Lecture 07. Vanishing Gradients and Fancy RNNs

Ⅰ. Vanishing Gradient Problem

RNN은 기존의 backpropagation과는 살짝 다른 BPTT를 사용한다. RNN의 recurrent한 부분은 시간에 대해 펼쳐서 forward pass로 activation 값을 내고, target과의 차이로 error를 계산하고 이 error를 각 node로 편미분함으로써 backpropagation을 진행한다.

BPTT에서는 각 레이어마다 weight가 동일해야하므로 모든 update가 동일하게 이루어져야 한다. 즉, 각 레이어마다 동일한 위치의 weight에 해당하는 모든 error 미분값을 다 더한 다음, 그 값을 backpropagation하여 weight를 한 번 업데이트하는 방법이 RNN에서의 BPTT이다.

-Gradient Problem : RNN backpropagation 시 gradient가 너무 작아지거나 반대로 너무 커져서

학습이 제대로 이루어지지 않는 문제

Vanilla RNN 셀 t번째 시점의 hidden state : $\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x\boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)$

t번째 hidden state에 대한 t-1번째 hidden state의 gradient :

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = \mathrm{diag}\left(\sigma'\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x\boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right)\boldsymbol{W}_h$$

i번째 시점에서의 손실 $J^{(i)}$에 대한 $h^{(1)}$의 gradient :

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$$

$$= \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{(i-j)} \prod_{j<t\leq i} \mathrm{diag}\left(\sigma'\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x\boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right)$$

If W_h is small, then this term gets vanishingly small as *i* and *j* get further apart

Norm 성질에 의해, 다음 부등식이 성립한다.

$$\left\|\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}}\right\| \leq \left\|\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}}\right\| \|\boldsymbol{W}_h\|^{(i-j)} \prod_{j<t\leq i} \left\|\mathrm{diag}\left(\sigma'\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x\boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right)\right\|$$

$$\Rightarrow \|W_h\|_2 = \sqrt{\lambda_{max}}$$

즉, W_h matrix의 L2 norm은 W_h의 가장 큰 고유값(eigenvalue)이다.

-RNN 역전파 시 chain rule에 의해 지속적으로 곱해짐으로써 완성되는 i번째 시점에서 손실에 대한 hidden state의 gradient의 L2 norm은 절대적으로 W_h의 L2 norm의 크기에 달려있다.

-W_h의 가장 큰 고유값이 1보다 작으면, 1보다 작은 값이 계속 곱해지는 것이기 때문에 gradient가 빠르게 사라져버리는 문제가 발생 -> gradient problem

-backpropagation과 관련한 gradient 문제와 별개로 activation function의 종류에 따라 Vanishing gradient 문제가 발생하기도 한다.

Ⅱ. Why is vanishing gradient a problem?

Gradient의 값이 매우 작아지는 현상은 RNN에 아래와 같은 문제를 발생시킨다.

1. 파라미터들이 가까이 위치한 dependency에 맞게 학습을 하고 멀리 떨어진 dependency에 대한 학습을 하지 못한다. 즉, weight는 long-term effects보다 near effects에 관해 update된다.
2. gradient값이 너무 작아져서 0에 가까워져 소실되어 버리는 경우, 결과적으로 판단하기에 이 값이 정말로 미래에 과거가 영향을 미치지 않아서 gradient값이 0이 된건지, 파라미터 값이 잘못 설정되어서 gradient가 0으로 소실되어 버린건지 구분할 수 없다.

Ⅲ. Effect of vanishing gradient on RNN-LM

긴 문장이 input으로 들어왔을 때, 마지막에 올 단어가 ticket이라는 것을 첫 번째 줄의 ticket으로 유추할 수 있지만, vanishing gradient 문제로 멀리 떨어진 단어들과의 dependeny를 학습하지 못하게 되고 ticket이 아닌 가까이 있는 printer로 잘못 유추해버릴 수 있다.

LM task: *When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her* ___*tickets*___

1. Exploding gradient problem

   -gradient가 너무 크면 SGD update step이 너무 커진다.

$$\theta^{new} = \theta^{old} - \overbrace{\alpha}^{\text{learning rate}} \underbrace{\nabla_\theta J(\theta)}_{\text{gradient}}$$

이는 너무 큰 step을 야기할 수 있고 큰 loss를 가진 bad parameter cofiguration에 이른다. 나쁜 경우에, 이는 네트워크에 INF 또는 NAN을 야기한다.

2. Gradient Clipping

-gradient가 일정 thresold를 넘어가면 gradient값의 L2 norm 값으로 나눠주는 방식

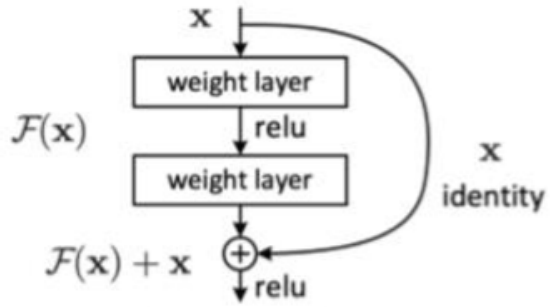
Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
end if

3. Is vanishing/exploding gradient just a RNN problem?

Feed-forward, convolutional을 포함한 모든 NN에서의 문제로, 대부분 domain에서 propagation을 할 때 점점 gradient가 작아져 lower layer에서는 update가 잘 되지 않아 학습하기 어려운 문제가 발생한다. 이를 해결하는 방법은 아래와 같다.

- ResNet(Residual connections)



Input x에 convolutional layer을 지나고 나온 결과를 더해줌으로써, 과거의 내용을 기억할 수 있도록 합니다. 과거의 학습 내용 보전+추가적으로 학습하는 정보

- DenseNet(Dense connections)

이전 layer들의 feature map을 계속해서 다음 layer의 입력과 연결하는 방식. ResNet은 feature map끼리 더하지만, DenseNet의 feature map끼리 concatenation시킴.

- HighwayNet(Highway connections)

ResNet과 비슷하다. T : transform gate, C: carry gate으로 output이 input에 대해 얼마나 변환되고 옮겨졌는지 표현함으로써 해결하는 방식이다.(LSTM에서 영감 받음)

Vanishing gradient문제는 여러 분야에서 매우 일반적이지만 특히 RNN과 같이 동일한 weight matrix를 반복적으로 곱하는 모델은 특히 불안정하므로 더욱 심각하다. 이를 해결하기 위해 제시된 모델이 LSTM, GRU이다.

IV. LSTM(Long Short-Term Memory)

RNN의 vanishing gradient문제로 발생하는 장기 의존성 문제를 해결하기 위해 RNN에서 메모리를 분리하여 따로 정보를 저장함으로써 한참 전의 데이터도 함께 고려하여 output을 만들어내는 모델.

-cell state : 이전 단계의 정보를 memory cell에 저장하여 흘려보내는 것. LSTM의 핵심적인 부분으로 input, forget, output gate들을 이용하여 정보의 반영여부를 결정한다.

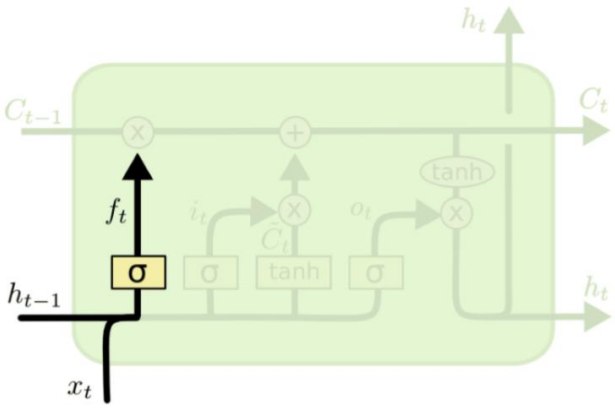
-현재 시점의 정보를 바탕으로 과거의 내용을 얼마나 잊을 지 곱해주고, 그 결과에 현재 정보를 더해서 다음 시점으로 정보를 전달하는 것

1. Forget gate layer

   -어떤 정보를 잊고 어떤 정보를 반영할지에 대한 결정을 하는 gate

   -t번째 시점에서의 x값과 t-1시점에서의 hidden state를 입력값으로 받아 sigmoid activation function을 통해 0과 1사이의 값을 출력한다.

   -출력한 값이 만약 0에 가깝게 나온다면 불필요한 정보들을 다 지워버린다는 것이고, 1에 가까울수록 이 정보에 대한 반영을 많이 한다는 의미이다.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

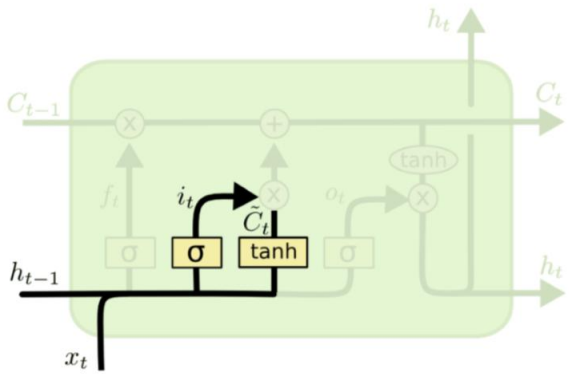
2. Input gate layer

-새로운 정보가 cell state에 저장될지를 결정하는 gate

-forget gate와 마찬가지로, input으로 h(t-1)과 x_t를 받고 두 개의 layer가 존재한다. 이 두 gate에서의 output값이 곱해짐으로써 현재의 정보를 반영할 것인지 결정한다.

-input gate : sigmoid 함수에 의해 0에서 1사이의 값으로 출력하는 부분으로 현재의 정보를 반영할 지를 결정한다.'

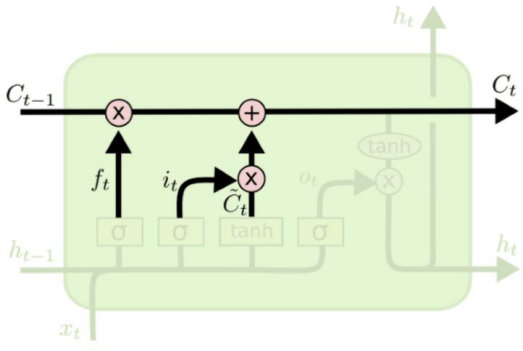
-update gate : cell state에 더해질 후보 값들의 벡터를 만드는 layer



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

3. Update Cell state

최종적으로 과거의 정보가 삭제될 것인지, 유지될 것인지 forget gate를 통해 결정하고, 현재 input값이 반영되는지 안되는지는 input gate에서 결정된다. 이 두 값이 더해져서 다음 cell state의 입력값으로 들어가게 되고 이렇게 cell state가 update된다.



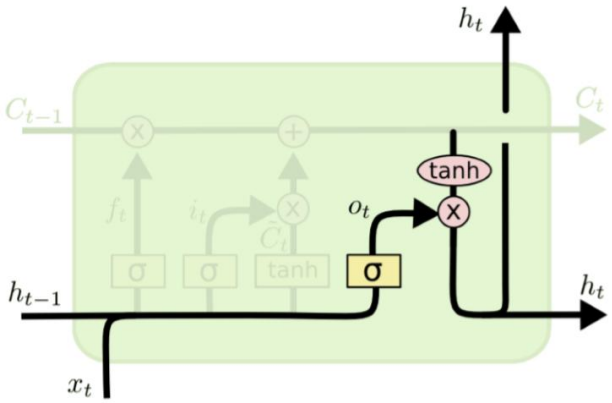
$$C_t = \underbrace{f_t * C_{t-1}}_{\text{과거}} + \underbrace{i_t * \tilde{C}_t}_{\text{현재}}$$

4. Output gate Layer

마지막으로, 출력값을 반환하는 output gate가 존재하고, 최종 output은 cell state를 바탕으로 필터링한 값이 된다.

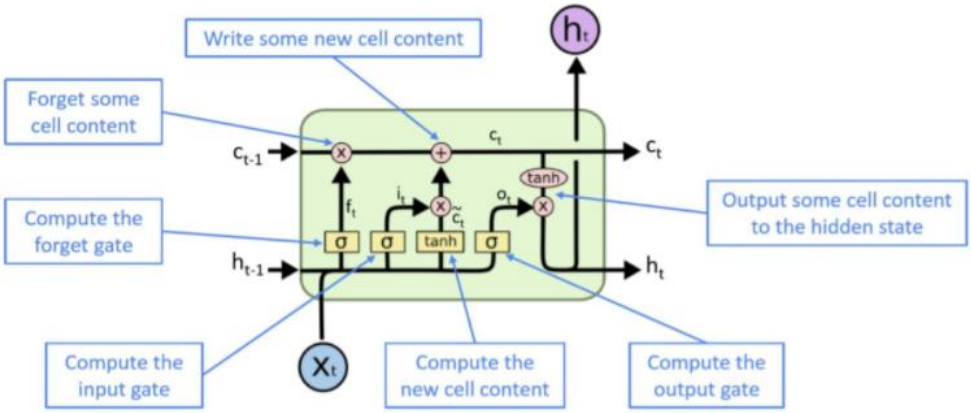
-먼저 sigmoid 함수에 input들이 들어가 0에서 1사이의 값을 출력한다.(cell state의 어느 부분을 output으로 내보낼지를 결정한다.)

-다음, cell state가 tanh에 들어가서 나온 출력값과 output gate에서 나온 값이 곱해져서 t시점에서의 hidden state가 나오게 된다.(최종 output값 & 다음 state의 input값)



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$



모든 state와 gate는 길이가 n인 벡터이고 모든 gate는 sigmoid를 통과해서 0과 1사이의 숫자로 나오고, lstm은 전 hidden state와 현재 input context를 기반으로 계산되므로 dynamic한 모델이다.

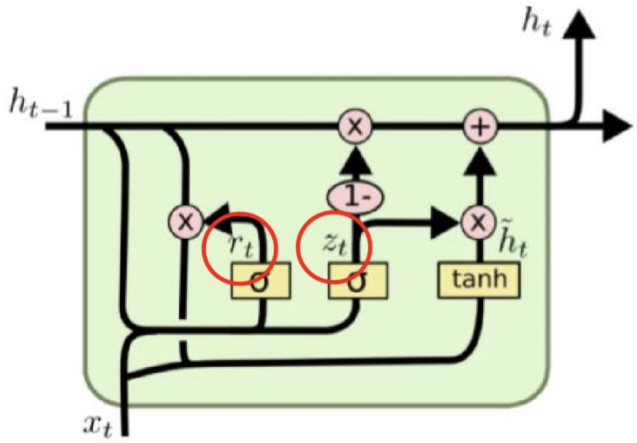
5. How does LSTM solve vanishing gradients?

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)} \qquad \boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

만약 forget gate가 1로 설정, input gate가 0으로 설정되면, t시점에서의 cell state는 이전의 정보가 완전히 보존되는 채로 hidden state를 update를 하기 때문에 cell의 정보가 완전하게 보전될 것이다.(장기 의존성 문제 해결) 하지만 LSTM도 vanishing/exploding gradient 문제가 아예 없다고 보장할 수는 없다.

V. GRU(Gated Recurrent Units)

LSTM의 강점을 가져오되, 불필요한 복잡성을 제거한 모델.



-매 time step t마다 input Xt와 hidden state ht는 있지만 cell state는 존재하지 않고 사실상 hidden state에 합쳐진다.

-LSTM처럼 gate들을 통해서 정보의 흐름을 통제하는데, update gate와 reset gate가 있다.

- **Reset gate** :

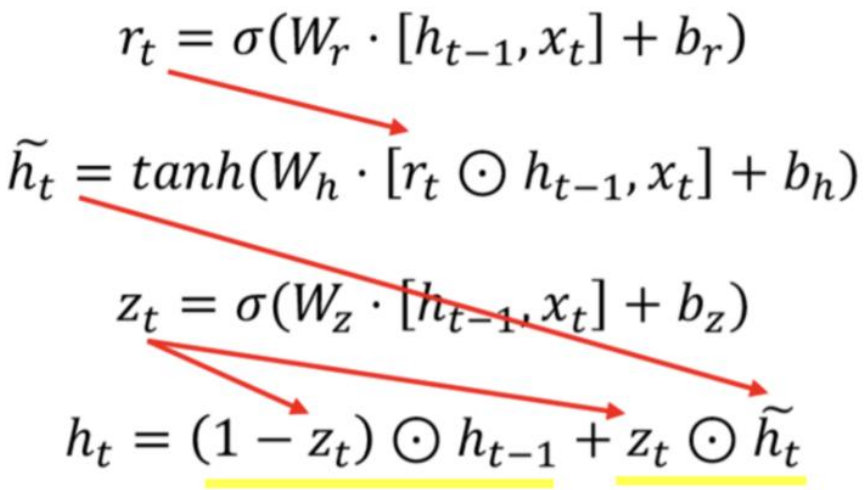
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

과거의 정보를 적당히 리셋시키는 것이 목적

- **Update gate**:

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

Forget gate + input gate



1. 먼저 reset gate를 통해 임시적인 hidden state를 만들고(직전의 hidden state에 곱함으로써 직전의 hidden state값을 그대로 이용하지 않고 reset을 해서 이 값과, 현 시점의 x값을 통해 현 시점의 candidate를 계산한다.)
2. Update gate를 통해 구한 현시점과 과거 시점의 정보양 비율을 결정하고
3. 1-zt에 이전 hidden state값을 곱하고 zt에 현시점의 candidate값을 곱해서 최종 hidden state를 계산한다.

GRU도 update gate로부터 나온 값을 0으로 설정하면 이전 hidden state gate의 값이 계속 보존된다는 의미로, gradient vanishing 문제를 어느 정도 해결했다고 볼 수 있다.

*GRU과 계산속도가 더 빠르고 더 적은 파라미터를 갖는다. LSTM을 default로 설정하고(특히 데이터가 long dependency를 가지고 있거나 training data가 많을 때) 더 효과적임을 원하면 GRU로 바꾼다.

VI. More fancy RNN variants

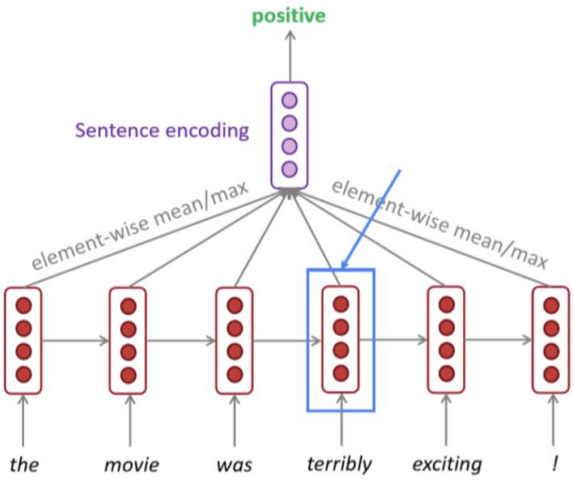
1. Bidirectional RNNs

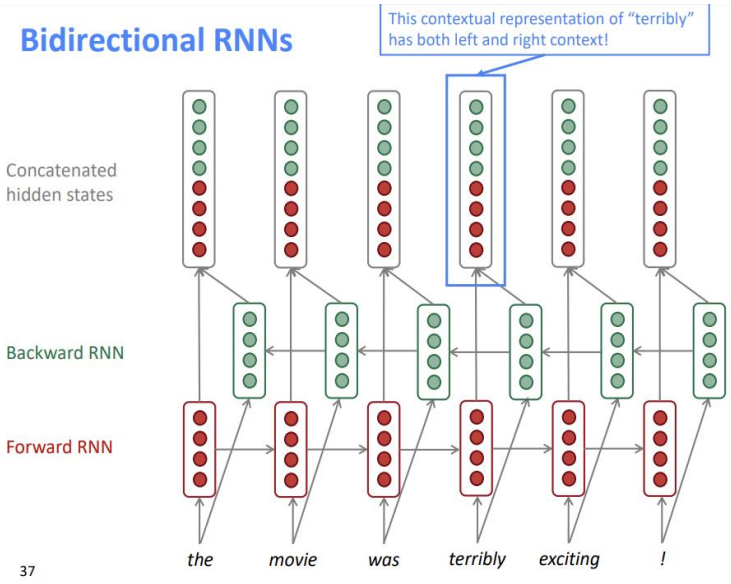
   -left, right 두 방향으로 모두 정보를 이용하기 위한 방법

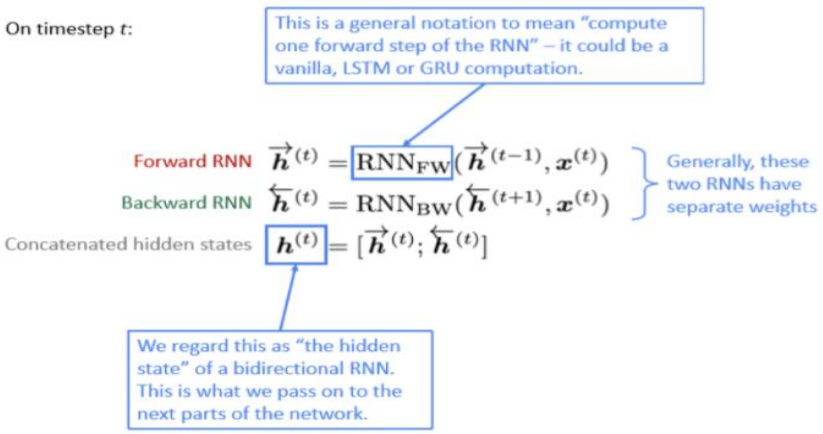
   -분리된 weight를 가지고 있는 forward RNN과 backward RNN을 학습한 후 각 hidden State를 concat해서 최종적인 representation을 형성한다.

   -forward RNN : 정방향으로 입력받아 hidden state 생성

-backward RNN : 역방향으로 입력받아 hidden state생성(두 hidden state연결해서 전체 모델의 hidden state로 사용)







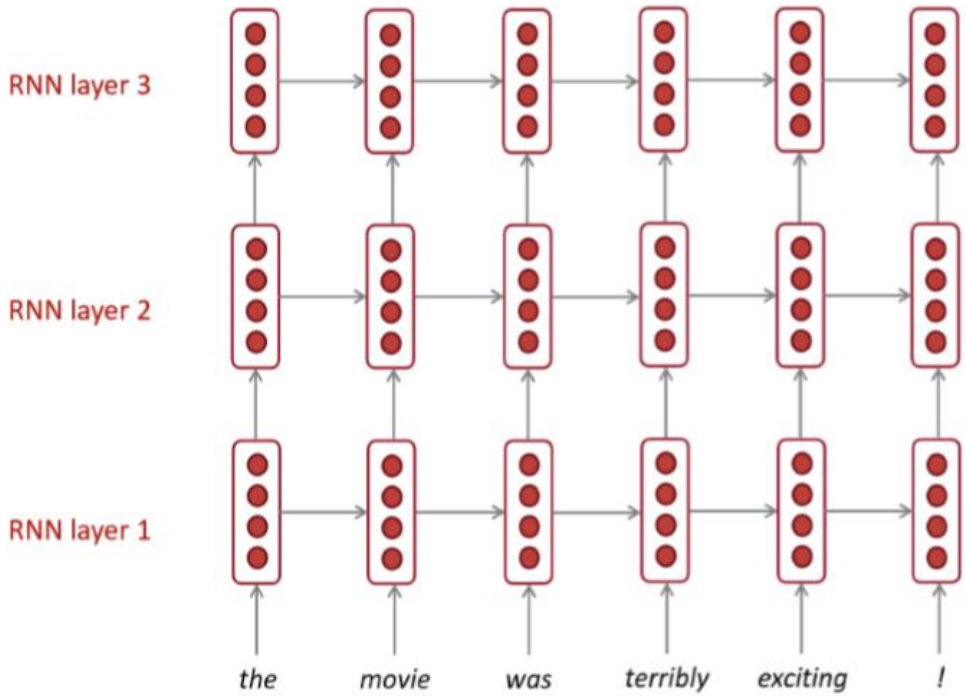
BERT(Bidirectional Encoder Representations from Transformers)가 이를 기반으로 만들어짐.

2. Multi-layer RNNs=Stacked RNNs

-RNN을 여러 층으로 사용한 모델. 성능이 좋은 RNN들은 보통 이 모델임.

-lower RNN에서는 lower level의 feature들을, higher RNN에서는 higher level의 feature들을 학습할 수 있다.

-보통 2~4개 정도의 layer를 쌓음.



-BERT 같은 Transformer-based network는 24개의 layer까지 사용할 수 있음.