Deep Learning for NLP
Momory

https://youtu.be/uQ61qZK4_1M

Deep Learning for NLP:
Memory

I 고려대학교 산업경영공학과
I 서덕성

INDEX

- Transduction Bottleneck
- RNN revisited
- Attention (Read Only Memory)
  - Early fusion / Late fusion
- Register Machine (Random Access Memory)
- Stacks: Neural PDAs (Pushdown Automata)

기계번역에서 bootleneck 문제와 long sequence를 제한된 벡터에 담을 경우 한계가 되는 문제를 해결하기 위해 Momory라는 개념을 도입한 방법론에 대한 설명

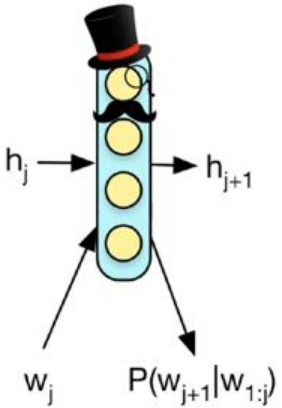
RNN을 살펴보고, 해결방안에 대해서 세가지 모델을 살펴보자.

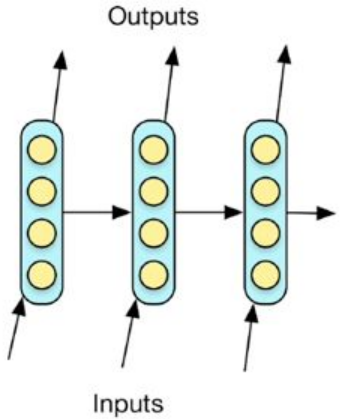
Transduction Bottleneck

01 02 03 04 05

❖ RNN

- 이전의 내 정보들과 연결된 구조
- 개념적으로 sequence의 history를 모형화
- Feedforward nets를 shared weight를 이용해 unfold
- Long(ish) range dependencies를 고려할 수 있음
- Recurrent hidden layer는 다음 label의 분포를 산출함





Transduction Bottleneck

01

❖ RNN

02

- Applications

03

  - Language modeling

$$P(s) = \prod_{t \in (1,T)} P(s_t | s_1, \dots, s_{t-1})$$

04

  - Sequence labelling

$$P(l_t | s_1, \dots, s_t)$$

05

  - Sentence classification

$$P(l|s)$$

- 물론, RNN보다 더 쉽게 접근하는 방식들도 있음

RNN은 다음과 같은 어플리케이션 등에 사용되고 있다.

Transduction Bottleneck

01 ❖ Transduction with conditional models

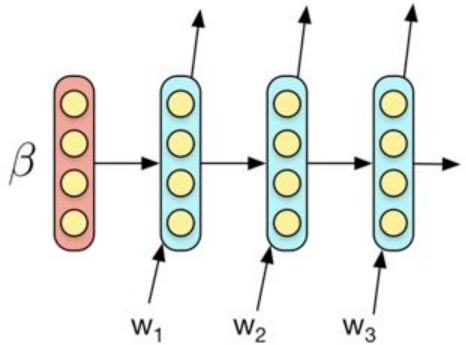
02 ▪ Encoding 된 것으로부터 Decoding 진행 (기존 RNN구조)

03 • Encoding 정보가 처음에는 큰 영향을 줄 수 있지만, 뒤로 갈수록 작은 영향을 주는 구조

04

05

$$P(\mathbf{w}|\beta)$$



기계번역

인코딩 정보를 축약한 베타가 주어졌을때 W를 통해서 Decoding을 진행

Transduction Bottleneck

01 ❖ Transduction with conditional models

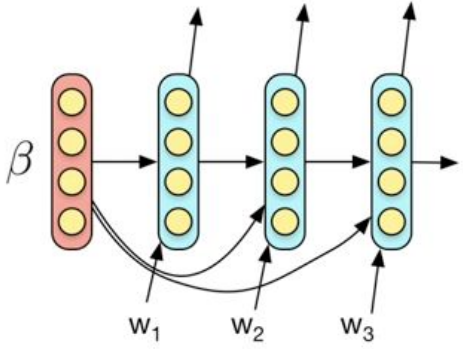
02 ▪ Encoding 된 것으로부터 Decoding 진행 (Attention)

03 • 꾸준히 Decoding하는 부분에 영향을 주는 구조

04

05

$$P(\mathbf{w}|\beta)$$



Attention 추가

베타에서 필요한 정보를 w1, w2, w3에서 선택적으로 받아서 다음 단어를 창출하는 모델을 만들 수 있다.

Transduction Bottleneck

01

❖ Seq2Seq Mapping with RNNs

02

- Source seq.로부터 Target seq.로 transform하는 것이 목표

03

  - Source seq.

$$\boldsymbol{s} = s_1, s_2, \dots, s_m$$

04

  - Target seq.

$$\boldsymbol{t} = t_1, t_2, \dots, t_n$$

05

  - Model

$$P(t_{i+1} | t_1 \dots t_i; \boldsymbol{s})$$

- Source seq.를 encoding하고, 다음 시점에서의 target의 분포를 산출
- MLE를 이용해서 모형 학습
- 데이터 형태: $s_1\ s_2\ \dots s_m$ ||| $\mathrm{t}_1 \mathrm{t}_2 \dots \mathrm{t}_n$
- ||| 전까지의 output은 당연히 무시

Source seq = encoding

Targert seq = decoding

Source의 정보를 포함한 다음에 타켓에 대한 정보를 이전타켓까지만 확인하고, 다음 타켓에 대한 확률 분포를 구한다.

LE를 최대화 방식을 이용하여 학습

Transduction Bottleneck

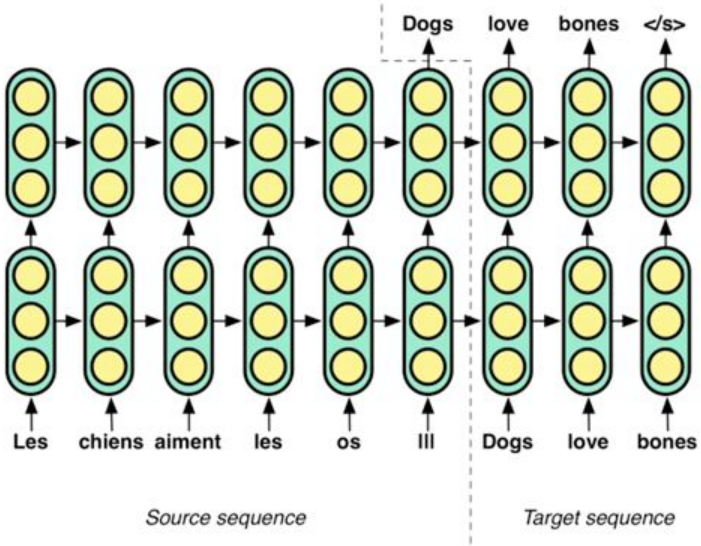
01 ❖ Deep LSTMs for Translation

02 ▪ $P(some\ english\ |du\ francais)$

03

04

05



(Sutskever et al. NIPS 2014)

기계번역 예시1

Transduction Bottleneck

01 ❖ Deep LSTMs for Translation

02 ▪ $P(numerical\ result \mid simple\ python\ scripts)$

03 • Character by character

04

05

Input:

```
j=8584
for x in range(8):
  j+=920
b=(1500+j)
print((b+7567))
```

Target: 25011.

Input:

```
i=8827
c=(i-5347)
print((c+8704) if 2641<8500 else 5308)
```

Target: 12184.

(Zaremba and Sutskever, 2014)

예시 2

파이썬 캐릭터를 입력으로 주어진다.

출력으로 수치적 값으로 나온다.

?

Transduction Bottleneck

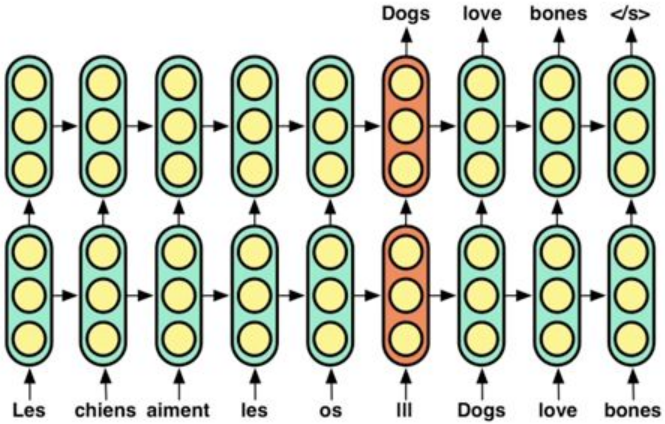
01 ❖ The Bottleneck for Simple RNNs

02 ▪ 한계점

03 • Source 부분이 길어질수록 제한된 길이의 vector에 많은 정보를 담는다는 것이 한계가 있을 것

04 • Target source 부분이 먼저 학습이 진행되어, encoding 파트에서는 Gradient-starved한 경우 발생

05



▪ 강의에서 진행할 내용

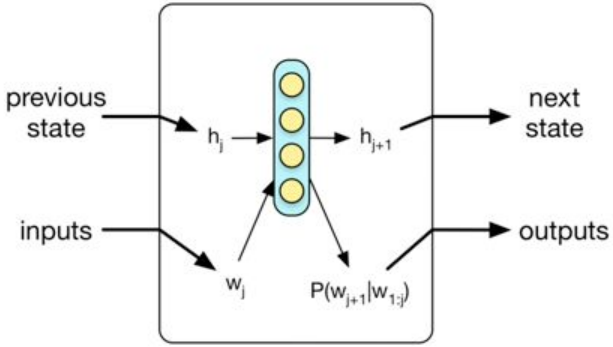
• 제한된 길이의 vector가 아니고 유동적으로 접근해보자

유동적 : 메모리를 이용하자.

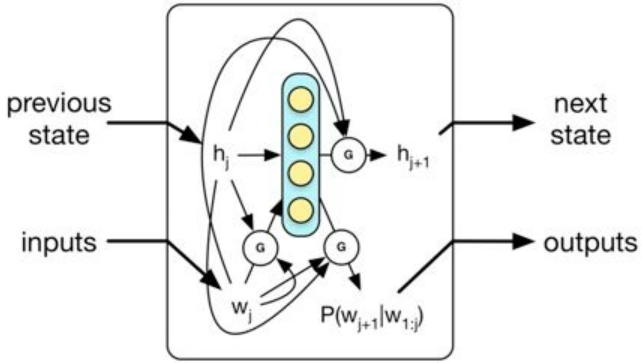
RNN revisited

01 02 03 04 05

❖ RNNs



❖ Cells (LSTM, GRU, ...)



◆ Modeling a function

- ✓ Input: $x_t \in X$
- ✓ Output: $y_t \in Y$
- ✓ Previous state: $p_t \in P$
- ✓ Updated state: $n_t \in N$
- ✓ Typically P=N
- ✓ RNN: $X * P \rightarrow Y * N$
- ✓ gradient 이용 학습

용어 정리

RNN revisited

01

02

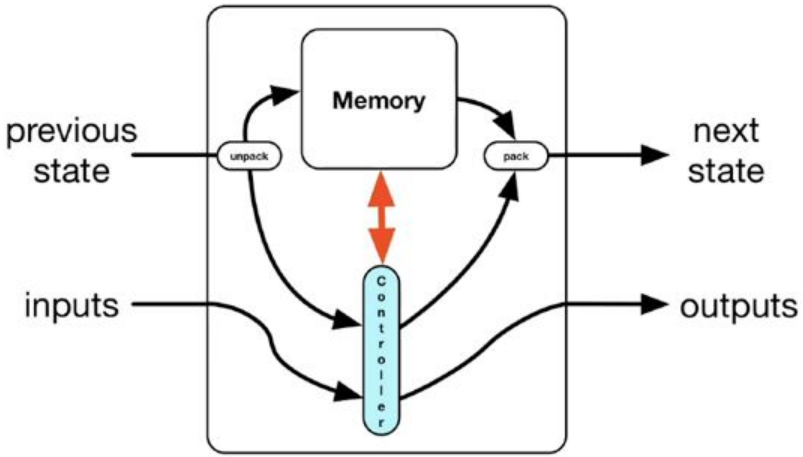
03

04

05

❖ The Controller-Memory Split

- 미분가능성을 유지한 채로, cell을 변형시킴 (nest / stack / sequenced)
- Memory 파트와 Controller 파트로 나누어서 바라보는 관점을 소개
  - Memory: 정보 저장 공간
  - Controller: seq.를 다루는 RNN 모형



미분가능성 = 학습

인코딩된 벡터를 메모리에 담아둔다.

디코딩쪽에 seq를 다루는 RNN 모형

Attention: ROM

01
02
03
04
05

❖ Attention

- 기존에 배운 Attention과 유사
- 앞서 출현한 정보에서 특정 부분을 중점적으로 바라보고 출력을 결정
- 본 강의에서는 Attention을 Memory라는 파트에 걸어 줌
- Controller는 Input / Output logic을 담당

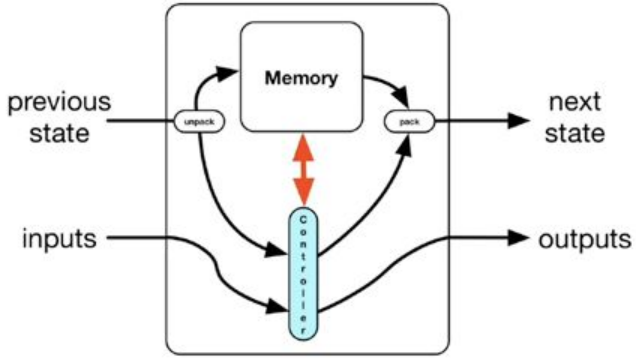
Read Only Memory

Attention: ROM

01
02
03
04
05

❖ ROM for encoder-decoder models

- Encoder는 array of representations (Memory)를 만듦
- Decoder는 controller + memory model로, memory를 attention matrix로 이용
- Memory에 대한 gradient가 encoder에 대한 gradient로 작용하여 encoder의 gradient starved가 완화됨
- Large seq.에 대한 information도 다룰 수 있음



문제를 해결하기 위한 메모리에 참조목록을 만든다.

메모리: encoding정보가 함축되어 있다.

그라디언트가 인코딩 파트로 가면서 소실되는데, 메모리가 학습되면은 인코딩 정보가 학습되는 효과

긴 정보의 베타를 메모리에 스태킹형태로 보관하기 때문에 large seq 용이

Controller: RNN

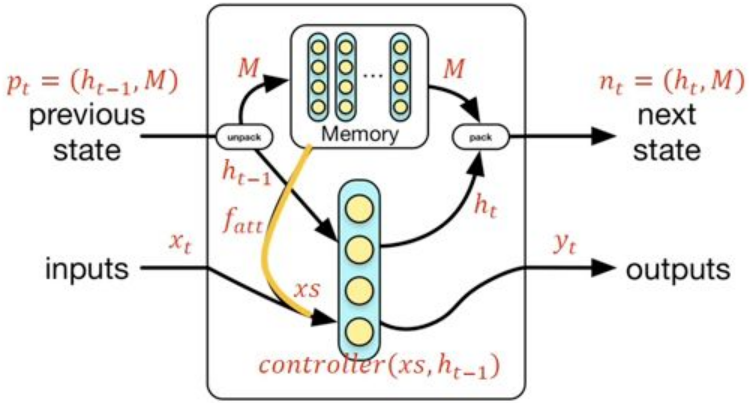
Attention: ROM

01 02 03 04 05

❖ Early Fusion

- RNN: $X * P \rightarrow Y * N$
  - $p_t = (h_{t-1}, M)$ $M: memory$
  - $n_t = (h_t, M)$
  - $xs = x_t \oplus f_{att}(h_{t-1}, M)$
  - $y_t, h_t = controller(xs, h_{t-1})$

e.g. $f_{att}(h, M) = M * softmax(h * W_{att} * M)$

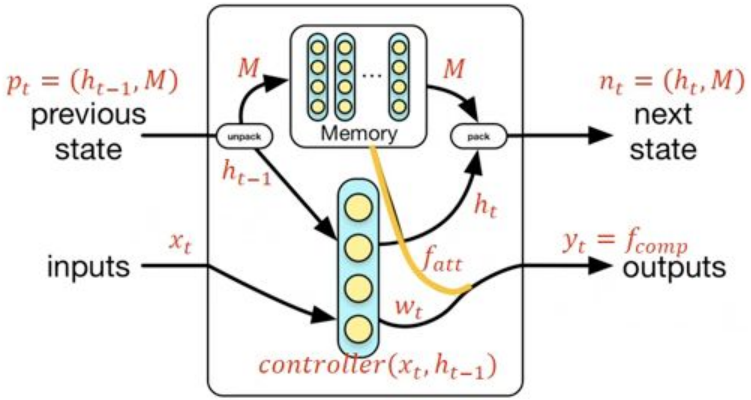


Attention: ROM

01 02 03 04 05

❖ Late Fusion

- RNN: $X * P \rightarrow Y * N$
  - $p_t = (h_{t-1}, M)$ $M: memory$
  - $n_t = (h_t, M)$
  - $w_t, h_t = controller(x_t, h_{t-1})$
  - $y_t = f_{comp}(w_t, f_{att}(h_t, M))$

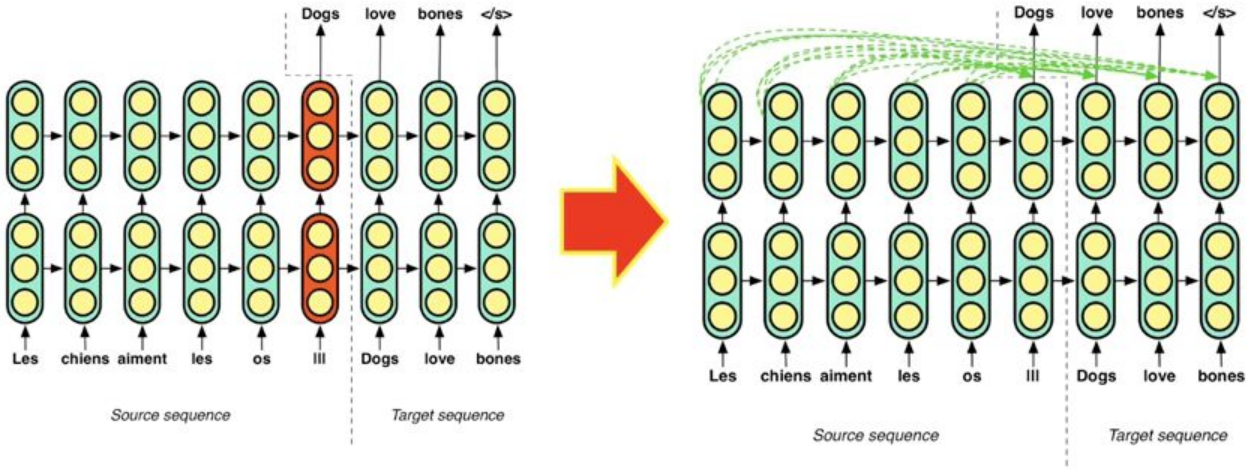


Late fusion

Attention: ROM

01
02
03
04
05

❖ Skipping the bottleneck



Dogs를 뽑아낼때
메모리를 참조한다.

Attention: ROM

01
02
03
04
05

REASONING ABOUT ENTAILMENT WITH NEURAL ATTENTION

Tim Rocktäschel
University College London
t.rocktaschel@cs.ucl.ac.uk

Edward Grefenstette & Karl Moritz Hermann
Google DeepMind
{etg,kmh}@google.com

Tomáš Kočiský & Phil Blunsom
Google DeepMind & University of Oxford
{tkocisky,pblunsom}@google.com

❖ Recognizing Textual Entailment (RTE)

- Premise(전제)와 hypothesis(가설)이 주어지고 둘의 관계를 분류
- Class: Contradiction / Neutral / Entailment

예) A man is crowd surfing at a concert

The man is at a football game	Contradiction
The man is drunk	Neutral
The man is at a concert	Entailment

텍스트 함의 인식

http://l2r.cs.uiuc.edu/~danr/Teaching/CS546-13/TeChapter.pdf

ROM 예시 논문

“A man is crowd surfing at a concert” 전제

“The ma is at a football game”

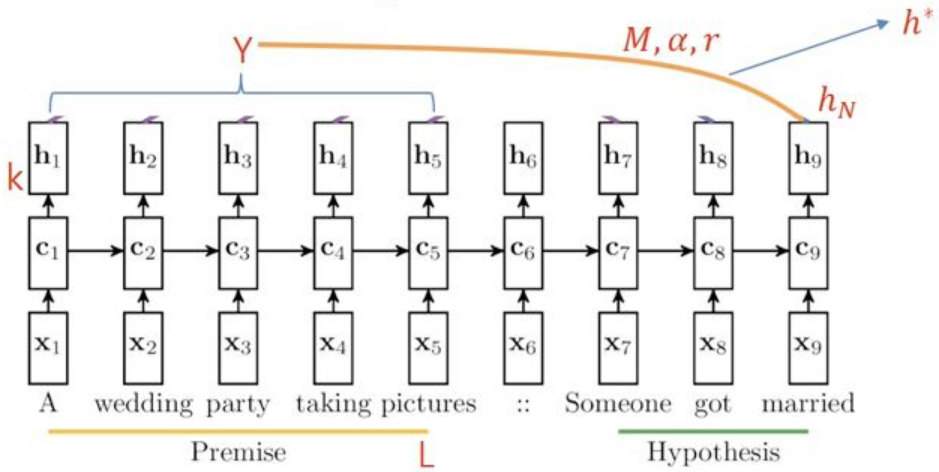
가설

Contradiction으로 분류

Attention: ROM

❖ Recognizing Textual Entailment (RTE) _ attention

- 2개의 LSTM을 사용 (제일 마지막 h에만 attention을 걸음)
  - $Y \in R^{k*L}$: k차원의 hidden vector를 L개 모음(premise encoding)
  - h_N: Premise, hypothesis를 모두 통과한 최종 output
  - $M = \tanh\left(W^y Y + W^h h_N \otimes e_L\right)$ $\quad M \in R^{k*L}$
  - $\alpha = softmax(w^T M)$ $\quad \alpha \in R^L$
  - $r = Y\alpha^T$ $\quad r \in R^k$
  - $h^* = \tanh(W^p r + W^x h_N)$ $\quad h^* \in R^k$



텍스트 함의 인식

M ; 메모리

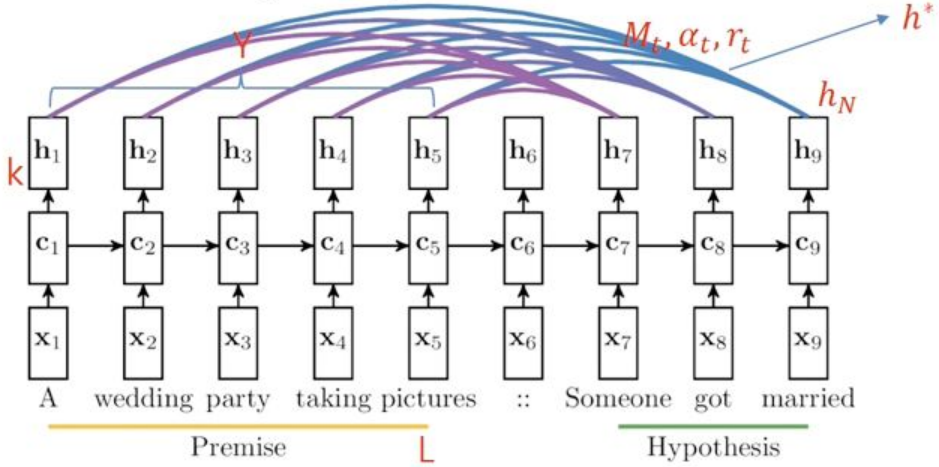
알파: 어텐션 정도(Y의 어떤 데이터를 중요하게 볼것인가?)

r : 메모리에서 어떤 Y를 중점적으로 볼것인지?

Attention: ROM

❖ Recognizing Textual Entailment (RTE) _ word-by-word

- 2개의 LSTM을 사용 (제일 마지막 h에 걸릴 attention을 점차 모음)
  - $Y \in R^{k*L}$: k차원의 hidden vector를 L개 모음(premise encoding)
  - h_N: Premise, hypothesis를 모두 통과한 최종 output
  - $M_t = \tanh\left(W^y Y + [W^h h_t + W^r r_{t-1}] \otimes e_L\right)$ $\quad M \in R^{k*L}$
  - $\alpha_t = softmax(w^T M_t)$ $\quad \alpha \in R^L$
  - $r_t = Y\alpha_t^T + \tanh(W^t r_{t-1})$ $\quad r \in R^k$
  - $h^* = \tanh(W^p r_N + W^x h_N)$ $\quad h^* \in R^k$

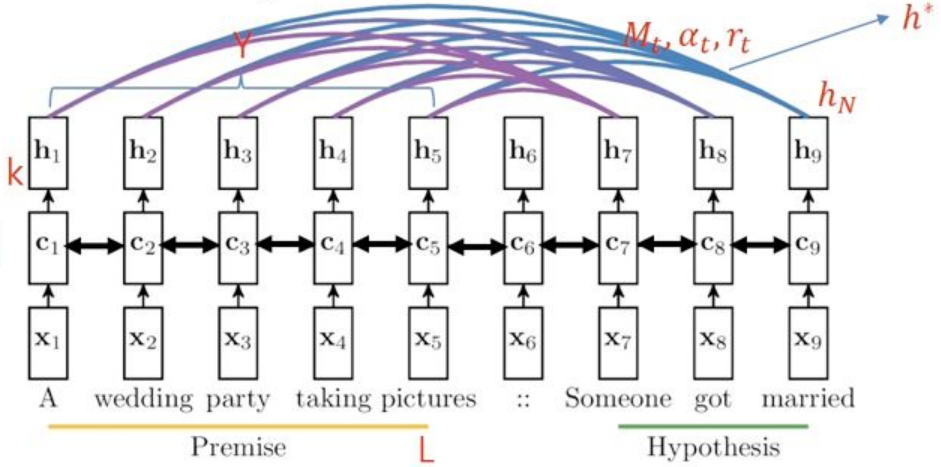


이전의 M, a, r을 이용해서 다음의 M,a,r을 만들어 간다.

Attention: ROM

❖ Recognizing Textual Entailment (RTE) _ word-by-word

- 2개의 LSTM을 사용 (제일 마지막 h에 걸릴 attention을 점차 모음)
  - $Y \in R^{k*L}$: k차원의 hidden vector를 L개 모음(premise encoding)
  - h_N: Premise, hypothesis를 모두 통과한 최종 output
  - $M_t = \tanh\left(W^y Y + [W^h h_t + W^r r_{t-1}] \otimes e_L\right)$ $\quad M \in R^{k*L}$
  - $\alpha_t = softmax(w^T M_t)$ $\quad \alpha \in R^L$
  - $r_t = Y\alpha_t^T + \tanh(W^t r_{t-1})$ $\quad r \in R^k$
  - $h^* = \tanh(W^p r_N + W^x h_N)$ $\quad h^* \in R^k$



자매품: bi-dir. LSTM

성능이 좋은 것은?

Word-by-work, 1way rnn

Bi-rnn, rnn

Attention, word-by-word

Attention: ROM

01 02 03 04 05

❖ Recognizing Textual Entailment (RTE)

- Attention visualization
  - 어느 부분에 집중했는지 시각적으로 확인

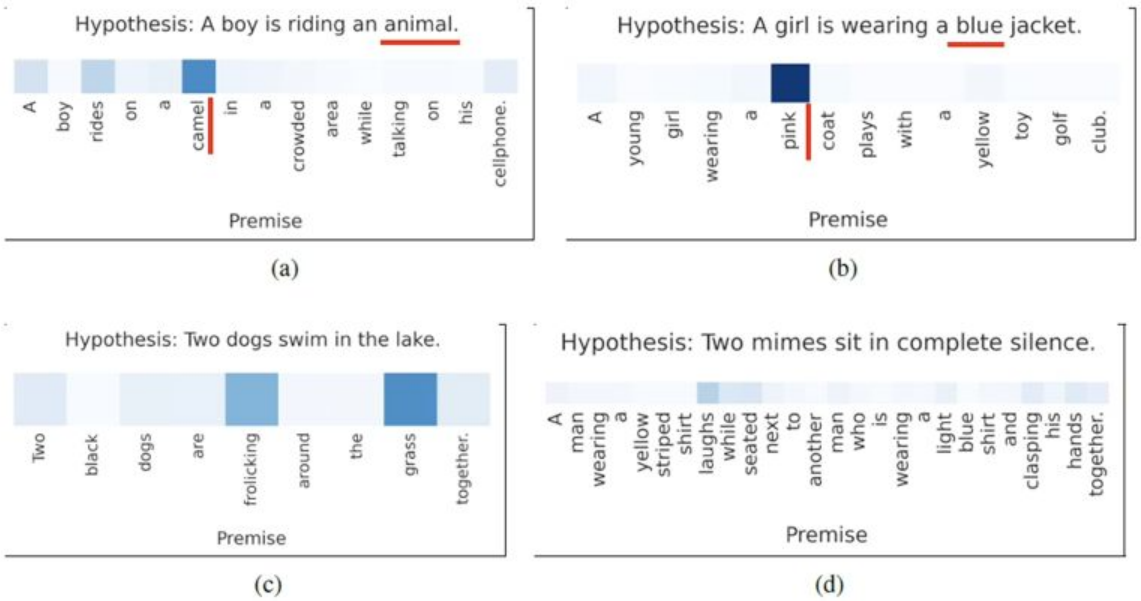


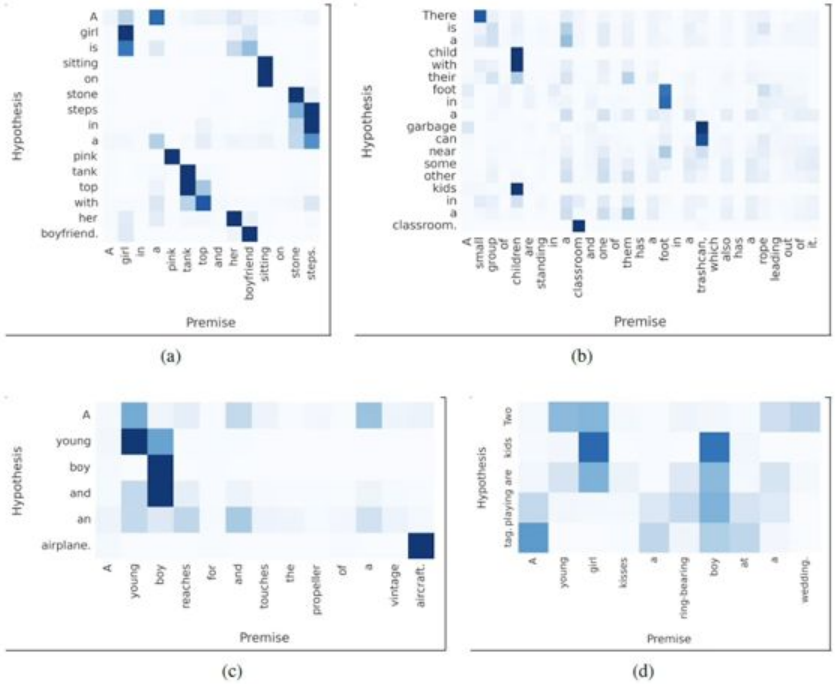
Figure 2: Attention visualizations.

Attention: ROM

01 02 03 04 05

❖ Recognizing Textual Entailment (RTE)

- Attention visualization
  - 어느 부분에 집중했는지 시각적으로 확인



Word-by-Word 관점

Attention: ROM

01 ❖ Attention summary

02 ▪ Original seq2seq의 한계를 Memory를 이용해 해소

03 • Bottleneck / 긴 글에 대한 표현력 저하 해소

04 ▪ 여러 문제에 적용 가능한 방법론

05 ▪ Encoding 부분을 학습하는데 효과적

❖ 한계점

▪ Memory를 읽기만 하는데, controller에 의한 Memory update가 필요

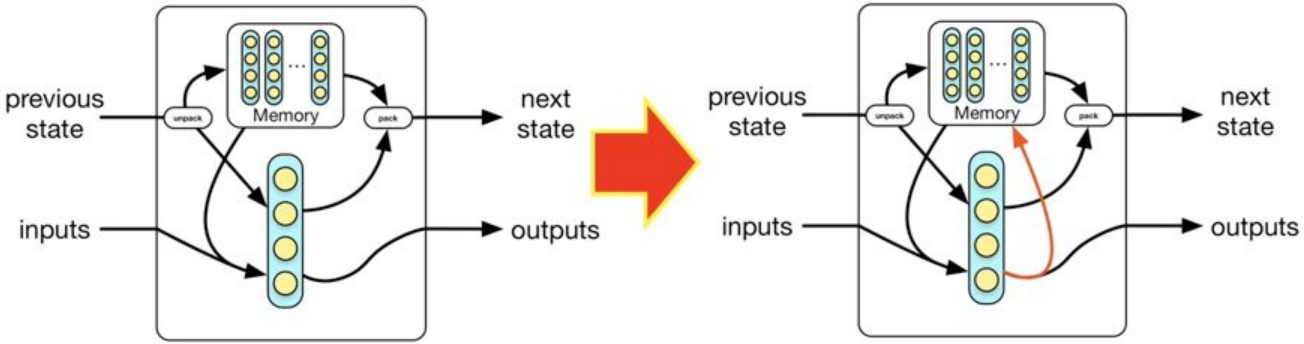
▪ Encoding에서 엄청 정보 표현을 잘 해야 하는 책임이 있음

Register Machines: RAM

01 02 03 04 05

❖ ROM -> RAM

- 앞서 언급한 한계점인 controller가 Memory에 관여 안 하는 문제에 집중
  - Controller는 ROM에서는 Input / Output만을 산출함
  - RAM에서는 Read, write를 만들어서 Memory에 접근
  - Memory에서 뭘 읽고, 뭘 지울지에 관여

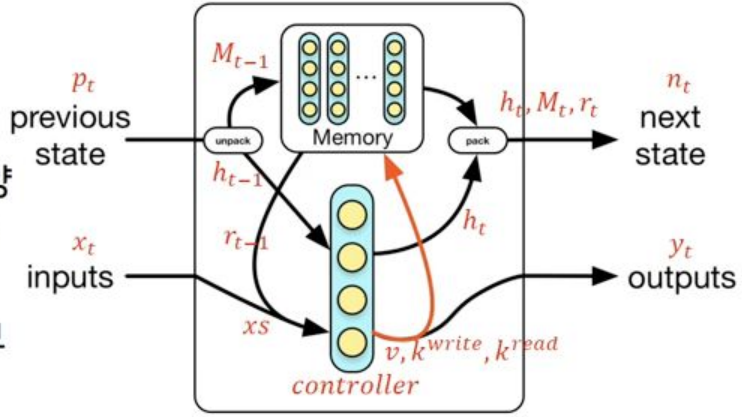


Register Machines: RAM

01 02 03 04 05

❖ RAM

- 추가된 변수
  - k^{read}: Memory를 읽을 양
  - k^{write}: Memory에 쓸 양
  - v: 새로 저장할 정보
  - r: Memory에서 읽은 정보



- RNN: $X * P \rightarrow Y * N$
  - $p_t = (h_{t-1}, M_{t-1}, r_{t-1})$
  - $n_t = (h_t, , M_t, r_t)$
  - $xs = x_t \oplus r_{t-1}$
  - $y_t, k^{read}, k^{write}, v, h_t = controller(xs, h_{t-1})$
  - $r_t = f_{read}(k^{read}, M_{t-1})$
    - $e.g. f_{read} = f_{att}$
  - $M_t = f_{write}(v, k^{write}, M_{t-1})$
    - $e.g. M_t[i] = a[i] \cdot v + (1 - a[i]) \cdot M_{t-1}[i]$
    - $Where\ a = softmax(k_{write} \cdot W_{write} \cdot M_{t-1})$

Stacks: Neural PDAs

01
02
03
04
05

❖ 자료구조

- Stack
  - Last in, first out(LIFO)
  - 쌓여 있는 책을 위에서부터 하나씩 가져가는 것으로 묘사
- Queue
  - First in, first out(FIFO)
  - 마트에서 줄 선 사람들이 앞에서부터 한 명씩 빠져나가는 것으로 묘사
- Deque
  - Double-ended queue
  - 양쪽으로 더해지고 빼질 수 있는 자료구조

자료구조의 형태를 Neural structure의 형태로 개선

https://www.slideshare.net/Arxhman/stacks-queues-deques

Stacks: Neural PDAs

01
02
03
04
05

Learning to Transduce with Unbounded Memory

Edward Grefenstette
Google DeepMind
etg@google.com

Karl Moritz Hermann
Google DeepMind
kmh@google.com

Mustafa Suleyman
Google DeepMind
mustafasul@google.com

Phil Blunsom
Google DeepMind and Oxford University
pblunsom@google.com

https://arxiv.org/pdf/1506.02516.pdf

❖ Memory에 쌓는 부분부터 RNN이 좀 더 관여

- Conclusion
  - Even in tasks for which benchmarks obtain high accuracies, the memory-enhanced LSTMs converge earlier, and to higher accuracies, while requiring considerably fewer parameters than all but the simplest of Deep LSTMs.

- 어떻게 자료구조를 Neural structure에 쓸 수 있을까?
  - 미분 가능하게 만드는 것이 관건

- 먼저 stack을 살펴보고, queue, deque로 확장

http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01
02
03
04
05

❖ Neural Stack

- Push: adds a new value to the stack, (between 0 and 1)
- Pop: removes value(s) from the stack, (between 0 and 1)
- Read: returns the top value(s) on the stack
- 앞선 자료구조와의 차이
  - continuous한 입출력
  - 각 stack마다 가지고 있는 strength (지분)

Stack = memory

http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

❖ Neural Stack

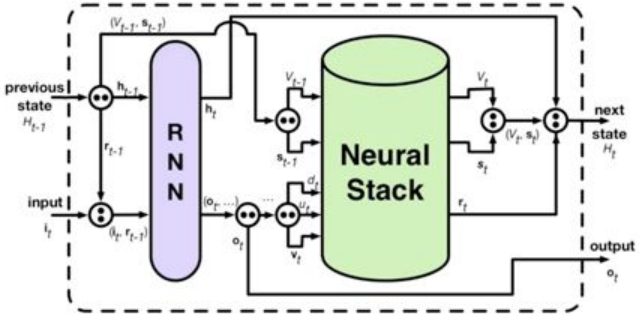
- 수식

$$V_t[i] = \begin{cases} V_{t-1}[i] & \text{if } 1 \le i < t \\ \mathbf{v}_t & \text{if } i = t \end{cases} \qquad (\text{Note that } V_t[i] = \mathbf{v}_i \text{ for all } i \le t)$$

$$\mathbf{s}_t[i] = \begin{cases} max(0, \mathbf{s}_{t-1}[i] - max(0, u_t - \sum_{j=i+1}^{t-1} \mathbf{s}_{t-1}[j])) & \text{if } 1 \le i < t \\ d_t & \text{if } i = t \end{cases}$$

$$\mathbf{r}_t = \sum_{i=1}^{t} (min(\mathbf{s}_t[i], max(0, 1 - \sum_{j=i+1}^{t} \mathbf{s}_t[j]))) \cdot V_t[i]$$

- V_t: Memory
- v_t: Memory에 추가할 새 value
- s_t: v의 V_t에서의 strength
- u_t: pop operation의 strength
- d_t: push operation의 strength
- r_t: output read value, stack의 맨 위 1에 해당하는 정보



Strength = 지분

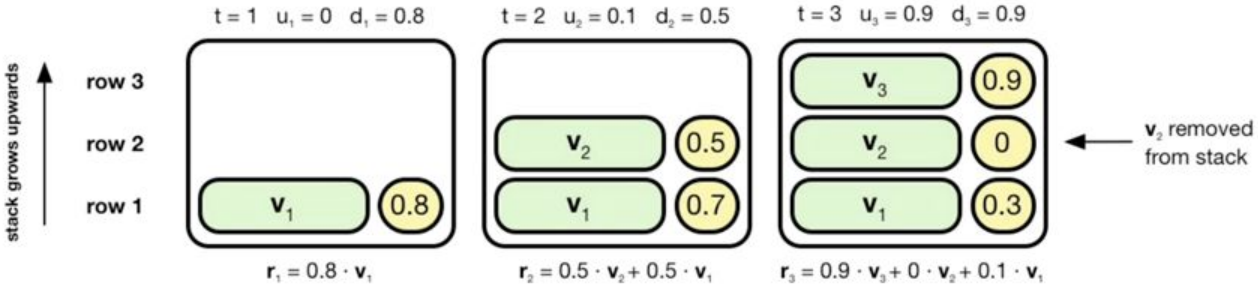
http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01 02 03 04 05

❖ Neural Stack

- 예



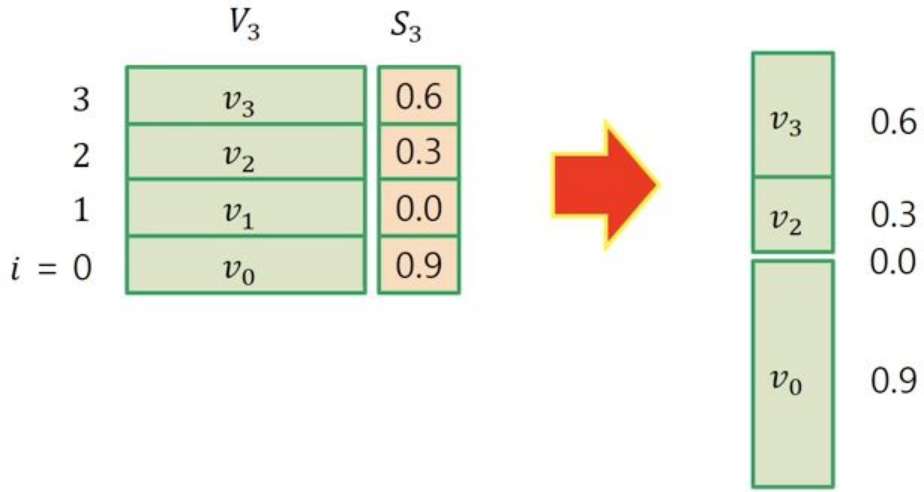
http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01 02 03 04 05

❖ Neural Stack

- 참고한 자료에서 S를 높이로 표현했는데, 좀 더 직관적이므로 참조



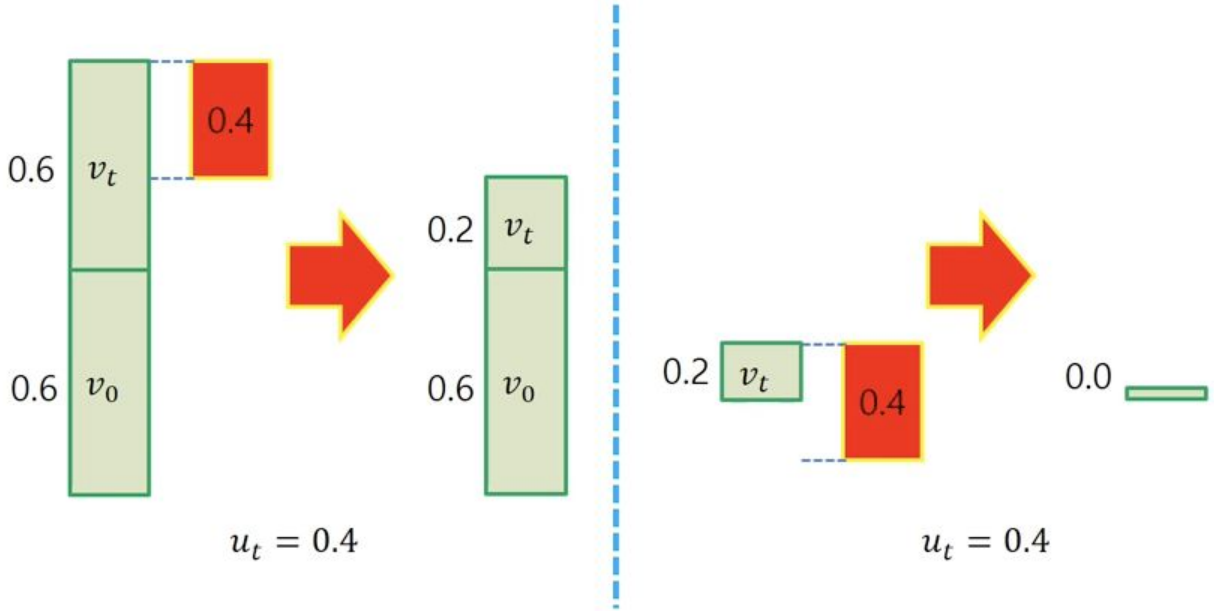
http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01 02 03 04 05

❖ Neural Stack

- Operations
  - Pop: removes height from top of the stack



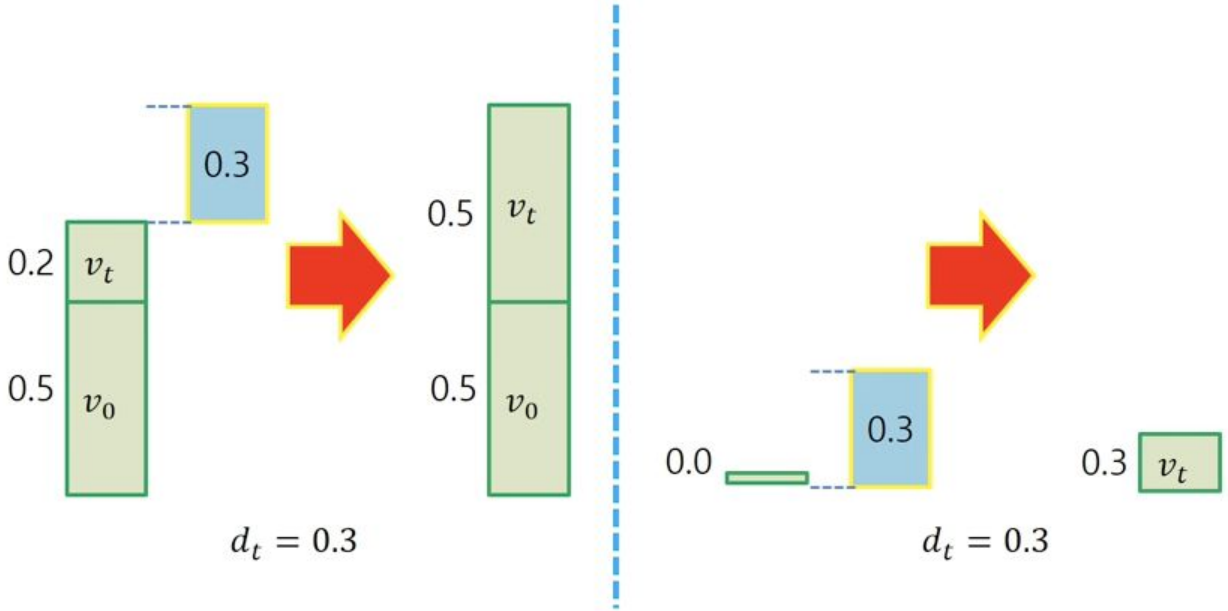
http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01 02 03 04 05

❖ Neural Stack

- Operations
  - Push: adds a value to the stack

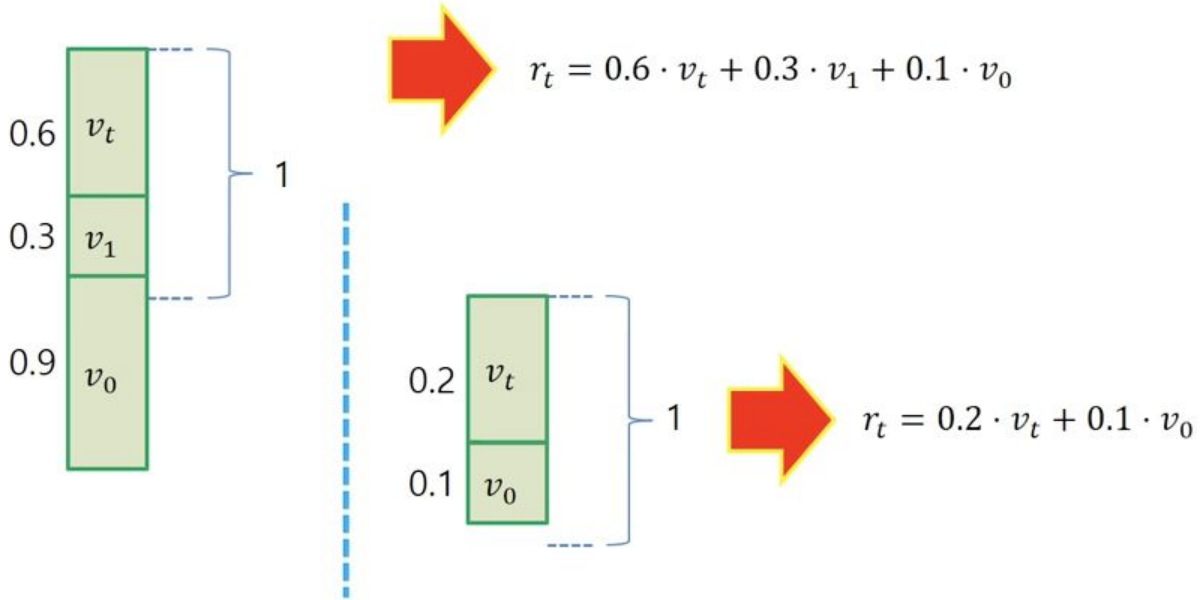


http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01 02 03 04 05

- Neural Stack
  - Operations
    - Read: outputs top 1.0 of height



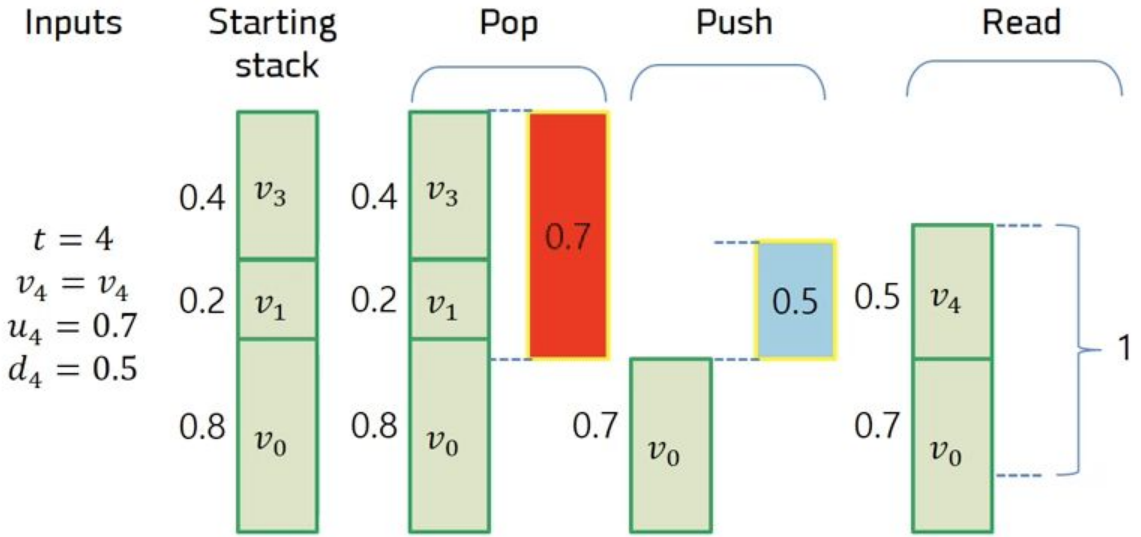
http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01 02 03 04 05

❖ Neural Stack

- Full update



Output: $r_4 = 0.5 \cdot v_4 + 0.5 \cdot v_0$

http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01 02 03 04 05

❖ Neural Queue

- 수식

$$\mathbf{s}_t[i] = \begin{cases} max(0, \mathbf{s}_{t-1}[i] - max(0, u_t - \sum_{j=1}^{i-1} \mathbf{s}_{t-1}[j])) & \text{if } 1 \leq i < t \\ d_t & \text{if } i = t \end{cases}$$

$$\mathbf{r}_t = \sum_{i=1}^{t} (min(\mathbf{s}_t[i], max(0, 1 - \sum_{j=1}^{i-1} \mathbf{s}_t[j]))) \cdot V_t[i]$$

- Queue는 FIFO이므로, pop operation만 stack과 다름

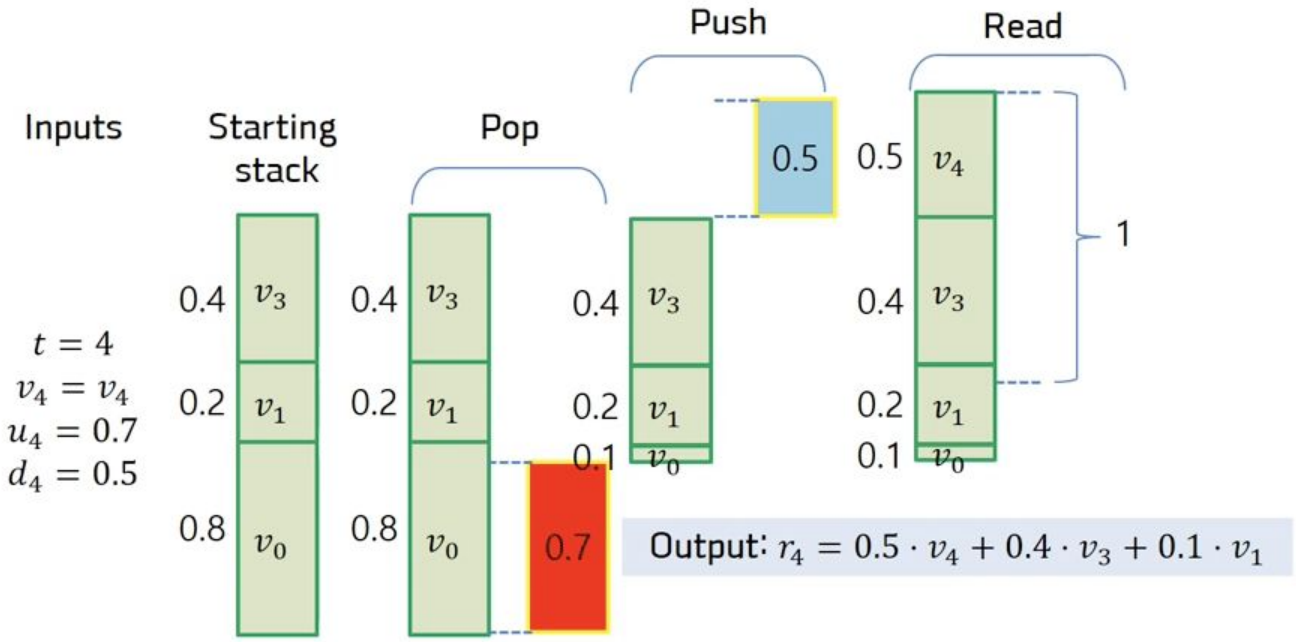
http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01 02 03 04 05

❖ Neural Queue

- Full update



http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01 ❖ Neural Deque

02 ▪ 수식

03 • 위, 아래 모두 pop, push 될 수 있으므로, 수식이 Top, Bottom으로 나뉨

04

$$V_t[i] = \begin{cases} \mathbf{v}_t^{bot} & \text{if } i = 1 \\ \mathbf{v}_t^{top} & \text{if } i = 2t \\ V_{t-1}[i-1] & \text{if } 1 < i < 2t \end{cases}$$

05

$$\mathbf{s}_t^{top}[i] = max(0, \mathbf{s}_{t-1}[i] - max(0, u_t^{top} - \sum_{j=i+1}^{2(t-1)-1} \mathbf{s}_{t-1}[j])) \quad \text{if } 1 \le i < 2(t-1)$$

$$\mathbf{s}_t^{both}[i] = max(0, \mathbf{s}_t^{top}[i] - max(0, u_t^{bot} - \sum_{j=1}^{i-1} \mathbf{s}_t^{top}[j])) \quad \text{if } 1 \le i < 2(t-1)$$

$$\mathbf{s}_t[i] = \begin{cases} \mathbf{s}_t^{both}[i-1] & \text{if } 1 < i < 2t \\ d_t^{bot} & \text{if } i = 1 \\ d_t^{top} & \text{if } i = 2t \end{cases}$$

$$\mathbf{r}_t^{top} = \sum_{i=1}^{2t} (min(\mathbf{s}_t[i], max(0, 1 - \sum_{j=i+1}^{2t} \mathbf{s}_t[j]))) \cdot V_t[i]$$

$$\mathbf{r}_t^{bot} = \sum_{i=1}^{2t} (min(\mathbf{s}_t[i], max(0, 1 - \sum_{j=1}^{i-1} \mathbf{s}_t[j]))) \cdot V_t[i]$$

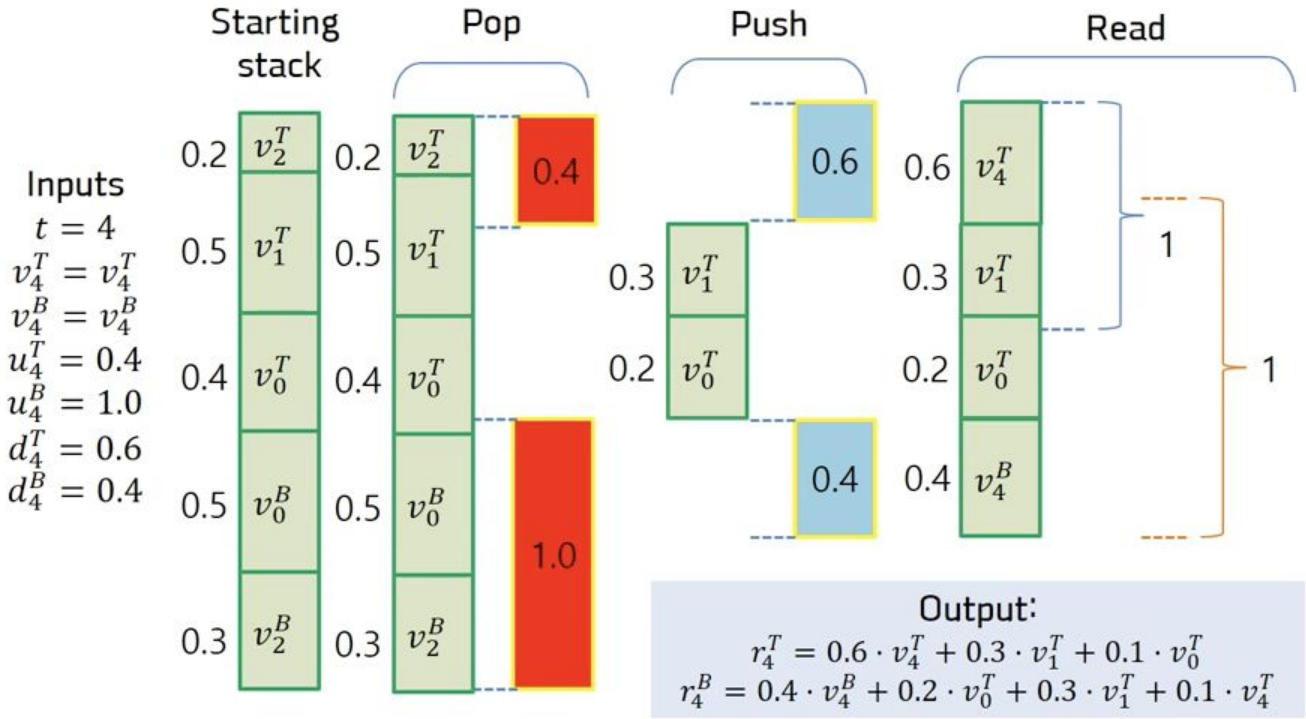
http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

01 02 03 04 05

❖ Neural Deque

- Full update



http://machinaut.com/a-graphical-explanation-of-deepminds-neural-stacks/

Stacks: Neural PDAs

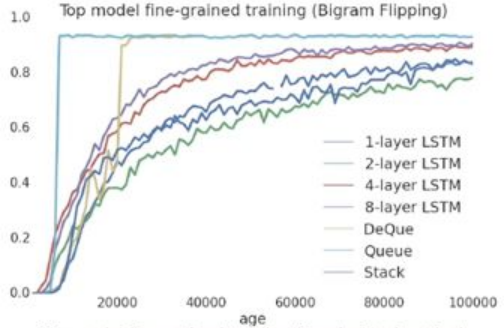
01 02 03 04 05

❖ Result

- 성능도 기존 LSTM에 비해 좋고, 수렴 속도도 빠름

Experiment	Model	Training		Testing	
		Coarse	Fine	Coarse	Fine
Sequence Copying	4-layer LSTM	0.98	0.98	0.01	0.50
	Stack-LSTM	0.89	0.94	0.00	0.22
	Queue-LSTM	1.00	1.00	1.00	1.00
	DeQue-LSTM	1.00	1.00	1.00	1.00
Sequence Reversal	8-layer LSTM	0.95	0.98	0.04	0.13
	Stack-LSTM	1.00	1.00	1.00	1.00
	Queue-LSTM	0.44	0.61	0.00	0.01
	DeQue-LSTM	1.00	1.00	1.00	1.00
Bigram Flipping	2-layer LSTM	0.54	0.93	0.02	0.52
	Stack-LSTM	0.44	0.90	0.00	0.48
	Queue-LSTM	**0.55**	**0.94**	**0.55**	**0.98**
	DeQue-LSTM	**0.55**	**0.94**	**0.53**	**0.98**
SVO to SOV	8-layer LSTM	0.98	0.99	0.98	0.99
	Stack-LSTM	1.00	1.00	1.00	1.00
	Queue-LSTM	1.00	1.00	1.00	1.00
	DeQue-LSTM	1.00	1.00	1.00	1.00
Gender Conju-gation	8-layer LSTM	0.98	0.99	0.99	0.99
	Stack-LSTM	0.93	0.97	0.93	0.97
	Queue-LSTM	1.00	1.00	1.00	1.00
	DeQue-LSTM	1.00	1.00	1.00	1.00

(a) Comparing Enhanced LSTMs to Best Benchmarks



(b) Comparison of Model Convergence during Training

Figure 2: Results on the transduction tasks and convergence properties

Q & A

01

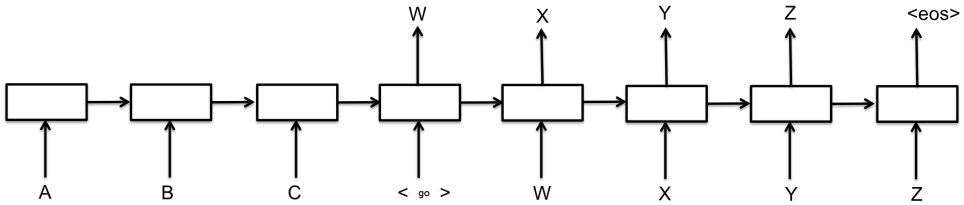
02

03

04

05

문장을 학습하는 딥러닝 RNN의 Seq2Seq 모델 설명

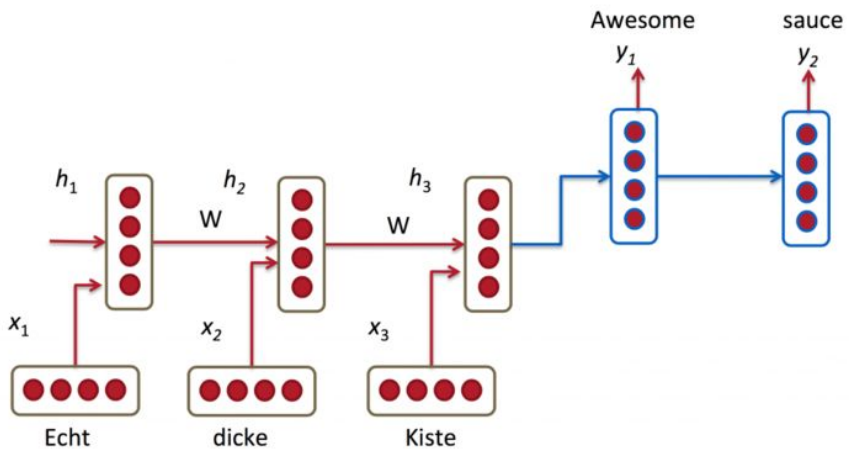


Seq2Seq는 2개의 RNN 셀로 구성됩니다. 위 그림에서 ABC를 입력으로 받는 것이 인코더 RNN 셀이고 오른쪽의 <go>WXYZ를 입력으로 받는 것이 디코더 RNN 셀입니다.

그림에서는 모두 8개의 셀이 있지만 실제로는 2개의 인코더와 디코더 셀만 존재합니다. RNN은 자신의 출력도 입력으로 받기 때문에 위 그림은 순차적인 입력을 풀어서 표현한 것입니다.
ABC라고 입력하면 WXYZ라고 대답하도록 훈련하고 싶다면 인코더에 ABC를 순서대로 집어넣습니다. 그리고 <go>WXYZ를 디코더에 순차적으로 입력하고 각각 출력이 WXYZ<eos>가 되도록 학습합니다.

정리하자면 ABC<go>WXYZ -> ___WXYZ<eos>으로 훈련을 시킨다고 보시면 됩니다. 여기서 <go>는 문장의 시작을, <eos>는 문장의 끝을 나타냅니다. 그리고 각 단어는 Word2Vec 같은 방법을 사용해서 벡터로 표현합니다.이렇게 트레이닝이 된 모델을 사용해서 문장을 예측할 수 있습니다. ABC라고 인코더에 순차적으로 입력하고 디코더에 <go>를 넣은후 그 출력인 W를 다음번 입력으로 합니다. 이렇게 반복해서 디코더에 넣다가 <eos>가 나오면 종료합니다. 디코더의 출력을 합치면 WXYZ<eos>의 답변문장을 구할 수 있습니다.

딥러닝과 NLP에서 Attention and Memory



위의 그림에서 "Echt", "Dicke"및 "Kiste" 단어가 인코더에 입력되고 특수 신호(표시되지 않음)가 표시된 후 디코더가 번역된 문장을 만들기 시작합니다. 디코더는 특수한 문장 토큰이 생성될 때까지 단어를 생성합니다. 여기서, 벡터 h는 인코더의 내부 상태를 나타냅니다.

자세히 보면, 디코더가 인코더에서 마지막으로 hidden state(위의 h_3)를 기반으로 번역을 생성한다고 가정할 수 있습니다. 이 h_3 벡터는 원문 문장에 대해 알아야 할 모든 것을 인코딩해야 합니다. 그 의미를 완전히 포착해야 합니다

https://blog.naver.com/rkdwnsdud555/221224418283

http://www.wildml.com/2016/01/attention-and-memory-in-deep-learning-and-nlp/

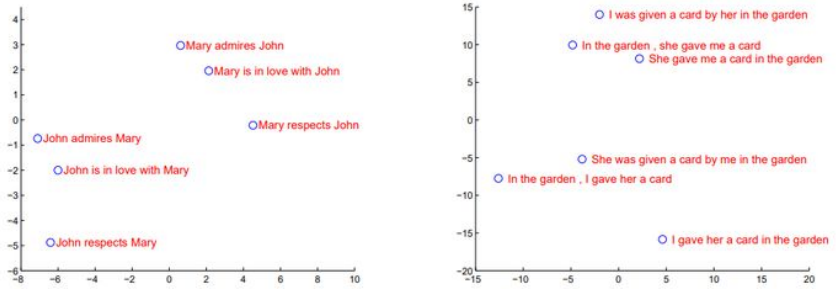


Figure 2: The figure shows a 2-dimensional PCA projection of the LSTM hidden states that are obtained after processing the phrases in the figures. The phrases are clustered by meaning, which in these examples is primarily a function of word order, which would be difficult to capture with a bag-of-words model. Notice that both clusters have similar internal structure.

좀 더 기술적인 용어로, 그 벡터는 문장을 embedding하는 것입니다. 실제로, 차원축소를 위해 PCA 또는 t-SNE를 사용하여 낮은 차원 공간에 여러 문장의 임베딩 정보를 플로팅하면 의미론적으로 유사한 문구가 서로 가까이 있음을 알 수 있습니다. 정말 놀라울 따름입니다.

그러나, 잠재적으로 매우 긴 문장에 대한 모든 정보를 단일 벡터로 인코딩한 다음 디코더 작업을 기반으로 훌륭한 변환을 생성 할 수 있다고 가정하는 것은 다소 무리가 있습니다. 원본 문장의 길이가 50단어라고 가정해 봅시다.

영어 번역의 첫 번째 단어는 원본 문장의 첫 단어와 관련성이 높습니다. 그러나 이것은 디코더가 50단계 전의 정보를 고려해야한다는 것을 의미하며 그 정보는 어떻게든 벡터에 인코딩되어야합니다.

RNN은 그러한 장거리 의존성을 다루는데 문제가있는 것으로 알려져있다. 이론적으로 LSTM과 같은 아키텍처는 이를 처리 할 수 있어야 하지만 실제로는 장거리 의존성이 여전히 문제가됩니다.

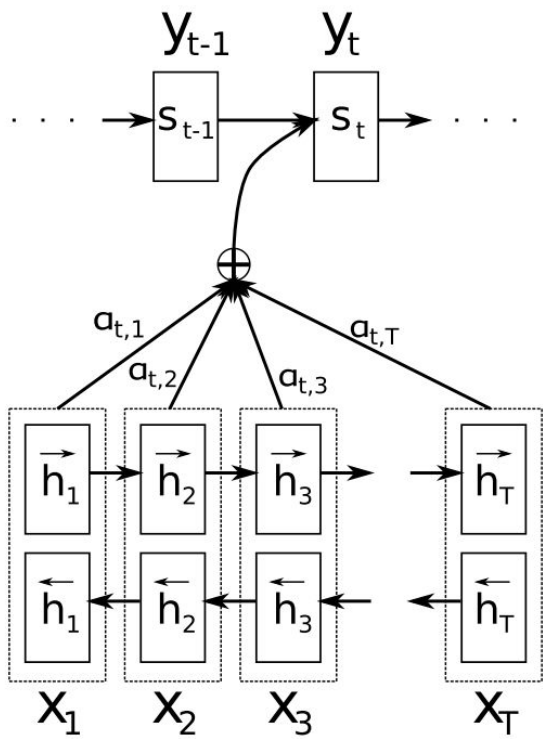
예를 들어, 원본 시퀀스를 역방향으로(인코더로 역방향 공급하는) 디코더에서 인코더의 관련 부분까지의 경로를 단축하므로 훨씬 더 나은 결과를 산출한다는 것을 연구원은 발견했습니다. 마찬가지로 입력 시퀀스를 두 번 전달하면 네트워크가 더 잘 기억하는 데 도움이되는 것처럼 보입니다.

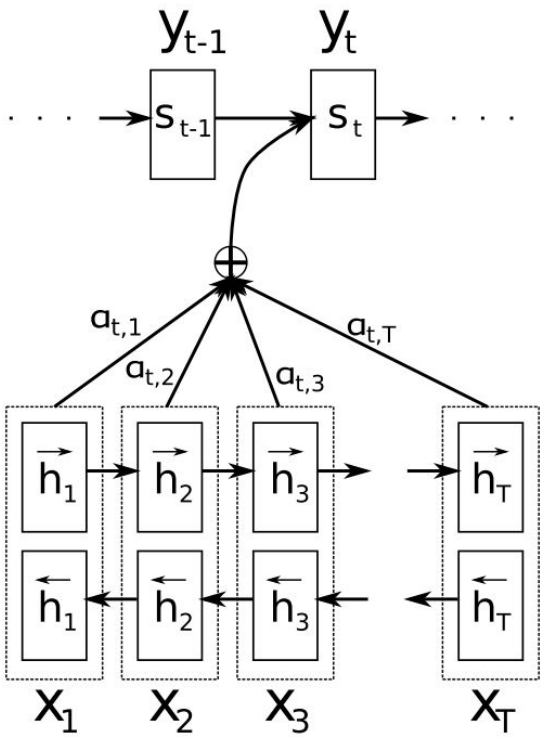
저는 문장을 뒤집는 접근법을 생각합니다. 이것은 실제 상황에서 더 잘 작동하지만, 근본적인 해결책은 아닙니다. 대부분의 번역 벤치 마크는 프랑스어와 독일어와 같은 언어로 이루어지며 영어와 매우 유사합니다(심지어 중국어 단어 순서는 영어와 매우 유사합니다). 그러나 대부분의 경우 문장의 마지막 단어가 영어 번역의 첫 번째 단어로 예측 할 수 있는 언어(예 : 일본어)가 있습니다. 이 경우 입력을 반대로하면 상황이 악화됩니다. 그래서, 대안은 무엇입니까? 주의 메커니즘.

attention 메커니즘으로 우리는 더 이상 고정 길이의 벡터로 전체 소스 문장을 인코드하지 않습니다. 오히려 디코더가 출력 생성의 각 단계에서 원본 문장의 다른 부분에 "attend"하도록 허용합니다.

우리는 모델이 입력된 문장과 지금까지 무엇을 만들어 냈는지를 기반으로 무엇에 주목할지를 학습합니다. 따라서 영어와 독일어와 같이 잘 정렬된 언어의 경우 디코더는 아마도 순차적으로 주목할 것입니다.

첫 번째 단어를 만들 때 첫 번째 단어에 주목하는 등. 그것이 바로 [Neural Machine Translation by Jointly Learning to Align and Translate]에서 한 일이며, 다음과 같이 보입니다.





여기서, y는 디코더에 의해 생성된 번역된 단어이고 x는 원본 문장입니다. 위의 그림은 양방향 RNN을 사용하지만 중요하지 않으며 반대 방향을 무시할 수 있습니다.

중요한 부분은 각 디코더 출력 단어 y_t가 마지막 상태뿐만 아니라 모든 입력 상태의 가중치 조합에 의존한다는 것입니다. a는 각 출력에 대해 각 입력 상태가 얼마나 많이 고려되어야 하는지를 정의하는 가중치입니다.

따라서, $a_{3,2}$가 큰 수의 경우, 이것은 디코더가 목표 문장의 세 번째 단어를 생성하는 동안 원본 문장의 두 번째 상태에 많은 주의를 기울인다는 것을 의미합니다. a는 일반적으로 1로 합계되기 위해 노말라이즈됩니다.(입력 상태에 대한 분포입니다).