10.1. Long Short-Term Memory

- 개요

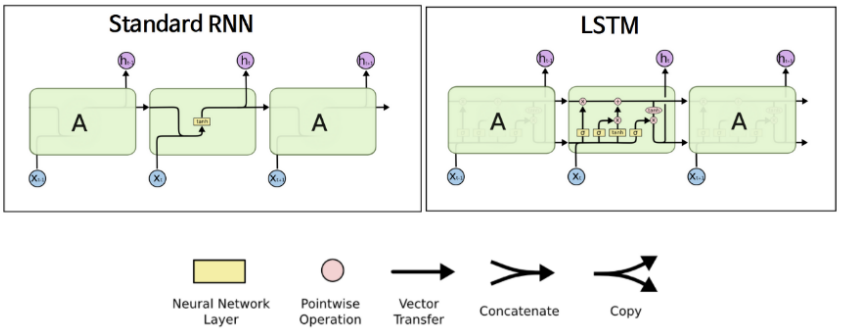
LSTM은 Vanishing gradient(exploding gradient) 문제와 단어 간의 사이가 멀어질수록 잘 기억을 하지 못하는 'long term dependency(장기 의존성)' 문제를 해결하기 위하여 제안된 RNN의 특별한 유형중 하나이다. 1997년 처음 Hochreiter & Schmidhuber에 의하여 소개된 이후 많은 사람들에 의하여 가공되고 유명해졌으며, 현재 광범위하게 사용되고 있다.

long-term dependency (장기 의존성)이란 시퀀스 데이터의 길이가 길어질수록, 과거의 중요한 정보에 대한 학습이 어려워지는 문제이다. 그대로 두면 vanishing gradient가 발생하고, gradient를 좀 키우면 exploding gradient가 발생한다. 이 문제를 해결하기 위해서 두가지 접근이 존재하는데, 하나는 stochastic gradient descent(SGD) 방법으로 대체하는 것이고 다른 하나는 LSTM과 GRU와 같은 더 세련된 RNN모델을 사용하는 것이다.

- LSTM 구조

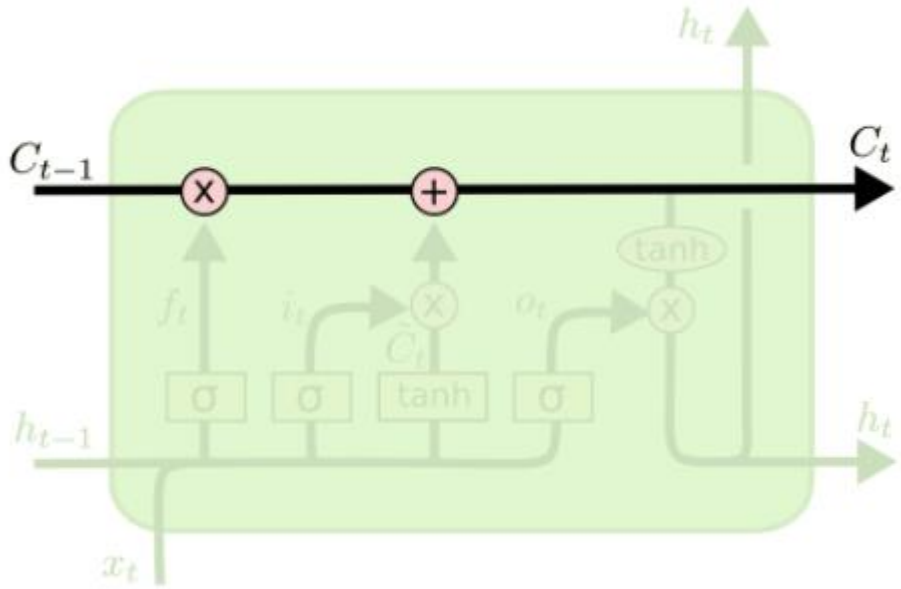
LSTM의 구조는 기본 RNN과 같이 체인처럼 이어져있는(chain-like) 모양이지만, 아주 특별한 방식으로 상호작용하는 4개의 길(cell state, forget gate, input gate, output gate)이 있다.

LSTM은 RNN과 다르게 화살표로 나타나는 상태(state)가 2개(cell-state, hidden-state)로 되어있다. cell-state를 C(t)라고 표기하며 장기 상태(long term state)이라고 하며, hidden-state를 h(t)라고 표기하며 단기상태(short term state)라고 한다.



- Cell State

LSTM의 핵심은 모델 구조도에서 위의 수평선에 해당하는 cell state이다. cell state는 컨베이어 벨트와 같은 역할로 전체 체인을 지나면서, 일부 선형적인 상호작용만을 진행한다. 정보가 특별한 변화 없이 흘러가는 것으로 이해하면 된다. LSTM에서 cell state를 업데이트하는 것이 가장 중요한 일이며, 'gate라는 구조가 cell state에 정보를 더하거나 제거하는 조절 기능을 한다.

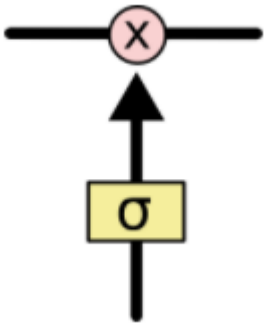


- Gate

Gate = 'sigmoid layer outputs'+'pointwise multiplication operation'

gate란 정보를 선택적으로 통과시키는 길로, 시그모이드 뉴럴 네트워크 층과 point wise multiplication 연산으로 이루어져있다.

LSTM은 cell state를 조절하고 보호하기 위하여 총 3개의 gate(forget, input, output)를 갖는다. 시그모이드 층은 0과 1사이의 값을 출력하는데, 이 값들은 각 원소들을 얼마나 많이 통과시킬 것인지를 의미한다. 예를들어 숫자 '0'은 어떤것도 통과시키지 않음을 의미하며, 숫자 '1'은 모든 것을 통과시킴을 의미한다.

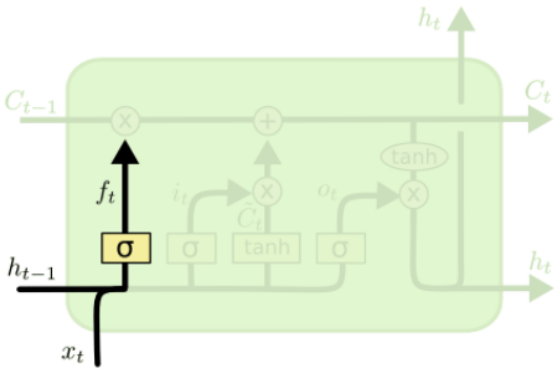


- LSTM 진행 단계

1. forget gate

LSTM의 첫번째 단계는, cell state에서 어떤 정보를 버릴 것인가를 결정하는 것이며, 이것은 "forget

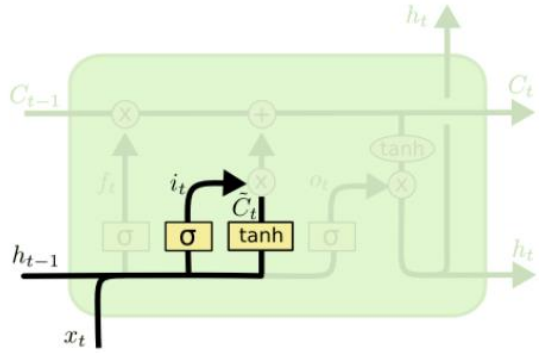
gate layer"의 시그모이드 층에서 결정된다. Forget gate에서는 이전 hidden 값 h(t-1)과 현재 입력 값 x(t)를 확인하고 이전 cell state C(t-1)에 있는 각 요소에 대해서 0과 1사이의 값을 생성한다. 숫자 '1'은 "완벽히 보존"을 의미하고 '0'은 "완벽히 제거"를 의미한다. 이 값은 최종 C(t)를 결정할때, C(t-1)과 곱해지는 값으로 이전의 cell state 정보(C(t))를 얼마나 통과시킬지를 결정한다.



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

2. Input gate

Input gate에서는 어떤 새로운 정보를 cell state에 저장/추가할 것인가하는 업데이트 정보를 결정한다. 여기에는 두가지 파트가 있는데, 1) "input gate layer" 이라고 불리는 시그모이드 층에서 어떤 값을 업데이트할지 결정하고, 2) tanh 층에서는 state에 추가될 수 있는 새로운 후보 값(벡터, 틸다C(t)) 를 생성한다.



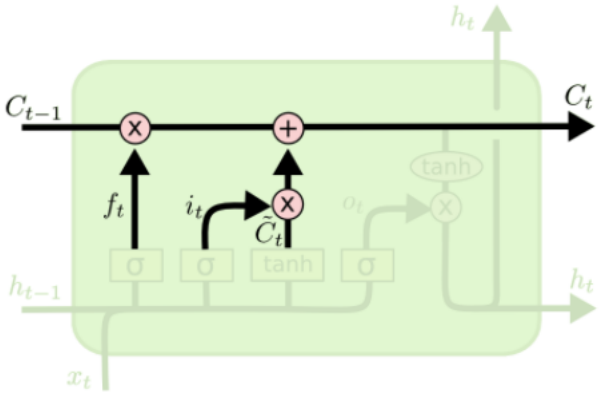
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

3. Cell state 업데이트

이제 오래된 cell state C(t-1)을 새로운 cell stat인 C(t)로 업데이트할 차례이다. 전 단계에서 무엇을 할지(어떤것을 잊을지, 어떤것을 새롭게 업데이트할지)는 이미 결정하였으니 이제 실제로 그것을 실행한다.

1) [forget layer] 오래된 cell state C(t-1)에 f(t)를 곱하여, 잊기로 했던 것들을 잊기!

2) [input layer] i(t)*C'(t) --> C(t)의 후보



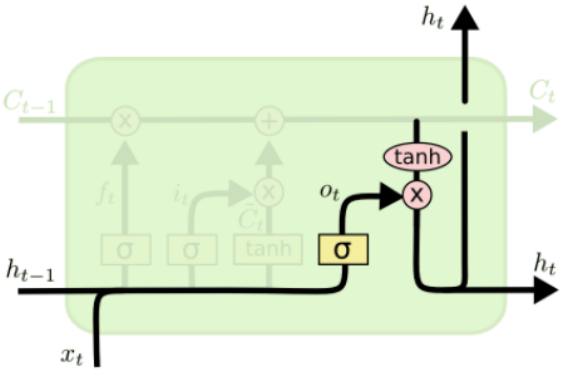
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

forget 잊는다 기존의 정보를 input 업데이트 한다 새로운 정보를

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

4. Ouptut gate

마지막으로 무엇을 출력(output)할지 결정한다. 이 출력은 현재 cell state인 C(t)를 가지고 정하지만, 모든 C(t)를 사용하지 않고, 일부만 필터링하여 출력한다.

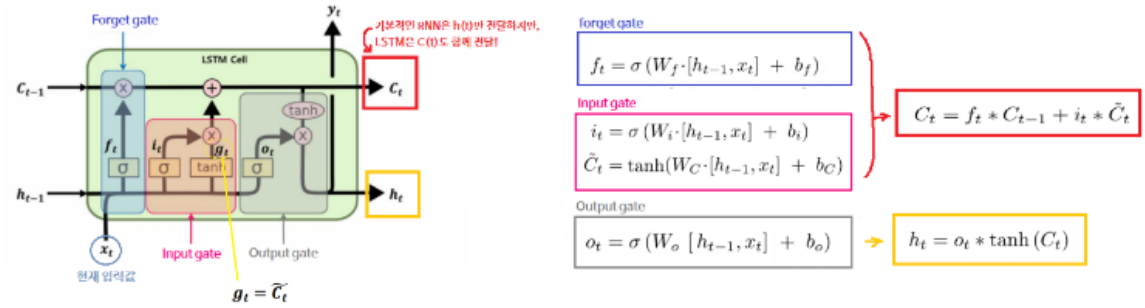


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

(1) 첫째로, 우리는 시그모이드 층을 써서 현재 cell state 중에서 어떤 부분만 출력을 할지를 정한다. -> o(t)

(2) 둘째로, cell state를 tanh 함수(-1과 1사이의 값을 출력)를 사용하고 위에서 구한 o(t)와 곱하여 최종적으로 출력할 부분을 결정한다.

5. 전체 그림

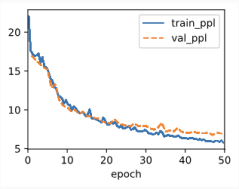


- Code

```
class LSTM(d2l.RNN):
    def __init__(self, num_inputs, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        self.rnn = nn.LSTM(num_inputs, num_hiddens)

    def forward(self, inputs, H_C=None):
        return self.rnn(inputs, H_C)

lstm = LSTM(num_inputs=len(data.vocab), num_hiddens=32)
model = d2l.RNNLM(lstm, vocab_size=len(data.vocab), lr=4)
trainer.fit(model, data)
```




```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has a the time travelly'
```

10.2. Gated Recurrent Units

- GRU의 등장 배경

-LSTM은 RNN의 치명적인 한계점이었던, 'Long-term dependency' 문제를 해결하면서 긴 시퀀스를 가진 데이터에서도 좋은 성능을 내는 모델이 되었지만, 복잡한 구조 때문에 RNN에 비하여 파라미터가 많이 필요하게 되었다. 파라미터가 많아지는데 데이터가 충분하지 않은 경우, 오버피팅이 발생하는데, 이러한 단점을 개선하기 위하여 LSTM의 변형인 GRU가 등장하게 되었다. GRU는 2014년 조경현 외 3인의 논문에서 처음 제안되었습니다.

- GRU

GRU의 핵심은 아래 두가지이다.

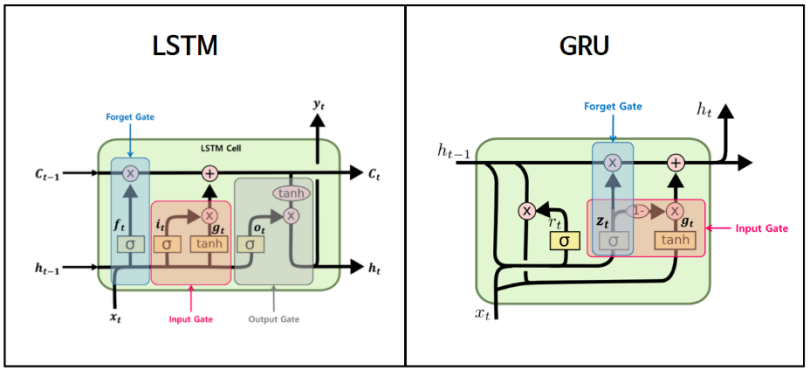
(1) LSTM의 forget gate와 input gate를 통합하여 하나의 'update gate'를 만든다.

(2) Cell State와 Hidden State를 통합한다.

GRU는 LSTM에 비하여 파라미터수가 적기 때문에 연산 비용이 적게 들고, 구조도 더 간단하지만, 성능에서도 LSTM과 비슷한 결과를 낸다.

- LSTM vs GRU

LSTM	GRU
gate 수 3개(forget, input, output)	gate 수 2개(reset, update)
Control the exposure of memory content (cell state)	Expose the entire cell state to other units in the network
Has separate input and forget gates	Performs both of these operations together via update gate
More parameters	Fewer parameters



1. LSTM의 Cell State(C(t))와 Hidden state(h(t))가 GRU에서는 하나의 벡터 (h(t))로 합쳐졌다.

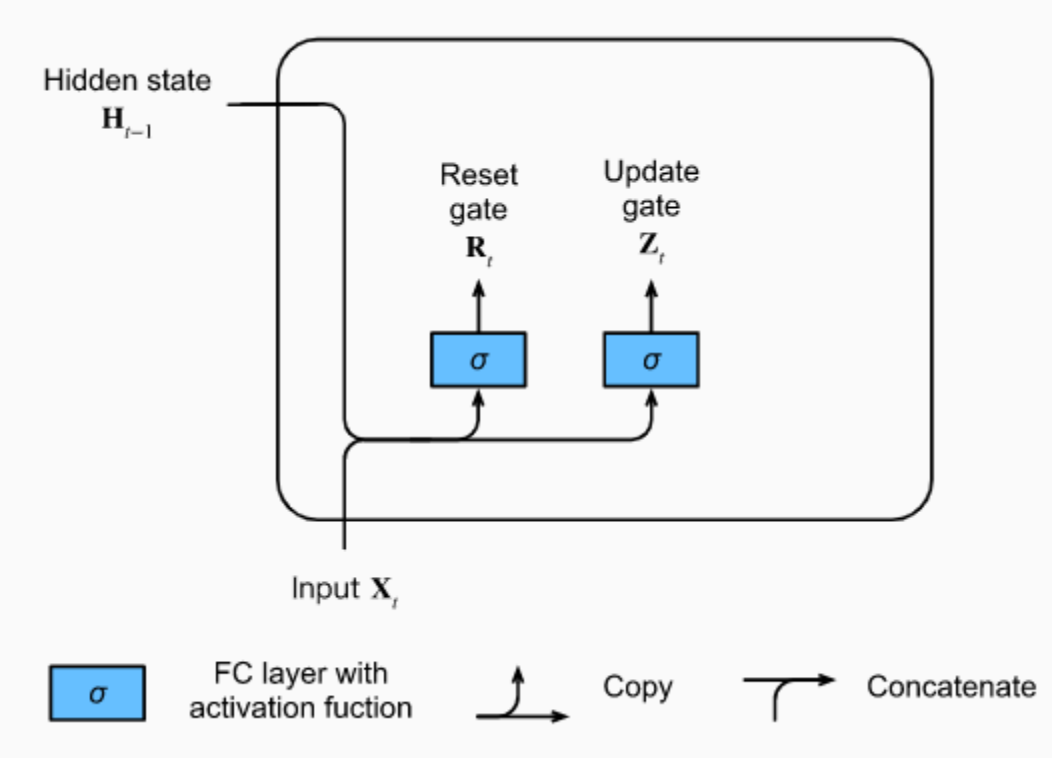
2. LSTM의 forget, input gate는 update gate로 통합, output gate는 없어지고, reset gate로 대체

3. LSTM에서는 forget과 input이 서로 독립적이었으나, GRU에서는 전체 양이 정해져 있어(=1), forget한 만큼 input하는 방식으로 제어한다. 이는 gate controller인 z(t)에 의해서 조절된다.

=> z(t)가 1이면 forget gate가 열리고, 0이면 input gate가 열린다.

- GRU 진행 단계

1. Gate



$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$

$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$

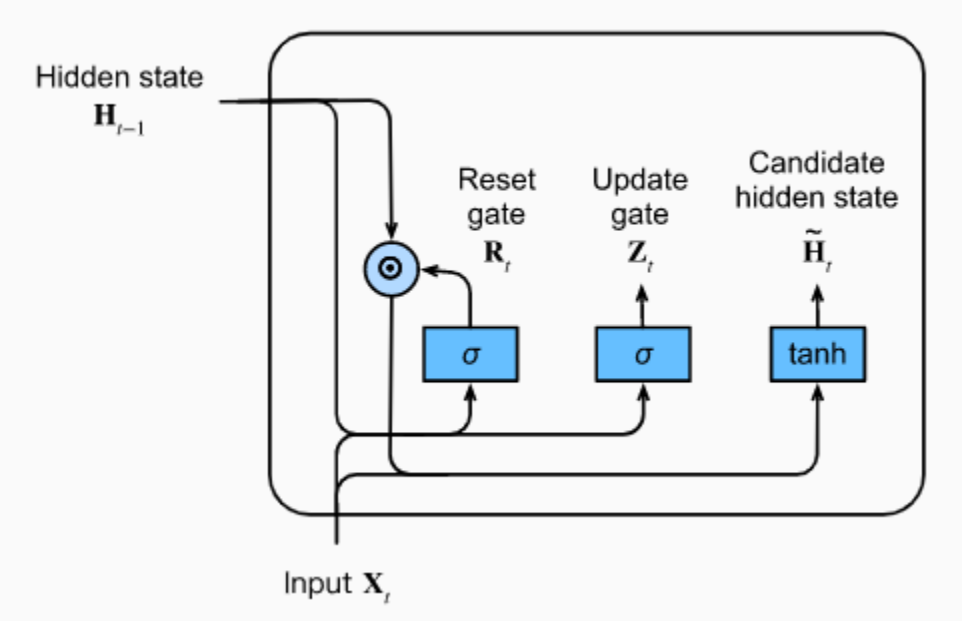
[1] Reset gate, r(t)

: 이전의 정보를 얼마나 잊어야하는지를 결정한다.

[2] Update gate, z(t)

: 이전의 정보(기억)을 얼마나 통과(유지)시킬지를 결정한다.

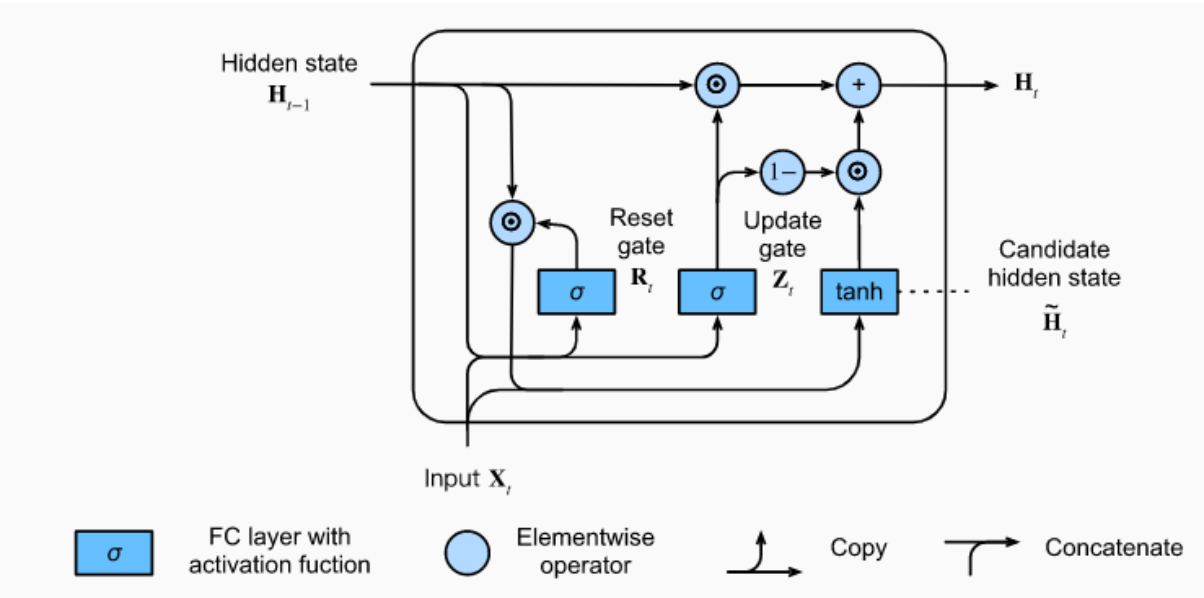
2. Candidate Hidden State



$$\tilde{h}_t = \tanh\left(W x_t + r_t \circ U h_{t-1}\right)$$

hidden layer의 후보(candidate)로 reset gate인 r(t)와 Uh(t-1)을 곱하여, 이전 타임 스텝에서 무엇을 지워야할지를 결정한다. tanh 함수는 -1 에서 1 사이의 값을 가지므로, candidate hidden state도 -1에서 1사이 값을 가진다. Updatㄷ gate를 아직 반영하지 않았기 때문에 아직까지 'candidate' hidden state인 것이다.

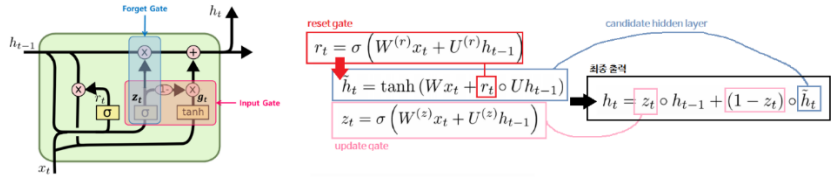
3. Hidden State



$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

이제 update gate를 합칠 차례입니다. update gate는 얼마나 새로운 hidden state(h(t)) 가 이전 hidden state(h(t-1)과 같을지 아니면 새로운 cadidate hidden state이 많이 사용될지를 결정한다. z(t)가 0이라면 old state(h(t-1))을 유지하고, 1이라면 candidate state(tilda h(t))로 완전히 대체된다.

4. 전체 그림



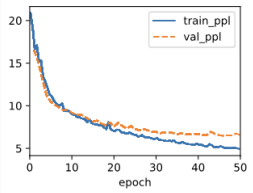
- Code

```
class GRU(d2l.RNN):
    def __init__(self, num_inputs, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        self.rnn = nn.GRU(num_inputs, num_hiddens)
```

The code is significantly faster in training as it uses compiled operators rather than Python.

PYTORCH MXNET JAX TENSORFLOW

```
gru = GRU(num_inputs=len(data.vocab), num_hiddens=32)
model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=4)
trainer.fit(model, data)
```



After training, we print out the perplexity on the training set and the predicted sequence following the provided prefix.

PYTORCH MXNET JAX TENSORFLOW

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has so it and the time '
```

10.3. Deep Recurrent Neural Networks

10.4. Bidirectional Recurrent Neural Networks

- 개요

지금까지의 시퀀스 데이터의 학습에서의 목표는 '지금까지 주어진 것을 보고 다음을 예측'하는 것이었고, 그 분야로는 시계열 또는 언어 모델이 있었다. 하지만, 시퀀스의 과거의 값들 뿐만 아니라 이후의 값들에 의해서도 값이 결정된다면 어떨까? 아래 세 문장을 보자.

i) I am _____.

ii) I am _____ hungry.

iii) I am _____ hungry, and I can eat half a pig.

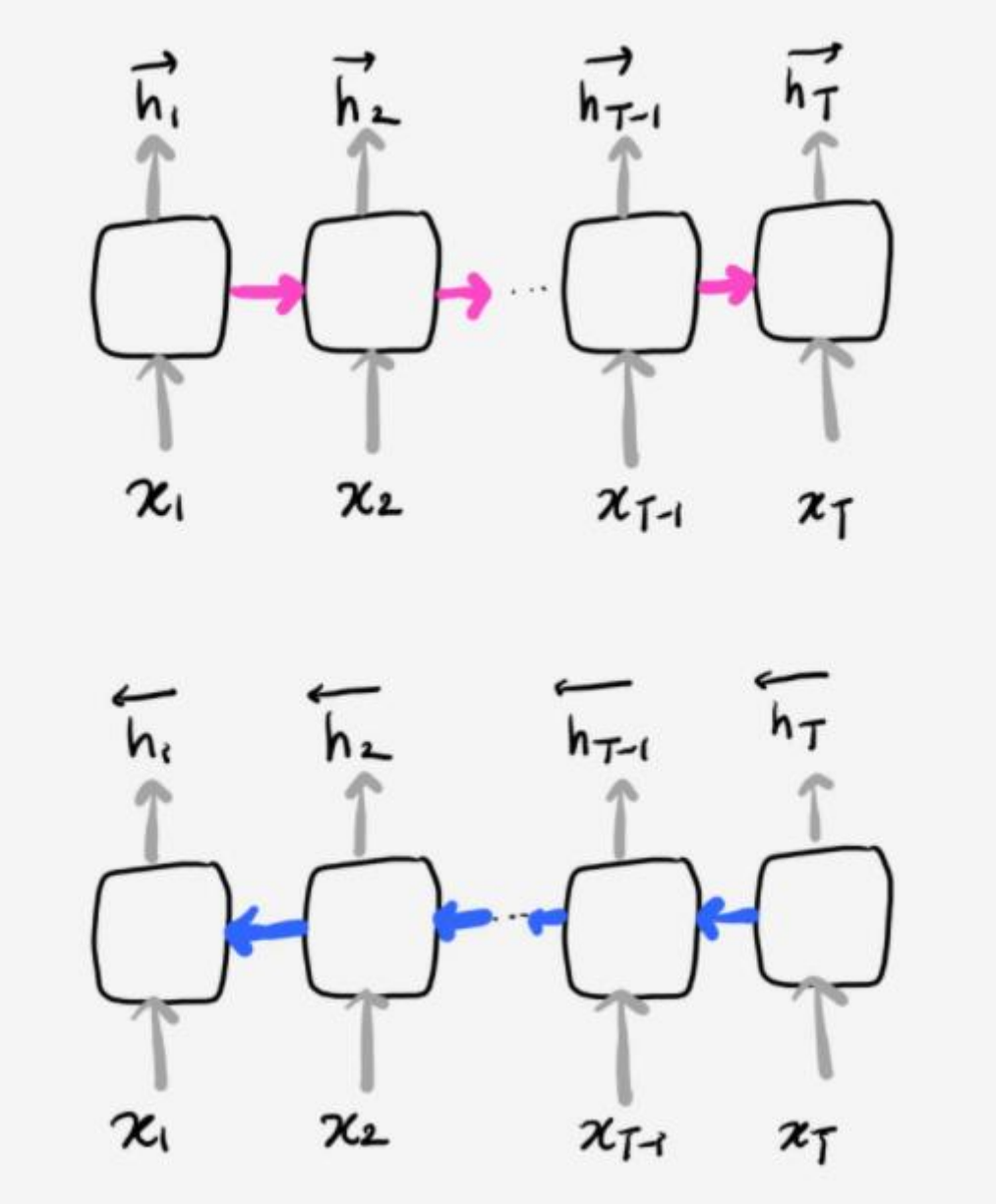
우리는 주어진 정보의 양에 따라 빈칸을 'happy'로 그리고 'not'이나 'very'와 같은 서로 아주 다른 단어들로 채울 수 있다. 즉, 구(phrase)의 끝의 정보를 알 수있다면 그것은 어떤 단어를 선택하느냐에 상당히 중요한 정보를 전달한다.

하지만, LSTM과 GRU와 같은 과거의 모델들은 예측하고자 하는 단어 뒤에 있는 정보를 이용할 수 없기 때문에 이러한 문제에서는 성능이 좋지 않았다. 예를 들어, 개체명인식을 잘 하기 위해서는 주어로 등장한 "Green"이 녹색을 뜻하는지, 사람이름 (Mr. Green)을 의미하는지를 알기위해서는 해당 단어 뒤에 단어들이 쓰였는가가 중요하다. 하지만 LSTM과 GRU으로는 'Green' 한 단어만 보고 예측을 해야하는 상황이 벌어지게 된다.

이런 기존 모델의 한계점을 개선하기 위하여 시퀀스의 이전부분 뿐만 아니라 이후 부분까지 결합하여 예측하는 Bidirectional RNN 모델이 제안되었으며, 1997년 Schuster & Paliwal의 논문에 게재되었다.

- 구조

2개의 RNN이 함께 쌓여있고, 최종 output은 두 RNN의 은닉상태를 통해 계산된다. 첫번째 토큰부터 시작하는 forward mode의 RNN에, 맨 마지막에서 시작해서 끝에서 앞으로 가는 backward mode의 RNN unit을 함께 쌓는다.



Bidirectional RNN에서의 각 time step의 final output은 아래와 같이 forward h(t)와 backward h(t)를 동시에 결합하여 출력한다.

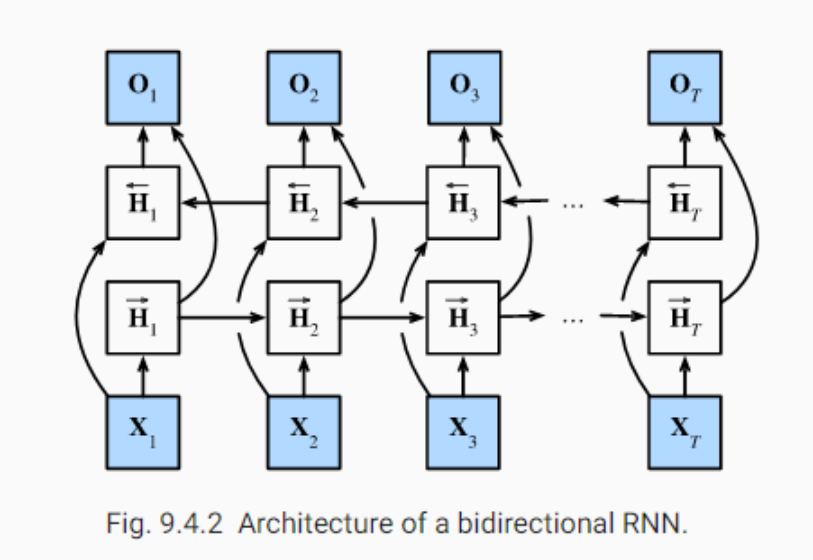


Fig. 9.4.2 Architecture of a bidirectional RNN.

10.5. Machine Translation and Dataset

10.6. The Encoder-Decoder Architecture

10.7. Sequence-to-Sequence Learning for Machine Translation

- 개요

- 한 언어의 문장을 다른 언어의 문장으로 변환하는 기계번역에서 예를 들어 영어를 프랑스어로 번역한다고 했을 때 발생하는 문제점이 있다. 일반적인 번역에서 영어 문장과 번역한 후의 프랑스어 문장은 서로 단어수가 정확히 일치하지 않는다. 또한, 영어 문장에서의 단어의 순서와 프랑스 문장에서의 순서도 같지 않다

[English] the black cat drank milk (5 words)

[French] le chat noir a bu du lait (7 words)

기존의 RNN 모델은 하나의 입력값에는 하나의 은닉층(hidden state)과 하나의 출력값이 존재했기 때문에, 출력값의 길이는 입력값의 길이와 늘 동일할 수밖에 없다. 영어 문장에서 존재하는 수와 번역 후 프랑스어 문장에 존재하는 단어수가 같아야하는 제약이 존재하게 된 것이다. 이러한 한계점을 해결하기 위하여 'Sequence to Sequence' 모델이 제안되었다.

- 기본 구조

기본 아키텍처는 2중 RNN으로 이루어져있다. 이 중 처음 번역할 문장을 입력값을 받는 RNN을 Encoder라고 하고, 뒤에서 번역된 문장을 생성하는 RNN을 Decoder라고 한다.

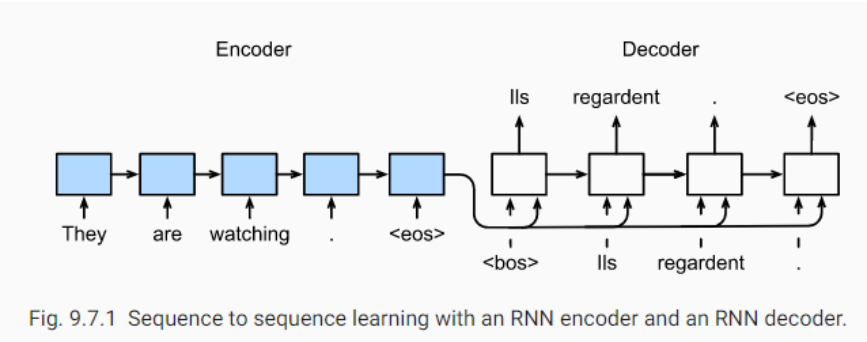


Fig. 9.7.1 Sequence to sequence learning with an RNN encoder and an RNN decoder.

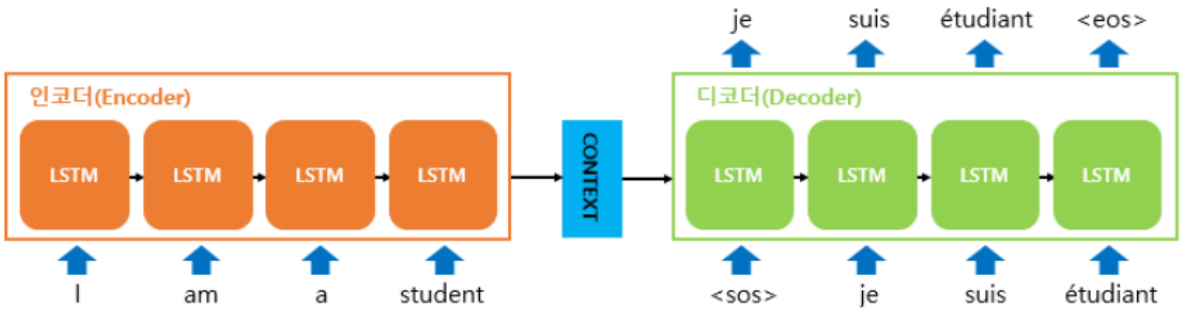
- Encoder

1) Encoder가 하는 일 - 입력 문장을 하나의 벡터로 만들기

Encoder는 입력 시퀀스(문장)를 정해진 크기의 context vector(c)로 변환하여 Decoder에게 넘겨준다. 이 context vector(c)에 입력 시퀀스(x1, x2....xT)의 정보를 담는다(인코딩). 즉, Encoder는 문장을 받아서 하나의 숫자벡터로 표현하며, Decoder는 숫자벡터를 받아서 문장으로 표현합니다.

2) Encoder의 특징 - 입력 문장을 통째로 처리/Non-Autoregressvie

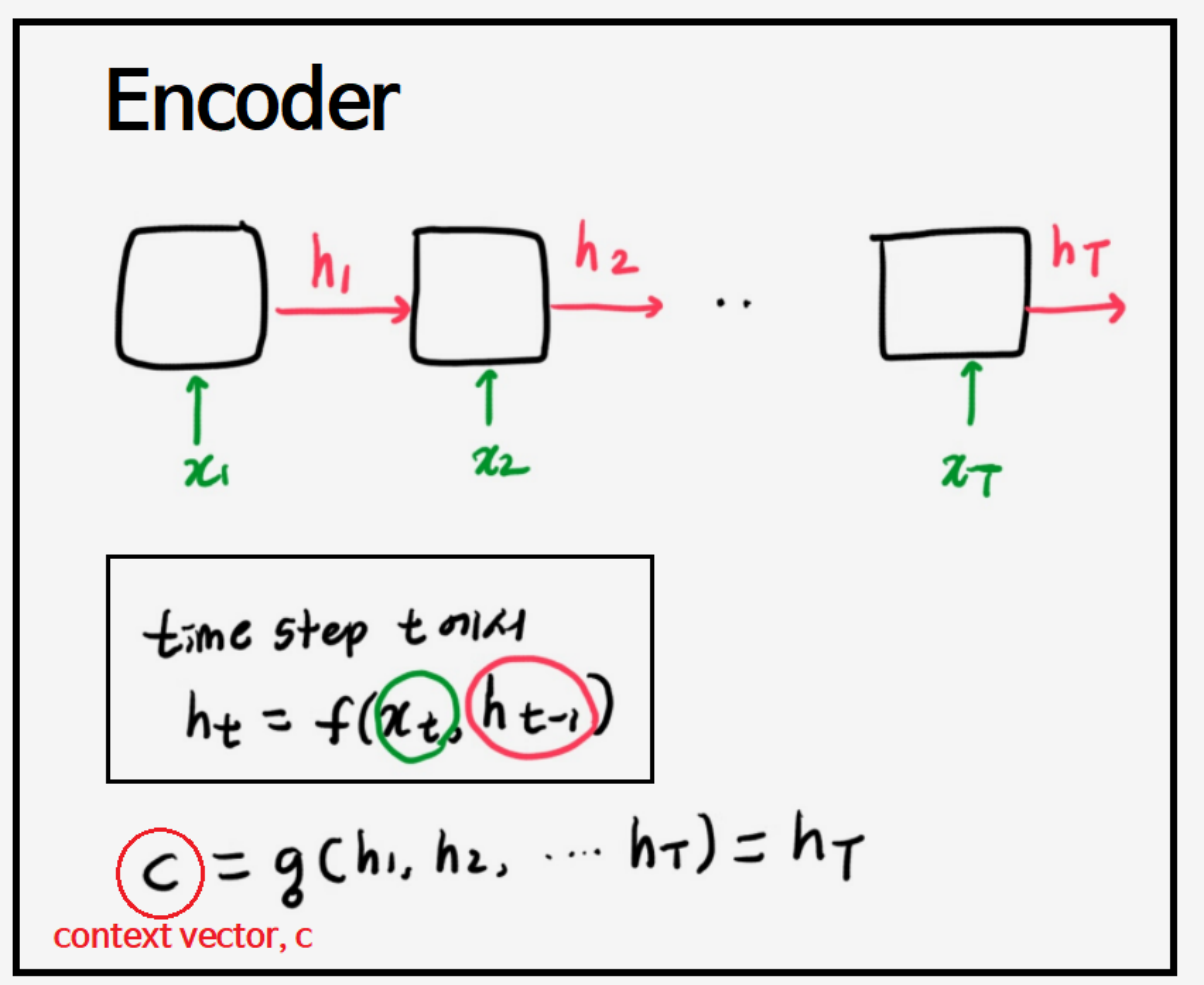
Encoder는 각 time-step의 출력값이 이전 time-step의 영향을 받는 auto-regressive task가 아닌 Non-auto-regressive task에 해당하기 때문에 Bidirectional RNN을 사용할 수 있다.



Encoder unit은 LSTM, GRU와 같은 모델이지만, 예측을 하지 않기 때문에 각 time step별 출력값을 두지 않는다. 이 RNN 모델에서의 유일한 출력값은 마지막 은닉상태인 h(T) 이며, 만일 모델이 LSTM이라면 h(T)와 c(T)가 될 것이다.

h(T) 값은 시간 정보를 가지고 있지 않으며, 오로지 입력값에 대한 표현 정보만을 가지고 있다. 그래서 이것을 'encoding'이라고 부르며 원래의 입력값에 대한 요약된 표현이 된다.

- time step t에서의 은닉상태 h(t)는 입력값 x(t)와 이전 은닉상태의 함수가 된다.



- Decoder

1) Decoder가 하는 일 - Encoder가 압축한 숫자벡터를 받아서 문장 생성

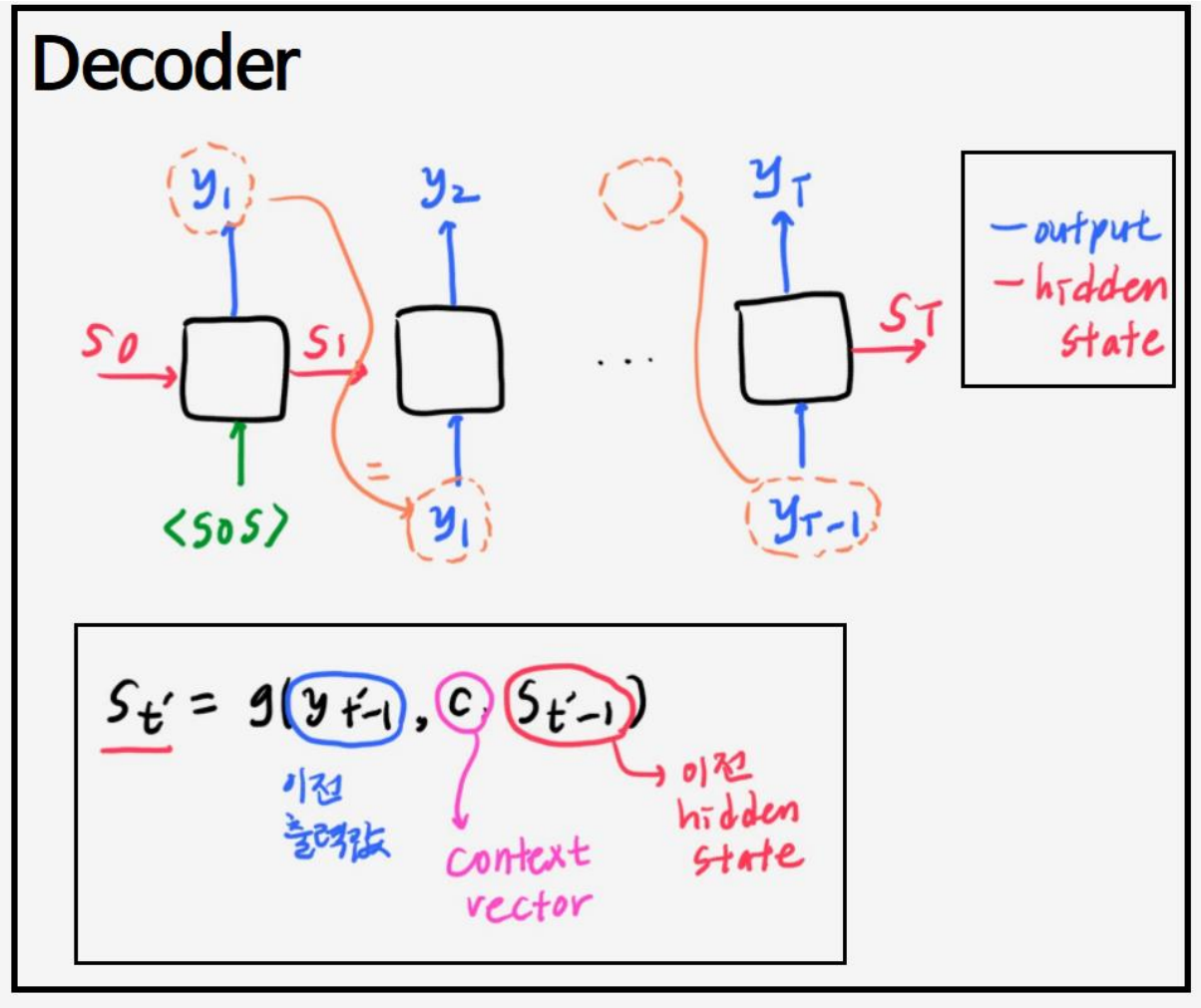
Encoder가 출력한 숫자 벡터(context vector)와 현재까지 출력한 단어들을 바탕으로 다음 단어를 예측

2) Decoder의 특징

Decoder는 출력 시퀀스의 특정 time-step이전의 단어들을 가지고 현재 time-step을 예측하기 때문에 Conditional language model이라고 할 수 있다. Encoder와 다르게 auto-regressive task에 속하기 때문에, Bidirectional RNN은 쓸수 없으며 Uni-directional RNN을 사용할 수 밖에 없다. Decoder는 Encoder RNN과는 다른 고유의 가중치를 가진 RNN unit이다. Encoder의 마지막 은닉상태(hidden-state)를 Decoder의 처음 은닉상태(hidden-state)로 넣는다.

Decoder의 time step t'에서의 출력값(y('t'))의 확률은 context 벡터 c와 이전 출력 시퀀스(y(1),...,y(T))에 따라서 달라진다. Decoder는 Encoder와 별개의 unit이지만 크기는 같아야하는데 그 이유는 Encoder의 유일한 출력값인 은닉상태(h(T))를 입력값으로 받아야하기 때문이다. 즉 Encoder의 출력값인 마지막 step의 h(T)는 Decoder의 첫 입력 은닉상태인 s(0)이 된다. Decoder에

서는 첫 입력값으로 문장의 시작을 알리는 입력값인 '<start-of-sentence>/<sos>' 벡터를 지정한다. 은닉상태 s(t)는 이전 출력값인 y(t'-1)과 convex vector c와 이전 은닉상태 s(t'-1)의 함수이다. 정확하게 이야기하면, Decoder는 단어를 단순히 생성하는것이 아니라, 출력될 수 있는 여러 단어들의 확률을 예측한다. 즉 벡터에 softmax 함수를 써서 확률값으로 변환한 후, 가장 나올 확률이 높은 단어 1개를 출력한다. 이를 최적화를 위해서 cross-entropy loss함수를 사용한다.



- Generator

Generator가 하는 일은 Decoder의 은닉상태(hidden-state)로 현재 time-step의 출력 단어 분포 예측하는 것이다. Decoder의 현재 time-step의 은닉상태(hidden-state)를 받아서 각 단어별 확률(분포)를 예측하고, Cross Entropy Loss로 최적화한다. 은닉상태(hidden-state)과 vocabulary_size는 다르기 때문에, 크기가 (hs, |V|)인 W(gen) 벡터를 곱해준다.

- 활용

앞에서는 기계번역만을 예로 들었는데, Seq2Seq 로 해결할 수 있는 작업은 그 외에도 많다.

1) Question Answering (QA)

이야기(story)와 문제(question)가 주어졌을때, 답(answer)을 생성해내는 문제이다. 일종의 독해력 테스트(test of reading comphrehension)인 것이다. 알버트 아인슈타인에 대한 위키피디아 페이지가 주어졌을 때, "아인슈타인의 이론 중 가장 유명한것이 무엇입니까?"라고 질문한다면 뉴럴네트워크는 "Relativity"라고 대답해야할 것이다. Seq2Seq 구조에서 문제를 본다면, 이야기와 문제가 합해져서 입력 시퀀스가 될 것이고, 이 입력 시퀀스는 인코더를 거쳐서 'thought vector(=context vector)'가 되고 이 thought vector는 디코더를 거치면서 답으로 디코딩 될 것이다.

2) Chatbots(챗봇)

챗봇의 경우, 텍스트로 요청(request)을 하면 반응(response)을 하는 문제로, 기계번역과 동일하게 시퀀스를 입력해서 시퀀스를 받는 문제이다. 챗봇 외에도 텍스트 요약(Text Summarization) 또는 STT(Speech to Text) 등이 있습니다.