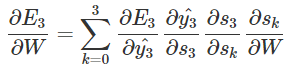
<https://aikorea.org/blog/rnn-tutorial-3/>

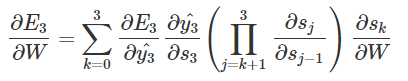
<https://jeongukjae.github.io/posts/cs224n-lecture-7-vanishing-gradients-fancy-rnns/>

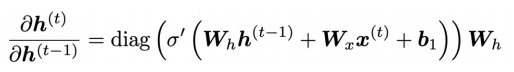
**Vanishing Gradient Problem**

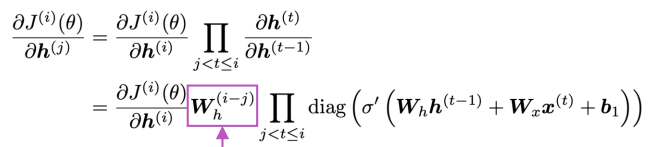
앞서 말했듯이 각 time step의 에러를 더하여 update가 진행되고 각 time step의 에러의 gradient 또한 여러 time step에 걸쳐서 chain rule의 합으로 표현된다.

(빨강) (파랑)

하지만 여기서 문제가 되는 것이 파랑 부분. 사실 더 구체적으로 생각해보면

처럼 된다.

참고로 의 값은 의 형태로 나온다. 따라서 야코비안 행렬(최종 gradient 값)에 Wh의 영향이 매우 크다.(중복적으로 곱해진다.)

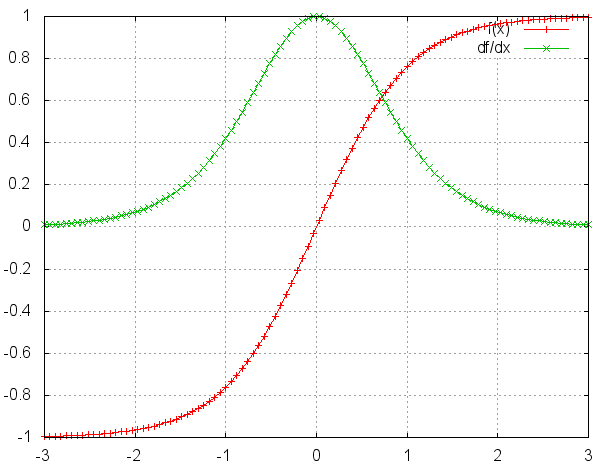


따라서 Wh의 largest eigenvlaue가 1보다 작다면 최종 gradient는 shrink exponentially, 1보다 크다면 최종 gradient는 explode exponentially

또한 추가적으로 feed-forward 시에는 tanh가 계속 곱해지니 -1~1값이 전해져서 결국에는 소실될 가능성이 크다.

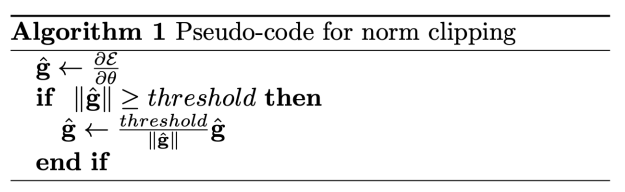
따라서 현재 time step 에서부터 엄청 초반의 time step 까지 gradient를 계속 곱한 것의 summation을 최종 gardient로 삼게된다. 하지만 위 식에서 k가 낮을수록 gradient가 계속 곱해져서 explode 또는 vanish될 수 있다. 이렇게 되면 그 iteration의 chain rule 값은 폭발 또는 소멸하게 된다. 이러면 gradient 값이 적절하지 않기 때문에 문제가 된다.

RNN에서는 주로 tanh, sigmoid를 쓴다.(ReLU는 explode 많이 된다..? 뇌피셜)



또한, tanh와 sigmoid는 다음과 같은 gradient graph를 가지는데, 이는 야코비안 행렬(dE/dW에서 위에 언급한 것처럼 Wh)(2-norm 값의 최대값은 1이라고 증명됨.)의 어느 값이 매우 작거나 크다면 그 뉴런은 포화가 되어 매우 작은 gradient를 연속적으로 곱하게 된다.

=> 이는 time step이 몇 번만 지나도 사라지고, long-range dependency를 제대로 못배운다. 반대로 gradient 값이 크다면 explode가 된다. 하지만 explode는 gradient 값이 Nan(Not a Number, 매우 큰 or 매우 작은 값) or Inf로 떠서 쉽게 알아차릴 수 있고, gradient 값이 너무 크다면 미리 정해준 적당한 값으로 자르면 된다.(gradient clipping) => 따라서 더 발견하기 힘들고 해결법이 필요한 gradient vanish의 해결이 시급했다.

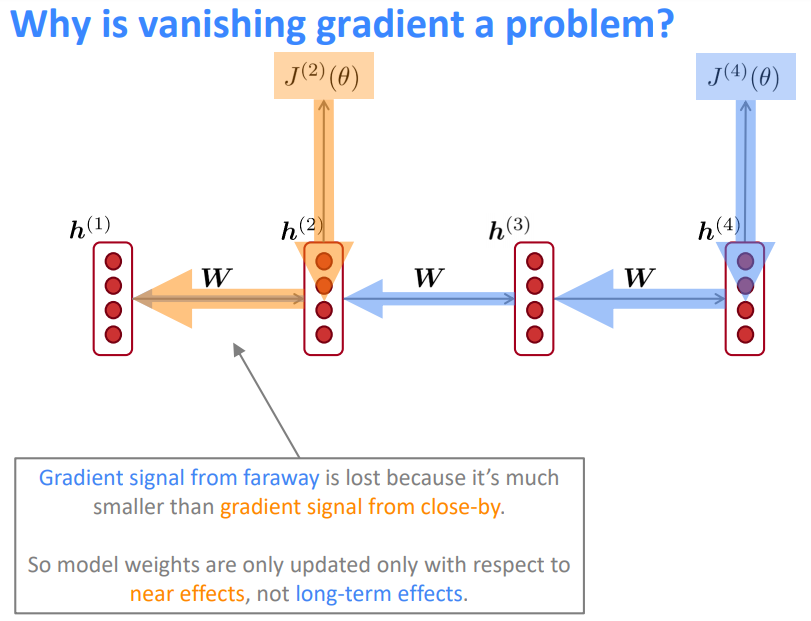


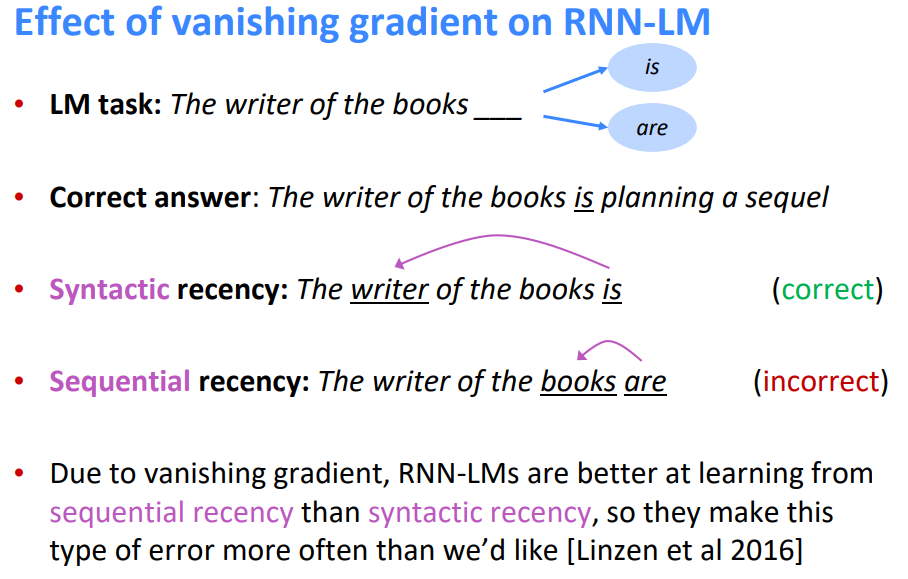
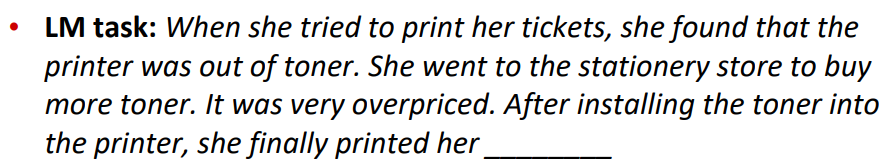
1. W를 좋은 값으로 초기화

2. Regularization을 잘 정해준다.

3. ReLU를 사용(미분값의 최대치가 로 정해져, gradient 값이 없어지는 일이 적다.)

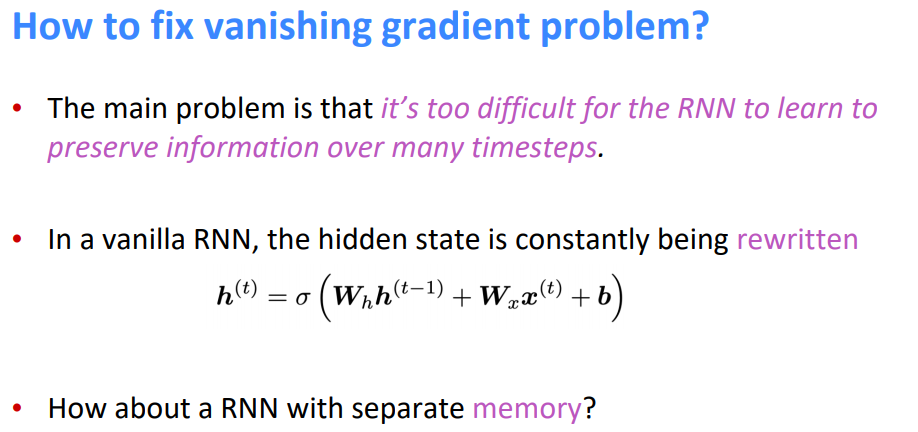
4. LSTM(Long Short-Term Memory, 1997)이나 GRU(Gated Recurrent Unit, 2014, LSTM 간략화 버전) 이용 = RNN의 GV 문제 해결 위해 디자인 / 긴 시퀀스 처리 가능





RNN의 hidden state를 계속 forward 할 때마다 rewrite 하게 된다. (특히 non-linearity function을 거치면서)

따라서 나중에 쓸 수 있도록 각 hidden state 정보를 저장하는 separate memory가 있다면? => over many timesteps, preserve information 가능



**LSTM**

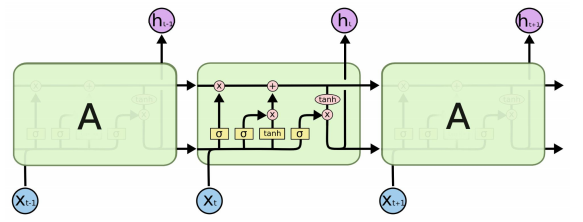
<https://ratsgo.github.io/natural%20language%20processing/2017/03/09/rnnlstm/>

Step t에서, hidden state h(t)와 cell state c(t)가 있다. (둘 다 n-length vector 이고, cell-state는 long-term information을 저장, LSTM은 erase/write/read information from cell 이 가능)

어떤 information이 erase/write/read 될지는 그에 따른 gates에 따라 선택됨. (gates 역시 n-length vector), gates의 성분 값들은 0~1 사이의 값을 가짐.(0 = closed, 1 = open)

Gates = dynamic 하다. Current context에 따라 value가 계산된다.

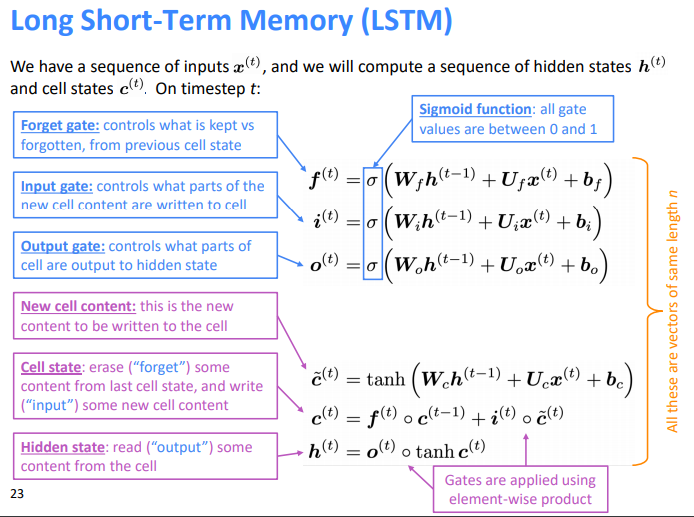
아래 그림 처럼 각 gates에 weights가 존재하며 cell-state를 정의하고, hidden state는 cell-state와 output gate로부터 정의된다.



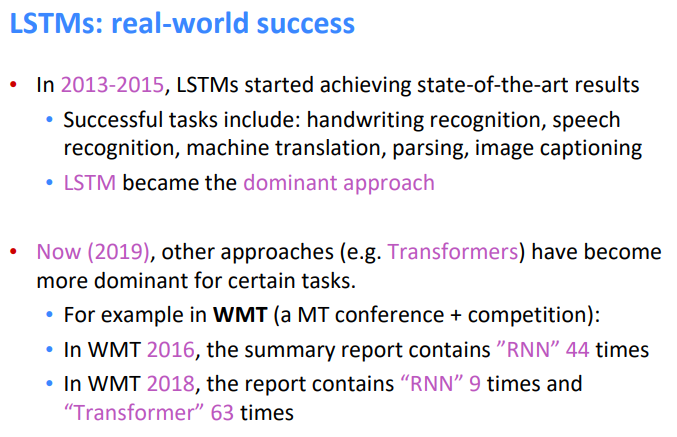
f(t)는 과거 정보를 잊기 위한 게이트 (0~1, 0이면 이전 정보를 잊고, 1이면 이전 상태를 온전히 기억)  
i(t)는 현재 정보를 기억하기 위한 게이트(0~1, 0이면 잊고, 1이면 기억하고)  
0~1의 정도는 각 게이트의 웨이트들에 따라 달라지게 되니까, 그 웨이트들을 학습하여 최적화하는것이 목표이다.

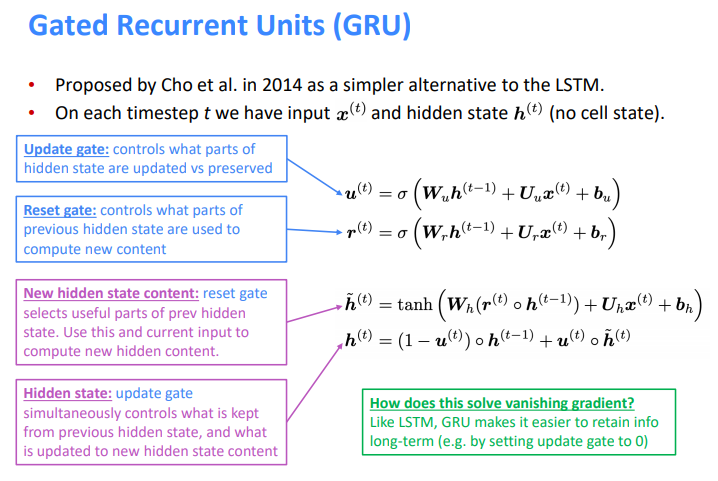
Ex) every timestep 마다, 모든것을 기억하도록 f(t)가 설정되도록 학습된다면(1에 가깝다면) 정보가 다음 cell로 전해질 것이다. 하지만 RNN은 그런 세부 조절 없이 웨이트를 온전히 공유하기 때문에 그런 조절이 힘들다. ?? 질문 그렇다면 Wf, Uf, Wi, Ui, Wo, Uo, Wc, Uc 다 공유되는 웨이트들 인가.?? = 각 유닛마다 다르다. = 공유되지 않는다.

결론적으로 long-distance dependency를 보장하도록 하며 vanish, explode gradient를 줄일 수 있다.



c(t)~ 는 이번에 새롭게 cell에 쓰여질 내용 / c(t)는 f(t), i(t), c(t)~를 사용해 계산된 마지막 new cell content / h(t)는 최종 output을 위해 o(t)와 c(t)를 이용한 hidden state

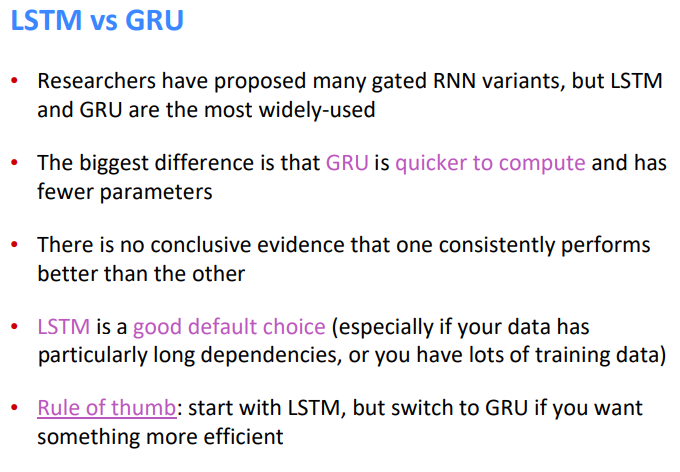




GRU는 cell state가 없다.(간략화됨)

u(t)는 이전 hidden state의 내용이 얼마나 보존되는지 (외부 보존 조절) / r(t)는 h(t)~ 계산을 위해 사용되는데 이전 hidden state으 내용을 얼마나 보존? (내부 보존 조절)

**그렇다면 LSTM vs GRU,,,??**



라고 한다.

vanishing/exploding gradient! => feed-forward, convolution net 에서도 문제이다. 특히 network가 깊어질수록 심해진다. (이유: chain rule의 연속 -> gradient 값 변동에 큰 영향(네트워크 깊다면 더 많은 chain rule) = 초반 레이어는 학습 잘 안될 수 있다. / non-linearity<sigmoid, tanh>의 선택)

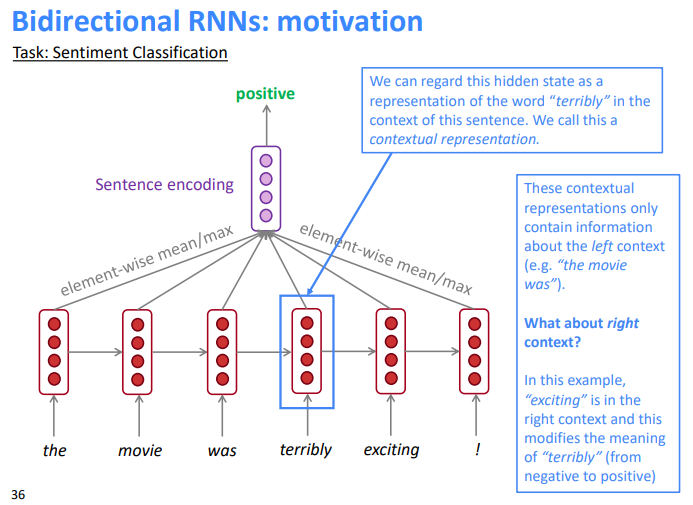
=> 따라서 ResNet 같은 경우 direct connections/Residual connections 이용 (이런 아이디어 포함된 획기적인 Network 필요 = skip-connection 이라고도 함. => identity connection은 preserve information이 layers 사이에서 가능하다.

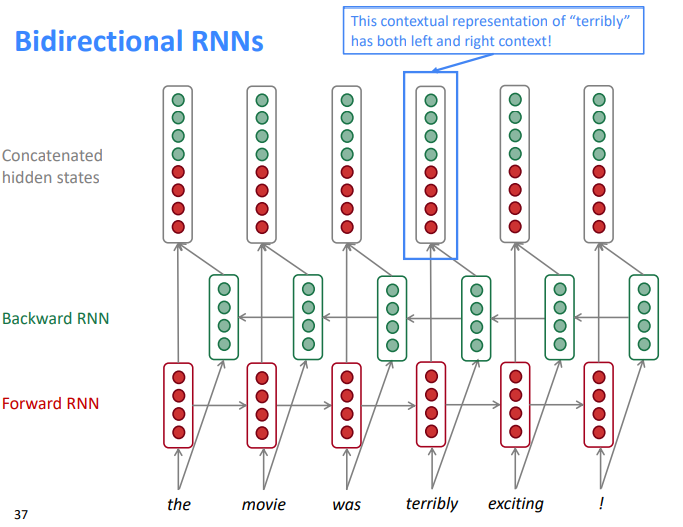
=> Dense Connection을 이용한 DenseNet 도 있다.

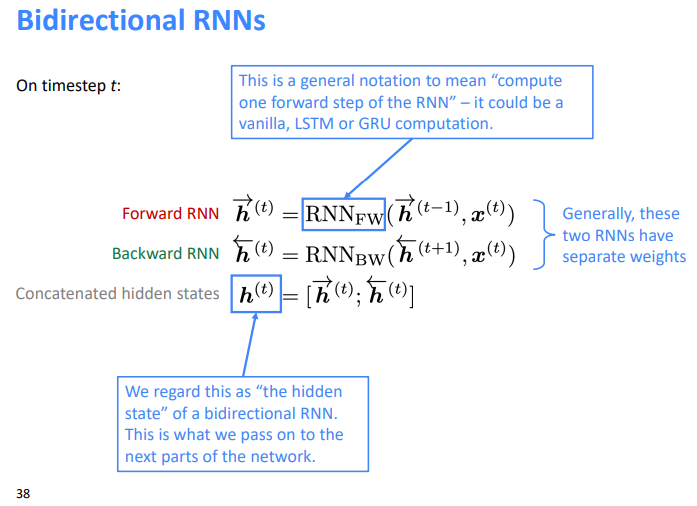
=> Highway connections “HighwayNet” / dynamic gate로 identity connection과 transformation layer가 조절된다. / LSTM에서 영감 받아 deep feedforward/convolutional net에 적용

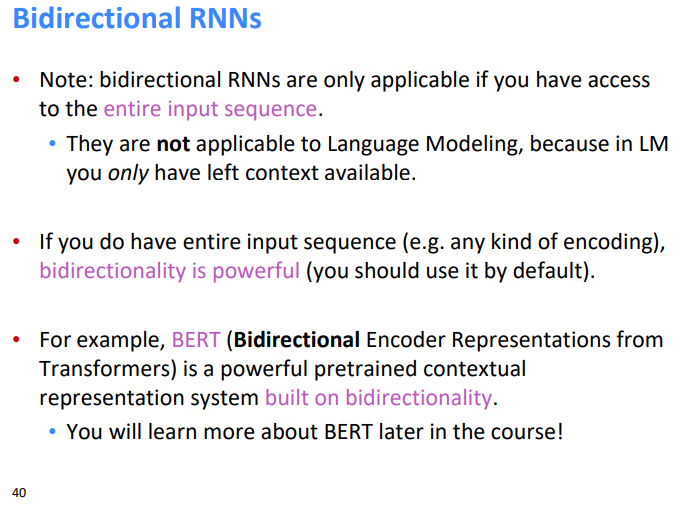
결론: 많은 모델들이 vanishing/exploding gradient를 해결하기 위해 고안되었고 특히 RNN은 repeated multiplication by the same weight matrix의 문제가 크다.

**Bidirectional RNNs**









앞서 배운 LM 모델링(빈칸 찾기) 등에는 사용할 수 없다. (entire input sequence일 때 가능)  
하지만 encoding 처럼 entire input sequence라면 강력하다.

BERT의 경우도 Bidirectional의 개념을 사용한다.

**Multi-layer RNN**

