YBIGTA 10기 노혜미, 박승리

Recurrent Neural Networks - Modeling Sequences and Stacks(RNN)

- RNN은 임의적인 크기를 가지도록 구조화된 input들을 고정된 크기의 vector로 표현할 수 있게 해준다.

The RNN Abstraction

- Input: input vectors의 정렬된 list -> $x_1, \ldots, x_n$
- output: output vectors의 정렬된 lsit->$y_1, \ldots, y_n$
- Input과 output외에도 rnn이 입출력으로 가지는 것이 있다. 바로 state다.
- state: 초기 state vector s_0, $s_1, \ldots, s_n$
- y_i는 대응하는 s_i의 결과값이다.
- x_i가 순서가 있는 형식으로 RNN에 전해지고 s_i와 y_i는 input인 $x_{1:i}$을 관측한 후의 상태를 나타낸다.
- 즉, s_i와 y_i는 x_i만이 고려된 것이 아니라 x_1부터 x_i까지 고려된 것이다.
- y_i는 그 이후의 예측을 위해 사용된다.
- RNN 기반의 언어 모델들은 n-gram 기반 모델과 비교했을 때 어려운 점이 적었다.

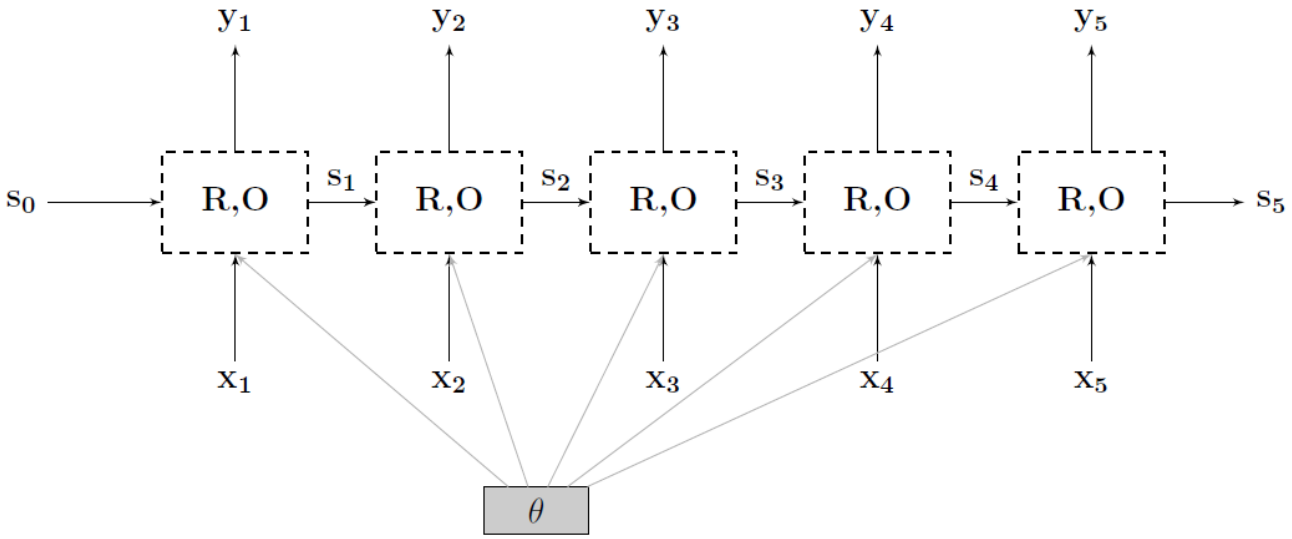
$$RNN(s_0, x_{1:n}) = s_{1:n}, y_{1:n}$$

$$s_i = R(s_{i-1}, x_i)$$

$$y_i = O(s_i)$$

$$x_i \in R^{d_{in}}, y_i \in R^{d_{out}}, si \in R^{f(d_{out})}$$

- RNN에 s_0와 $x_{1:n}$을 넣을 경우, $s_{1:n}$과 $y_{1:n}$이 결과로 얻어진다.
- 왜냐하면 s_i는 이전 상태인 s_{i-1}과 x_i을 통해 얻어지고 y_i역시 s_i를 통해 얻어지기 때문이다.
- x_i과 y_i의 차원은 input과 output의 개수이다. s_i는 논문에서 output dimension을 함수에 넣어서 얻은 값이라고 했는데, 임의의 값 정도로 생각하면 될 것 같다.



- RNN 쭉 펴본다고 했을 때, 위와 같은 구조를 지닌다.
- 위에서 언급했던 대로, s_{i-1}과 x_i가 input으로 들어가 s_i가 나오고 또 s_i가 들어가 y_i가 나오는 것을 알 수 있다.
- θ는 보통 시각화하지 않는 건데 굳이 넣은 이유는 동일한 parameter가 모든 time step동안 공유되는 것을 강조하기 위해서이다.

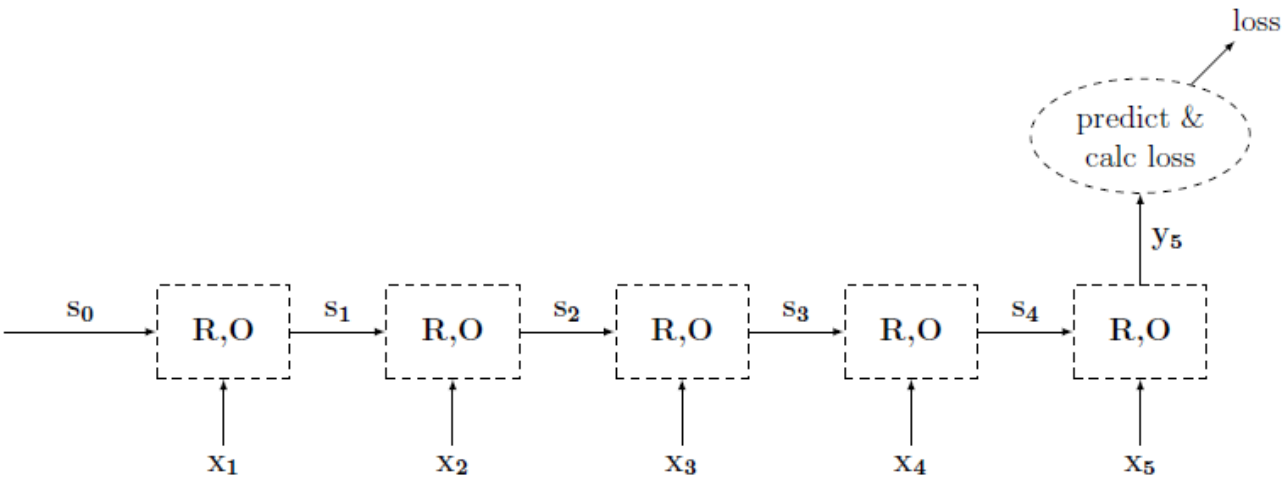
$$
\begin{aligned}
\mathbf{s_4} &= R(\mathbf{s_3}, \mathbf{x_4}) \\
&= R(\overbrace{R(\mathbf{s_2}, \mathbf{x_3})}^{\mathbf{s_3}}, \mathbf{x_4}) \\
&= R(R(\overbrace{R(\mathbf{s_1}, \mathbf{x_2})}^{\mathbf{s_2}}, \mathbf{x_3}), \mathbf{x_4}) \\
&= R(R(R(\overbrace{R(\mathbf{s_0}, \mathbf{x_1})}^{\mathbf{s_1}}, \mathbf{x_2}), \mathbf{x_3}), \mathbf{x_4})
\end{aligned}
$$

- i=4인 경우만 보더라도, s_4는 전체 input $x_{1:4}$를 모두 고려한다는 것을 알 수 있다. 즉, s_i는 모든 input $x_{1:i}$를 고려한다.
- 그래서 y_n뿐만 아니라 s_n도 전체 input 순서를 encoding*하는 것으로 생각할 수 있다.

*순서를 encoding한다는 것은 순서를 특정한 값으로 해석해준다고 생각하면 될 것 같다. 예를 들면 "안녕하세요. 저는 누구예요"라는 문장의 순서를 encoding한다면 "안녕하세요"는 제일 먼저 나오니 1, "저는 누구예요"는 그 다음에 나오니 2 이런 식으로 되는 것이다. (이 예시에서 encoding이 어떤 것인지 느낌만 받길 바란다.)

RNN Training

Acceptor



- 한 가지 option은 supervision* signal을 오직 final output vector에 대해base로 하는 것이다.

*supervision은 학습을 할 때, 자신이 예측해야 할 것이 무언인지를 알고 학습하는 것이다.

- 이런 관점에서 RNN은 acceptor이다.
- final state가 어떤지 보고 결과를 결정한다.
- 예를 들면, 단어에서 character 하나하나를 읽는 RNN이 해당 단어의 발화의 일부분*을 예측하는데 final state를 이용하는 것이 포함된다. 또한 문장을 읽는 RNN이 final state에 기반해서 해당 문장의 감정이 긍정을 의미하는지 부정을 의미하는지 예측하거나 타당한 명사구인지 판단하는 것들이 이에 속한다.

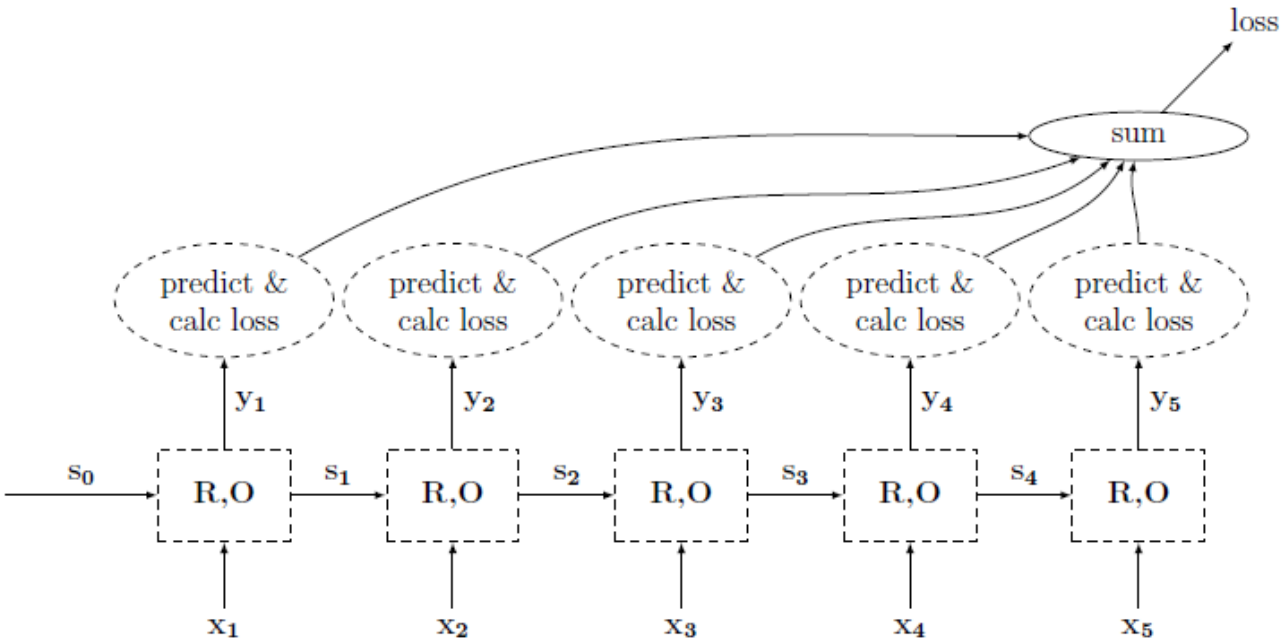
*만약 'hello'라는 음성을 잘게 쪼개 예측하면 hhheeellloo가 될 것이다. 이때 각각의 부분들을 예측하는데 RNN을 사용한다고 생각하면 된다.

- 이러한 경우에서 loss는 $y_n = O(s_n)$의 관점에서 정의된다. 그리고 error gradients는 나머지 순서를 통해 backpropagate 될 것이다.

Encoder

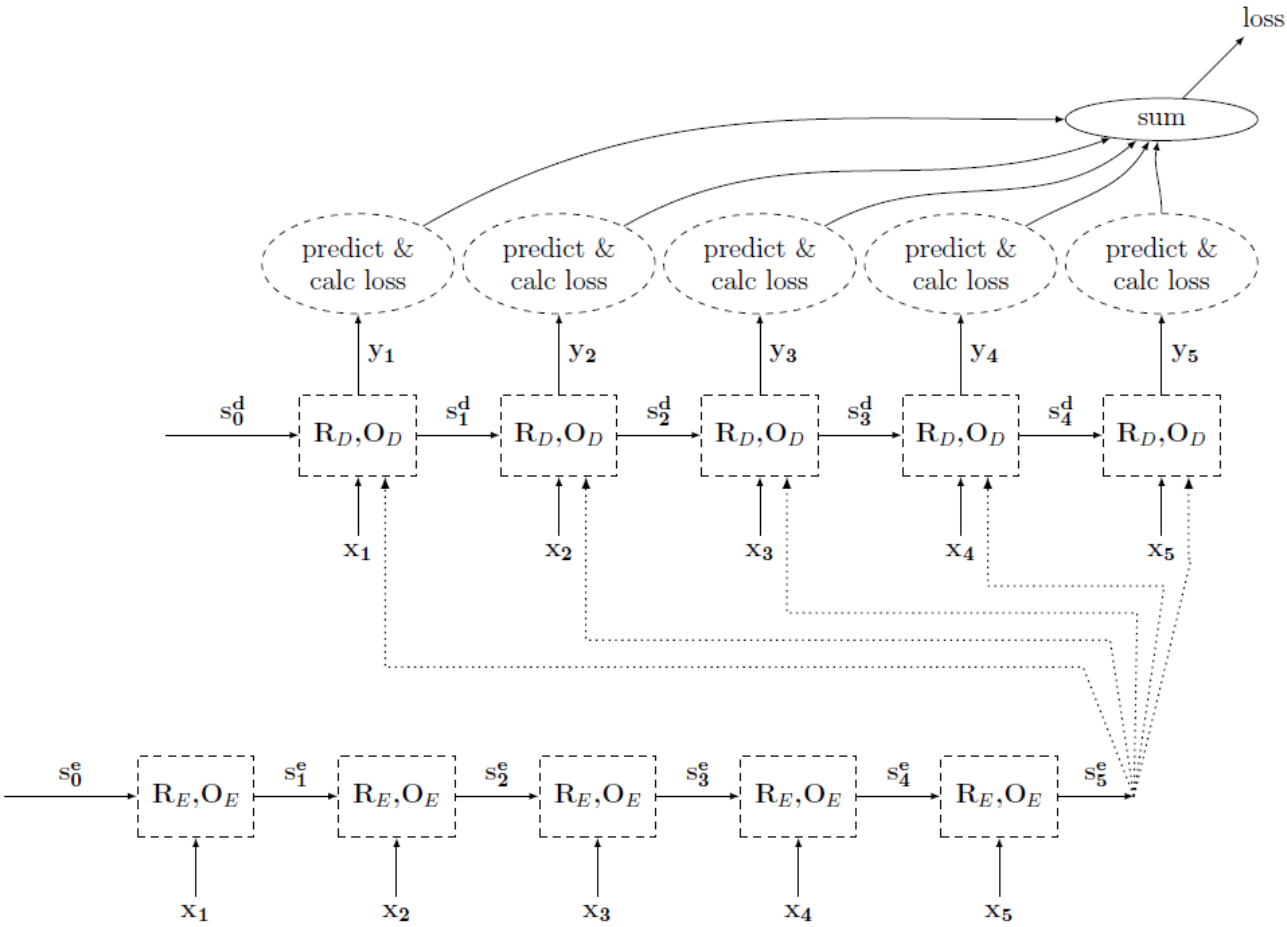
- acceptor의 경우와 비슷하게, encoder supervision에서 final output vector, y_n만을 이용한다. 즉, 예측하고자 하는 값을 얻을 때 마지막 output vector만 쓴다는 뜻이다.
- 하지만 예측이 오로지 final vector에서만 이루어지는 acceptor와 달리, encoder에서 final vector는 sequence 정보를 encoding하는 것으로 취급하고 다른 signal과 함께 추가적인 정보로써 사용된다. 즉, 예측을 하는데 final vector만을 사용하는 것이 아니라 final vector외에도 다른 것들을 이용한다는 것이다.
- 예를 들면, extractive document summarization system은 처음에 RNN을 통해 전체 문서를 요약하는 vector y_n을 얻을 것이다. 그 다음, y_n은 요약하는 데 포함될 문장들을 고르기 위해서 다른 feature들과 함께 사용될 것이다.

Transducer



- 각각의 input에 대한 output을 만드는 것이다.
- 실제 label y_i를 기반으로 각각의 output $\hat{y}_i$에 대한 local loss signal을 계산할 수 있다.
- 총 loss는 단순히 local loss를 합할 수도 있고 평균을 낼 수도 있다.
- 한 가지 예는 sequence tagger이다. 첫번째부터 i번째 단어를 기반으로 i번째 단어의 tag 할당을 예측하는데 y_i를 input으로 생각한다. 즉, 각각의 단어에 대한 tagging을 하는 것이다.
- transduction setup을 흔히 쓰는 경우는 언어 모델링에서이다. 첫번째부터 i번째 단어들의 순서는 i+1번째 단어의 분포를 예측하는데 사용된다.

Encoder - Decoder

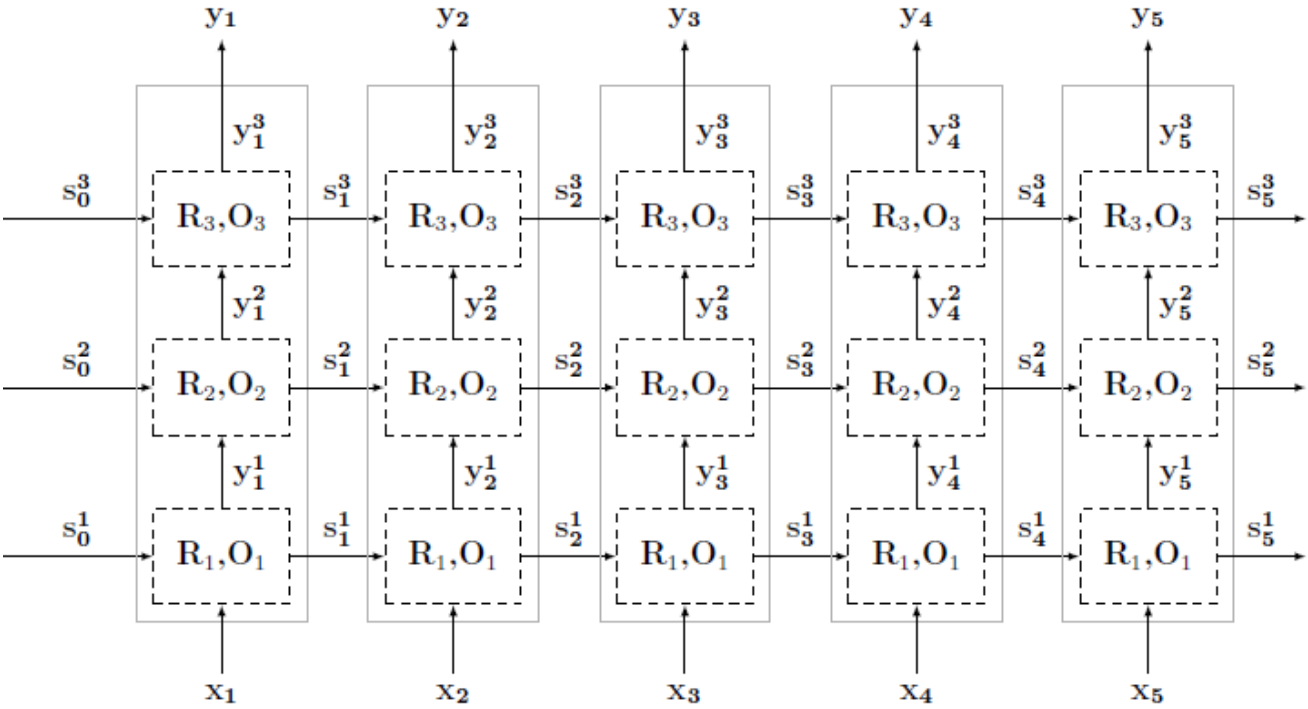


- RNN이 sequence를 vector representation y_n으로 encoding하는데 사용되고, y_n은 decoder로서 사용되는 또 다른 RNN의 보조적인 input으로 사용된다.
- 예를 들면, 기계 번역에서 첫번째 RNN은 source sentence를 vector representation인 y_n으로 encoding하고, 이 state vector가 decoder 역할을 하는 RNN으로 들어간다. 이때 decoder RNN은 y_n뿐만 아니라 이전에 예측된 단어들을 기반으로 타겟 언어 문장의 단어들을 예측하도록 학습된다. *supervision은 decoder RNN에서만 일어나지만, gradient는 encoder RNN까지 backprop된다.

*쉽게 말하면, "안녕"->"Hi"로 번역할 때 "안녕"을vector로 encoding한 다음 그 vector를 decoding해 "Hi"라는 문장을 얻는 것이다.

- 효율을 높이는 방법은 source sentence를 반대로 encoder에 넣는 것이다. n개의 단어중에 x_n을 문장의 첫번째 단어로 대응시키는 것이다. 이렇게 할 경우 decoder RNN이 source sentence의 첫번째 단어와 targer sentence의 첫번째 단어의 관계를 만들기가 더 쉽다고 한다.
- 또 다른 사용 예시는 sequence transduction이다. 첫번째부터 n번째까지의 tag를 만들기 위해, encoder RNN은 문장을 고정된 크기의 vector로 encoding하는데 사용된다. 이 vector는 또 다른 transducer RNN에 초기 state vector로 넘겨진다. 즉, i번째의 tag label을 예측하는데 $x_{i:n}$과 함께 사용된다.
- 이러한 접근은 삭제에 의한 문장 요약 모델에 사용된다.

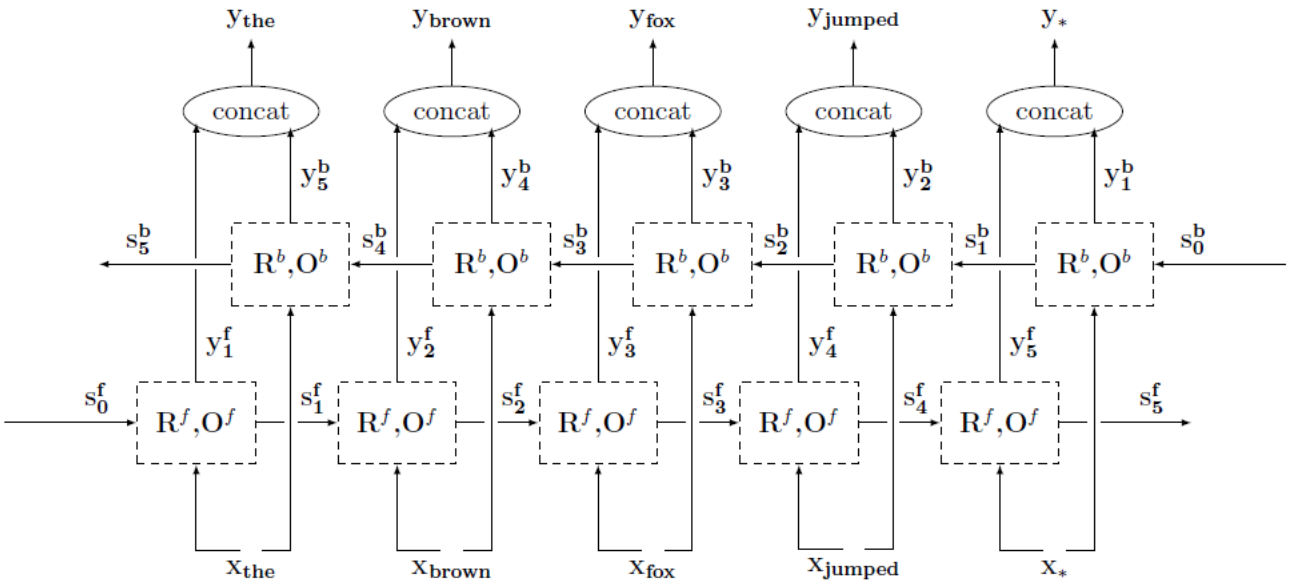
Multi-layer (stacked) RNNs



- RNN은 layer로 쌓일 수 있다. k개의 RNNs를 생각해보자. j번째 RNN은 states $s_{1:n}^j$을 가지고 output $y_{1:n}^j$를 가진다. 첫번째 RNN에 대한 input은 $x_{1:n}$이다. j번째 RNN에 대한 input은 이전, 즉 j-1번째 RNN의 output이므로 $y_{1:n}^{j-1}$이다. 전체 formation에 대한 output은 마지막 RNN의 output인 $y_{1:n}^k$이다. 그러한 layered 구조를 보통 *deep RNNs*라고 부른다.

  *위첨자는 몇 번재 RNN인지 아래첨자는 해당 RNN에서 몇 번째 input, state, output인지를 나타낸다.
- 구체적으로 어떻게 좋은 지는 잘모르지만, deep RNNs이 얕은 RNN보다 더 성능이 좋았다. 특히 4-layers deep architecture는 encoder-decoder frame work를 사용하는 기계 번역에서 좋은 성능을 냈다.

BI-RNN



(front와 back의 방향을 유심히 봐보자.)

- i번째 단어를 예측하는데 과거의 단어 $x_{1:i}$도 유용하지만, 다음의 단어인 $x_{i:n}$도 유용한다. 이러한 사실을 반영한 것이 BI-RNN이다.
- focus 단어가 해당 단어를 둘러싸고 있는 k개의 단어 window를 기반으로 카테고리화되는 sliding-window approach를 통해 분명해졌다.* 고정된 window 크기를 통해 focus 단어의 이전의 단어와 이후의 단어를 고려하는 것이 가능해졌다.

*https://github.com/YBIGTA/Deep_learning/blob/master/RNN/nlp/%EA%B5%AC%ED%98%84/skip-gram%20with%20tensorflow.md

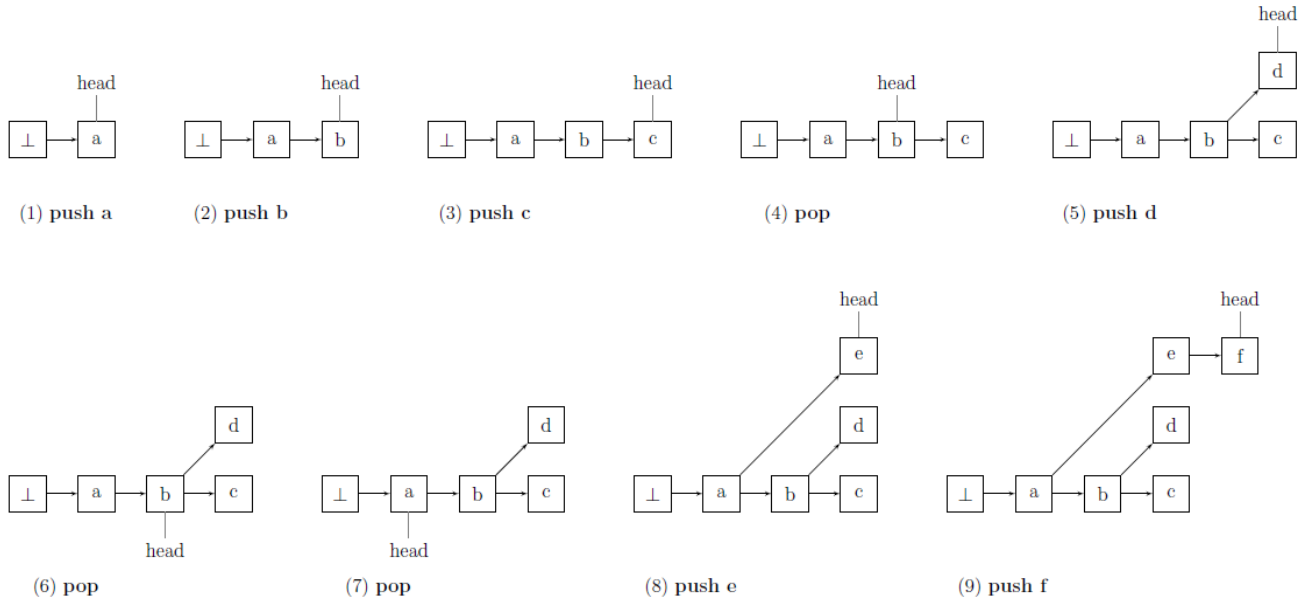
window개념이 궁금하다면 해당 글을 훑어보자.

- 두 개의 분리된 상태, s_i^f와s_i^b를 유지하면서 BI-RNN은 작동한다.
- forward state인 s_i^f는 $x_1, x_2, \ldots, x_i$를 기반으로 하고, backward state인 s_i^b는 $x_n, x_{n-1}, \ldots, x_i$를 기반으로 한다. 이 두 상태는 서로 다른 RNNs에 의해 만들어진다. 첫번째 RNN(R^f, O^f)는 input sequence로 $x_{1:n}$을 받고, 두번째 RNN(R^b, O^b)는 input sequence로 $x_{n:1}$을 받는다. s_i는 forward와 backward states로 구성된다.
- i번째에서의 output은 두 개의 output vector의 concatenation(연결)에 기반해서 만들어진다. $y_i = [y_i^f; y_i^b] = [O^f(s_i^f); O^b(s_i^b)]$가 된다. 즉, 과거와 미래 모두를 고려하는 것이다.*

*R과 O가 뭔지 헷갈린다면 위부분을 다시 한번 보자.

- y_i는 예측에 곧바로 사용될 수도 있고 더 복잡한 네트워크의 input으로 주어질 수도 있다.
- 두 개의 RNN은 서로 독립적이지만, i번째에서의 error gradient는 두 개의 RNN을 통해 앞 뒤로 전달될 것이다.

RNNs for Representing Stacks

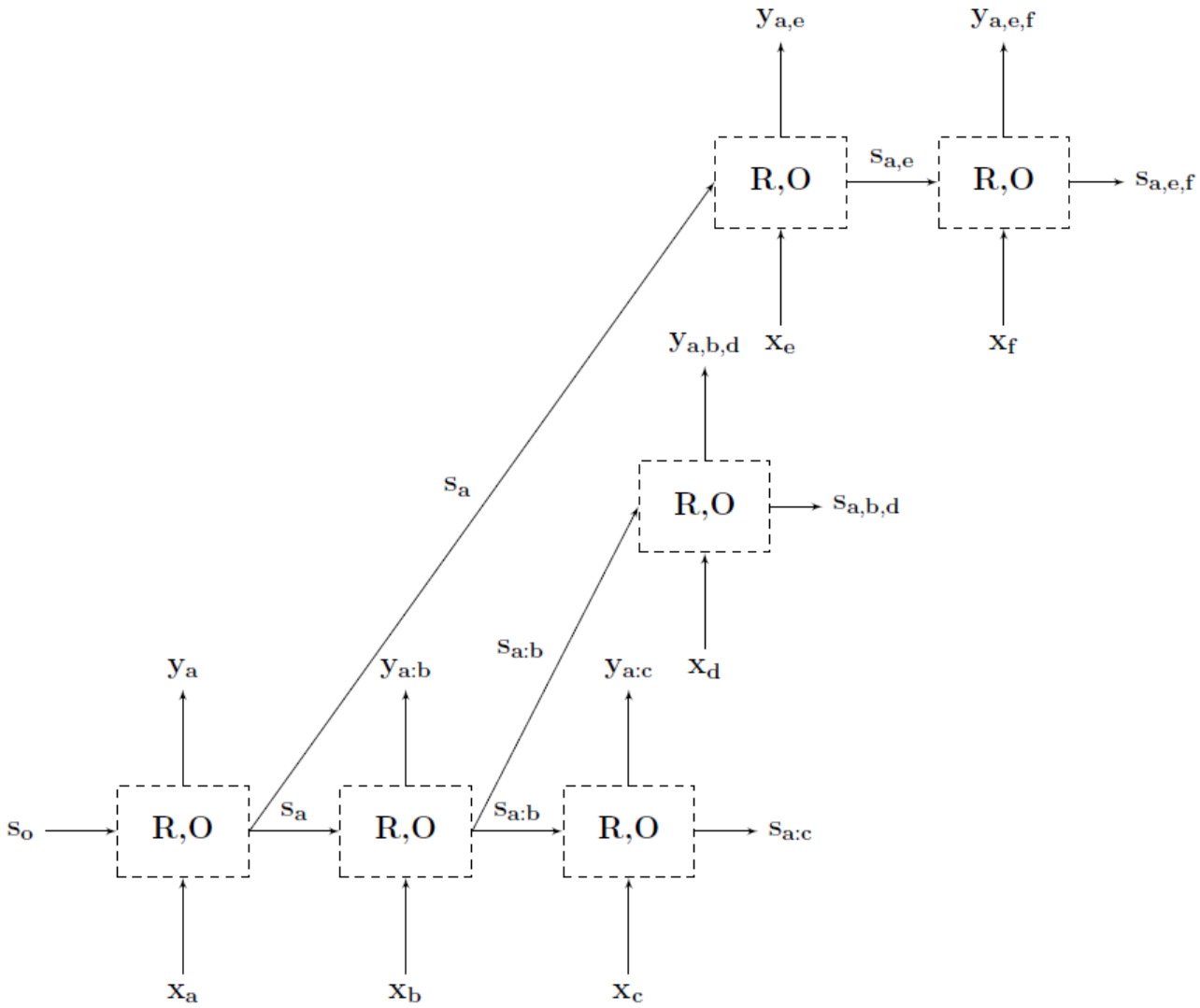


- stack-RNN을 이해하기 전에 stack을 이해해보자.
- 지속적이고 불변적인 자료 구조는 자료가 수정됐을 때도 원래의 상태를 유지할 수 있다.
- persistent stack 구조는 linked list의 head를 가리키는 pointer로서 stack을 사용한다.
- push를 통해 list에 element를 추가하면 새로운 head가 반환된다.
- pop을 통해 원래의 list는 온전하게 하면서 head의 parent가 반환된다.*

*이전의 element가 head가 된다는 뜻이다.

- 이러한 과정을 반복하면 결국 chain structure가 아닌 tree structure가 된다.

- 주어진 nobe로부터 error를 backpropagating하는 것은 stack에 있는 모든 element에 영향을 끼칠 것이다.
- 이러한 stack 구조를 반영한 게 바로 stack-RNN이다.



(state와 y의 아래첨자를 유심히 보자.)

Reference

Yoav Goldberg(2015), A Primer on Neural Network Models
for Natural Language Processing

http://u.cs.biu.ac.il/~yogo/nnlp.pdf