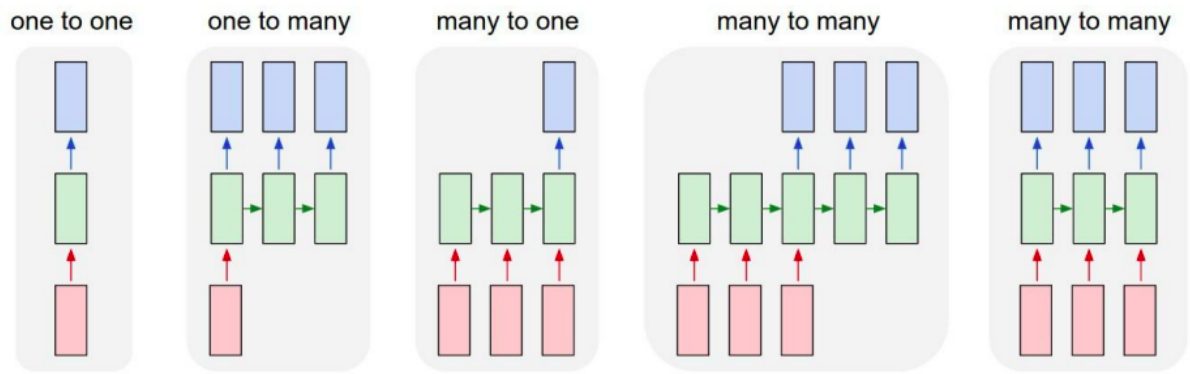
Lecture 10 Recurrent Neural Networks

RNN(순환 신경망)

입출력을 **시퀀스 단위**로 처리하는 시퀀스 모델

시퀀스 순서가 있는 데이터



맨 왼쪽 Vanilla Neural Networks

▼ Process Sequences

one to many - Image Captioning

many to one - 감정 분류

many to many - 기계 번역

many to many - 영상 프레임별 예측

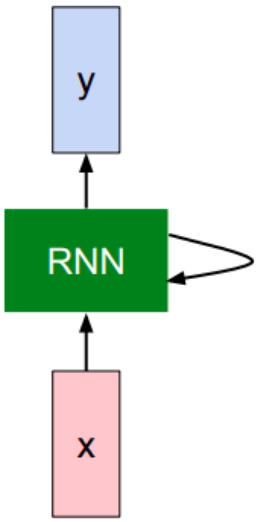
입/출력은 고정이지만 sequencial processing이 요구되는 경우에도 RNN 사용

RNN 동작과정

$$h_t = f_W(h_{t-1}, x_t)$$

new state / some function with parameters W / old state / input vector at some time step

이전 hidden state와 **현재 입력**을 받아서 **다음 hidden state**를 출력



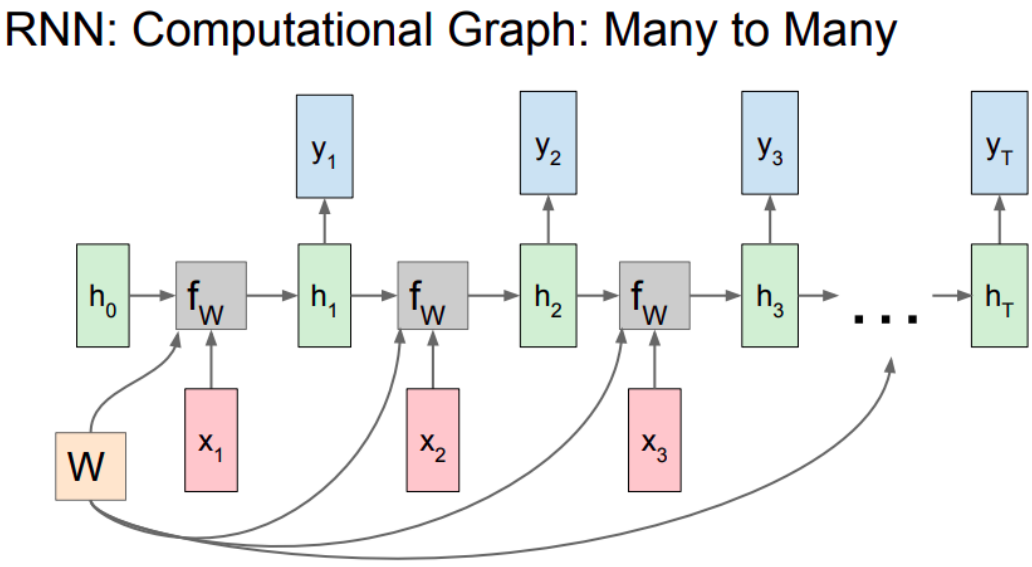
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

1. RNN이 x를 **입력**받음
2. RNN 내부의 **hidden state** 값 업데이트
3. **출력** 값 y를 내보냄 (h_t를 입력으로 하는 FC-layer)

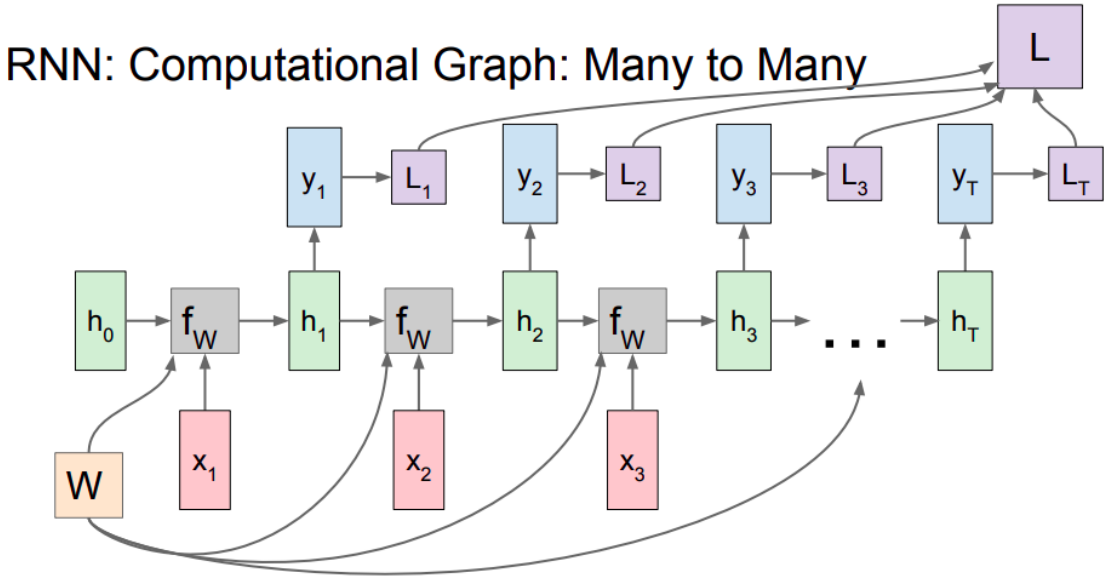
RNN이 **hidden state** 를 거치며 **재귀적**으로 feedback 한다

동일한 가중치 행렬W가 매번 사용됨 (h와 x는 매번 달라짐)



가변 입력 및 가변 출력

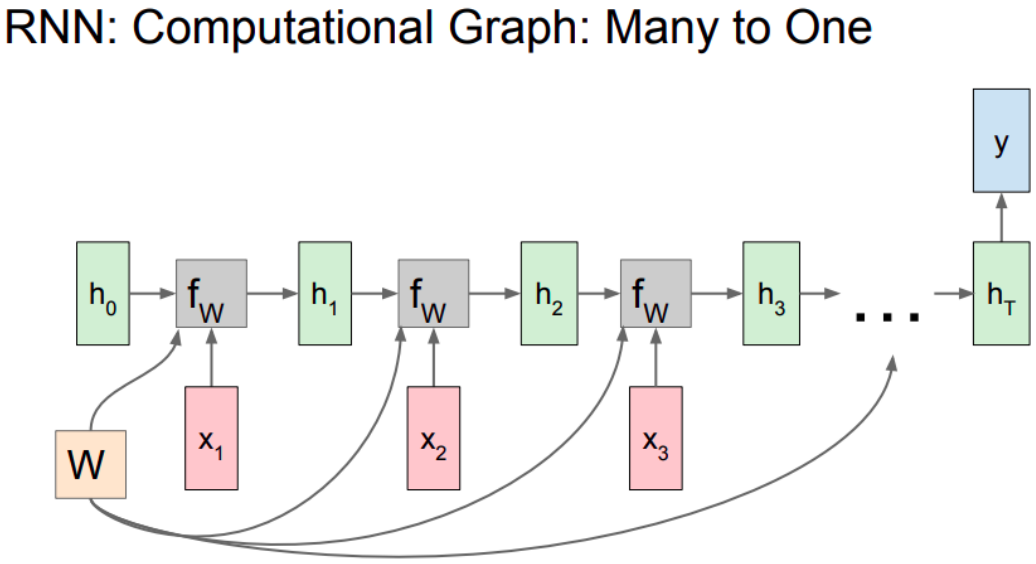
▼ **Loss**



1. 각 스텝마다 **개별**적으로 y_t에 대한 **Loss값** L_t를 구함
2. 값을 다 더해 **최종 Loss**를 구함

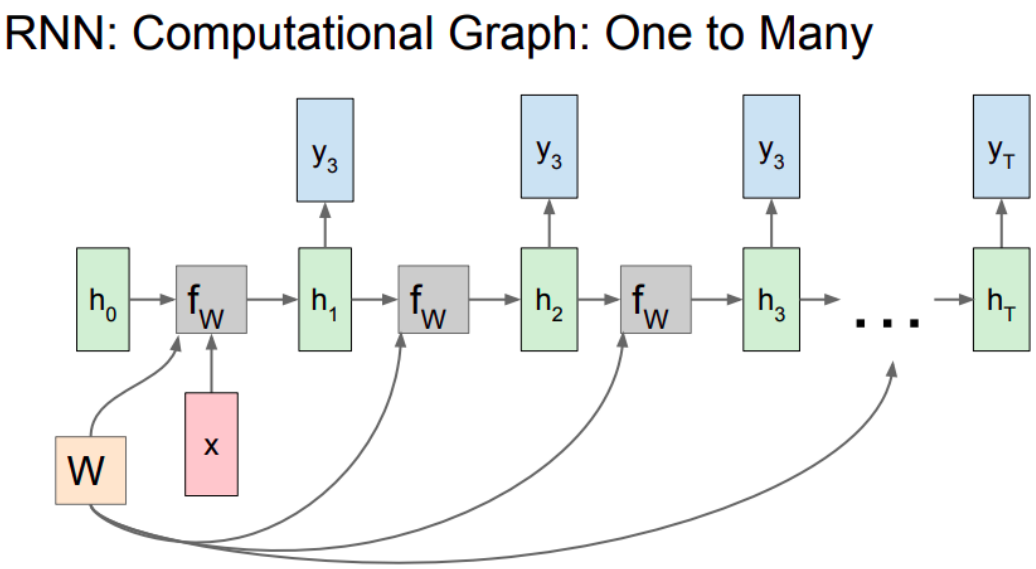
▼ **backprop을 위한 행렬 W의 그레디언트**

1. 각 스텝에서 W에 대한 **local 그레디언트**를 전부 계산 (**dLoss / dW**)
2. local 그레디언트 값들을 합산해 **최종 gradient**를 구함



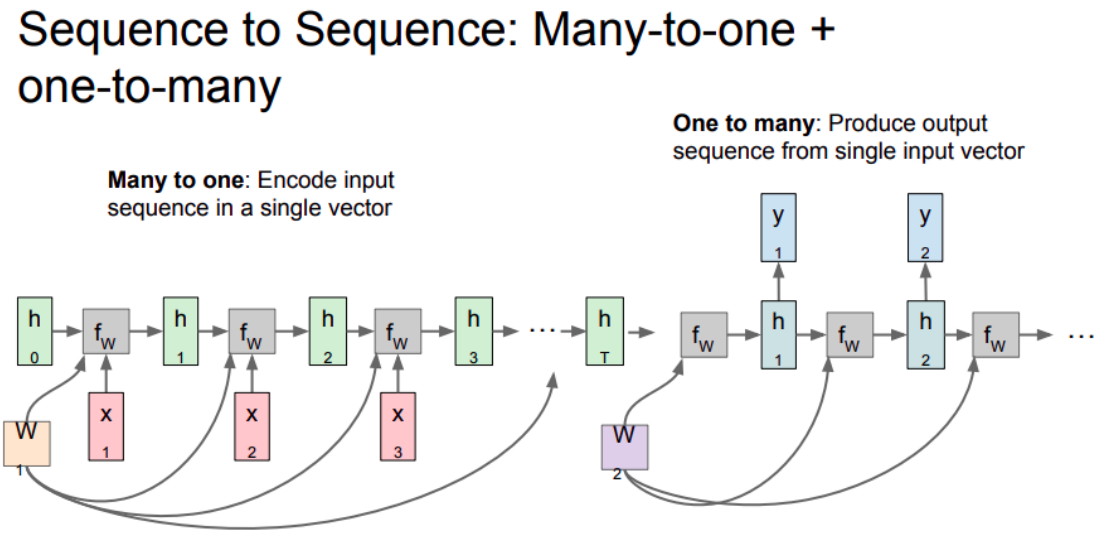
가변 입력 및 고정 출력

최종 hidden state를 통과하면서 전체 sequence에 대한 요약 출력



고정 입력 및 가변 출력

고정 입력이 initial hidden state를 초기화 시킴



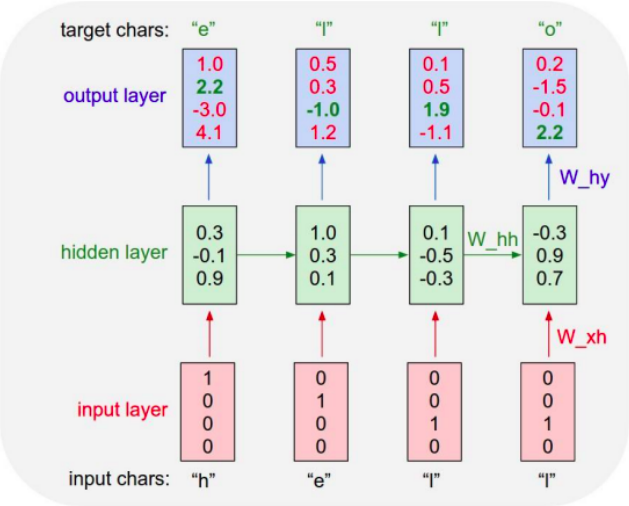
many to one과 one to many의 결합

- **Encoder many to one**, **가변 입력**을 **하나의 벡터**로 요약
- **Decoder one to many**, Encoder로부터 **하나의 벡터**를 입력받아 **가변 출력**

Example

학습

training sequence: “hello”



Vocabulary:
[h,e,l,o]

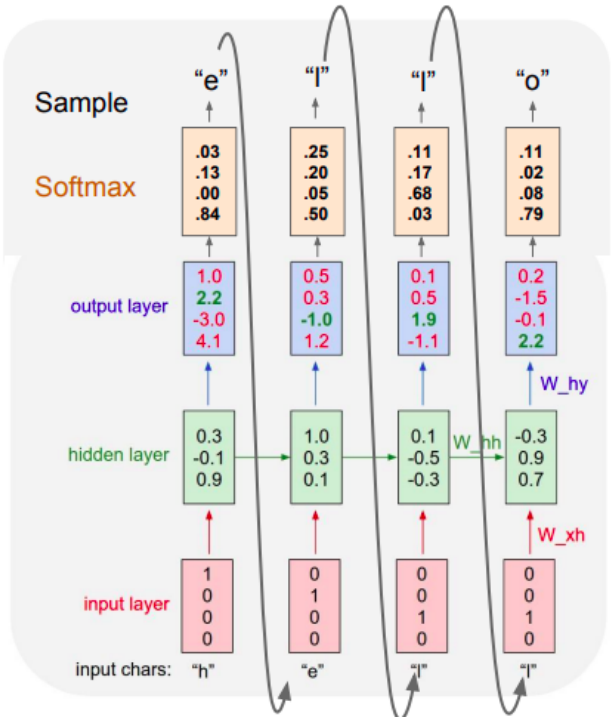
Example training sequence:
"hello"

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

▼ 시퀀스 입력해서 학습

1. `h` 입력 → `o` 출력, 틀림 `e` 가 정답
2. `e` 입력 → `o` 출력, 틀림 `l` 이 정답
3. 반복하면서 hidden state 업데이트

모델 샘플링



Vocabulary:
[h,e,l,o]

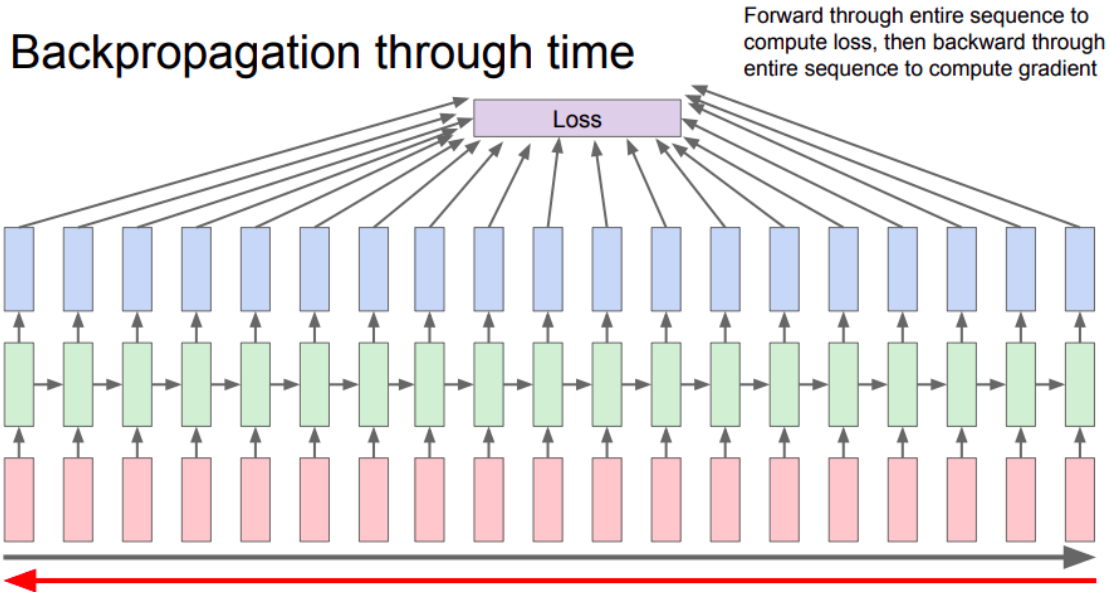
At test-time sample characters one at a time, feed back to model

▼ 모델 샘플링을 통해 학습할 때 봤을 법한 문장을 모델 스스로 생성

1. input
2. hidden layer
3. output, 모든 문자에 대한 스코어를 얻음

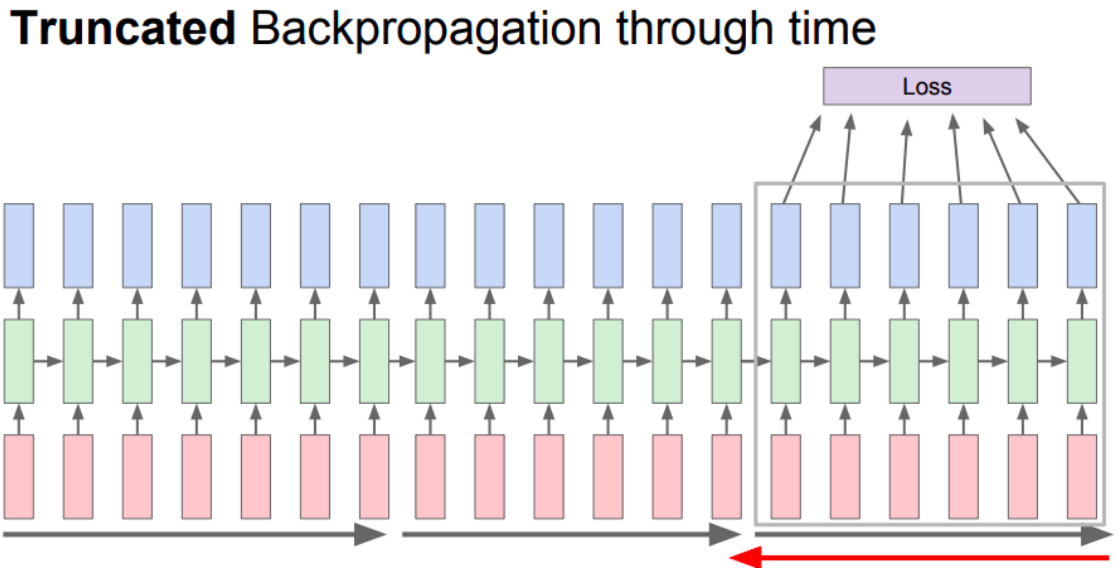
4. softmax를 거쳐 스코어를 확률분포로 표현
5. 다음 글자를 **샘플링**함
6. 뽑힌 다음 글자를 다음 스텝의 입력으로 넣어줌
7. 반복해서 새로운 문장을 생성

가장 높은 스코어를 선택하지 않고 **확률분포에서 샘플링**하는 방식은 모델에서의 **다양성**을 얻을 수 있음



전체 시퀀스가 끝날 때까지 출력값 생성, backward pass에서도 전체 시퀀스를 가지고 loss 계산

시퀀스가 **아주 길 경우** 문제, 학습 과정이 매우 느리고 메모리 사용량도 굉장이 큼

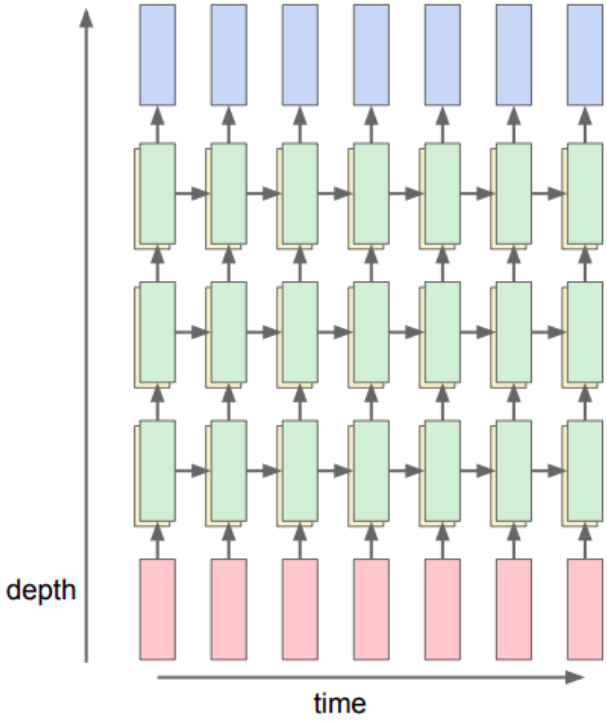


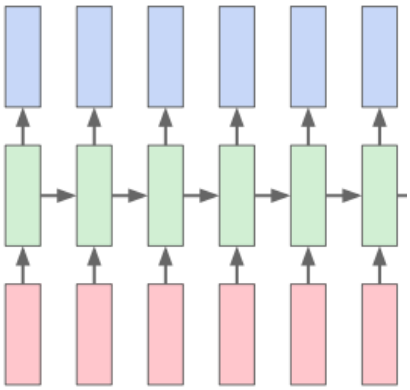
전체 스텝을 여러개의 batch사이즈로 쪼개어 학습하며 backprop를 근사 시키는 방식

한 batch 만큼만 forward pass를 하고 loss를 계산해 gradient step을 진행

- **다음 batch**의 forward pass를 진행할 때 **이전 hidden state** 이용
- **gradient step**은 **현재 batch**에서만 진행
- **backprop** 또한 **현재 batch**만큼만 진행

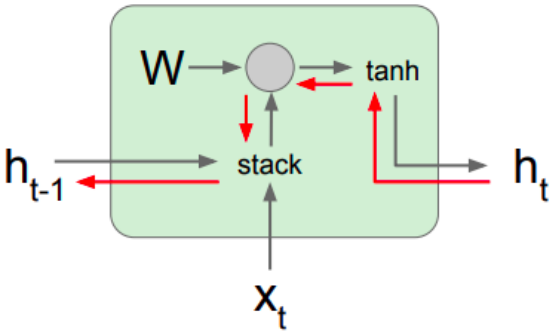
Multi-layer RNNs





단일 RNN

vanilla RNN



$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left((W_{hh} \quad W_{hx})\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

forward pass

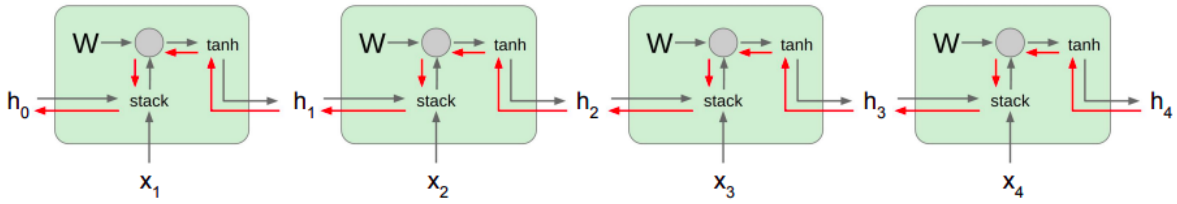
1. h_{t-1} 과 x_t 를 stack으로 쌓음
2. 가중치 행렬 W와 행렬곱 연산

3. tanh를 씌워 다음 hidden state h_t 를 도출

backward pass

1. h_t 에 대한 loss의 미분값
2. loss에 대한 h_{t-1} 의 미분값

행렬곱 gate의 backprop을 구할 때 가중치 행렬을 곱하게 됨



모든 RNN Cell을 거치는 과정에서 cell 하나를 통과할 때마다 각 cell의 가중치 행렬 W가 관여

h_0 의 그레디언트를 계산하는 식에 **아주 많은 가중치 행렬이 개입**

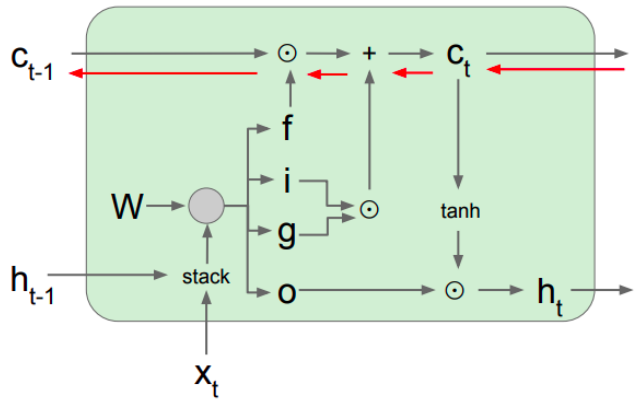
- singular value < 1: **Vanishing gradients**
- singular value > 1: **Exploding gradients**

  ▼ Gradient clipping

  그레디언트의 L2 norm이 임곗값보다 큰 경우 최대 임곗값을 넘지 못하도록 조정

LSTM

vanishing gradient 및 exploding gradient 문제를 완화하고 그레디언트가 잘 전달이 되도록 아키텍쳐 고안



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

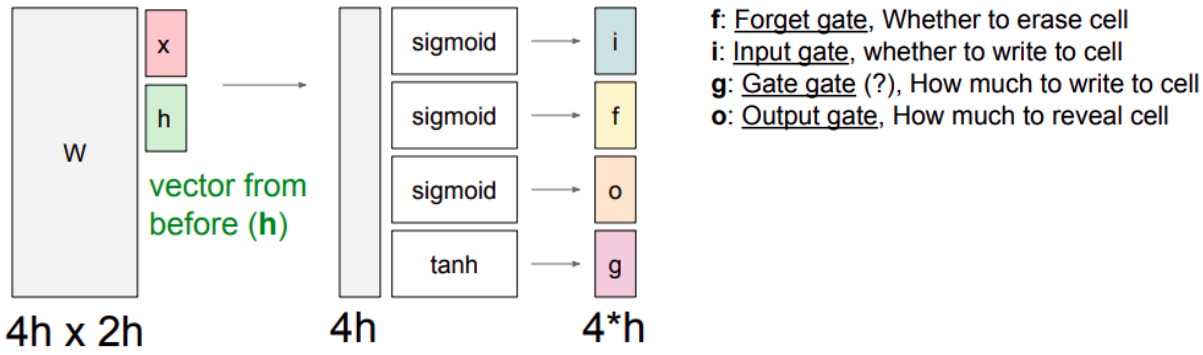
$$h_t = o \odot \tanh(c_t)$$

Cell 하나 당 **2개**의 **hidden state**

- h_t hidden state
- c_t cell state, LSTM 내부에만 존재하여 밖에 노출되지 않음

forward pass

1. h_{t-1} 와 x_t 를 stack으로 쌓음
2. 가중치 행렬 W와 행렬곱 연산 → **4개**의 **gates** 계산
3. gates 값들로 c_t 업데이트
4. c_t 로 h_t 업데이트



- `i` input gate, **입력** x_t 에 대한 가중치
- `f` forget gate, 이전 스텝의 cell의 정보를 얼마나 **망각**할 지에 대한 가중치
- `o` output gate, c_t 를 얼마나 밖에 드러낼지 대한 가중치
- `g` gate gate(?), input cell을 얼마나 포함시킬 지 결정

각 gate에서 사용하는 비선형 함수가 다름

▼ `element wise multiplication` ($\odot$) 같은 크기의 두 행렬의 각 성분을 곱하는 연산

$$\overset{G}{\begin{bmatrix}3 & 5 & 7\\ 4 & 9 & 8\end{bmatrix}} \circ \overset{H}{\begin{bmatrix}1 & 6 & 3\\ 0 & 2 & 9\end{bmatrix}} = \overset{N}{\begin{bmatrix}3\times1 & 5\times6 & 7\times3\\ 4\times0 & 9\times2 & 8\times9\end{bmatrix}}$$

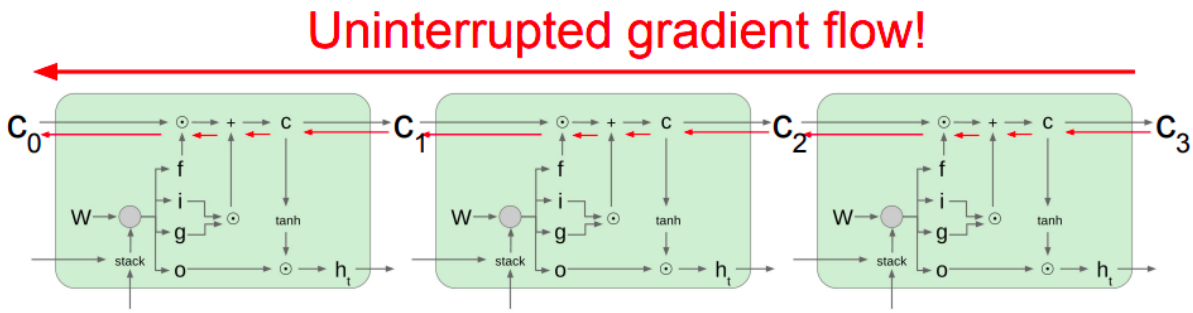
이미지 출처 https://medium.com/linear-algebra/part-14-dot-and-hadamard-product-b7e0723b9133

- $f \odot c_{t-1} + i \odot g$ 기존의 정보를 얼마나 망각하고, 얼마만큼 새로운 정보로 대체할지 결정
  - ▼ $f \odot c_{t-1}$ 이전 cell state를 계속 기억할지 말지 결정
    - $f = 0$ 이면 이전 cell state (c_t) 를 잊음
    - $f = 1$ 이면 이전 cell state를 그대로 기억
  - ▼ $i \odot g$ 새로운 정보를 얼마나 기억할지 결정
    - 해당 cell을 사용하고 싶으면 $i = 1$
    - 해당 cell 을 사용하고 싶지 않으면 $i = 0$
- $o \odot tanh(c_t)$ 최종 cell state을 조정하여 hidden state로 출력

backward pass

- **additional operation** backprop

- **element wise multiply**로 upstream gradient 직접 전달
- 그레디언트 = upstrean gradient 와 forget gate element wise 곱 → cell state backprob



forget gate가 매 스텝마다 변하므로 gradient exploding, vanishing 문제 완화

참고 https://warm-uk.tistory.com/54

이미지 출처 CS231n 2017 lecture10 slides

Q&A

Q1

34분 56초 참고: 학습된 모델에서 샘플링을 해보면 C 소스코드 스러운 것들이 나온다는데, 막상 코드들은 컴파일 되지 않는다고 나와있습니다. 그럼 학습을 계속 시킨다면 언젠간 문법에 맞는 c코드가 나올까요?

A1

예시로 보인 모델은 아직 완벽한 C언어를 구사할 수 없는, 성능이 부족한 모델로 보입니다. 학습을 무작정 여러번 시키기보다는 epoch, hidden layer 등등.. 다양한 하이퍼파라미터를 잘 조정해서 모델의 성능을 높이면 올바른 C코드를 작성하는 모델이 될 수 있을 것 같습니다.

Q2

RNN에서 gradient vanishing이 문제라면, 활성화함수를 tanh이 아닌 다른 함수를 사용해서 해결할 순 없나요? 그리고 batch normalization같은 정규화를 이용해서 RNN 학습과정을 안정화 시킬 수 없나요?

A2

1. 먼저 tanh는 (-1, 1) 범위 내의 값으로 출력이되므로 다음 hidden state의 값이나 y값의 크기가 적당한 범위 내를 유지할 수 있는데, ReLU를 사용하면 RNN Cell을 여러번 거치면서 출력값이 발산하는 문제가 발생할 수 있을 것 같습니다.

   참고 https://www.youtube.com/watch?v=8HyCNIVRbSU 4:00~4:30

2. RNN에서 **Batch Normalization**은 효율적이지 않습니다. RNN에서 Batch Normalization은 **수직적인 방향**(RNN 스택, 각각의 step에 대한 output)으로는 **적용할 수 있지만**, **수평적인 방향**(각각의 스텝들 사이)에서는 **적용할 수 없습니다**. RNN은 수평방향으로 적용 시, rescaling이 반복되면서 그레디언트가 발산하는 문제가 발생할 수 있습니다.

   RNN에서 효과적인 정규화 방법은 **Layer Normalization**가 있습니다. Batch 차원이 아닌 **Feature** 차원에서 정규화가 이루어집니다.

참고 https://cvml.tistory.com/27

참고 https://towardsdatascience.com/curse-of-batch-normalization-8e6dd20bc304

참고 https://box-world.tistory.com/73

Q3

강의 50분 쯤의 Multilayer RNN에서 입력이 첫 번째 RNN으로 들어가서 첫 번째 hidden state를 만든다고 했는데, 여기서 hidden state는 어떤 output을 만들어 내나요?

A2

첫 번째 RNN cell에서 나온 hidden state가 첫 번째 RNN cell의 output이 되고 이 hidden state는 다음 RNN cell의 input으로 들어가게 됩니다

참고
https://www.reddit.com/r/MLQuestions/comments/9hpkc4/difference_between_output_and_hidden_state_in_rnn/

Q4

전체를 다 역전파하면 시간이 오래 걸려서 Truncated Backpropagation을 사용한다고 하는데, 이렇게 부분만 역전파하는 것과 전체를 역전파하는것에 모델의 성능개선 차이가 크게 없는지(?) 궁금합니다.

A4

Backpropagation Through Time 방식에서 계산량 및 메모리 비용 문제를 개선하기 위해 Truncated Backpropagation 방식을 사용합니다. Truncated Backpropagation 방식은 backprop을 완전히 계산하는 것은 아니지만 여러개의 Batch로 쪼개서 각각의 Batch에 해당하는 만큼의 backprop을 계산해 **실제 gradient에 근사시키는 방식**입니다. 실제 gradient에 근사 시켰기 때문에 학습과정에서 gradient의 차이로 인해 모델 성능이 더 떨어지지는 않을 것이라고 생각이 됩니다. 아주 긴 data sequence가 들어왔을 때에는 계산량 및 메모리 비용을 고려하면 Truncated Backpropagation 방식을 사용하는 것이 훨씬 효율적일 것 같습니다.

참고 https://machinelearningmastery.com/gentle-introduction-backpropagation-time/