LSTM and GRU

(Long Short Term Memory and Gated Recurrent Unit)

Winter Vacation Capstone Study

TEAM Kai.Lib

발표자 : 김수환

2020.01.06 (MON)

LSTM의 등장배경

▪ 게이트가 추가된 RNN

RNN은 순환 경로를 포함하여 과거의 정보를 기억할 수 있었다. 구조가 단순하여 구현도 쉽게 할 수 있었지만 안타깝게도 성능이 좋지 못하다. 그 원인은 많은 경우 시계열 데이터에서 시간적으로 많이 떨어진 장기long term 의존 관계를 잘 학습할 수 없다는 데 있다. 해서 요즘에는 단순한 RNN 대신 LSTM이나 GRU라는 계층이 주로 쓰인다.

LSTM이나 GRU에는 게이트gate라는 구조가 더해져 있는데, 이 게이트 덕분에 시계열 데이터의 장기 의존 관계를 학습할 수 있다. 이번 스터디에서는 LSTM과 GRU 내부적으로 어떠한 구조로 되어있으며, 어떻게 이 구조가 '장기 기억'을 가능하게 하는지를 이해한다

Tom was watching TV in his room. Mary came into the room. Mary said hi to ?

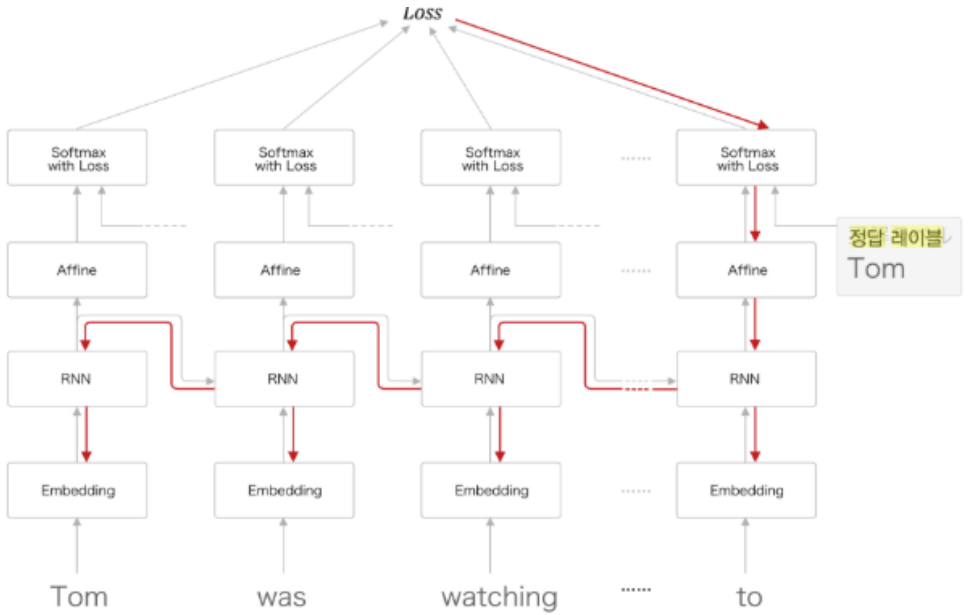
"?"에 들어가는 단어는 "Tom"이다. RNN에서 이 문제에 올바르게 답하려면, 현재 맥락에서 "Tom이 방에서 TV를 보고 있음"과 "그 방에 Mary가 들어옴"이란 정보를 기억해둬야 한다. 다시 말해 이런 정보를 RNN 계층의 은닉 상태에 인코딩해 보관해둬야한다.

LSTM의 등장배경

- **RNN의 문제점**

Tom was watching TV in his room. Mary came into the room. Mary said hi to ?

그럼 이 예를 RNNLM 학습의 관점에서 생각해보면 여기서 정답 레이블로 "Tom"이라는 단어가 주어졌을 때, RNNLM에서 기울기가 어떻게 전파되는지를 살펴보자. 물론 학습은 BPTT로 수행한다. 따라서 정답 레이블이 "Tom"이라고 주어진 시점으로부터 과거 방향으로 기울기를 전달하게 된다. 그림으로는 다음처럼 된다.

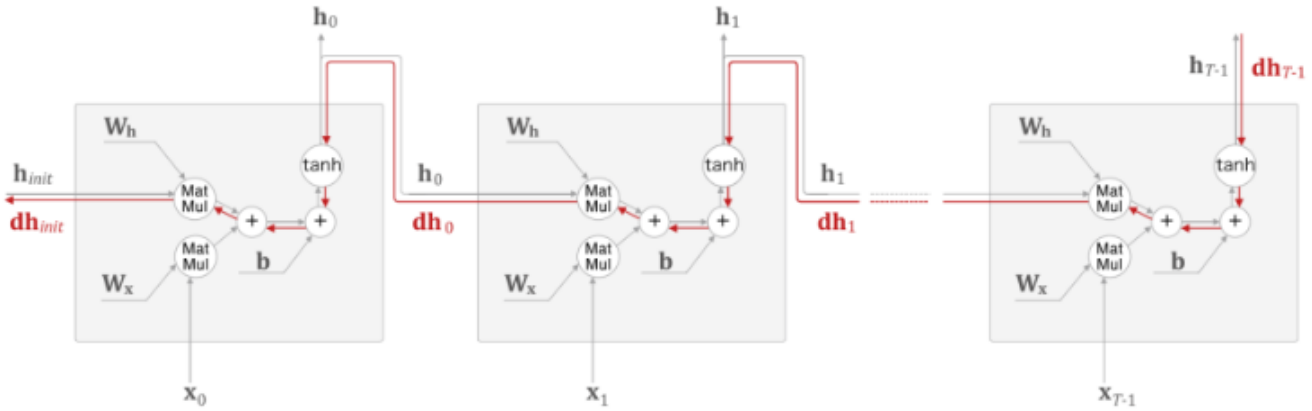


다음 그림처럼 RNN 계층이 과거 방향으로 의미 있는 기울기를 전달함으로써 시간 방향의 의존 관계를 학습할 수 있는 것이다. 하지만 현재의 단순한 RNN 계층에서는 시간을 거슬러 올라갈수록 기울기가 작아지거나 (**기울기 소실**) 혹은 커질 수 있으며 (**기울기 폭발**), 대부분 둘 중 하나의 운명을 걷게 된다.

RNN의 문제점

▪ 기울기 소실과 기울기 폭발의 원인

그럼 RNN 계층에서 기울기 소실(혹은 기울기 폭발)이 일어나는 원인을 살펴보자. 생각을 간단하게 하기 위해 다음 그림과 같이 RNN 계층에서의 시간 방향 기울기 전파에만 주목해보자.

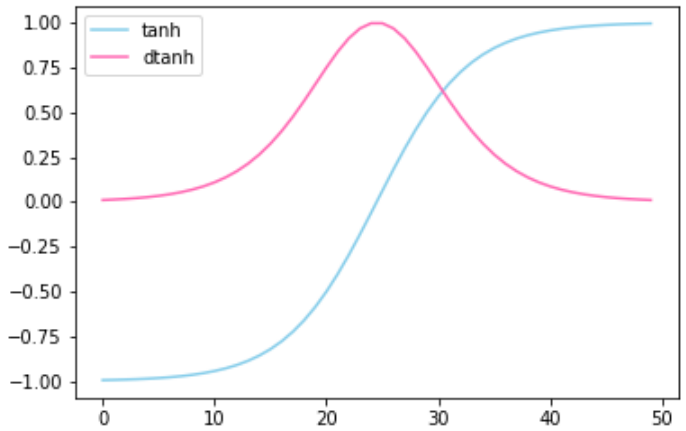


길이가 T인 시계열 데이터를 가정하여 T번째 정답 레이블로부터 전해지는 기울기가 어떻게 변하는지 살펴보자. 시간 방향 기울기에 주목하면 역전파로 전해지는 기울기는 차례로 'tanh', '+', 'Matmul(행렬곱)' 연산을 통과한다는 것을 알 수 있다.
'+'의 역전파는 상류에서 전해지는 기울기를 그대로 하류로 흘려보낼 뿐이니, 'tanh'와 'Matmul'에만 주목해보자.
우선 'tanh'부터 보자.

RNN의 문제점

- **기울기 소실과 기울기 폭발의 원인 - tanh**

앞의 스터디에서 tanh 함수의 미분에서 봤듯이, y = tanh(x)일 때의 미분은 $\frac{\partial y}{\partial x}$ = 1-y^2 이다. 이때 y = tanh(x)의 값과 그 미분 값을 각각 그래프로 그리면 다음 그림처럼 된다.



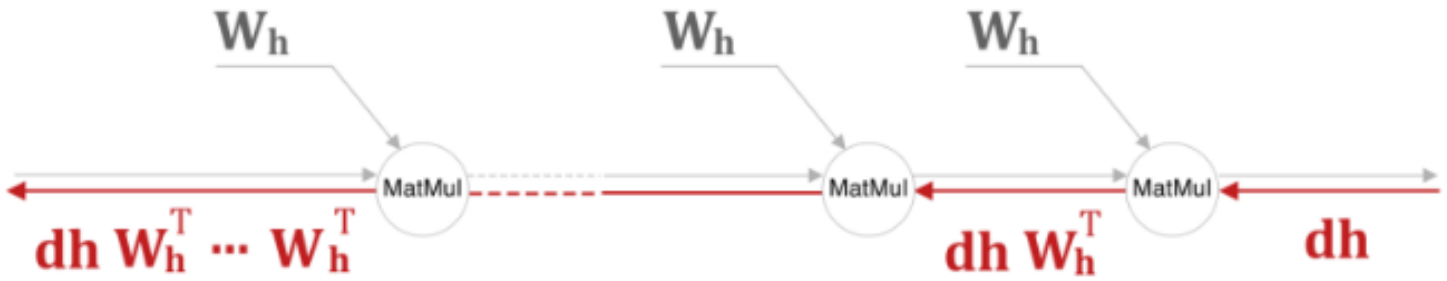
dtanh에 주목하면 그 값은 1.0이하이고, x가 0으로부터 멀어질수록 작아진다. 달리 말하면 이는 역전파에서는 기울기가 tanh 노드를 지날 때마다 값은 계속 작아진다는 뜻이다. 그래서 tanh 함수를 T번 통과하면 기울기도 T번 반복해서 작아지게 된다.

NOTE_ RNN 계층의 활성화 함수로는 주로 tanh 함수를 사용하는데, 이를 ReLU로 바꾸면 기울기 소실을 줄일 수 있다.
「Improving performance of recurrent neural network with rely nonlinearity」 논문에서는 ReLU를 사용해 성능을 개선했다.

RNN의 문제점

- **기울기 소실과 기울기 폭발의 원인 - Matmul**

다음으로 Matmul 노드에 주목해보자. 여기서는 이야기를 단순하게 하기 위해 tanh 노드를 무시하기로 한다.

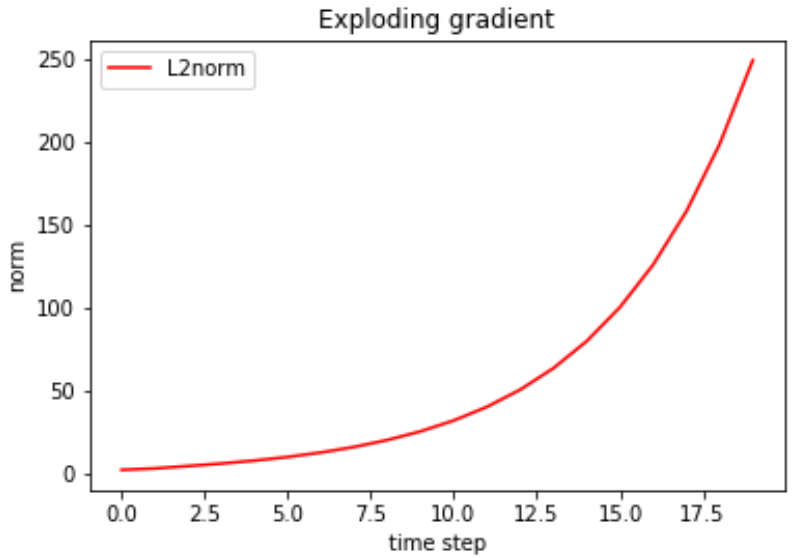


다음처럼 상류로부터 **dh**라는 기울기가 흘러들어왔을 때 이때 Matmul 노드에서의 역전파는 $\mathbf{dhW_h^T}$라는 행렬 곱으로 기울기를 계산한다. 그리고 같은 계산을 시계열 데이터의 시간 크기만큼 반복한다. 여기에서 주목할 점은 이 행렬 곱셈에서는 매번 똑같은 가중치인 $\mathbf{W_h}$가 사용된다는 점이다. 간단한 코드로 한 번 실험해보자.

RNN의 문제점

- **Exploding gradient**

```python
import numpy as np
import matplotlib.pyplot as plt

N = 2
H = 3
T = 20

dh = np.ones((N, H))
np.random.seed(3)
Wh = np.random.randn(H, H)

norm_list = list()
for t in range(T):
    dh = np.matmul(dh, Wh.T)
    norm = np.sqrt(np.sum(dh**2)) / N
    norm_list.append(norm)

plt.figure()
plt.plot(norm_list, label='L2norm', color = 'red')
plt.legend()
plt.title('Exploding gradient')
plt.xlabel('time step')
plt.ylabel('norm')
plt.savefig('gradient_explosion.png')
```

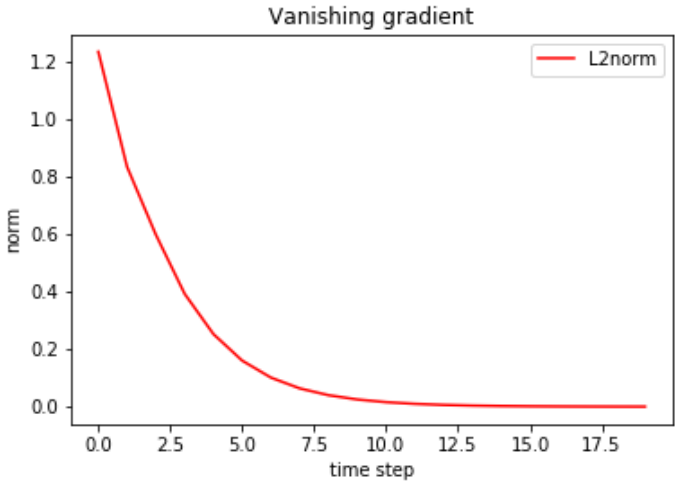


역전파의 Matmul 노드 수 (T)만큼 dh를 갱신했을 때, 기울기의 크기는 시간에 비례해 지수적으로 증가함을 알 수 있다.
이것이 바로 기울기 폭발exploding gradient이다. 이러한 기울기 폭발이 일어나면 결국 Overflow를 일으켜 NaNNot a Number 같은 값을 발생시킨다.

RNN의 문제점

- **Vanishing gradient**

```python
import numpy as np
import matplotlib.pyplot as plt

N = 2
H = 3
T = 20

dh = np.ones((N, H))
np.random.seed(3)
Wh = np.random.randn(H, H)

norm_list = list()
for t in range(T):
    dh = np.matmul(dh, Wh.T) / 2
    norm = np.sqrt(np.sum(dh**2)) / N
    norm_list.append(norm)

plt.figure()
plt.plot(norm_list, label='L2norm', color = 'red')
plt.legend()
plt.title('Vanishing gradient')
plt.xlabel('time step')
plt.ylabel('norm')
plt.show()
```

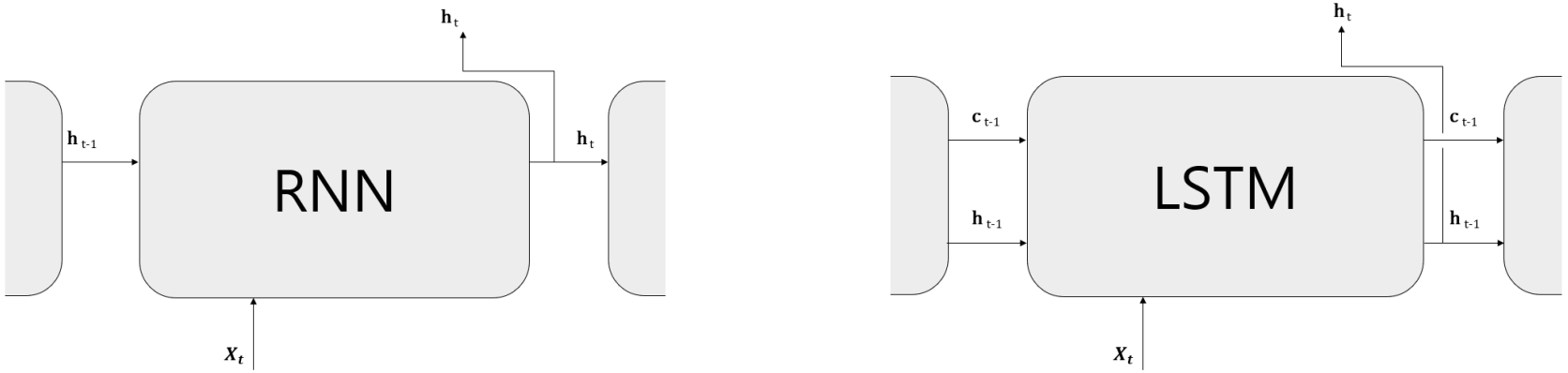


이번에는 초기값을 절반으로 줄이고 똑같이 돌려봤을 때,
기울기가 지수적으로 감소하는 것을 확인할 수 있다.
이것이 기울기 소실**Vanishing gradient**이다. 이처럼 기울기 소실이 일어나서 기울기가 일정 수준 이하로 작아지면 가중치 매개변수가 더 이상 갱신되지 않는다.

LSTM

▪ 기울기 소실과 LSTM

이제 이러한 기울기 소실을 일으키지 않는다는 (혹은 일으키기 어렵게 한다는) LSTM 구조에 대해 살펴보고 이 구조를 개량한 GRU의 구조까지 살펴보자.



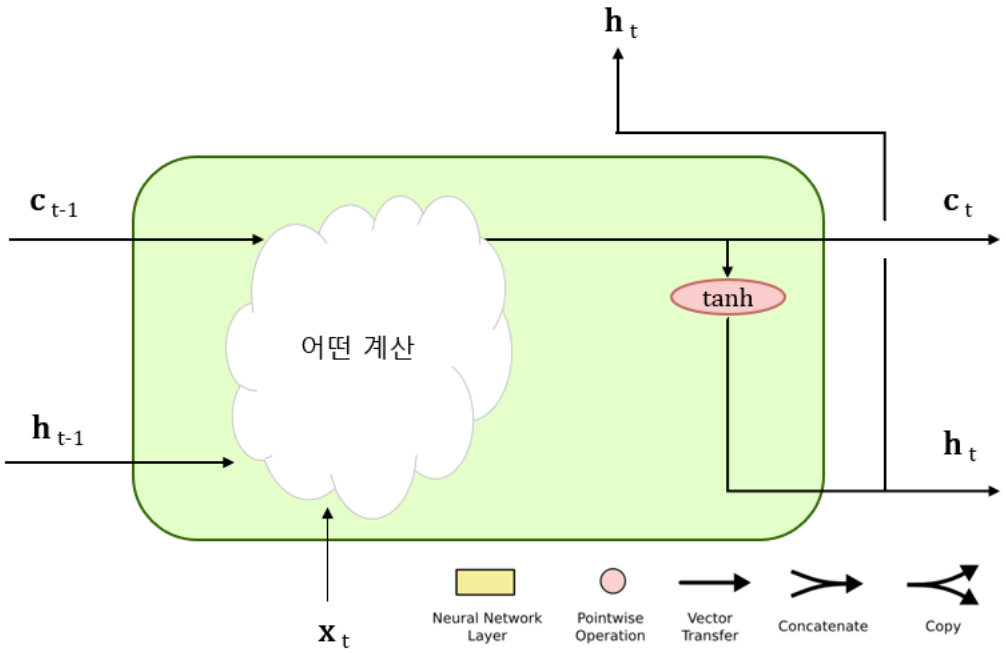
RNN 계층과 LSTM 계층 비교

다음은 RNN과 LSTM의 인터페이스를 비교한 그림이다. 그림에서 보듯 LSTM 계층의 인터페이스에는 **c**라는 경로가 있다는 차이가 있다. C를 기억 셀memory cell(혹은 단순히 '셀')이라 하며, LSTM 전용의 **기억 매커니즘**이다.
기억 셀의 특징은 데이터를 자기 자신으로만 (LSTM 계층 내에서만) 주고받는다는 것이다. 즉, LSTM 계층 내에서만 완결되고, 다른 계층으로는 출력하지 않습니다. 반면, LSTM의 은닉 상태 **h**는 RNN 계층과 마찬가지로 다른 계층으로 출력된다.

NOTE_ LSTM의 출력을 받는 쪽에서 보면 LSTM의 출력은 은닉 상태 벡터 h뿐이다. 그러므로 c의 존재 자체를 생각할 필요가 없다.

LSTM

- **LSTM 계층 조립하기**



기억 셀 c_t를 바탕으로 은닉 상태 h_t를 계산하는 LSTM 계층

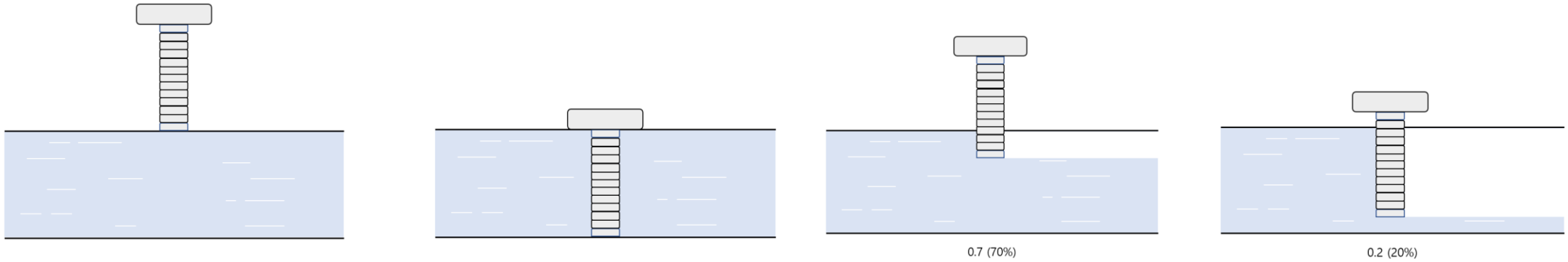
이제 LSTM의 구조를 차분히 알아보자. 앞서 이야기한 것처럼, LSTM에는 기억 셀 **c_t**가 있다. 이 **c_t**에는 시각 t에서의 LSTM의 기억이 저장되어 있는데, 과거로부터 시각 t까지에 필요한 모든 정보가 저장돼 있다고 가정하자. 그리고 필요한 정보를 모두 간직한 이 기억을 바탕으로 외부 계층에 은닉 상태 **h_t**를 출력한다. 이때 출력하는 **h_t**는 다음 그림과 같이 기억 셀의 값을 **tanh** 함수로 변환한 값이다. 그림처럼 현재의 기억 셀 **c_t**는 3개의 입력 **(c_{t-1}, h_{t-1}, x_t)**으로부터 '어떤 계산'을 수행하여 구할 수 있다. 여기서 핵심은 갱신된 **c_t**를 사용해 은닉상태 **h_t**를 계산한다는 것이다. 또한 이 계산은 **h_t = tanh(c_t)**인데,이는 c_t의 각 요소에 tanh 함수를 적용한다는 뜻이다.
LSTM 구조에서의 핵심은 **h_t**는 단기상태(short term state), **c_t**는 장기상태(long term state)라고 볼 수 있다.

LSTM

▪ LSTM의 게이트gate

진도를 더 나가기 전에, 이쯤에서 '게이트'라는 기능에 대해 얘기해보자. 게이트는 데이터의 흐름을 제어한다. 마치 다음 그림처럼 물의 흐름을 멈추거나 배출하는 것이 게이트의 역할이다.

그림1 비유하자면 게이트는 물의 흐름을 제어한다.



LSTM에서 사용하는 게이트는 '열기/닫기 ' 뿐 아니라 어느 정도 열지를 조절할 수 있다. 다음 그림처럼 게이트의 열림 상태는 0.0~1.0 사이의 실수로 나타난다. 그리고 그 값이 흐르는 물의 양을 결정한다. 여기서 중요한 것은 '게이트를 얼마나 열까'라는 것도 데이터로부터 자동으로 학습한다는 점이다.

NOTE_ 게이트는 게이트의 열림 상태를 제어하기 위해서 전용 가중치 매개변수를 이용하며, 이 가중치 매개변수는 학습 데이터로부터 갱신된다. 참고로 게이트의 열림 상태를 구할 때는 시그모이드 함수를 사용하는데, 그 이유는 시그모이드 함수의 출력이 0.0~1.0 사이의 실수이기 때문이다.

LSTM

▪ output 게이트 - (1)

다시 LSTM 이야기로 돌아와보자. 바로 앞에서 은닉상태 $\mathbf{h}_t$는 기억 셀 $\mathbf{c}_t$에 단순히 tanh 함수를 적용했을 뿐이라고 설명했다.
이번에는 tanh($\mathbf{c}_t$)에 게이트를 적용하는 걸 생각해보자.
즉, tanh($\mathbf{c}_t$)의 각 원소에 대해 '**그것이 다음 시각의 은닉 상태에 얼마나 중요한가**'를 조정한다.

output 게이트의 열림 상태 (다음 몇 %만 흘려보낼까)는 입력 $\mathbf{x}_t$와 이전 상태 $\mathbf{h}_{t-1}$로부터 구한다. 이때의 계산은 다음과 같다.

Sigmoid

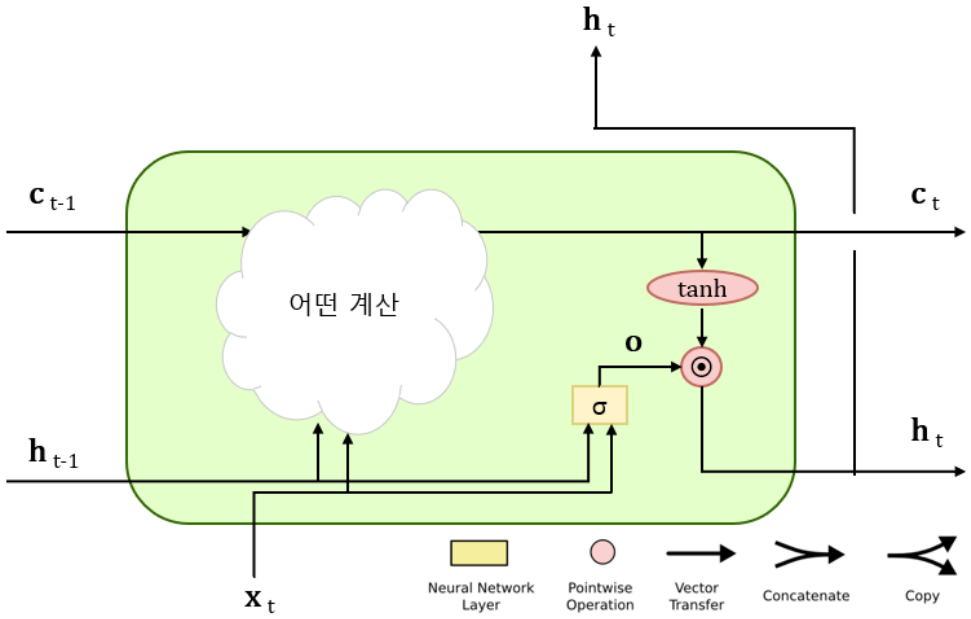
$$\mathbf{o} = \sigma(\mathbf{x}_t \mathbf{W}_x^{(o)} + \mathbf{h}_{t-1} \mathbf{W}_h^{(o)} + \mathbf{b}^{(o)})$$

Output Gate

위의 식을 보게되면 RNN 계층의 계산에서 tanh가 아닌 Sigmoid를 사용했다는 점만이 다르다는 것을 확인할 수 있다.

LSTM

- **output 게이트 - (2)**



Output 게이트가 추가된 LSTM 계산 그래프

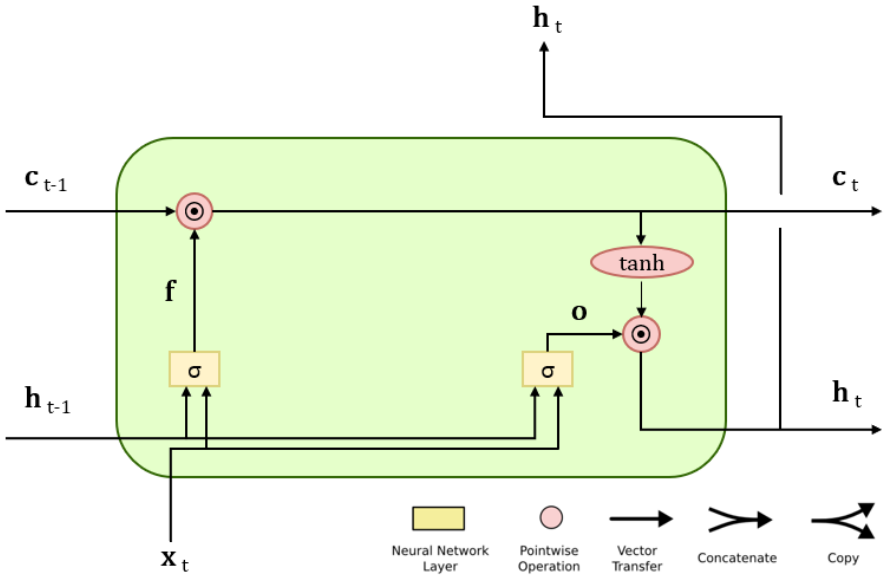
output 게이트에서 수행하는 식을 을 σ로 표기했다. 그리고 σ의 출력을 $\mathbf{o}$라고 하면 $\mathbf{h_t}$는 $\mathbf{o}$와 tanh($\mathbf{c_t}$)의 곱으로 계산된다. 이때 말하는 '곱'이란 원소별element-wise이며, 이것을 아다마르 곱Hardamard product이라고도 한다. 아다마르 곱을 기호로는 ⊙로 나타내며 다음과 같은 계산을 수행한다.

$$\mathbf{h_t} = \mathbf{o} \odot \tanh(\mathbf{c_t})$$

NOTE_ tanh의 출력은 -1.0~1.0의 실수고, 이 -1.0~1.0의 수치를 인코딩된 '정보'의 강약(정도)을 표시한다고 해석할 수 있다. 한편 시그모이드 함수의 출력은 0.0~1.0의 실수이며, 데이터를 얼마만큼 통과시킬지를 정하는 비율이된다. 주로 게이트에서는 시그모이드 함수가, 실질적인 '정보'를 지니는 데이터에는 tanh 함수가 활성화 함수로 사용된다.

LSTM

- **forget 게이트**



forget 게이트까지 추가된 LSTM 계산 그래프

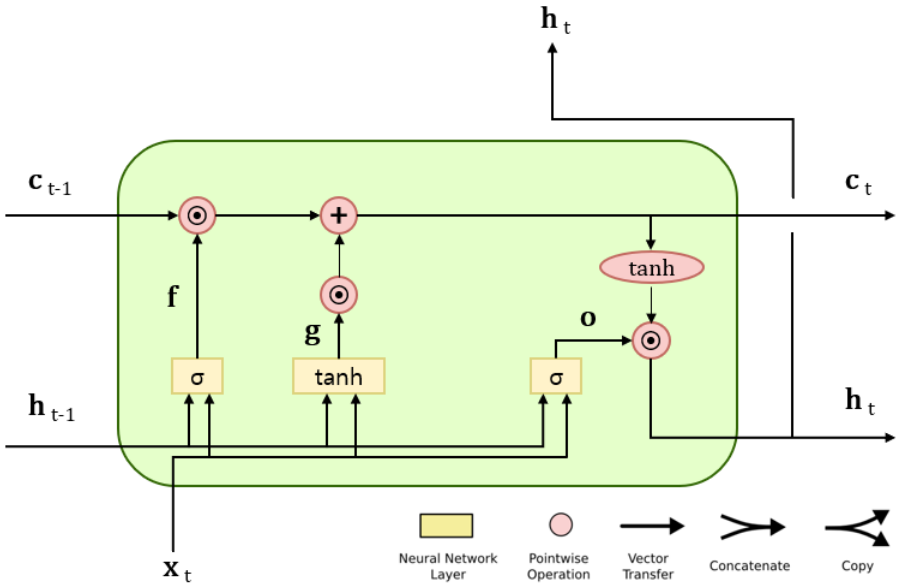
다음으로 할 일은 기억 셀에 '무엇을 잊을까'를 지시하는 것이다. 이것도 역시 게이트를 사용해 해결한다.
c_{t-1}의 기억 중에서 불필요한 기억을 잊게 해주는 게이트를 **forget 게이트**(망각 게이트)라고 한다. Forget 게이트가 수행하는 일련의 계산을 **σ** 노드로 표기했다. **σ** 안에서는 다음 식의 계산을 수행한다.

$$f = \sigma(x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)})$$

그리고 이 **f**와 이전 기억 셀인 c_{t-1}과의 아다마르 곱, 즉 $c_t = f \odot c_{t-1}$을 계산하여 c_t를 구한다.

LSTM

- **새로운 기억 셀**



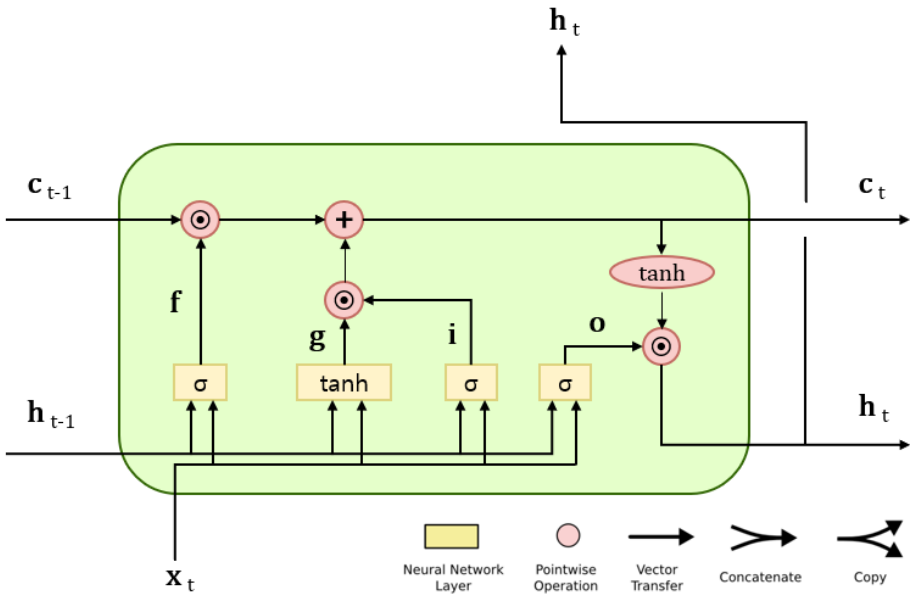
새로운 기억 셀에 필요한 정보가 추가된 LSTM 계산 그래프

forget 게이트를 거치면서 이전 시각의 기억 셀로부터 잊어야 할 기억이 삭제됐다. 이 상태로는 기억 셀이 잊는것 밖에 하지 못하므로 새로 기억해야 할 정보를 추가해야한다. 그러기 위해서 위의 그림과 같이 **tanh 노드를 추가**한다. ('정보'가 담겨있으므로)
위의 그림에서 보듯 tanh 노드가 계산한 결과가 이전 시각의 기억 셀 $\mathbf{c_{t-1}}$에 더해진다. 기억 셀이 새로운 '정보'가 추가된 것이다.
주의할 점은 이 tanh 노드는 '게이트'가 아니며, 새로운 '정보 ' 를 기억 셀에 추가하는 것이 목적이다. 따라서 활성화 함수로는 시그모이드 함수가 아닌 tanh 함수가 사용된다. 이제 잊는 것 뿐만이 아닌, 새로운 정보까지 추가가 되었다.

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

LSTM

- **input 게이트**



input 게이트까지 추가된 LSTM 계산 그래프

마지막으로 새로운 정보가 들어있는 **g**에 게이트를 하나 추가할 생각이다. 여기에서 새롭게 추가하는 게이트를 **input 게이트**라고 한다. Input 게이트를 **g**의 각 원소가 새로 추가되는 정보로써의 가치가 얼마나 큰지를 판단한다. 즉, 새로운 정보를 무비판적으로 수용하는 것이 아니라, 적절히 취사선택하는 것이 이 게이트의 역할이다.

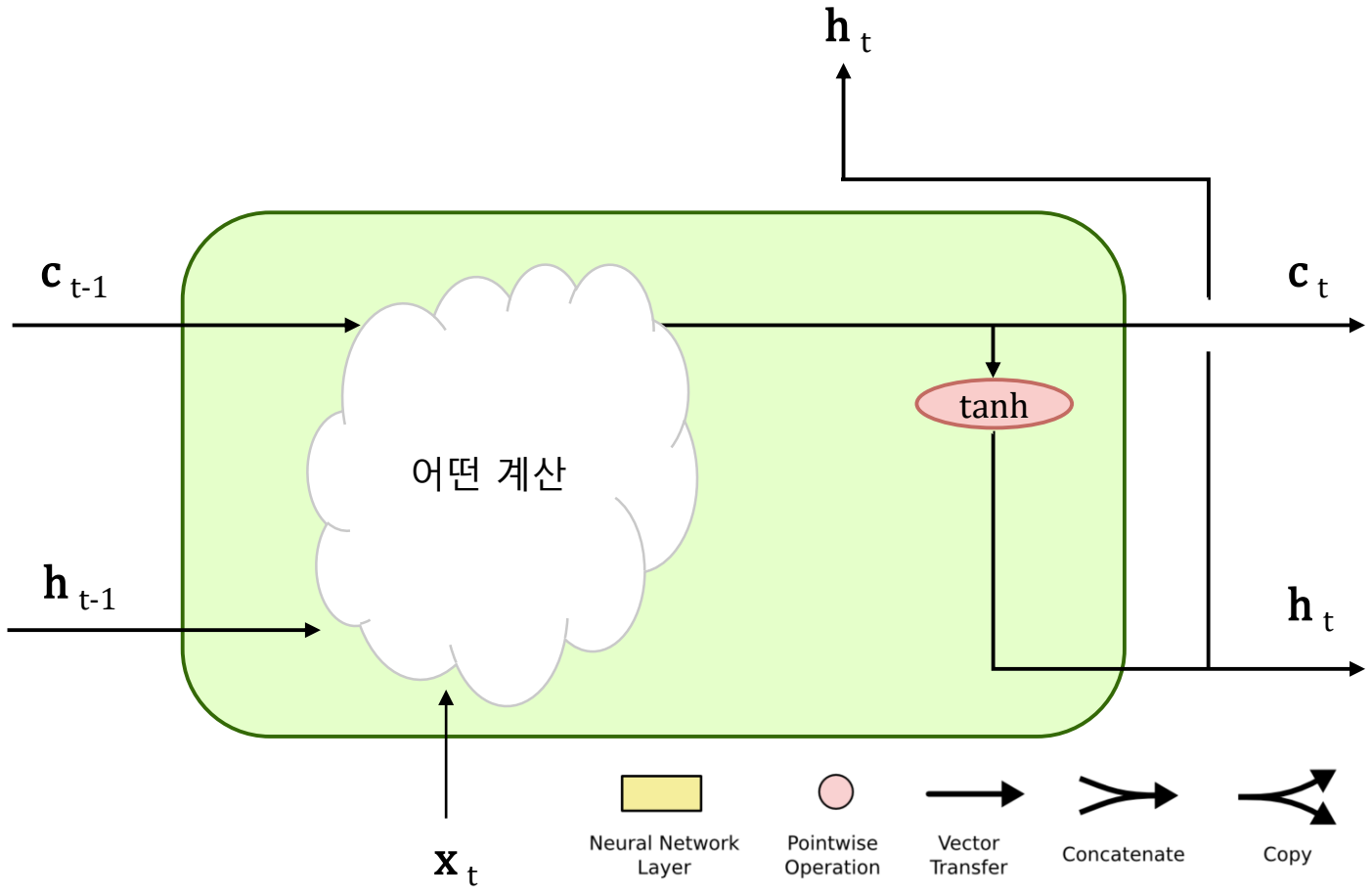
$$\mathbf{i} = \sigma(\mathbf{x}_t \mathbf{W}_x^{(i)} + \mathbf{h}_{t-1} \mathbf{W}_h^{(i)} + \mathbf{b}^{(i)})$$

그런 다음 i와 g의 아다마르 곱 결과를 기억 셀에 추가한다. 이상이 LSTM 안에서 이뤄지는 처리이다.

LSTM Review

Long Short Term Memory (LSTM)

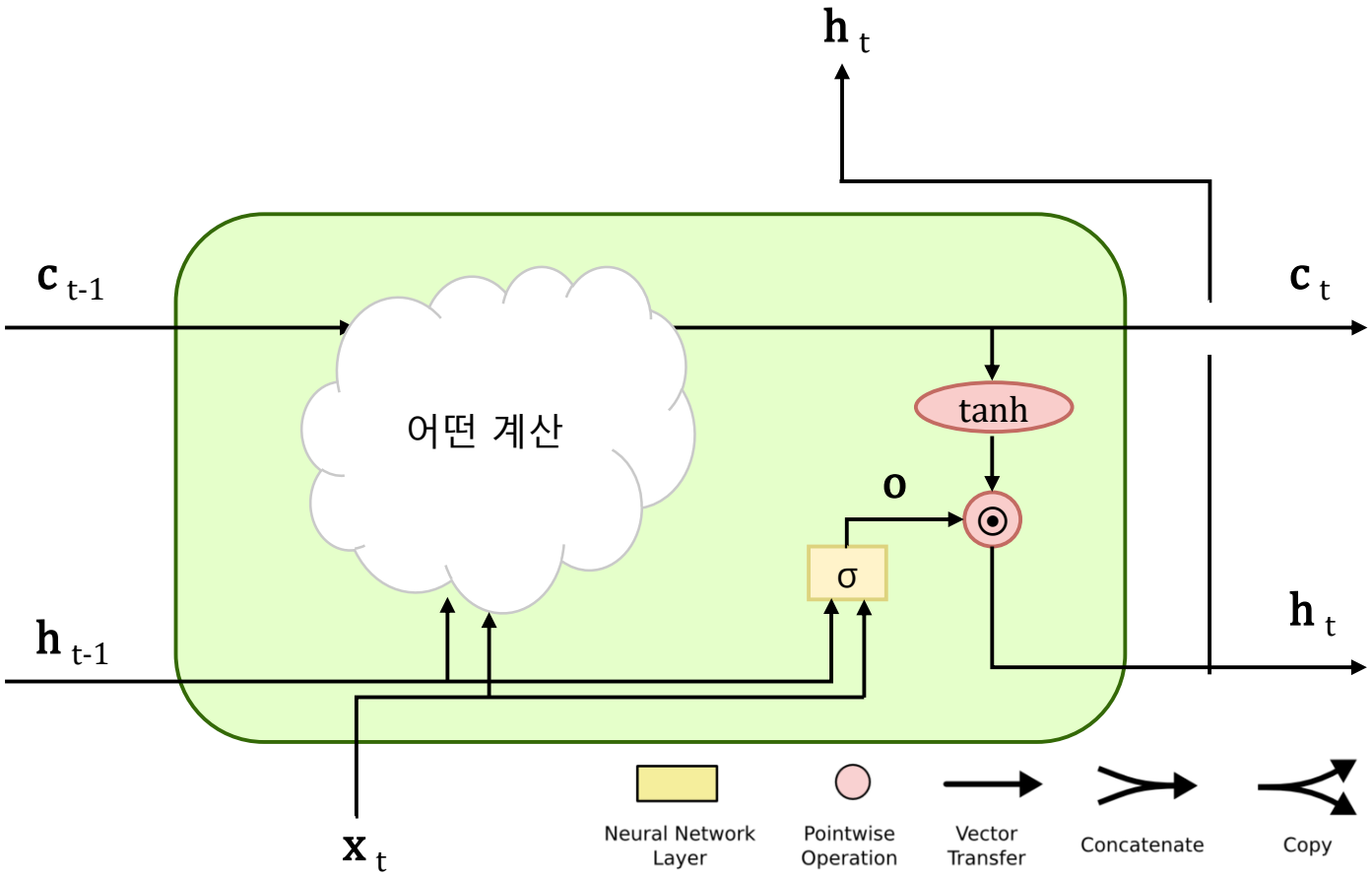
기억 셀 c_t를 바탕으로 은닉상태 h_t를 계산하는 LSTM 계층



LSTM Review

Long Short Term Memory (LSTM)

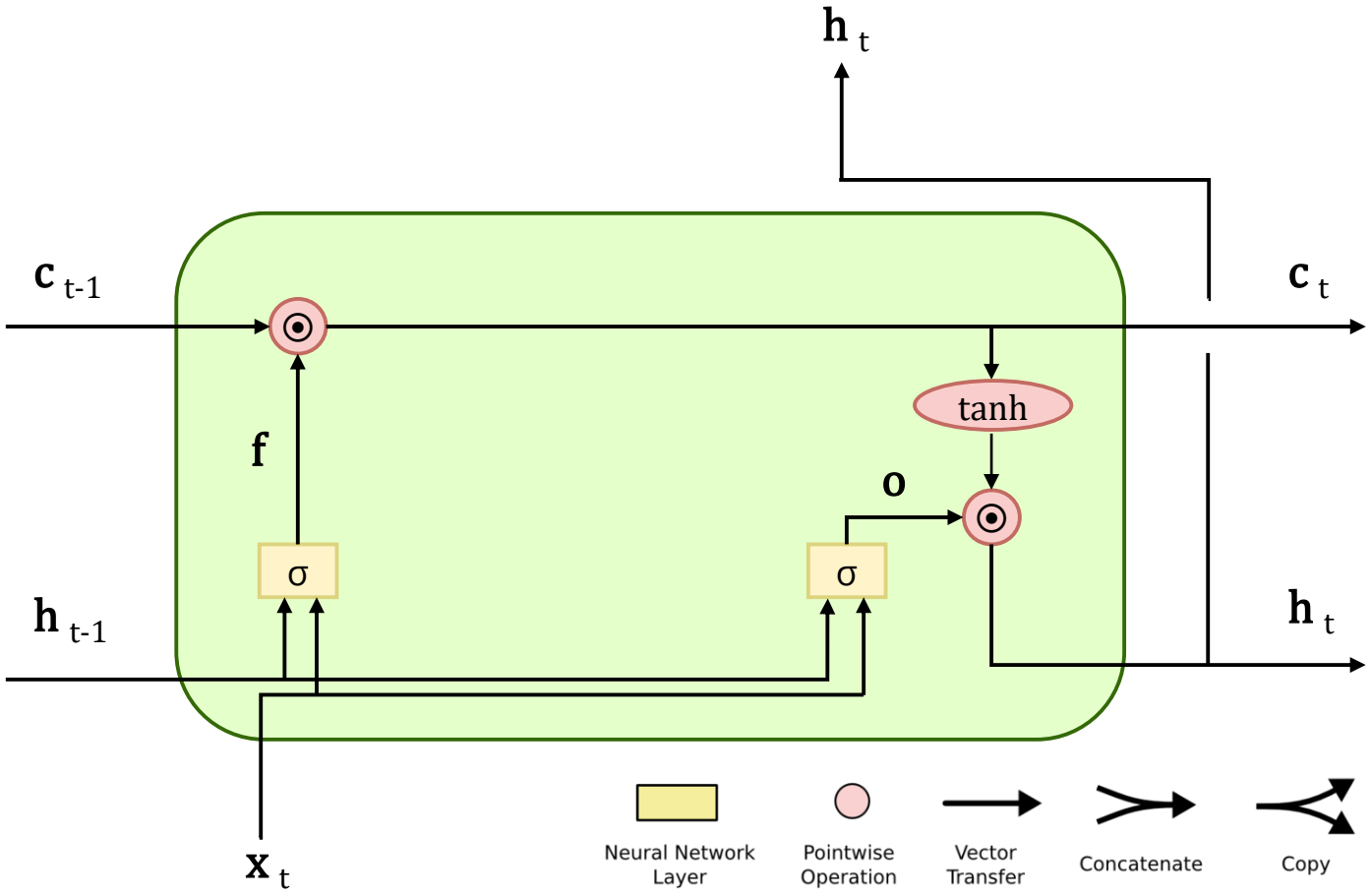
output 게이트 추가 (o gate)



LSTM Review

Long Short Term Memory (LSTM)

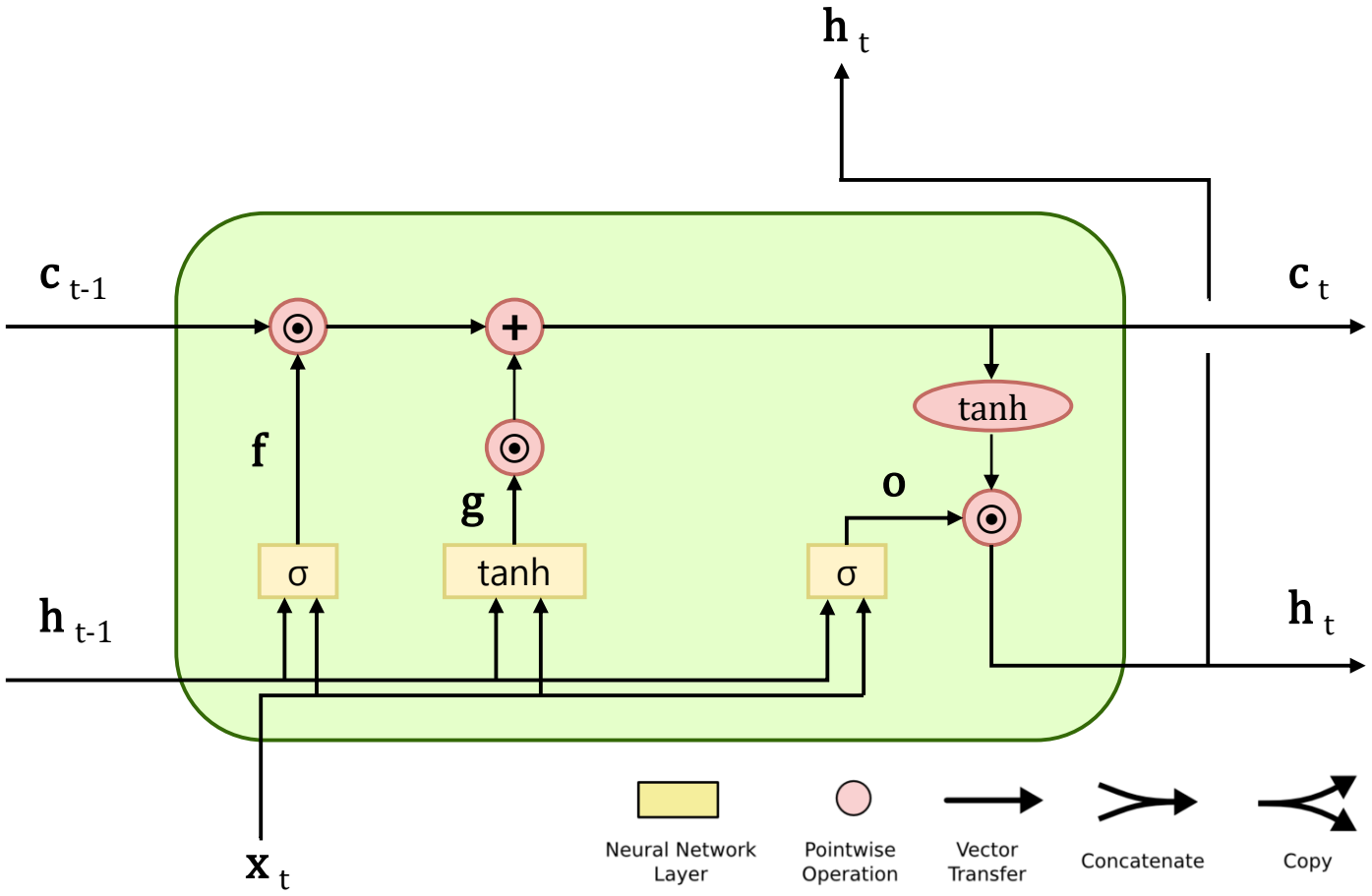
forget 게이트 추가 (f gate)



LSTM Review

Long Short Term Memory (LSTM)

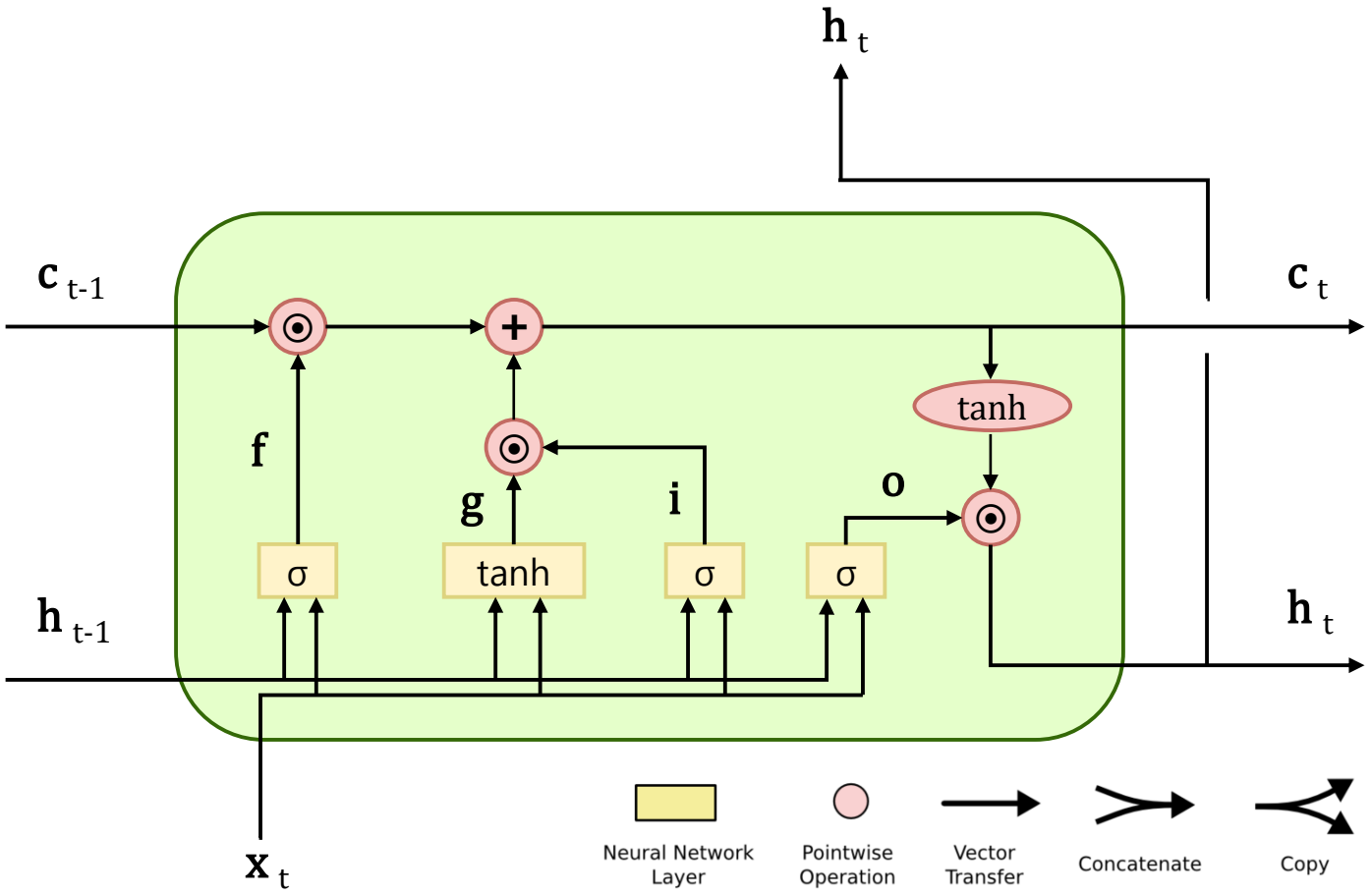
새로운 기억 셀에 필요한 정보를 추가 (g gate)



LSTM Review

Long Short Term Memory (LSTM)

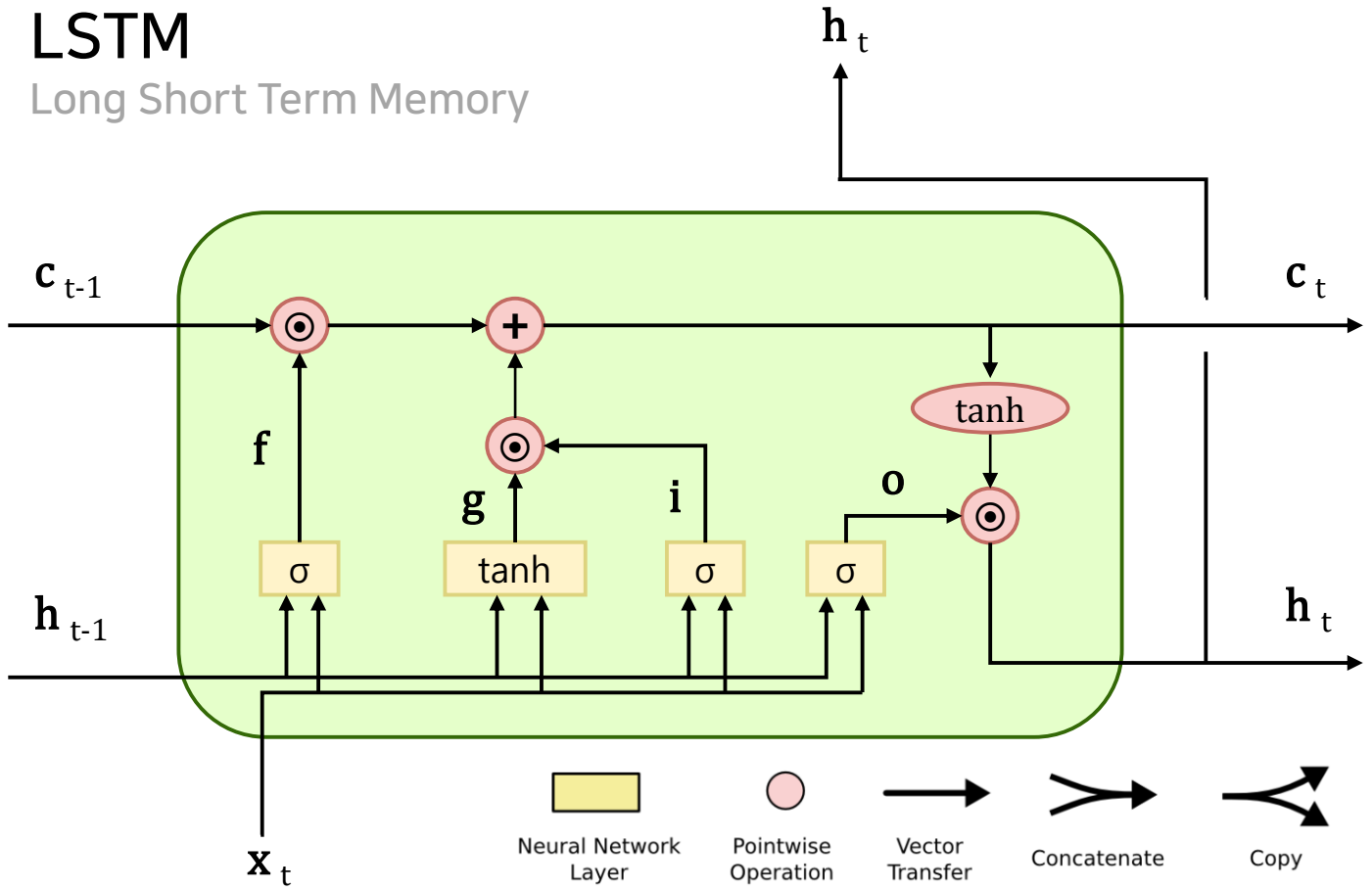
Input 게이트 추가 (i gate)



LSTM Review

Long Short Term Memory (LSTM)

LSTM의 계산 그래프



$$f = \sigma(x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)})$$

$$g = \tanh(x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)})$$

$$i = \sigma(x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)})$$

$$o = \sigma(x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)})$$

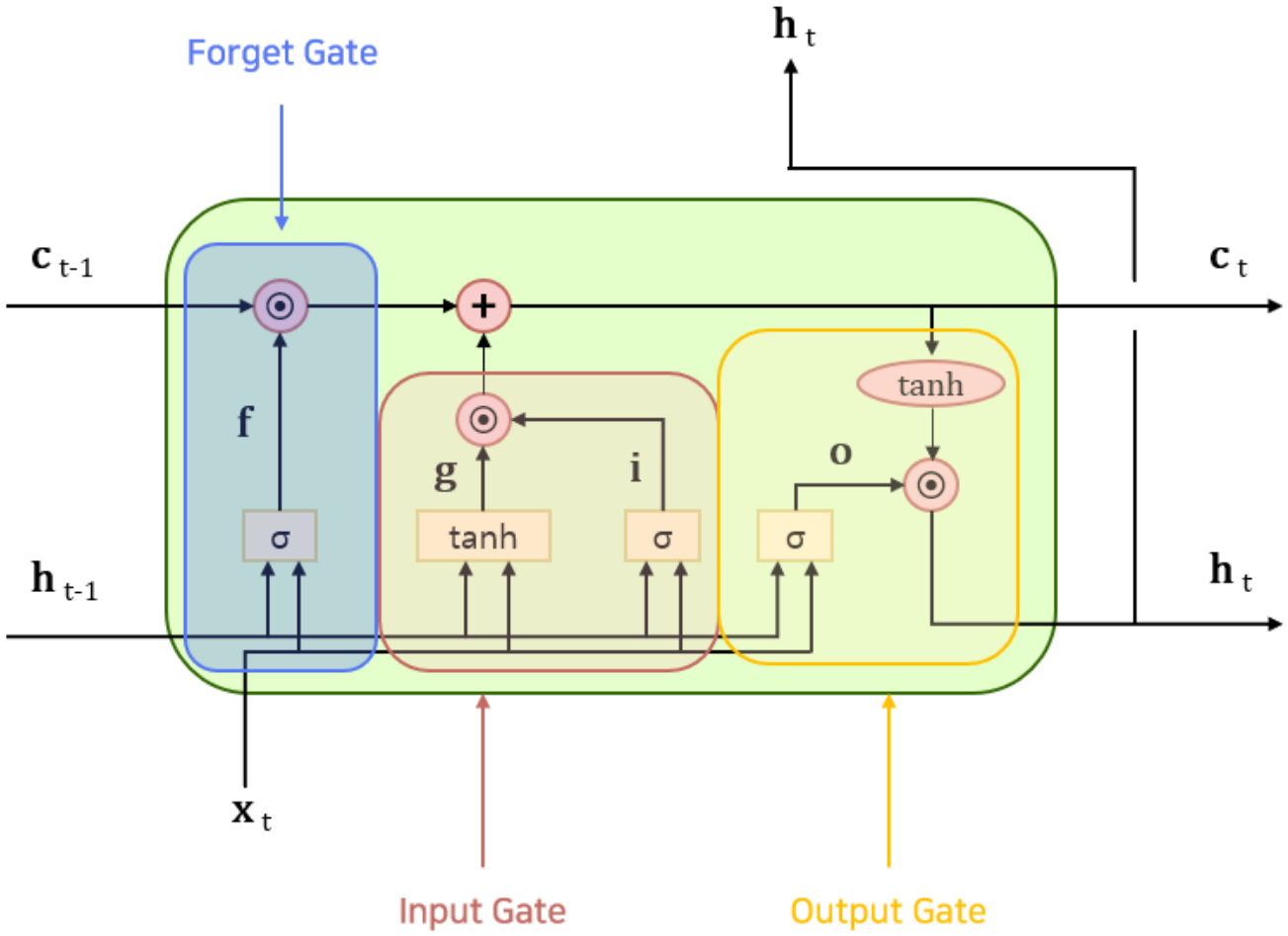
$$c_t = f \odot c_{t-1} + g \odot i$$

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

LSTM Review

Long Short Term Memory (LSTM)

LSTM의 계산 그래프



$$f = \sigma(x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)})$$

$$g = \tanh(x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)})$$

$$i = \sigma(x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)})$$

$$o = \sigma(x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)})$$

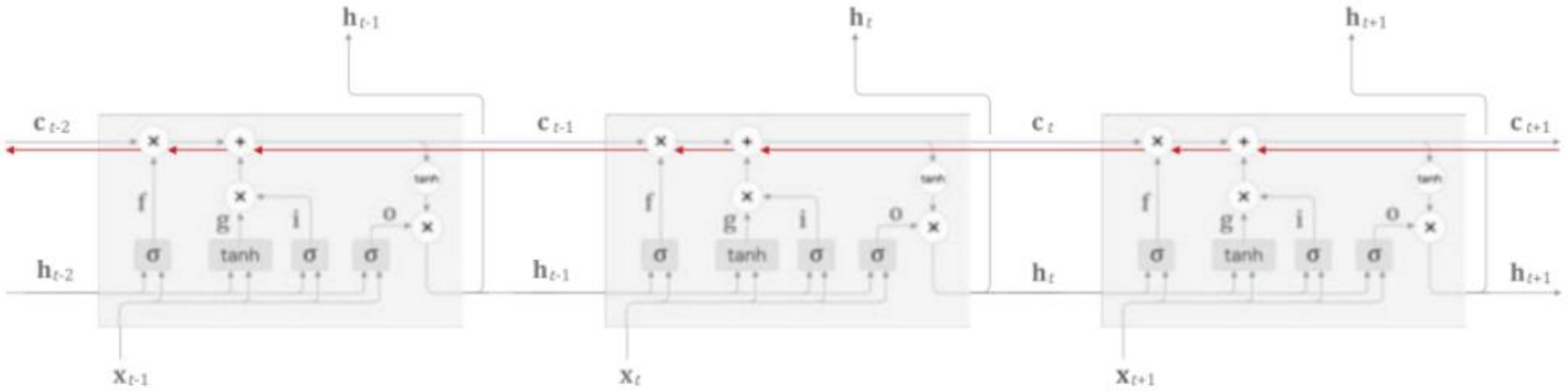
$$c_t = f \odot c_{t-1} + g \odot i$$

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

LSTM

▪ LSTM의 기울기 흐름

LSTM의 구조는 설명했지만, 이것이 어떤 원리로 기울기 소실을 없애주는 걸까? 그 원리는 기억 셀 **c**의 역전파에 주목하면 볼 수 있다.



다음은 기억 셀에만 집중하여, 그 역전파의 흐름을 그린 것이다. 이때 기억 셀의 역전파에서는 '+'와 'X' 노드만을 지나게 된다. '+' 노드는 상류 기울기를 그대로 흘릴 뿐이므로 남는 것은 'X' 노드인데, 이 노드는 **'행렬 곱'이 아닌 아다마르 곱을 계산**한다. RNN의 역전파에서는 똑같은 가중치 행렬을 사용해서 '행렬 곱'을 반복했고, 그래서 기울기 소실 혹은 폭발이 일어났다. 반면 이번 LSTM의 역전파에서는 '행렬 곱'이 아닌 '원소별 곱'이 이뤄지고, 매 시각 다른 게이트 값을 이용해 원소별 곱을 계산한다. 이처럼 **매번 새로운 게이트 값**을 이용하므로 곱셈의 효과가 누적되지 않아 기울기 소실이 일어나기 어려운 것이다.

NOTE_ 위 그림의 'X' 노드의 계산은 forget 게이트가 제어한다. 역전파 계산시 forget 게이트의 출력과 상류 기울기의 곱이 계산되므로 forget 게이트가 '잊어야 한다'고 판단한 기억 셀의 원소에 대해서는 그 기울기가 작아지고, '잊어서는 안 된다'고 판단한 원소에 대해서는 그 기울기가 약화되지 않은 채로 과거 방향으로 전해진다. 따라서 중요한 정보의 기울기는 소실 없이 전파된다.

LSTM 구현

▪ Affine 변환

이제 LSTM을 구현해보자. 다음은 LSTM에서 수행하는 계산을 정리한 수식들이다.

$$f = \sigma(x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)})$$

$$g = \tanh(x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)})$$

$$i = \sigma(x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)})$$

$$o = \sigma(x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)})$$

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

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

주목할 부분은 위에서부터 4개의 f, g, i, o의 수식에 포함된 아핀 변환affine transformation이다. (여기서 아핀 변환이란 $xW_x + hW_h + b$ 형태의 식을 말한다) 네 수식에서는 아핀 변환을 개별적으로 수행하지만, 이를 하나의 식으로 정리해 계산할 수 있다. 그 방법을 시각적으로 설명한 것이 바로 다음 슬라이드의 그림이다.

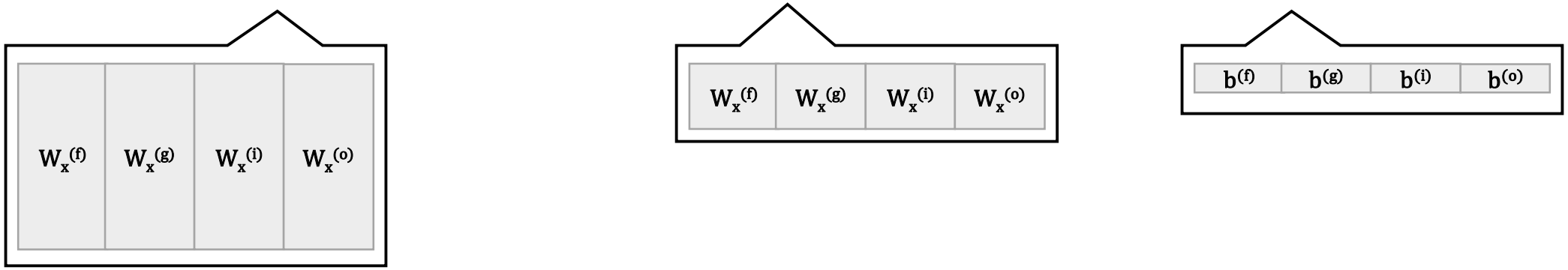
LSTM 구현

- **Affine 변환**

$$x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)}$$
$$x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)}$$
$$x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)}$$
$$x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)}$$

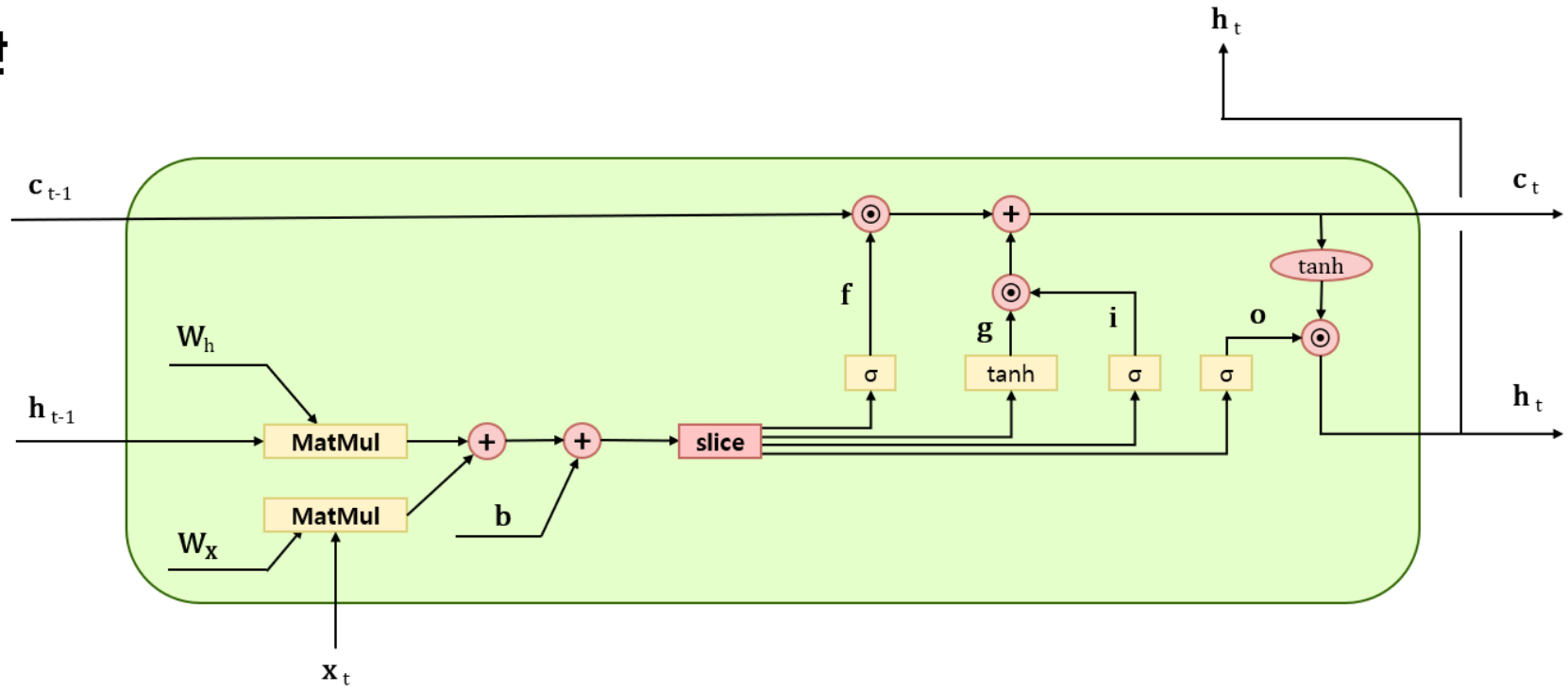
$$x_t [W_x^{(f)} \; W_x^{(g)} \; W_x^{(i)} \; W_x^{(o)}] + h_{t-1} [W_x^{(f)} \; W_x^{(g)} \; W_x^{(i)} \; W_x^{(o)}] + [b^{(f)} \; b^{(f)} \; b^{(i)} \; b^{(o)}]$$

$$x_t W_x + h_{t-1} W_h + b$$



LSTM 구현

- **Affine 변환**



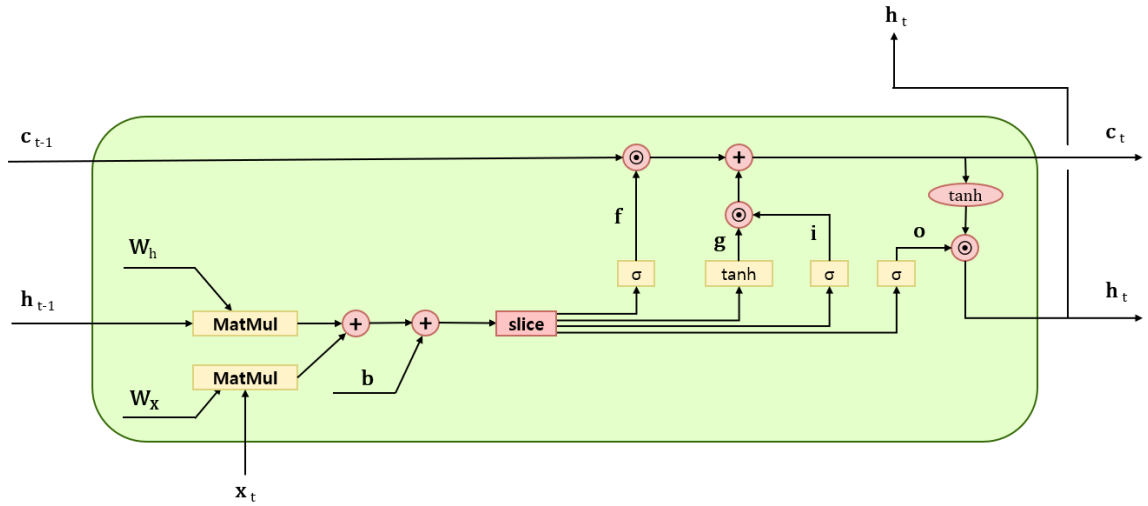
4개분의 가중치를 모아 아핀 변환을 수행하는 LSTM의 계산 그래프

그림에서 보듯 4개의 가중치와 편향을 하나로 모아서 처리할 수 있고, 그렇게 하면 개별적으로 총 4번을 수행하던 아핀 변환을 단 1회의 계산으로 끝마칠 수 있다. 일반적으로 행렬 라이브러리는 '큰 행렬'을 한꺼번에 계산할 때가 각각을 따로 계산할 때보다 빠르기 때문에 속도도 빠르고 소스 코드도 간결해진다. 다음 슬라이드에서는 이를 바탕으로 LSTM의 순전파까지 구현해보자.

LSTM 구현

- **LSTM의 순전파 구현**

```
class LSTM:
    def __init__(self, Wx, Wh, b):
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
        self.cache = None

    def forward(self, x, h_prev, c_prev):
        Wx, Wh, b = self.params
        N, H = h_prev.shape  # N : mini-batch size, H : hidden size

        # Affine Transformation
        # x : NxD, Wx : Dx4H, h_prev : NxH, Wh : Hx4H, affine : Nx4H
        affine = np.matmul(x, Wx) + np.matmul(h_prev, Wh) + b

        # slice
        f = affine[:, :H]
        g = affine[:, H:2*H]
        i = affine[:, 2*H:3*H]
        o = affine[:, 3*H:]

        f = np.sigmoid(f)
        g = np.tanh(g)
        i = np.sigmoid(i)
        o = np.sigmoid(o)

        c_next = f * c_prev + g * i
        h_next = o * np.tanh(c_next)

        self.cache = (x, h_prev, c_prev, i, f, g, o, c_next)
        return h_next, c_next
```

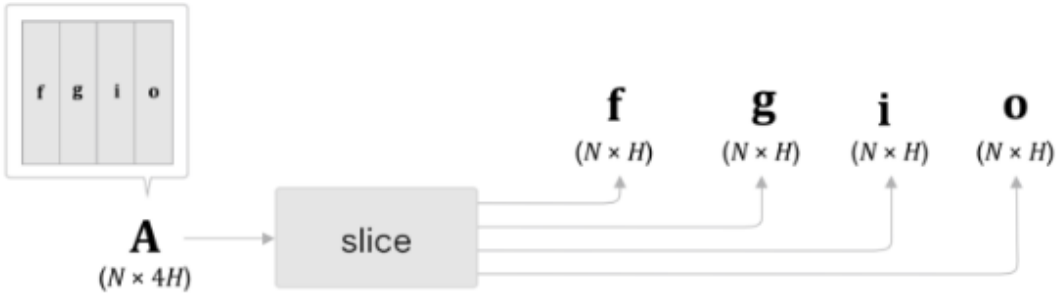


4개분의 가중치를 모아 아핀 변환을 수행하는 LSTM의 계산 그래프

LSTM 구현

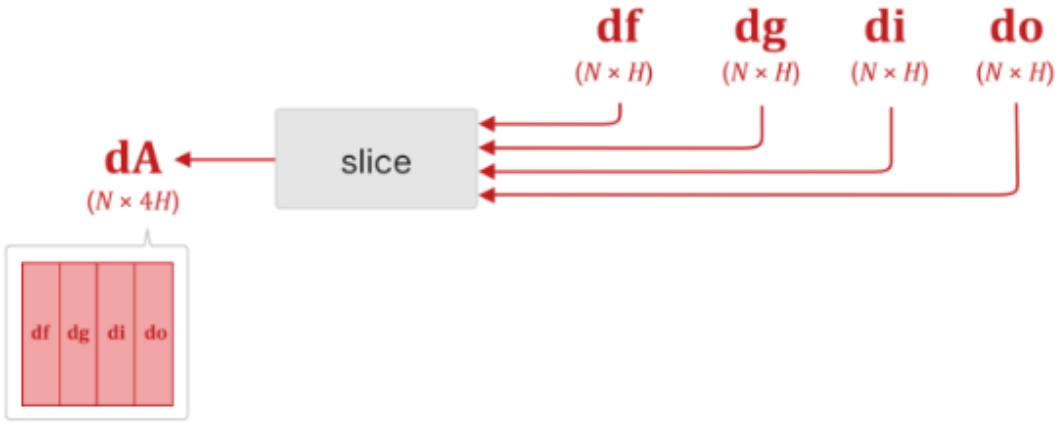
▪ LSTM의 역전파

LSTM의 역전파는 앞의 그림의 계산 그래프를 역방향으로 전파해 구할 수 있다. 앞에서까지 한 지식을 활용하면 쉽게 구현 가능하다. 다만, slice 노드에 대한 처리는 다음과 같이 처리한다.



다음 그림에서 보듯 slice 노드의 역전파에서는 4개의 행렬을 연결한다. 그림에서는 **df, dg, di, do**를 연결해서 **dA**를 만들었다. NumPy가 제공하는 np.hstack() 메서드를 사용하면 인수로 주어진 배열들을 가로로 연결해준다. (세로 연결은 np.vstack()로 한다.) 따라서 다음의 처리를 다음 한 줄로 끝낼 수 있다.

```
dA = np.hstack((df, dg, di, do))
```

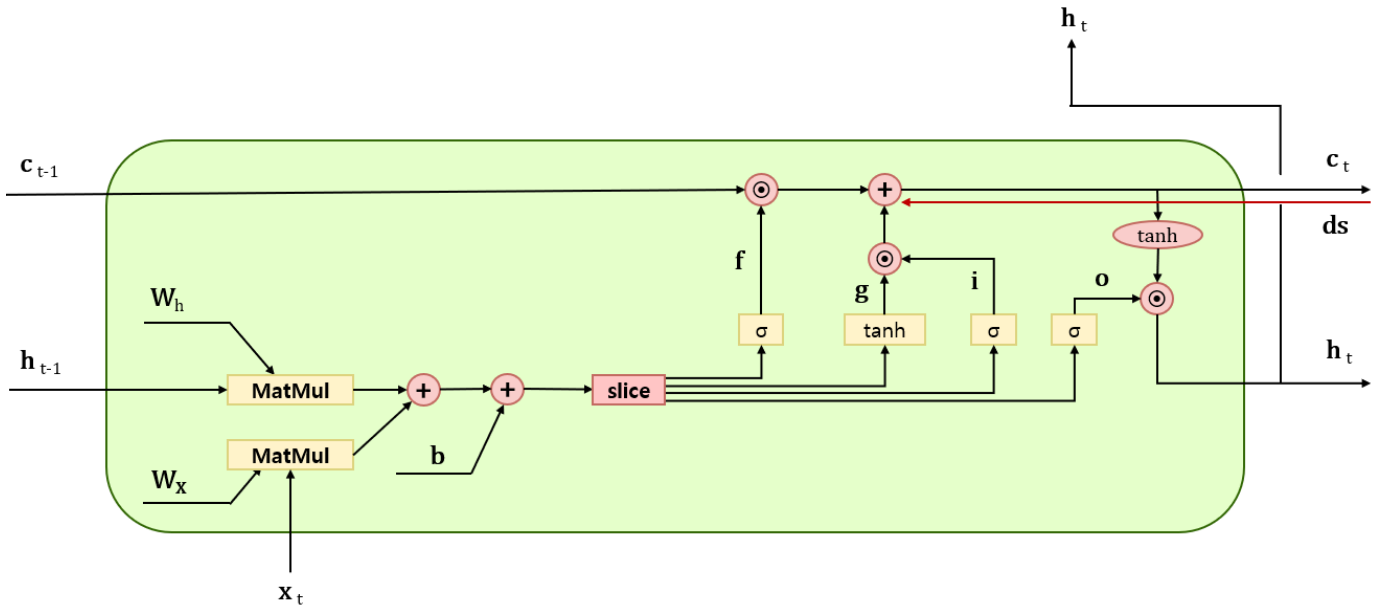


이를 이용해서 구현한 LSTM의 역전파는 다음 슬라이드에서 설명한다.

LSTM 구현

- **LSTM의 역전파**

```
def backward(self, dh_next, dc_next):
    Wx, Wh, b = self.params
    x, h_prev, c_prev, i, f, g, o , c_next = self.cache

    tanh_c_next = np.tanh(c_next)

    ds = dc_next+(dh_next*o) * (1-tanh_c_next**2)

    dc_prev = ds * f

    di = ds * g
    df = ds * c_prev
    dg = ds * i
    do = dh_next * tanh_c_next

    di *= i * (1 - i)  # dSigmoid
    df *= f * (1 - f)  # dSigmoid
    do *= o * (1 - o)  # dSigmoid
    dg *= (1 - g ** 2)  # dtanh

    dA = np.hstack((df, dg, di, do))

    dWh = np.dot(h_prev.T, dA)
    dWx = np.dot(x.T, dA)
    db = dA.sum(axis=0)

    self.grads[0][...] = dWx
    self.grads[1][...] = dWh
    self.grads[2][...] = db

    dx = np.dot(dA, Wh.T)
    dh_prev = np.dot(dA, Wh.T)

    return dx, dh_prev, dc_prev
```



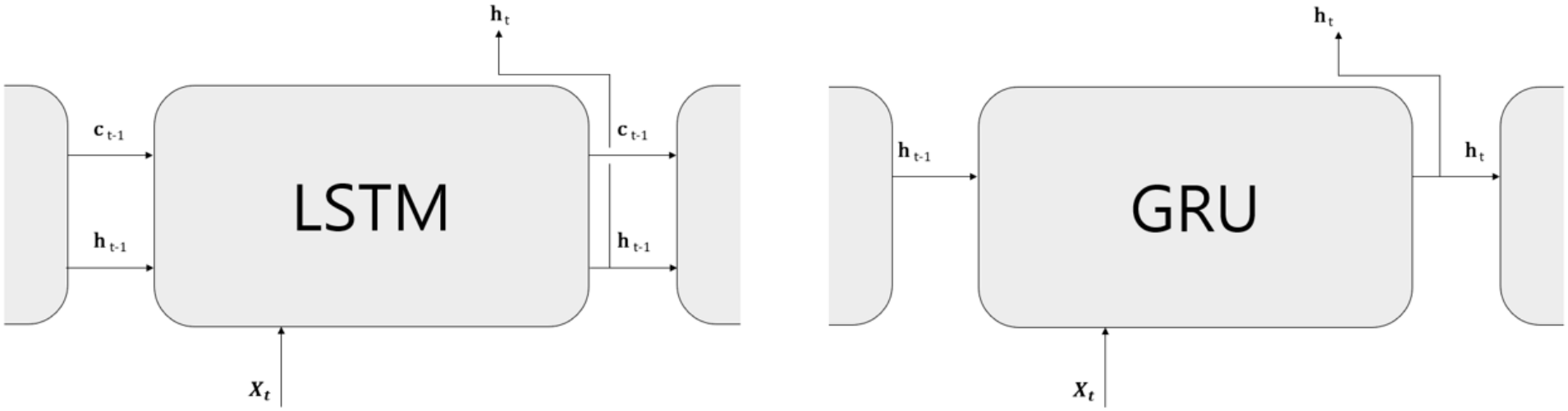
4개분의 가중치를 모아 아핀 변환을 수행하는 LSTM의 계산 그래프

GRU

▪ GRU (Gate Recurrent Unit)

앞에서까지 LSTM에 대해 자세하게 설명했다. LSTM은 아주 좋은 계층이지만 매개변수가 많아서 계산이 오래 걸리는 게 단점이다. 그래서 최근에는 LSTM을 대신할 '게이트가 추가된 RNN'이 많이 제안되고 있다. 이번에는 그 중 유명하고 검증된 GRU**Gated Recurrent Unit** 라는 유명하고 검증된 '게이트가 추가된 RNN'을 소개한다. LSTM의 게이트를 사용한다는 개념은 유지한 채, 매개변수를 줄여 계산 시간을 줄여준다고 한다. LSTM과 GRU의 인터페이스만 비교하더라도 둘의 차이점이 명확하게 드러난다.

GRU의 인터페이스



LSTM과 GRU 비교

GRU

▪ GRU의 계산 그래프

그럼 GRU 내부에서 수행하는 계산을 살펴보자. GRU에서 수행하는 계산을 수식으로 먼저 본 후, 그에 해당하는 계산 그래프를 보자.

$$\mathbf{z} = \sigma(\mathbf{x}_t\mathbf{W}_x^{(z)} + \mathbf{h}_{t-1}\mathbf{W}_h^{(z)} + \mathbf{b}^{(z)})$$

$$\mathbf{r} = \sigma(\mathbf{x}_t\mathbf{W}_x^{(r)} + \mathbf{h}_{t-1}\mathbf{W}_h^{(r)} + \mathbf{b}^{(r)})$$

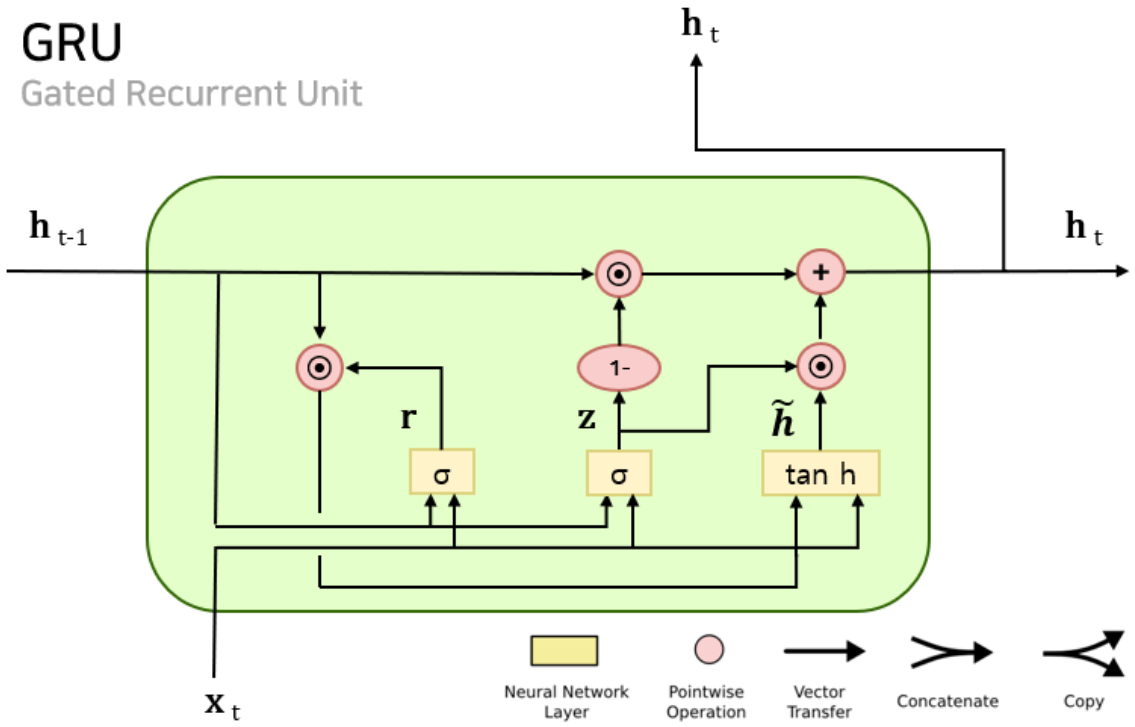
$$\tilde{\boldsymbol{h}} = \tanh(\mathbf{x}_t\mathbf{W}_x + (\mathbf{r} \odot \mathbf{h}_{t-1})\mathbf{W}_h + \mathbf{b})$$

$$\mathbf{h}_t = (1\text{-}\mathbf{z}) \odot \mathbf{h}_{t-1} + \mathbf{z} \odot \tilde{\boldsymbol{h}}$$

GRU에서 수행하는 계산은 이 4개의 식으로 표현된다. 6개였던 LSTM에 비해 간단해진 것을 확인할 수 있다. 그리고 GRU의 계산 그래프는 다음 슬라이드의 그림과 같다.

GRU

- **GRU의 계산 그래프**

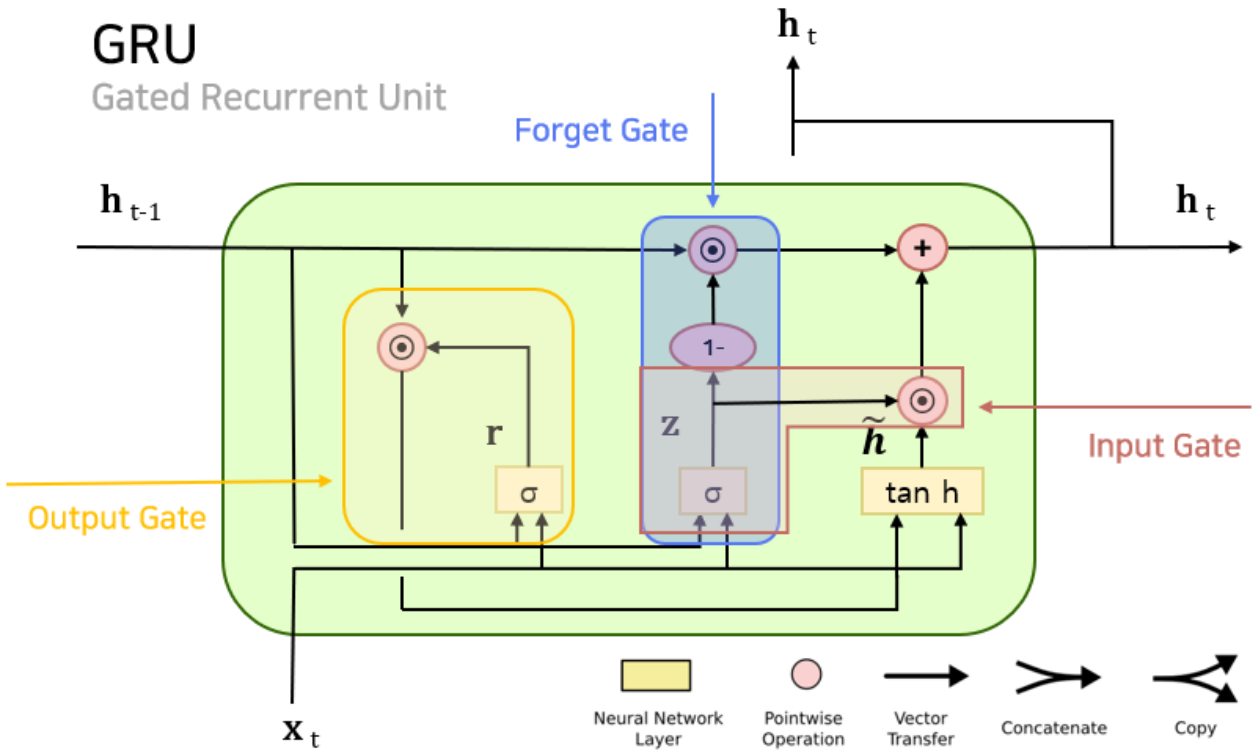


GRU의 계산 그래프

GRU는 이처럼 LSTM을 '더 단순하게 ' 만든 아키텍처이다. 따라서 LSTM보다 계산 비용을 줄이고 매개변수 수도 줄일 수 있다. 어떠한 원리로 돌아가는지는 다음 슬라이드에서 살펴보자.

GRU

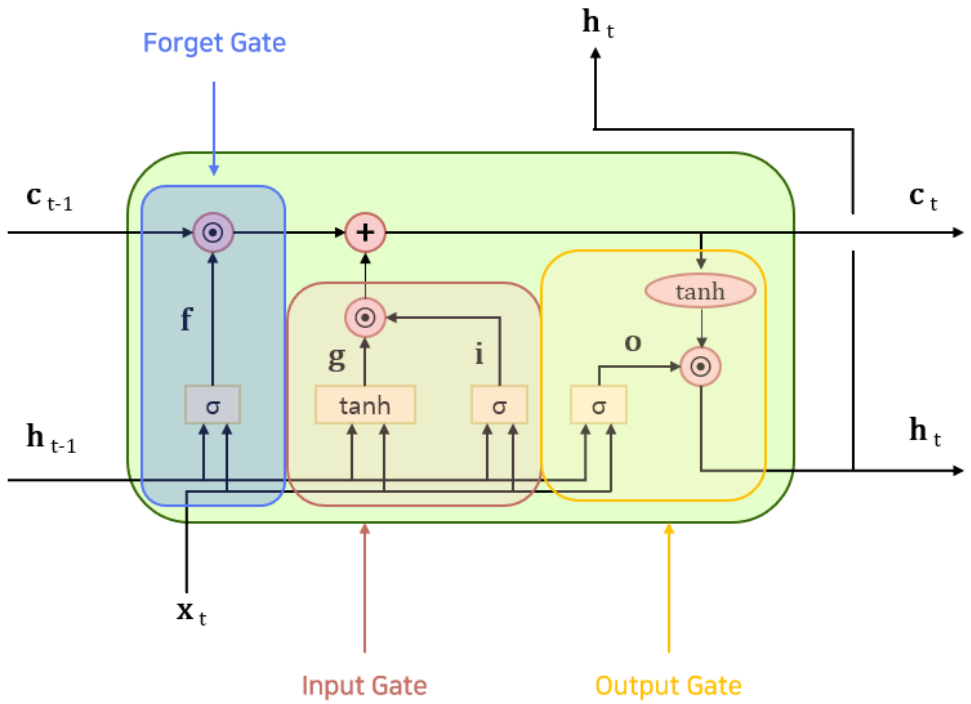
- **GRU의 계산 그래프**



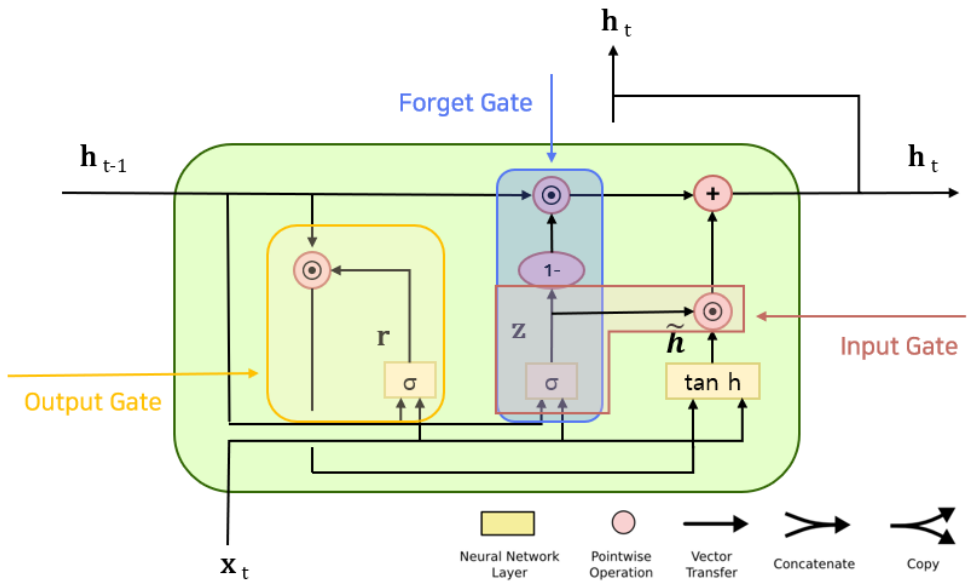
GRU의 계산 그래프

위의 그림처럼 GRU에는 기억 셀은 없고, 시간 방향으로 전파하는 것은 은닉 상태 **h**뿐이다. 그리고 **r**과 **z**라는 개의 게이트를 사용한다. 여기서 **r**은 reset 게이트, **z**는 update 게이트이다. Reset게이트 **r**은 과거의 은닉 상태를 얼마나 '무시'할지를 정한다. 만약 **r**이 0이면 $\tilde{\boldsymbol{h}} = \tanh(\mathbf{x}_t\mathbf{W}_x + (\mathbf{r} \odot \mathbf{h}_{t-1})\mathbf{W}_h + \mathbf{b})$ 으로부터, 새로운 은닉상태 $\tilde{\boldsymbol{h}}$ 는 입력 $\mathbf{x}_t$만으로 결정된다. 즉, 과거의 은닉 상태를 완전히 무시된다. 한편, update 게이트는 은닉 상태를 갱신하는 게이트이다. **LSTM의 forget 게이트와 input 게이트의 2가지 역할을 혼자 담당하는 것**이다. forget 게이트로써의 기능은 $(1\text{-}\mathbf{z}) \odot \mathbf{h}_{t-1}$ 부분이다. 과거의 은닉 상태에서 잊어야 할 정보를 삭제한다. 그리고 input 게이트로써의 기능은 $\mathbf{z} \odot \tilde{\boldsymbol{h}}$ 부분이다. 이에 따라 새로 추가된 정보에 input 게이트의 가중치를 부여한다.

LSTM vs GRU



LSTM의 계산 그래프



GRU의 계산 그래프

LSTM과 GRU 중 어느 쪽을 사용해야 하는지를 묻는다면, 답은 주어진 문제와 하이퍼파라미터 설정에 따라 승자가 달라진다고 대답할 수 있다. 최근 연구세어슷 LSTM이 많이 사용된다. 한편, GRU도 꾸준히 인기를 끌고 있다. GRU는 매개변수가 적고 계산량도 적기 때문에, 데이터셋이 작거나 모델 설계 시 반복 시도를 많이 해야 할 경우 특히 접합할 수 있다.

부록 - Hardamard Product

- **Hardamard Product**

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \circ \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}\,b_{11} & a_{12}\,b_{12} & a_{13}\,b_{13} \\ a_{21}\,b_{21} & a_{22}\,b_{22} & a_{23}\,b_{23} \\ a_{31}\,b_{31} & a_{32}\,b_{32} & a_{33}\,b_{33} \end{bmatrix}$$

일반 행렬 곱은 mxn 행렬과 nxp 꼴의 행렬을 곱하지만, Hardamard product는 mxn과 mxn의 같은 꼴을 가지는 행렬끼리 같은 위치의 원소끼리 각각 곱한다

부록 - GRU 구현

```
class GRU:
    def __init__(self, Wx, Wh):
        self. Wx, self.Wh = Wx, Wh
        self.dWx, self.dWh = None, None
        self.cache = None

    def forward(self, x, h_prev):
        H, H3 = self.Wh.shape
        Wxz, Wxr, Wx = self.Wx[:, :H], self.Wx[:, H:2 * H], self.Wx[:, 2 * H:]
        Whz, Whr, Wh = self.Wh[:, :H], self.Wh[:, H:2 * H], self.Wh[:, 2 * H:]

        z = np.sigmoid(np.dot(x, Wxz) + np.dot(h_prev, Whz))
        r = np.sigmoid(np.dot(x, Wxr) + np.dot(h_prev, Whr))
        h_hat = np.tanh(np.dot(x, Wx) + np.dot(r*h_prev, Wh))
        h_next = (1-z) * h_prev + z * h_hat

        self.cache = (x, h_prev, z, r, h_hat)

        return h_next
```

```
def backward(self, dh_next):
    H, H3 = self.Wh.shape
    Wxz, Wxr, Wx = self.Wx[:, :H], self.Wx[:, H:2 * H], self.Wx[:, 2 * H:]
    Whz, Whr, Wh = self.Wh[:, :H], self.Wh[:, H:2 * H], self.Wh[:, 2 * H:]
    x, h_prev, z, r, h_hat = self.cache

    dh_hat =dh_next * z
    dh_prev = dh_next * (1-z)

    # tanh
    dt = dh_hat * (1 - h_hat ** 2)
    dWh = np.dot((r * h_prev).T, dt)
    dhr = np.dot(dt, Wh.T)
    dWx = np.dot(x.T, dt)
    dx = np.dot(dt, Wx.T)
    dh_prev += r * dhr

    # update gate(z)
    dz = dh_next * h_hat - dh_next * h_prev
    dt = dz * z * (1-z)
    dWhz = np.dot(h_prev.T, dt)
    dh_prev += np.dot(dt, Whz.T)
    dWxz = np.dot(x.T, dt)
    dx += np.dot(dt, Wxz.T)

    # rest gate(r)
    dr = dhr * h_prev
    dt = dr * r * (1-r)
    dWhr = np.dot(h_prev.T, dt)
    dh_prev += np.dot(dt, Whr.T)
    dWxr = np.dot(x.T, dt)
    dx += np.dot(dt, Wxr.T)

    self.dWx = np.hstack((dWxz, dWxr, dWx))
    self.dWh = np.hstack((dWhz, dWhr, dWh))

    return dx, dh_prev
```