

밑바닥부터 시작하는 딥러닝 2: 5장 순환 신경망(RNN)

2020/02/18


5. 순환 신경망(RNN)

- 피드 포워드 신경망의 문제점을 지적
 - 지금까지 살펴본 신경망은 모두 피드 포워드(feed forward) 유형의 신경망
 - 피드포워드(feed forward): 흐름이 단방향인 신경망, 한 방향으로만 신호가 전달된다.
 - 구성이 단순하여 구조 이해가 쉽지만, 시계열 데이터를 잘 다루지 못한다는 단점이 있다.
- 순환 신경망(RNN)의 구조를 들여다보고 파이썬으로 구현

5.1 확률과 언어 모델:


1. word2vec을 확률 관점에서 바라보다

$w_1 \ w_2 \ \dots \ w_{t-1} \ \boxed{w_t} \ w_{t+1} \ \dots \ w_{T-1} \ w_T$



- word2vec의 CBOW 모델에서 맥락이 w_{t-1}, w_{t+1} 일 때, 타깃이 w_t 가 될 확률: $P(w_t | w_{t-1}, w_{t+1})$
- 만약 맥락을 왼쪽 윈도우 만으로 한정한다면: $P(w_t | w_{t-2}, w_{t-1})$

$w_1 \ w_2 \ \dots \ w_{t-2} \ w_{t-1} \ \boxed{w_t} \ w_{t+1} \ \dots \ w_{T-1} \ w_T$



- 이런 CBOW 모델의 학습을 통해 손실 함수를 최소화하는 가중치 매개변수를 찾는다
 - 이를 통해 맥락으로부터 타깃을 정확하게 추측하고 단어의 의미가 인코딩된 '단어의 분산 표현'을 얻을 수 있다.
 - 그렇다면, 이렇게 '맥락으로부터 타깃을 추측하는 작업'을 어디에 이용할 수 있을까?
 - $P(w_t | w_{t-2}, w_{t-1})$ 은 어떤 실용적인 쓰임이 있을까?

5.1.2 언어 모델

- 언어 모델(Language Model): 단어 나열에 확률을 부여. 다시 말해, 특정한 단어의 시퀀스에 대해 그 시퀀스가 일어날 가능성이 어느 정도인지(얼마나 자연스러운 단어 순서인지)를 확률로 평가
 - 예를 들어,
 - "you say goodbye" => 높은 확률(ex. 0.092)
 - "you say good die" => 낮은 확률(ex. 0.000000032)
 - 다양하게 응용 가능
 - 기계 번역, 음성 인식 등
 - 새로운 문장 생성(7장)
 - w_1, \dots, w_m 이라는 m개의 단어로 된 문장이 있을 때,
 - 단어가 w_1, \dots, w_m 이라는 순서로 출현할 확률: $P(w_1, \dots, w_m)$
 - 확률의 곱셈 정리를 이용해 위 식을 정리하면,
 - $P(w_1, \dots, w_m) = P(w_m|w_1, \dots, w_{m-1}) P(w_{m-1}|w_1, \dots, w_{m-2}) \cdots P(w_3|w_1, w_2) P(w_2|w_1) P(w_1) = \prod_{t=1}^m P(w_t|w_1, \dots, w_{t-1})$
 - 이 사후확률은 타깃 단어보다 왼쪽에 있는 모든 단어를 맥락으로 했을 때의 확률 식과 같다.

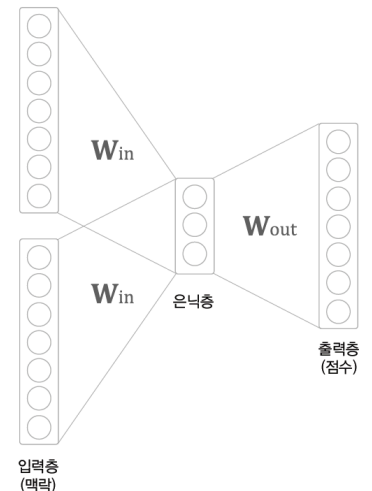
$w_1 \ w_2 \ \cdots \ w_{t-1} \ w_t \ \cdots \ w_m$

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

- 따라서 $P(w_t|w_1, w_2, \dots, w_{t-1})$ 를 계산할 수 있으면 언어모델의 동시 확률 $P(w_1, \dots, w_m)$ 도 구할 수 있다!

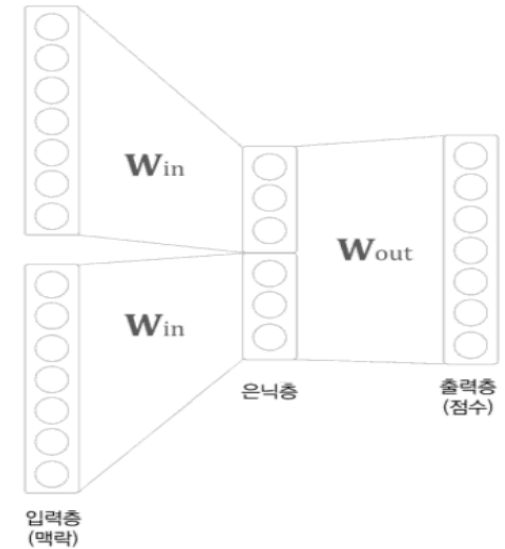
5.1.3 CBOW 모델을 언어 모델로?

- 그렇다면 word2vec의 CBOW 모델을 언어 모델에 적용하려면 어떻게 해야 할까?
 - 맥락의 크기를 특정 값 (window 값)으로 한정하여 근사적으로 나타낼 수 있다.
 - $P(w_1, \dots, w_m) = \prod_{t=1}^m P(w_t | w_1, \dots, w_{t-1}) \approx \prod_{t=1}^m P(w_t | w_{t-2}, w_{t-1})$
 - 맥락을 왼쪽 2개 단어로 한정
 - CBOW 모델의 사후 확률($P(w_t | w_{t-2}, w_{t-1})$)에 따라 근사적으로 나타낼 수 있다.
 - 마르코프 연쇄/모델(Markov Chain/Model): 미래의 상태가 현재 상태에만 의존해 결정
 - 여기서는 직전의 2개의 단어에만 의존해 다음 단어가 정해지는 모델이므로 '2층 마르코프 연쇄'라고 부를 수 있다.
- 예시에서는 맥락을 2개로 한정지었지만, 이 맥락의 크기는 임의로 설정할 수 있다.
 - 이렇게 임의로 길이를 설정할 수 있다고 해도, 결국 **특정 길이로 '고정'**되게 되며, 그 맥락보다 더 왼쪽에 있는 정보는 무시된다.
 - "Tom was watching TV in his room. Mary came into the room. Mary said hi to ? "
 - 문장의 맥락을 고려하면 ? 안에는 'Tom' 또는 'him'이 들어가야 한다.
 - 하지만 정답을 구하기 위해서는 ?로부터 18번째나 앞에 나오는 Tom을 기억해야 한다.
 - 맥락이 이보다 작았다면, 이 문제에 대한 정답을 찾을 수 없었을 것.
 - 맥락의 크기를 키울 수는 있지만, CBOW 모델은 맥락 안의 단어 순서가 무시된다는 한계가 있음
 - (you, say)와 (say, you)는 같은 맥락으로 취급됨



5.1.3 CBOW 모델을 언어 모델로?

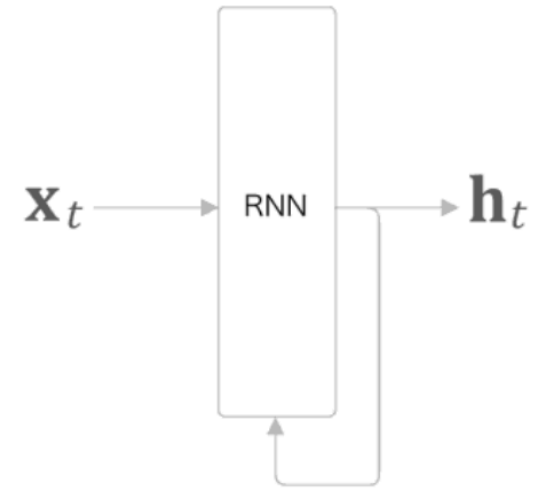
- 따라서 맥락의 단어 순서도 고려한 모델을 만들어야 함
 - 맥락의 단어 벡터를 은닉층에서 연결하는 방식
 - 신경 확률론적 언어 모델(Neural Probabilistic Language Model, [28])에서 이 방식 사용
 - 하지만 이 방법은 맥락의 크기에 비례해 가중치 매개변수도 늘어나게 됨
- **순환 신경망(RNN)을 이용**
 - RNN은 맥락이 아무리 길더라도 그 맥락의 정보를 기억하는 매커니즘을 가짐
 - 아무리 긴 시계열 데이터에라도 대응 가능



5.2 RNN이란:

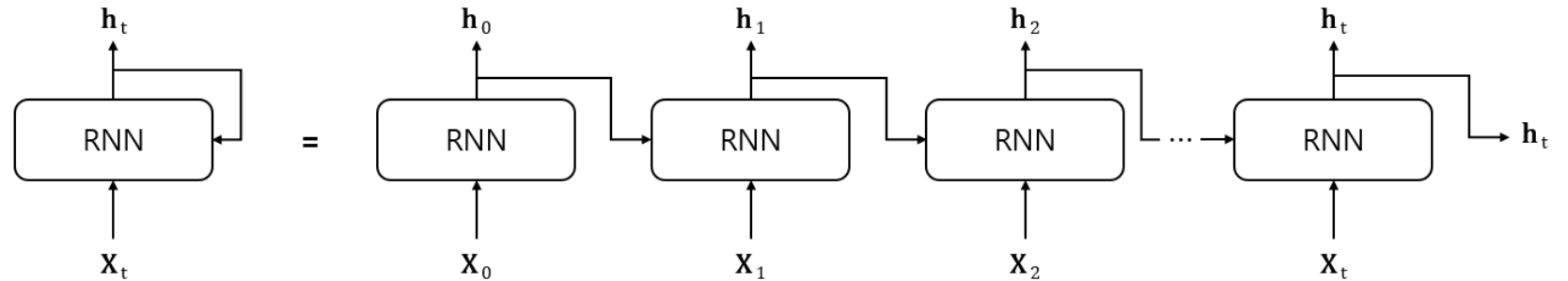
1. 순환하는 신경망

- RNN(Recurrent Neural Network): 순환 신경망
- Recurrent = > '순환한다'
 - 어느 한 지점에서 시작해, 시간이 지나 다시 원래 장소로 돌아오는 과정을 반복하는 것
 - 여기서 주목할 사실은, 순환 하기 위해서는 '닫힌 경로'가 필요하다는 것!
 - 그래야 데이터가 같은 장소를 반복해 왕래할 수 있다.
 - 그렇게 데이터가 순환하면서 정보가 끊임없이 갱신되게 된다.
- RNN의 특징은 바로 순환하는 경로(닫힌 경로)가 있다는 것.
 - 이 경로를 따라 데이터는 끊임없이 순환할 수 있으며, 데이터가 순환되기 때문에 과거의 정보를 기억하는 동시에 최신 데이터로 갱신 가능
 - 입력: x_t (t는 시간) => 시계열 데이터 $w_0, w_1, \dots, w_t, \dots$ 가 RNN 계층에 입력됨을 표현한 것
 - 그 입력에 대응해 $h_0, h_1, \dots, h_t, \dots$ 가 출력



5.2.2 순환 구조 펼치기

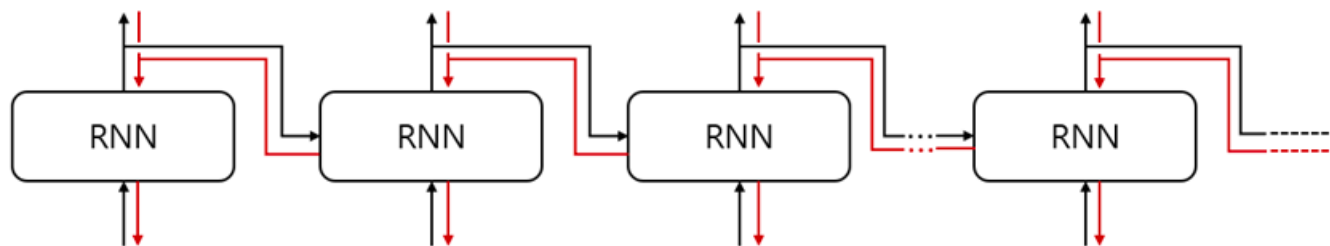
- RNN 계층의 순환 구조:



- 각 시각 t 의 RNN 계층은 그 계층으로의 입력과 1개 전의 RNN 계층으로부터 출력을 받아 현 시각의 출력을 계산한다.
=> $h_t = \tanh(h_{t-1}W_h + x_tW_x + b)$
 - W_x : 입력 x 를 출력 h 로 변환하기 위한 가중치
 - W_h : RNN 출력을 다음 시각의 출력으로 변환하기 위한 가중치
 - b : 편향
 - h_{t-1}, x_t : 행벡터
- 출력 h_t 는 다른 계층을 향해 위쪽으로 출력되는 동시에 다음 시각의 RNN 계층(자기 자신)을 향해 오른쪽으로도 출력 => 하나의 출력이 분기해 전달됨!
- 현재의 출력(h_t)은 한 시각 이전의 출력(h_{t-1})에 기초해 계산됨
 - 즉 RNN은 h 라는 상태를 기억해 1 스텝이 진행될 때 마다, 위의 activation function식 형태로 갱신
 - 따라서 RNN의 출력 h_t 를 은닉 상태(hidden state) 혹은 은닉 상태 벡터(hidden state vector)라고도 함.

5.2.3 BPTT

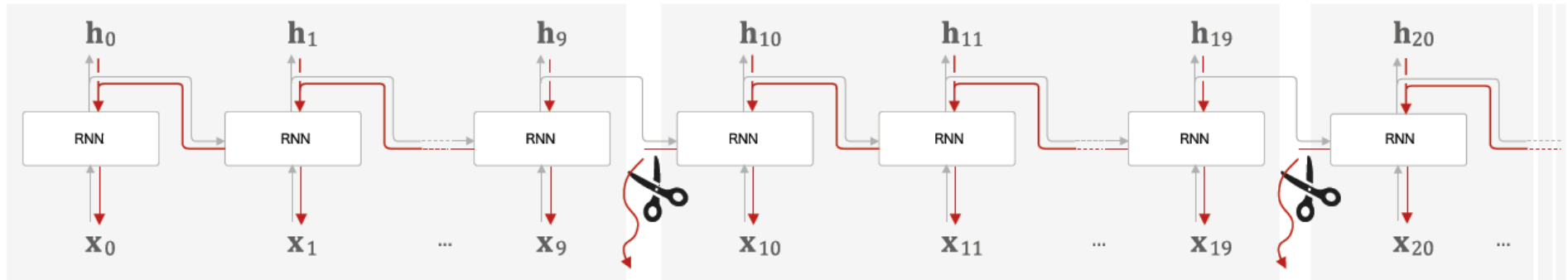
- RNN의 학습도 보통 신경망과 같은 순서로 진행된다.



- **BPTT(Backpropagation Through Time):** 시간 방향으로 펼친 신경망의 오차역전파법
 - 하지만 긴 시계열 데이터를 학습할 때 문제가 발생한다.
 - 시계열 데이터의 시간 크기가 커지는 것이 비례하여 BPTT가 소비하는 컴퓨팅 자원도 증가
 - 시간의 크기 커질수록 역전파 시의 기울기가 불분명해진다.
 - BPTT를 이용해 기울기를 구하는 경우, 매 시각 RNN 계층의 중간 데이터를 메모리에 유지해두어야 한다.

5.2.4 Truncated BPTT

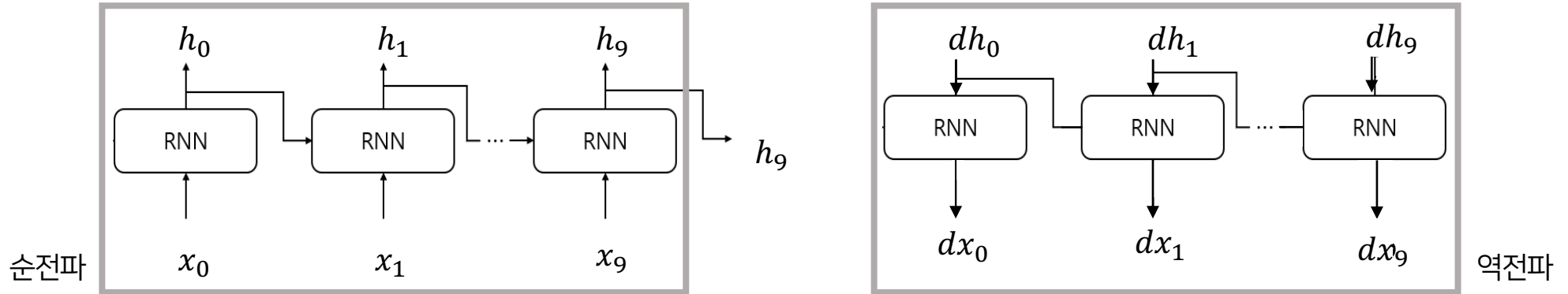
- **Truncated BPTT:** 시간축 방향으로 너무 길어진 신경망을 적당한 지점에서 잘라 작은 신경망 여러개로 만들고, 잘라낸 작은 신경망에서 오차역전파법을 수행하는 방법



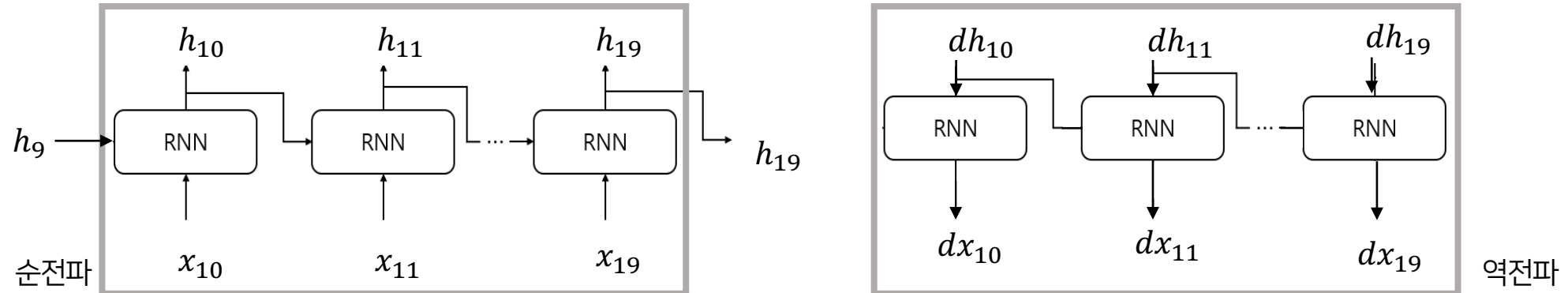
- 순전파의 연결은 그대로 유지하고, 역전파의 연결만 적당한 길이로 잘라내 잘라낸 신경망 단위로 학습을 수행
- eg) 길이가 1,000인 시계열 데이터
 - 자연어 문제라면 단어 1,000개짜리 말뭉치
 - 하지만 길이가 1,000개인 시계열 데이터를 다루려면 RNN 계층이 가로로 1,000개나 늘어선 신경망이 됨
 - 계산량과 메모리 사용량에서 문제
 - 신경망 통과할 때 마다 기울기 값이 작아지는 vanishing gradient 문제 발생
 - 역전파의 연결을 적당한 길이로 끊으면 이를 해결할 수 있음!
- 역전파의 연결은 끊어지지만 순전파의 연결은 끊어지지 않기 때문에 데이터는 꼭 순차적으로 입력되어야 한다.

5.2.4 Truncated BPTT

- Truncated BPTT 방식으로 RNN을 학습시켜보자.
 - 첫번째 블록 입력 데이터(x_0, \dots, x_9)을 RNN 계층에 입력

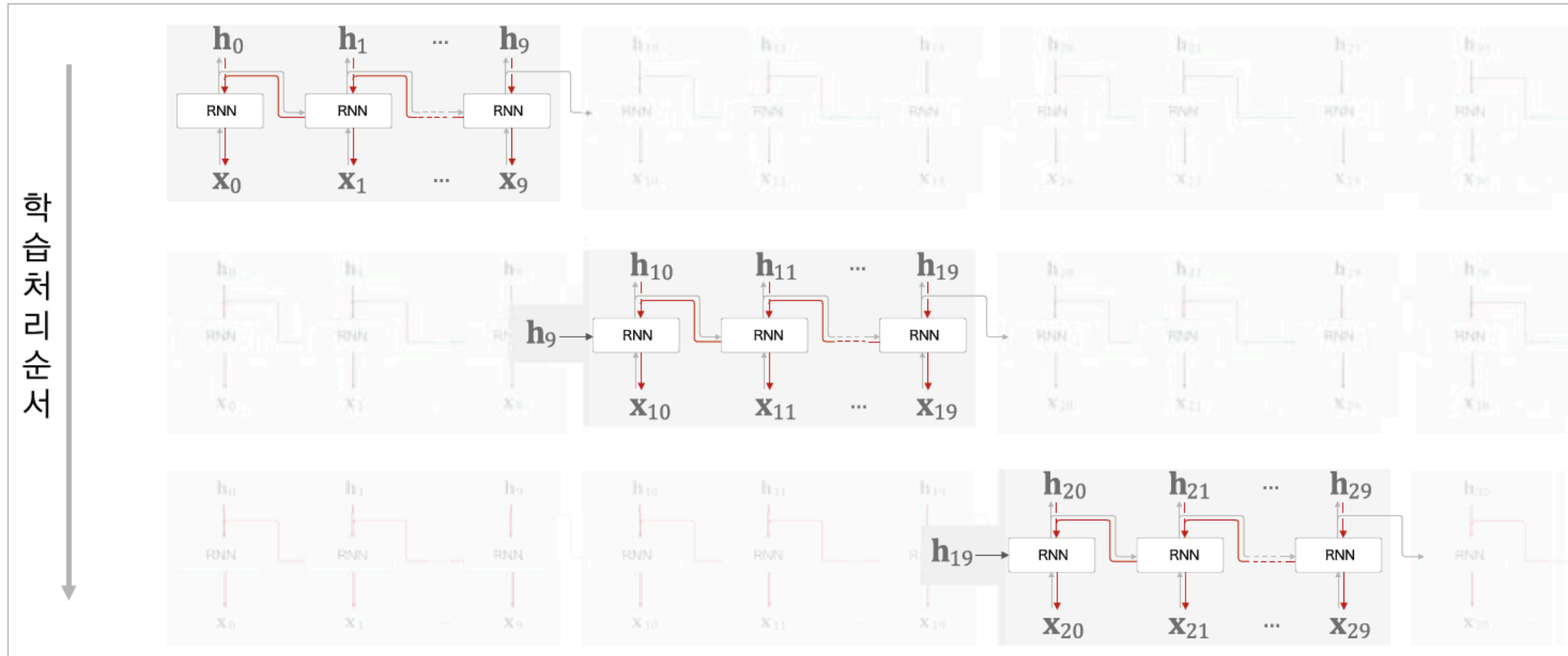


- 앞선 시간으로부터의 기울기는 끊겼기 때문에 이 블록 내에서만 back propagation이 완결
- 순전파를 먼저 수행한 뒤 역전파 수행 => 기울기를 구함
- 이어서 다음 블록의 입력 데이터(x_{10} 에서 x_{19})를 입력



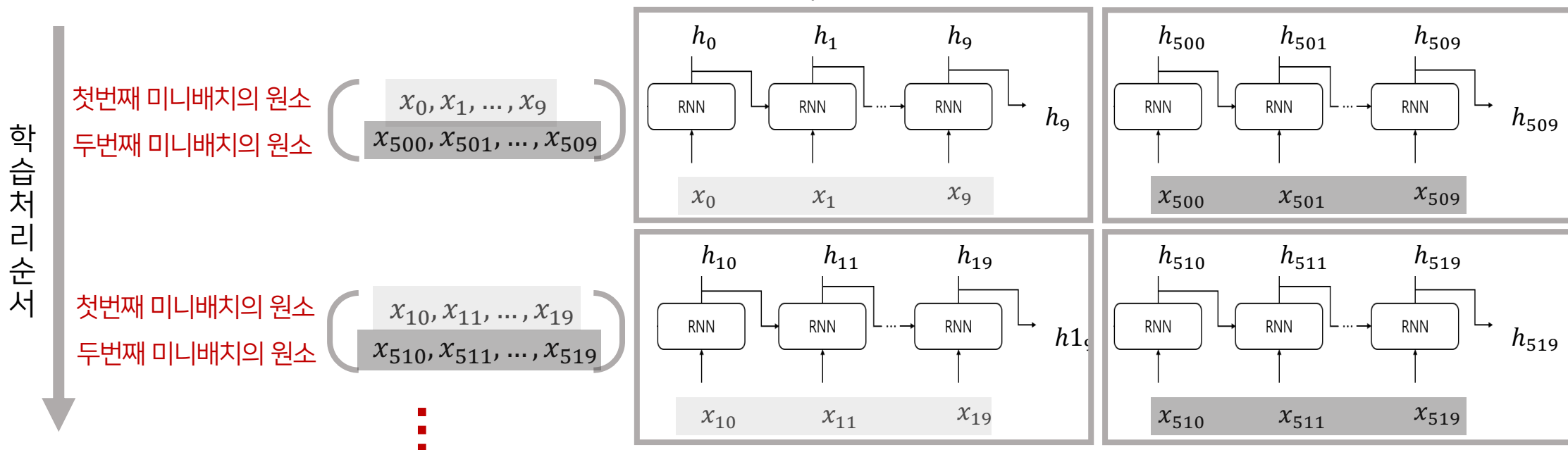
- 앞 블록의 마지막 은닉 상태 h_9 으로 순전파 계속 연결

5.2.4 Truncated BPTT



5.2.5 Truncated BPTT 미니배치 학습

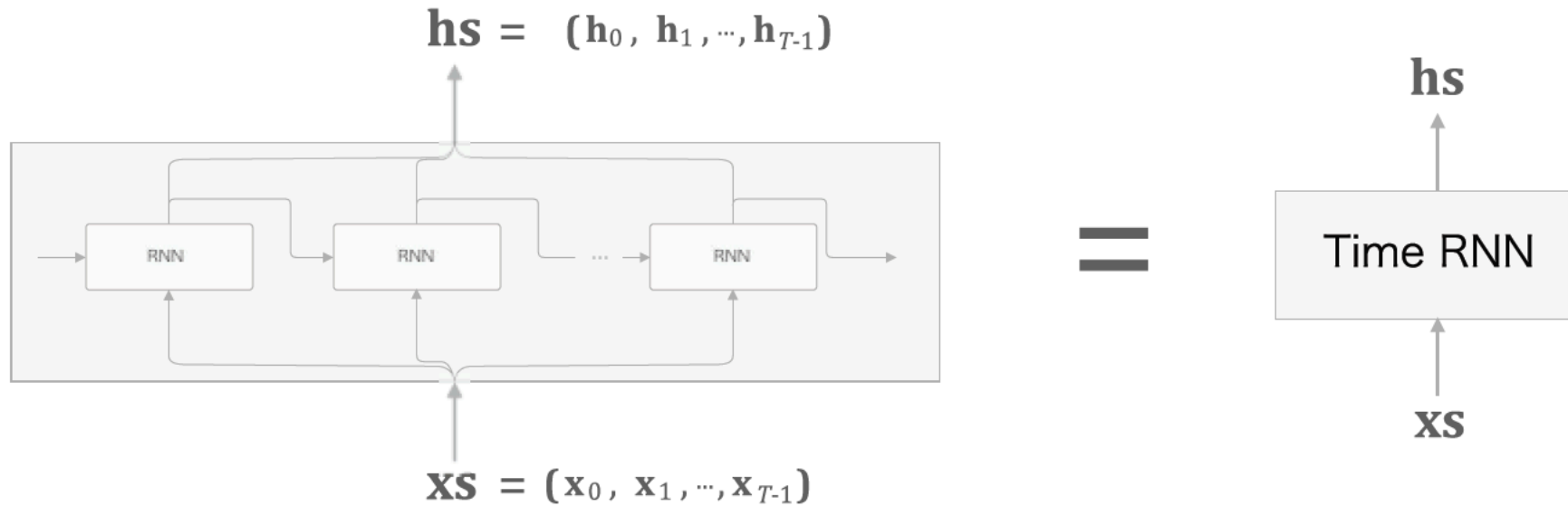
- 데이터를 주는 시작 위치를 각 미니배치의 시작 위치로 옮겨주고 난 뒤 학습 진행
- eg) 길이가 1,000인 시계열 데이터에서 시각의 길이를 10개 단위로 잘라 Truncated BPTT로 학습, $\text{mini_batch} = 2$
 - 첫번째 미니배치: 처음부터 순서대로 데이터를 제공
 - 두번째 미니배치: 500번째 데이터를 시작 위치로 정하고, 그 위치부터 다시 순서대로 데이터 제공



- 각 미니배치의 시작 위치를 오프셋으로 옮겨준 후 순서대로 제공
- 데이터를 순서대로 입력하다가 끝에 도달하면 다시 처음부터 입력

5.3 RNN 구현

- RNN 계층을 T개 연결한 신경망(Time RNN)을 구현해보자.



- RNN의 한 단계를 처리하는 클래스를 RNN이라는 이름으로 구현
- 이 RNN 클래스를 이용해 T개 단계의 처리를 한꺼번에 수행하는 TimeRNN이라는 클래스로 구현

5.3.1 RNN 계층 구현

- RNN 처리를 한 단계만 수행하는 RNN 클래스부터 구현해보자.
- RNN의 순전파 식: $h_t = \tanh(h_{t-1}W_h + x_tW_x + b)$

```
# 5.3.1 RNN 계층 구현
class RNN:
    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

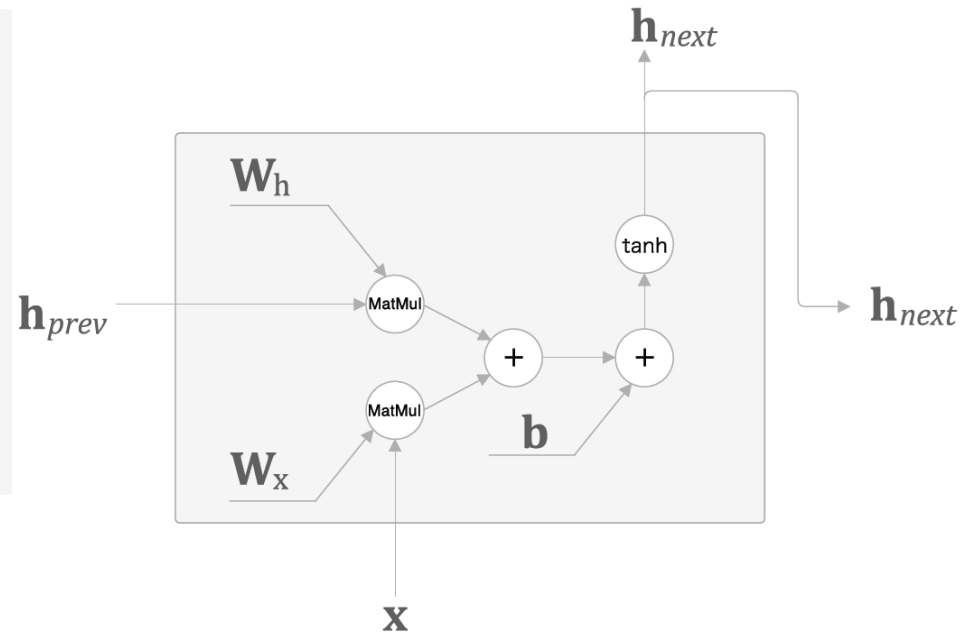
    def forward(self, x, h_prev):
        Wx, Wh, b = self.params
        t = np.dot(h_prev, Wh) + np.dot(x, Wx) + b
        h_next = np.tanh(t)

        self.cache = (x, h_prev, h_next)
        return h_next
```

가중치 2개와 편향 1개를 인수로 받는다.

아래로부터의 입력 x와
왼쪽으로부터의 입력 h_prev

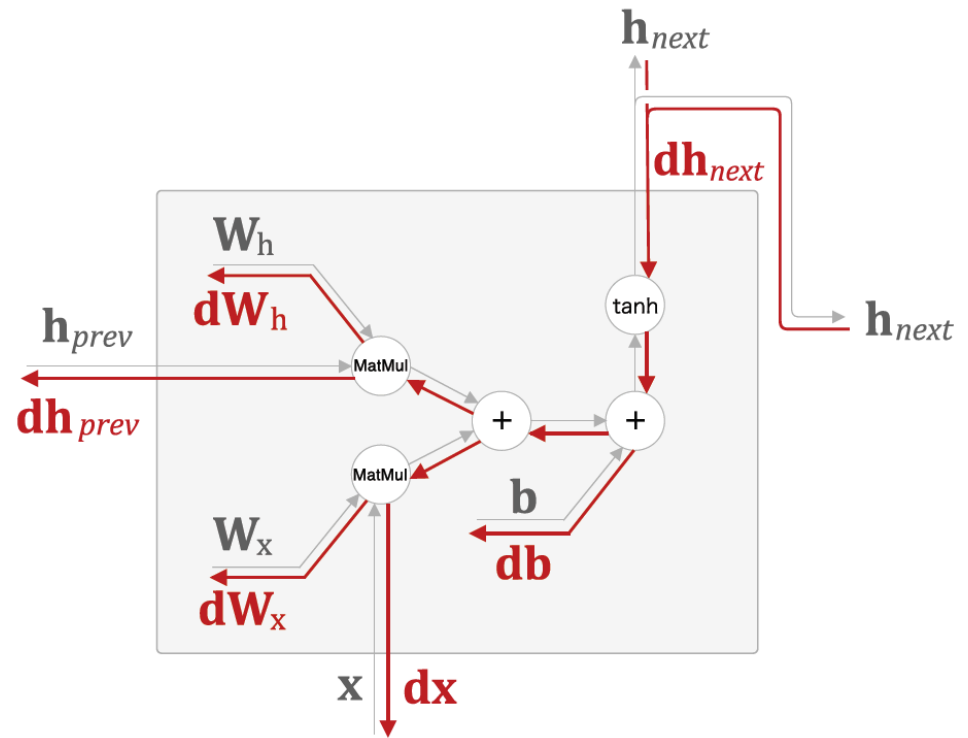
그림 5-19 RNN 계층의 계산 그래프(MatMul 노드는 행렬의 곱셈을 나타냄)



5.3.1 RNN 계층 구현

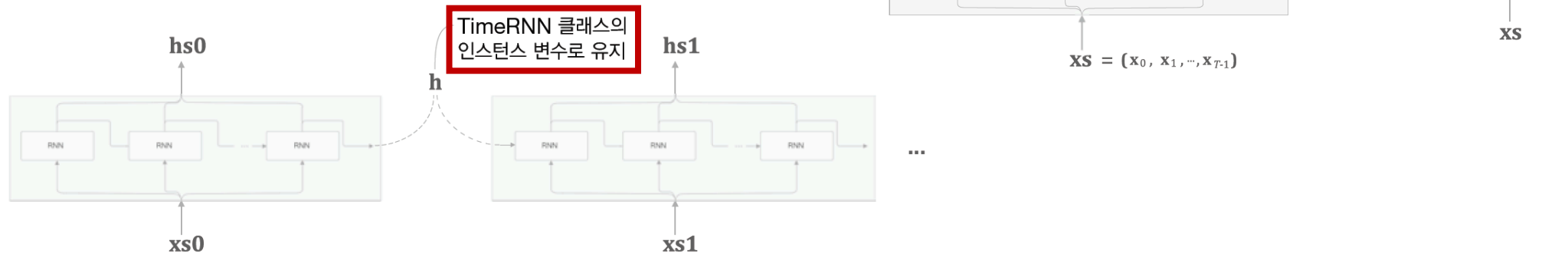
- RNN 처리를 한 단계만 수행하는 RNN 클래스부터 구현해보자.
- 역전파는 순전파때와는 반대 방향으로 수행하면 된다. 그림 5-20 RNN 계층의 계산 그래프(역전파 포함)

```
def backward(self, dh_next):  
    Wx, Wh, b = self.params  
    x, h_prev, h_next = self.cache  
  
    dt = dh_next * (1 - h_next ** 2)  
    db = np.sum(dt, axis=0)  
    dWh = np.dot(h_prev.T, dt)  
    dh_prev = np.dot(dt, Wh.T)  
    dWx = np.dot(x.T, dt)  
    dx = np.dot(dt, Wx.T)  
  
    self.grads[0][...] = dWx  
    self.grads[1][...] = dWh  
    self.grads[2][...] = db  
  
    return dx, dh_prev
```



5.3.2 Time RNN 계층 구현

- Time RNN 계층: T개의 RNN 계층으로 구성
 - RNN 계층의 은닉 상태 h 를 인스턴스 변수로 유지
 - 은닉 상태를 '인계'받는 용도로 이용된다.



5.3.2 Time RNN 계층 구현

class TimeRNN:

```
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.dh = None, None # h: forward() 매서드를 불렀을 때의 마지막 RNN 계층의 은닉 상태를 저장
                             # dh: backward() 매서드를 불렀을 때 하나 앞 블록의 은닉 상태의 기울기를 저장
self.stateful = stateful
```

```
def set_state(self, h):
    self.h = h # TimeRNN 계층의 은닉 상태를 설정하는 매서드
```

```
def reset_state(self):
    self.h = None # TimeRNN 계층의 은닉 상태를 초기화하는 매서드
```

- * stateful: 은닉 상태를 인계받을지 여부(불리언 값)
 - stateful = True:
 - 은닉 상태를 유지
 - stateful = False:
 - 은닉 상태를 영행렬로 초기화

5.3.2 Time RNN 계층 구현

- TimeRNN 계층의 forward():

```
def forward(self, xs): xs: T개 분량의 시계열 데이터를 하나로 모은 것
    Wx, Wh, b = self.params
    N, T, D = xs.shape
    D, H = Wx.shape

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

    if not self.stateful or self.h is None: # 은닉 상태 h => 처음 호출 시(self.h가 None인 경우), 또는 stateful이 False일 때 영행렬로 초기화
        self.h = np.zeros((N, H), dtype='f')

    for t in range(T):
        layer = RNN(*self.params) # RNN 계층 생성
        self.h = layer.forward(xs[:, t, :], self.h) # 인스턴스 변수 layers에 생성한 RNN계층을 추가하고, 그 계층은 시각 t일 때 은닉상태 h를 계산
        hs[:, t, :] = self.h # 계산한 은닉상태 h를 hs에 해당 인덱스(시각) 값으로 설정한다.
        self.layers.append(layer)

    return hs
```

- Time RNN 계층의 forward() 메서드가 불리면, 인스턴스 변수 h에는 마지막 RNN 계층의 은닉 상태가 저장됨.
- 다음번 forward() 매서드 호출 시
 - stateful = True: 먼저 저장된 h값 사용
 - stateful = False: h 영행렬로 초기화

5.3.2 Time RNN 계층 구현

- TimeRNN 계층의 backward():

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

    # dxs: 아래로 흘러보낼 기울기
    dxs = np.empty((N, T, D), dtype='f')
    dh = 0 # dh: 이전 시각의 은닉 상태 기울기
    grads = [0, 0, 0]
    for t in reversed(range(T)): # 순전파때와는 역으로 호출
        layer = self.layers[t]
        dx, dh = layer.backward(dhs[:, t, :] + dh) # 합산된 기울기
        dxs[:, t, :] = dx # dxs의 해당 인덱스에 저장

        for i, grad in enumerate(layer.grads):
            grads[i] += grad # 각 RNN 계층의 가중치 기울기를
                               # 합산해 멤버변수 self.grads에 덮어
                               # 씌운다.

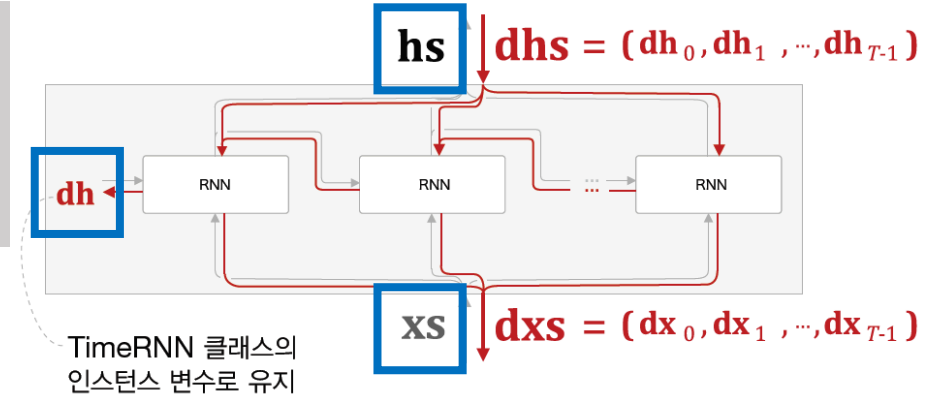
    for i, grad in enumerate(grads):
        self.grads[i][...] = grad
    self.dh = dh

    return dxs
```

Truncated BPTT를 사용하기 때문에 이 블록 이전 시각의 역전파는 필요하지 않지만, 이후 seq2seq(7장)에서 필요하기 때문에 저장해 놓음.

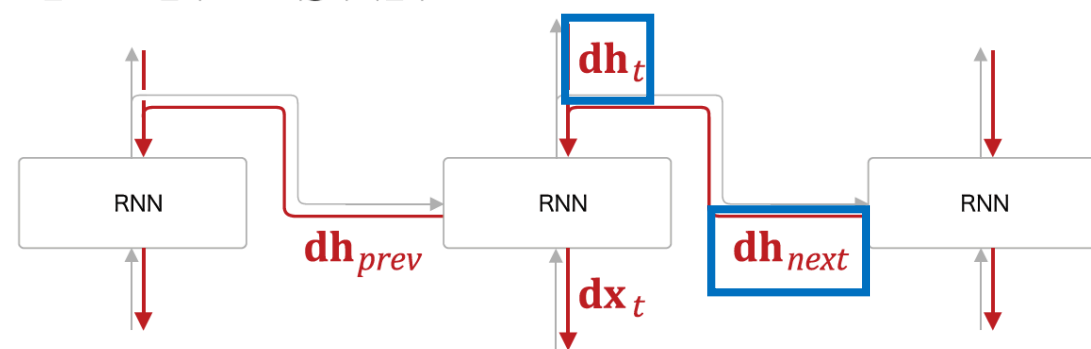
여러 개의 RNN 계층이 Time RNN을 구성하고 있으며, 그 RNN 계층들은 똑같은 가중치를 사용하고 있다. 따라서 Time RNN 계층의 최종 가중치 기울기는 각 RNN 계층의 가중치를 모두 더한 게 된다.

그림 5-23 Time RNN 계층의 역전파



순전파에서 출력이 2개로 분기 => 역전파에서는 각 기울기가 합산되어 전해진다. ($dh_t + dh_{next}$)

그림 5-24 t번째 RNN 계층의 역전파



5.4 시계열 데이터 처리 계층 구현

- RNN을 사용해 '언어 모델'을 구현해보자
 - 지금까지는 RNN 계층과 시계열 데이터를 한꺼번에 처리하는 Time RNN 계층을 구현
 - 5.4절에서는 시계열 데이터를 처리하는 계층 몇 개를 더 구현해보자.
 - RNN을 사용한 언어 모델은 영어로 RNN Language Model이므로 RNNLM이라고 지칭

5.4.1 RNNLM의 전체 그림

그림 5-25 RNNLM의 신경망(왼쪽이 펼치기 전, 오른쪽은 펼친 후)

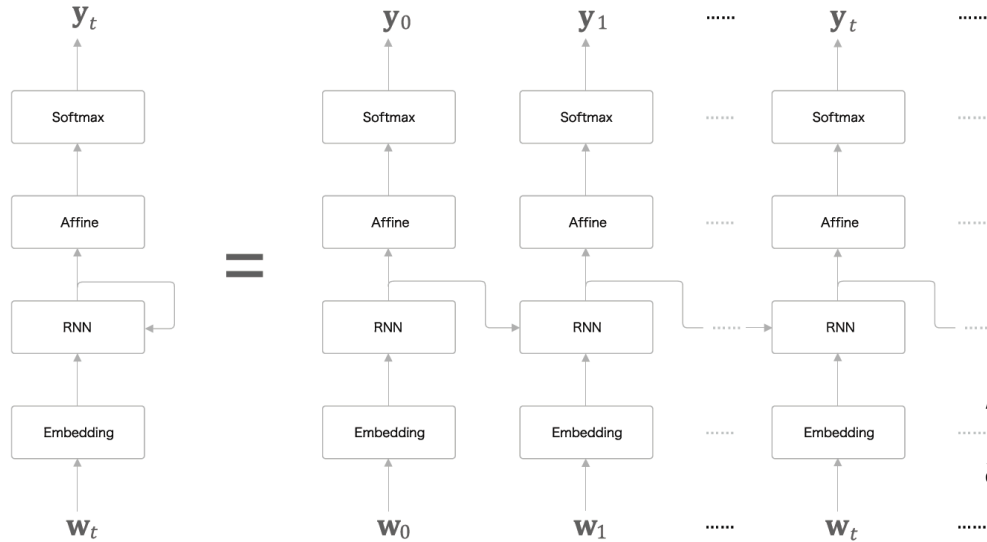
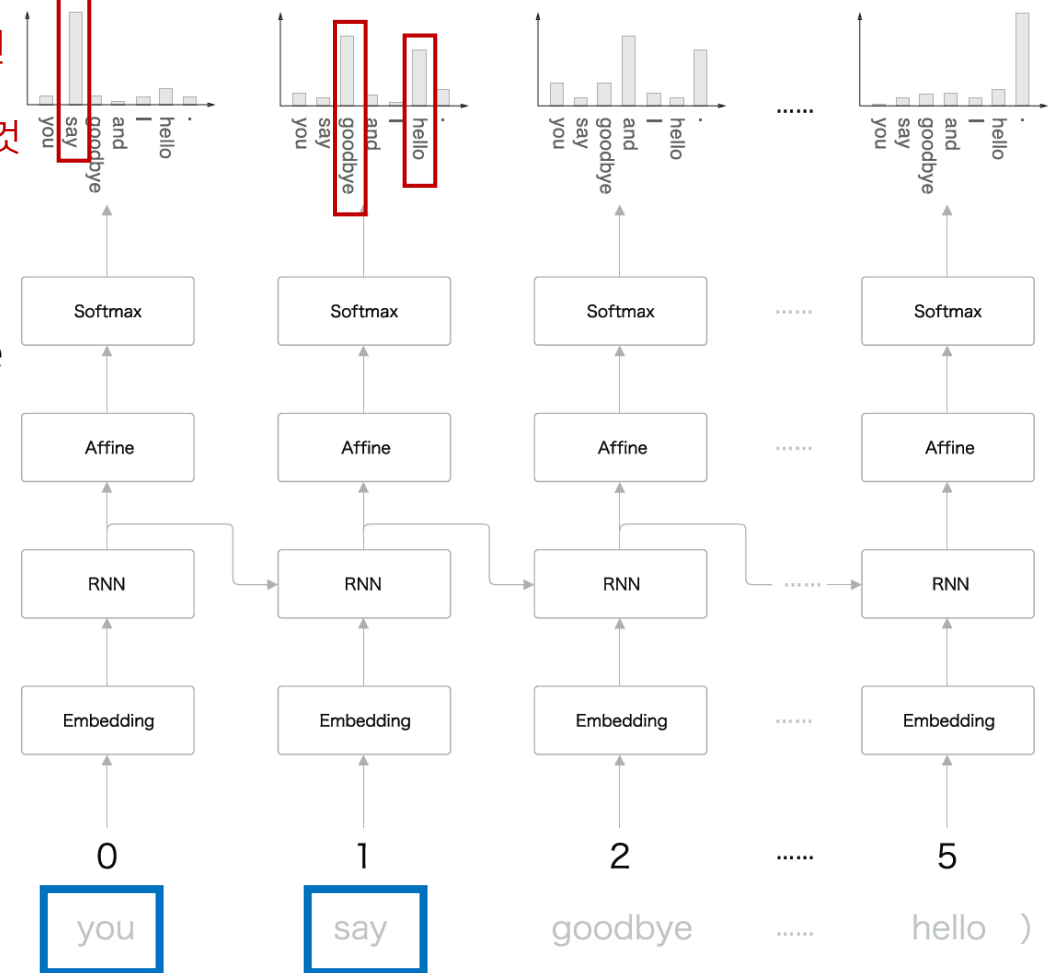


그림 5-26 샘플 말뭉치로 "you say goodbye and I say hello ."를 처리하는 RNNLM의 예

1) "you"가 입력된 경우 "say"의 확률 분포가 가장 높은 것을 확인

"You say goodbye and I say hello."



2) "say"가 입력되었을 때는 "goodbye"와 "hello"에서 높은 확률이 나옴 => "you say"라는 맥락을 기억!

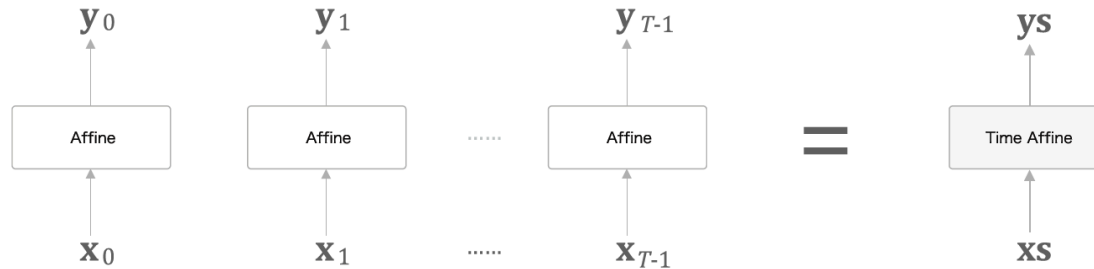
- **RNNLM**: 지금까지 입력된 단어를 '기억'하고, 그것을 바탕으로 다음에 출현할 단어를 예측
 - RNN계층이 과거에서 현재로 데이터를 계속 흘려 보내줌으로써 과거의 정보를 인코딩해 저장 (기억)

5.4.2 Time 계층 구현

- 시계열 데이터를 한꺼번에 처리하는 계층(Time Embedding, Time Affine, Time Softmax with Loss)을 구현

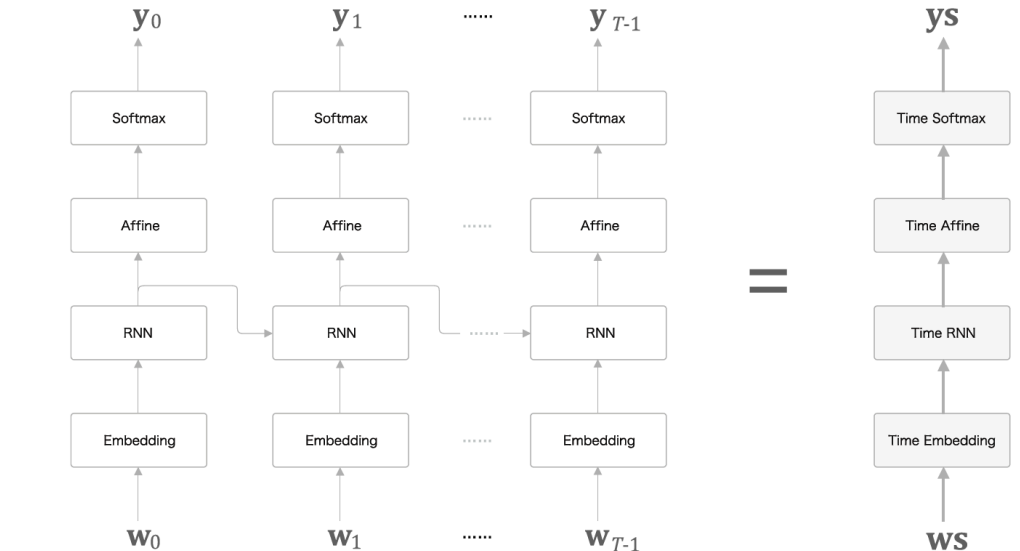
- Time Affine 계층:**

- T개의 Affine 계층을 준비해 각 시각의 데이터를 개별적으로 처리



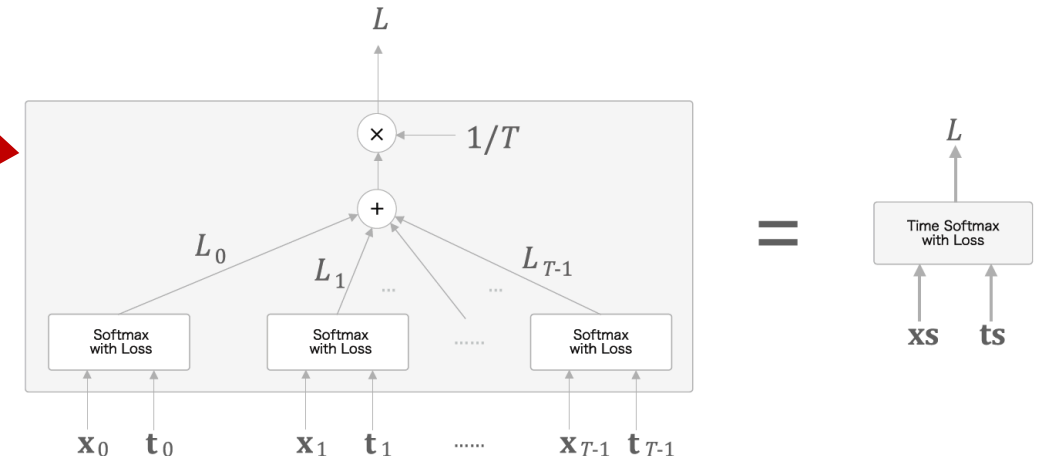
- Time Embedding 계층:**

- T개의 Embedding 계층을 준비해 각 계층이 각 시각의 데이터를 처리



- Time Softmax with Loss 계층:**

- x_n : 아래층으로부터 전해지는 '점수'
- t_n : 정답 레이블
- $L = \frac{1}{T}(L_0 + L_1 + \dots + L_{T-1})$
 - T개의 Softmax with Loss 계층이 각각의 손실을 산출,
 - 그 손실들을 합산해 평균한 값이 최종 손실이 된다.



5.5 RNNLM 학습과 평가:

1. RNNLM 구현

- SimpleRnnlm 클래스:

```
class SimpleRnnlm:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn
```

가중치 초기화

```
embed_W = (rn(V, D) / 100).astype('f')
rnn_Wx = (rn(D, H) / np.sqrt(D)).astype('f')
rnn_Wh = (rn(H, H) / np.sqrt(H)).astype('f')
rnn_b = np.zeros(H).astype('f')
affine_W = (rn(H, V) / np.sqrt(H)).astype('f')
affine_b = np.zeros(V).astype('f')
```

Xavier 초깃값을 이용:
이전 계층의 노드가 n 개라
면 표준편차가 $\frac{1}{\sqrt{n}}$ 인 분포를
초깃값으로 사용

계층 생성

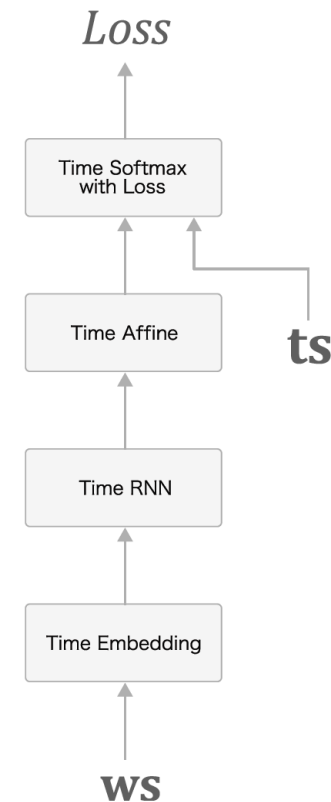
```
self.layers = [
    TimeEmbedding(embed_W),
    TimeRNN(rnn_Wx, rnn_Wh, rnn_b, stateful=True),
    TimeAffine(affine_W, affine_b)
]
```

```
self.loss_layer = TimeSoftmaxWithLoss()
self.rnn_layer = self.layers[1]
```

stateful = True: Time RNN 계층은
이전 시각의 은닉 상태를 계승할 수 있게
된다!

모든 가중치와 기울기를 리스트에 모은다.

```
self.params, self.grads = [], []
for layer in self.layers:
    self.params += layer.params
    self.grads += layer.grads
```



```
def forward(self, xs, ts):
    for layer in self.layers:
        xs = layer.forward(xs)
    loss = self.loss_layer.forward(xs, 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):
    self.rnn_layer.reset_state()
```

신경망 상태를 초기화하는 메서드

5.5.2 언어 모델의 평가

- 언어 모델은 주어진 과거 단어(정보)로부터 다음에 출현할 단어의 확률분포를 출력
- 퍼플렉시티(perplexity, 혼란도)**: 언어 모델의 예측 성능을 평가하는 척도
 - 간단히 말하면 '확률의 역수'
 - "you say goodbye and I say hello"



- 모델 1: "you"라는 단어를 주었을 때, 정답이 "say"라면 그 확률은 0.8, 퍼플렉시티는 $\frac{1}{0.8} = 1.25$
- 모델 2: 정답 "say"의 확률을 0.2라고 예측했으므로 퍼플렉시티는 $\frac{1}{0.2} = 5$
- 따라서 퍼플렉시티가 작을 수록 좋다!
- 그렇다면 1.25나 5라는 값은 어떻게 해석해야 할까? => 분기수로 해석할 수 있다.
- 분기수(number of branches)**: 다음에 취할 수 있는 선택사항의 수, 즉 다음에 출현할 수 있는 단어의 후보 수
 - 분기 수 = 1.25: 다음에 출현할 수 있는 단어의 후보를 1개 정도로 좁힘
 - 분기 수 = 5: 후보가 아직 5개나 된다는 의미

5.5.2 언어 모델의 평가

- 지금까지는 입력 데이터가 하나일 때의 퍼플렉시티
- 입력 데이터가 여러 개일 때는 다음과 같이 계산한다.

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

$$perplexity = e^L$$

- N: 데이터의 총 개수
- t_n : 원핫 벡터로 나타낸 정답 레이블
- t_{nk} : n개째 데이터의 k번째 값
- y_{nk} : 확률분포(Softmax의 출력)
- L: 신경망의 손실

5.5.3 RNNLM의 학습 코드

- PTB 데이터셋의 처음 1,000개 단어만 이용해 학습 수행

```
# 5.5.3 RNNLM의 학습 코드

# 하이퍼파라미터 설정
batch_size = 10
wordvec_size = 100
hidden_size = 100 # RNN의 은닉 상태 벡터의 원소 수
time_size = 5      # Truncated BPTT가 한 번에 펼치는 시간 크기
lr = 0.1
max_epoch = 100

# 학습 데이터 읽기(전체 중 1000개만)
corpus, word_to_id, id_to_word = ptb.load_data('train')
corpus_size = 1000
corpus = corpus[:corpus_size]
vocab_size = int(max(corpus) + 1)

xs = corpus[:-1] # 입력
ts = corpus[1:]  # 출력(정답 레이블)
data_size = len(xs)
print('말뭉치 크기: %d, 어휘 수: %d' % (corpus_size, vocab_size))

# 학습 시 사용하는 변수
max_iters = data_size // (batch_size * time_size)
time_idx = 0
total_loss = 0
loss_count = 0
ppl_list = []
```

5.5.3 RNNLM의 학습 코드

- PTB 데이터셋의 처음 1,000개 단어만 이용해 학습 수행

```
# 모델 생성
model = SimpleRnnlm(vocab_size, wordvec_size, hidden_size)
optimizer = SGD(lr)

1) { # 미니배치의 각 샘플의 읽기 시작 위치를 계산      각 미니배치에서 샘플을 읽기 시작하는 위치를 계산
      jump = (corpus_size - 1) // batch_size
      offsets = [i * jump for i in range(batch_size)]

      for epoch in range(max_epoch):      offsets: 데이터를 읽는 시작 위치 저장
          for iter in range(max_iters):
              # 미니배치 취득
              batch_x = np.empty((batch_size, time_size), dtype='i')
              batch_t = np.empty((batch_size, time_size), dtype='i')
              2) { for t in range(time_size):
                    for i, offset in enumerate(offsets):      미니배치 획득
                        batch_x[i, t] = xs[(offset + time_idx) % data_size]
                        batch_t[i, t] = ts[(offset + time_idx) % data_size]
                    time_idx += 1

              # 기울기를 구하여 매개변수 갱신
              loss = model.forward(batch_x, batch_t)
              model.backward()
              optimizer.update(model.params, model.grads)
              total_loss += loss
              loss_count += 1

              3) { # 에폭마다 퍼플렉시티 평가
                    ppl = np.exp(total_loss / loss_count)
                    print('| 에폭 %d | 퍼플렉시티 %.2f' % (epoch+1, ppl))
                    ppl_list.append(float(ppl))
                    total_loss, loss_count = 0, 0
                    에폭마다 퍼플렉시티 평가
```

- 데이터 제공 방법

- Truncated BPTT 방식 사용

- 데이터는 순차적으로 주고, 각각의 미니배치에서 데이터를 읽는 시작 위치를 조정
- 1)에서는 각 미니배치가 데이터를 읽기 시작하는 위치를 계산해 offsets에 저장

- 2)에서 데이터를 순차적으로 읽음

- batch_x와 batch_t 준비
- time_idx를 1씩 늘리면서 말뭉치에서 time_idx 위치의 데이터를 얻는다
- 1)에서 계산한 offsets를 이용해 각 미니배치에 오프셋을 추가
- 말뭉치를 읽는 위치가 말뭉치 크기 넘어설 경우, 말뭉치의 크기로 나눈 나머지를 인덱스로 사용해 처음으로 돌아간다.

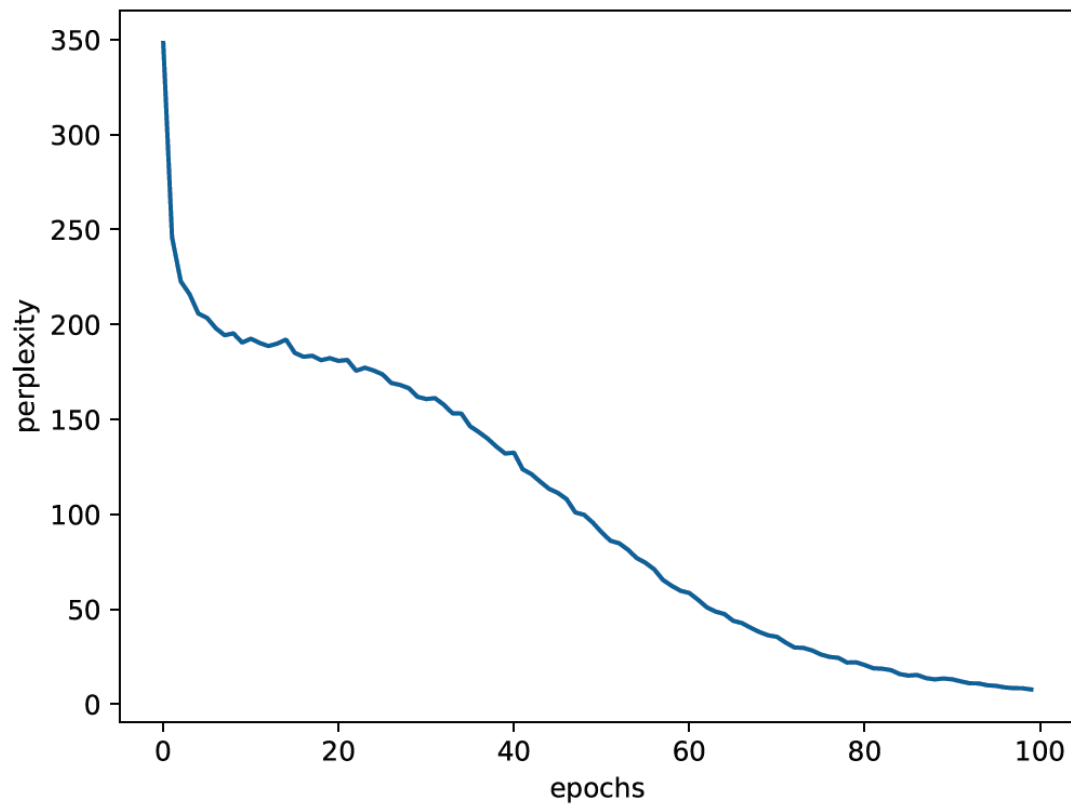
- 퍼플렉시티 계산

- 3)에서 에폭마다 손실의 평균을 구해 퍼플렉시티 구함.

5.5.3 RNNLM의 학습 코드

- 학습을 진행할수록 퍼플렉서티가 순조롭게 낮아짐
- 다만 아직 크기가 작은 말뭉치로 실험
 - 현재 모델로는 아직 큰 말뭉치에 대응 불가
 - 다음 장에서 개선

그림 5-33 퍼플렉서티 추이



5.5.4 RNNLM의 Trainer 클래스

- RNNLM의 학습을 수행해주는 RNNTrainer 클래스를 제공
 - 앞의 학습 코드를 클래스로 구현

```
...  
# 모델 생성  
model = SimpleRnnlm(vocab_size, wordvec_size, hidden_size)  
optimizer = SGD(lr)  
trainer = RnnlmTrainer(model, optimizer)  
  
trainer.fit(xs, ts, max_epoch, batch_size, time_size)
```

- RNNTrainer 클래스에 model과 optimizer를 주어 초기화 한 다음 fit()을 호출해 학습 수행
 - 그 내부에서는 앞 절에서 수행한 일련의 작업이 진행
 - 1) 미니배치를 '순차적'으로 만들어
 - 2) 모델의 순전파와 역전파를 호출하고
 - 3) 옵티마이저로 가중치를 갱신하고
 - 4) 퍼플렉시티를 구한다.

5.6 정리

- 순환 신경망(RNN) 계층의 구조를 살펴보고, RNN 계층과 Time RNN 계층을 구현
- RNN을 이용해 언어 모델을 만듦.
 - RNN을 이용한 신경망 구성을 통해 아무리 긴 시계열 데이터라도 중요 정보를 RNN의 은닉 상태에 기억
 - 그러나 실제 문제에서는 아직 잘 학습하지 못함
- 다음 장에서 RNN의 문제점과 RNN을 대체하는 새로운 계층들(LSTM, GRU)을 살펴봄