Data - Structure - Study(DSS)

Winter Vacation Soo-lab Study

TEAM Kai.saekies

발표자 : 김영웅 양근제

2020.01.03 (FRI)

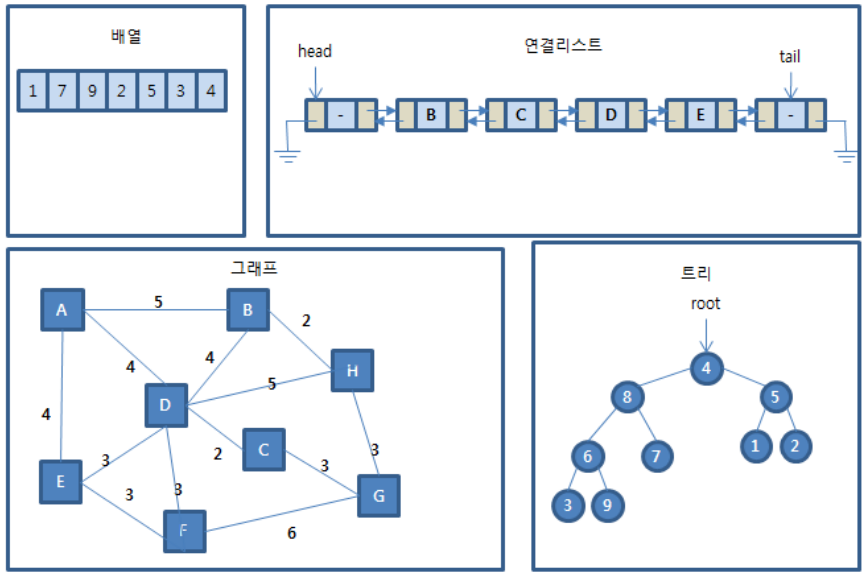
Index

Data Structure

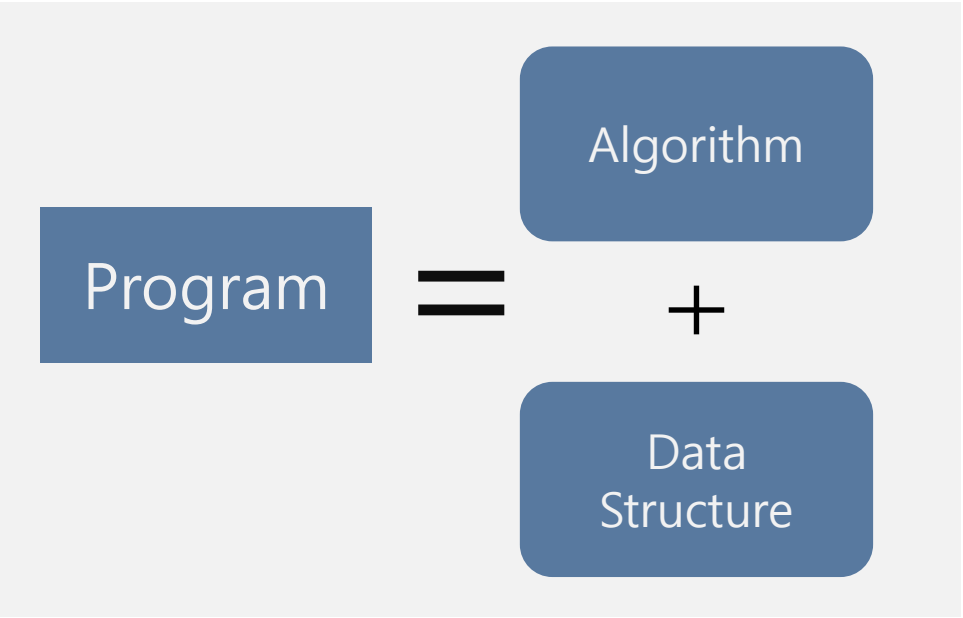
컴퓨터 과학에서 효율적인 접근 및 수정을 가능케 하는 자료의 조직, 관리, 저장을 의미.

↓

데이터 값의 모임, 데이터 간의 관계, 그리고 데이터에 적용할 수 있는 함수나 명령을 의미

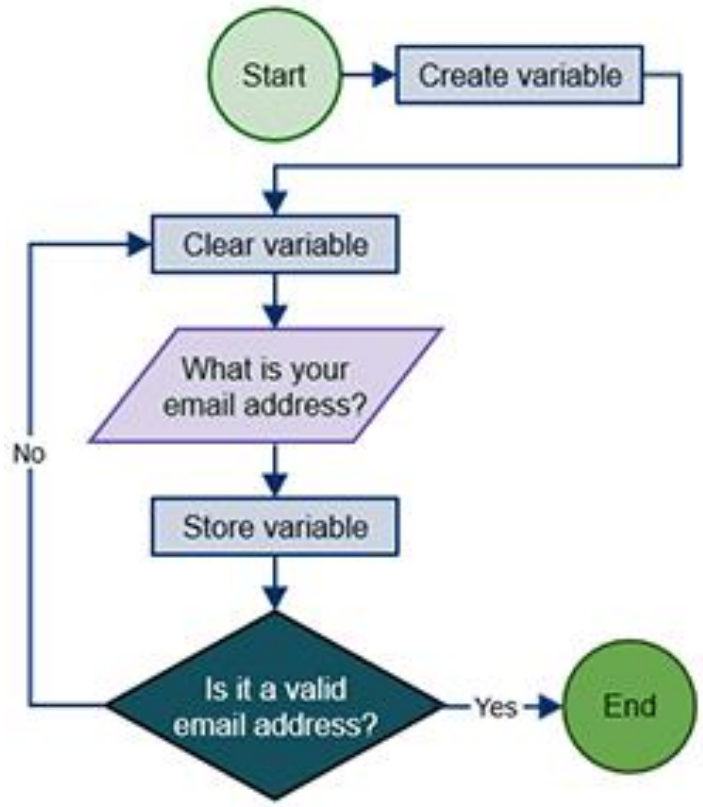


다양한 형태의 자료구조:array, linked list, graph, tree



Algorithm

- 어떤 문제가 주어졌을 때, 문제를 해결하기 위한 단계적 절차



Email 발송 알고리즘

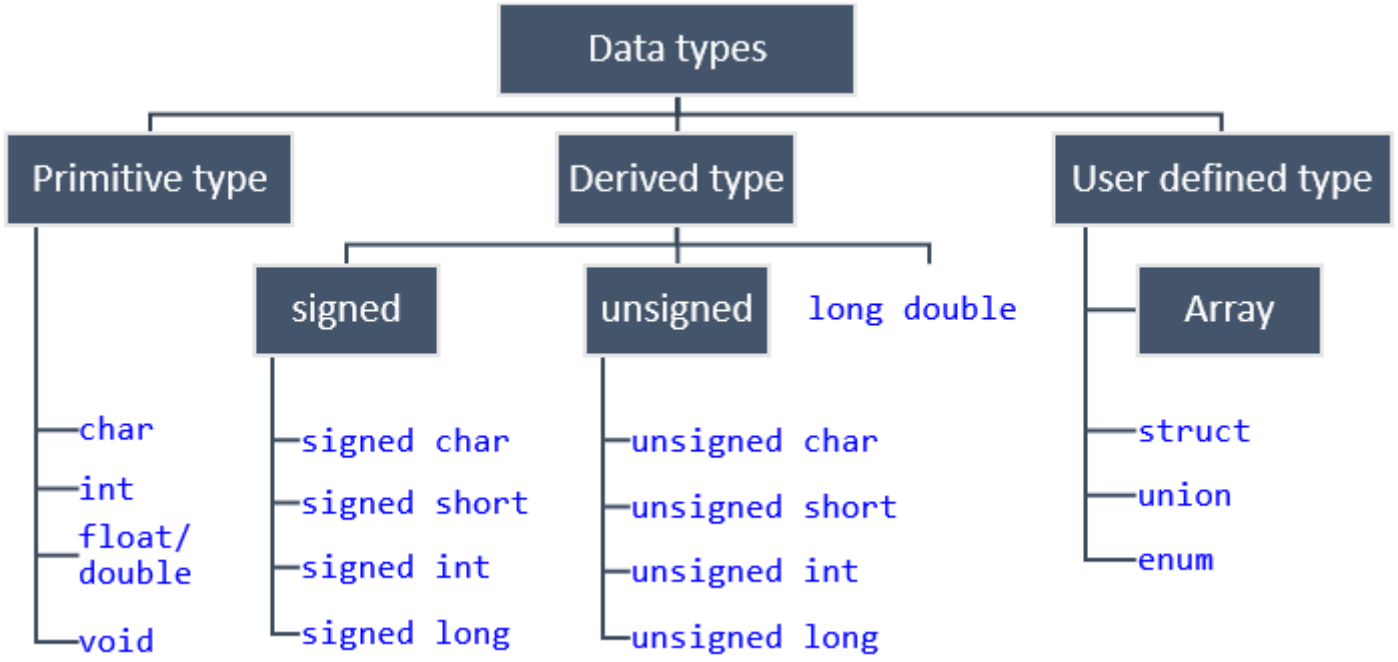
알고리즘을 기술하는 4가지 방법

- 자연어
- Flowchart
- Pseudo – code
- Programming language

Data Type(자료형)

Data Type

- 여러 종류의 데이터를 식별하는 분류
- 넓은 의미로 해당 자료형에 대한 가능한 값, 해당 자료형에서 수행을 마칠 수 있는 명령들, 데이터의 의미, 해당 자료형의 값을 저장하는 방식을 의미

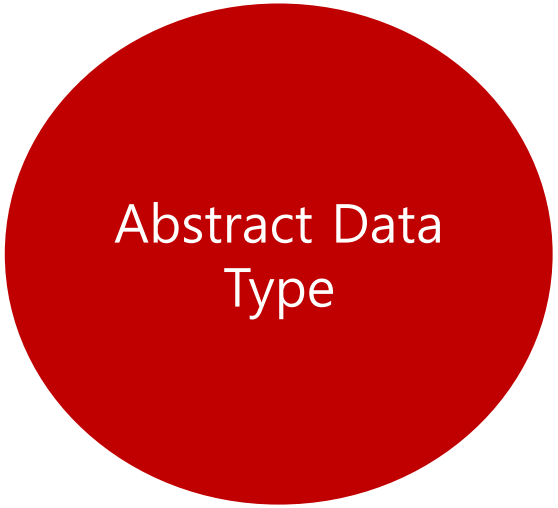


다양한 자료형(C)

Data Type(자료형)

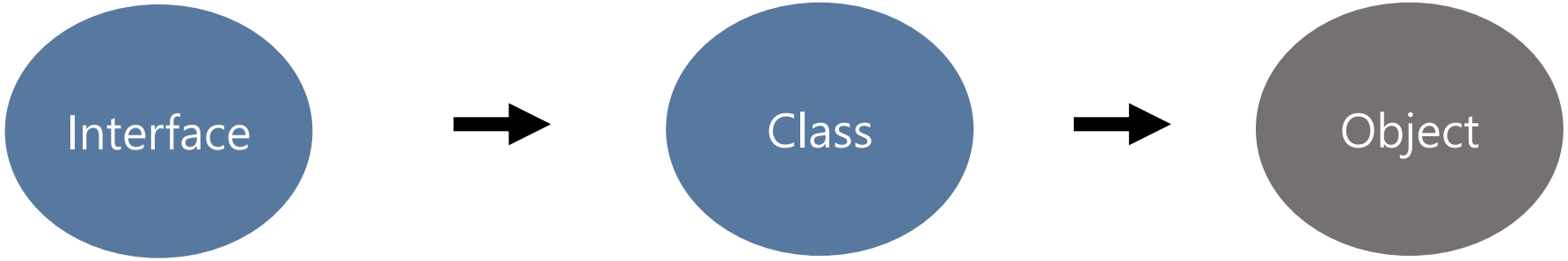
Abstract Data Type(ADT) – 추상 자료형

- 프로그램의 대상이 되는 사물 또는 대상을 추상화 하여 정의



ADT의 장점

- 설계와 구현의 분리
- Data를 사용할 때 모든 정보를 다 이해하지 않아도 된다.
  정보은닉 기법(information hiding)



Java의 추상 자료형

알고리즘 성능분석

- **Time Complexity(시간 복잡도)**
  알고리즘에 사용되는 연산 횟수 ≠ 실행 시간

```
for(int i = 0; i < n ; i++)
{
    printf("hello!");
}
```

n

```
for(int i = 0; i < n ; i++)
{
    for(int j = 0; j < n;j++)
    {
        printf("hello");
    }
}
```

n^2

- **Space Complexity(시간 복잡도)**
  알고리즘에 사용되는 메모리의 총량

```
int n[2] = {1,1};
```

n

```
int n[2][2] = {{1,1},{1,1}};
```

n^2

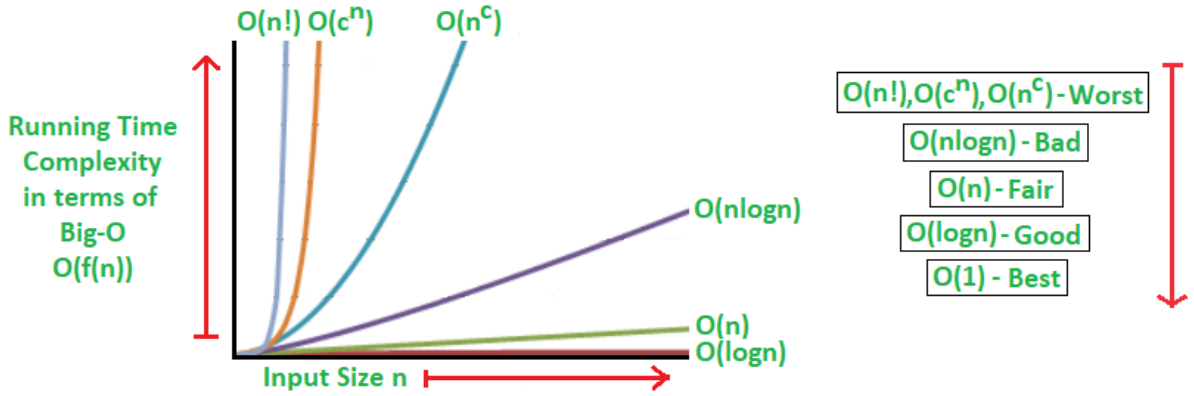
알고리즘 성능분석

시간 복잡도 표현 방식

- **Big – o notation** – 상한 표기
- Big – omega notation – 하한 표기
- Big – theta notation – 상한과 하한 표기

빅오 표기법 : 두 함수 f(n)과 g(n)이 주어졌을 때 모든 n>n0에 대하여 |f(n)| <= c*|g(n)|을 만족하는 2개의 상수 c와 n0가 존재하면 f(n)=O(g(n))이다.

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$

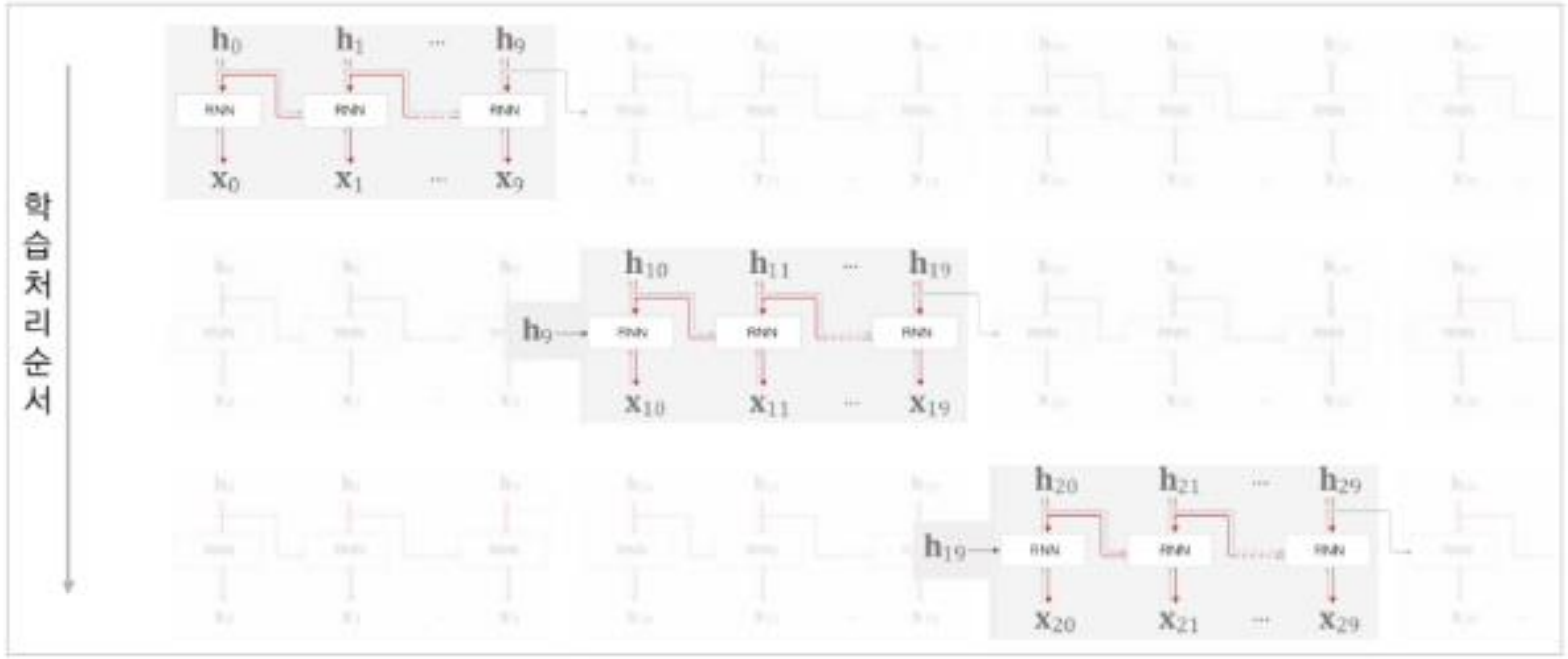


알고리즘 성능분석

Big - O	Example
O(1)	단순 출력문 , push, pop
O(log n)	Tree
O(n log n)	병합 정렬
$O(n^2)$	Nested for, Bubble sort
$O(2^n)$	피보나치 수열
O(n!)	n순열

RNN의 Backpropagation

- Truncated BPTT (3)



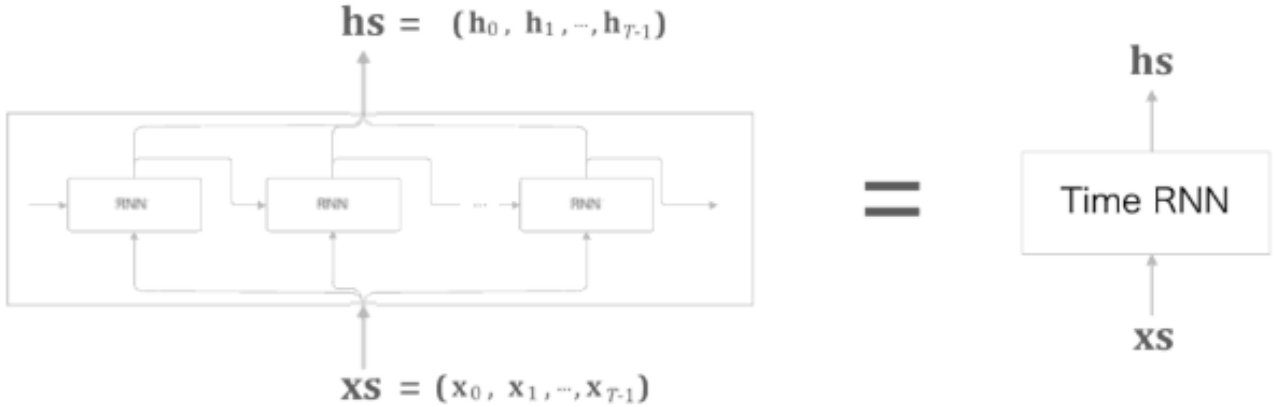
위의 그림처럼 Truncated BPTT에서는 데이터를 순서대로 입력해 학습한다. 이런 식으로 순전파의 연결을 유지하면서 블록 단위로 오차역전파법을 적용할 수 있다.

RNN 구현

- RNN 계층

지금부터 구현해야 할 것은 결국 가로 방향으로 성장한 신경망이다. 그리고 Truncated BPTT 방식의 학습을 따른다면, 가로 크기가 일정한 일련의 신경망을 만들면 된다. 우리가 다룰 신경망은 길이가 T인 시계열 데이터를 입력으로 받고, (T는 임의의 값), 각 시각의 Hidden State (은닉 상태)를 T개 출력한다.

Time RNN 계층 : 순환 구조를 펼친 후의 계층들을 하나의 계층으로 간주한다



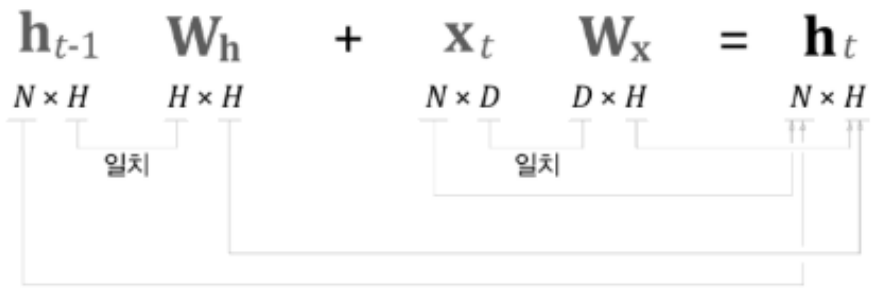
먼저 RNN의 한 단계를 처리하는 클래스를 RNN이란 이름으로 구현한다. 그리고 이 RNN 클래스를 이용해 T개 단계의 처리를 한꺼번에 수행하는 계층을 TimeRNN이란 이름의 클래스로 완성시킨다.

RNN 구현

- RNN 계층 구현

$$h_t = \tanh(h_{t-1}W_h + X_tW_x + b)$$

RNN의 순전파는 위의 식과 같다. 여기에서 우리는 데이터를 미니배치로 모아 처리한다. 행렬을 계산할 때는 행렬의 '형상 확인'이 중요하므로 우리가 계산할 형상 확인을 해보자. (미니배치 크기가 N, 입력벡터의 차원수가 D, 은닉 상태 벡터의 차원수가 H)



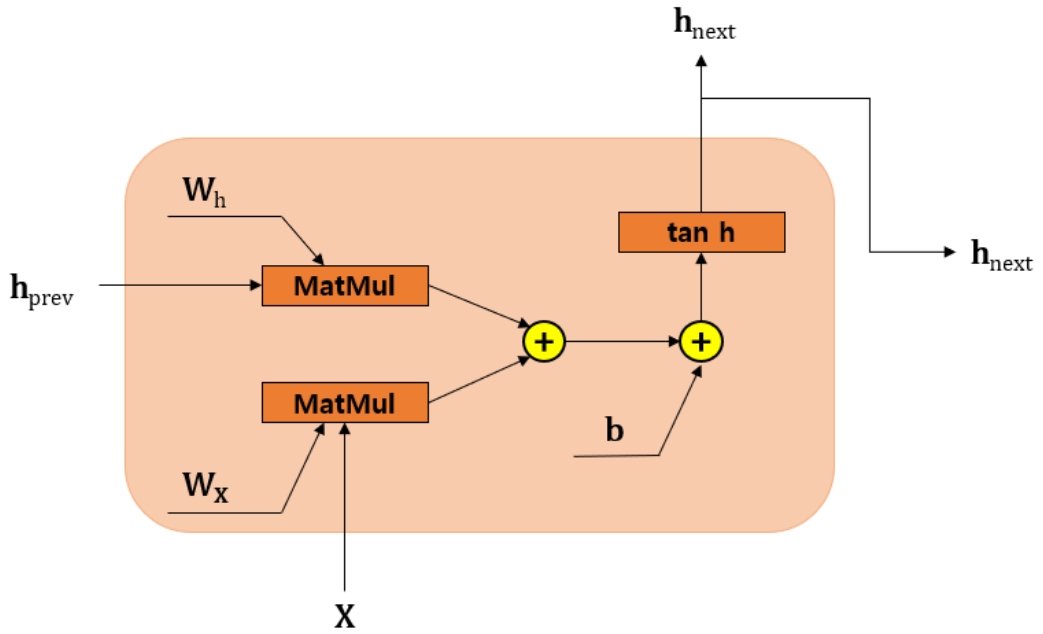
이상을 바탕으로 RNN 클래스의 초기화와 순전파, 역전파를 구현해보자.

RNN 구현

- RNN 계층의 순전파

```python
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
        h = np.matmul(x, Wx) + np.matmul(h_prev, Wh) + b
        h = np.tanh(h)
        self.cache = (x, h_prev, h)
        return h
```

RNN 초기화 및 순전파



RNN 계층의 계산 그래프

- __init__()

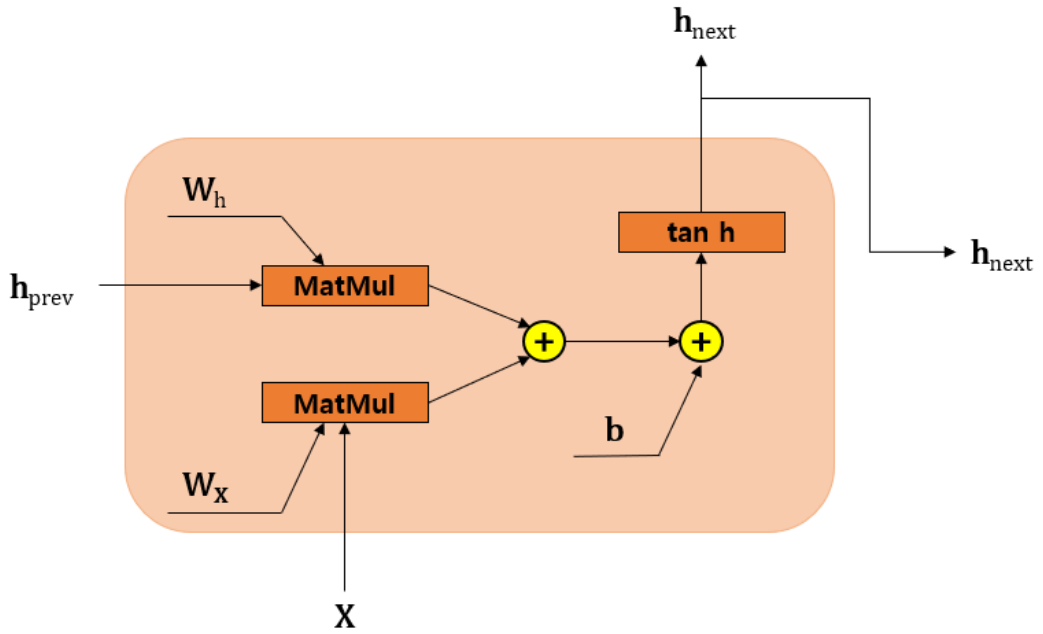
RNN의 초기화 메서드는 가중치 2개와 편향 1개를 인수로 받는다. 여기서 인수로 받은 매개변수를 인스턴스 변수 params에 리스트로 저장한다. 그리고 각 매개변수에 대응하는 형태로 기울기를 초기화한 후 grads에 저장한다. 마지막으로 역전파 계산 시 사용하는 중간 데이터를 담을 cache를 None로 초기화한다.

RNN 구현

- RNN 계층의 순전파

```python
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
        h = np.matmul(x, Wx) + np.matmul(h_prev, Wh) + b
        h = np.tanh(h)
        self.cache = (x, h_prev, h)
        return h
```

RNN 초기화 및 순전파



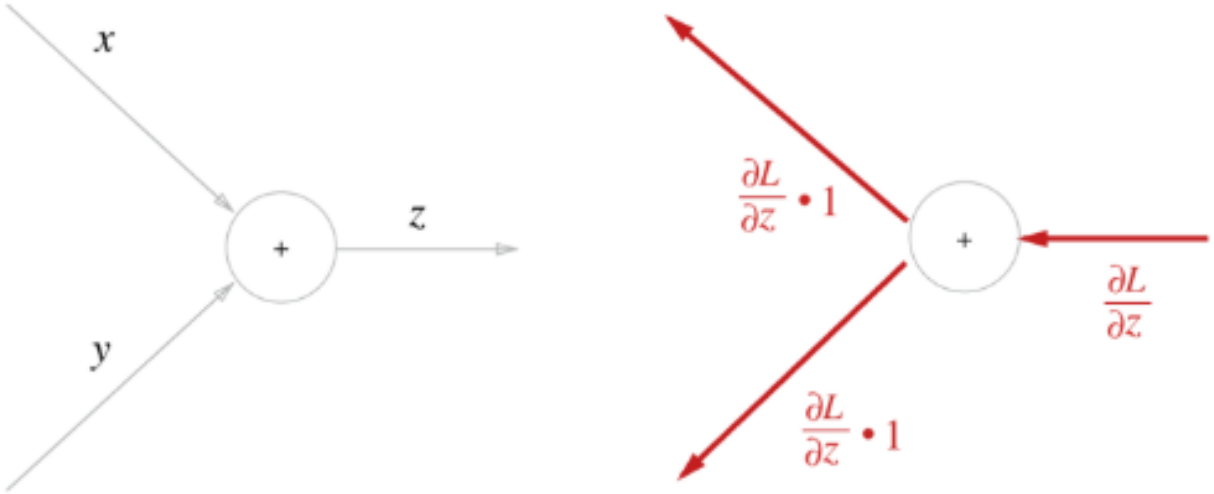
RNN 계층의 계산 그래프

- forward()

순전파인 forward 메서드는 입력 x와 이전 층의 입력 h_prev를 인수로 받는다.
그 다음은 RNN 계층의 계산 그래프 상의 과정을 코드로 옮긴 것 뿐이다.

역전파 복습

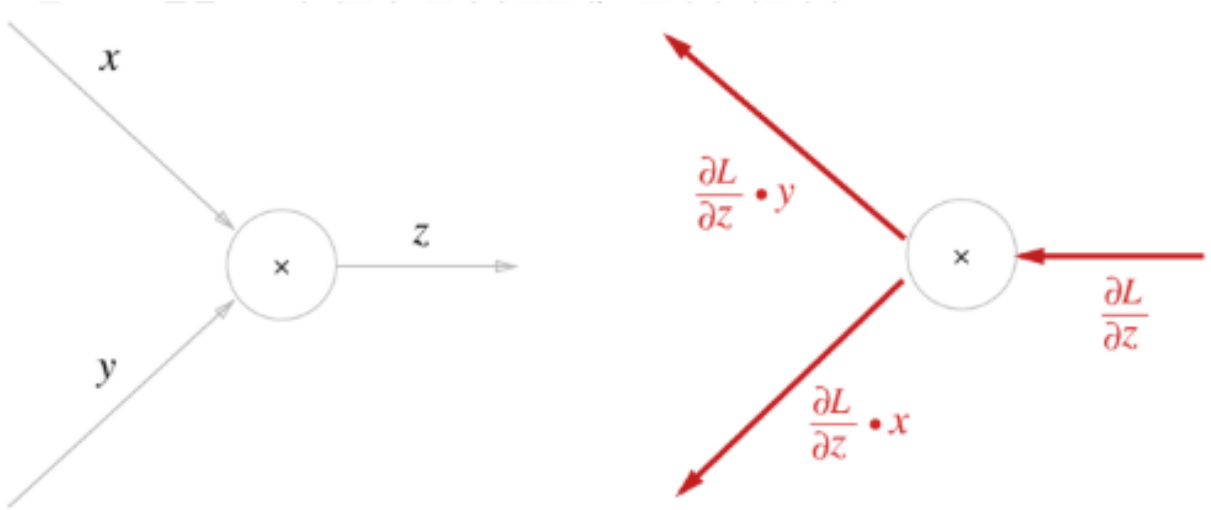
- 덧셈 노드의 역전파



덧셈노드의 역전파는 상류에서 전해진 미분을 그대로 흘려보낸다.
z = x + y라는 식이 있을때 역전파를 생각해보면 $\frac{\partial z}{\partial x}$ = 1, $\frac{\partial z}{\partial y}$ = 1이 된다. => 덧셈 노드는 입력값을 그대로 흘린다.

역전파 복습

- 곱셈 노드의 역전파



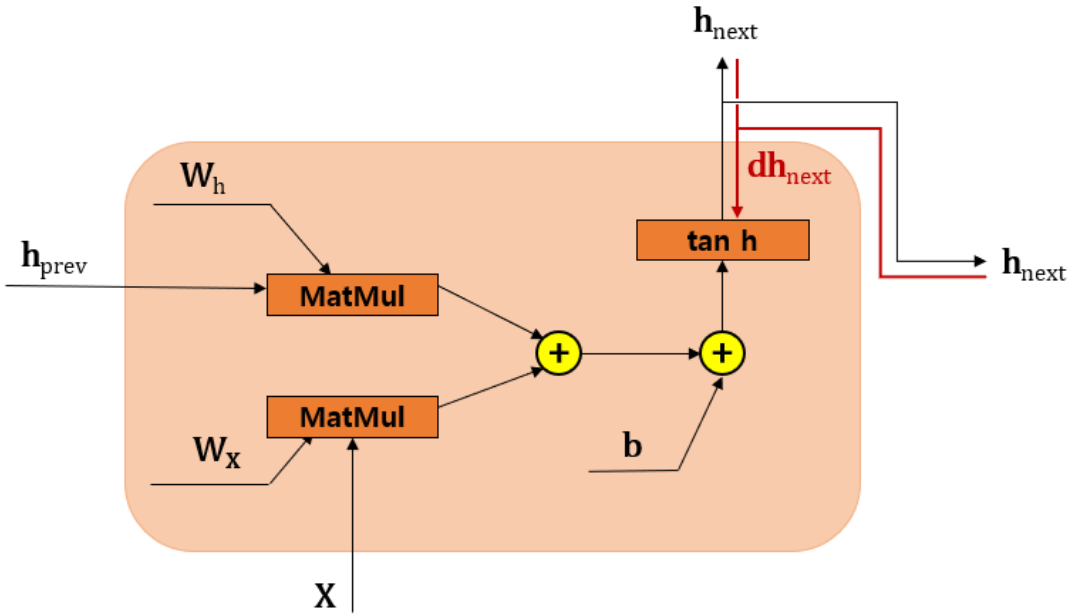
곱셈 노드 역전파는 상류의 값에 순전파 때의 입력 신호들을 '서로 바꾼 값'을 곱해서 하류로 보낸다.
z = xy 라는 식의 역전파를 생각해보면 $\frac{\partial z}{\partial x}$ = y, $\frac{\partial z}{\partial y}$ = x이 된다. 그러므로, 상류에서 들어온 신호에 순전파 때의 입력 신호들을 서로 바꾼 값을 곱해서 하류로 흘려보내면 된다.

RNN 구현

- RNN 계층의 역전파 - (1) dh_{next}

```
class RNN:
    def backward(self, dh_next):
        Wx, Wh, b = self.params
        x,  h_prev, h = self.cache
```

RNN 역전파



RNN 계층의 계산 그래프 (역전파 포함)

- backward()

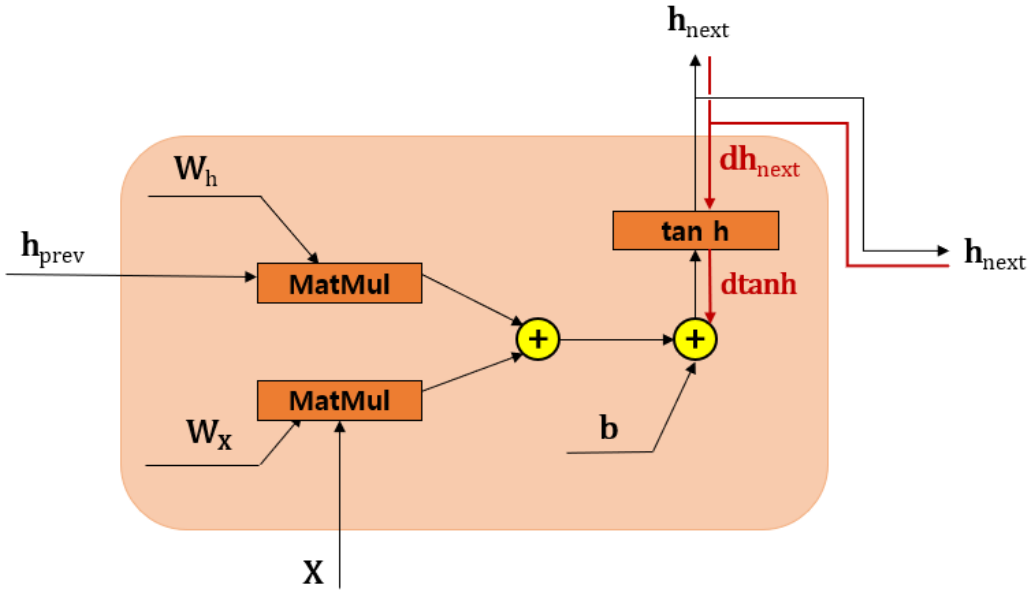
backward()는 다음 층의 역전파인 dh_next를 인수로 받는다.
그리고 __init__()에서 역전파 계산을 위해 미리 저장해놓은 cache 변수로부터 x, h_prev, h를 저장한다.

RNN 구현

- RNN 계층의 역전파 - (2) dtanh

```
class RNN:
    def backward(self, dh_next):
        Wx, Wh, b = self.params
        x,  h_prev, h = self.cache

        dtanh = dh_next * (1 - h ** 2)
```

RNN 역전파



RNN 계층의 계산 그래프 (역전파 포함)

- backward()

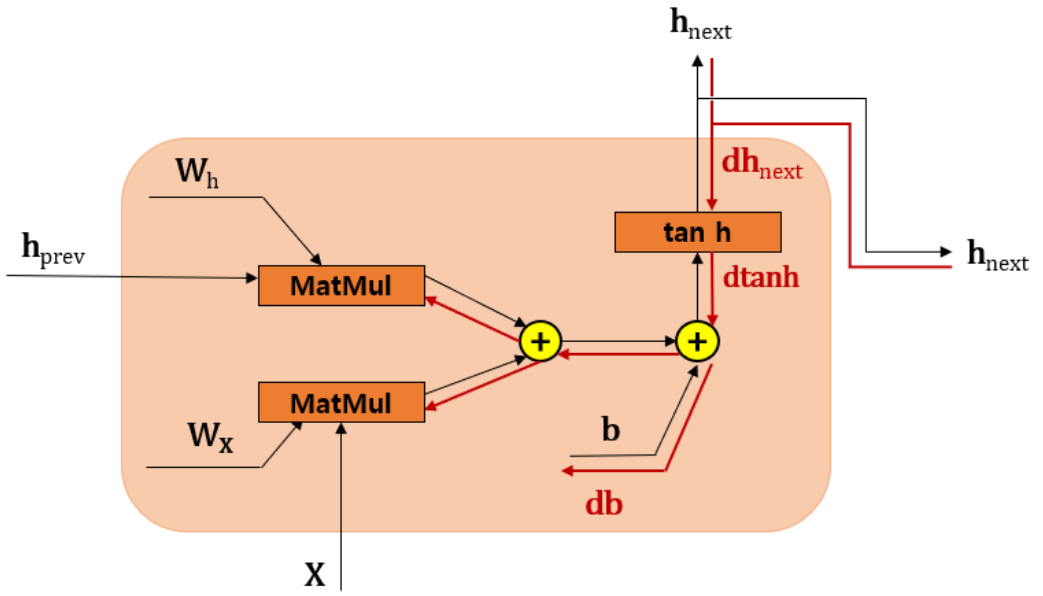
앞에서 언급된 tanh의 미분 계산에 따라 dtanh를 계산한다.

RNN 구현

- RNN 계층의 역전파 - (3) 덧셈 노드

```
class RNN:
    def backward(self, dh_next):
        Wx, Wh, b = self.params
        x,  h_prev, h = self.cache

        dtanh = dh_next * (1 - h ** 2)
        db = np.sum(dtanh, axis = 0)
```

RNN 역전파



RNN 계층의 계산 그래프 (역전파 포함)

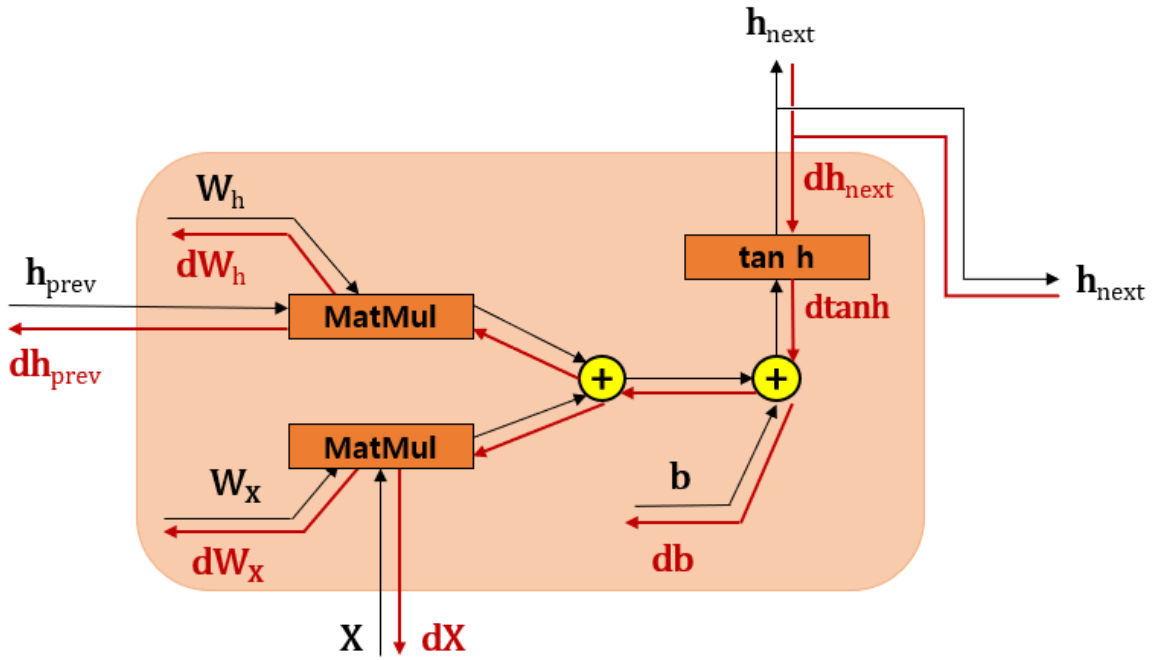
- backward()

계산 그래프 상에서 덧셈 노드의 경우 역전파를 그대로 흘려보낸다. (앞에서 언급)
그러므로 db를 제외한 역전파는 dtanh를 그대로 흘려보내주면 되므로 따로 계산하지 않는다.
여기서는 미니배치 단위 학습을 고려해서 만든 코드이므로db는 NxH의 형상이라고 고려하여 (N은 미니배치) 편향의 역전파는 데이터를 단위로 한 축인 axis=0의 총합을 구한다

RNN 구현

- RNN 계층의 역전파 - (4) 곱셈 노드

```
def backward(self, dh_next):
    Wx, Wh, b = self.params
    x,  h_prev, h = self.cache

    dtanh = dh_next * (1 - h ** 2)
    db = np.sum(dtanh, axis = 0)
    dWh = np.matmul(h_prev.T, dtanh)
    dh_prev = np.matmul(dtanh, Wh.T)
    dWx = np.matmul(x.T, dtanh)
    dx = np.matmul(dtanh, Wx.T)

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

    return dx, dh_prev
```



RNN 계층의 계산 그래프 (역전파 포함)

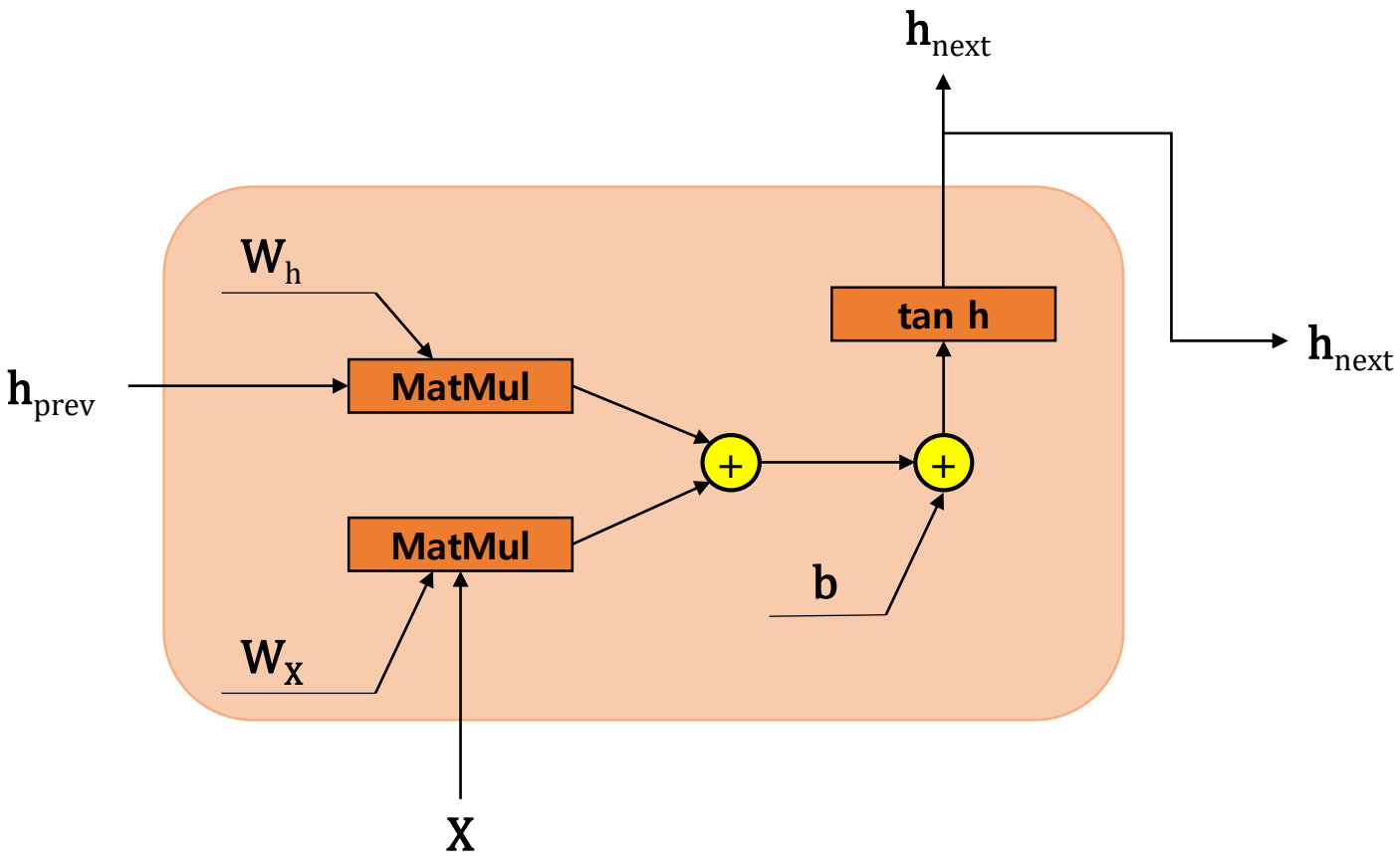
- backward()

계산 그래프 상에서의 곱셈 노드의 경우 상류에서 들어온 값에 순전파 때의 입력 신호들을 '서로 바꾼 값'을 곱하면 된다.
해당 법칙에 따라 나머지 모든 역전파를 계산한다.

Review

Recurrent Neural Network (RNN)

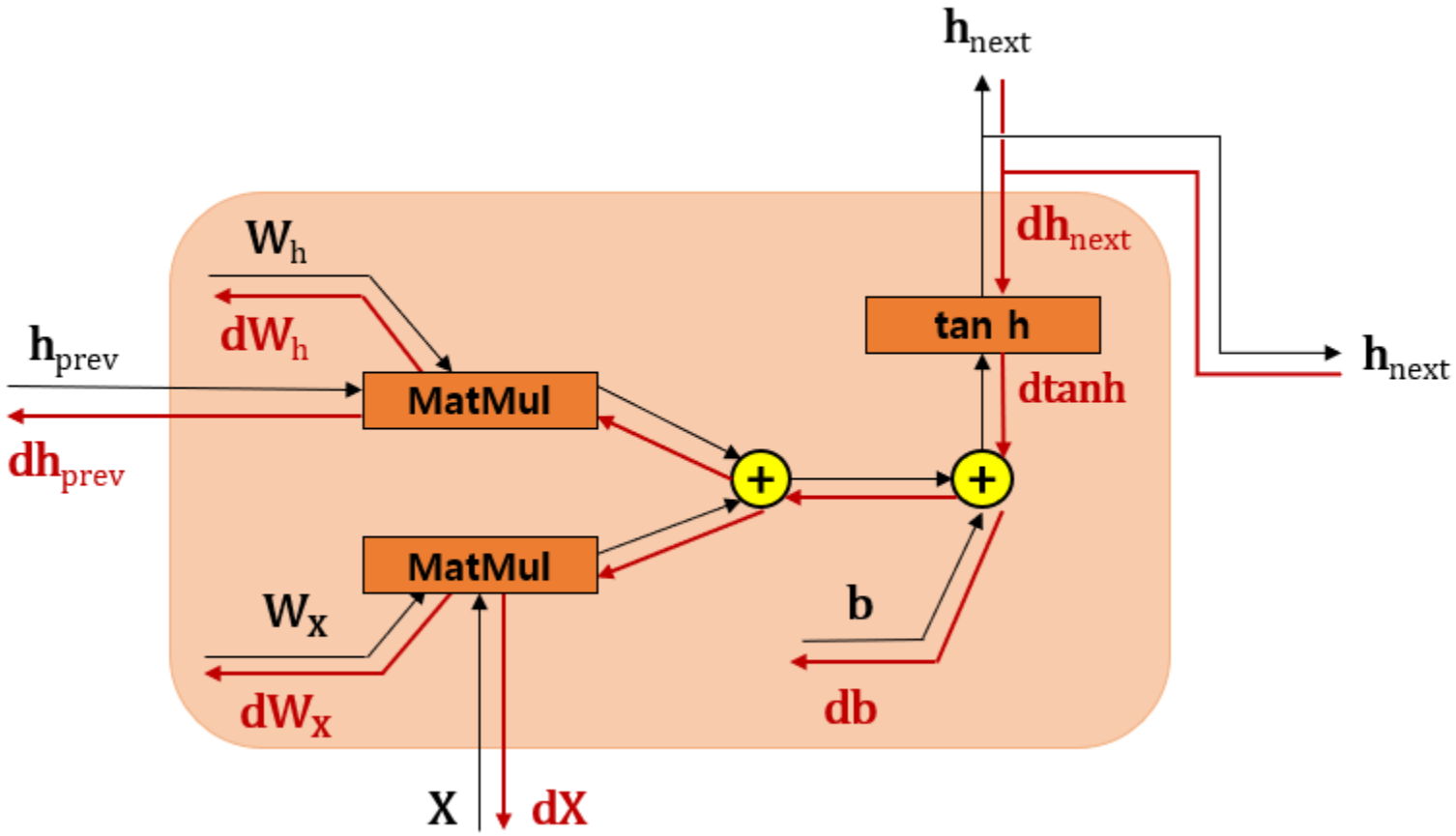
forward (순전파)



Review

Recurrent Neural Network (RNN)

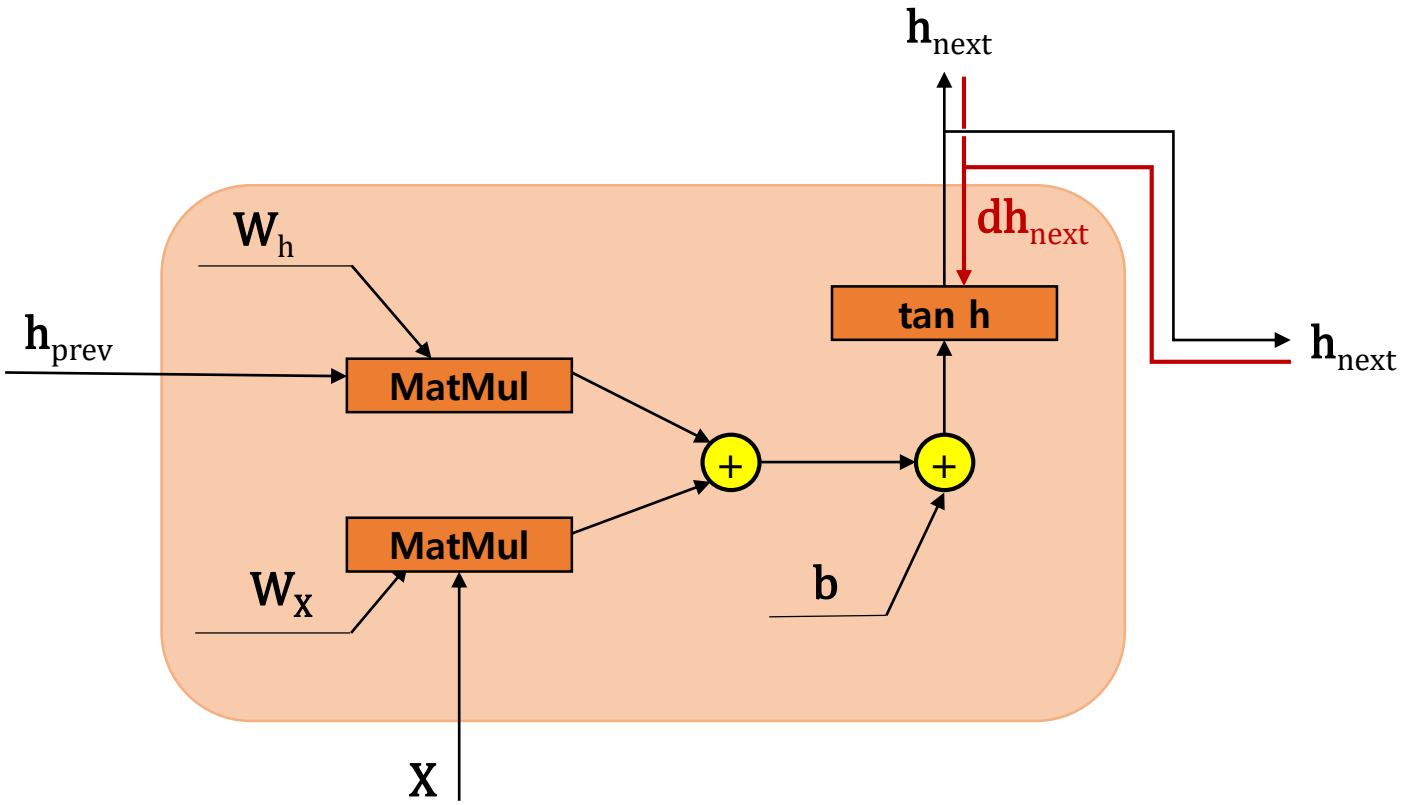
backward (역전파)



Review

Recurrent Neural Network (RNN)

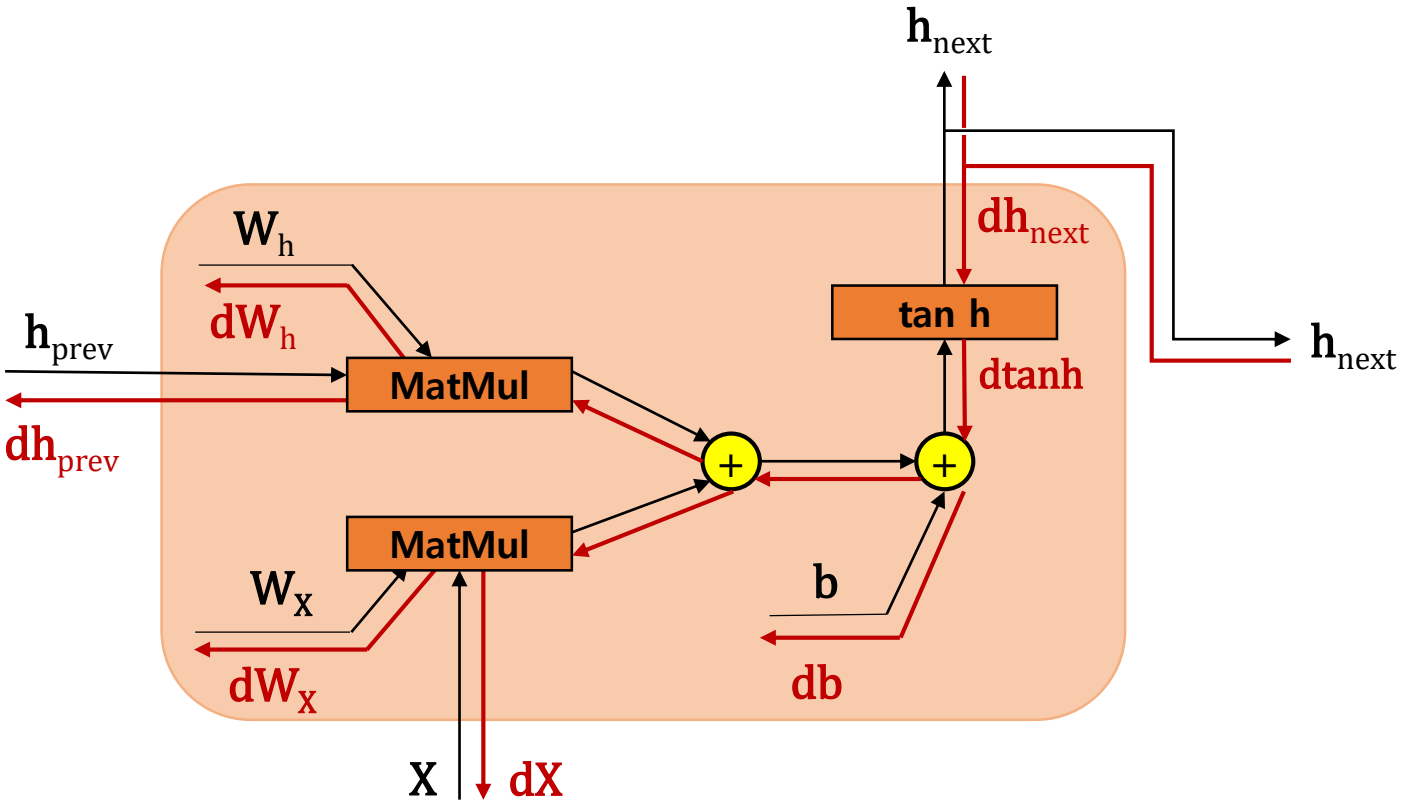
backward (역전파) - (1) dh_next



Review

Recurrent Neural Network (RNN)

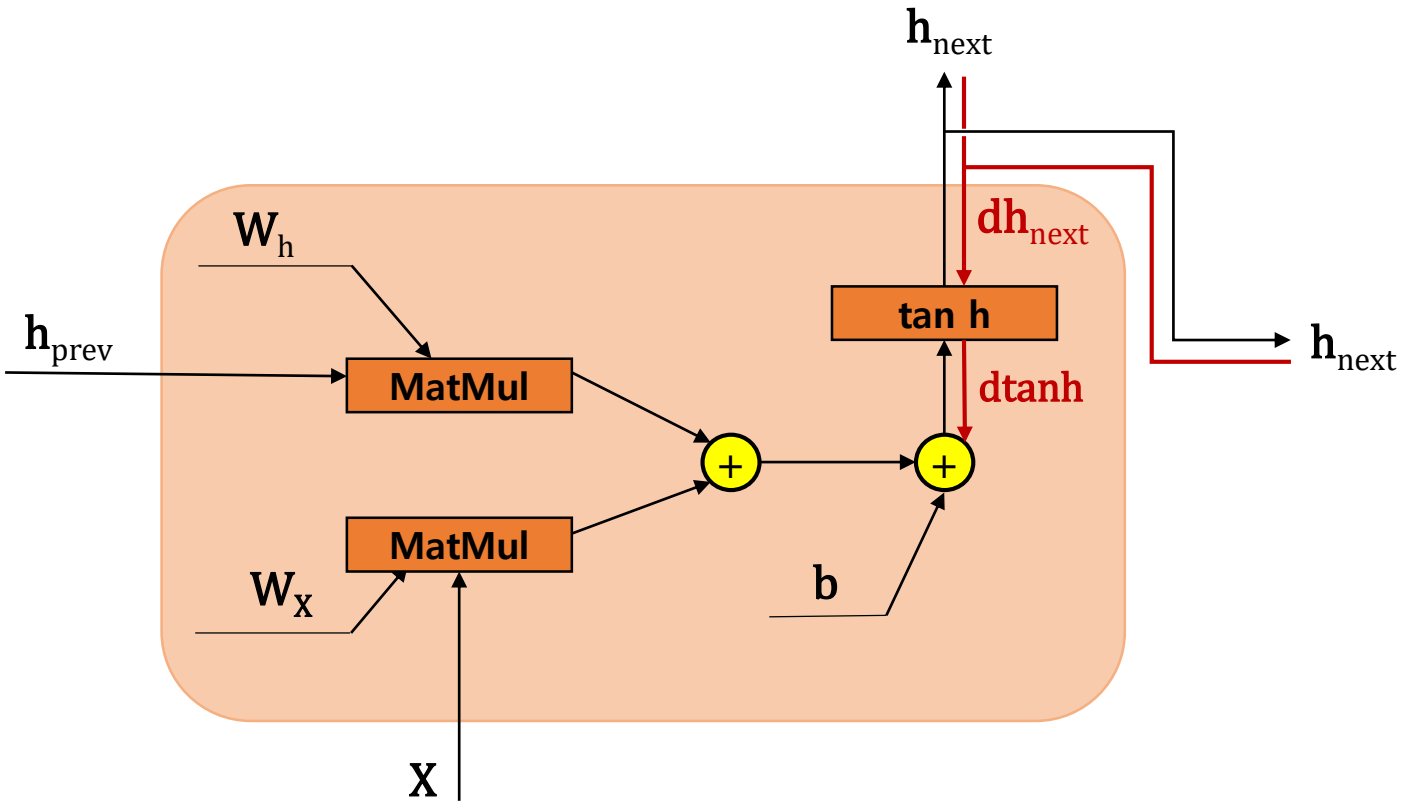
backward (역전파) – (4) 곱셈 노드



Review

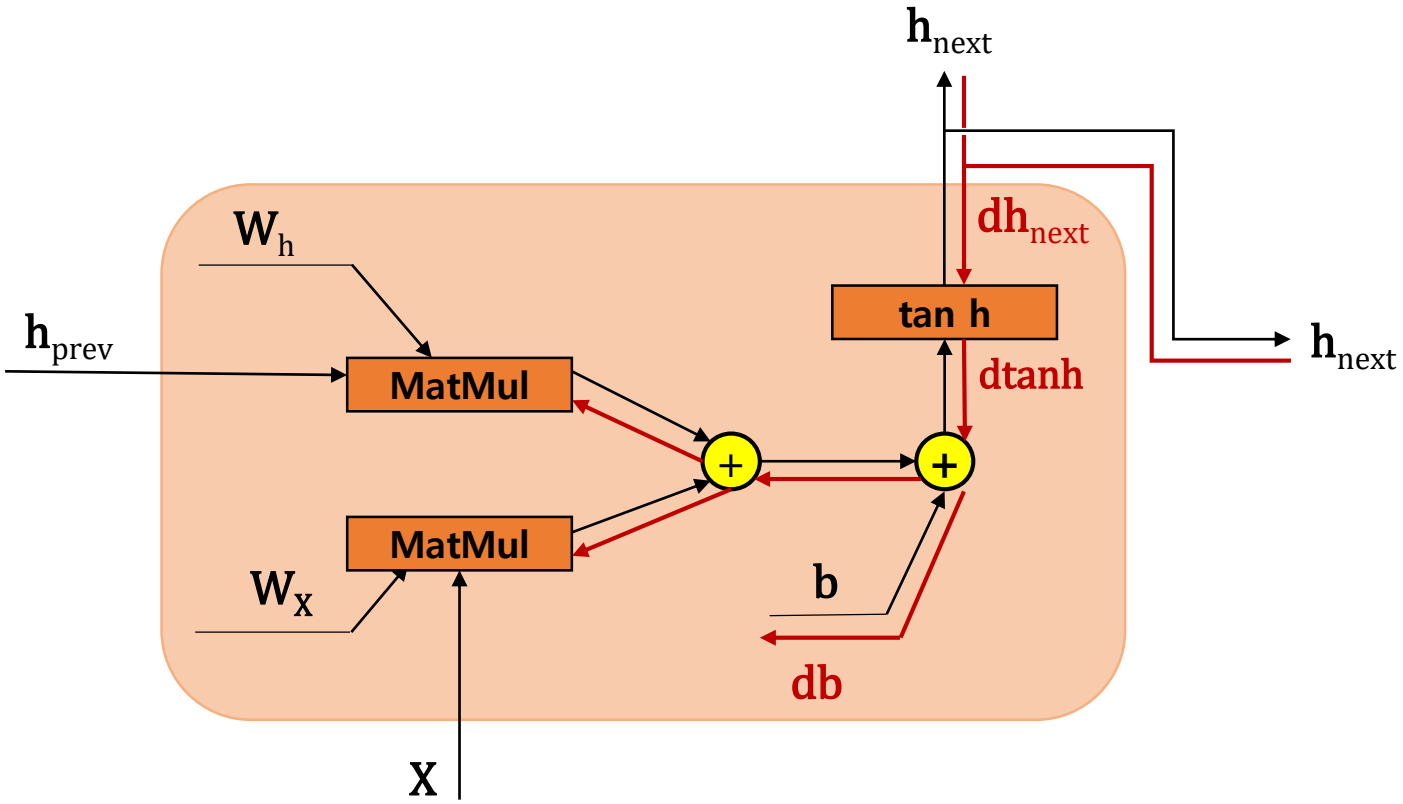
Recurrent Neural Network (RNN)

backward (역전파) - (2) dtanh



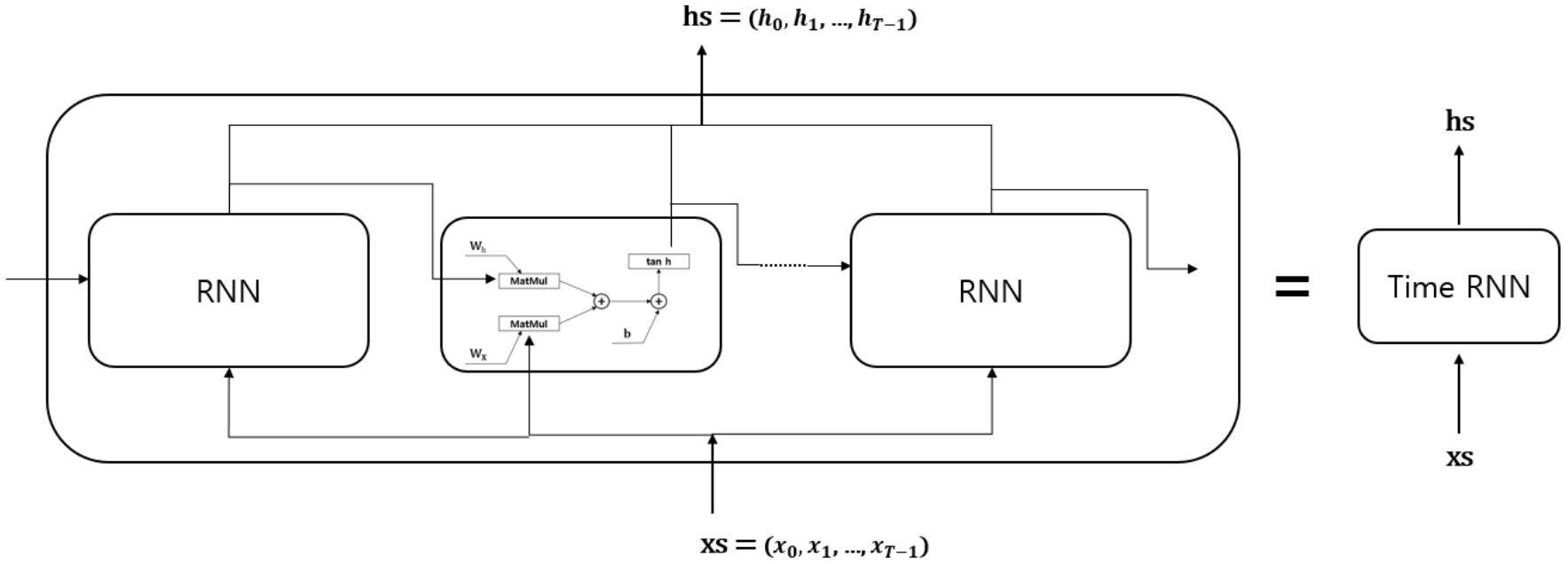
Recurrent Neural Network (RNN)

backward (역전파) – (3) 덧셈 노드



Time RNN 계층 구현

- Time RNN

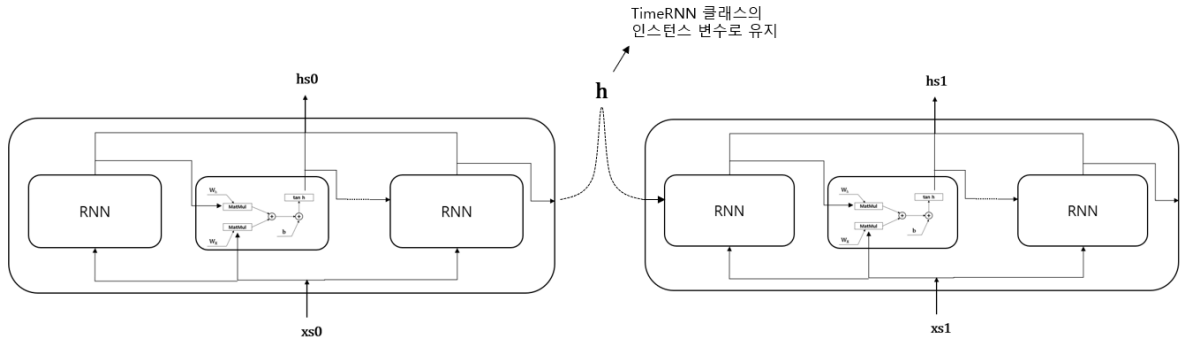


앞에서까지는 하나의 RNN 계층에서의 순전파, 역전파에 대해서 살펴봤다. 이제 **RNN 계층 T개를 연결한 신경망**을 완성해보자.
우리는 이렇게 T개의 RNN을 연결한 신경망을 TimeRNN이라고 부를 것이다.

Time RNN 계층 구현

- TimeRNN 구현

```python
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

        # RNN의 마지막 Hidden State와 dh를 저장하는 변수
        self.h, self.dh = None, None
        # RNN의 은닉 상태를 유지해야 할지를 결정하는 변수
        # stateful 이 True면, 은닉 상태 유지, stateful이 False면 은닉 상태 초기화
        self.stateful = stateful

    # RNN의 Hidden State 설정
    def set_state(self, h):
        self.h = h

    # RNN의 Hidden State 초기화
    def reset_state(self):
        self.h = None
```



TimeRNN 계층

NOTE_ 긴 시계열 데이터를 처리할 때는 RNN의 은닉 상태를 유지해야한다. 이처럼 은닉 상태를 유지할지 여부를 많은 딥러닝 프레임워크에서 'stateful'이라는 단어로 지정한다.

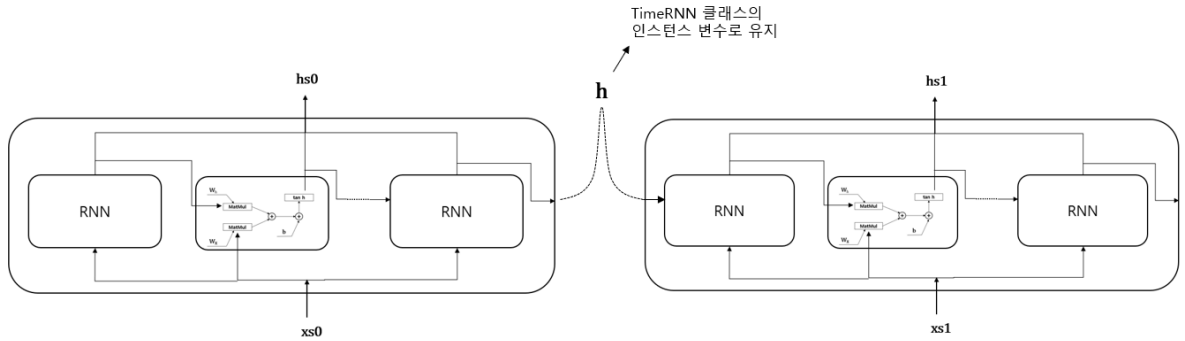
- __init__()

초기화 메서드는 가중치화 편향, 그리고 **stateful**이라는 불리언 값을 인수로 받는다.
인스턴스 변수 중 **layers** 변수는 다수의 RNN 계층을 리스트로 저장하는 용도이다. 그리고 인스턴스 변수 **h**는 forward() 메서드를 불렀을 때 마지막 RNN 계층의 은닉 상태를 저장하고, **dh**는 backward()를 불렀을 때 하나 앞 블록의 은닉 상태의 기울기를 저장한다. (dh는 현재 구현상에서는 필요없으나 이후 seq2seq에서 사용을 위함)

Time RNN 계층 구현

- TimeRNN 구현

```python
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

        # RNN의 마지막 Hidden State와 dh를 저장하는 변수
        self.h, self.dh = None, None
        # RNN의 은닉 상태를 유지해야 할지를 결정하는 변수
        # stateful 이 True면, 은닉 상태 유지, stateful이 False면 은닉 상태 초기화
        self.stateful = stateful

    # RNN의 Hidden State 설정
    def set_state(self, h):
        self.h = h

    # RNN의 Hidden State 초기화
    def reset_state(self):
        self.h = None
```



TimeRNN 계층

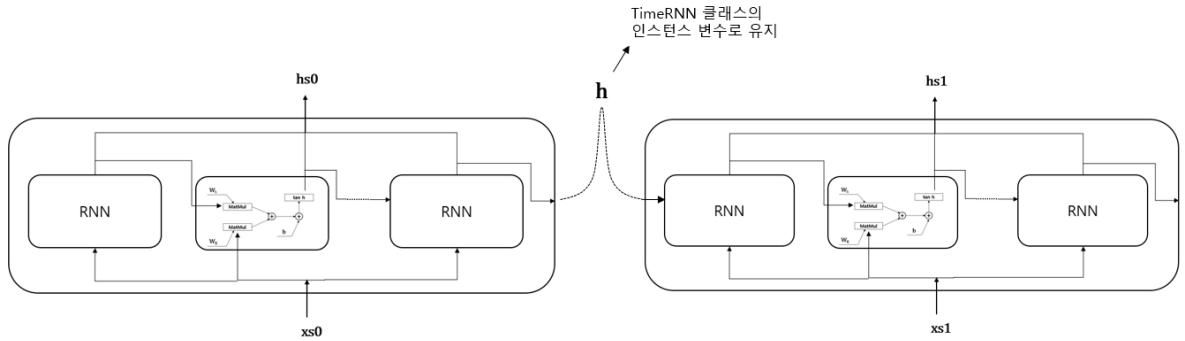
- set_state() & reset_state()

Hidden State (**h**) 를 설정 혹은 초기화 시키는 메서드

Time RNN 계층 구현

- TimeRNN 구현

```
def forward(self, xs):
    Wx, Wh, b = self.params
    N, T, D = xs.shape  # xs.shape = NxTxD
    D, H = Wx.shape  # Wx.shape = DxH

    self.layers = list()
    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')

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

    return hs
```



TimeRNN 계층

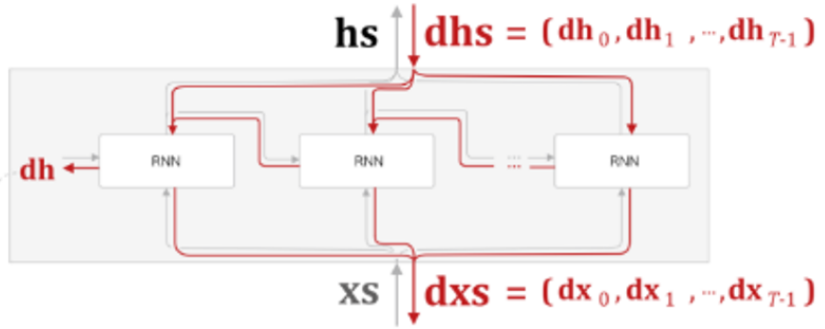
- forward()

순전파 메서드인 forward(xs)는 아래로부터 입력 xs를 받는다. Xs는 T개 분량의 시계열 데이터를 하나로 모은 것이다. 미니배치 처리를 고려했을 때의 xs의 형상은 NxTxD 가 된다. (N : 미니배치, T : 시계열 데이터 수, D : 입력벡터 차원 수)
은닉상태 h는 처음 호출 시 영행렬로 초기화를 하고, 시계열 데이터의 수만큼 for문을 돌면서 Hidden State를 업데이트한다.
그리고 이 Hidden State의 집합임 **hs**를 반환한다

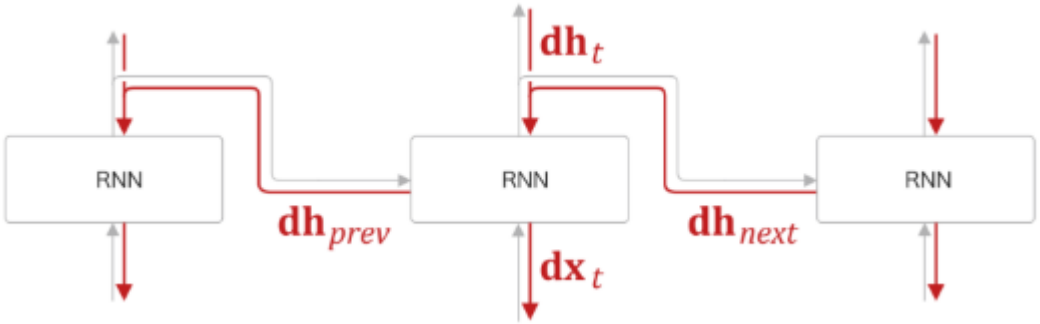
Time RNN 계층 구현

- TimeRNN 구현

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

    dxs = np.empty((N, T, D), dtype='f')
    dh = 0  # 맨 처음 dh는 0 (Truncated BPTT이기 때문)
    grads = [0, 0, 0]  # [dWx, dWh, db] 초기화

    for t in reversed(range(T)):
        layer = self.layers[t]
        dx, dh = layer.backward(dhs[:, t, :] + dh)  # 합산된 기울기
        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
```



TimeRNN 계층의 역전파



t번째 RNN 계층의 역전파

- backward()

TimeRNN의 순전파시에 출력이 2개로 분기된다. 순전파 시 분기했을 경우, 그 역전파에서는 각 기울기가 합산되어 전해진다. 따라서 역전파 시 RNN 계층에서는 합산된 기울기 $(dh_t + dh_{next})$가 입력된다. 미리 만들어뒀던 RNN 계층의 backward()를 호출하여 각 시각의 기울기 dx를 구해서 dxs의 해당 인덱스(시각)에 저장한다. 그리고 가중치 매개변수에 대해서도 각 RNN 계층의 가중치 기울기를 합산하여 최종 결과를 멤버 변수 self.grads에 덮어쓴다.