

# Ch.5

---

# RNN

# RNN

- 지금까지 살펴본 신경망은 feed forward, 단방향 신경망이다.
- 구성이 단순해 많은 곳에 응용 가능
- 시계열 데이터의 패턴을 충분히 학습하지 못한다.

→ 순환 신경망 (RNN)이 등장!

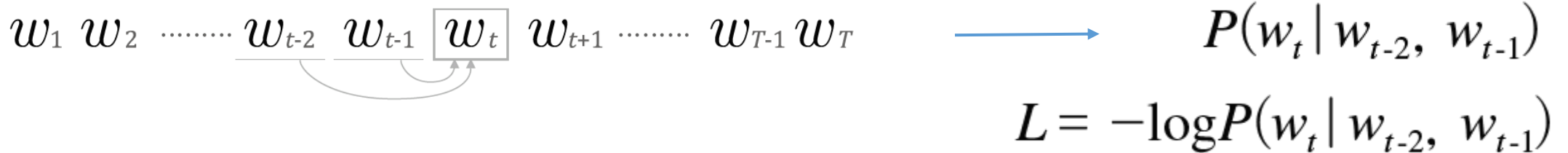
# 확률과 언어 모델

- CBOW모델 학습의 목적은

맥락으로부터 타깃을 정확히 추측하는 것.

-> 단어의 의미가 인코딩 된 단어의 분산 표현 획득

그림 5-2 왼쪽 윈도우만 맥락으로 고려한다.

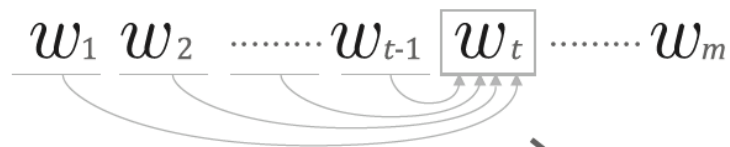

$$w_1 \ w_2 \ \cdots \ w_{t-2} \ w_{t-1} \ \boxed{w_t} \ w_{t+1} \ \cdots \ w_{T-1} \ w_T \quad \longrightarrow \quad P(w_t | w_{t-2}, w_{t-1})$$
$$L = -\log P(w_t | w_{t-2}, w_{t-1})$$

맥락을 왼쪽 윈도우 만으로 한정해 보자.

# 언어 모델

- 언어 모델은 단어 나열에 확률을 부여한다.
- 번역, 음성인식, 문장 생성 용도로 활용 가능.
- 문장의 '자연스러움'을 평가.
- m개의 단어로 된 문장이  $w_1 \sim w_m$ 의 순서로 출현할 확률

$$P(w_1, \dots, w_m) = \prod_{t=1}^m P(w_t | w_1, \dots, w_{t-1})$$



$$= P(w_t | w_1, w_2, \dots, w_{t-1})$$

# CBOW 모델을 언어 모델로

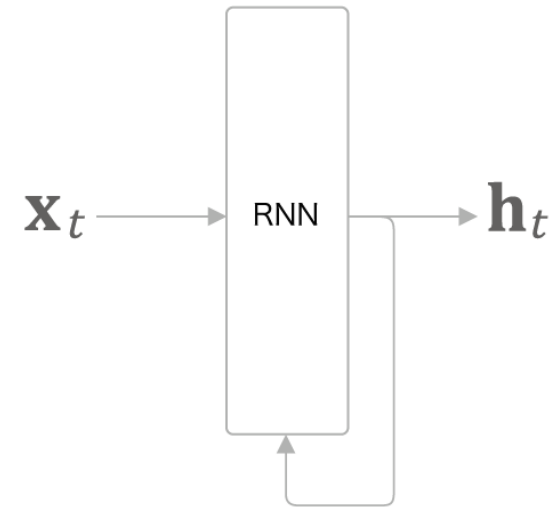
- 맥락의 크기를 특정 값으로 한정해 근사적으로 나타낼 수 있다.
- 확률이 직전 N개의 사건에 의존할 때  
'N층 마르코프 연쇄'라 한다.
- 고정된 맥락의 크기는 그 보다 왼쪽에 있는 정보를 무시한다.
- 또한 CBOW 모델에서는 맥락 내의 단어 순서가 무시된다.  
(벡터의 '합'이 은닉층에 들어가기 때문)

→ 맥락의 정보를 기억하는 RNN

# RNN

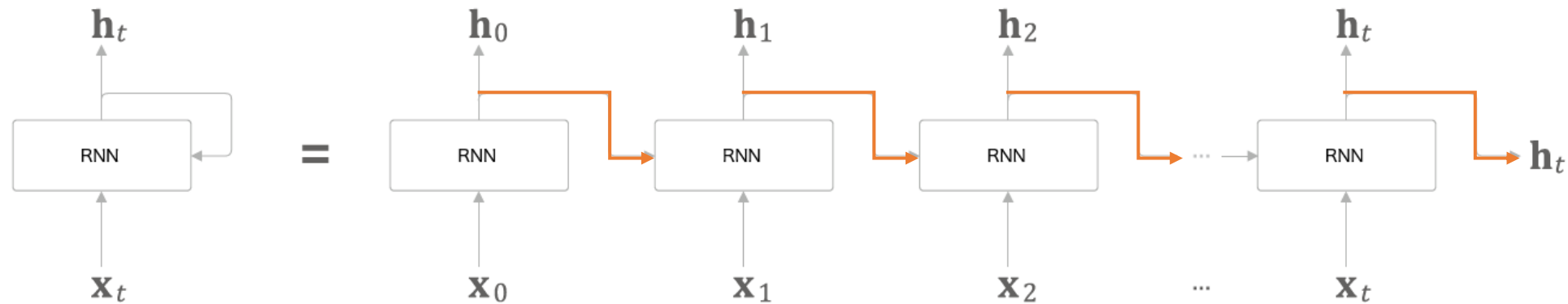
- 순환하는 신경망 -> 닫힌(순환하는) 경로
- $x_t, h_t$ 의 t는 시각을 뜻함.
- $x_t$ 가 단어 벡터라고 한다면  
분산 표현이 순서대로 RNN계층에 입력됨.

그림 5-6 순환 경로를 포함하는 RNN 계층



- RNN의 순환 구조를 펼치면 다음과 같다.
- 다수의 RNN계층이 전부 '같은' 계층.
- 해당 계층으로의 입력과 1개 전의 RNN계층의 출력을 받아  
현 시각의 출력을 계산한다.

그림 5-8 RNN 계층의 순환 구조 펼치기

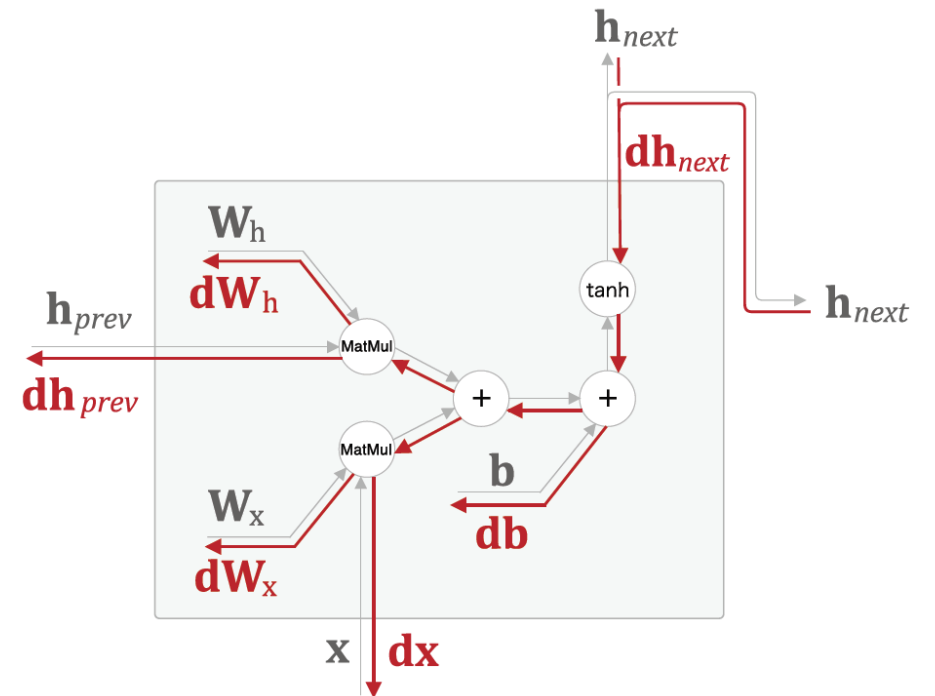


- RNN에는 2개의 가중치가 존재
- $x \rightarrow h$ 로 변환하기 위한  $W_x$ ,
- RNN 출력을 다음 시각 출력으로 변환하기 위한  $W_h$
- $x_t, h_t$ 는 행벡터

- $\mathbf{h}_t = \tanh(\mathbf{h}_{t-1} \mathbf{W}_h + \mathbf{x}_t \mathbf{W}_x + \mathbf{b})$
- $h$  라는 '은닉 상태'가 갱신된다

$$\begin{array}{ccccccc}
 \mathbf{h}_{t-1} & \mathbf{W}_h & + & \mathbf{x}_t & \mathbf{W}_x & = & \mathbf{h}_t \\
 N \times H & H \times H & & N \times D & D \times H & & N \times H \\
 \text{일치} & & & \text{일치} & & & \\
 \hline
 & & & & & & \uparrow \uparrow \uparrow
 \end{array}$$

그림 5-20 RNN 계층의 계산 그래프(역전파 포함)

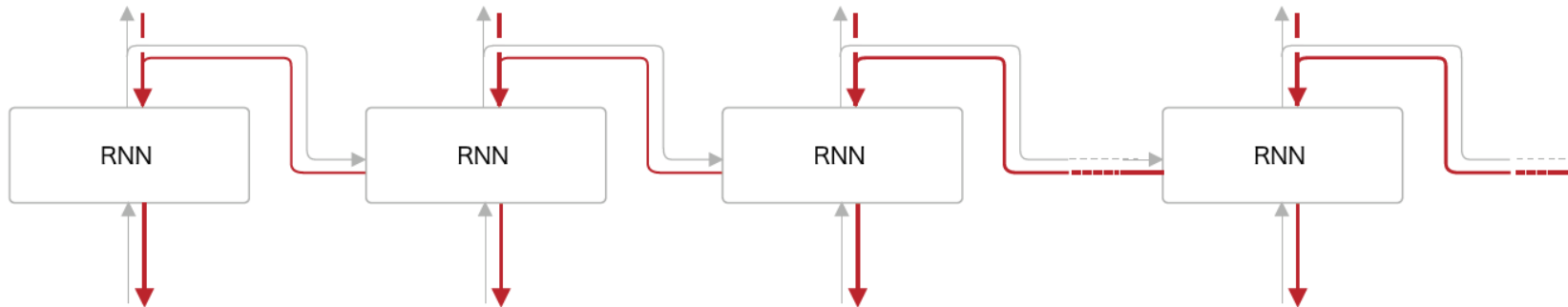




# BPTT (BackProp Through Time)

- 순환 구조를 펼치면 일반적인 역전파 적용 가능
- 시간 방향으로 펼친 신경망의 오차 역전파 'BPTT'
- 긴 시계열 데이터 학습 시,  
소모하는 컴퓨팅 자원과 기울기 소실 문제

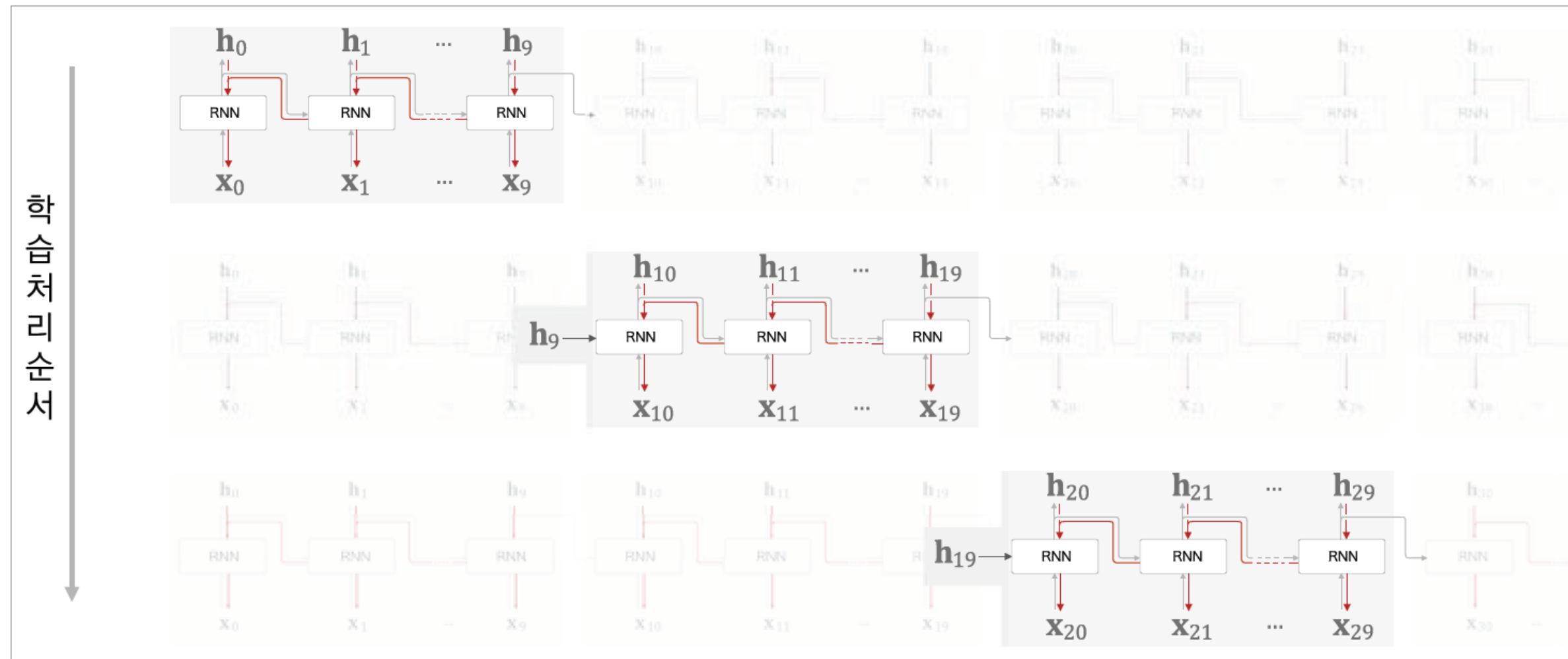
그림 5-10 순환 구조를 펼친 RNN 계층에서의 오차역전파법



# Truncated(잘린) BPTT

- 너무 길다면 끊으면 된다.
- 적당한 길이로 자른 작은 신경망 단위로 역전파 수행
- 순전파의 연결은 유지되어야 하므로 데이터를 '순서대로' 입력해야 한다.

그림 5-14 Truncated BPTT의 데이터 처리 순서



# RNN의 미니배치 학습

그림 5-15 미니배치 학습 시 데이터를 제공하는 시작 위치를 각 미니배치(각 샘플)로 옮긴다.

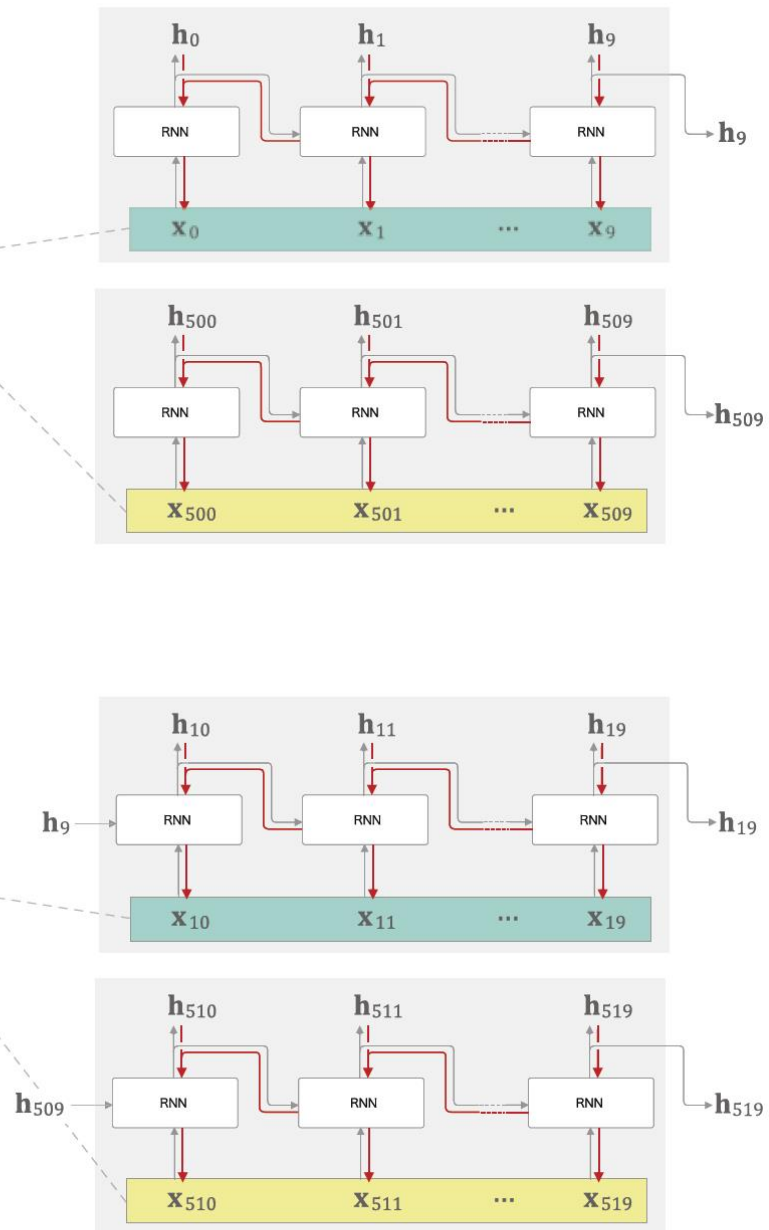
학습  
처리  
순서

첫 번째 미니배치의 원소

$(x_0, x_1, \dots, x_9, x_{500}, x_{501}, \dots, x_{509})$

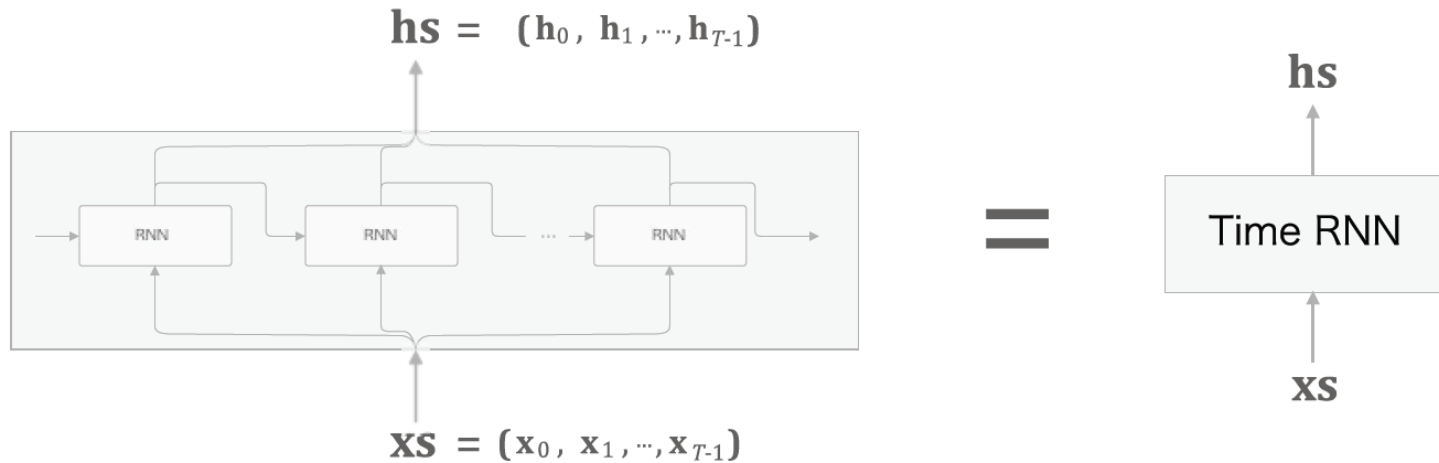
두 번째 미니배치의 원소

$(x_{10}, x_{11}, \dots, x_{19}, x_{510}, x_{511}, \dots, x_{519})$



# 구현

그림 5-17 Time RNN 계층: 순환 구조를 펼친 후의 계층들을 하나의 계층으로 간주한다.



- 은닉 상태를 변수  $h$ 로 보관해 다음 블록에 인계

```

1 class RNN:
2     def __init__(self, Wx, Wh, b):
3         self.params = [Wx, Wh, b]
4         self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
5         self.cache = None
6
7     def forward(self, x, h_prev):
8         Wx, Wh, b = self.params
9         t = np.matmul(h_prev, Wh) + np.matmul(x, Wx) + b
10        h_next = np.tanh(t)
11
12        self.cache = (x, h_prev, h_next)
13
14        return h_next
15
16    def backward(self, dh_next):
17        Wx, Wh, b = self.params
18        x, h_prev, h_next = self.cache
19
20        dt = dh_next * (1 - h_next**2)
21        db = np.sum(dt, axis = 0)
22        dWh = np.matmul(h_prev.T, dt)
23        dh_prev = np.matmul(dt, Wh.T)
24        dWx = np.matmul(x.T, dt)
25        dx = np.matmul(dt, Wx.T)
26
27        self.grads[0][...] = dWx # [...] --> DeepCopy
28        self.grads[1][...] = dWh
29        self.grads[2][...] = db
30
31        return dx, dh_prev

```

$$\begin{array}{ccccccc}
 \mathbf{h}_{t-1} & \mathbf{W}_h & + & \mathbf{x}_t & \mathbf{W}_x & = & \mathbf{h}_t \\
 N \times H & H \times H & & N \times D & D \times H & & N \times H \\
 \underbrace{\hspace{1.5cm}}_{\text{일치}} & & & \underbrace{\hspace{1.5cm}}_{\text{일치}} & & & 
 \end{array}$$

# RNN 계층 T개를 연결한 Time RNN

## Time RNN 계층 구현

```
1 class TimeRNN:
2     #stateful True = 은닉 상태 유지 (순전파를 끊지 않고 계속 전파함), False = 은닉상태를 0행렬로 초기화
3     def __init__(self, Wx, Wh, b, stateful=False):
4         self.params = [Wx, Wh, b]
5         self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
6         self.layers = None # RNN 계층들 저장용
7
8         self.h, self.dh = None, None
9         # forward 호출 시 return되는 h 값 저장
10        # dh = backward 호출 시 앞 블록의 은닉 상태의 기울기 저장
11        self.stateful = stateful
12
13    def set_state(self, h):
14        self.h = h
15
16    def reset_state(self):
17        self.h = None
18
19    def forward(self, xs):
20        Wx, Wh, b = self.params
21        N, T, D = xs.shape # 미니배치 크기, 시계열데이터 분량, 입력벡터 차원 수
22        D, H = Wx.shape
23
24        self.layers = []
25        hs = np.empty((N, T, H), dtype='f')
26
27        if not self.stateful or self.h is None:
28            self.h = np.zeros((N, H), dtype='f')
29
30        for t in range(T):
31            layer = RNN(*self.params) # * 는 리스트의 원소들을 추출하여 메소드의 인수로 전달함
32                                     # self.params의 Wx, Wh, b를 RNN의 __init__에 전달
33            self.h = layer.forward(xs[:, t, :], self.h) # 마지막 RNN계층의 은닉 상태가 저장됨
34            hs[:, t, :] = self.h
35            self.layers.append(layer)
36
37        return hs
```

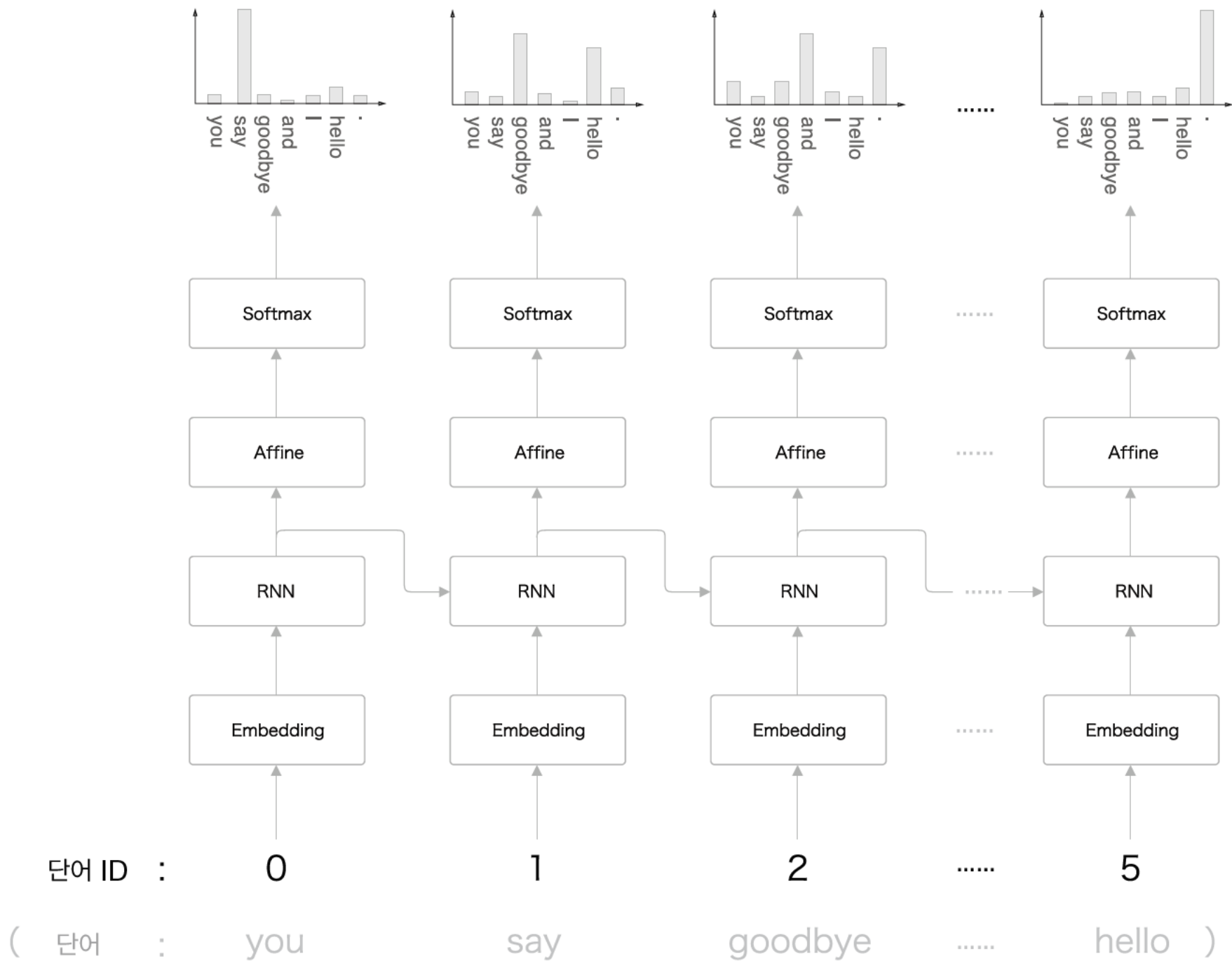
```
38
39    def backward(self, dhs):
40        Wx, Wh, b = self.params
41        N, T, H = dhs.shape
42        D, H = Wx.shape
43
44        dxs = np.empty((N, T, D), dtype='f')
45        dh = 0
46        grads = [0, 0, 0]
47        for t in reversed(range(T)):
48            layer = self.layers[t]
49            dx, dh = layer.backward(dhs[:, t, :] + dh)
50            dxs[:, t, :] = dx
51
52            for i, grad in enumerate(layer.grads):
53                self.grads[i][...] = grad
54            self.dh = dh
55
56        return dxs
```

# RNNLM

- RNN을 사용한 언어 모델
- RNN은 맥락을 기억한다
- 과거의 정보를 응집된 은닉 상태 벡터로 저장해 둔다
- 입력된 단어를 기억하고 다음에 출현할 단어를 예측한다

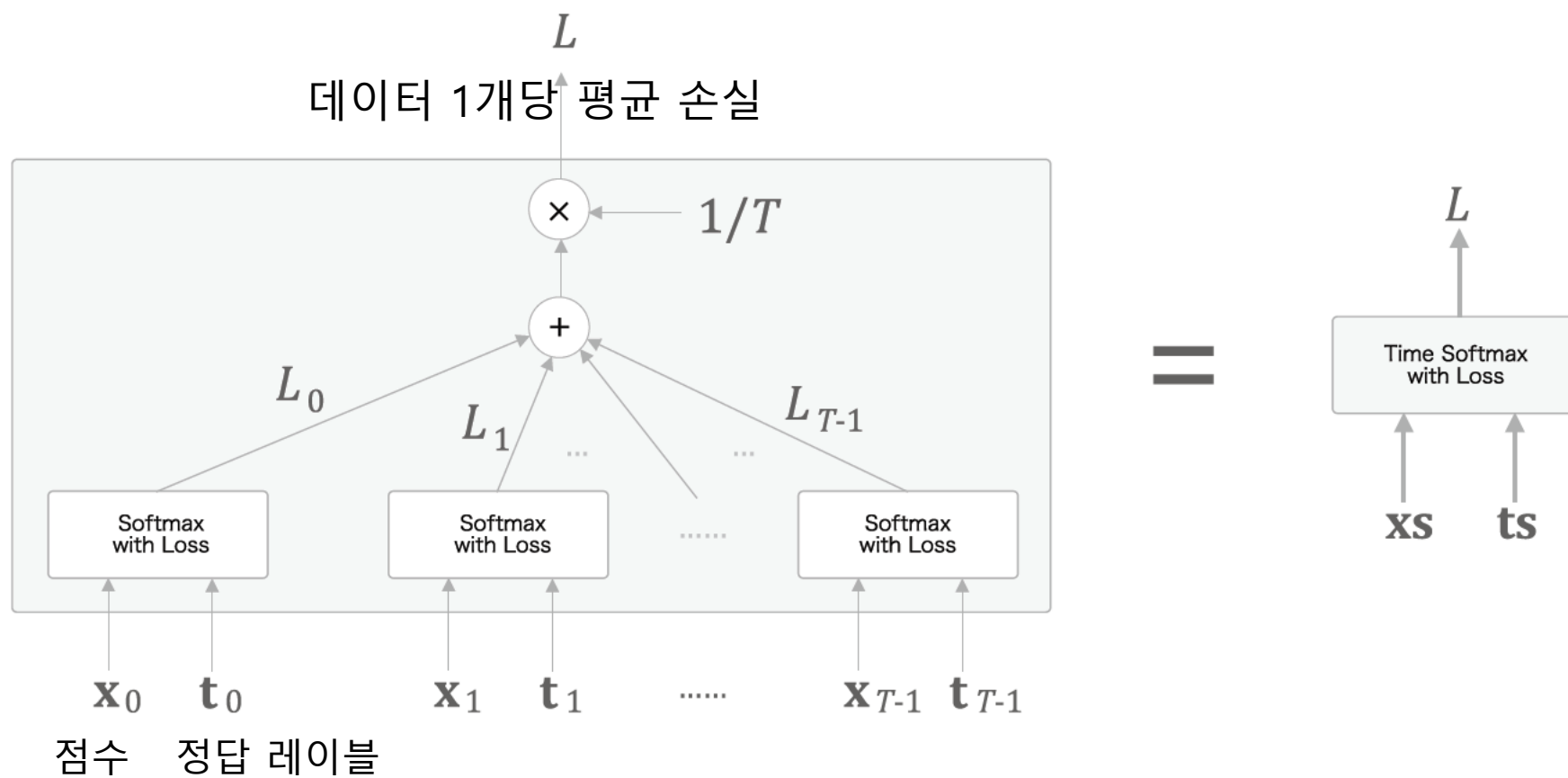


그림 5-26 샘플 말뭉치로 “you say goodbye and I say hello.”를 처리하는 RNNLM의 예



# Softmax with loss 계층

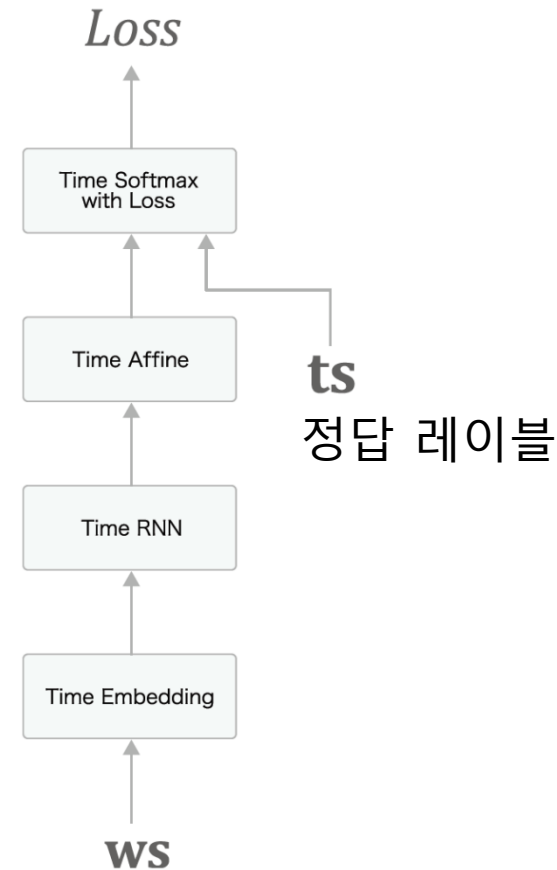
그림 5-29 Time Softmax with Loss 계층의 전체 그림



# RNNLM 구현과 평가

```
1 class SimpleRnnlm:
2     def __init__(self, vocab_size, wordvec_size, hidden_size):
3         V, D, H = vocab_size, wordvec_size, hidden_size
4         rn = np.random.randn
5
6         # 가중치 초기화
7         embed_W = (rn(V, D) / 100).astype('f')
8         rnn_Wx = (rn(D, H) / np.sqrt(D)).astype('f')
9         rnn_Wh = (rn(H, H) / np.sqrt(H)).astype('f')
10        # Xavier 초깃값 : 이전 계층의 노드가 n개라면 표준편차가 1/sqrt(n) 인 분포로 초기화
11        rnn_b = np.zeros(H).astype('f')
12        affine_W = (rn(H, V) / np.sqrt(H)).astype('f')
13        affine_b = np.zeros(V).astype('f')
14
15        # 계층 생성
16        # stateful = True --> 이전 시각의 은닉 상태를 계승
17        self.layers = [
18            TimeEmbedding(embed_W),
19            TimeRnn(rnn_Wx, rnn_Wh, rnn_b, stateful=True),
20            TimeAffine(affine_W, affine_b)
21        ]
22        self.loss_layer = TimeSoftmaxWithLoss()
23        self.rnn_layer = self.layers[1]
24
25        # 모든 가중치와 기울기를 리스트에 모은다.
26        self.params, self.grads = [], []
27        for layer in self.layers:
28            self.params += layer.params
29            self.grads += layer.grads
30
31        def forward(self, xs, ts):
32            for layer in self.layers:
33                xs = layer.forward(xs)
34            loss = self.loss_layer.forward(xs, ts)
35            return loss
36
37        def backward(self, dout=1):
38            dout = self.loss_layer.backward(dout)
39            for layer in reversed(self.layers):
40                dout = layer.backward(dout)
41            return dout
42
43        def reset_state(self):
44            self.rnn_layer.reset_state()
```

그림 5-30 SimpleRnnlm의 계층 구성: RNN 계층의 상태는 클래스 내부에서 관리



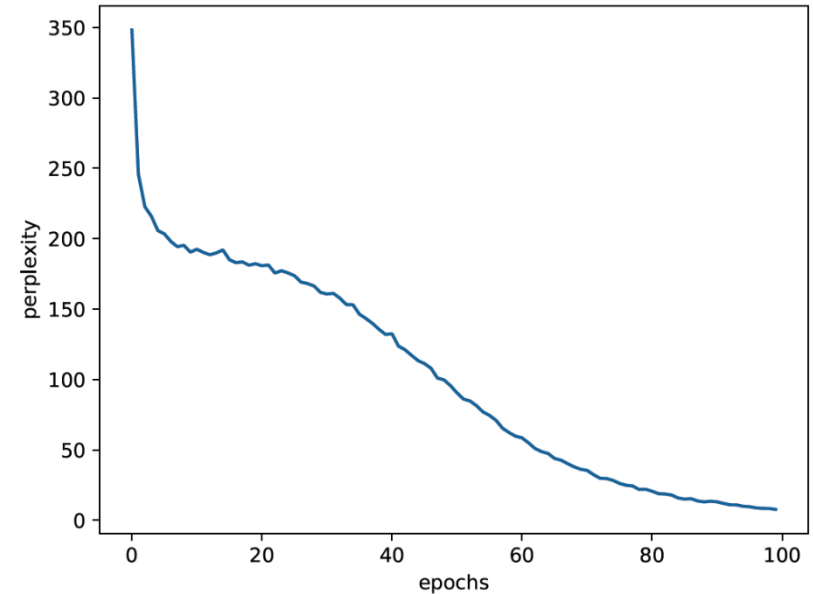
# 언어 모델의 평가

- 퍼플렉서티 (perplexity)
  - 확률의 역수(데이터 1개 개준)
  - 다음에 취할 수 있는 선택의 수
  - 낮을 수록 예측이 잘 된다고 판단

- 다수의 입력 데이터에 대해

- Perplexity =  $e^L$ ,  $L = -\frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk}$

그림 5-33 퍼플렉서티 추이



L: 신경망의 손실. CEE와 같은 식

N: 데이터의 총 개수

$t_n$ : 원-핫 벡터로 나타낸 정답 레이블

$t_{nk}$ : n번째 데이터의 k번째 값

$y_{nk}$ : 확률분포(신경망에서 Softmax의 출력)

# Ch.6

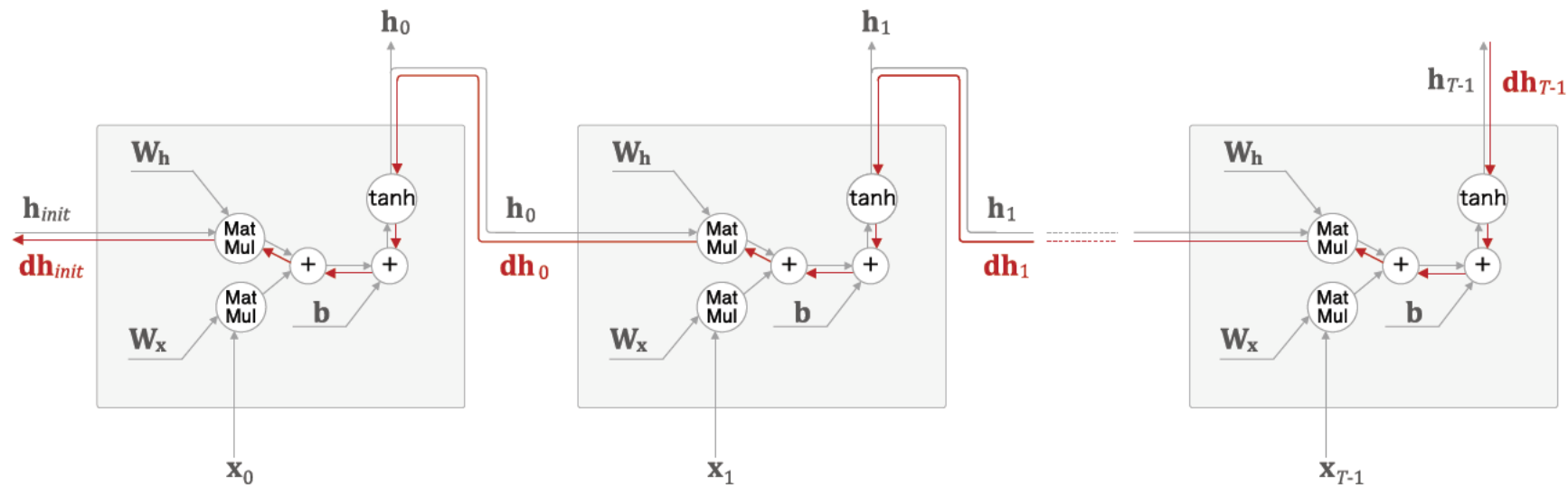
---

게이트가 추가된 RNN

# RNN의 문제점

- 시계열 데이터의 장기 의존 관계를 학습하기 어렵다

그림 6-5 RNN 계층에서 시간 방향으로의 기울기 전파



- $\tanh(x)$ 의 미분 값은 0~1 이므로 곱할 수록 그 값이 줄어든다
- 동일한  $W_h^T$ 가 계속 곱해질 때 기울기는 기하급수적으로 증가하거나 소실된다

그림 6-7 RNN 계층의 행렬 곱에만 주목했을 때의 역전파의 기울기

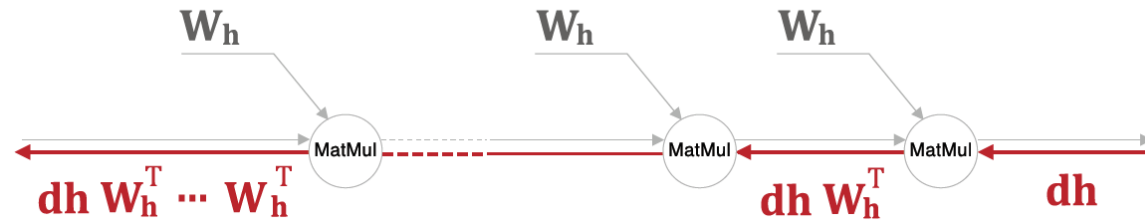
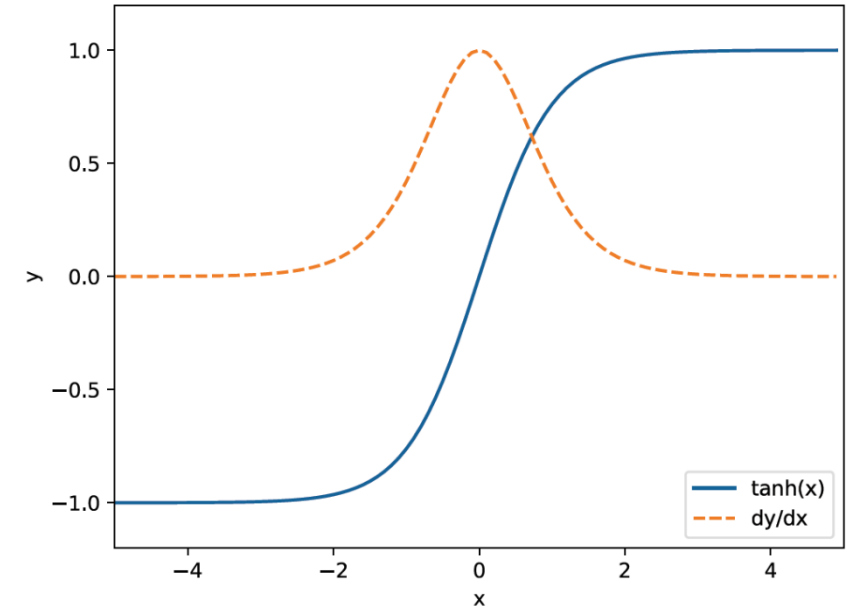


그림 6-6  $y = \tanh(x)$ 의 그래프(점선은 미분)



- 기울기 클리핑으로 해결
- 기울기를 하나로 모은  $\hat{g}$ 의 L2 norm이 threshold를 넘으면

$$\hat{g} = \frac{threshold}{\|\hat{g}\|} \hat{g}$$

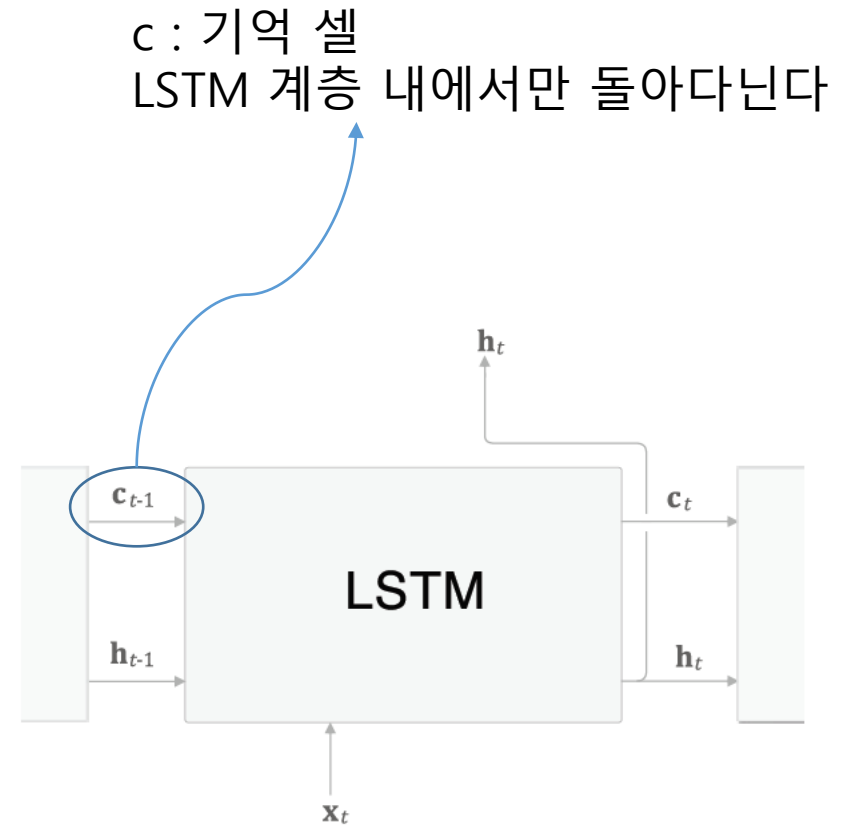
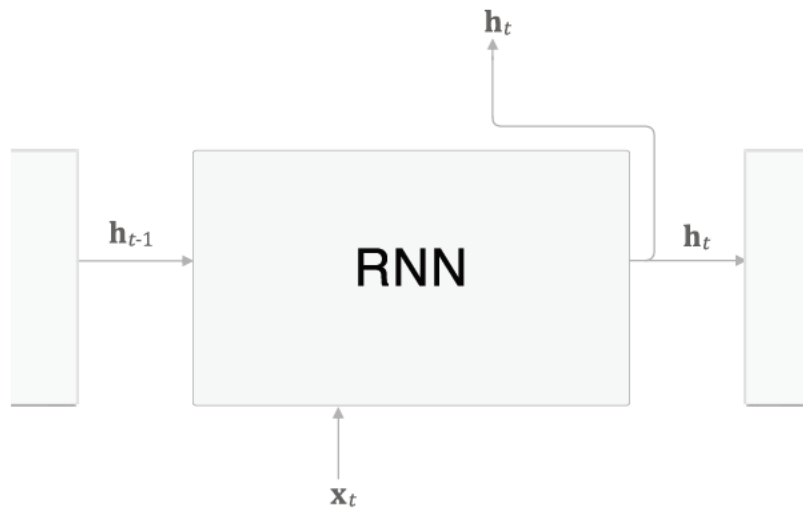
- 단순하지만 많은 경우에 잘 작동한다

```
1 def clip_grads(grads, max_norm):
2     total_norm = 0
3     for grad in grads:
4         total_norm += np.sum(grad**2)
5     total_norm = np.sqrt(total_norm)
6
7     rate = max_norm / (total_norm + 1e-6)
8     if rate < 1:
9         for grad in grads:
10             grad *= rate
```



# 기울기 소실과 LSTM

그림 6-11 RNN 계층과 LSTM 계층 비교



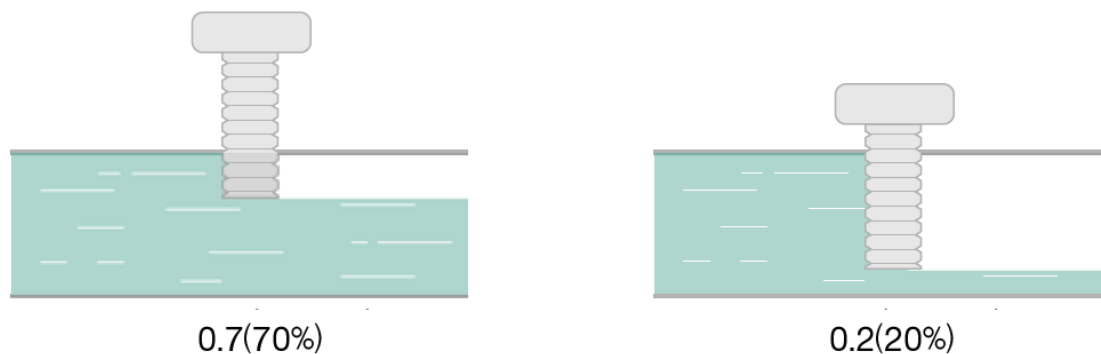
- $c_t$ 에는 시각  $t$ 에서의 LSTM의 기억이 저장됨
- 과거부터  $t$ 까지에 필요한 모든 정보가 저장되었다고 가정
- $c_t$ 는 세 개의 입력( $c_{t-1}, h_{t-1}, x_t$ )으로부터 '어떤 계산'을 수행함
- $h_t = \tanh(c_t)$

-- 게이트 --

데이터의 흐름 제어

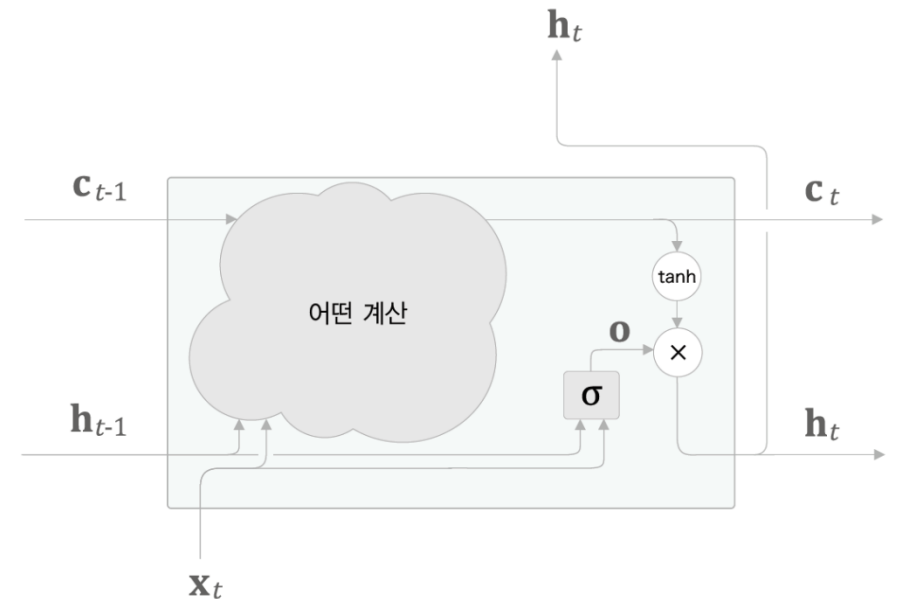
열림 상태로 sigmoid 사용

그림 6-14 물이 흐르는 양을 0.0~1.0 범위에서 제어한다.



# Output gate

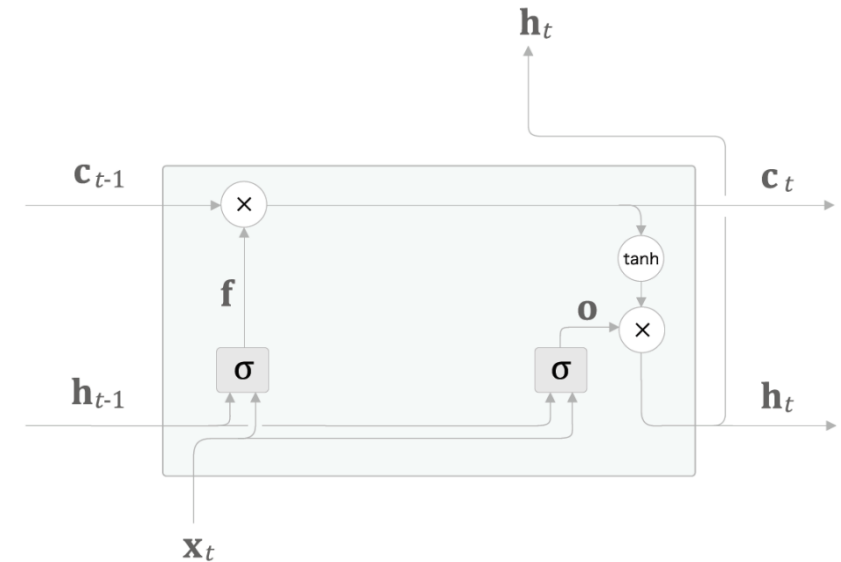
- $\tanh(c_t)$ 의 각 원소에 대해
- 다음 시각의 은닉 상태에 얼마나 중요한가를 조정
- 열림 상태  $\mathbf{o} = \sigma(\mathbf{x}_t \mathbf{W}_x^{(o)} + \mathbf{h}_{t-1} \mathbf{W}_h^{(o)} + \mathbf{b}^{(o)})$  그림 6-15 output 게이트 추가
- $\mathbf{h}_t = \mathbf{o} \odot \tanh(\mathbf{c}_t)$  (원소별 곱)



# Forget gate

- $c_{t-1}$ 의 기억 중에서 불필요한 기억을 잊게 해 준다
- $\mathbf{f} = \sigma(\mathbf{x}_t \mathbf{W}_x^{(f)} + \mathbf{h}_{t-1} \mathbf{W}_h^{(f)} + \mathbf{b}^{(f)})$
- $c_t = \mathbf{f} \odot c_{t-1}$

그림 6-16 forget 게이트 추가

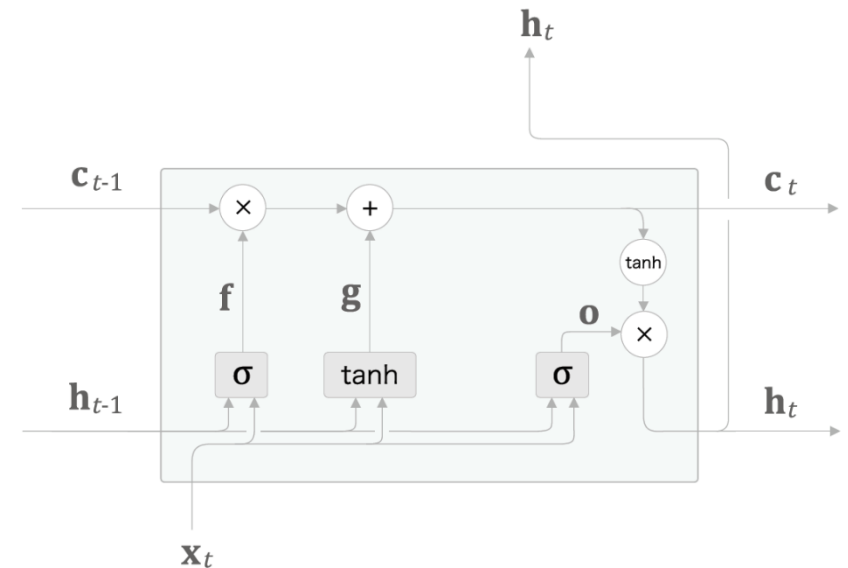


# 새로운 기억 셀

- 새로 기억해야 할 정보를 기억 셀에 추가
- 게이트가 아니며 정보 추가가 목적

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

그림 6-17 새로운 기억 셀에 필요한 정보를 추가



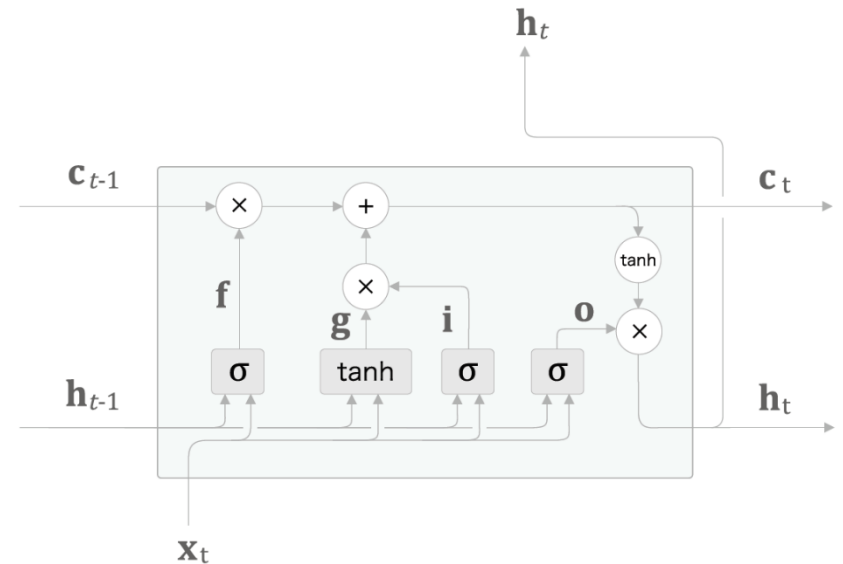
# Input gate

- g에 게이트를 추가
- g에 새롭게 추가되는 정보의 가치를 판단

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

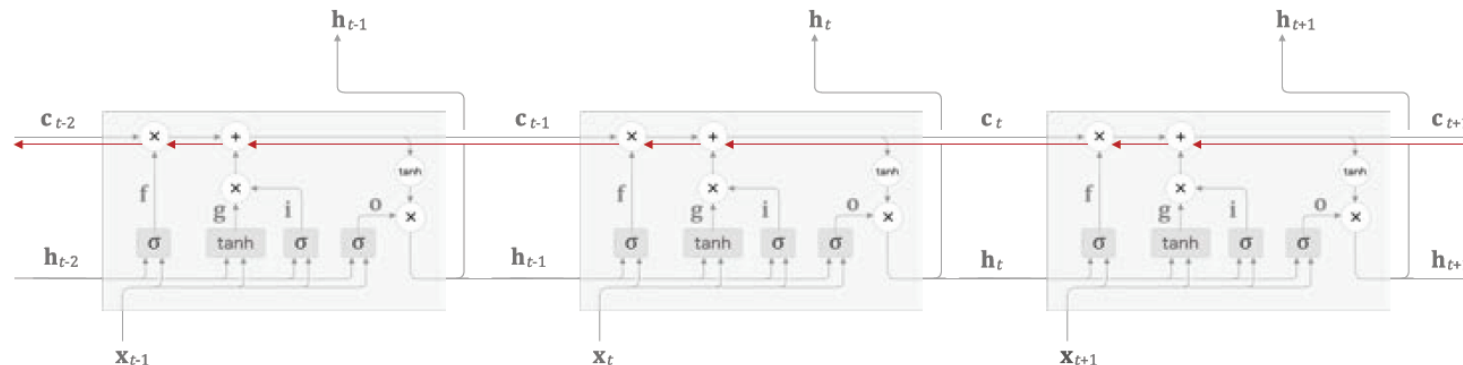
- $\mathbf{c}_t = \mathbf{f} \odot \mathbf{c}_{t-1} + \mathbf{g} \odot \mathbf{i}$

그림 6-18 input 게이트 추가



- 기억 셀의 역전파 시  $\oplus$  와  $\otimes$  노드만 지남
  - $\otimes$  노드는 원소별 곱이고, 게이트 값은 매번 바뀌므로 기울기 소실이 거의 일어나지 않음
- 
- Forget gate가 잊어야 한다고 판단하면 기울기가 작아지고
  - 잊으면 안된다고 판단하면 기울기가 약화되지 않은 채로 과거로 전파됨
- 오래 기억해야 할 정보가 소실없이 전파된다.

그림 6-19 기억 셀의 역전파



# LSTM 구현

- 4개분의 가중치를 하나로 모아서 slicing하며 계산

그림 6-21 4개분의 가중치를 모아 아핀 변환을 수행하는 LSTM의 계산 그래프

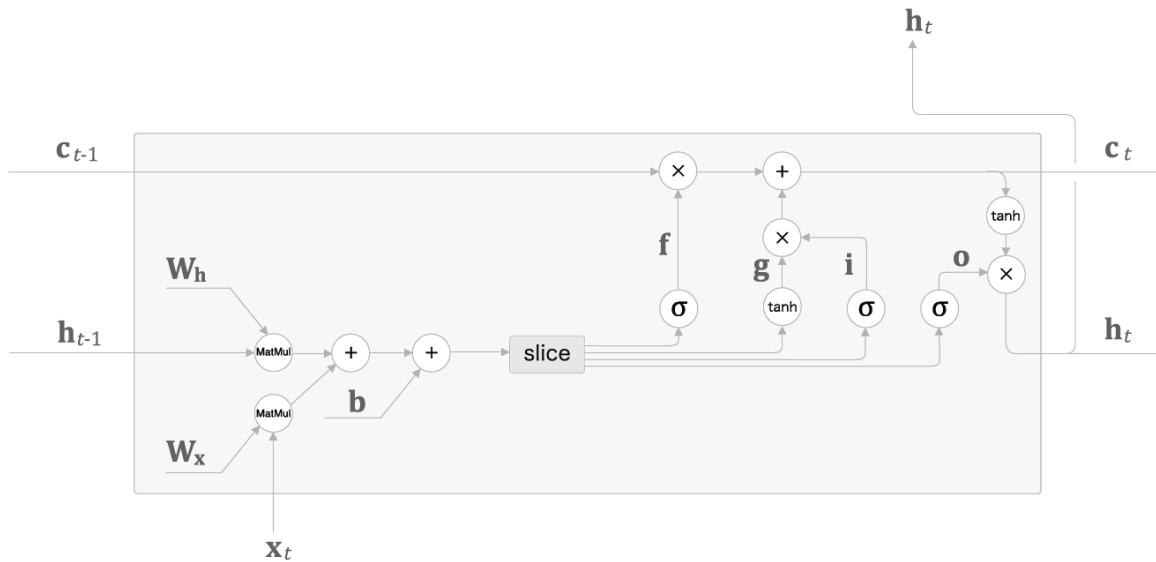
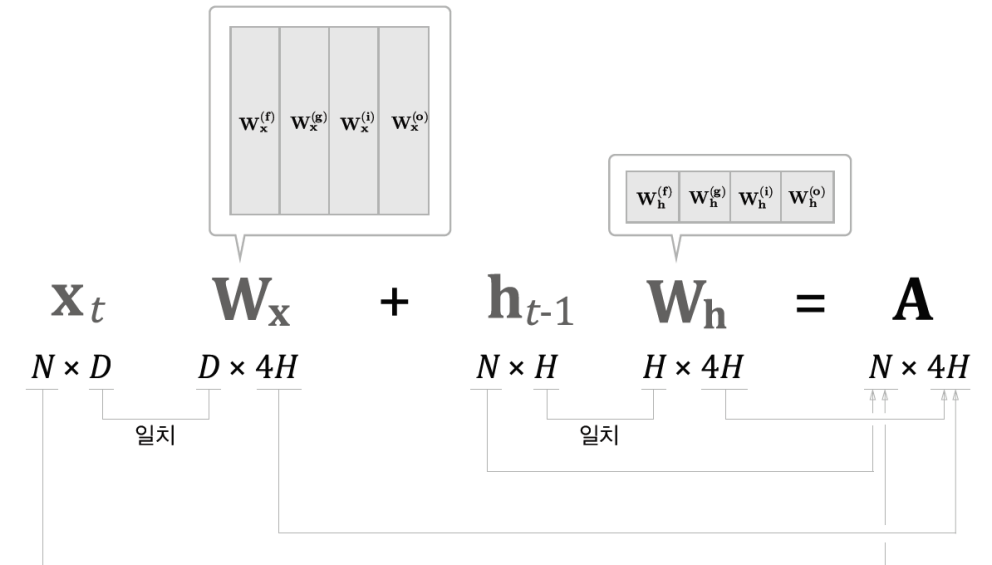


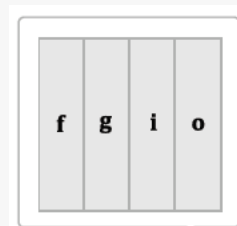
그림 6-22 아핀 변환 시의 형상 추이(편향은 생략)





# LSTM 구현

```
1 class LSTM:
2     def __init__(self, Wx, Wh, b):
3         self.params = [Wx, Wh, b]
4         self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
5         self.cache = None # 순전파 중간결과 보관용
6
7     def forward(self, x, h_prev, c_prev):
8         Wx, Wh, b = self.params
9         N, H = h_prev.shape
10        # 미니배치 수 N, 입력데이터 차원 수 D, c와 h의 차원 수 H
11        A = np.matmul(x, Wx) + np.matmul(h_prev, Wh) + b
12
13        # slice
14        f = A[:, :H] # forget gate
15        g = A[:, H:2*H] # 새로운 정보를 기억 셀에 추가 (게이트 아님!)
16        i = A[:, 2*H:3*H] # input gate
17        o = A[:, 3*H:] # output gate
18
19        f = sigmoid(f)
20        g = np.tanh(g)
21        i = sigmoid(i)
22        o = sigmoid(o)
23
24        c_next = f * c_prev + g * i
25        h_next = o * np.tanh(c_next)
26
27        self.cache = (x, h_prev, c_prev, i, f, g, o, c_next)
28        return h_next, c_next
```



$A$   
( $N \times 4H$ )

slice

$f$   
( $N \times H$ )

$g$   
( $N \times H$ )

$i$   
( $N \times H$ )

$o$   
( $N \times H$ )

## Time LSTM 구현

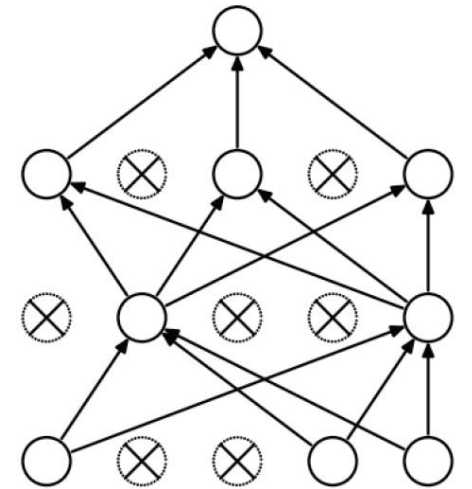
```
1 class TimeLSTM:
2     def __init__(self, Wx, Wh, b, stateful=False):
3         self.params = [Wx, Wh, b]
4         self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
5         self.layers = None
6         self.h, self.c = None, None
7         self.dh = None
8         self.stateful = stateful
9
10    def forward(self, xs): # T개 분량의 시계열 데이터를 하나로 모은 xs
11        Wx, Wh, b = self.params
12        N, T, D = xs.shape
13        # 미니배치 수 N, LSTM 개수 T, 입력데이터 차원 수 D
14        H = Wh.shape[0]
15        # h의 차원 수 H
16        self.layers = []
17        hs = np.empty((N, T, H), dtype='f')
18
19        if not self.stateful or self.h is None:
20            self.h = np.zeros((N, H), dtype='f')
21        if not self.stateful or self.c is None:
22            self.c = np.zeros((N, H), dtype='f')
23
24        for t in range(T):
25            layer = LSTM(*self.params) # parameter 복붙
26            self.h, self.c = layer.forward(xs[:, t, :], self.h, self.c)
27
28            hs[:, t, :] = self.h # 이 hs가 다시 윗줄의 forward()의 h로
29
30            self.layers.append(layer)
31
32    return hs
33
```

```
34    def backward(self, dhs): # 상류에서 오는 기울기 dhs
35        Wx, Wh, b = self.params
36        N, T, H = dhs.shape
37        D = Wx.shape[0]
38
39        dxs = np.empty((N, T, D), dtype='f') # 하류로 보내는 기울기 dxs
40        dh, dc = 0, 0
41
42        grads = [0, 0, 0]
43        for t in reversed(range(T)):
44            layer = self.layers[t]
45            dx, dh, dc = layer.backward(dhs[:, t, :] + dh, dc)
46            dxs[:, t, :] = dx # 각 시각의 기울기를 구해 dxs의 해당 시각 t에 저장
47            for i, grad in enumerate(layer.grads):
48                grads[i] += grad
49                # 각 RNN계층의 가중치 기울기를 합산하여
50
51        for i, grad in enumerate(grads):
52            self.grads[i][...] = grad # 최종 결과를 self.grads에 저장
53            self.dh = dh
54
55        return dxs
56
57    def set_state(self, h, c=None):
58        self.h, self.c = h, c
59
60    def reset_state(self):
61        self.h, self.c = None, None
```

# RNNLM 추가 개선

- LSTM계층을 깊이 쌓을 경우 정확도를 높일 수 있다
- LSTM 다층화 시 종종 과적합 발생
- RNN은 일반적 feed-forward신경망보다 과적합이 쉽게 일어남

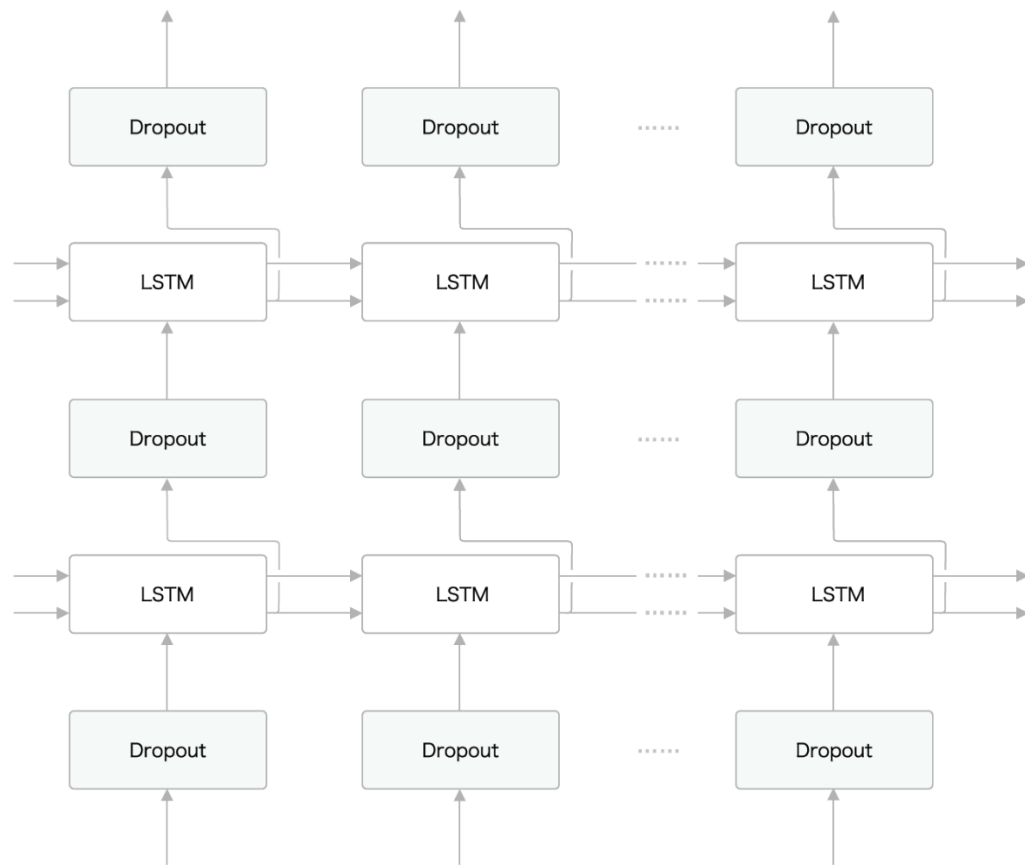
→ 훈련 데이터의 양 늘리기 & 모델의 복잡도 줄이기  
복잡도를 줄이는 '드롭아웃'을 적용해 보자



(b) 드롭아웃을 적용한 모습

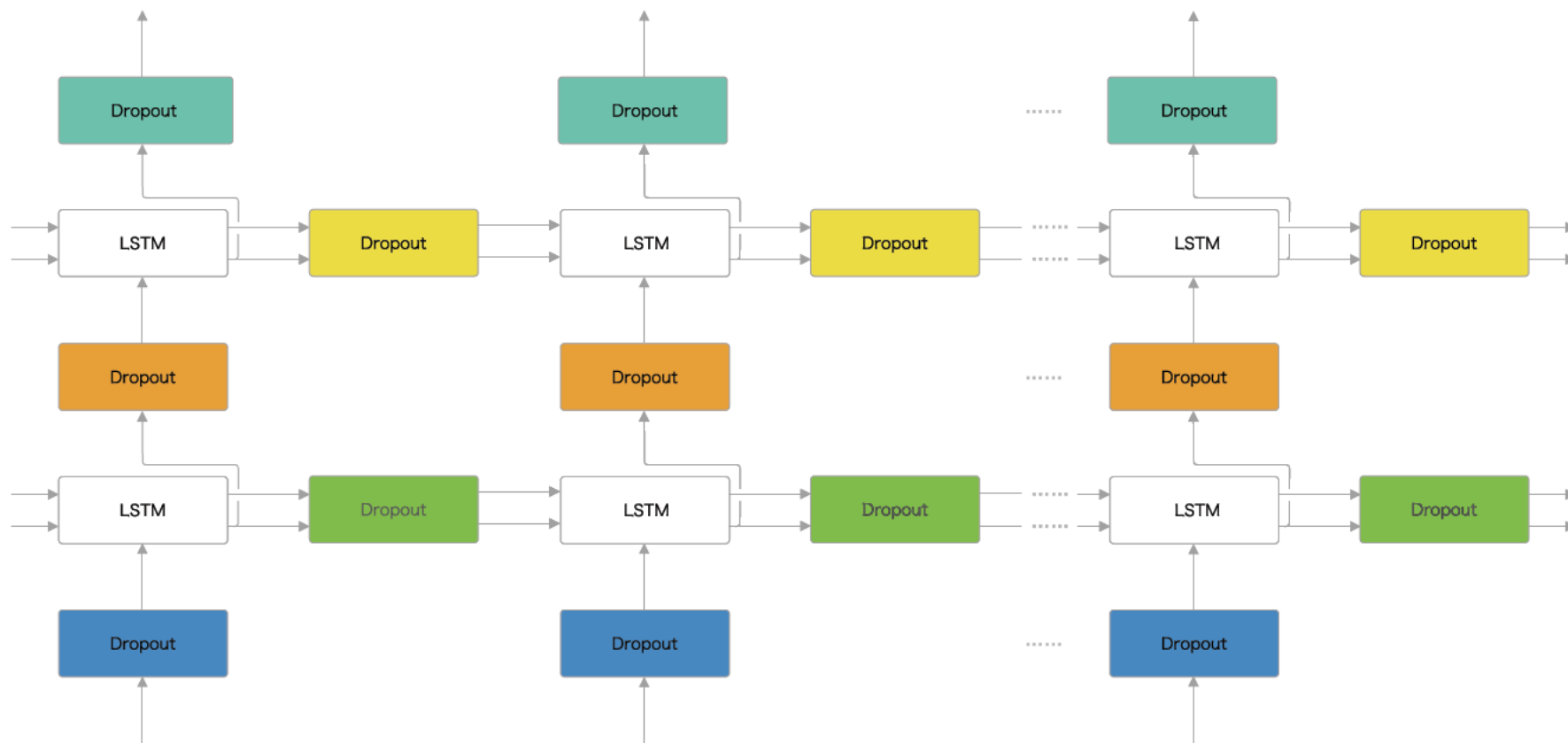
- 드롭아웃 계층이 들어가는 위치는 깊이(상하)방향으로 한다  
→ 시간(좌우)방향 진행 시 정보를 잃지 않음

그림 6-33 좋은 예: 드롭아웃 계층을 깊이 방향(상하)방향으로 삽입



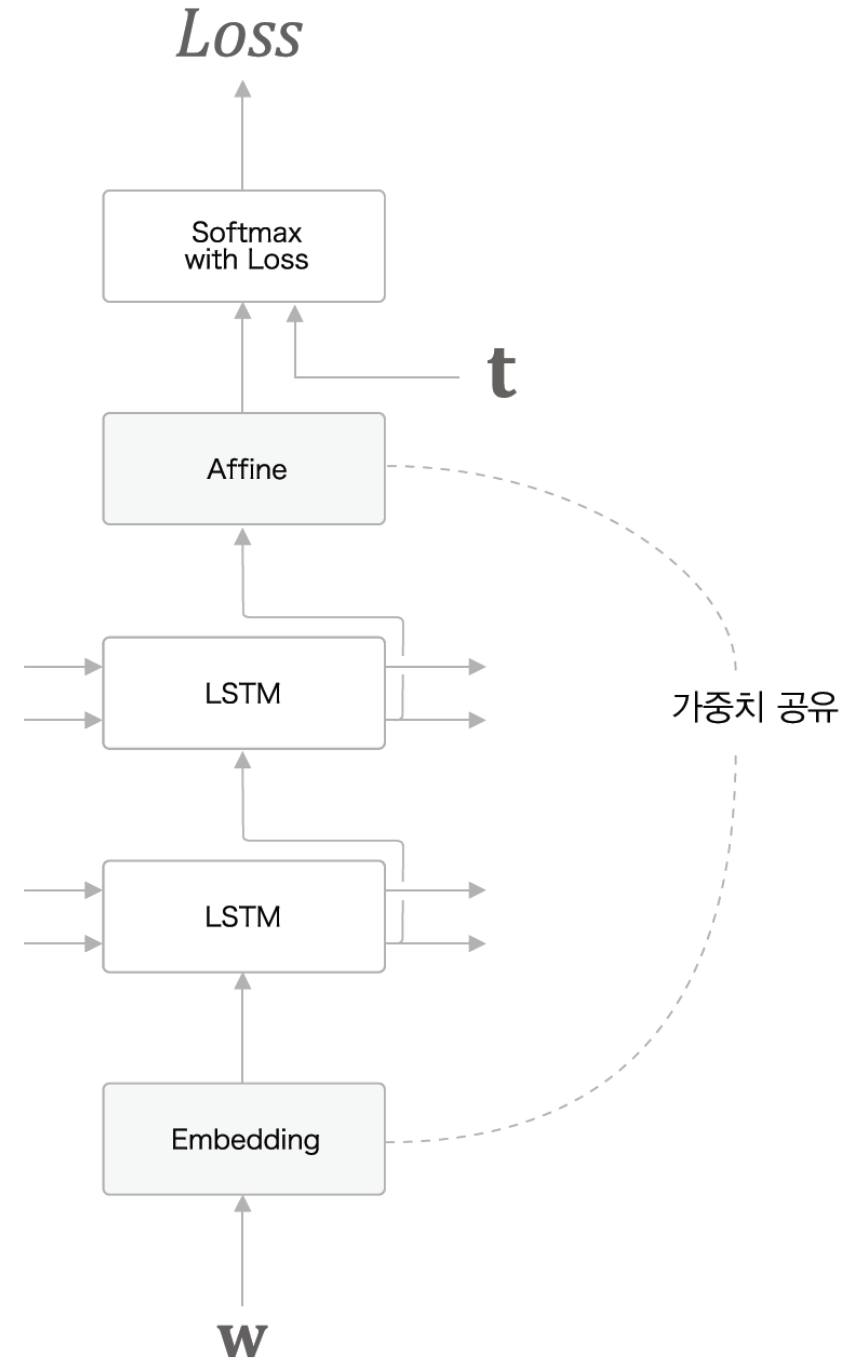
- 드롭아웃을 시간 방향으로도 적용시킨 '변형 드롭아웃'
  - 동일한 마스크를 공유하므로 정보를 잃는 방법도 고정됨
- 정보의 지수적 손실 차단

**그림 6-34** 변형 드롭아웃의 예: 색이 같은 드롭아웃끼리는 같은 마스크를 이용한다. 이처럼 같은 계층에 적용되는 드롭아웃끼리는 공통의 마스크를 이용함으로써 시간 방향 드롭아웃도 효과적으로 작동할 수 있다.



# 가중치 공유

- 두 계층이 가중치를 공유함으로써 학습할 매개변수가 줄고(과적합 억제), 정확도도 향상된다



# 개선된 RNNLM 구현

```
8 class BetterRnnlm(BaseModel):
9     ...
10     LSTM 계층을 2개 사용하고 각 층에 드롭아웃을 적용한 모델이다.
11     아래 [1]에서 제안한 모델을 기초로 하였고, [2]와 [3]의 가중치 공유(weight tying)를 적용했다.
12     [1] Recurrent Neural Network Regularization (https://arxiv.org/abs/1409.2329)
13     [2] Using the Output Embedding to Improve Language Models (https://arxiv.org/abs/1608.05859)
14     [3] Tying Word Vectors and Word Classifiers (https://arxiv.org/pdf/1611.01462.pdf)
15     ...
16     def __init__(self, vocab_size=10000, wordvec_size=650,
17                   hidden_size=650, dropout_ratio=0.5):
18         V, D, H = vocab_size, wordvec_size, hidden_size
19         rn = np.random.randn
20
21         embed_W = (rn(V, D) / 100).astype('f')
22         lstm_Wx1 = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
23         lstm_Wh1 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
24         lstm_b1 = np.zeros(4 * H).astype('f')
25         lstm_Wx2 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
26         lstm_Wh2 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
27         lstm_b2 = np.zeros(4 * H).astype('f')
28         affine_b = np.zeros(V).astype('f')
29
30         self.layers = [
31             TimeEmbedding(embed_W),
32             TimeDropout(dropout_ratio),
33             TimeLSTM(lstm_Wx1, lstm_Wh1, lstm_b1, stateful=True),
34             TimeDropout(dropout_ratio),
35             TimeLSTM(lstm_Wx2, lstm_Wh2, lstm_b2, stateful=True),
36             TimeDropout(dropout_ratio),
37             TimeAffine(embed_W.T, affine_b) # weight tying!!
38         ]
39         self.loss_layer = TimeSoftmaxWithLoss()
40         self.lstm_layers = [self.layers[2], self.layers[4]]
41         self.drop_layers = [self.layers[1], self.layers[3], self.layers[5]]
42
43         self.params, self.grads = [], []
44         for layer in self.layers:
45             self.params += layer.params
46             self.grads += layer.grads
47
```

```
48     def predict(self, xs, train_flg=False):
49         for layer in self.drop_layers:
50             layer.train_flg = train_flg
51
52         for layer in self.layers:
53             xs = layer.forward(xs)
54         return xs
55
56     def forward(self, xs, ts, train_flg=True):
57         score = self.predict(xs, train_flg)
58         loss = self.loss_layer.forward(score, ts)
59         return loss
60
61     def backward(self, dout=1):
62         dout = self.loss_layer.backward(dout)
63         for layer in reversed(self.layers):
64             dout = layer.backward(dout)
65         return dout
66
67     def reset_state(self):
68         for layer in self.lstm_layers:
69             layer.reset_state()
```

•

```
self.layers = [  
    TimeEmbedding(embed_W),  
    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!!  
]
```

드롭아웃

가중치 공유

다층 LSTM



# 정리

- 단순 RNN학습에서는 기울기의 소실과 폭발이 문제가 된다
- 기울기 폭발에는 클리핑,  
소실에는 게이트가 추가된 RNN(LSTM, GRU 등)이 효과적이다
- LSTM에는 input, forget, output 게이트가 있다
- 게이트는 전용 가중치를 가지며 0~1 사이의 실수를 출력한다
- 언어 모델 개선에는 LSTM 다층화, 드롭아웃, 가중치 공유 등의 기법이 효과적이다

감사합니다

---