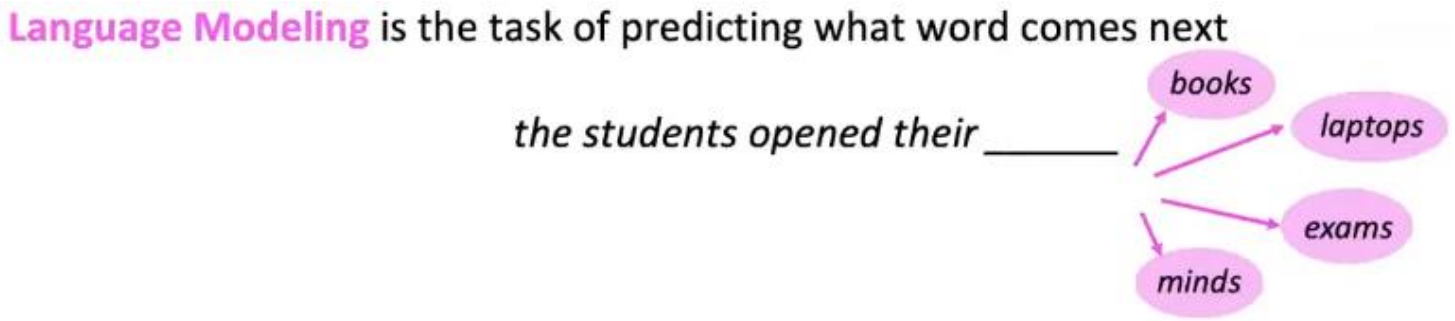
Recurrent Neural Networks and Language Models

ToBig's 20기 전소연

Language Model



- Language Modeling : 주어진 문자열이나 단어들의 순서를 바탕으로 다음 단어를 예측하는 태스크

$$P(\boldsymbol{x}^{(t+1)} | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

- Language Model : 이전 context $(x^{(t)}, \ldots, x^{(1)})$가 주어졌을 때 $x^{(t+1)}$의 probability distribution을 계산해주는 모델.

Language Model

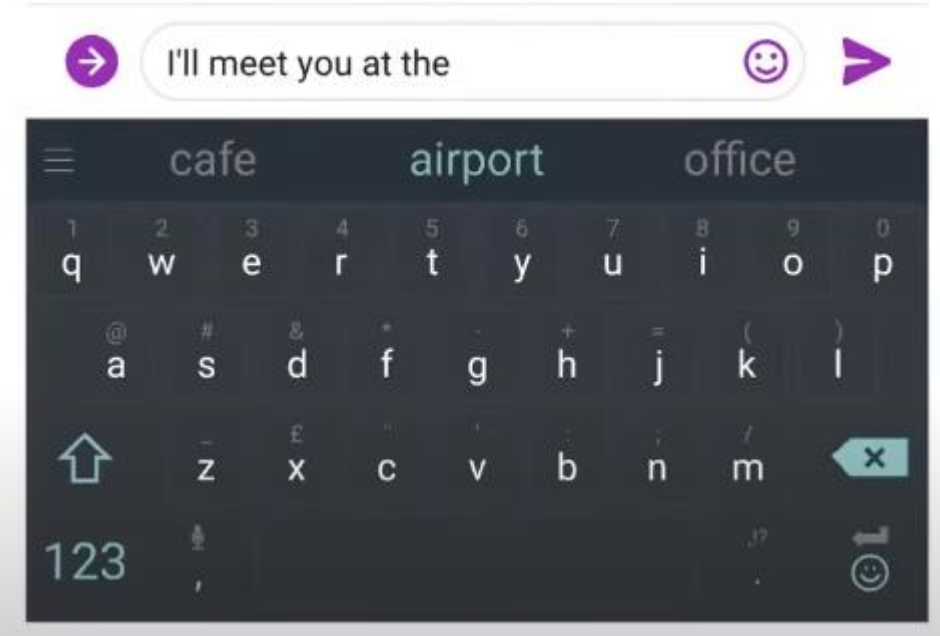
$$P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}) = P(\boldsymbol{x}^{(1)}) \times P(\boldsymbol{x}^{(2)} | \boldsymbol{x}^{(1)}) \times \cdots \times P(\boldsymbol{x}^{(T)} | \boldsymbol{x}^{(T-1)}, \ldots, \boldsymbol{x}^{(1)})$$

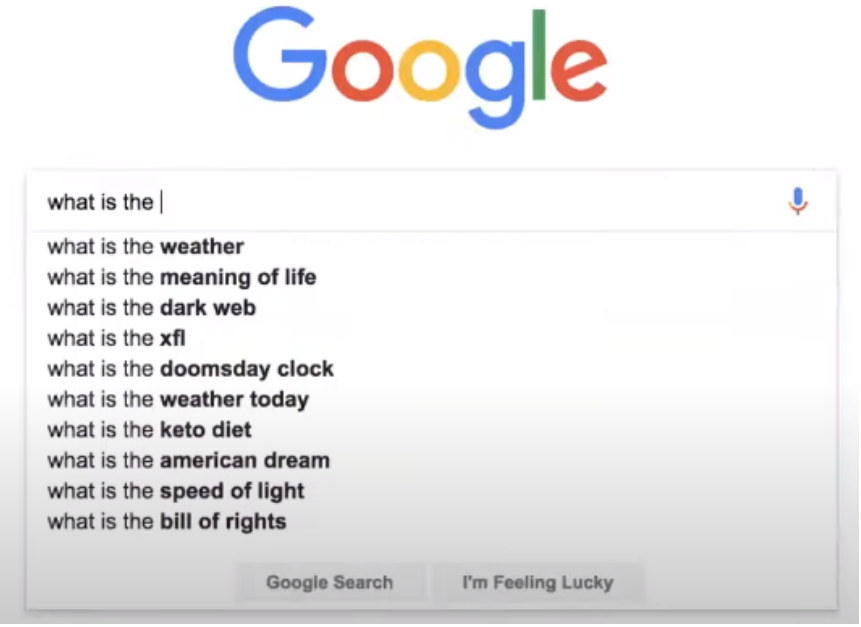
$$= \underbrace{\prod_{t=1}^{T} P(\boldsymbol{x}^{(t)} | \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)})}$$

This is what our LM provides

- 어떤 시퀀스를 얻을 확률을 우변과 같이 각각의 조건부 확률들의 곱으로 분해하여 생각 가능.
- 조건부 확률(t-1 시점까지의 단어들이 주어졌을 때 t 시점의 단어를 얻을 확률)을 LM에 의해 알고 있으므로 시퀀스의 확률을 계산 가능.

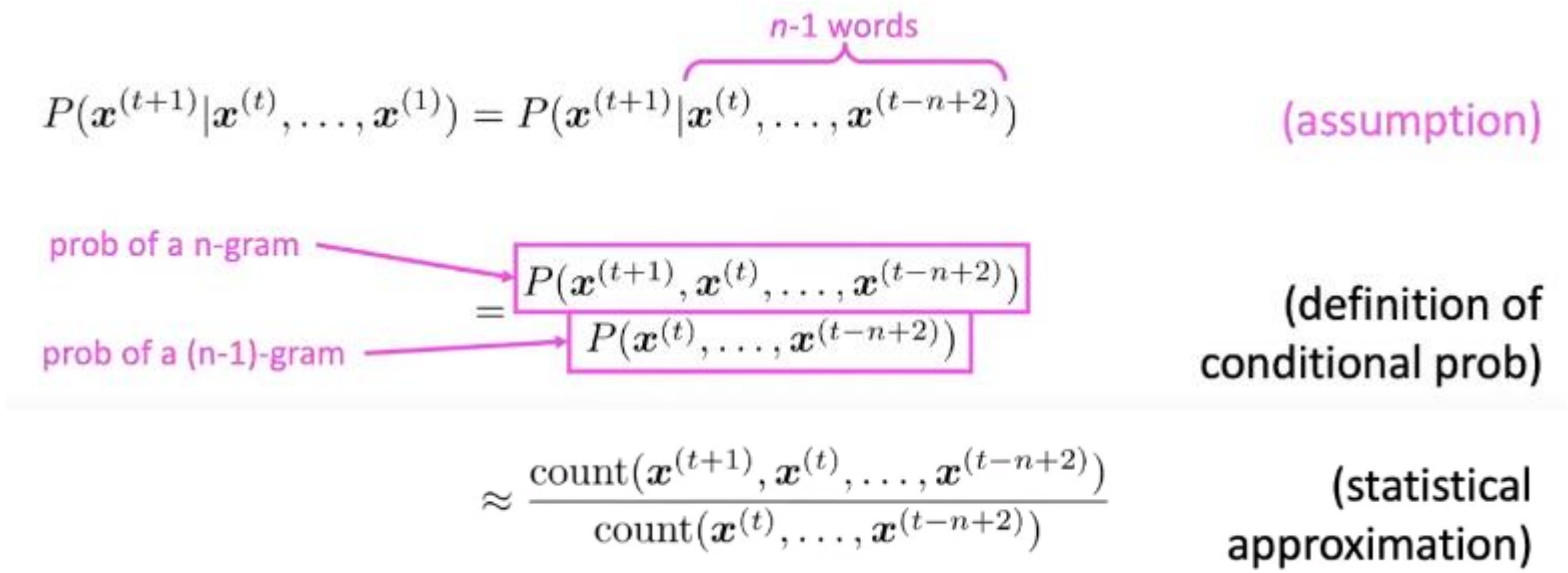
Language Model





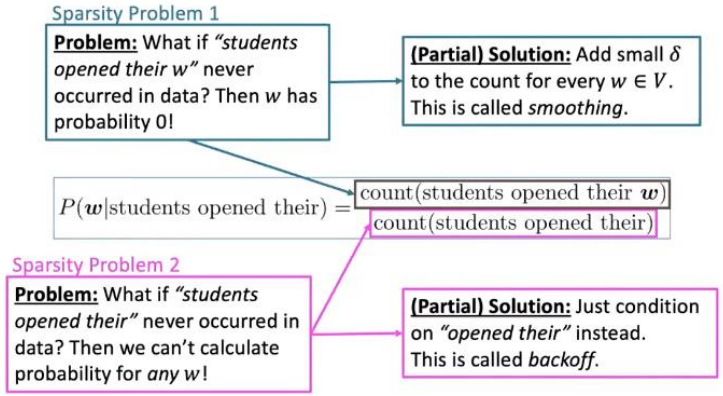
- 자동완성 기능, 검색어 추천 등

N-gram LM

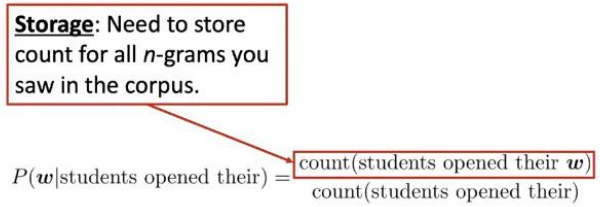


- Deep Learning 등장 이전에 주로 사용.
- 단어가 발생할 확률은 바로 이전의 n-1개 단어의 영향만 받는다고 가정. (그보다 더 먼 단어는 고려하지 않음)
- count(빈도)를 기반으로 확률값을 계산.
- n-gram : n개의 연속적인 단어를 모은 덩어리.
- n의 값에 따라 unigrams, bigrams, trigrams, 4-grams 등으로 불림.

N-gram LM - Problems



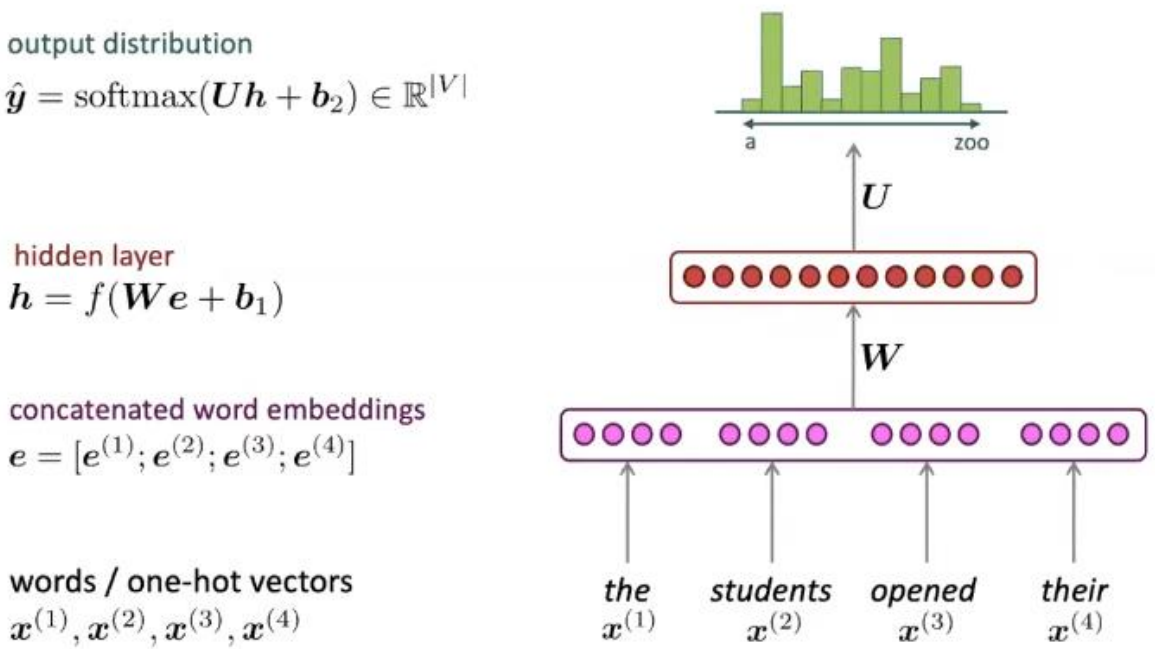
Sparsity problems : 빈도를 구하고자 하는 n-gram 또는 n-1 gram이 텍스트에 없었을 경우 확률이 0이 되거나 아예 계산 불가능.



Storage Problems : 모든 n-gram에 대한 빈도를 저장해야 하므로 n이 커지거나 corpus가 커지면 모델 용량이 너무 커짐.

Incoherence Problems : 자연스럽지 않은 텍스트를 생성할 수 있음. 이를 해결하고자 n을 증가시킬 경우 위의 두 문제가 심해짐.

Fixed-Window Neural Language Model



- 예측하려는 단어보다 이전에서 고정된 window만큼의 단어를 기반으로 다음에 나타날 단어의 확률을 예측.
- 입력 받은 단어들을 각각 one-hot 인코딩 및 임베딩.
- concat하여 hidden layer와 softmax를 통과시켜 vocabulary 내의 모든 단어들에 대한 확률분포를 얻음.

Fixed-Window Neural Language Model

개선된 점

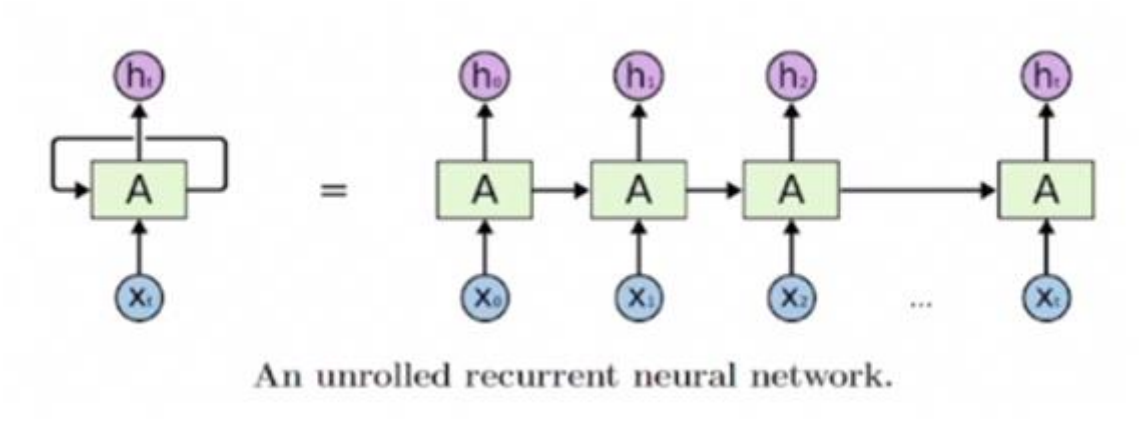
- Vocabulary 내의 모든 단어에 대한 확률분포를 결과로 얻으므로 sparsity problem이 발생 X.
- 모든 n-gram을 저장할 필요가 없다.

해결하지 못한 지점

- window size를 늘리면 가중치 행렬 W가 너무 커지거나 과적합되는 등의 문제로 인해 window size를 크게 할 수 없음. 맥락 정보의 손실이 있을 수 있다.
- 단어의 위치에 따라서 매번 다른 가중치를 사용해야 한다.

→ 어떤 길이의 시퀀스에 대해서도 잘 학습할 수 있는 모델이 필요.

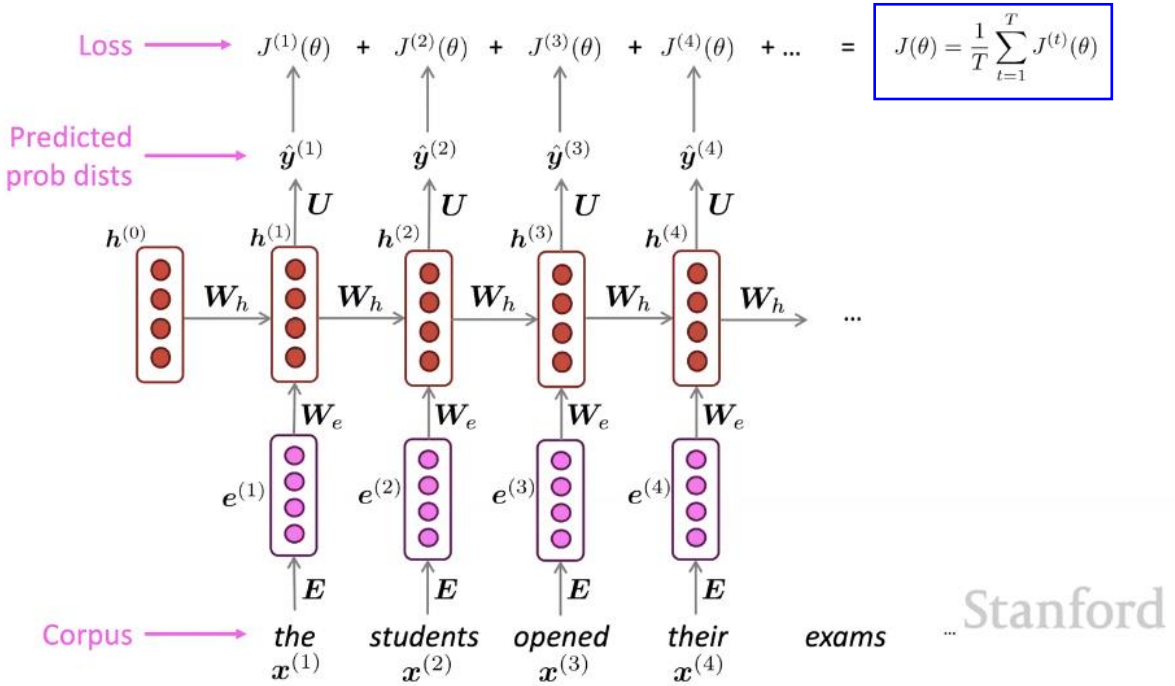
RNN (Recurrent Neural Network)



An unrolled recurrent neural network.

- 어떤 sequence data가 입력 또는 출력으로 들어오는 상황에서, 각 타임스텝에서 들어오는 입력 X_t와 이전 타임스텝의 RNN 모듈에서 계산한 hidden state vector h_{t-1}을 받아서 현재 타임스텝의 hidden state vector를 출력하는 형태를 가지고 있음.
- 즉 현재 타임스텝의 출력 결과는 이전 타임스텝의 출력의 영향을 받음.
- 매 타임스텝마다 **동일한 파라미터를 가진 RNN**을 사용하므로, '재귀적인 호출'의 특성을 보여준다 하여 Recurrent Neural Network라는 이름을 가지게 되었음.

Training a RNN LM



$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e \boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

$$\boldsymbol{W_h}: (d_h \times d_h),\ \boldsymbol{W_e}: (d_e \times d_h)$$

- input : corpus 내의 특정 단어에 대한 one-hot vector
- input 벡터에 대한 임베딩 벡터와 이전 타임스텝의 hidden state vector를 이용, $h^{(t)}$ 를 얻음.
- $h^{(t)}$를 U와의 곱으로 선형변환한 뒤 softmax 를 통과시켜 확률분포를 최종 결과로 얻음.
- 매 타임스텝마다 loss를 계산 후 평균을 내 전체 loss를 계산.
- Loss Function : Cross Entropy (multi-class classification의 경우)

Backpropagation for RNNs

$$\frac{\partial J^{(t)}(\theta)}{\partial W_h} = \frac{\partial J^{(t)}(\theta)}{\partial \hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}} \cdot \frac{\partial h^{(t)}}{\partial W_h}$$

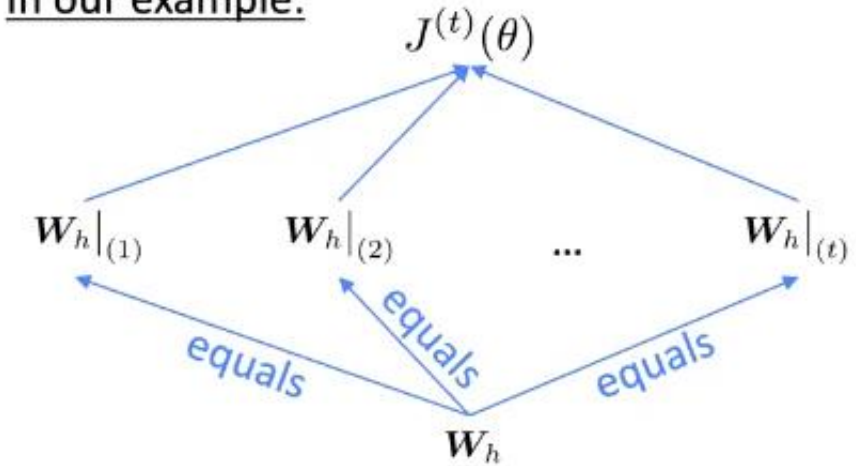
$$+ \frac{\partial J^{(t)}(\theta)}{\partial \hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}} \cdot \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \cdot \frac{\partial h^{(t-1)}}{\partial W_h}$$

$$\vdots$$

$$+ \frac{\partial J^{(t)}(\theta)}{\partial \hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}} \cdot \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \cdot \dots \cdot \frac{\partial h^{(2)}}{\partial h^{(1)}} \cdot \frac{\partial h^{(1)}}{\partial W_h}$$

$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}(\theta)}{\partial \hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}} \cdot \frac{\partial h^{(t)}}{\partial h^{(i)}} \cdot \frac{\partial h^{(i)}}{\partial W_h}$$

In our example:



Apply the multivariable chain rule:

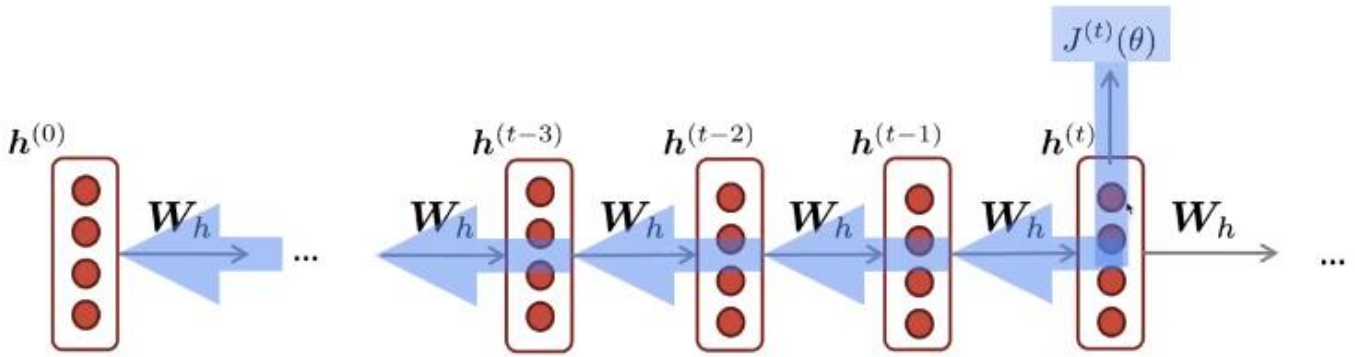
$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\bigg|_{(i)} \frac{\partial W_h|_{(i)}}{\partial W_h}$$

$= 1$

$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\bigg|_{(i)}$$

- RNN은 $h^{(t)}$ 를 계산하기 위해 $h^{(t-1)}, h^{(t-2)}, \cdots, h^{(1)}$까지 관여하므로 $J^{(t)}(\Theta)$의 W_h 에 대한 gradient는 위와 같다. (gradient sum)

Backpropagation for RNNs



$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\Big|_{(i)}$$

Question: How do we calculate this?

Answer: Backpropagate over timesteps *i=t,…,0*, summing gradients as you go. This algorithm is called **"backpropagation through time"** [Werbos, P.G., 1988, *Neural Networks* **1**, and others]

- 이러한 RNN에서의 backpropagation 알고리즘을 BPTT 알고리즘이라고 한다.

Evaluating LM - Perplexity

- The standard evaluation metric for Language Models is perplexity.

$$\text{perplexity} = \prod_{t=1}^{T} \left(\frac{1}{P_{\text{LM}}(\boldsymbol{x}^{(t+1)} | \ \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})} \right)^{1/T}$$

Normalized by number of words

Inverse probability of corpus, according to Language Model

- This is equal to the exponential of the cross-entropy loss $J(\theta)$:

$$= \prod_{t=1}^{T} \left(\frac{1}{\hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}}} \right)^{1/T} = \exp\left(\frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}} \right) = \exp(J(\theta))$$

Lower perplexity is better!

Stanford

Evaluating LM - Perplexity

n-gram model →

Increasingly complex RNNs ↓

Model	Perplexity
Interpolated Kneser-Ney 5-gram (Chelba et al., 2013)	67.6
RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013)	51.3
RNN-2048 + BlackOut sampling (Ji et al., 2015)	68.3
Sparse Non-negative Matrix factorization (Shazeer et al., 2015)	52.9
LSTM-2048 (Jozefowicz et al., 2016)	43.7
2-layer LSTM-8192 (Jozefowicz et al., 2016)	30
Ours small (LSTM-2048)	43.9
Ours large (2-layer LSTM-2048)	39.8

↓ Perplexity improves (lower is better)

Evaluating LM – BLEU score (Bilingual Evaluation Understudy)

$$BLEU = \min(1, \frac{length\_of\_prediction}{length\_of\_reference})(\prod_{i=1}^{4} precision_i)^{\frac{1}{4}}$$

- n=1~4에 대한 각각의 precision의 기하평균
- 기계 번역의 성능이 얼마나 뛰어난지를 측정하기 위해 사용.
- 기계 번역 결과와 사람이 직접 번역한 결과가 얼마나 유사한지 비교하여 번역에 대한 성능을 측정하는 방법.
- 언어에 구애 받지 않고 사용할 수 있으며, 계산 속도가 빠름.
- BLEU는 PPL과는 달리 높을 수록 성능이 더 좋음을 의미.
- 문장의 길이가 짧은 경우를 보정하기 위해, brevity penalty를 적용하여 위에서 구한 기하평균에 곱해준다.
- ** brevity penalty : reference 문장보다 짧은 예측문장을 내놓을 시, 1이하의 값을 곱해서 precision의 값을 낮게 보정.

Gradients Vanishing

(linear case)

Consider the gradient of the loss $J^{(i)}(\theta)$ on step i, with respect to the hidden state $\boldsymbol{h}^{(j)}$ on some previous step j. *Let* $\ell = i - j$

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

$$= \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \boldsymbol{W}_h = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{\ell} \qquad \text{(value of } \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}\text{)}$$

If W_h is "small", then this term gets exponentially problematic as ℓ becomes large

- What's wrong with $\boldsymbol{W}_h^{\ell}$?
- Consider if the eigenvalues of $\boldsymbol{W}_h$ are all less than 1: (sufficient but not necessary)

$$\lambda_1, \lambda_2, \ldots, \lambda_n < 1$$
$$\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n \text{ (eigenvectors)}$$

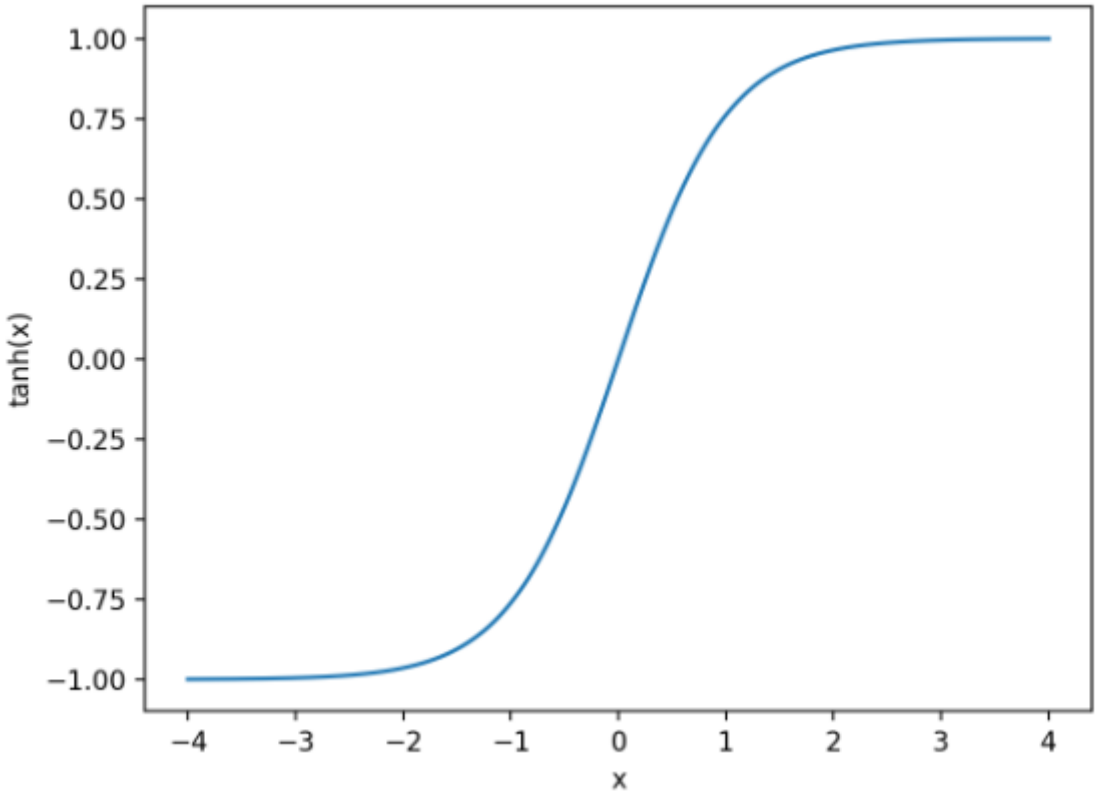
- We can write $\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{\ell}$ using the eigenvectors of $\boldsymbol{W}_h$ as a basis:

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{\ell} = \sum_{i=1}^{n} c_i \lambda_i^{\ell} \boldsymbol{q}_i \approx \boldsymbol{0} \text{ (for large } \ell\text{)}$$

Approaches 0 as ℓ grows, so gradient vanishes

Gradients Vanishing

(Non-linear case)



- Activation function이 non-linear 함수인 일반적인 경우 W의 고유값 외에도 activation function 의 특성에 의해 backpropagation의 영향이 있을 수 있음.
- ex) input이 -1과 1인 경우 gradient 의 차이가 매우 큰 반면 input이 2와 4인 경우 차이가 크지 않음.

Exploding Gradients

Toy Example

- $h_t = \tanh(w_{xh}x_t + w_{hh}h_{t-1} + b), t = 1,2,3$
- For $w_{hh} = 3, w_{xh} = 2, b = 1$

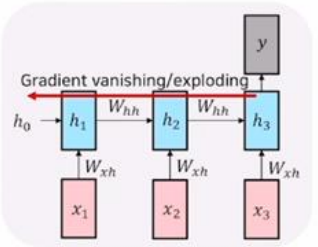
$h_3 = \tanh(2x_3 + 3h_2 + 1)$

$h_2 = \tanh(2x_2 + 3h_1 + 1))$

$h_1 = \tanh(2x_1 + 3h_0 + 1))$

...

$h_3 = \tanh(2x_3 + 3\tanh(2x_2 + 3h_1 + 1) + 1)$



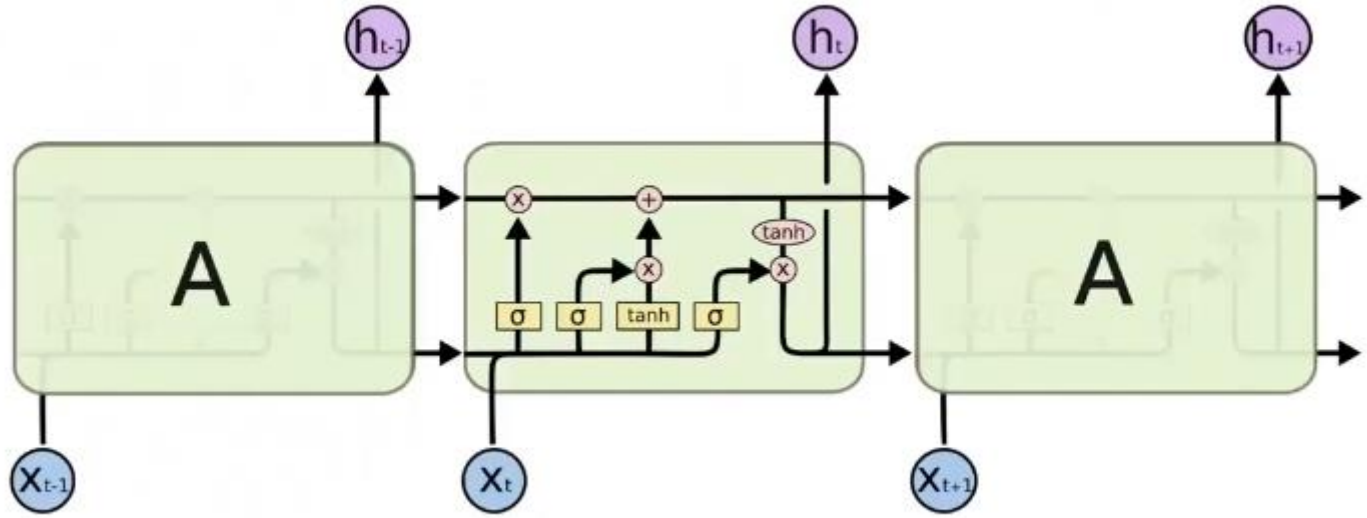
- **Gradient clipping**: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

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

- **Intuition**: take a step in the same direction, but a smaller step

LSTM (Long Short-Term Memory)



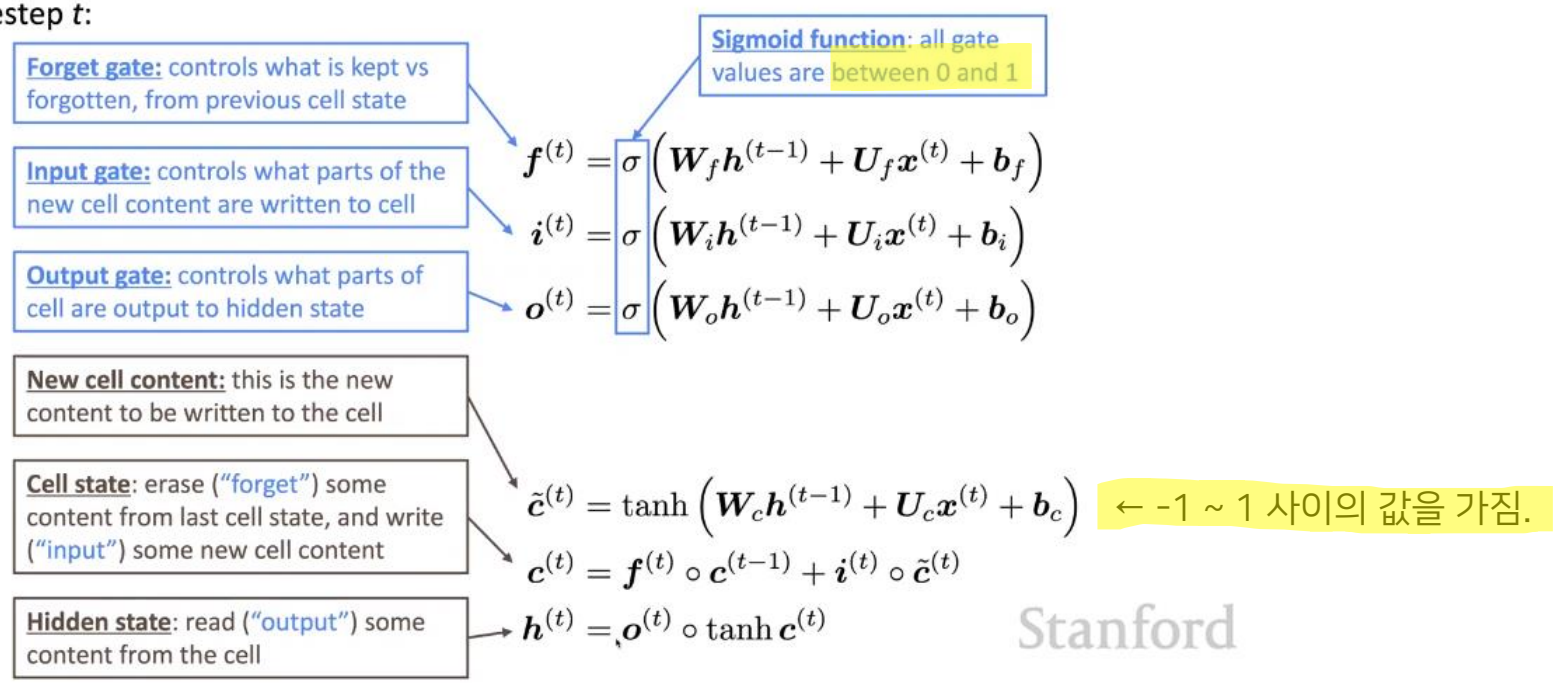
• RNN의 h_t는 단기 기억을 저장하는 소자라고 볼 수 있는데 LSTM은 시퀀스가 타임스텝에 의해 진행됨에 따라 이 단기 기억을 보다 길게 기억할 수 있도록 개선했다는 의미를 이름에 담고 있음.

• Cell state에는 핵심 정보들을 모두 담아두고, 필요할 때마다 hidden state를 가공해 time step에 필요한 정보만 노출하는 형태로 정보가 전파됨.

• hidden state vector : cell state vector를 한번 더 가공하여 그 타임스텝에서 노출할 필요가 있는 정보만 남긴 필터링된 정보. 현재 타임스텝에서의 예측값을 계산해주는 다음 output layer의 입력으로 사용됨.

LSTM (Long Short-Term Memory)

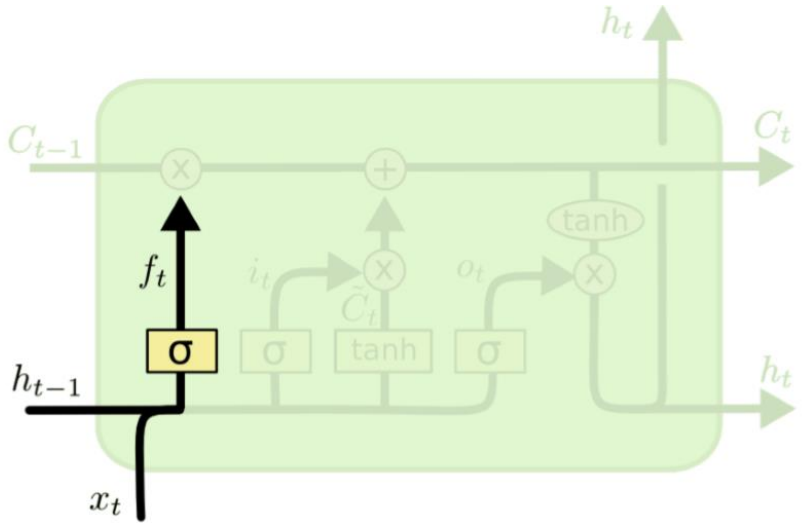
We have a sequence of inputs $x^{(t)}$, and we will compute a sequence of hidden states $h^{(t)}$ and cell states $c^{(t)}$. On timestep t:



← -1 ~ 1 사이의 값을 가짐.

- Forget gate : f_t는 c_{t-1}의 각 dimension에 대해 얼만큼씩 기억할지를 결정한다.
- Input gate : 현재의 정보를 cell에 반영할지 말지를 결정.
- Output gate : cell 정보를 hidden state 에 어느 정도 사용할지를 결정.

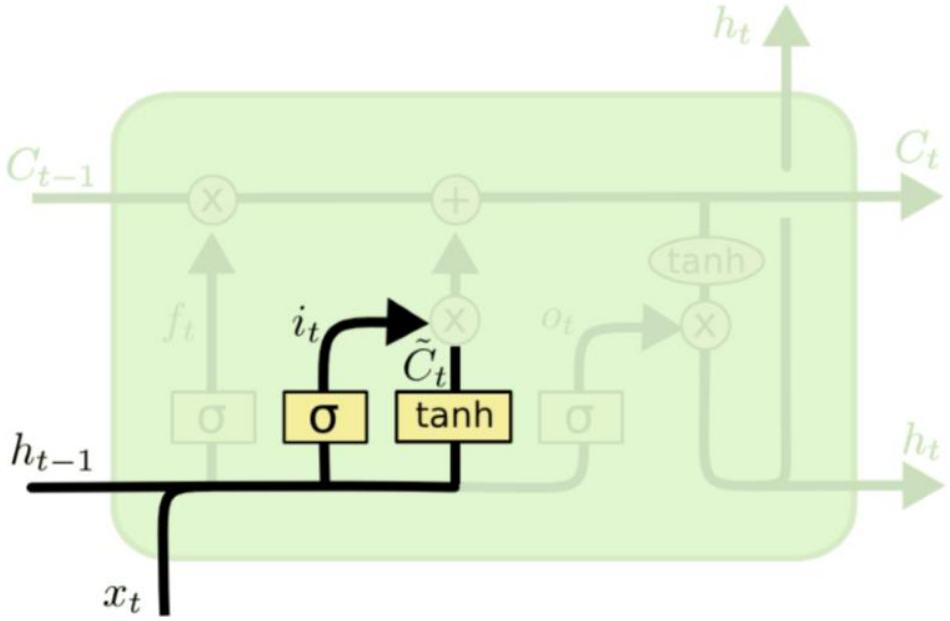
LSTM (Long Short-Term Memory) – Forget Gate



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

- sigmoid를 통과시켜 구한 f_t와 c_{t-1}을 element-wise로 곱해 벡터를 얻는다. 예를 들어 f_t = [0.7 0.4 0.8] 이라 한다면 c_{t-1}의 첫 번째 요소에 대해서는 70%만 기억하겠다는 의미가 된다. 즉 f_t는 c_{t-1}의 각 dimension에 대해 얼만큼씩 기억할지를 결정한다.

LSTM (Long Short-Term Memory) – Input Gate, New Cell Content

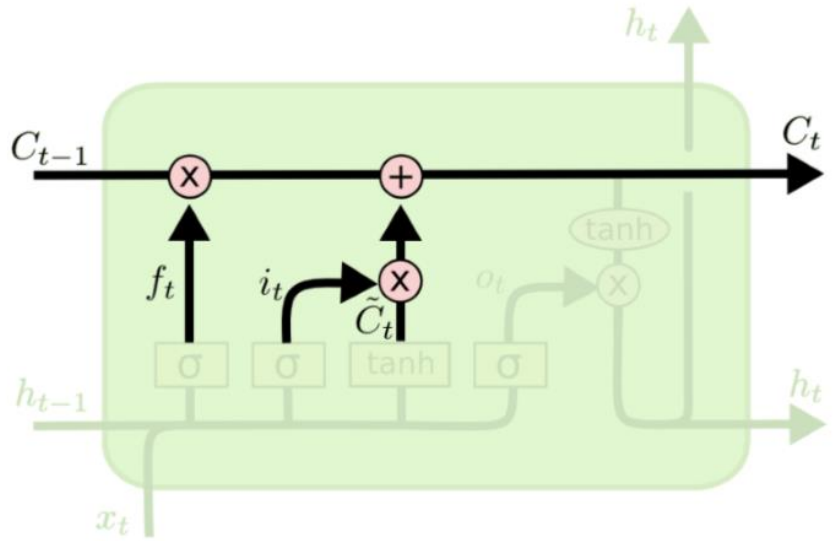


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

<- 현재 타임스텝에서 LSTM에서 계산되는 유의미한 정보라고 생각할 수 있다.

LSTM (Long Short-Term Memory) – New Cell State



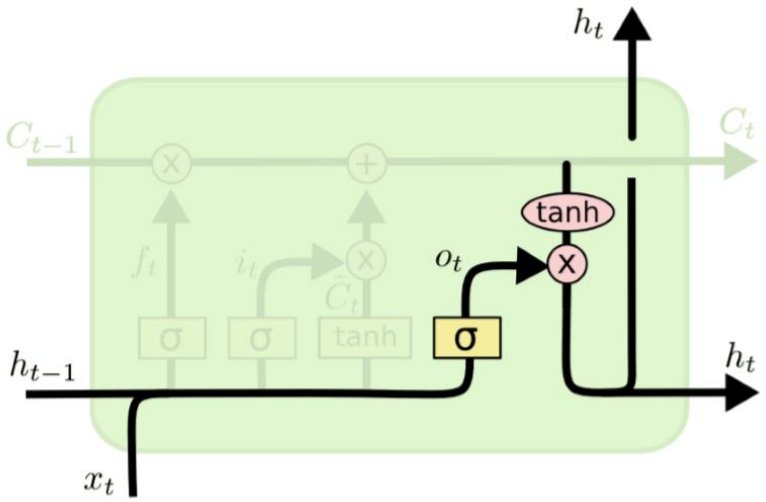
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

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

- 현재 타입스텝의 cell state C_t : C_{t-1}에서 일부 정보를 지운 f_t C_{t-1}에 i_t $\tilde{C}_t$를 더하여 구한다.
- 왜 $\tilde{C}_t$를 바로 더해주지 않고 i_t를 곱하는가? : 한번의 선형변환으로 C_{t-1}에 더해줄 정보를 만들기 어려운 경우에는 정해주고자 하는 값보다 좀 더 큰 값들로 구성된 정보를 gate gate의 형태로 만들어준 후, 각 dimension별로 또 특정 비율을 덜어내서 두 단계에 걸쳐 C_{t-1}에 더해줄 정보를 만드는 것이다.

LSTM (Long Short-Term Memory) – Output Gate



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

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

- Output gate : cell state의 어느 부분을 output으로 내보낼지를 결정
- 이제 h_t를 위와 같이 output gate와 tanh(C_t)를 곱하여 구한다.
- h_t는 다음 RNN에 input으로 전해짐과 동시에 위로 올라가는 path를 가지고 있는데 이 경로가 바로 예측을 위해 output layer의 입력으로 주어지는 의미한다.

How does LSTM Solve Vanishing Gradient?

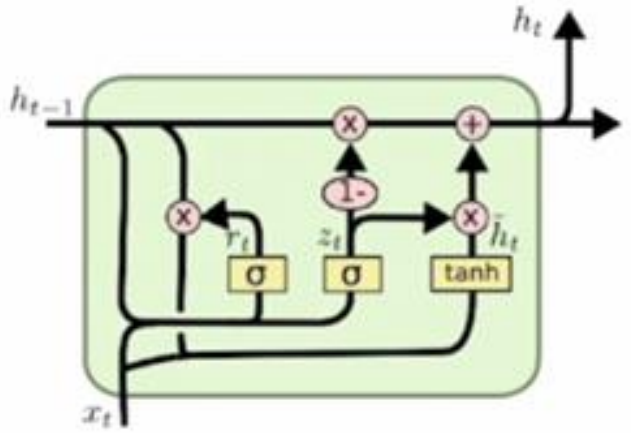
How does LSTM solve vanishing gradients?

- The LSTM architecture makes it easier for the RNN to preserve information over many timesteps
  - e.g., if the forget gate is set to 1 for a cell dimension and the input gate set to 0, then the information of that cell is preserved indefinitely.

- Forget gate 값이 1, input gate 값이 0이라면 t 시점의 cell state 를 구할 때 t-1 시점의 cell state가 완벽하게 보존됨.
- 매번 동일한 가중치 행렬을 곱해주는 것이 아니라 매 타임스텝에서 cell state를 구할 때마다 다른 forget gate 를 곱하고, 필요로 하는 정보를 곱셈이 아닌 덧셈을 통해 얻는다는 점에서 gradient vanishing 을 해결.
- 그러나 LSTM 또한 시퀀스가 길어지면 그래디언트 소실 문제를 완전히 해결할 수는 없다.

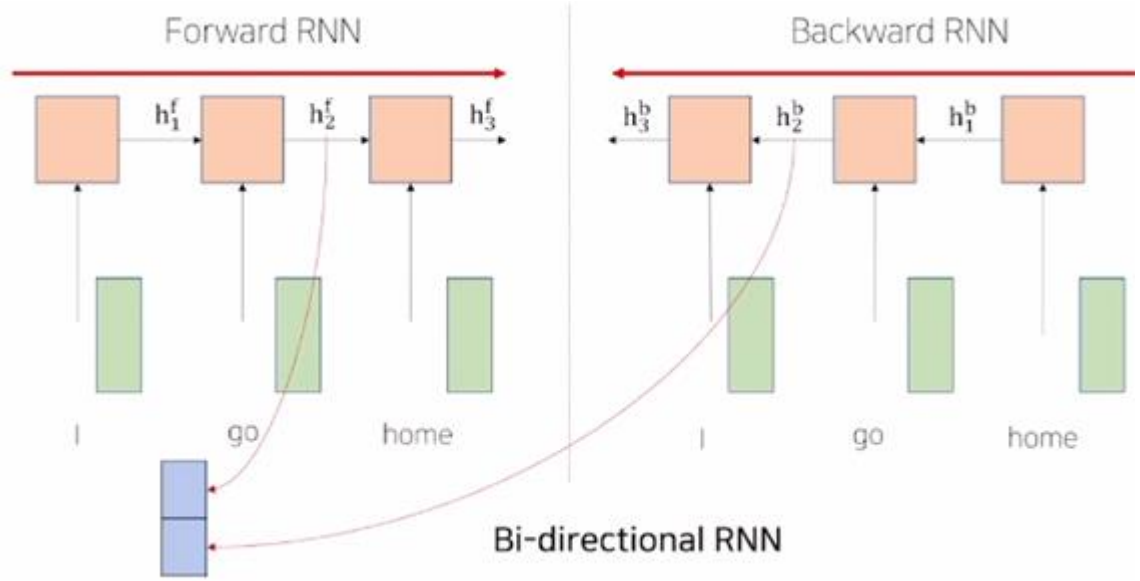
GRU (Gated Recurrent Units)

- $z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$
- $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$
- $\tilde{h_t} = \tanh(W \cdot [r_t \cdot h_{t-1}, x_t])$
- $h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h_t}$
- c.f) $C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C_t}$
  in LSTM



- LSTM의 모듈 구조를 정량화해서 보다 적은 메모리 사용량과 빠른 계산이 가능하게끔 만든 모델.
- 특징 1) LSTM에 존재하던 cell state vector와 hidden state vector를 일원화하여 hidden state vecotr만이 존재한다.
  GRU에서 h_t는 LSTM의 cell state vector C_t와 조금 더 비슷한 역할을 한다고 볼 수 있다.
- 특징 2) GRU에서는 input gate 만 사용한다. 위 식의 z_t가 input gate에 해당한다. forget gate 자리에는 (1-z_t)를 사용한다.

Bidirectional RNNs



On timestep t:

This is a general notation to mean "compute one forward step of the RNN" – it could be a vanilla, LSTM or GRU computation.

Forward RNN $\overrightarrow{\boldsymbol{h}}^{(t)} = \boxed{\mathrm{RNN}_{\mathrm{FW}}}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$

Backward RNN $\overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$

Generally, these two RNNs have separate weights

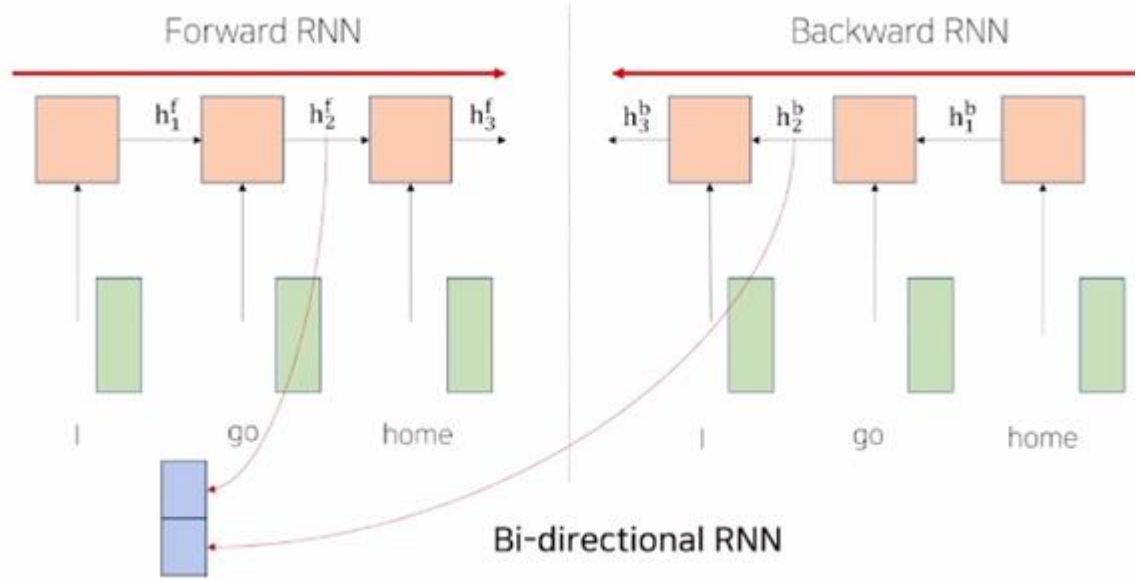
Concatenated hidden states $\boldsymbol{h}^{(t)} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$

We regard this as "the hidden state" of a bidirectional RNN. This is what we pass on to the next parts of the network.

- forward RNN과 backward RNN 이 두 개의 별개의 모듈을 병렬적으로 만든다.

  각각에서 얻은 hidden state vector를 concat 하여 차원을 두배로 만든 벡터가 최종 hidden state vector.

Bidirectional RNNs



On timestep t:

This is a general notation to mean "compute one forward step of the RNN" – it could be a vanilla, LSTM or GRU computation.

Forward RNN $\overrightarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{FW}}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$

Backward RNN $\overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$

Generally, these two RNNs have separate weights

Concatenated hidden states $\boldsymbol{h}^{(t)} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$

We regard this as "the hidden state" of a bidirectional RNN. This is what we pass on to the next parts of the network.

- forward RNN과 backward RNN 이 두 개의 별개의 모듈을 병렬적으로 만든다.

  각각에서 얻은 hidden state vector를 concat 하여 차원을 두배로 만든 벡터가 최종 hidden state vector.

Reference

- **cs224n winter 2021 Lecture 5 - Recurrent Neural networks (RNNs) , Lecture 6 - Simple and LSTM RNNs**
  https://www.youtube.com/watch?v=0LixFSa7yts&list=PLoROMvodv4rMFqRtEuo6SGjY4XbRIVRd4&index=6

- **부스트코스 자연어처리의 모든 것**
  https://www.boostcourse.org/ai330/lecture/1455688?isDesc=false

- https://yjg-lab.tistory.com/241

- https://wikidocs.net/31695