

밑바닥부터 시작하는 딥러닝 2: 6장 게이트가 추가된 RNN

2020/03/19

6. 게이트가 추가된 RNN

- 5장에서 설명한 기본 RNN의 문제점을 알아보고, 이를 대신하는 계층으로써 LSTM, GRU와 같은 '게이트가 추가된 RNN'을 소개
 - 5장의 RNN: 순환 경로를 포함해 과거의 정보를 기억하며, 구조가 단순해 쉽게 구현이 가능
 - 시계열 데이터에서 시간적으로 멀리 떨어진 장기(long term) 의존 관계를 잘 학습할 수 없기 때문에 성능지 좋지 못함.
 - 요즘에는 단순한 RNN 대신 **LSTM이나 GRU**라는 계층이 주로 쓰임
 - LSTM, GRU => 게이트(gate)라는 구조가 더해져 시계열 데이터의 장기 의존 관계를 학습할 수 있다.
 - LSTM의 구조를 살펴보고, 이 구조가 '장기 기억'을 가능하게 하는 매커니즘을 이해
 - LSTM을 사용해 언어 모델을 만들고, 데이터를 학습해보자.

6.1 RNN의 문제점:

1. RNN의 복습

- RNN 계층: 시계열 데이터인 x_t 를 입력하면 h_t 를 출력
 - h_t : RNN 계층의 은닉 상태, 과거의 정보 저장
 - RNN은 바로 이전 시각의 은닉 상태를 이용해 과거 정보를 계승한다.

그림 6-2 RNN 계층의 계산 그래프(MatMul 노드는 행렬 곱을 나타냄)

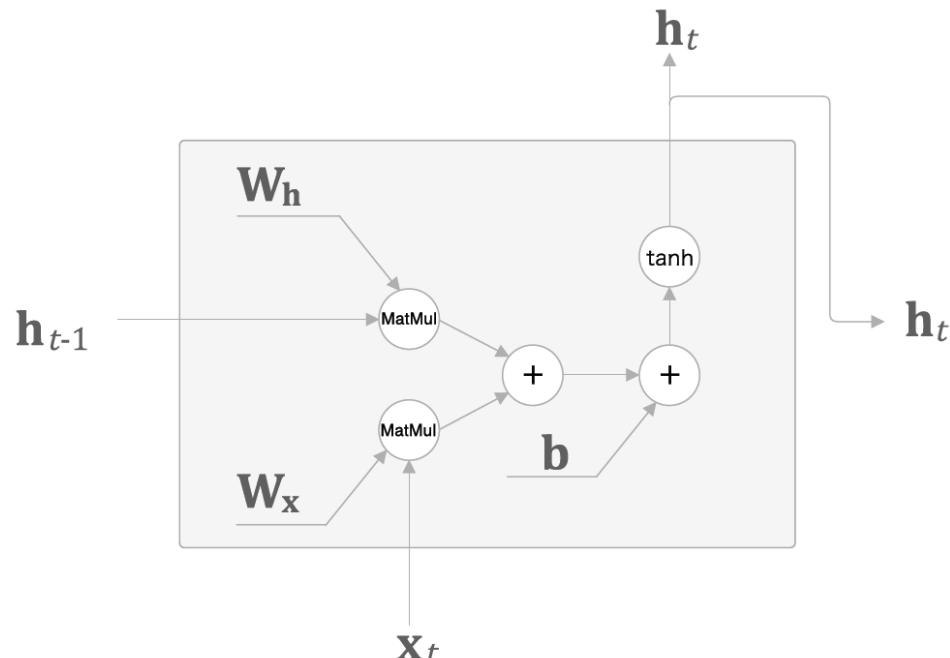
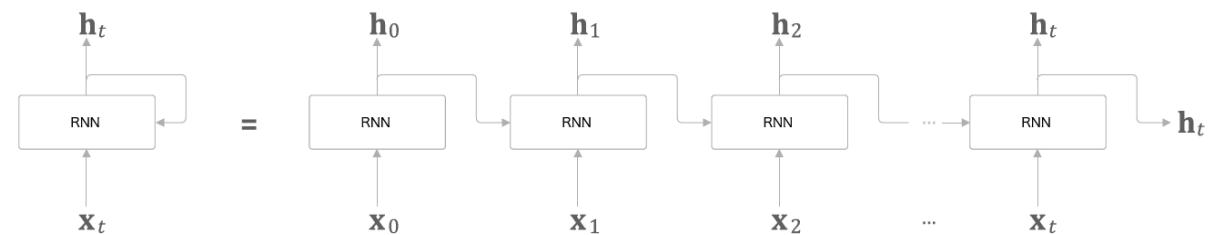
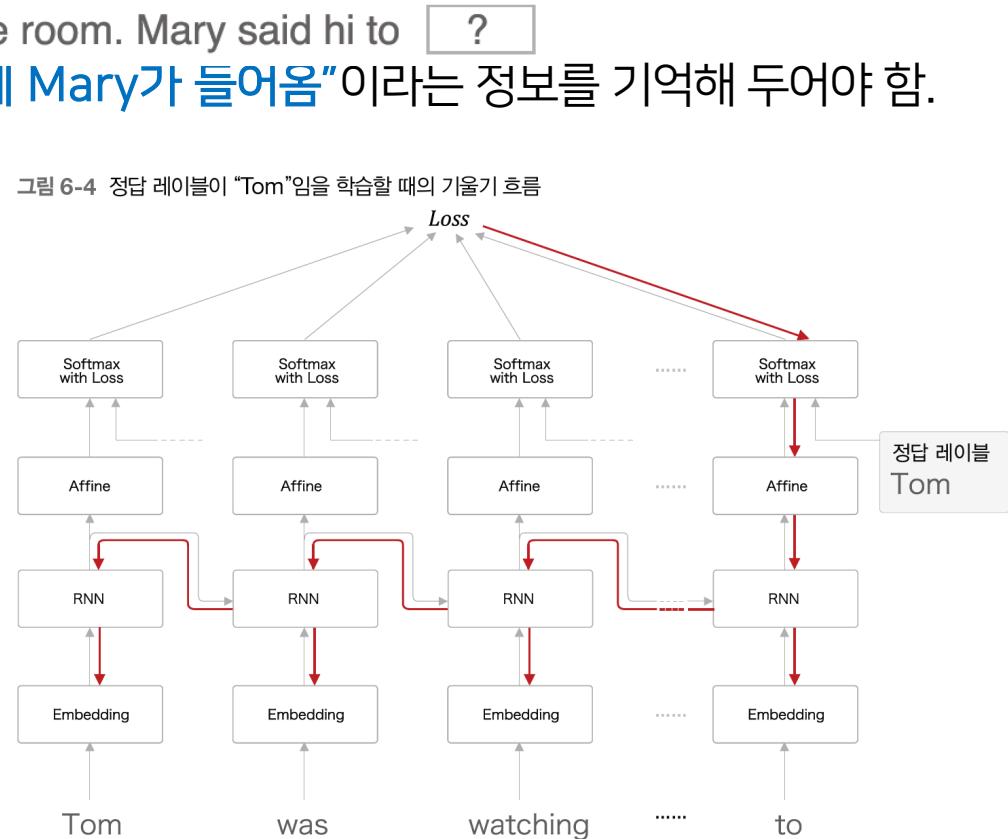


그림 6-1 RNN 계층: 순환을 펼치기 전과 후



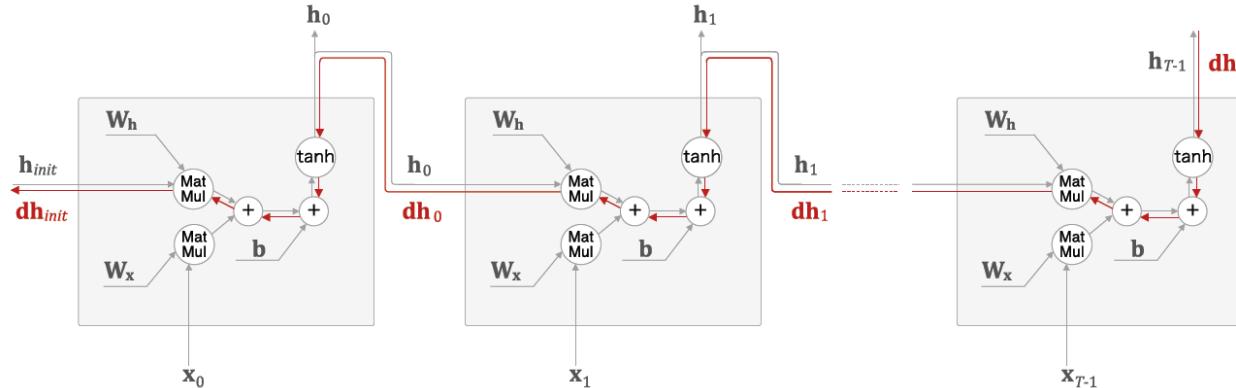
6.1.2 기울기 소실 또는 기울기 폭발

- 언어 모델 => 주어진 단어들을 기초로 다음에 출현할 단어를 예측
 - 앞 장에서는 RNN을 사용해 언어 모델을 구현(RNNLM)
- Tom was watching TV in his room. Mary came into the room. Mary said hi to ?
- 올바르게 대답하려면 “Tom이 방에서 TV를 보고 있음”과 “그 방에 Mary가 들어옴”이라는 정보를 기억해 두어야 함.
 - 즉, 이런 정보를 RNN 계층의 은닉 상태에 인코딩해 보관해두어야 함.
 - 이러한 모델을 학습할 때에는 정답 레이블이 주어진 시점부터 과거 방향으로 기울기를 전달하게 된다.
 - RNN 계층이 과거 방향으로 ‘의미 있는 기울기’를 전달해 시간 방향의 의존 관계를 학습
 - 이때 기울기는 학습해야 할 의미가 있는 정보를 가지고 있고, 이를 과거로 전달함으로써 장기 의존 관계를 학습
 - 그런데 만약 이 기울기가 중간에 사그라들면(거의 아무런 정보도 남아 있지 않게 되면) 가중치 매개 변수는 전혀 갱신되지 않게 된다.
 - 즉, 장기 의존 관계를 학습할 수 없게 된다.
 - 하지만 우리가 만든 단순한 RNN 계층에서는 시간을 거슬러 올라갈수록 기울기가 작아지거나 커지는 문제가 발생한다.



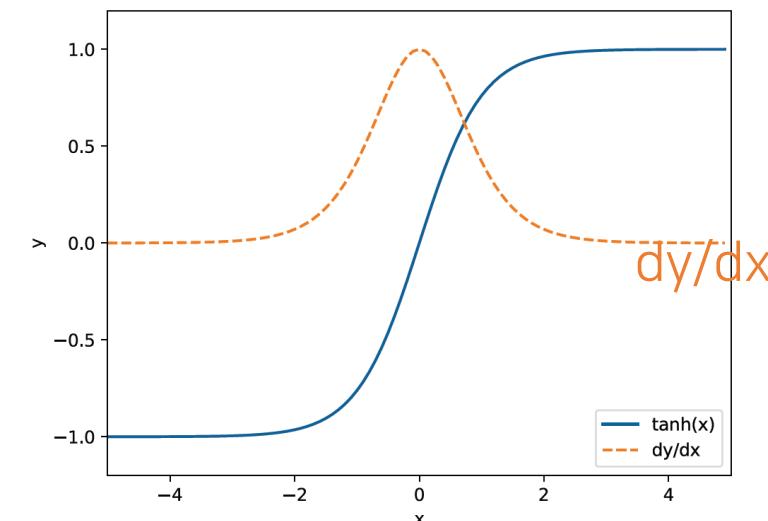
[29] Talathi, Sachin S., and Aniket Vartak : "Improving performance of recurrent neural network with relu nonlinearity." arXiv preprint arXiv:1511.03771(2015).

6.1.3 기울기 소실과 기울기 폭발의 원인



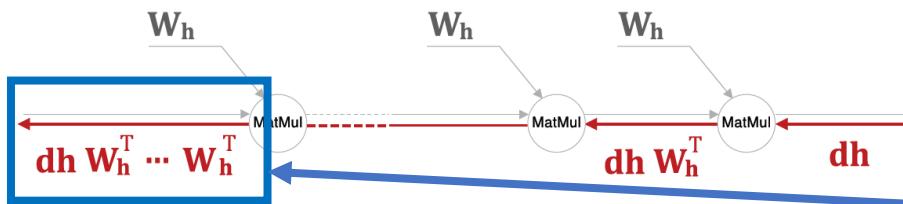
* RNN 계층의 활성화 함수로 tanh를 주로 사용하는데, 이를 ReLU로 바꾸면 기울기 소실을 줄일 수 있음[29]
 - $\text{ReLU}(x) = \max(0, x)$ 이므로 입력 x 가 0 이상이면 역전파 시 기울기를 그대로 흘려보내기 때문에 기울기가 작아지지 않는다.

- 길이가 T 인 시계열 데이터를 가정해보자.
 - 이때 시간 방향 기울기에 주목하면 역전파로 전해지는 기울기는 차례로 'tanh', '+', 'MatMul' 연산을 통과
- 'tanh'의 경우:
 - $y = \tanh(x)$ 의 미분은 $\frac{\partial y}{\partial x} = 1 - y^2$
 - 미분 값이 1.0 이하이며, x 가 0으로부터 멀어질수록 작아짐.
 - 즉, **기울기가 tanh 노드를 지날 때마다 값은 계속 작아짐.**
 - tanh 함수를 T 번 통과하면 기울기도 T 번 반복해 작아짐.



6.1.3 기울기 소실과 기울기 폭발의 원인

- 'MatMul'의 경우:
 - 설명을 단순하게 하기 위해 tanh 노드를 무시하기로 하면, 기울기는 '행렬 곱' 연산에 의해서만 변화

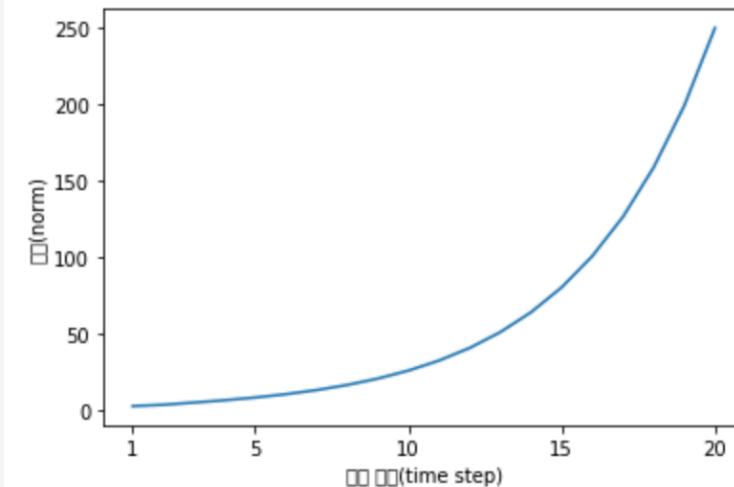


- 상류로부터 dh 라는 기울기가 흘러들어온다고 가정하면, MatMul 노드에서의 역전파는 dhW_h^T 라는 행렬 곱으로 기울기를 계산하며, 같은 계산을 시계열 데이터의 시간 크기만큼 반복
 - 매번 똑같은 가중치 W_h 가 사용됨.
 - 코드를 통해 확인해보면, 기울기의 크기는 시간에 비례해 지수적으로 증가하고 있다. => **기울기 폭발 (exploding gradients)**
 - 이런 기울기 폭발이 일어나면 결국 오버플로를 일으켜 NaN 같은 값을 발생시킴

```
# 6.1.3 기울기 소실과 기울기 폭발의 원인
N = 2      # 미니배치 크기
H = 3      # 은닉 상태 벡터의 차원 수
T = 20     # 시계열 데이터의 길이

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

norm_list = []
for t in range(T):
    dh = np.dot(dh, Wh.T)
    norm = np.sqrt(np.sum(dh**2)) / N
    norm_list.append(norm)
```



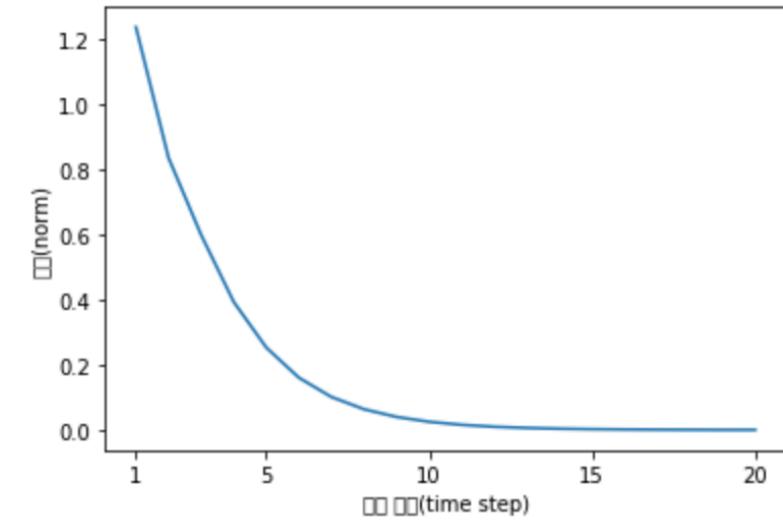
6.1.3 기울기 소실과 기울기 폭발의 원인

- 'MatMul'의 경우:

- W_h 에 0.5를 곱한 뒤 확인해보면 다음과 같은 그래프를 얻을 수 있다.

```
Wh = np.random.randn(H, H) * 0.5
```

- 기울기가 지수적으로 감소하는 **기울기 소실(vanishing gradients)**이 발생하게 된다
- 기울기 소실이 일어나면 기울기가 매우 빠르게 작아지며, 일정 수준 이하로 작아지면 가중치 매개변수가 더 이상 갱신되지 않으므로 장기 의존 관계를 학습할 수 없게 된다.

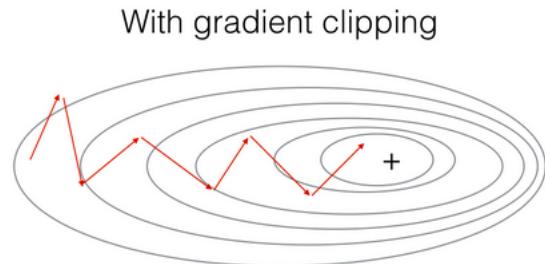
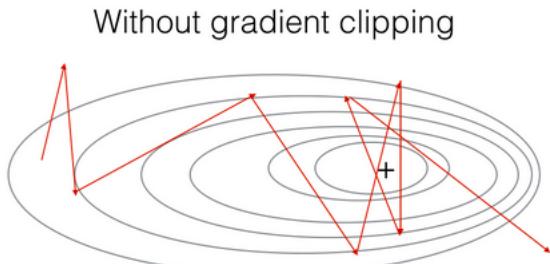


6.1.4 기울기 폭발 대책

- 기울기 클리핑(gradients clipping) 기법:

$$if \|\hat{g}\| \geq threshold:\\ \hat{g} = \frac{threshold}{\|\hat{g}\|} \hat{g}$$

- \hat{g} : 신경망에서 사용되는 모든 매개변수에 대한 기울기를 하나로 처리한다고 가정
 - 두 개의 가중치 $w1$ 과 $w2$ 매개변수를 사용하는 모델이 있으면 이 두 매개변수에 대한 기울기 $dw1$ 과 $dw2$ 를 결합한 것이 \hat{g}
 - 기울기의 L2 norm이 threshold를 초과하면, 기울기를 수정한다.



```
# 6.1.4 기울기 폭발 대책
dw1 = np.random.rand(3, 3) * 10
dw2 = np.random.rand(3, 3) * 10
grads = [dw1, dw2]
max_norm = 5.0
```

grads: 기울기의 리스트
max_norm: threshold

```
def clip_grads(grads, max_norm):
    total_norm = 0
    for grad in grads:
        total_norm += np.sum(grad ** 2)
    total_norm = np.sqrt(total_norm)

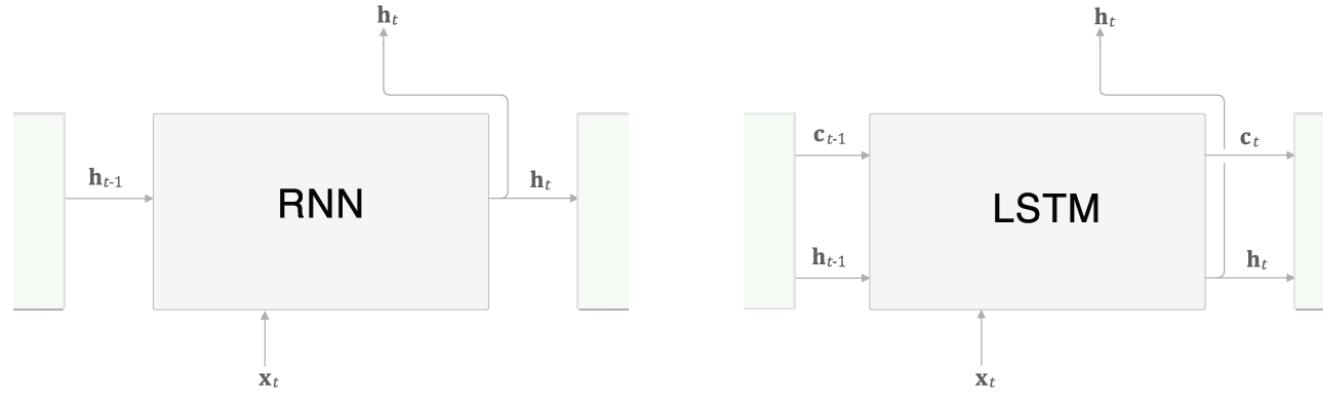
    rate = max_norm / (total_norm + 1e-6)
    if rate < 1:
        for grad in grads:
            grad *= rate
```

```
print('before:', dw1.flatten())
clip_grads(grads, max_norm)
print('after:', dw1.flatten())
```

```
before: [6.49144048 2.78487283 6.76254902 5.90862817 0.23981882 5.58854088
         2.59252447 4.15101197 2.83525082]
after: [1.49503731 0.64138134 1.55747605 1.36081038 0.05523244 1.28709139
        0.59708178 0.95601551 0.65298384]
```

6.2 기울기 소실과 LSTM: 1. LSTM의 인터페이스

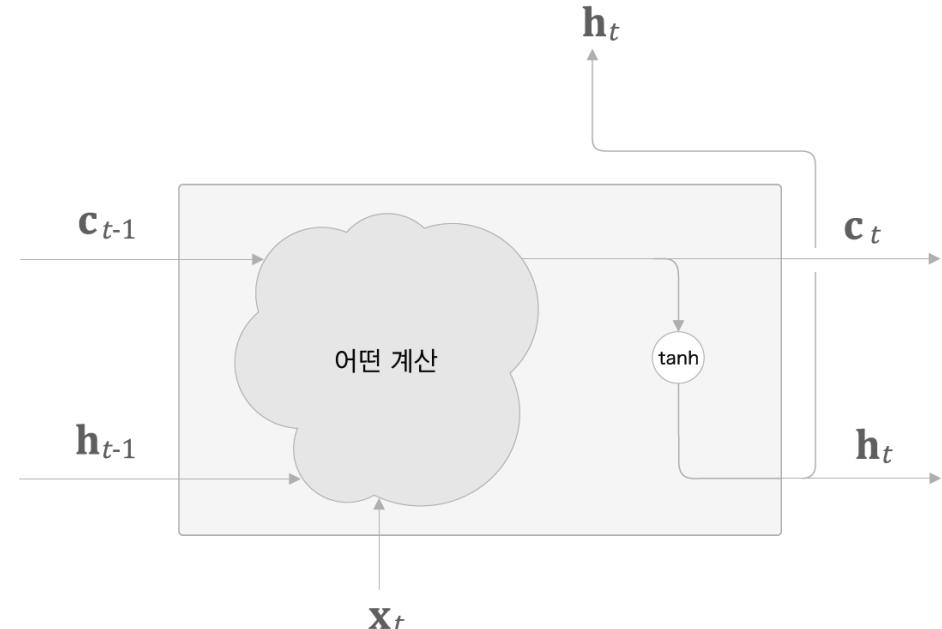
그림 6-11 RNN 계층과 LSTM 계층 비교



- LSTM에는 **c**라는 경로가 추가된다.
 - **c: 기억 셀(memory cell), LSTM 전용의 기억 메커니즘**
 - 기억 셀은 데이터를 자기 자신으로만(LSTM 계층 내에서만) 주고받는다.
 - 즉, LSTM 계층 내에서만 완결되고, **다른 계층으로는 출력하지 않는다.**
 - LSTM의 은닉 상태 h 는 RNN 계층과 마찬가지로 다른 계층으로 출력된다.

6.2.2 LSTM 계층 조립하기

- LSTM의 계층을 살펴보자.
 - **기억 셀 c_t** : 시각 t에서의 LSTM의 기억이 저장
 - 과거로부터 시작 t까지에 필요한 모든 정보가 저장돼 있다고 가정한다.
 - 필요한 정보를 모두 간직한 이 기억을 바탕으로 외부 계층과 다음 시각의 LSTM에 **은닉 상태 h_t** 를 출력한다.
 - 이때 h_t 는 기억 셀 c_t 의 값을 \tanh 함수로 변환한 값
 - 즉 갱신된 c_t 를 사용해 은닉 상태 h_t 를 계산한다.
 - 현재의 기억 셀 c_t 는 3개의 입력(c_{t-1}, h_{t-1}, x_t)으로부터 '어떤 계산'을 수행하여 구할 수 있다.



6.2.2 LSTM 계층 조립하기

- **게이트**: 데이터의 흐름을 제어

그림 6-13 비유하자면 게이트는 물의 흐름을 제어한다.

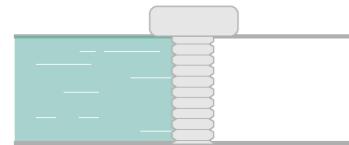
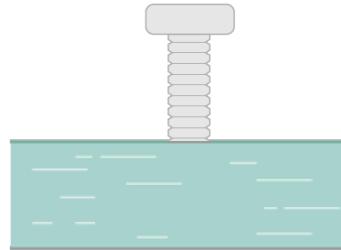
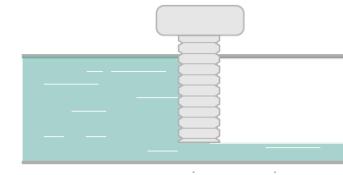
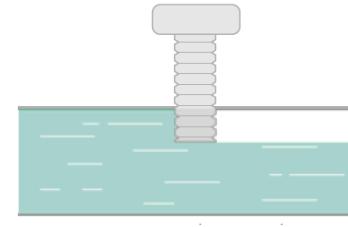


그림 6-14 물이 흐르는 양을 0.0~1.0 범위에서 제어한다.



- LSTM에서 사용하는 게이트는 단순히 열고 닫는 기능 뿐만 아니라 어느 정도 열지를 조절할 수 있다.
 - 다시 말해 다음 단계로 흘려 보낼 양을 제어한다.
 - ‘어느 정도’를 ‘열림 상태(openness)’라고 부르며 0.7, 0.2 처럼 제어할 수 있다.
 - 게이트의 열림 상태는 0.0 ~ 1.0 사이의 실수로 나타낸다.
 - 이 열림 상태 또한 데이터로부터 학습하게 되는데, 이때 시그모이드 함수를 사용해 계산하게 된다.
 - 시그모이드 함수의 출력 또한 0 ~ 1 사이의 실수이기 때문

6.2.3 output 게이트

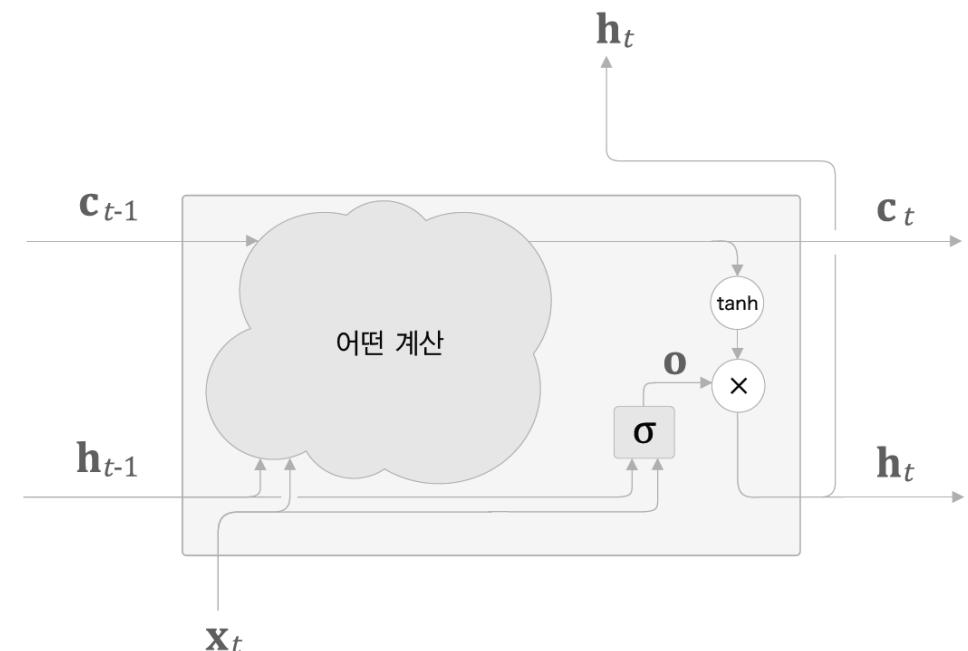
- **output 게이트:** 다음 은닉 상태 h_t 의 출력을 담당하는 게이트
 - 입력 x_t 와 이전 상태 h_{t-1} 로부터 output 게이트의 열림 상태를 구한다.

$$o = \sigma(x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)}) \quad [\text{식 6.1}]$$

- 이렇게 output 게이트에서 수행하는 [식 6.1]의 계산을 σ 로 표기
 - σ 의 출력을 o 라고 하면, h_t 는 o 와 $\tanh(c_t)$ 의 곱으로 계산
 - 여기서 말하는 ‘곱’ 이란 원소별 곱이며, 이것을 아다마르 곱 (Hadamard product, \odot)라고 한다.

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

\tanh 의 출력은 -1.0~1.0의 실수이며, 이 수치는 그 안에 인코딩된 ‘정보’의 강약(정도)를 표시한다고 해석할 수 있다.
반면 시그모이드 함수의 출력은 0.0~1.0의 실수이며, 데이터를 얼마만큼 통과시킬지를 정하는 비율을 뜻한다.
따라서 **게이트에서는 시그모이드 함수가, 실질적인 ‘정보를 지니는 데이터에는 \tanh 함수가 활성화 함수로 사용된다.**



6.2.4 forget 게이트

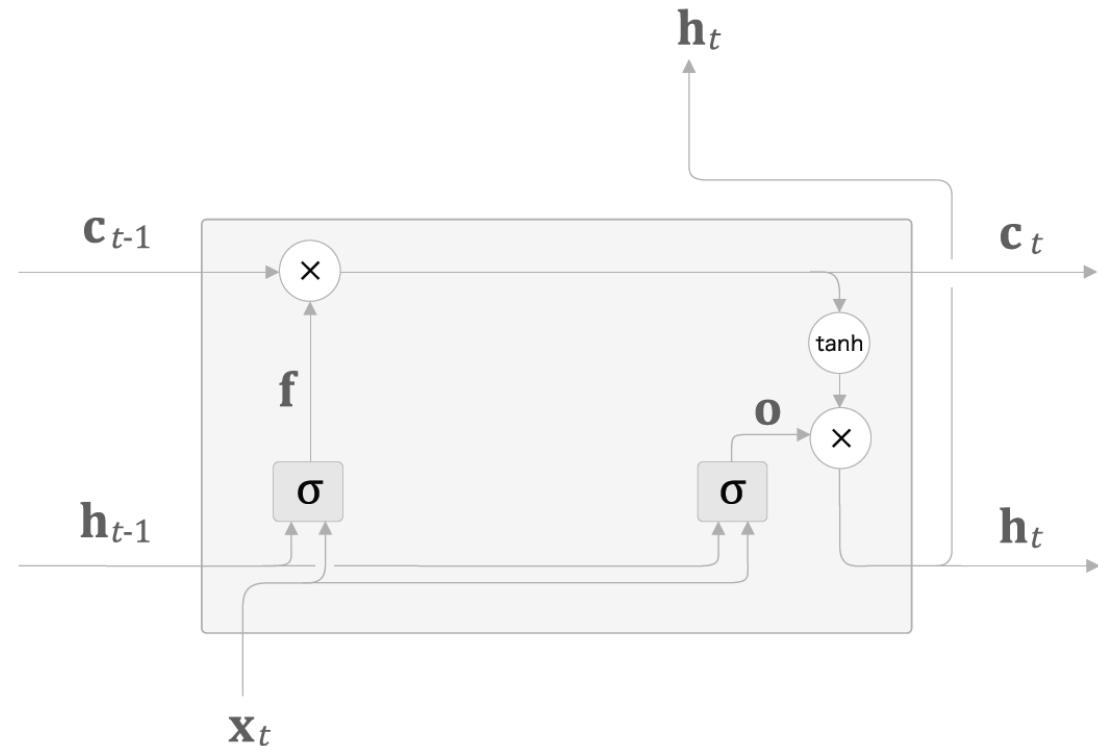
- forget 게이트: c_{t-1} 의 기억 중에서 불필요한 기억을 잊게 해주는 게이트

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

- forget 게이트의 출력 f 와 이전 기억 셀인 c_{t-1} 와의 원소별 곱, 즉 $c_t = f \odot \tanh(c_{t-1})$ 을 계산해 c_t 를 구한다.

$$c_t = f \odot \tanh(c_{t-1})$$

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

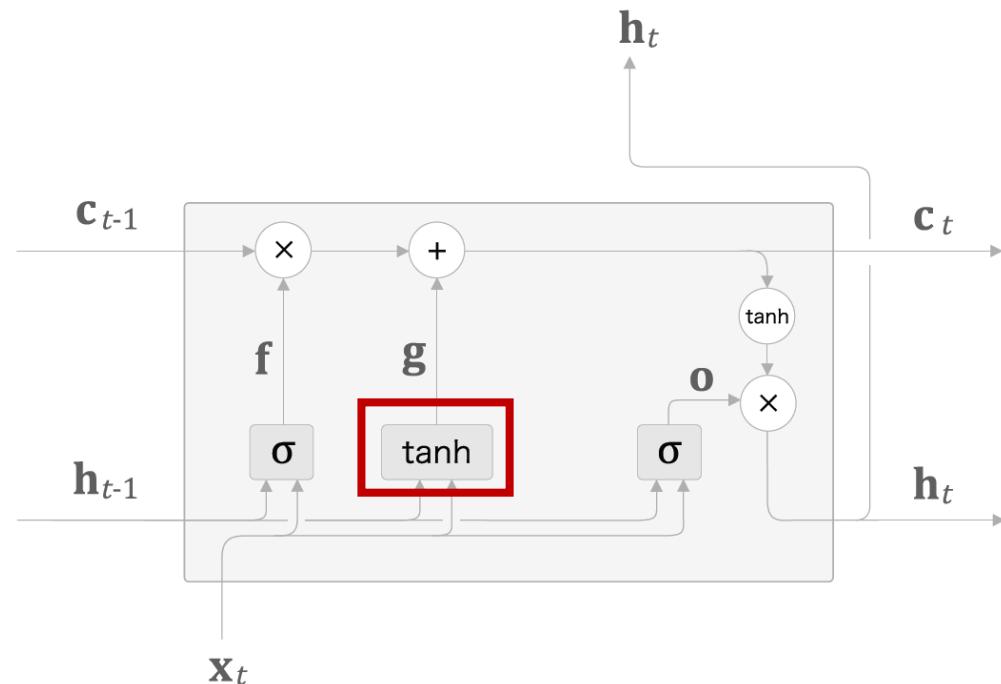


6.2.5 새로운 기억 셀

- forget 게이트를 거치면서 이전 시각의 기억 셀로부터 기억이 삭제
- 이제 새로 기억해야 할 정보를 기억 셀에 추가
 - tanh 노드가 계산한 결과가 c_{t-1} 에 더해짐
 - 즉, 기억 셀에 새로운 정보가 추가
 - 이 tanh는 게이트는 아니며 단순히 새로운 정보를 기억 셀에 추가하는 것이 목적인 노드
 - 따라서 활성화 함수로 시그모이드 함수가 아닌 tanh 함수가 사용

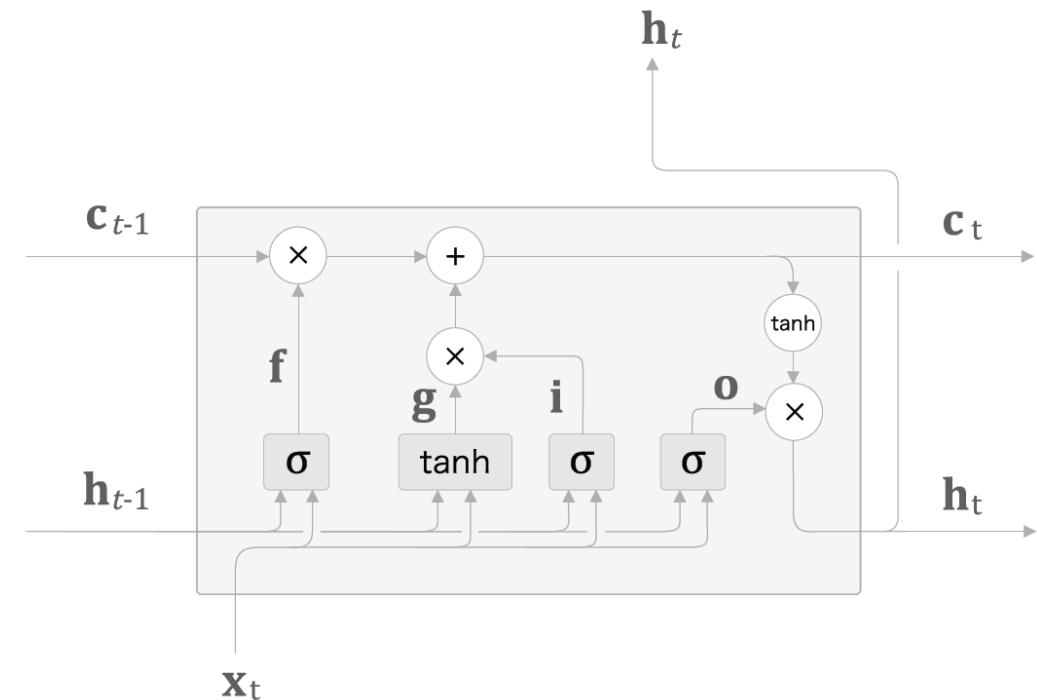
$$g = \tanh(x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)})$$

- 기억 셀에 추가하는 **새로운 기억을 g**로 표기
 - 이 g가 이전 시각의 c_{t-1} 에 더해짐으로써 새로운 기억이 생겨난다.



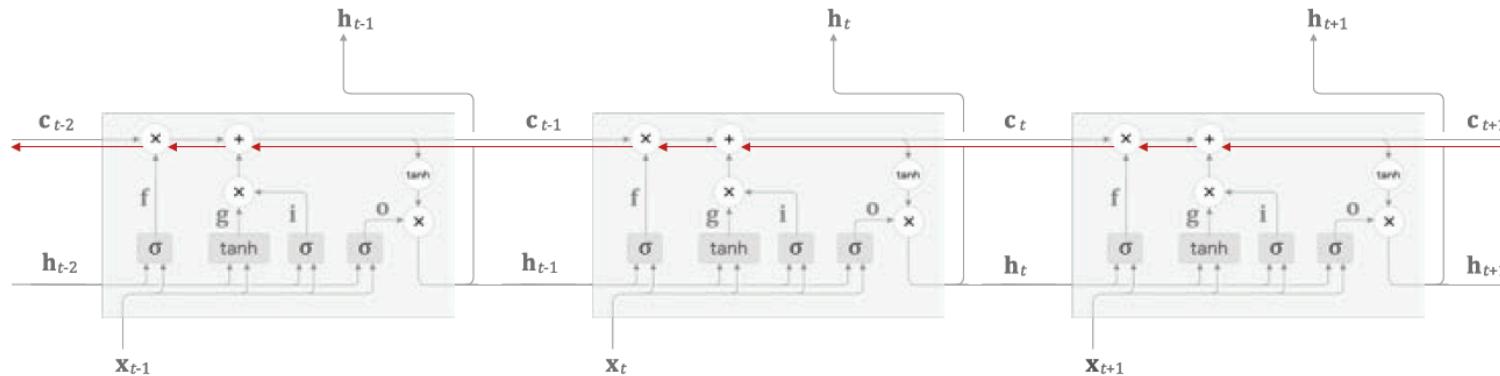
6.2.6 input 게이트

- 마지막으로 g 에 input 게이트를 새롭게 추가한다.
 - input 게이트:** g 의 각 원소가 새로 추가되는 정보로써의 가치가 얼마나 큰지를 판단
 - 새로운 정보를 무조건 수용하는 것이 아니라, 적절히 선택하는 역할
- $$i = \sigma(x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)})$$
- 이렇게 계산한 input 게이트의 출력 i 와 g 의 원소별 곱 결과를 기억 셀에 추가한다.

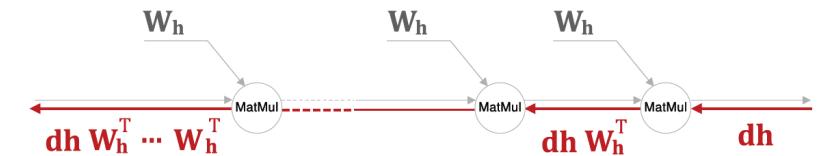


6.2.7 LSTM의 기울기 흐름

- 그렇다면 이 구조가 어떤 원리로 기울기 소실을 없애주는 걸까?



- 다음 그림은 기억 셀에만 집중해 그 역전파의 흐름을 그린 것
 - 이때 기억 셀의 역전파에서는 '+'와 'x' 노드만을 지니게 됨.
 - '+' 노드는 기울기 그대로 전달하므로 기울기 변화가 일어나지 않음
 - 'x' 노드는 행렬 곱이 아니라 원소별 곱(아마다르 곱)을 계산한다.
 - 5장의 RNN의 역전파에서는 똑같은 가중치 행렬을 사용해 행렬 곱을 반복했기 때문에 기울기의 소실(또는 폭발)이 일어남
 - 반면 LSTM의 역전파에서는 행렬 곱이 아닌 '원소별 곱'이 이뤄지고, 매 시각 다른 게이트 값을 이용해 계산
- 이렇게 매번 새로운 게이트 값을 이용해 곱셈의 효과가 누적되지 않아 **기울기 소실이 일어나기 어렵다.**
 - 'x' 노드의 계산은 forget 게이트가 제어
 - forget 게이트가 '잊어야 한다'고 판단한 기억 셀의 원소에 대해서는 그 기울기가 작아지게 됨.
 - 반면, forget 게이트가 '잊어서는 안된다'고 판단한 원소에 대해서는 그 기울기가 약화되지 않은 채로 전해짐.
- 따라서 LSTM의 기억 셀에서는 기울기 소실이 잘 일어나지 않는다!**



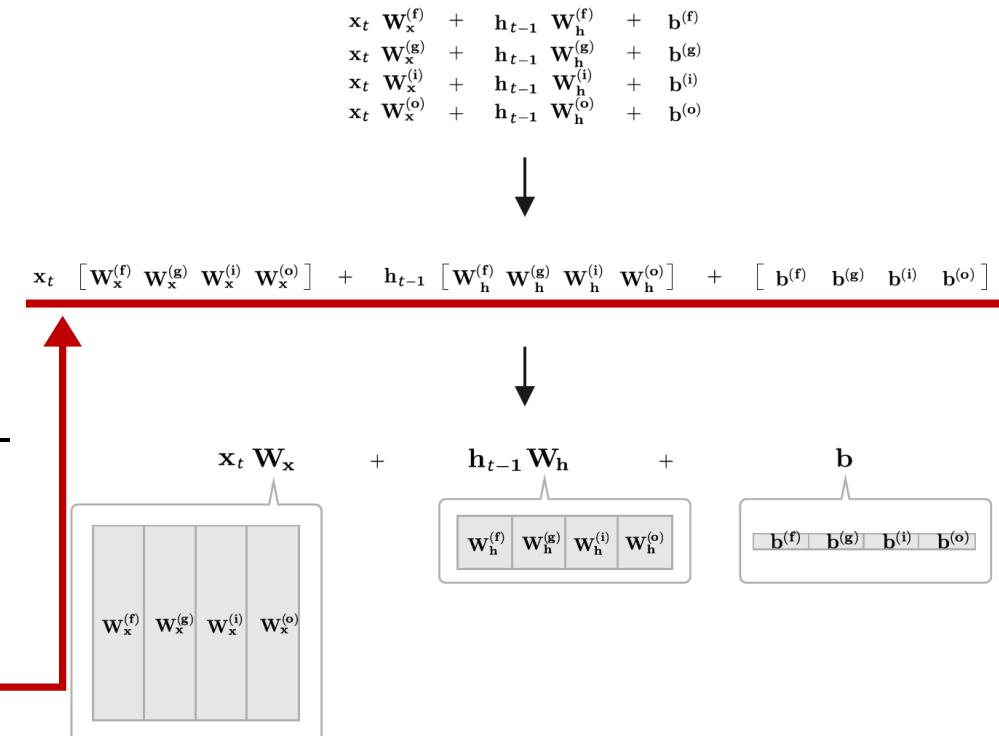
6.3 LSTM 구현

- 이제 LSTM을 구현해보자.
 - 우선 최초의 한 단계만 처리하는 LSTM 클래스를 구현
 - 이어서 T개의 단계를 한꺼번에 처리하는 Time LSTM 클래스 구현
- LSTM에서 수행하는 계산을 정리하면 다음과 같다:

$$\begin{aligned}
 f &= \sigma(x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)}) & g &= \tanh(x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)}) \\
 i &= \sigma(x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)}) & o &= \sigma(x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)}) \\
 c_t &= f \odot \tanh(c_{t-1}) & h_t &= o \odot \tanh(c_t)
 \end{aligned}$$

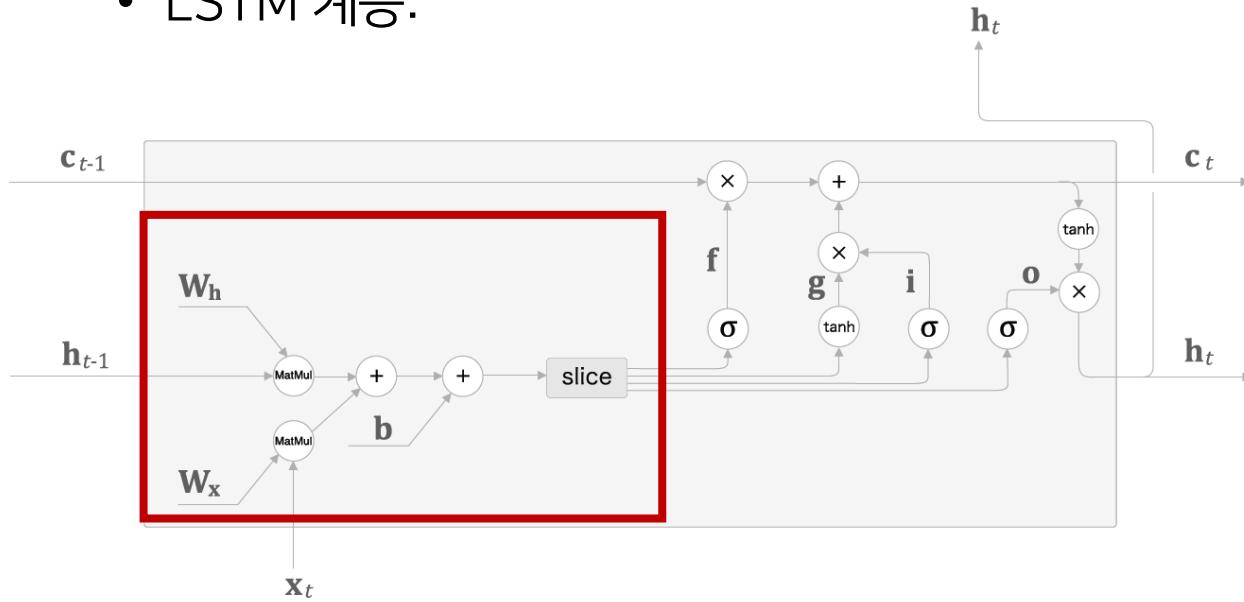
- f, g, i, o 게이트에서 개별적으로 수행되는 아핀 변환(affine transformation, $xW_x + hW_h + b$ 형태의 계산)을 하나의 식으로 정리해 계산해보자.

- 4개의 가중치를 하나로 모을 수 있음
- 4번을 수행하던 아핀 변환을 1번의 계산으로 완료
 - 계산 속도가 빨라짐
 - 가중치를 한 데로 모아 관리하기 때문에 소스코드 간결화



6.3 LSTM 구현

- LSTM 계층:



- 처음 4개분의 변환을 한꺼번에 수행한 뒤 slice 노드를 통해 4 개의 결과를 꺼낸다.

```
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
```

cache: 순전파 때 중간 결과를 보관했다가 역전파 계산에 사용

- Wx : 입력 x 에 대한 가중치 매개변수(4개분의 가중치가 담겨 있음)
- Wh : 은닉 상태 h 에 대한 가장추 매개변수(4개분의 가중치가 담겨 있음)
- b : 편향 (4개분의 편향이 담겨 있음)

6.3 LSTM 구현

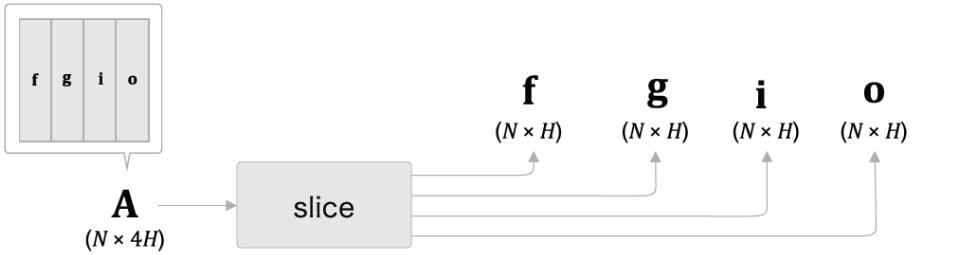
- LSTM 계층의 forward():

```
def forward(self, x, h_prev, c_prev): x: 현 시간의 입력  
    Wx, Wh, b = self.params  
    N, H = h_prev.shape  
  
    A = np.dot(x, Wx) + np.dot(h_prev, Wh) + b  
        # 가장 먼저 아핀 변환을 한다.  
        - A: 4개분의 아핀 변환 결과가 저장  
    f = A[:, :H]  
    g = A[:, H:2*H]  
    i = A[:, 2*H:3*H]  
    o = A[:, 3*H:]  
    }  
    # A를 슬라이스해 데이터를 꺼낸다.  
  
    f = sigmoid(f)  
    g = np.tanh(g)  
    i = sigmoid(i)  
    o = 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
```

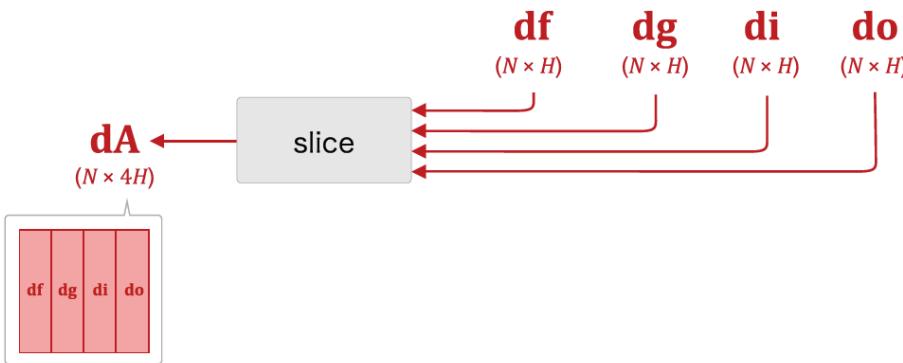
6.3 LSTM 구현

- LSTM 계층의 backward():

순전파



역전파



- 순전파에서 행렬을 네 조각으로 나눠 분배했으니 역전파에서는 반대로 4개의 기울기를 결합해야 한다.
=> `np.hstack()` 사용
 - `np.hstack()`: 인수로 주어진 배열들을 가로로 연결

```

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
    do = dh_next * tanh_c_next
    dg = ds * i

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

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

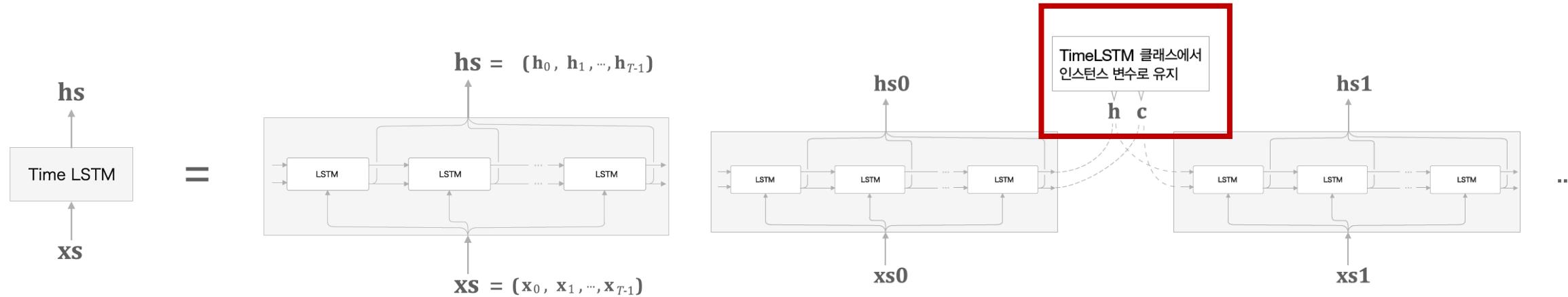
    dWh = np.dot(h_prev.T, dA)
    dWx = np.dot(x.T, dA)
    db = dA.sum(axis=0)

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

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

    return dx, dh_prev, dc_prev
  
```

6.3.1 Time LSTM 구현



- Time LSTM: T개분의 시계열 데이터를 한꺼번에 처리하는 계층
 - RNN에서는 Truncated BPTT를 사용해 학습
 - 역전파의 연결은 적당한 길이로 끊고, 순전파는 그대로 유지하기 때문에 은닉 상태와 기억 셀을 인스턴스 변수로 유지할 것!
 - 다음 번 forward()가 불렸을 때 이전 시각의 은닉 상태와 기억셀에서부터 시작할 수 있게 된다.

6.3.1 Time LSTM 구현

```
# 6.3.1 Time LSTM 구현
class TimeLSTM:
    def __init__(self, Wx, Wh, b, stateful=False):
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
        self.layers = None

        self.h, self.c = None, None
        self.dh = None
        self.stateful = stateful

    def forward(self, xs):
        Wx, Wh, b = self.params
        N, T, D = xs.shape
        H = Wh.shape[0]

        self.layers = []
        hs = np.empty((N, T, H), dtype='f')

        if not self.stateful or self.h is None:
            self.h = np.zeros((N, H), dtype='f')
        if not self.stateful or self.c is None:
            self.c = np.zeros((N, H), dtype='f')

        for t in range(T):
            layer = LSTM(*self.params)
            self.h, self.c = layer.forward(xs[:, t, :], self.h, self.c)
            hs[:, t, :] = self.h

            self.layers.append(layer)

        return hs
```

```
def backward(self, dhs):
    Wx, Wh, b = self.params
    N, T, H = dhs.shape
    D = Wx.shape[0]

    dxs = np.empty((N, T, D), dtype='f')
    dh, dc = 0, 0

    grads = [0, 0, 0]
    for t in reversed(range(T)):
        layer = self.layers[t]
        dx, dh, dc = layer.backward(dhs[:, t, :] + dh, dc)
        dxs[:, t, :] = dx
        for i, grad in enumerate(layer.grads):
            grads[i] += grad

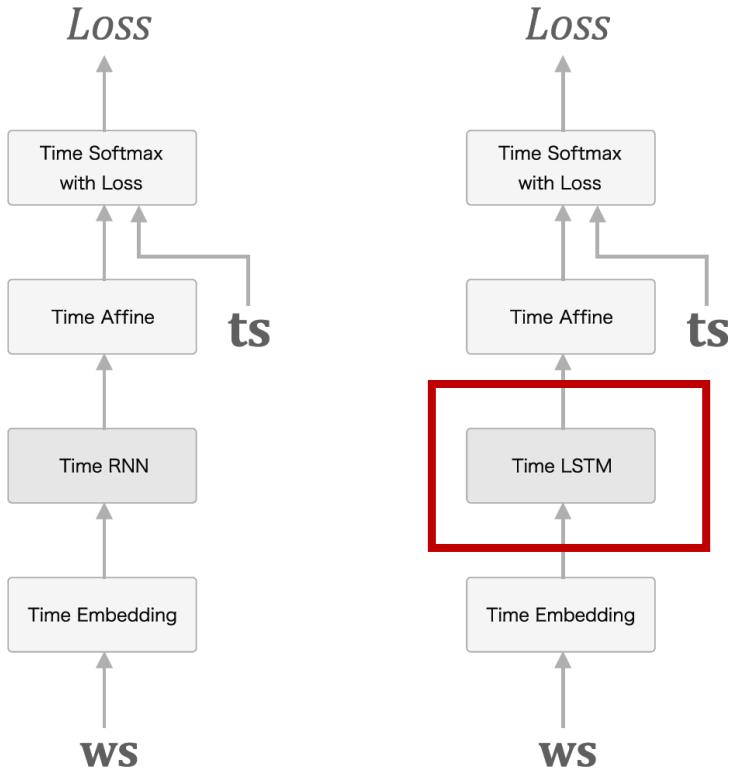
    for i, grad in enumerate(grads):
        self.grads[i][...] = grad
    self.dh = dh
    return dxs

def set_state(self, h, c=None):
    self.h, self.c = h, c

def reset_state(self):
    self.h, self.c = None, None
```

- Time RNN 계층과 같은 방법으로 구현
 - 은닉상태 h 와 함께 기억 셀 c 도 이용

6.4 LSTM을 사용한 언어 모델



```
def predict(self, xs):
    for layer in self.layers:
        xs = layer.forward(xs)
    return xs

def save_params(self, file_name='Rnnlm.pkl'):
    with open(file_name, 'wb') as f:
        pickle.dump(params, f)

def load_params(self, file_name='Rnnlm.pkl'):
    with open(file_name, 'rb') as f:
        self.params = pickle.load(f)
```

- 5장의 RNN을 사용한 언어 모델과 거의 비슷하며 LSTM을 사용한다는 차이밖에 없다.
 - predict(): Softmax 계층 직전까지를 처리하는 메서드, 7장에서 수행하는 문장 생성에 사용
 - load_params():
 - save_params()

6.4 LSTM을 사용한 언어 모델

```
# 학습 데이터 읽기
corpus, word_to_id, id_to_word = ptb.load_data('train')
corpus_test, _, _ = ptb.load_data('test')
vocab_size = len(word_to_id)
xs = corpus[:-1]
ts = corpus[1:]

# 모델 생성
model = Rnnlm(vocab_size, wordvec_size, hidden_size)
optimizer = SGD(lr)
trainer = RnnlmTrainer(model, optimizer)

# 기울기 클리핑을 적용하여 학습
1) trainer.fit(xs, ts, max_epoch, batch_size, time_size, max_grad,
               eval_interval=20)
   trainer.plot(ylim=(0, 500))

# 테스트 데이터로 평가
2) model.reset_state()
   ppl_test = eval_perplexity(model, corpus_test)
   print('테스트 퍼플렉서티:', ppl_test)

# 매개변수 저장
3) model.save_params()
```

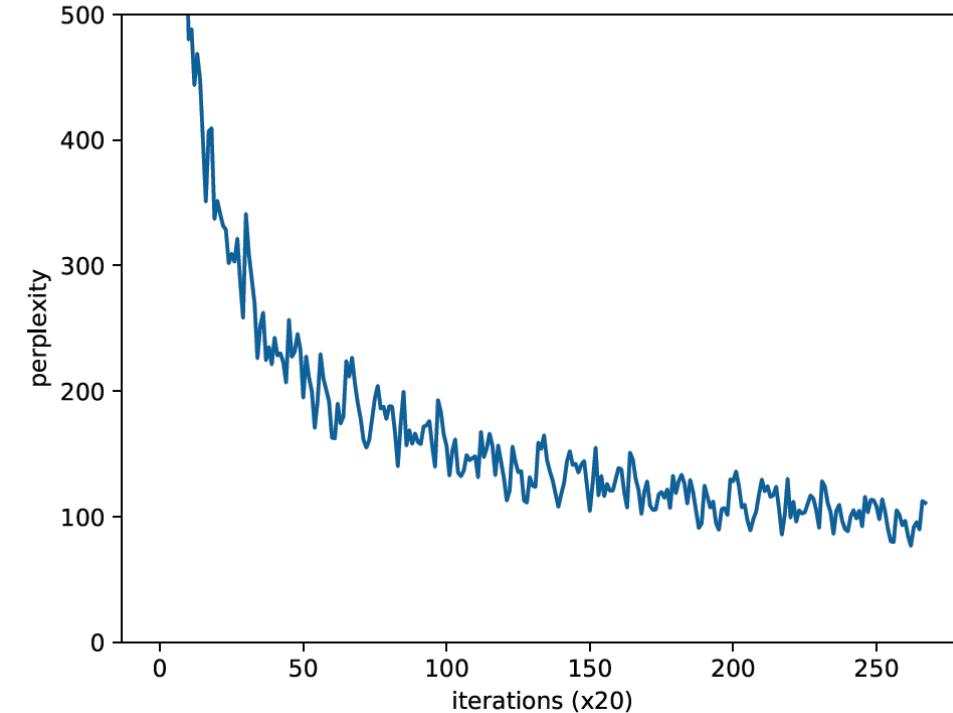
각 미니배치에서 샘플을 읽기
시작하는 위치를 계산

offsets: 데이터를 읽는 시작 위치 저장

- 1) **기울기 클리핑을 적용하여 학습**
 - RnnlmTrainer 클래스를 사용해 모델을 학습
 - fit() 메서드는 모델의 기울기를 구해 모델의 매개변수를 갱신
 - 이때 인수로 max_grad를 지정해 기울기 클리핑을 적용
 - 인수 eval_interval = 20은 20번째 반복마다 퍼플렉서티를 평가하라는 뜻
- 2) **학습이 끝난 후 테스트 데이터를 사용해 퍼플렉서티를 평가**
 - 이때 모델 상태(LSTM의 은닉 상태와 기억 셀)을 재설정하여 평가를 수행해야 한다.
- 3) **학습이 완료된 매개변수들을 파일로 저장**
 - 다음 장에서 학습된 가중치 매개변수를 사용해 문장을 생성할 때 사용

6.4 LSTM을 사용한 언어 모델

에폭 1	반복 1 / 1327	시간 0 [s]	퍼플렉시티 10000.54
에폭 1	반복 21 / 1327	시간 4 [s]	퍼플렉시티 2824.03
에폭 1	반복 41 / 1327	시간 8 [s]	퍼플렉시티 1221.40
에폭 1	반복 61 / 1327	시간 12 [s]	퍼플렉시티 1013.14
에폭 1	반복 81 / 1327	시간 16 [s]	퍼플렉시티 796.39
에폭 1	반복 101 / 1327	시간 19 [s]	퍼플렉시티 641.75
에폭 1	반복 121 / 1327	시간 23 [s]	퍼플렉시티 647.01
에폭 1	반복 141 / 1327	시간 27 [s]	퍼플렉시티 585.74
에폭 1	반복 161 / 1327	시간 31 [s]	퍼플렉시티 565.66
에폭 1	반복 181 / 1327	시간 35 [s]	퍼플렉시티 591.41
에폭 1	반복 201 / 1327	시간 39 [s]	퍼플렉시티 486.84
에폭 1	반복 221 / 1327	시간 42 [s]	퍼플렉시티 475.70
에폭 1	반복 241 / 1327	시간 46 [s]	퍼플렉시티 446.07
에폭 1	반복 261 / 1327	시간 50 [s]	퍼플렉시티 458.01
에폭 1	반복 281 / 1327	시간 54 [s]	퍼플렉시티 450.67
에폭 1	반복 301 / 1327	시간 58 [s]	퍼플렉시티 393.15
에폭 1	반복 321 / 1327	시간 62 [s]	퍼플렉시티 344.08
에폭 1	반복 341 / 1327	시간 65 [s]	퍼플렉시티 404.83
에폭 1	반복 361 / 1327	시간 70 [s]	퍼플렉시티 407.64

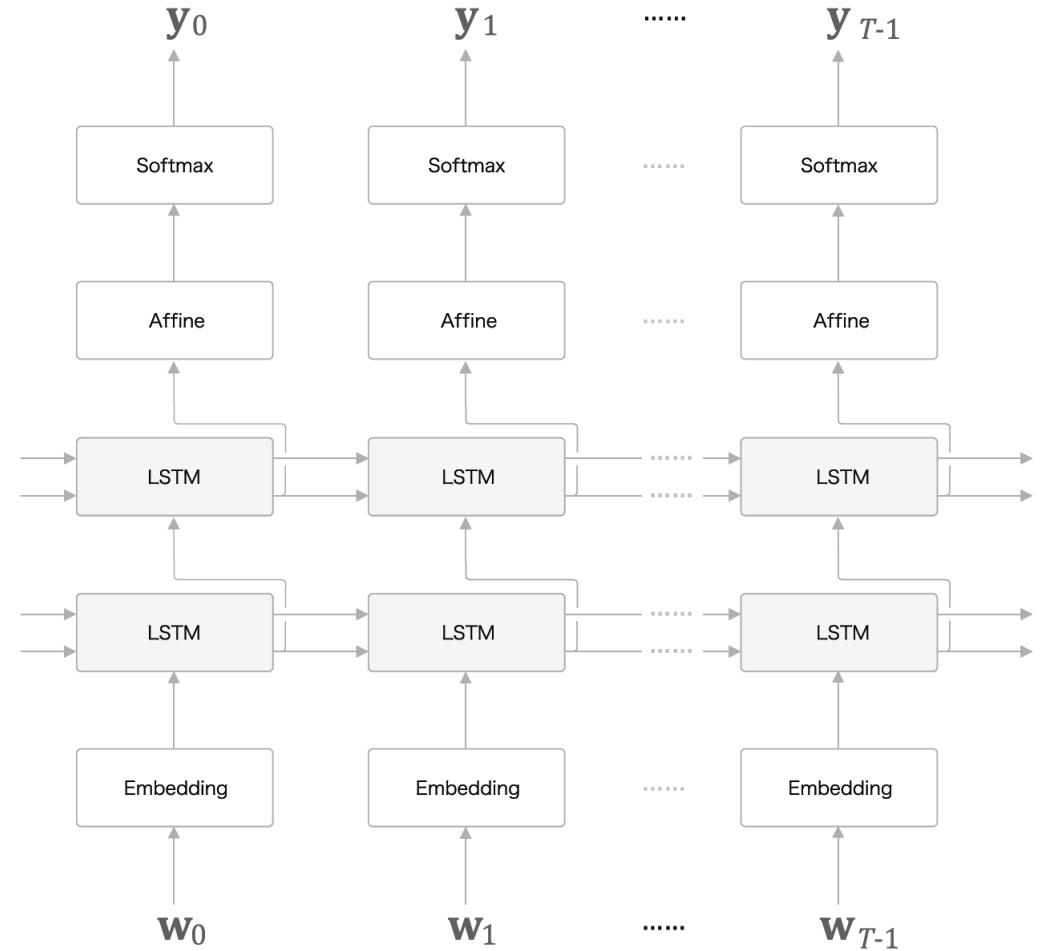


- 첫번째 퍼플렉시티 값 = 10000.84
 - 이는 다음에 나올 수 있는 후보 단어 수를 10,000개 정도로 좁혔다는 의미
- 학습을 계속 수행하면서 퍼플렉시티가 계속 좋아진다.

- 총 4 에폭(1327*4회)의 학습을 수행
- 테스트 데이터로 평가를 해보면 대략 136 전후
 - 하지만 2017년 기준 PTB 데이터셋의 퍼플렉시티가 60을 밀돌고 있음
 - 아직 개선할 부분이 많음

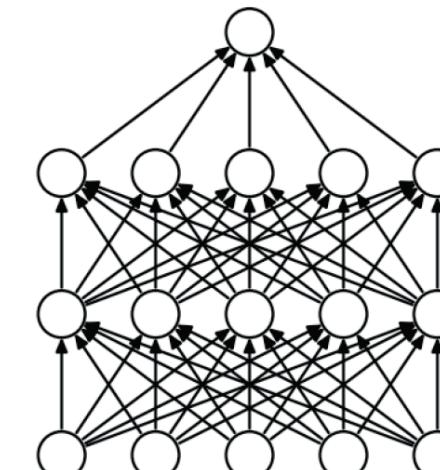
6.5 RNNLM 추가 개선: 1. LSTM 계층 다층화

- LSTM 계층을 깊게 쌓아 RNNLM 모델의 정확도를 높일 수 있다.
 - 오른쪽 그림은 LSTM을 2층으로 쌓은 모습
 - 첫 번째 LSTM의 은닉 상태가 두 번째 LSTM 계층에 입력
 - 이와 같은 방법으로 LSTM 계층을 여러겹 쌓을 수 있으며, 더 복잡한 패턴을 학습할 수 있게 된다.
- 그렇다면 몇 층을 쌓아야 할까? => **하이퍼파라미터**
 - 처리할 문제의 복잡도나 학습 데이터의 양에 따라 적절하게 결정
 - PTB 데이터셋에서는 LSTM 층 수가 2~4일 때 좋은 결과
 - 구글 번역의 GNMT 모델은 LSTM을 8층 쌓아 만든 신경망

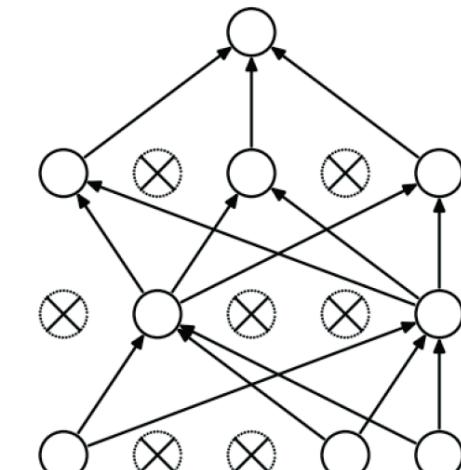


6.5.2 드롭아웃에 의한 과적합 억제

- 이처럼 LSTM을 다층화해 시계열 데이터의 복잡한 의존 관계를 학습할 수 있다.
 - 하지만 이런 모델은 종종 과적합(overfitting)을 일으킨다.
 - 특히 RNN은 일반적인 피드포워드 신경망보다 쉽게 과적합을 일으킨다.
- 오버피팅을 억제하는 전통적인 방법으로는 '훈련 데이터 양 늘리기', '모델의 복잡도 줄이기' 등이 있으며 '정규화'도 오버피팅을 줄이는데 효과적
 - 정규화(normalization): 모델의 복잡도에 패널티를 주는 방법
 - L2 정규화는 가중치가 너무 커지면 패널티를 부과
 - 드롭아웃 기법
- **드롭아웃(dropout)[9]:** 훈련 시 계층 내의 뉴런 몇 개 (eg. 50%)를 랜덤으로 무시하고 학습하는 방법



(a) 일반적인 신경망

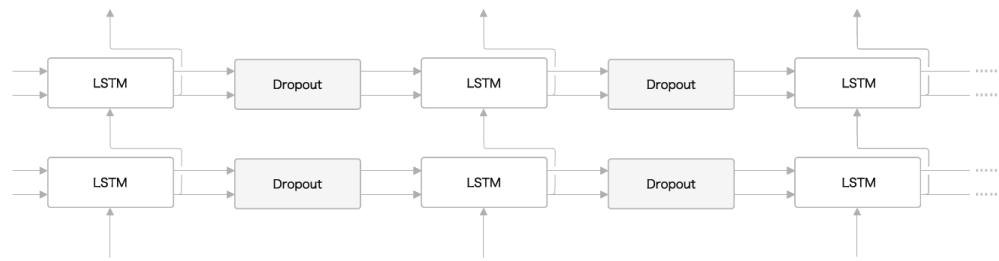


(b) 드롭아웃을 적용한 모습

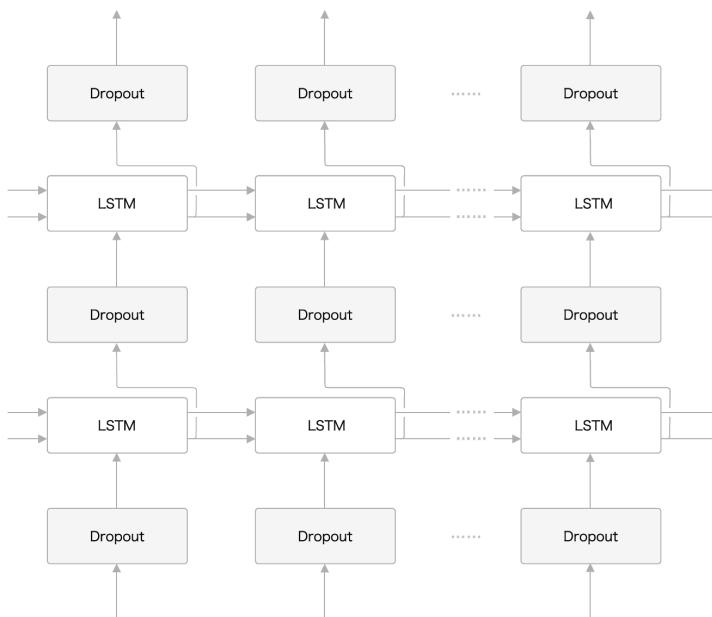
6.6.2 드롭아웃에 의한 과적합 억제

- 단순 피드포워드 신경망에서는 Dropout 계층을 활성화 함수 뒤에 삽입
 - 그렇다면 RNN을 사용한 모델에서는 어디에 삽입해야 할까?

1) LSTM 계층의 시계열 방향으로 삽입



2) 깊이 방향(상하 방향)으로 삽입



- 좋은 방법이 아님
- 학습 시 시간의 흐름에 따라 정보가 사라질 수 있음
- 즉, 흐르는 시간에 비례해 드롭아웃에 의한 노이즈가 축적

- 시간 방향(좌우 방향)으로 아무리 진행해도 정보를 잃지 않는다.
- 드롭아웃이 시간축과는 독립적으로 깊이 방향(상하 방향)에만 영향을 줌

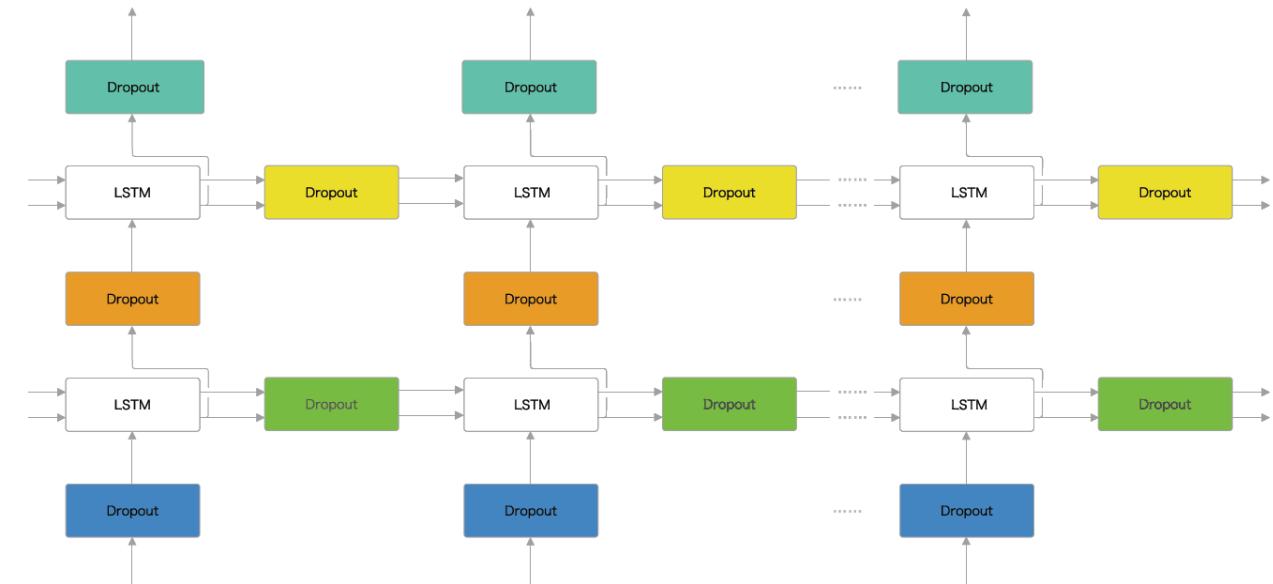
그림 6-31 피드포워드 신경망에 드롭아웃 계층을 적용하는 예



[36] Gal, Yarin, and Zoubin Ghahramani: " A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. " Advances in neural information processing systems. 2016.

6.6.2 드롭아웃에 의한 과적합 억제

- 이처럼 '일반적인 드롭아웃'은 시간 방향에는 적합하지 않다.
 - 하지만 최근 RNN의 시간 방향 정규화를 목표로 하는 방법이 다양하게 제안되고 있음.
 - [36]에서는 **변형 드롭아웃(Variational Dropout)**을 제안
 - 같은 계층에 속한 드롭아웃들은 같은 마스크를 공유
 - **마스크**: 데이터의 통과/차단을 결정하는 이진 형태의 무작위 패턴
 - 같은 계층의 드롭아웃끼리 마스크를 공유함으로써 마스크가 고정됨
 - 그 결과 정보를 잃게 되는 방법도 고정되므로, 일반적인 드롭아웃 때와 달리 정보가 지수적으로 손실되는 사태를 피할 수 있음
 - 깊이 방향은 물론 시간 방향에도 이용할 수 있어 언어 모델의 정확도를 향상

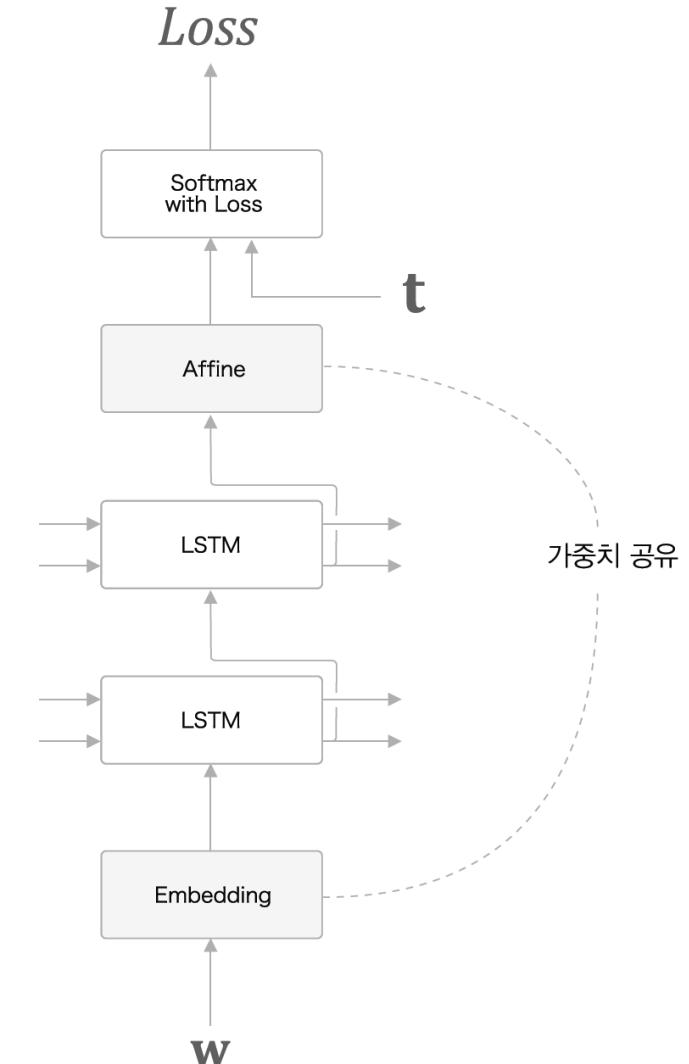


[38] Inan, Hakan, Khashayar Khosravi, and Richard Socher: "Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling." arXiv preprint arXiv:1611.01462(2016).

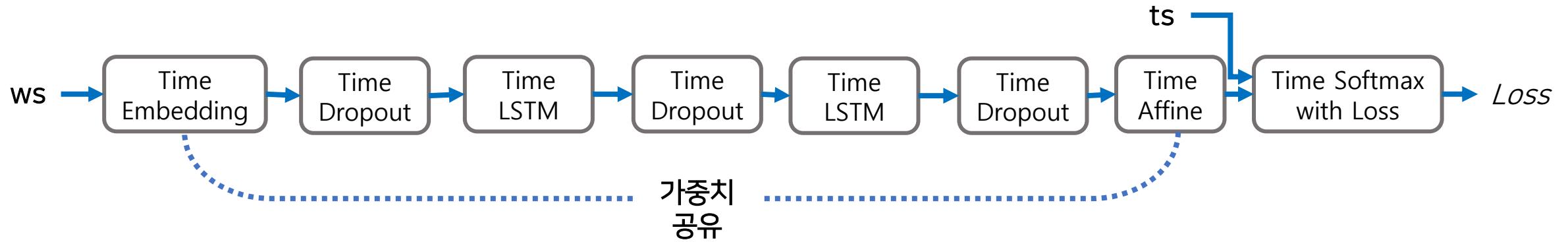
6.5.3 가중치 공유

- **가중치 공유(weight tying)**

- Embedding 계층의 가중치와 Affine 계층의 가중치를 연결(공유)하는 기법
- 학습해야 할 매개변수의 수가 크게 줄어들며, 동시에 정확도도 향상[38]
 - 오버피팅 억제
 - 학습하기가 더 쉬워지기 때문
- LSTM의 은닉 상태의 차원 수를 H 일 때
 - Embedding 계층의 가중치는 형상이 $V \times H$
 - Affine 계층의 가중치는 형상이 $H \times V$
 - 따라서 Embedding 계층의 가중치를 전치한 것을 Affine 계층의 가중치로 설정



6.5.4 개선된 RNNLM 구현



- LSTM 계층의 다층화(여기에서는 2층)
- 드롭아웃 사용(깊이 방향으로만 적용)
- 가중치 공유(Embedding과 Affine 계층에서 가중치 공유)

6.5.4 개선된 RNNLM 구현

- BetterRnnlm 계층:

```
class BetterRnnlm(BaseModel):  
    def __init__(self, vocab_size=10000, wordvec_size=650,  
                 hidden_size=650, dropout_ratio=0.5):  
        V, D, H = vocab_size, wordvec_size, hidden_size  
        rn = np.random.randn  
  
        embed_W = (rn(V, D) / 100).astype('f')  
        lstm_Wx1 = (rn(D, 4 * H) / np.sqrt(D)).astype('f')  
        lstm_Wh1 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')  
        lstm_b1 = np.zeros(4 * H).astype('f')  
        lstm_Wx2 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')  
        lstm_Wh2 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')  
        lstm_b2 = np.zeros(4 * H).astype('f')  
        affine_b = np.zeros(V).astype('f')  
  
        self.layers = [  
            TimeEmbedding(embed_W), Dropout  
            TimeDropout(dropout_ratio), TimeLSTM(lstm_Wx1, lstm_Wh1, lstm_b1, stateful=True),  
            TimeDropout(dropout_ratio), TimeLSTM(lstm_Wx2, lstm_Wh2, lstm_b2, stateful=True),  
            TimeDropout(dropout_ratio), TimeAffine(embed_W.T, affine_b) # weight tying!!  
        ]  
        self.loss_layer = TimeSoftmaxWithLoss()  
        self.lstm_layers = [self.layers[2], self.layers[4]]  
        self.drop_layers = [self.layers[1], self.layers[3], self.layers[5]]  
  
        self.params, self.grads = [], []  
        for layer in self.layers:  
            self.params += layer.params  
            self.grads += layer.grads
```

가중치 공유

LSTM 다층화

```
def predict(self, xs, train_flg=False):  
    for layer in self.drop_layers:  
        layer.train_flg = train_flg  
  
    for layer in self.layers:  
        xs = layer.forward(xs)  
    return xs  
  
def forward(self, xs, ts, train_flg=True):  
    score = self.predict(xs, train_flg)  
    loss = self.loss_layer.forward(score, ts)  
    return loss  
  
def backward(self, dout=1):  
    dout = self.loss_layer.backward(dout)  
    for layer in reversed(self.layers):  
        dout = layer.backward(dout)  
    return dout  
  
def reset_state(self):  
    for layer in self.lstm_layers:  
        layer.reset_state()
```

6.5.4 개선된 RNNLM 구현

...

- BetterRnnlm 계층:

- 매 에폭에서 검증(validation) 데이터로 퍼플렉시티를 평가
 - 그 값이 나빠졌을 경우에만 학습률을 낮춤
- 테스트 데이터로 얻은 최종 퍼플렉시티는 **75.76**
 - 개선 전 모델에서의 퍼플렉시티가 약 136이였음을 감안한다면 상당히 개선 됨!

```
model = BetterRnnlm(vocab_size, wordvec_size, hidden_size, dropout)
optimizer = SGD(lr)
trainer = RnnlmTrainer(model, optimizer)

best_ppl = float('inf')
for epoch in range(max_epoch):
    trainer.fit(xs, ts, max_epoch=1, batch_size=batch_size,
                time_size=time_size, max_grad=max_grad)

    model.reset_state()
    ppl = eval_perplexity(model, corpus_val)
    print('검증 퍼플렉시티: ', ppl)

    if best_ppl > ppl:
        best_ppl = ppl
        model.save_params()
    else:
        lr /= 4.0
        optimizer.lr = lr # 퍼플렉시티가 나빠졌다면 learning rate를 설정

    model.reset_state()
    print('-' * 50)

# 테스트 데이터로 평가
model.reset_state()
ppl_test = eval_perplexity(model, corpus_test)
print('테스트 퍼플렉시티: ', ppl_test)
```

6.5.5 첨단 연구로

그림 6-37 PTB 데이터셋에 대한 각 모델의 퍼플렉서티 결과(문현 [34]에서 발췌). 표의 ‘Parameters’는 파라미터의 총 개수, ‘Validation’은 검증 데이터에 대한 퍼플렉서티, ‘Test’는 테스트 데이터에 대한 퍼플렉서티다.

Model	Parameters	Validation	Test
Mikolov & Zweig (2012) - KN-5	2M [‡]	—	141.2
Mikolov & Zweig (2012) - KN5 + cache	2M [‡]	—	125.7
Mikolov & Zweig (2012) - RNN	6M [‡]	—	124.7
Mikolov & Zweig (2012) - RNN-LDA	7M [‡]	—	113.7
Mikolov & Zweig (2012) - RNN-LDA + KN-5 + cache	9M [‡]	—	92.0
Zaremba et al. (2014) - LSTM (medium)	20M	86.2	82.7
Zaremba et al. (2014) - LSTM (large)	66M	82.2	78.4
Gal & Ghahramani (2016) - Variational LSTM (medium)	20M	81.9 ± 0.2	79.7 ± 0.1
Gal & Ghahramani (2016) - Variational LSTM (medium, MC)	20M	—	78.6 ± 0.1
Gal & Ghahramani (2016) - Variational LSTM (large)	66M	77.9 ± 0.3	75.2 ± 0.2
Gal & Ghahramani (2016) - Variational LSTM (large, MC)	66M	—	73.4 ± 0.0
Kim et al. (2016) - CharCNN	19M	—	78.9
Merity et al. (2016) - Pointer Sentinel-LSTM	21M	72.4	70.9
Grave et al. (2016) - LSTM	—	—	82.3
Grave et al. (2016) - LSTM + continuous cache pointer	—	—	72.1
Inan et al. (2016) - Variational LSTM (tied) + augmented loss	24M	75.7	73.2
Inan et al. (2016) - Variational LSTM (tied) + augmented loss	51M	71.1	68.5
Zilly et al. (2016) - Variational RHN (tied)	23M	67.9	65.4
Zoph & Le (2016) - NAS Cell (tied)	25M	—	64.0
Zoph & Le (2016) - NAS Cell (tied)	54M	—	62.4
Melis et al. (2017) - 4-layer skip connection LSTM (tied)	24M	60.9	58.3
AWD-LSTM - 3-layer LSTM (tied)	24M	60.0	57.3
AWD-LSTM - 3-layer LSTM (tied) + continuous cache pointer	24M	53.9	52.8

다층 LSTM, 드롭아웃 기반의 정규화, 가중치 공유 사용

6.6 정리

- 단순한 RNN에서의 기울기 소실(폭발) 문제를 해결할 수 있는 게이트가 추가된 RNN(LSTM, GRU)에 대해 살펴보았다.
 - 게이트: 데이터와 기울기 흐름을 적절히 제어하는 메커니즘
- LSTM 계층을 사용한 언어 모델을 구현해 보았다.
 - PTB 데이터셋을 학습해 퍼플렉서티를 구함
 - LSTM 다층화, 드롭아웃, 가중치 공유 등의 기법을 적용해 언어 모델을 개선해 정확도를 향상
- 7장에서는 이렇게 구현한 언어 모델을 사용해 문장을 생성해보고 또 기계 번역처럼 한 언어를 다른 언어로 변환하는 모델을 자세히 살펴보도록 한다.