Recurrent Neural Network (RNN)

Winter Vacation Capstone Study

TEAM Kai.Lib

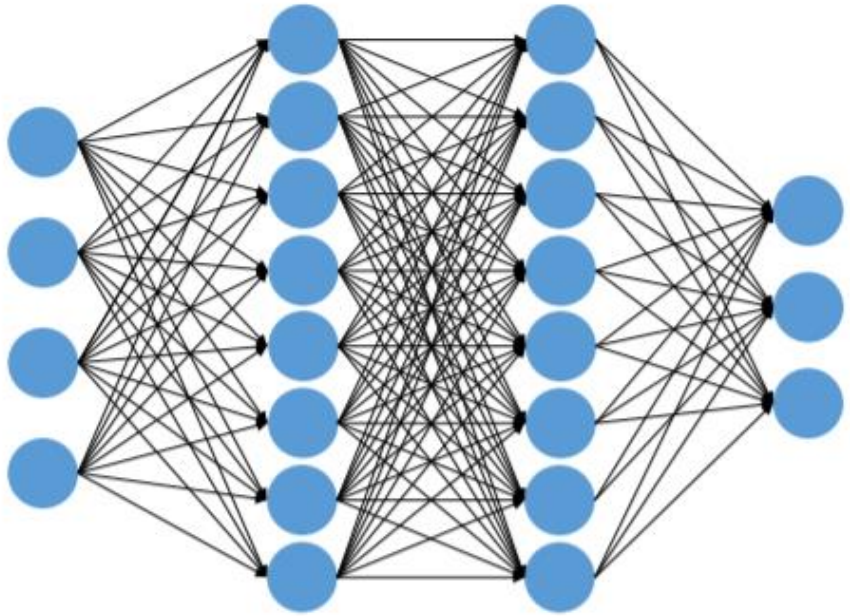
발표자 : 김수환

2019.12.27 (FRI)

RNN의 등장배경

▪ 피드포워드 신경망의 문제점

피드포워드란 흐름이 단방향인 신경망을 뜻한다.
피드포워드 구조는 구성이 단순하여 구조를 이해하기 쉽고 많은 문제에 응용할 수 있다는 장점이 있지만, 커다란 단점이 하나 있으니, 바로 **시계열 데이터를 잘 다루지 못한다**는 것이다.
더 정확하게 말하면, 단순한 피드포워드 신경망에서는 시계열 데이터의 성질(패턴)을 충분히 학습할 수 없다. 그래서 순환 신경망Recurrent Neural Network(RNN)이 등장하게 된다.



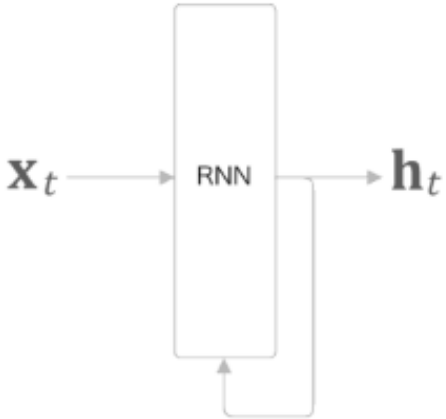
Feed Forward Neural Network

RNN(순환신경망) 이란

▪ 순환하는 신경망

RNN의 특징은 순환하는 경로 (닫힌 경로) 가 있다는 것이다. 이 순환 경로를 따라 데이터는 끊임없이 순활할 수 있다.
그리고 데이터가 순환되기 때문에 **과거의 정보를 기억하는 동시에 최신 데이터로 갱신**될 수 있다.

순환 경로를 포함하는 RNN 계층



NOTE_ 그림을 잘 보면 출력이 2개로 분기하고 있음을 알 수 있다.
여기서 말하는 '분기'란 같은 것이 복제되어 분기함을 의미한다.

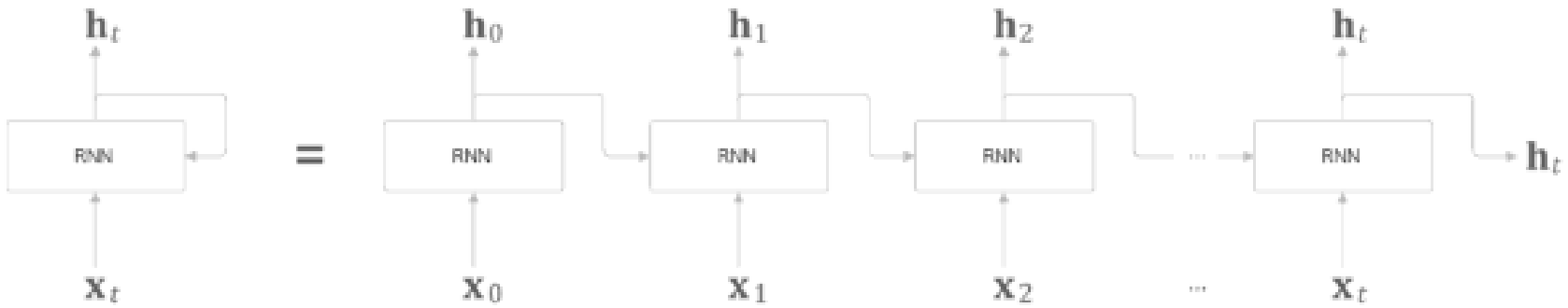
위의 그림처럼 RNN 계층은 순환하는 경로를 포함한다. 이 순환 경로를 따라 데이터를 계층 안에서 순환시킬 수 있다.
X_t를 입력받는데, t는 시각을 뜻한다. 이는 (X_0, X_1, ..., X_t) 가 RNN 계층에 입력됨을 표현한 것이다. 그리고 그 입력에 대응하여 (h_0, h_1, ..., h_t) 가 출력된다.

RNN(순환신경망) 이란

▪ 순환 구조 펼치기 (1)

RNN의 순환 구조는 피드포워드까지에서는 볼 수 없던 구조이지만, 이 순환 구조를 펼치면 친숙한 신경망으로 변신시킬 수 있다.

RNN 계층의 순환 구조 펼치기



위의 그림에서 보듯, RNN 계층의 순환 구조를 펼침으로써 오른쪽으로 진행하는 피드포워드 신경망과 같은 구조다.
하지만 RNN에서는 다수의 RNN 계층 모두가 실제로는 '같은 계층'인 것이 피드포워드 신경망과는 다르다.

위의 그림에서 알 수 있듯, 각 시각의 RNN 계층은 그 계층으로의 입력과 개 전의 RNN 계층으로부터의 출력을 받는다.
그리고 이 두 정보를 바탕으로 현 시각의 출력을 계산한다. 이때 수행하는 계산의 수식은 다음과 같다.

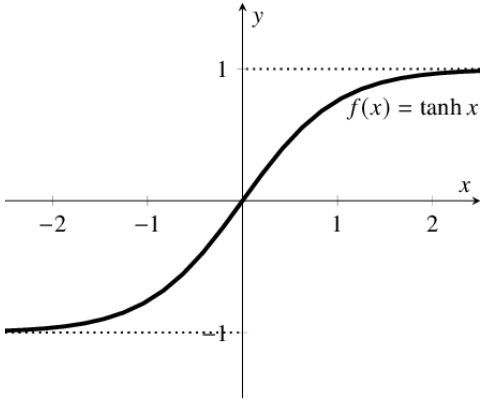
$$h_t = \tanh(h_{t-1}W_h + X_tW_x + b)$$

RNN(순환신경망) 이란

- **순환 구조 펼치기 (2)**

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

RNN에서는 가중치가 2개 있다. 하나는 입력 x를 출력 h로 변환하기 위한 가중치 W_x이고, 다른 하나는 1개의 RNN 출력을 **다음 시각의 출력**으로 변환하기 위한 가중치 W_h이다. 위의 식에서 행렬 곱을 계산하고, 그 합을 tanh 함수를 이용해 변환한다.



Hyperbolic tangent

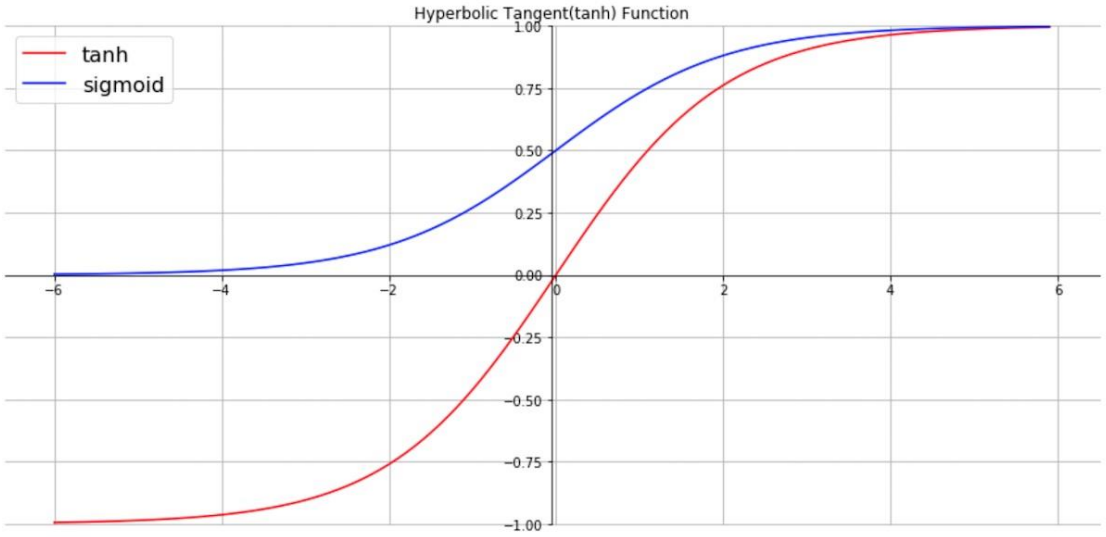
그 결과가 시각 t의 출력 h_t가 된다. 이 h_t는 다른 계층을 향해 위쪽으로 출력되는 동시에, 다음 시각의 RNN 계층 (자기 자신)을 향해 오른쪽으로도 출력된다.

이러한 구조를 생각해보면 현재의 출력 h_t는 한 시각 이전 출력 h_{t-1}에 기초해 계산됨을 알 수 있다. 다른 관점으로 보면 RNN은 h라는 상태를 가지고 있으며, 위의 식 형태로 갱신된다고 해석할 수 있다. 그래서 RNN 계층을 상태를 가지는 계층 혹은 메모리가 있는 계층이라고 한다.

NOTE_ RNN의 h는 상태를 기억해 시각이 1 스텝 진행될 때마다 위의 식 형태로 갱신된다. 여러 문헌들에서 RNN의 출력 H_t를 은닉 상태**Hidden State** 혹은 은닉 상태 벡터**Hidden State Vector**라고 한다.

RNN(순환신경망) 이란

- **Hyperbolic tangent**



Hyperbolic tangent vs Sigmoid

$$
\begin{aligned}
\frac{d}{dx}tanh(x) &= \left[\frac{e^x - e^{-x}}{e^x + e^{-x}}\right]' \\
&= \left[\frac{f(x)}{g(x)}\right]' \\
&= \frac{f'(x)g(x) - f(x)g'(x)}{g^2(x)} \\
&= \frac{(e^x - (-e^{-x}))(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(1 + e^{-x})^2} \\
&= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\
&= \frac{(e^x + e^{-x})^2}{(e^x + e^{-x})^2} - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\
&= 1 - \left[\frac{(e^x - e^{-x})}{(e^x + e^{-x})}\right]^2 \\
&= 1 - tanh^2(x)
\end{aligned}
$$

Hyperbolic tangent의 미분

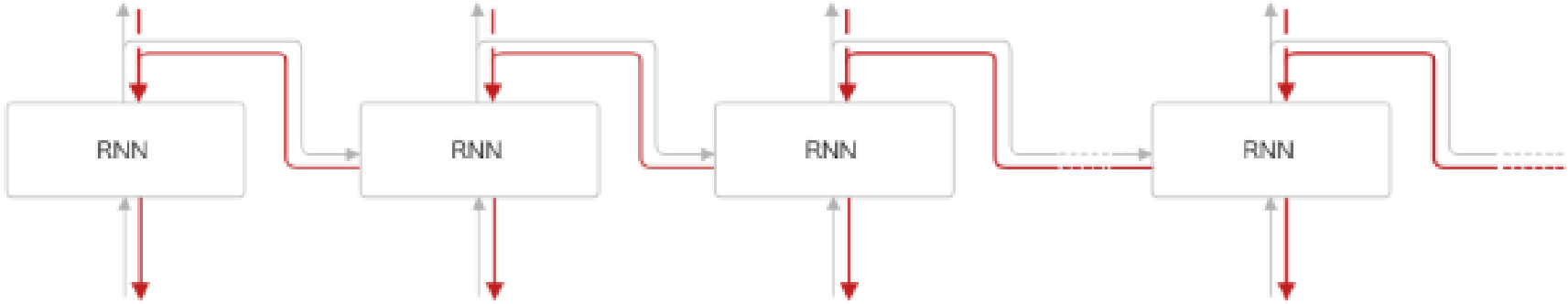
Hyperbolic tangent 함수는 Sigmoid 함수의 대체제로 사용될 수 있는 활성화 함수이다.
tanh는 Sigmoid와 매우 유사하다. tanh와 sigmoid의 차이점은 sigmoid의 출력 범위가 0 ~ 1인 반면 tanh의 출력 범위는 -1 ~ 1 사이라는 점이다. Sigmoid와 비교하여 tanh는 출력 범위가 더 넓고 경사면이 큰 범위가 더 크기 때문에 더 빠르게 수렴하여 학습하는 특성이 있다.

RNN의 Backpropagation

▪ BPTT (Backpropagation Through Time)

앞에서 봤듯이 RNN 계층은 가로로 펼친 신경망으로 간주할 수 있다. 따라서 RNN의 학습도 보통의 신경망과 같은 순서로 진행할 수 있다. 다음 그림과 같은 모습이 된다.

순환 구조를 펼친 RNN 계층에서의 오차역전파법



위의 그림에서 보듯, 순환 구조를 펼친 후의 RNN에는 피드포워드 오차역전파법을 적용할 수 있다. 즉, 먼저 순전파를 진행하고 이어서 역전파를 수행하여 원하는 기울기gradient를 구할 수 있다. 여기서의 오차역전파법은 '시간 방향으로 펼친 신경망의 오차역전파법'이라는 뜻으로 **BPTT**Backpropagation Through Time라고 한다.

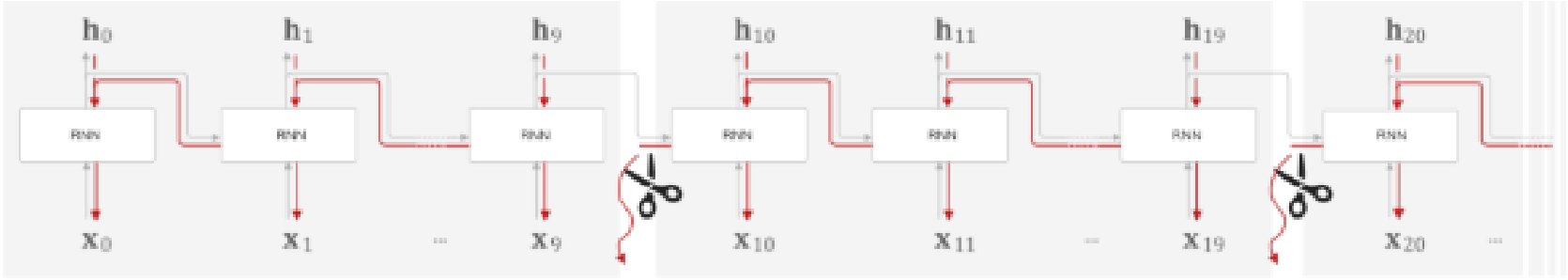
RNN의 Backpropagation

▪ Truncated BPTT (1)

이 BPTT를 이용하면 RNN을 학습할 수 있을 듯 하다. 하지만 그 전에 해결해야 할 문제가 하나 있다. 그것은 바로 긴 시계열 데이터를 학습할 때의 문제이다. 시계열 데이터의 시간 크기가 커지는 것에 비례하여 BPTT가 소비하는 컴퓨팅 자원도 증가하기 때문이다. 또한 시간 크기가 커지면 역전파 시에 기울기 값이 조금씩 작아져서 0에 가까워지는 문제도 발생한다.

Vanishing Gradient Problem : 신경망을 통과할 때마다 기울기 값이 조금씩 작여저서 0에 가까워지는 문제

이러한 문제를 해결하기 위해 신경망 연결을 적당한 길이로 '**끊는다**'. 시간축 방향으로 너무 길어진 신경망을 적당한 지점에서 잘라내 여러 개로 만든다는 아이디어다. 그리고 이 잘라낸 작은 신경망에서 오차역전파법을 수행한다. 이것이 **Truncated BPTT**라는 기법이다.



Truncated BPTT에서는 신경망의 연결을 끊지만, 제대로 구현하려면 **'역전파'의 연결만 끊고, 순전파의 연결은 반드시 그대로 유지**해야 한다. 즉, 순전파의 흐름은 끊어지지 않고 전파된다.

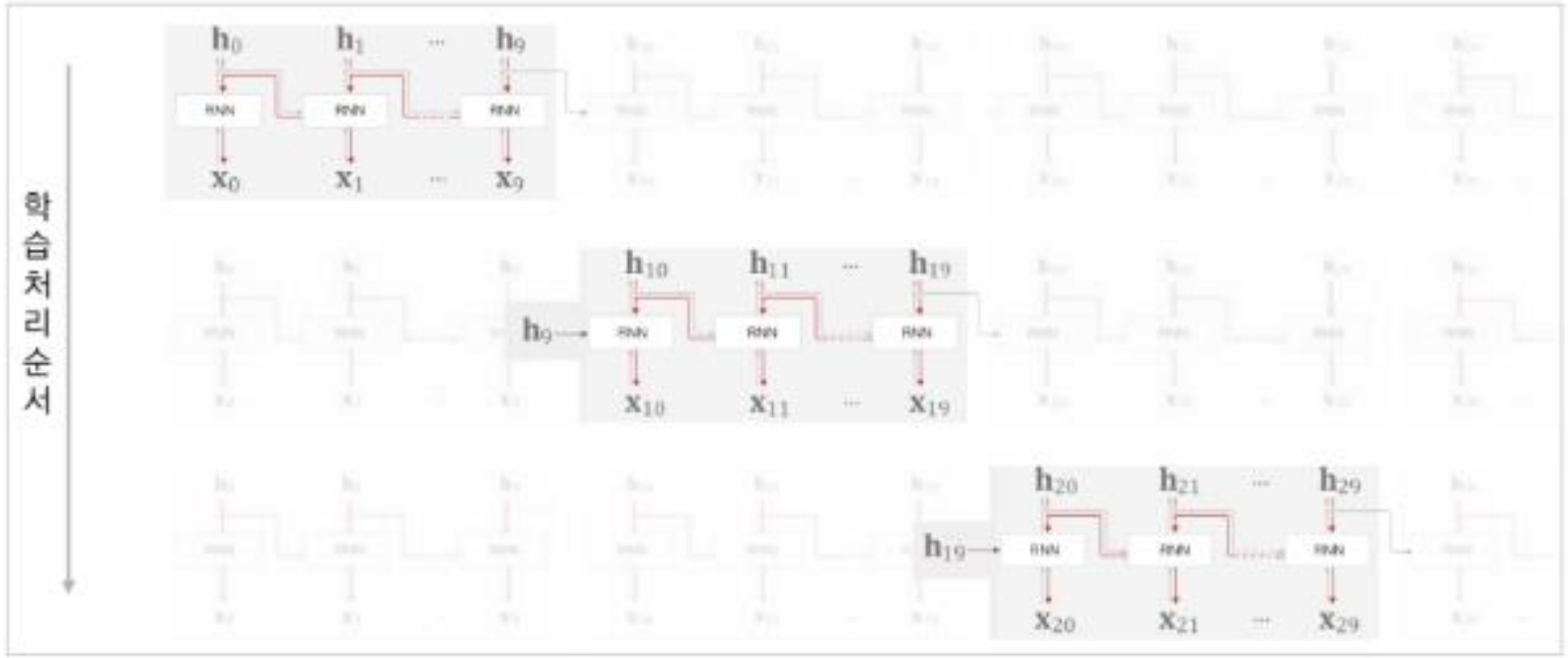
RNN의 Backpropagation

▪ Truncated BPTT (2)

Truncated BPTT를 구체적인 예를 가져와 살펴보자. 길이가 1,000인 시계열 데이터가 잇다고 해보자. 길이가 1,000인 시계열 데이터를 다루면서 RNN 계층을 펼치게 되면 계층이 가로로 1,000개나 늘어선 신경망이 된다. 물론 계층이 아무리 늘어서더라도 오차역전파법으로 기울기를 계산할 수는 있으나, 계산량과 메모리 사용량 등이 문제가 된다.
그래서 순전파의 연결은 유지한 채 역전파의 연결만 적당한 지점에서 끊는다. 이처럼 역전파의 연결을 잘라버리면, 그보다 미래의 데이터에 대해서는 생각할 필요가 없다. 각각의 블록 단위로, 미래의 블록과는 독립적으로 오차역전파법을 완성할 수 있다.

RNN의 Backpropagation

- **Truncated BPTT (3)**



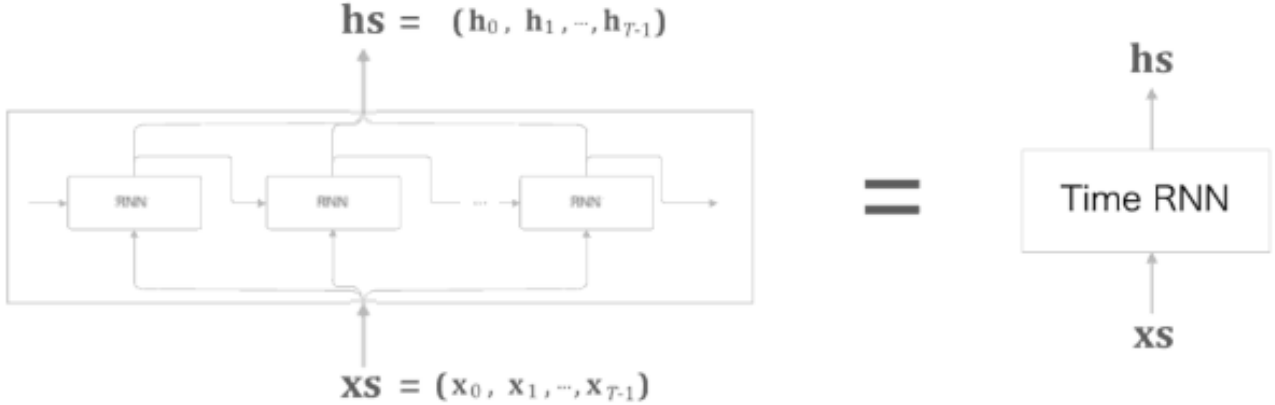
위의 그림처럼 Truncated BPTT에서는 데이터를 순서대로 입력해 학습한다. 이런 식으로 순전파의 연결을 유지하면서 블록 단위로 오차역전파법을 적용할 수 있다.

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



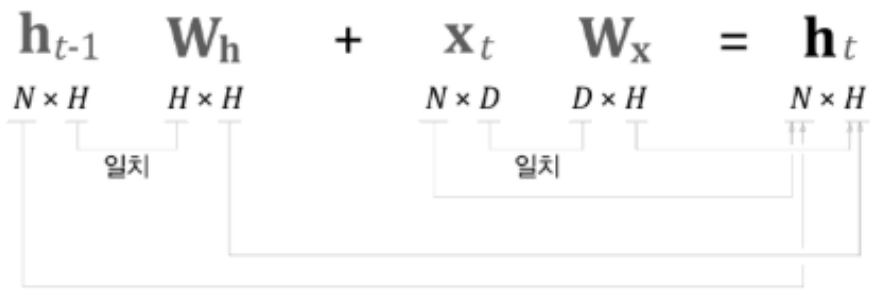
먼저 RNN의 한 단계를 처리하는 클래스를 RNN이란 이름으로 구현한다. 그리고 이 RNN 클래스를 이용해 T개 단계의 처리를 한꺼번에 수행하는 계층을 TimeRNN이란 이름의 클래스로 완성시킨다.

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.

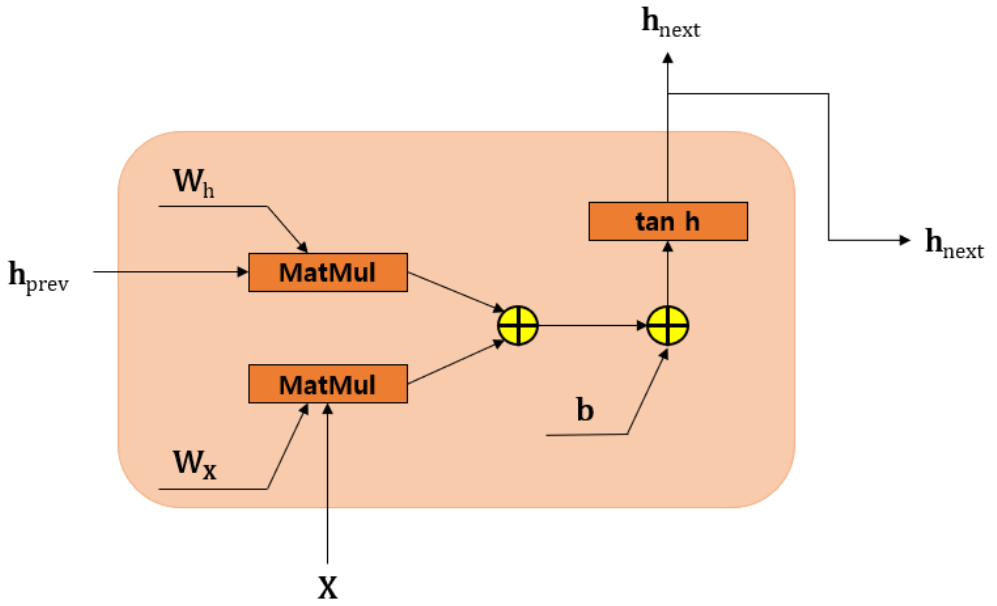
RNN 구현

- **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.

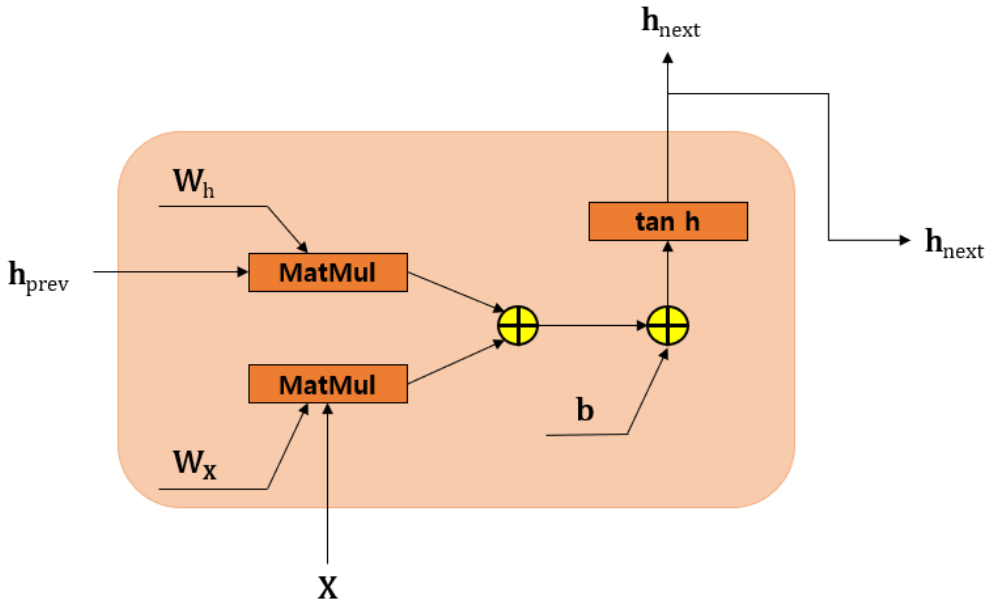
RNN 구현

- **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



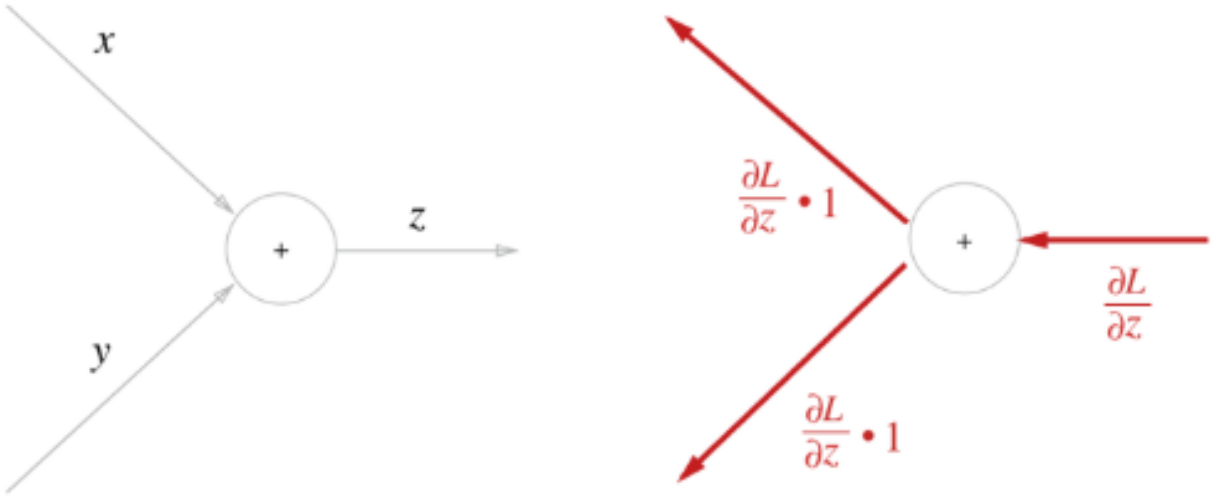
RNN 계층의 계산 그래프

- 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

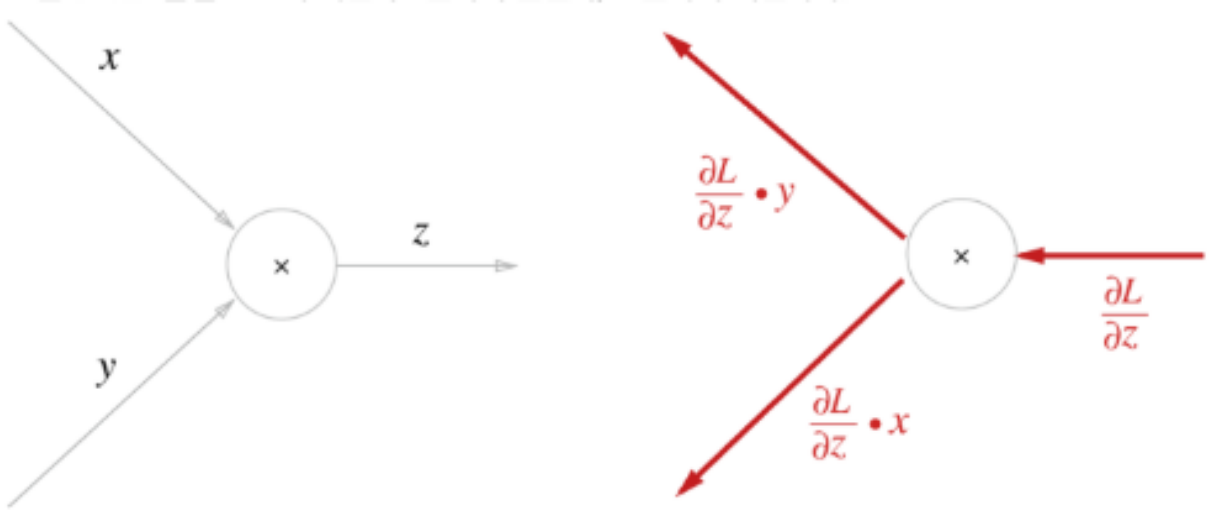
- **덧셈 노드의 역전파**



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**



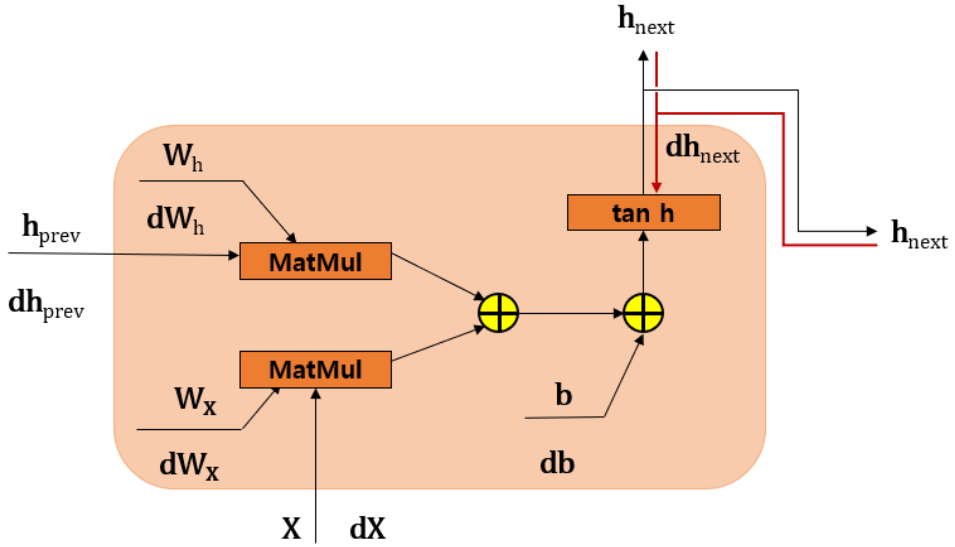
곱셈 노드 역전파는 상류의 값에 순전파 때의 입력 신호들을 '서로 바꾼 값'을 곱해서 하류로 보낸다.
z = xy 라는 식의 역전파를 생각해보면 $\frac{\partial z}{\partial x}$ = y, $\frac{\partial z}{\partial y}$ = x이 된다. 그러므로, 상류에서 들어온 신호에 순전파 때의 입력 신호들을 서로 바꾼 값을 곱해서 하류로 흘려보내면 된다.

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.

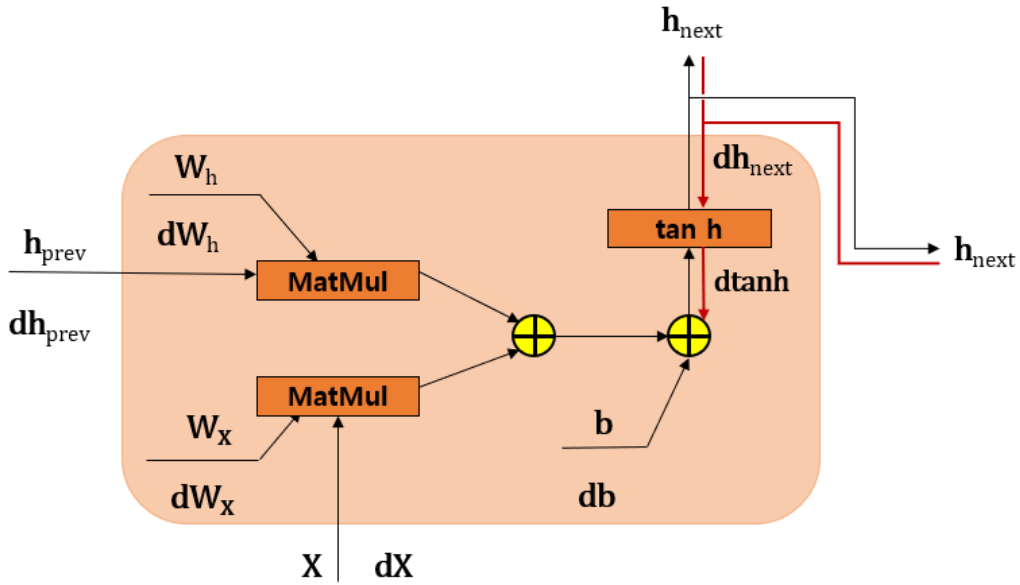
RNN 구현

- **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.

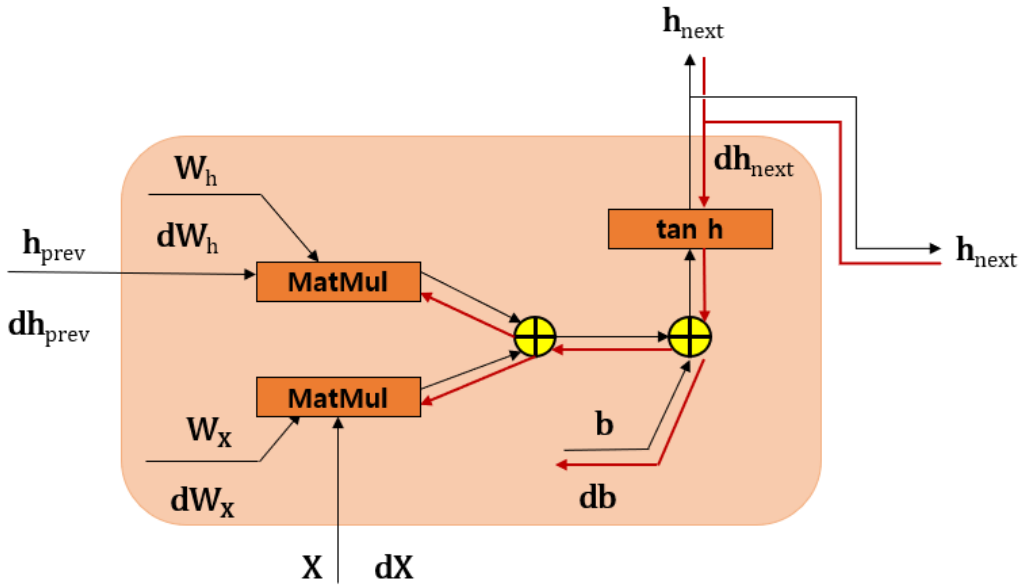
RNN 구현

- **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

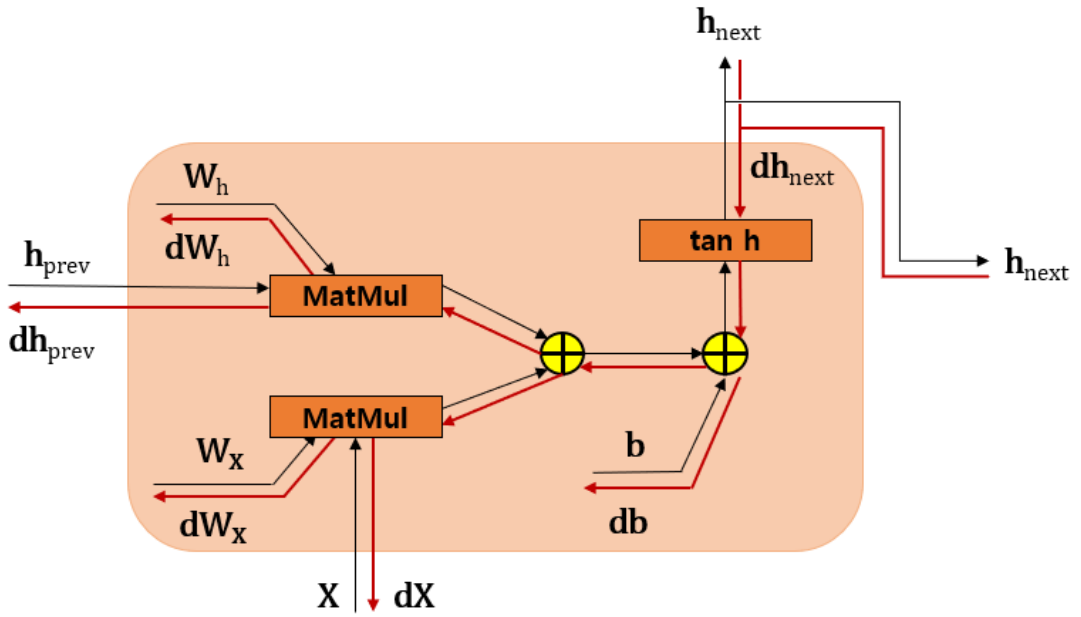
- **RNN 계층의 역전파 –** (4) 곱셈 노드

```python
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)

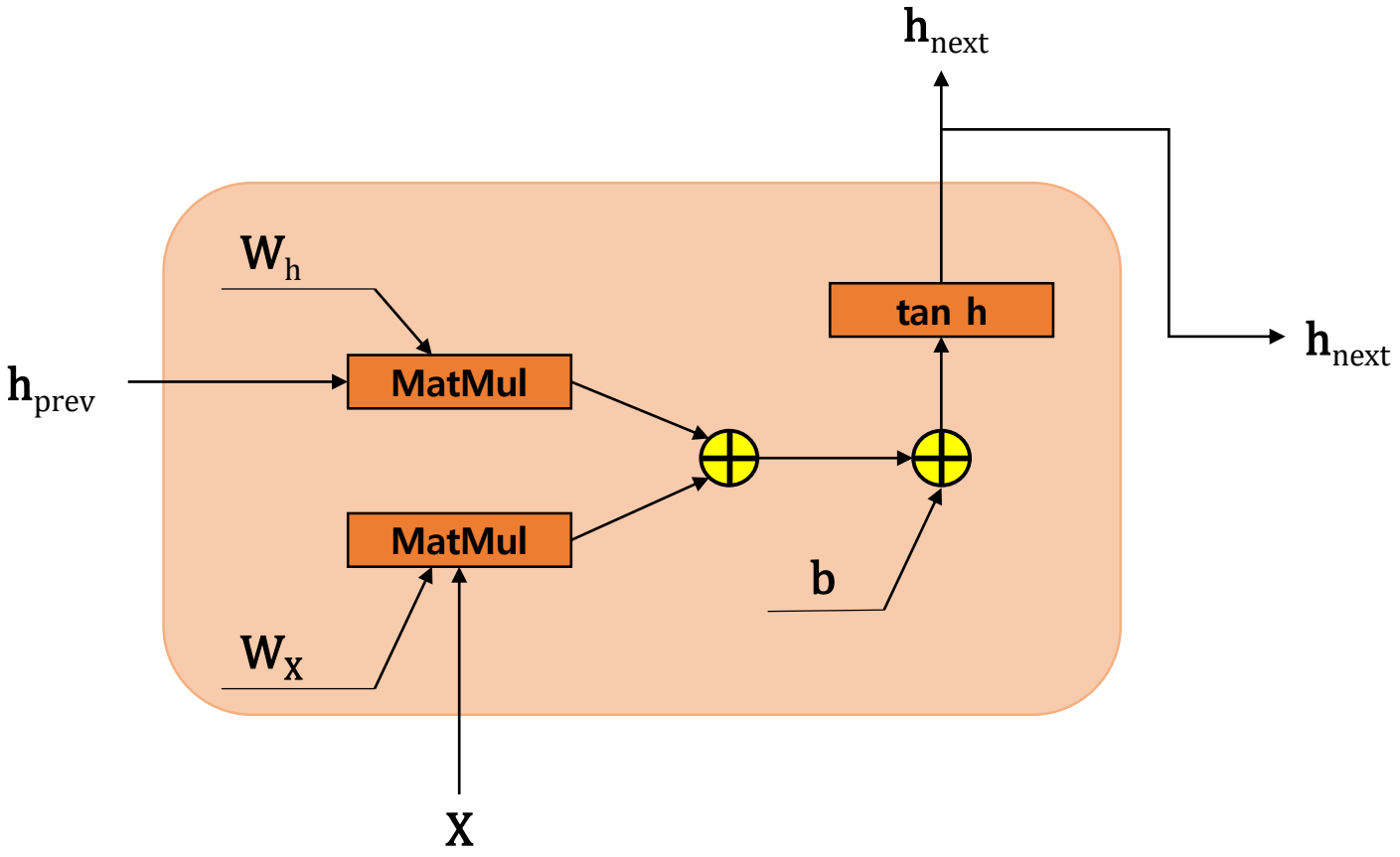
- backward()

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)

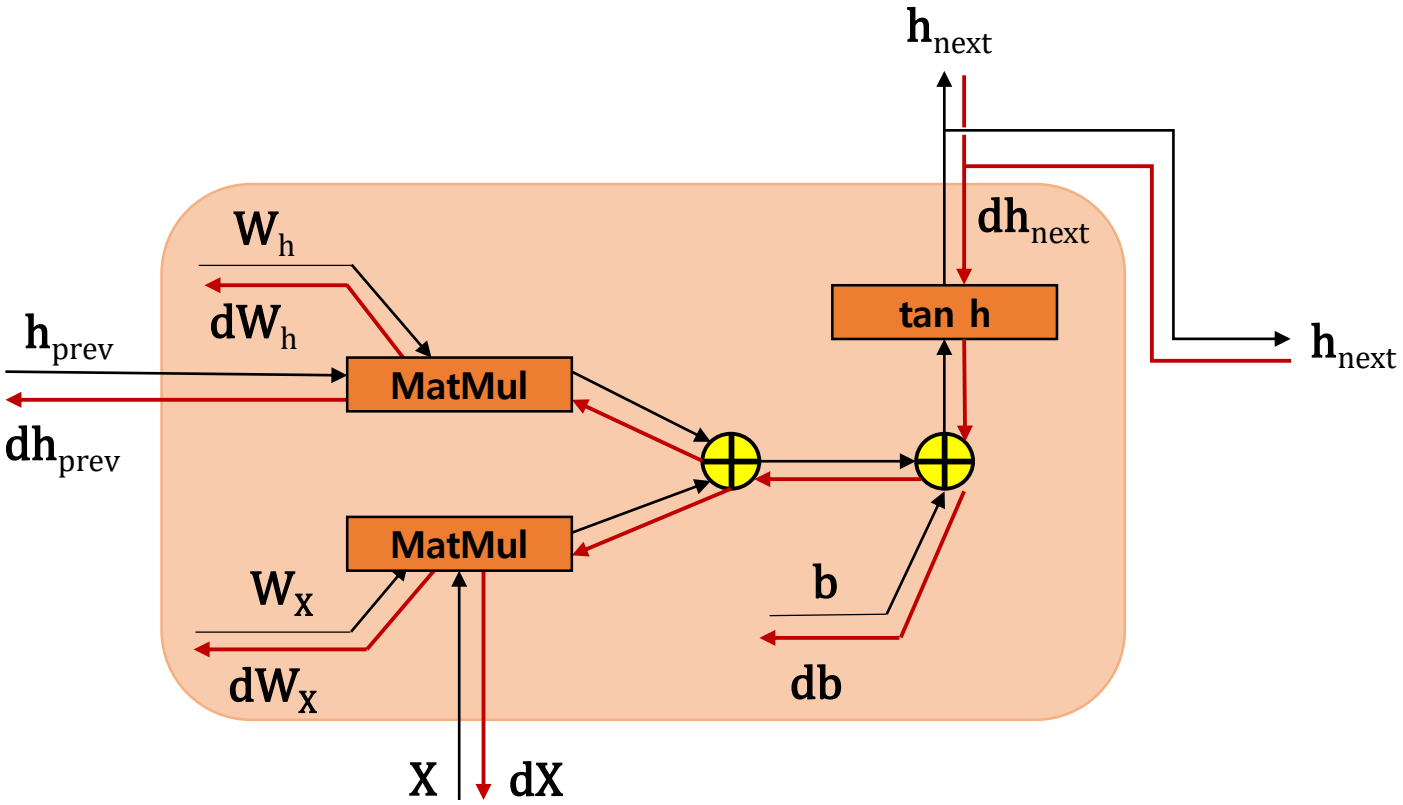
forward (순전파)



Review

Recurrent Neural Network (RNN)

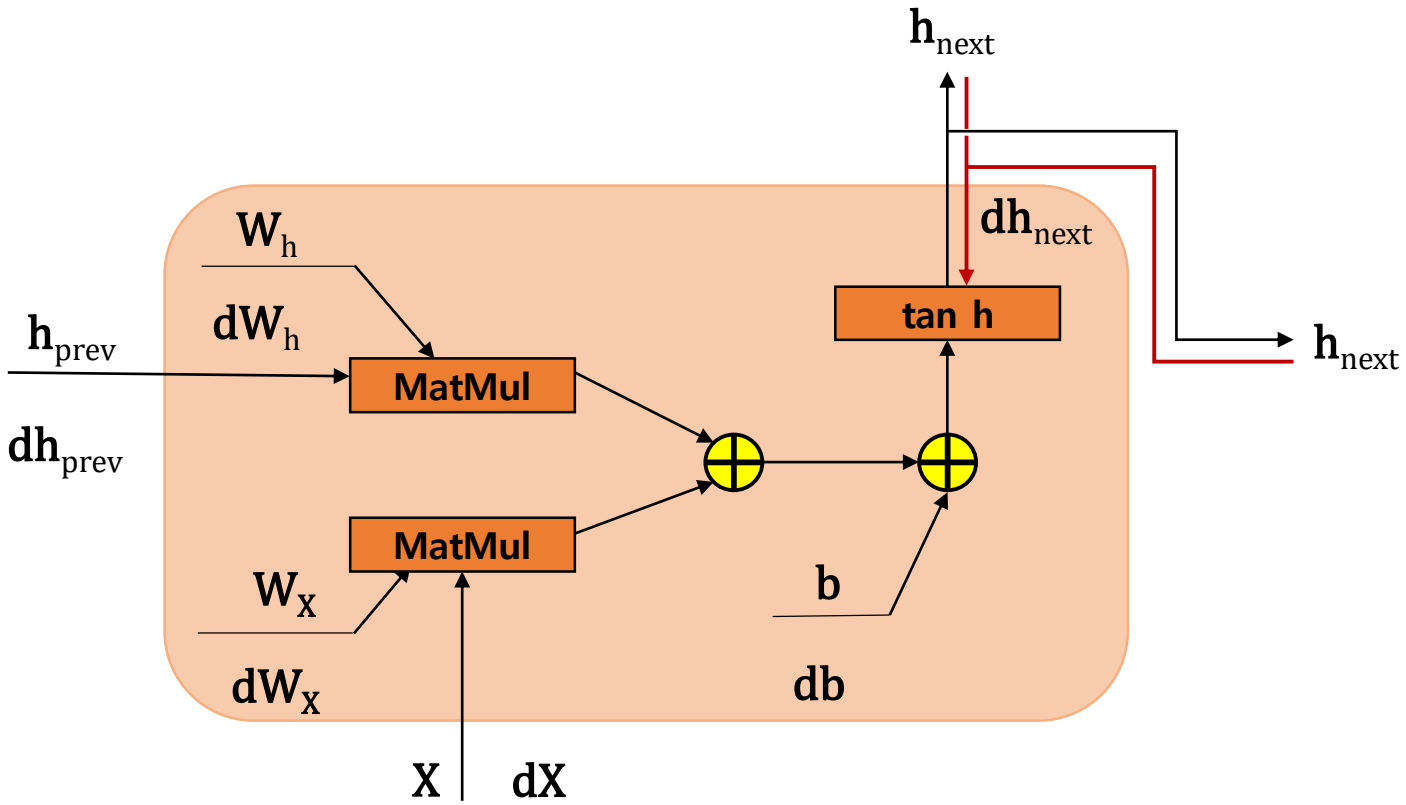
backward (역전파)



Review

Recurrent Neural Network (RNN)

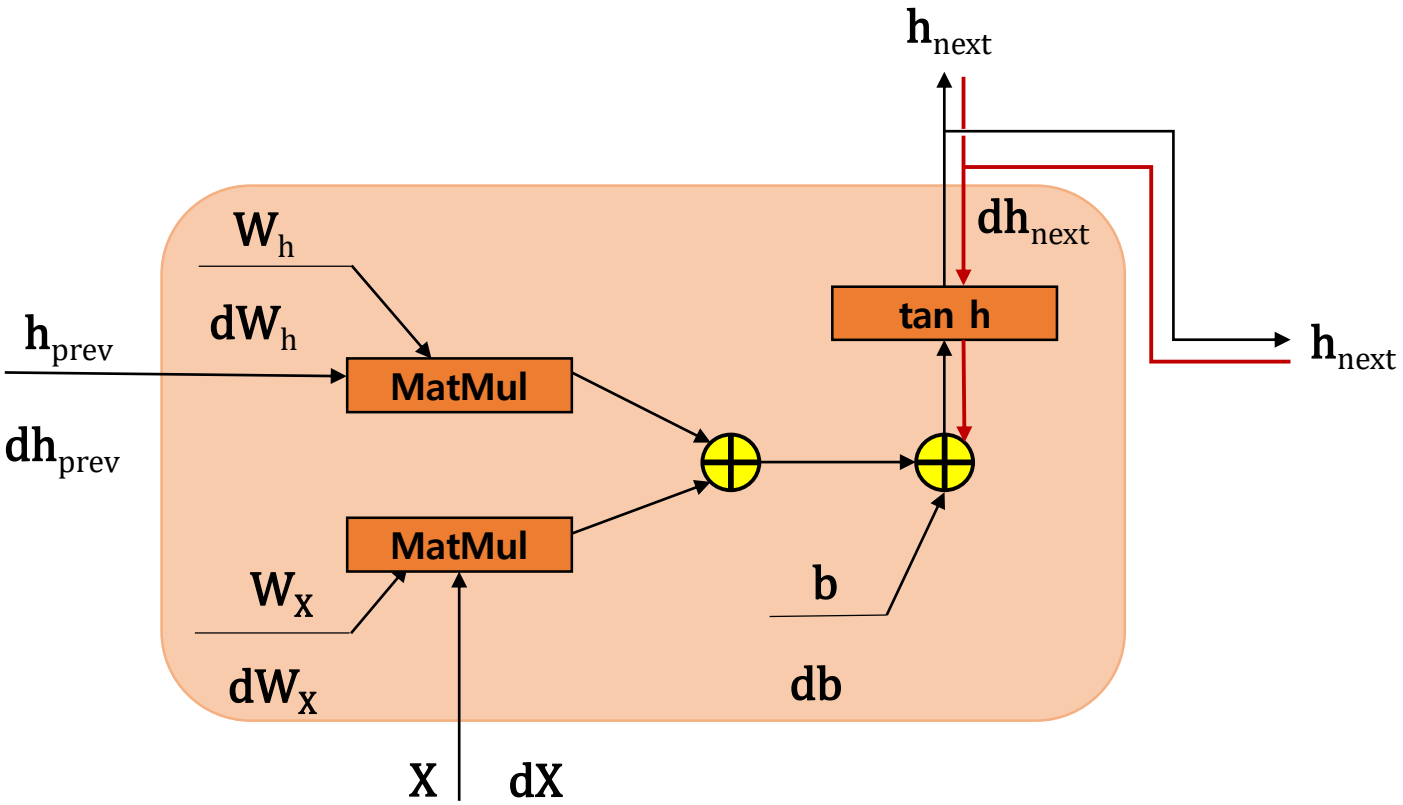
backward (역전파) - (1) dh_next



Review

Recurrent Neural Network (RNN)

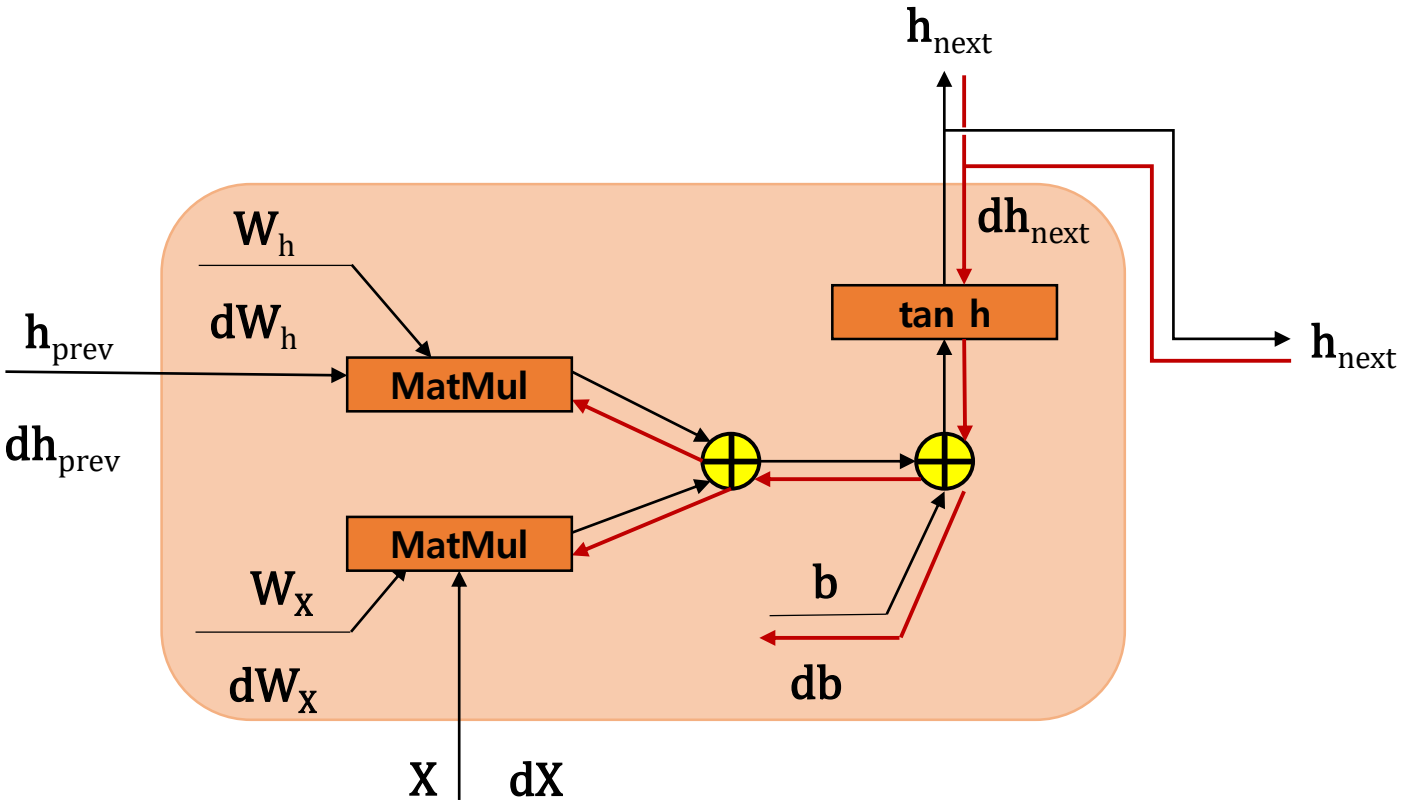
backward (역전파) - (2) dtanh



Review

Recurrent Neural Network (RNN)

backward (역전파) – (3) 덧셈 노드



Review

Recurrent Neural Network (RNN)

backward (역전파) – (4) 곱셈 노드