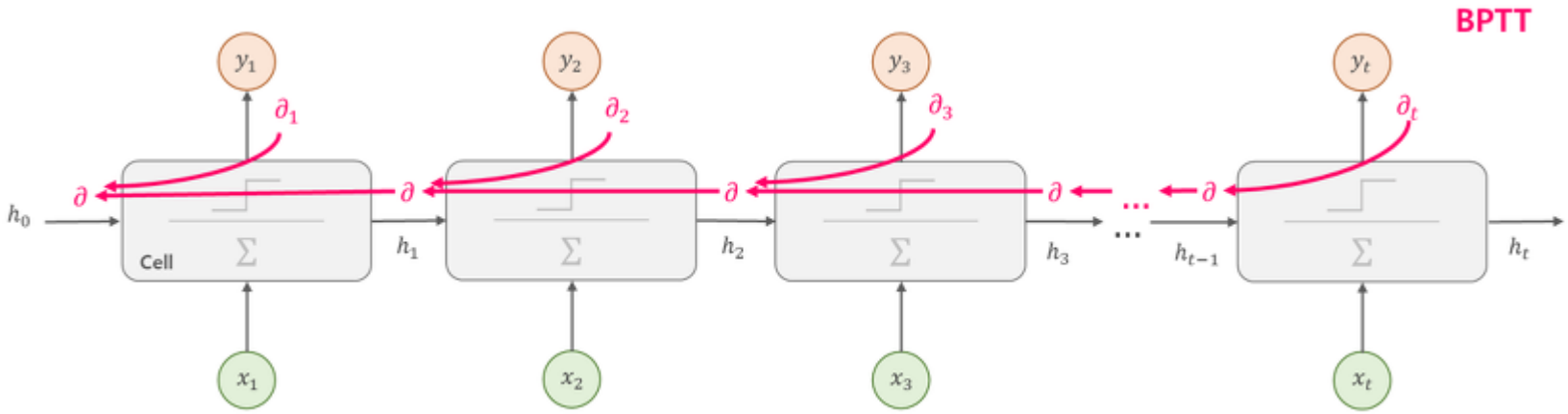
RNN

RNN 학습시키기

BPTT (Backpropagation Through Time)

RNN은 기존 신경망의 역전파(backprop)와는 달리 타임 스텝 별로 네트워크를 펼친 다음, 역전파 알고리즘을 사용

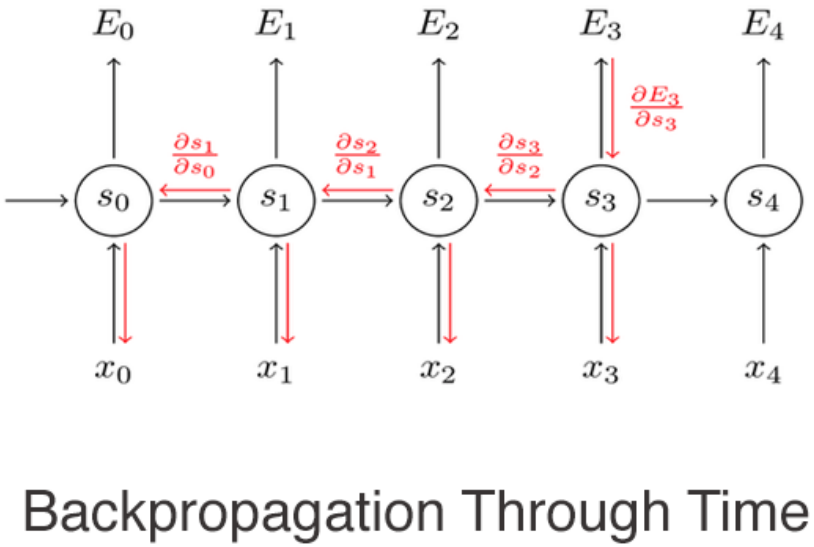
→ BPTT라고 함



RNN

RNN 학습시키기

BPTT (Backpropagation Through Time)



Backpropagation Through Time

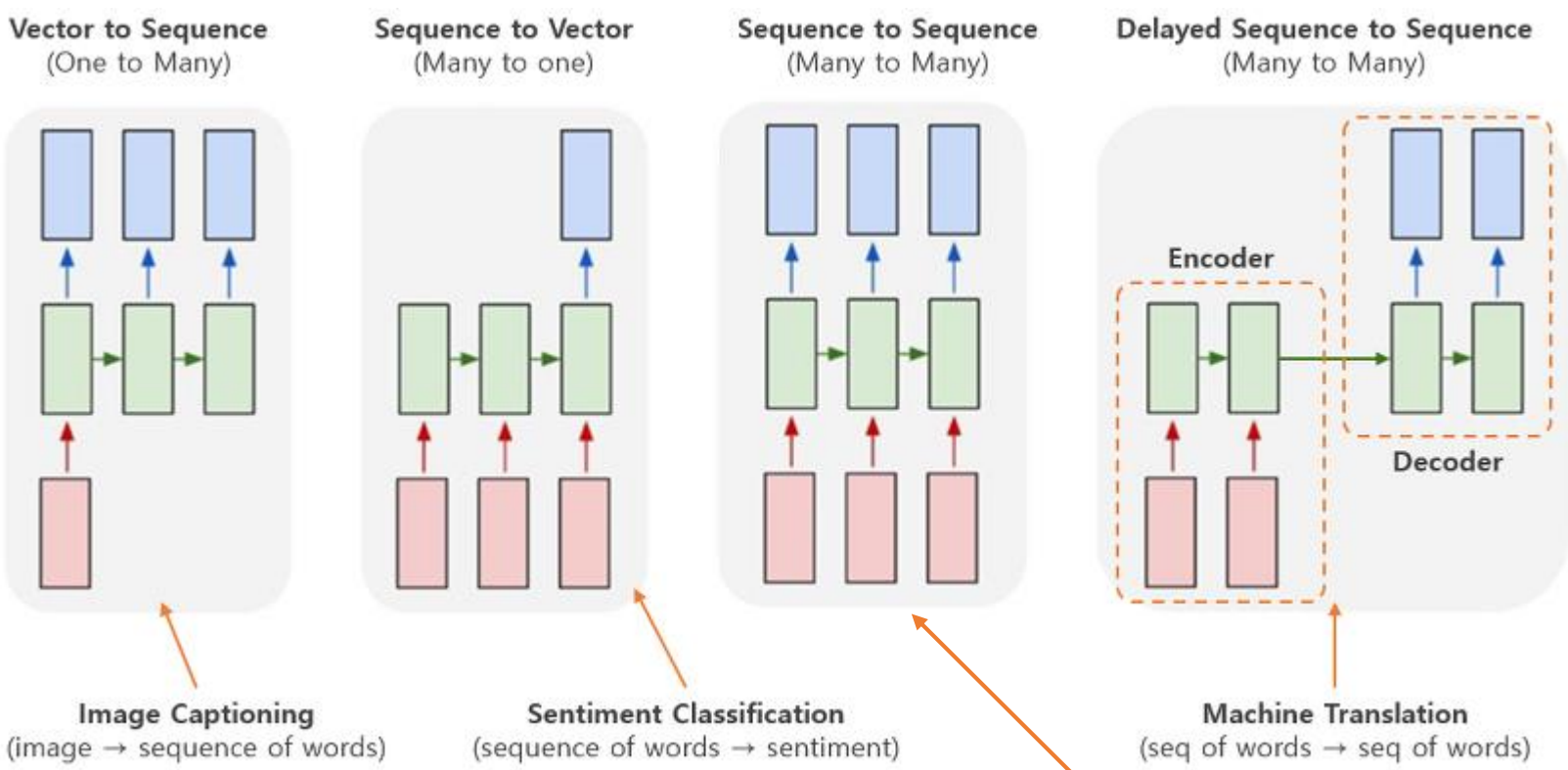
- E3에서 시작된 오류가 시간을 거슬러 가며 전파
- 또한 모든 시간에서의 웨이트는 공유가 되기 때문에 각 시간 별로의 오류를 합하면 최종적인 오류가 나옴

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y_3}} \frac{\partial \hat{y_3}}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$

RNN

RNN 종류 & 응용 기술

RNN의 종류 & 응용 기술



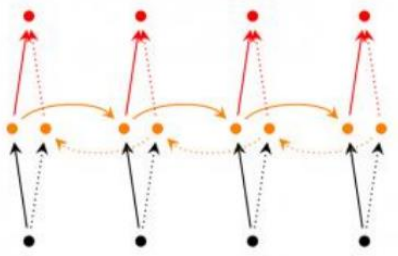
Many to Many : 문장에서 다음에 나올 단어를 예측하는 모델

RNN
RNN 종류 & 응용 기술

RNN의 종류 & 응용 기술

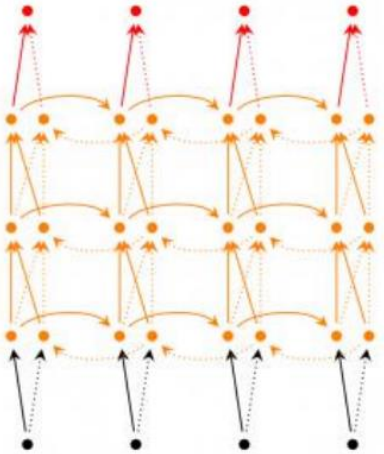
Bidirectional RNN

- 문장으로 치면 양쪽에서 오는 Context를 모두 알 수 있음
- 기본적으로 요즘 모델은 Bidirectional한 경우가 대다수



Deep (Bidirectional) RNN

- RNN을 여러 층으로 쌓음



LSTM

LSTM
LSTM 등장배경

RNN의 문제점?

LSTM
LSTM 등장배경

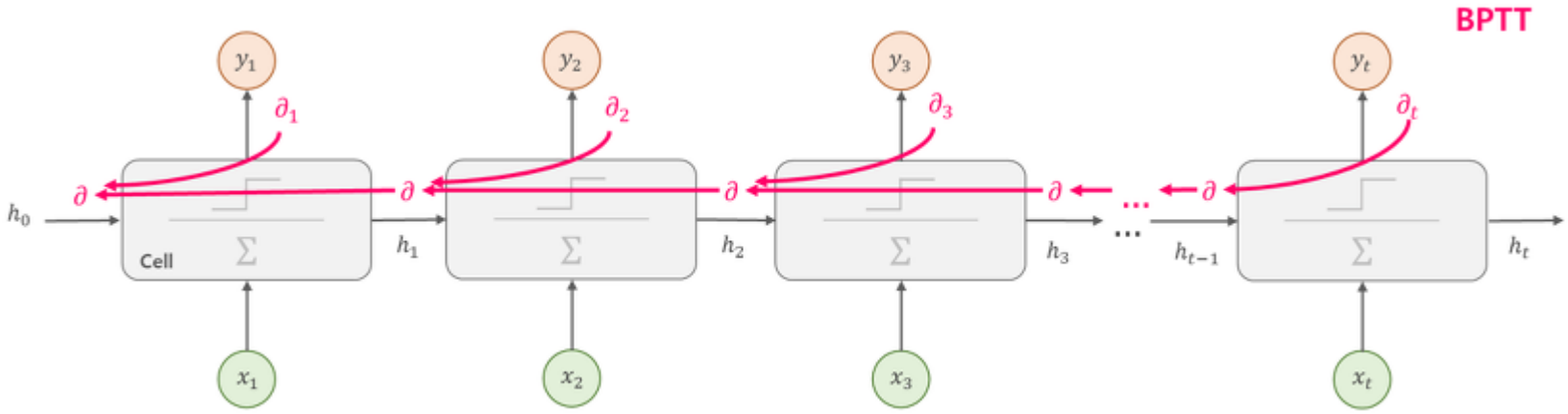
RNN의 문제점?

→ BPTT의 문제점!

LSTM
LSTM 등장배경

BPTT의 문제점!

→ BPTT는 아래의 그림 처럼 모든 타임 스텝 마다 처음 부터 끝까지 역전파



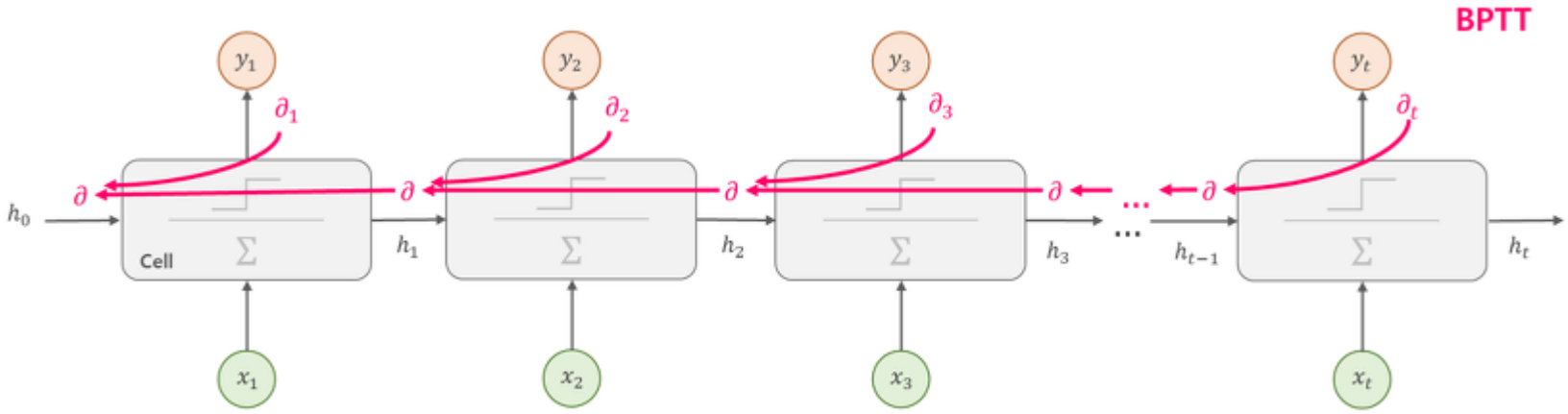
LSTM
LSTM 등장배경

BPTT의 문제점!

BPTT는 아래의 그림 처럼 모든 타임 스텝 마다 처음 부터 끝까지 역전파

→ Time step이 커지면 RNN을 펼쳤을 때 매우 깊어지며,

Gradient Vanishing & Exploding 문제 발생 & 학습 시간 매우 오래 걸리게 됨

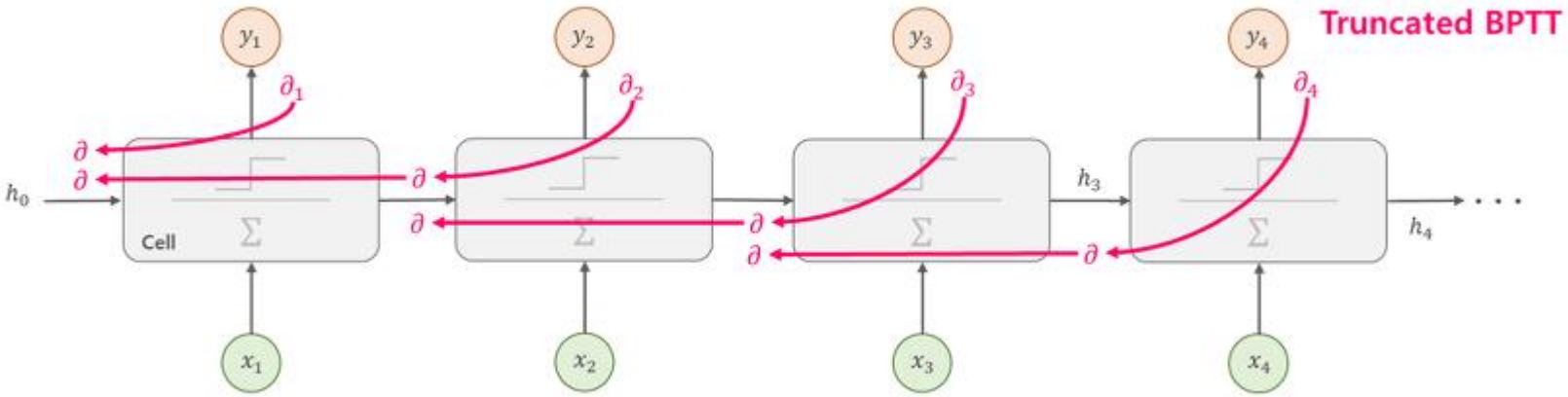


LSTM
LSTM 등장배경

BPTT? Truncated BPTT!

BPTT문제의 해결을 위해 타임 스텝을 일정 구간(보통 5 step)으로 나눠서 역전파 계산 (Truncated BPTT)

but, 학습 데이터가 장기간에 걸쳐 패턴 발생시 Long-Term에 대한 학습이 불가능

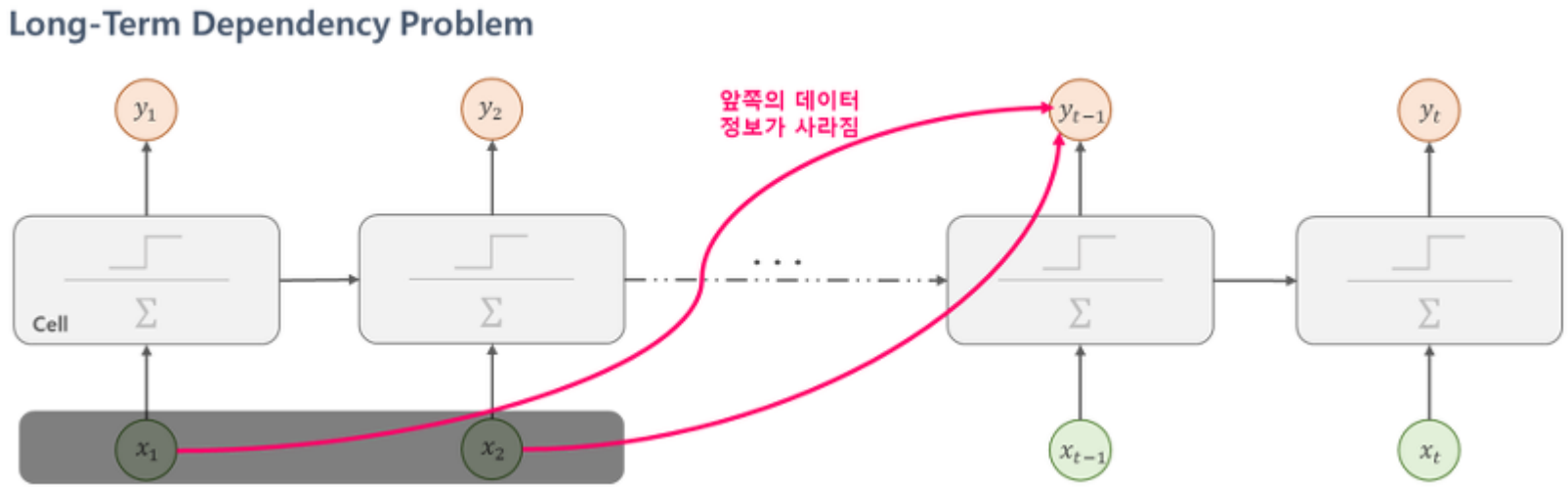


LSTM
LSTM 등장배경

Long-Term Dependency 문제

RNN은 이론적으로 모든 이전 타임 스텝이 영향을 주지만

앞쪽의 타임 스텝(예를 들어 $t = 0, t = 1$)은 타임 스텝이 길어질 수록 영향을 주지 못하는 문제가 발생

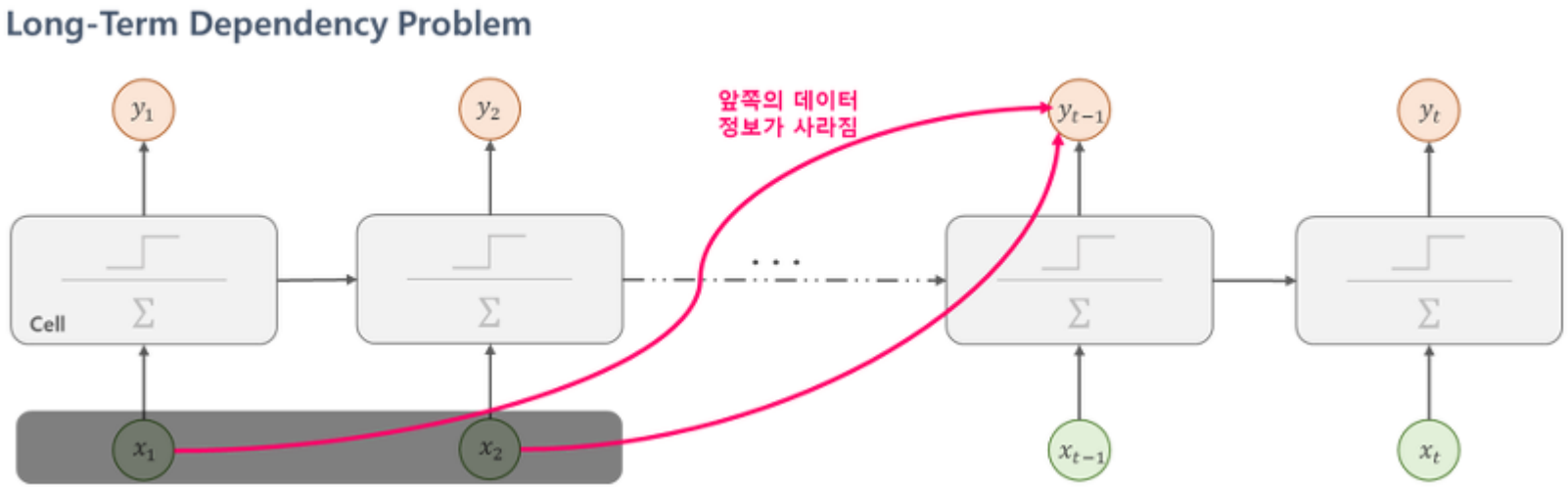


LSTM
LSTM 등장배경

Long-Term Dependency 문제

왜 발생해?

입력 데이터가 RNN Cell을 거치면서 특정 연산을 통해 데이터가 변환되어, 일부 정보는 타임 스텝 마다 사라지기 때문



LSTM
LSTM 등장배경

Long-Term Dependency 문제해결을 위해 다양한 셀 등장

→ 대표적으로 LSTM! (그리고 GRU)

LSTM
LSTM 알고리즘

LSTM?

→ **Long Short-Term Memory**

LSTM
LSTM 알고리즘

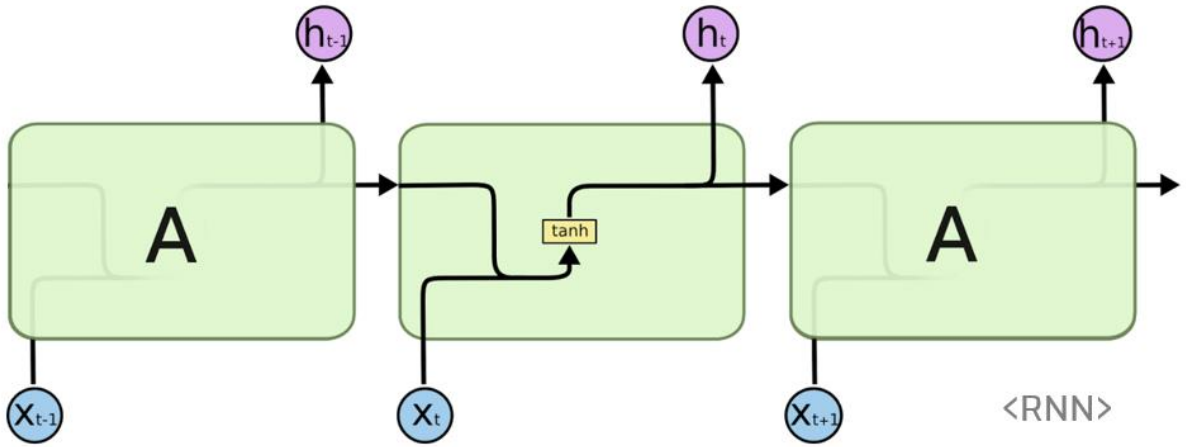
LSTM?

→ **Long Short-Term Memory**

: RNN에 Cell-state를 추가하자

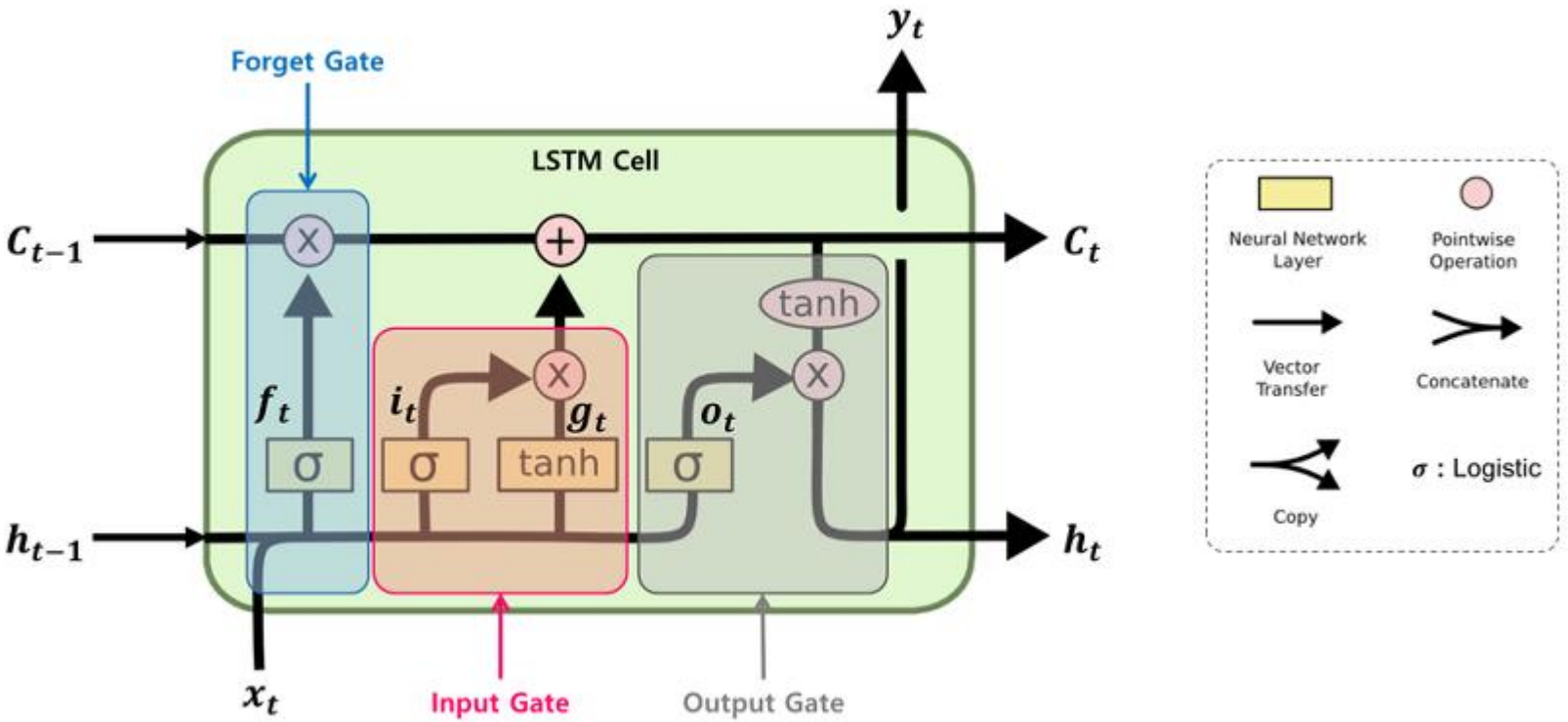
LSTM

LSTM 알고리즘



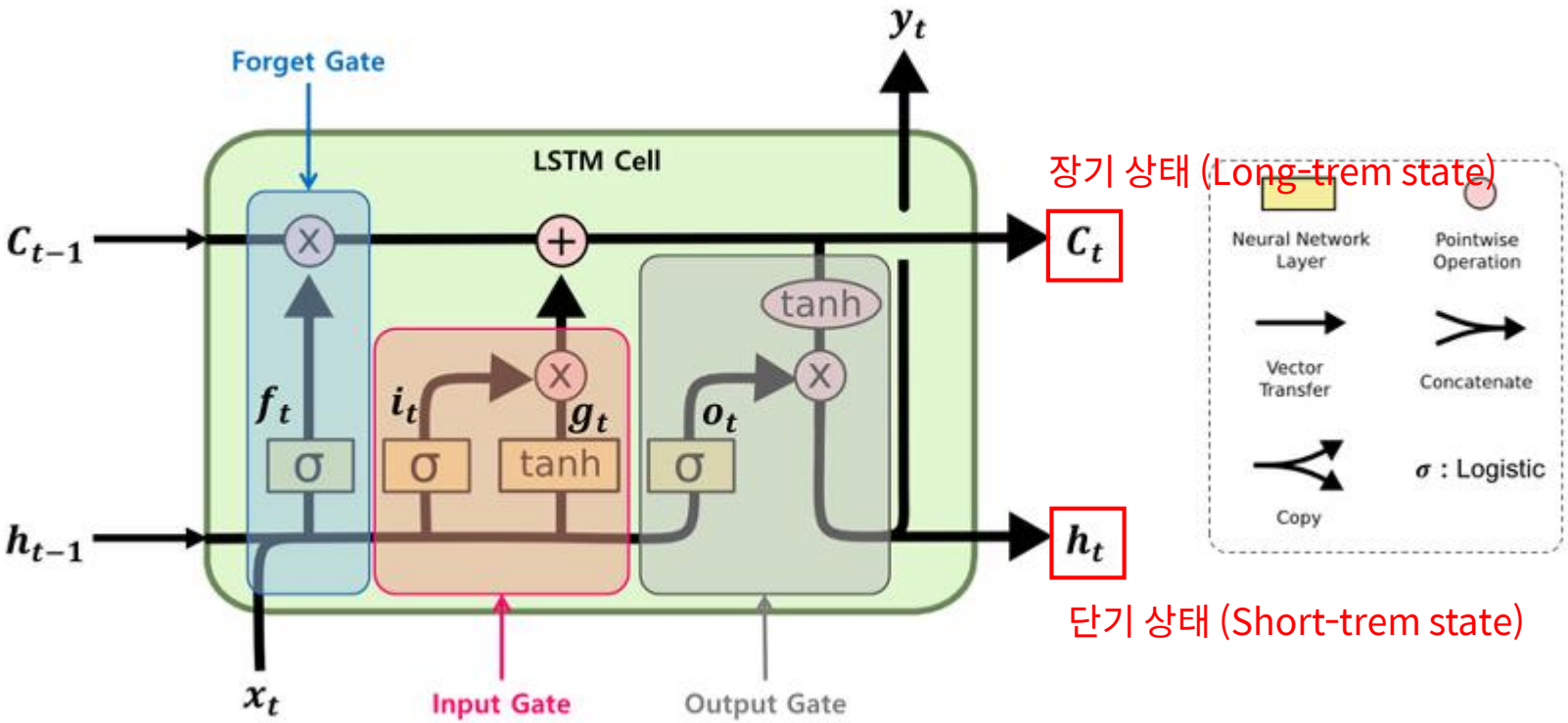
LSTM
LSTM 알고리즘

Long Short-Term Memory Cell



LSTM
LSTM 알고리즘

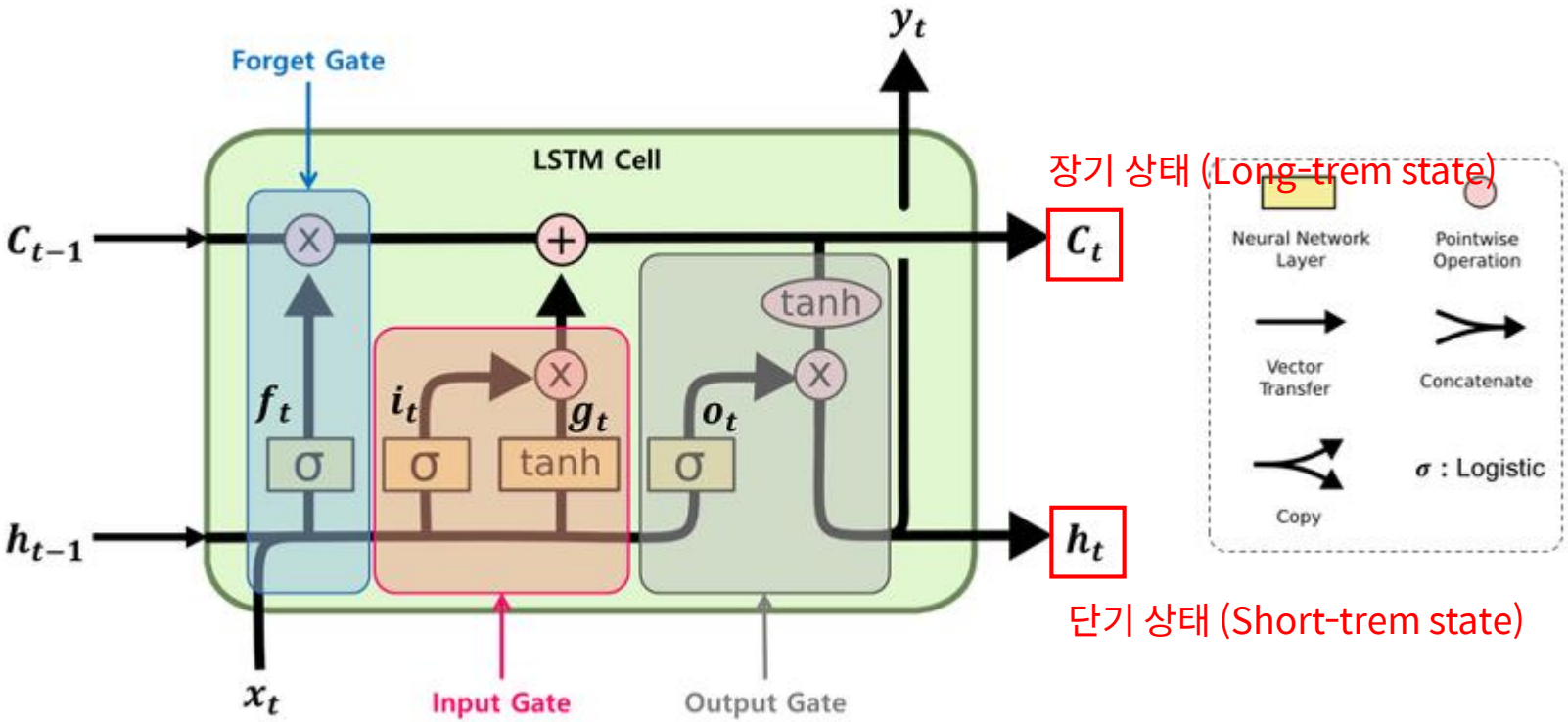
Long Short-Term Memory Cell



LSTM
LSTM 알고리즘

Long Short-Term Memory Cell

LSTM의 핵심 : 네트워크가 장기 상태(c_t)에서 기억할 부분, 삭제할 부분, 읽어 들일 부분을 학습하는 것



LSTM
LSTM 알고리즘

Long Short-Term Memory Cell

LSTM의 핵심 : 네트워크가 장기 상태(c_t)에서 기억할 부분, 삭제할 부분, 읽어 들일 부분을 학습하는 것

어떻게?

장기 기억c_{t-1}은 셀의 왼쪽에서 오른쪽으로 통과하게 되는데

1. forget gate를 지나면서 일부를 기억(정보)을 잃고
2. 그 다음 덧셈(+) 연산으로 input gate로 부터 새로운 기억 일부를 추가
3. 이렇게 만들어진 c_t는 별도의 추가 연산 없이 바로 출력됨
4. 이러한 장기 기억 c_t는 타임 스텝 마다 일부의 기억을 삭제하고 추가하는 과정을 거치게 됨
5. 그리고 덧셈 연산 후에 c_t는 복사되어 output gate의 함수로 전달되어 단기 상태h_t 와 셀의 출력인 y_t를 만듦

LSTM
LSTM 알고리즘

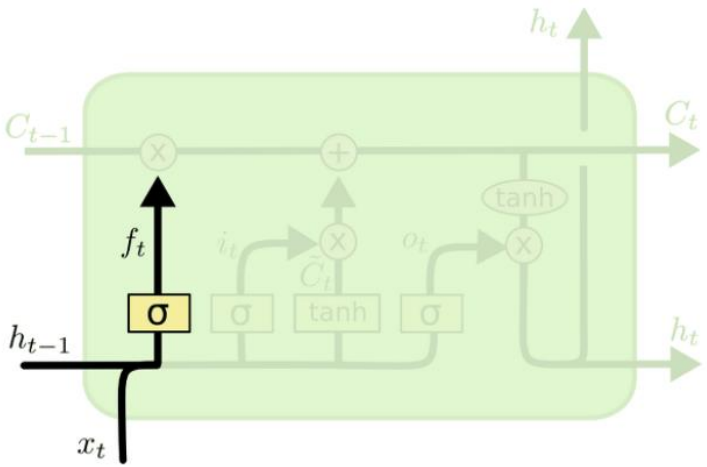
LSTM Forget Gate

Cell state에서 어떤 정보를 버릴지 선택하는 과정

Forget gate layer 라고 불리는 시그모이드 레이어로 만들어 짐

(0, 1사이의 출력 값을 가지는 h_{t-1}, x_t를 입력 값으로 받음)

0: 이 값을 유지해! / 1: 이 값을 버려!



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

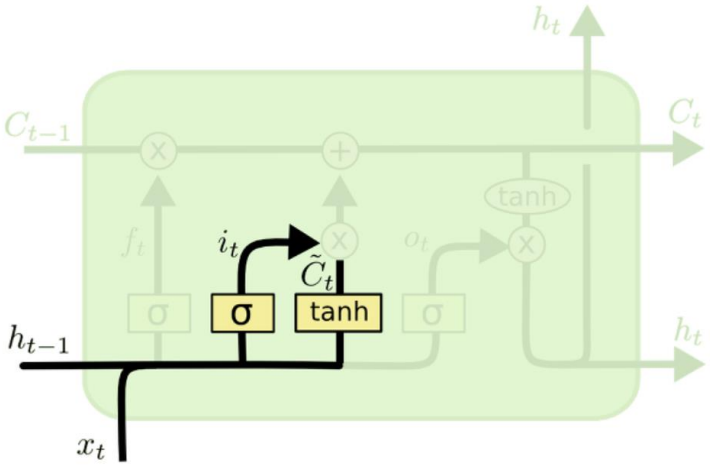
LSTM
LSTM 알고리즘

LSTM Input Gate

새로운 정보가 Cell State에 저장될 지 결정하는 과정

- 시그모이드 레이어 : 어떤 값을 우리가 업데이트 할지 결정
- tanh 레이어 : $\tilde{C}_t$ (Cell state에 더해질 수 있는 새로운 후보 값 생성)

다음으로 이 두 값을 합쳐서 다음 state에 영향을 줌



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

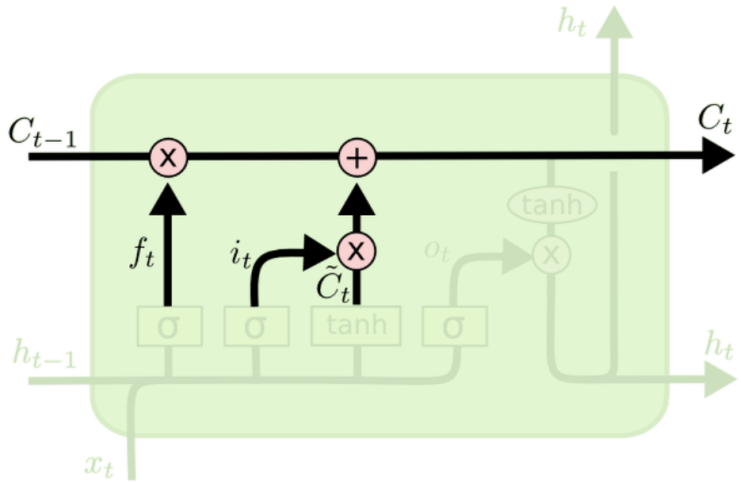
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

LSTM
LSTM 알고리즘

LSTM Cell State Update

오래된 Cell state(C_{t-1})를 새로운 state인 (C_t)로 업데이트

- C_{t-1}에 f_t로 곱함 (아까 잊기로 한 데이터 잊어버림)
- 새로운 후보 값이 기존 값에 영향 주도록 하기 위해 $i_t * \tilde{C}_t$를 더함



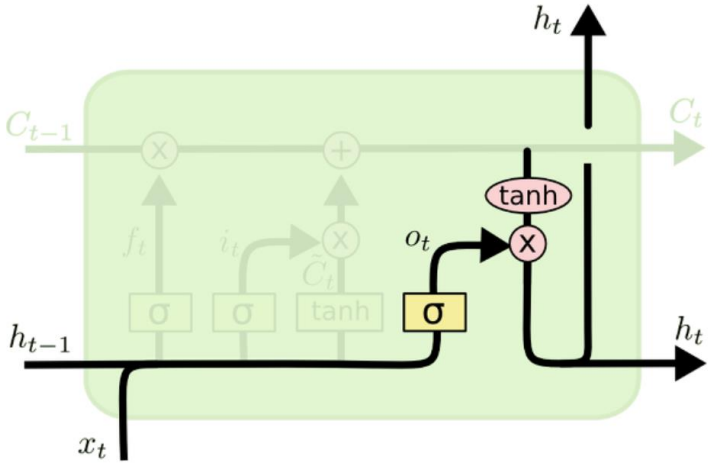
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

LSTM
LSTM 알고리즘

LSTM Output Gate

어떤 출력 값을 출력할지 결정

- 시그모이드 레이어 : 어떤 값을 출력할 지 결정
- cell state C_t를 tanh 함수를 거쳐 -1과 1사이 값을 뽑아 냄
- 위 결과를 sigmoid gate 출력 값과 곱



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM
LSTM 알고리즘

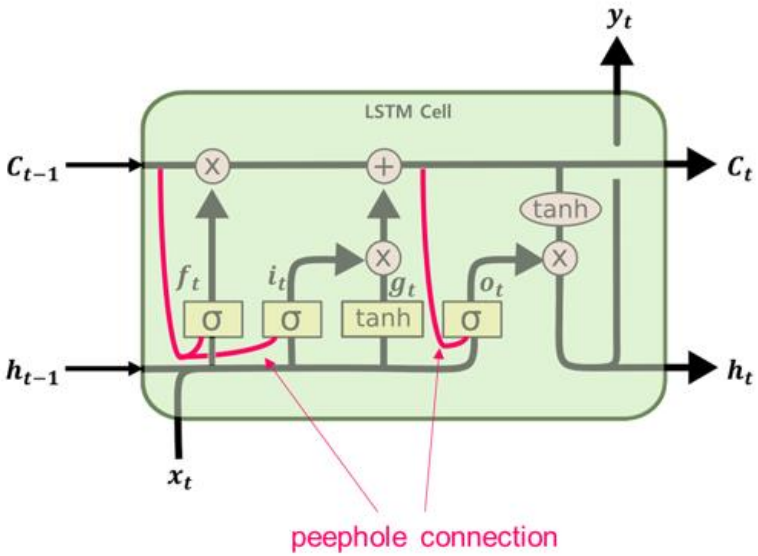
LSTM은 어떻게 Long-term dependency를 해결하는가?

- Cell State의 유무!
- 직접적으로 이전 셀과 현재 셀의 값을 더하기 때문에 이전의 셀에 대한 기억을 더 잘 할 수 있다!
- Vanishing Gradient 또한 해결 가능

LSTM
LSTM 변형

Peephole Connection

- 기존의 LSTM에서 gate controller(f_t, i_t, o_t)는 입력 x_t와 이전 타임 스텝의 단기 상태 h_{t-1}만 입력으로 받음
- 핍홀 연결을 아래의 그림과 같이 연결 해주면서 gate controller에 이전 타임 스텝의 장기 상태c_{t-1} 가 입력으로 추가되며, 좀 더 많은 맥락(context)를 인식할 수 있다



$$f_t = \sigma\left(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [C_t, h_{t-1}, x_t] + b_o\right)$$

GRU

GRU
GRU

GRU?

→ Gated Recurrent Unit

GRU
GRU

GRU?

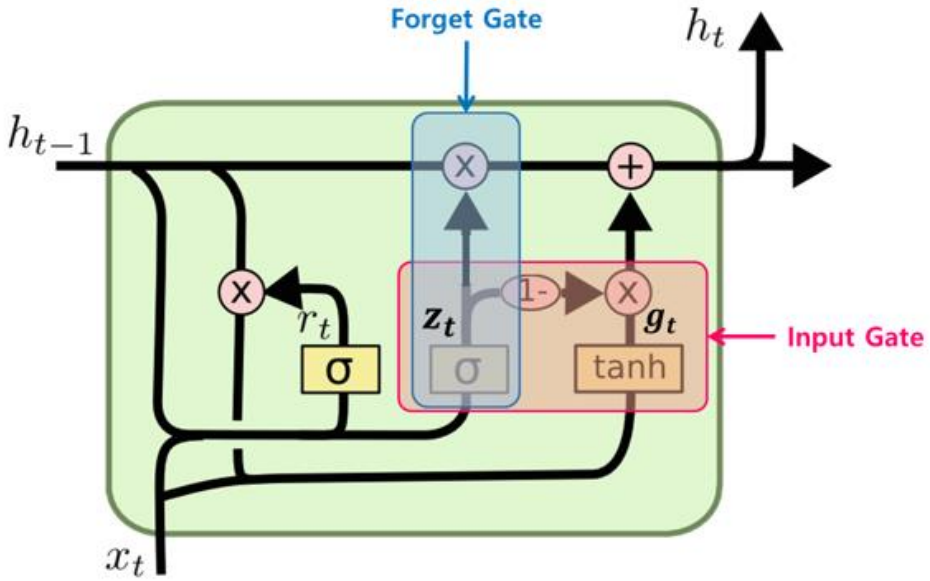
→ Gated Recurrent Unit

: LSTM cell의 간소화 버전!

GRU
GRU

Gated Recurrent Unit

- LSTM Cell에서 두 상태 벡터 c_t, h_t가 하나의 벡터 h_t로 합쳐 짐
- 하나의 gate controller z_t가 forget, input gate를 모두 제어
  - $z_t = 1$: forget gate open, input gate close / $z_t = 0$: 반대
  - → 즉, 이전(t-1)의 기억이 저장될 때 마다 타임 스텝 t의 입력은 삭제
- GRU Cell은 output gate가 없기 때문에 전체 상태 벡터 h_t가 타임 스텝 마다 출력,
  이전 상태 h_{t-1}의 어느 부분이 출력될지 제어하는 새로운 gate controller r_t 존재



$$\mathbf{r}_t = \sigma\left(\mathbf{W}_{xr}^T \cdot \mathbf{x}_t + \mathbf{W}_{hr}^T \cdot \mathbf{h}_{t-1} + \mathbf{b}_r\right)$$
$$\mathbf{z}_t = \sigma\left(\mathbf{W}_{xz}^T \cdot \mathbf{x}_t + \mathbf{W}_{hz}^T \cdot \mathbf{h}_{t-1} + \mathbf{b}_z\right)$$
$$\mathbf{g}_t = \tanh\left(\mathbf{W}_{xg}^T \cdot \mathbf{x}_t + \mathbf{W}_{hg}^T \cdot (\mathbf{r}_t \otimes \mathbf{h}_{t-1}) + \mathbf{b}_g\right)$$
$$\mathbf{h}_t = \mathbf{z}_t \otimes \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \otimes \mathbf{g}_t$$

Thank you.