CS 224n Summary

1. Word Vector

motivation: To understand natural language, a symbol must be introduced to represent each word or phase. The representation can be a one-hot vector where 1 is positioned in the word and 0 otherwise.

Issue: One-hot vector is sparse and orthogonal, thus two similar words(e.g. happy and joyful) has a distinctive word representation.

word2vec algorithm: Inspired by distributional semantics, which says a word's meaning is determined by the words that frequently appear near-by, word2vec(word embedding) algorithm tries to build up a word vector by its context words.

Continuous Bag of Words (CBOW): The model takes a window-sized context words(outside words) embeddings $v_{c-m}, v_{c-m+1}, \cdots, v_{c+m-1}, v_{c+m}$, , average them to get $\hat{v}$. Then, generate the center word embedding u_c, and if the averaged outside word and center word are in great similarity, the dot product between u_c and $\hat{v}$ achieves a higher score. After applying softmax function, we get predicted distribution $\hat{y}$. For evaluation, we derive the loss function for a single word: $H(\hat{y}, y) = -y_i \log \hat{y}_i$. In terms of whole text, our objective function is:

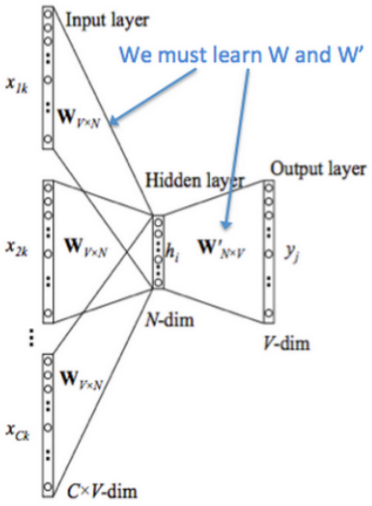
$$L(\theta) = \prod_{t=1}^{T} P(u_c | v_{c-m}, v_{c-m+1}, \cdots, v_{c+m-1}, v_{c+m}; \theta)$$

$$= \prod_{t=1}^{T} P(u_c | \hat{v}; \theta)$$

$$J(\theta) = -\frac{1}{T} \log L(\theta)$$

$$= -\frac{1}{T} \sum_{t=1}^{T} \log L(\theta)$$

$$= -\frac{1}{T} \sum_{t=1}^{T} \log P(u_c | \hat{v}; \theta)$$



Skip-Gram: Conversely, this model takes a center word embedding v_c as input and output a window-sized words $u_{c-m}, u_{c-m+1}, \cdots, u_{c+m-1}, u_{c+m}$. After passing a softmax unit, we have the probability distributions for each outside word $\hat{y}_{c-m}, \hat{y}_{c-m+1}, \cdots, \hat{y}_{c+m-1}, \hat{y}_{c+m}$. To evaluate the error between true labels $y_{c-m}, y_{c-m+1}, \cdots, y_{c+m-1}, y_{c+m}$, we have the loss function for a single word:

$$H(\hat{y}, y) = \sum_{\substack{j=0 \\ j \neq m}}^{2m} -y_{c-m+j} \log \hat{y}_{c-m+j}$$

In terms of the whole context, we have

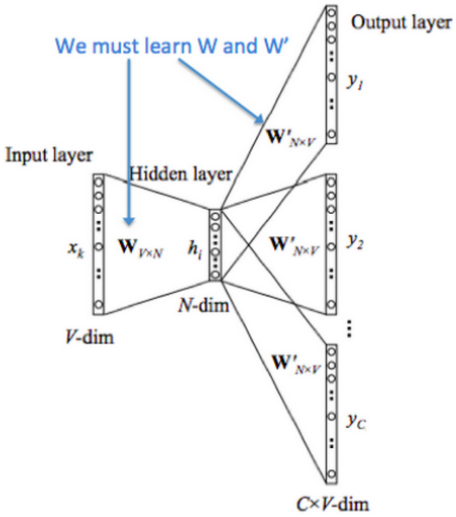
$$L(\theta) = \prod_{t=1}^{T} P(u_{c-m}, u_{c-m+1}, \cdots, u_{c+m-1}, u_{c+m} | v_c; \theta)$$

$$= \prod_{t=1}^{T} \prod_{\substack{j=0 \\ j \neq m}}^{2m} P(u_{c-m+j} | v_c; \theta)$$

$$J(\theta) = -\frac{1}{T} \log L(\theta)$$

$$= -\frac{1}{T} \sum_{t=1}^{T} \log L(\theta)$$

$$= -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{j=0 \\ j \neq m}}^{2m} \log P(u_{c-m+j} | v_c; \theta)$$



negative sampling: In order to mitigate the calculation of denominator of softmax function, we randomly sample K negative examples and our objective is to train K binary classifiers, thus we have loss function:

For CBOW:

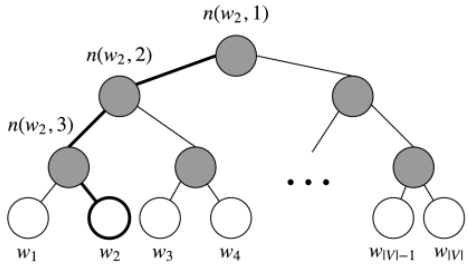
$$J(\theta) = -\log \sigma(u_c^T \cdot \hat{v}) - \sum_{k=1}^{K} \log \sigma(-\tilde{u}_k^T \cdot \hat{v})$$

For Skip-Gram:

$$J(\theta) = -\log \sigma(u_o^T \cdot v_c) - \sum_{k=1}^{K} \log \sigma(-\tilde{u}_k^T \cdot v_c)$$

where is the outside word u_o for the center word v_c.

hierarchical softmax: Build up a tree structure where each leaf node represents a word in the vocabulary. Each time we compute softmax, we traverse the softmax tree from the root, so the computation order can be reduced to $O(\log|V|)$.



Latent Semantic Analysis(LSA): The count-based model, which is different from prediction-based model like word2vec, counts the number of co-occurrence between word i and word j within a fixed window size and construct a co-occurrence matrix.

e.g. Suppose the following is our corpus.

“I enjoy flying.”

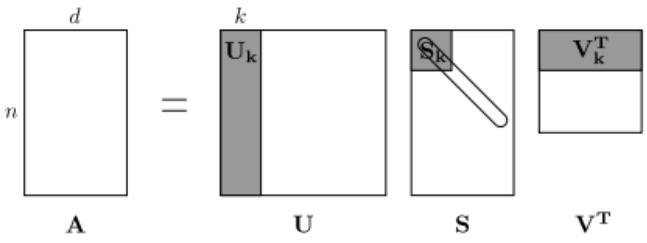
“I like NLP.”

“I like deep learning.”

the resulting co-occurrence matrix X is:

$$X = \begin{array}{c} \\ I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array} \begin{array}{c} \begin{array}{cccccccc} I & like & enjoy & deep & learning & NLP & flying & . \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

However, storing the entire $\mathbb{R}^{|V|\times|V|}$ entries is not efficient for the memory. An appliable method to reduce dimensionality is SVD(selecting the first k singular value).



The problems still exist when computing SVD to co-occurrence matrix. One is expensive computational cost(quadratic order) for SVD; another is each word in the window size has the same weight accounting for the center word.

GloVe(Global Vector for Word Representation): GloVe model tries to find the relations between word frequency and word similarity. Specifically, let X_{ij} be the number word j as a context occurring in word i, X_i be the total number of words occurring in word i, and $P_{ij} = P(j|i) = X_{ij}/X_i$ be the frequency of word j occurring in word i. Rather than building up word vector in a single word occurring frequency P_{ij}, GloVe takes the ratio of the two word occurring frequency P_{ik}/P_{jk} to evaluate the relevance of two words i and j.

e.g.

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Thus we have a loss function:

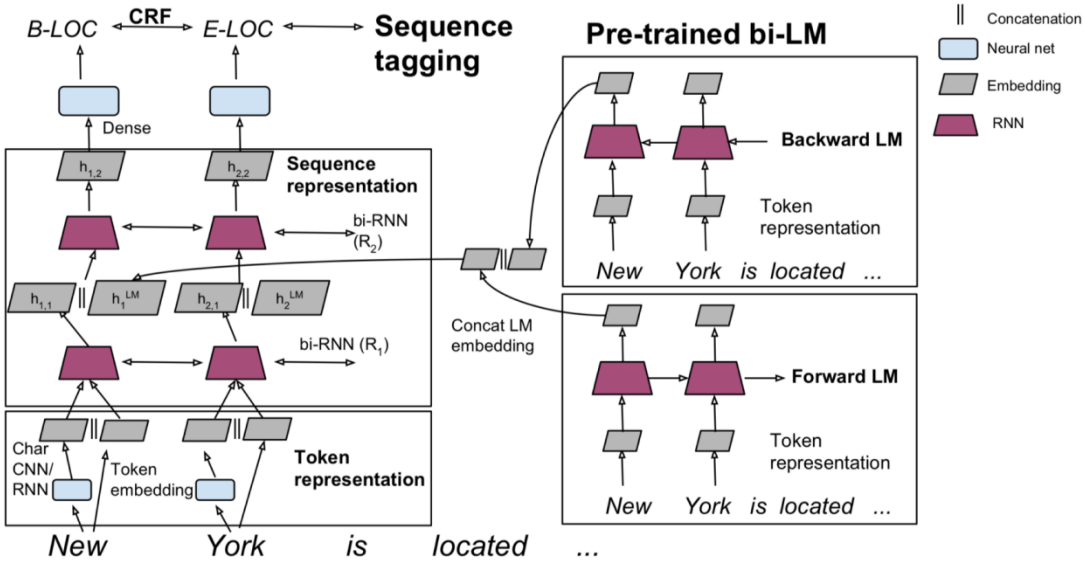
$$J = \sum_{i,j=1}^{|V|} f(X_{ij})\,(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log(X_{ij}))^2$$

where w_i and $\tilde{w}_j$ are two word vectors to be trained, b_i and $\tilde{b}_j$ are bias terms to ensure the similarity of two vectors and co-occurrence are symmetry. $f(X_{ij})$ is a weighted function that brings a larger value for frequent word pairs and lower for infrequent ones.

$$f(x) = \begin{cases} (x/x_{max})^{\alpha} & \text{if } x < x_{max} \\ 1 & \text{otherwise} \end{cases}$$

where α has a typical number $3/4$ and x_{max} is often 100.

Tag LM: embed a word by the hidden state in a pretrained language model, so the word is represented in a specific context. Then, concatenate the context-specific vector h_k^{LM} with the hidden state vector h_k, $[h_k^{LM}, h_k]$ to embed a specific word for downstream task.



Embedding from Language Models(ELMo): If our pretrained language model has multiple layers, unlike Tag LM which only uses the last layer's representation, we select all layers and weighted them by a factor s_j^{task}.

For each token k, first we compute the language model token representation R_k:

$$R_k = \{x_k^{LM}, \overrightarrow{h}_{k,j}^{LM}, \overleftarrow{h}_{k,j}^{LM} \mid j = 1,2,\cdots L\}$$

where x_k^{LM} is the representation of the word k in language model, i.e. $x_k^{LM} = h_{k,0}^{LM}$. Thus we have:

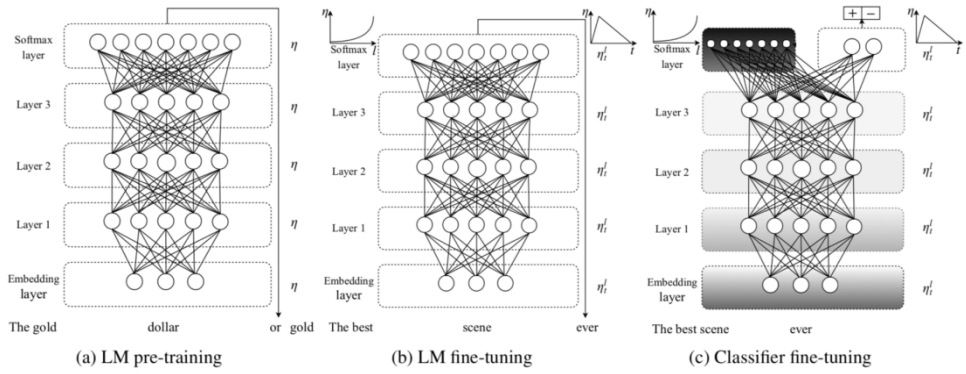
$$R_k = \{h_{k,j}^{LM} \mid j = 0,1,\cdots L\}$$

For ELMo embedding, there is parameter γ^{task}, where it sets to different value for different task.

$$ELMo_k^{task} = E(R_k, \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} h_{k,j}^{LM}$$

Contextualized Word Vector(CoVe): pretrain a seq2seq machine translation model with attention mechanism, and use the encoder to produce context word representation for a specific word. Finally, combine static word embedding(Word2Vec, GloVe, etc.) to contextualized word embedding.

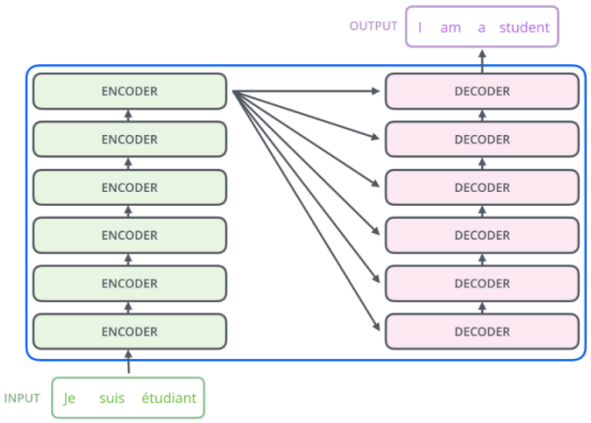
ULMFiT: transfer learning in NLP. The pretrained language model is trained on a large language corpus, then fine-tune language model on target task data. Finally, the fine-tuned language model is used on a specific task(text classification).



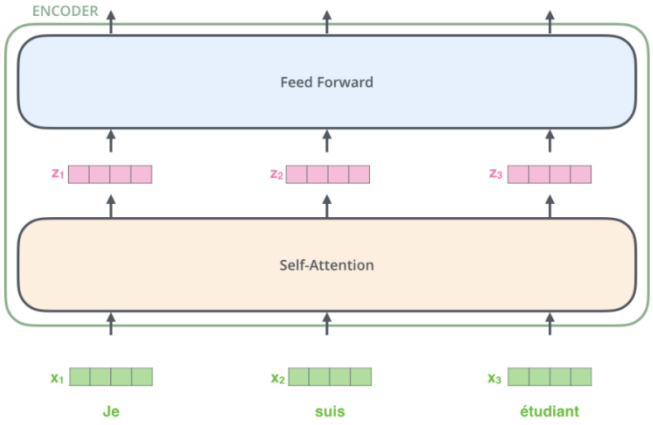
(a) LM pre-training (b) LM fine-tuning (c) Classifier fine-tuning

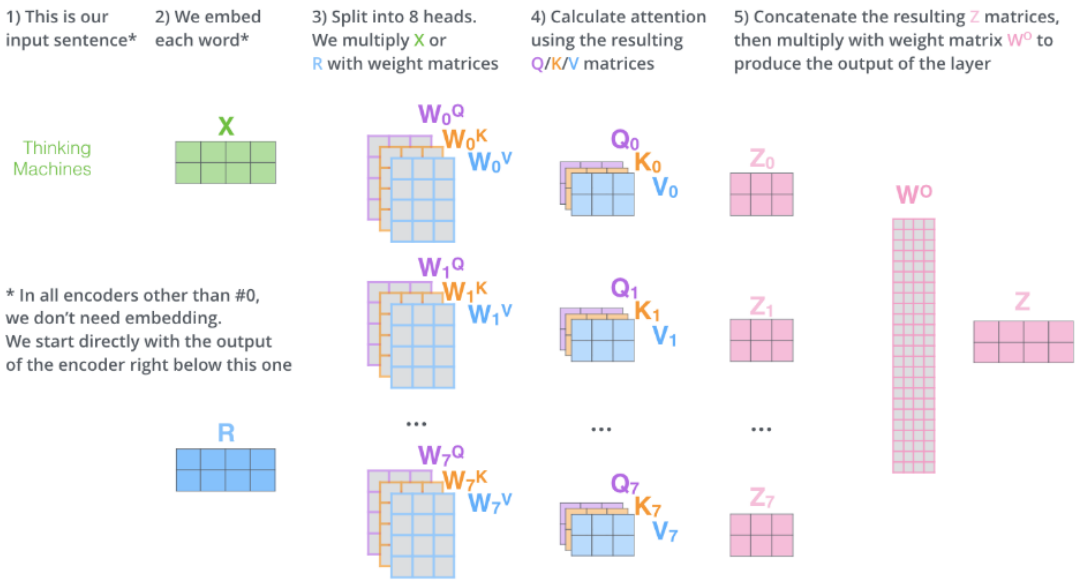
Transformer:

The model has two components: encoder and decoder, each of which is stacked with each other.

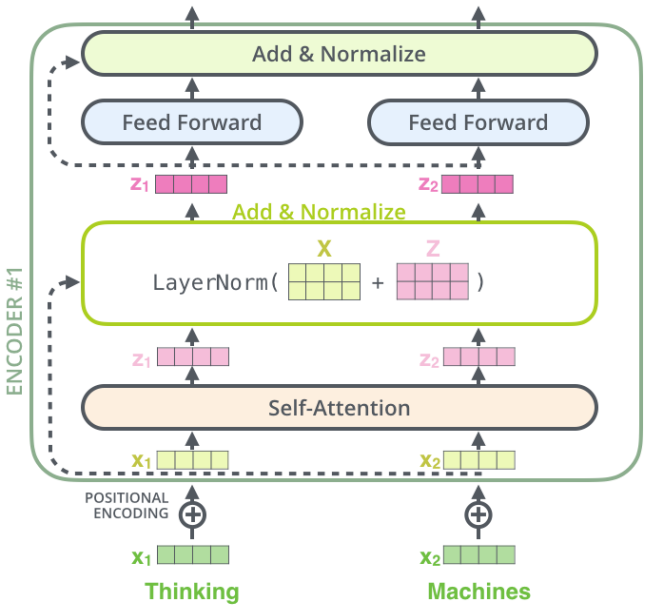


The encoder has two sub-layers, one is self-attention, which allows encoder to look at other words each time the encoder encodes a specific word. Instead of only one attention mechanism, there are multiple of them(multi-head attention), enabling our model paying attention to multiple position of a sentence. Another part is a fully-connected layer which maps the dimension of output of multi-head attention to another dimension.

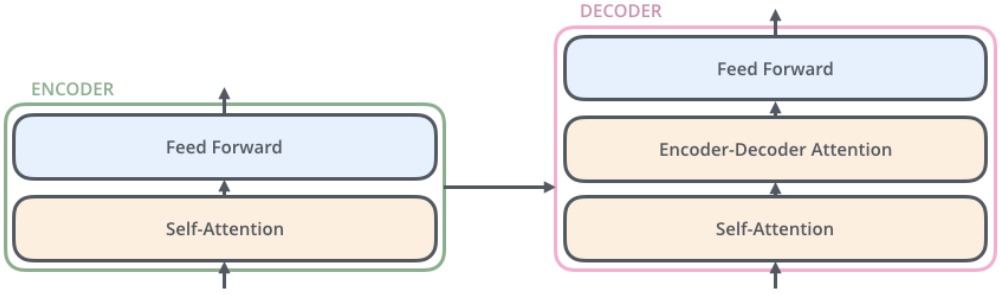


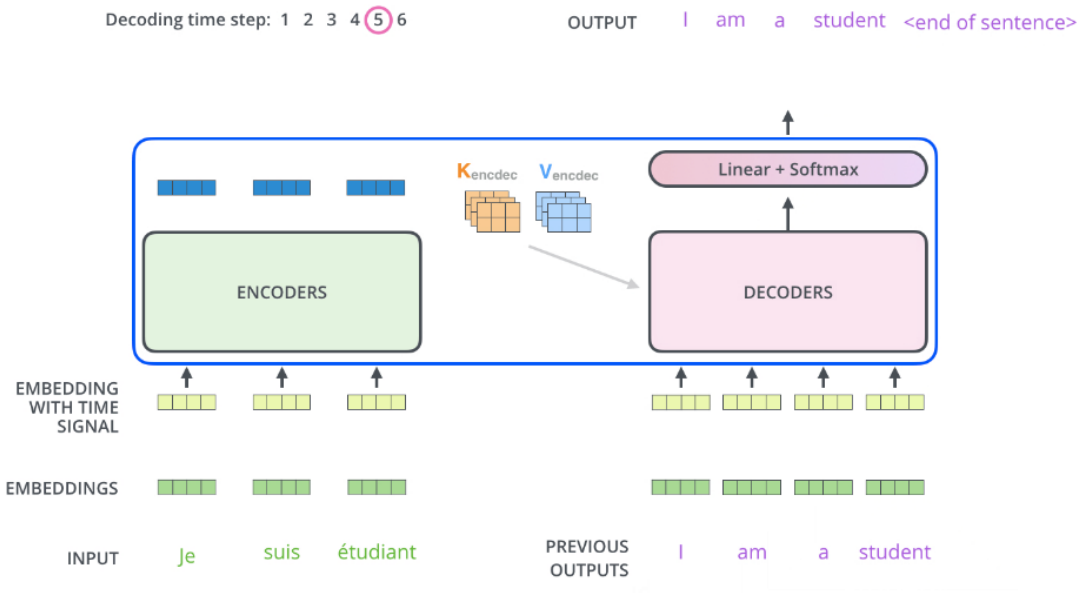


Before feeding into the encoder, we introduce positional encoding which determines the position of each word, or distance between different words. In addition, within each encoder, the residual block and layer normalization is applied.



The decoder is very similar to encoder, except that the decoder has an encoder-decoder attention which only attends to parts of the top layer of encoder representation.



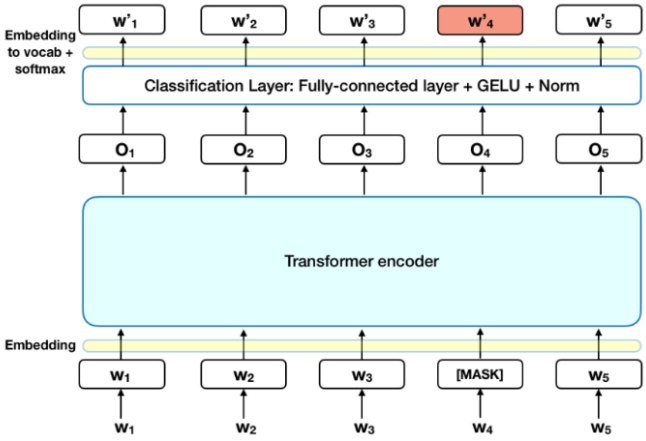


Bidirectional Encoder Representations from Transformers(BERT):

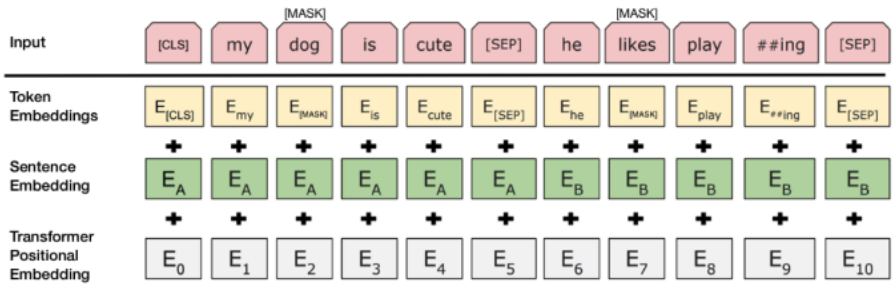
The encoder part of Transformer as the token representations, then use them to train a language model.

The LM contains two tasks: masked word prediction and next sentence prediction. Both two tasks are jointly trained and optimized.

In masked word prediction task, we randomly replace 15% of words in training corpus with mask token, then feed the entire words of a sentence into the Transformer, and predict the those masked word.



While in terms of next sentence prediction task, firstly we embed a pair of sentence in three different ways, token embedding(WordPiece), positional embedding(similar to Transformer) and sentence embedding, which is a 2×1 vector to specify what word belongs to which sentence. After feeding into Transformer layer to get the context vectors, the vector at the position of special token [CLS] is used for prediction.



2. Neural Networks

motivation: To represent a complex mapping function from input(bunch of word vectors) to output(can be a class label, a sentence, etc.). Each unit can be regarded a single binary classifier.

linear classifier: Input: embedded words. Parameters: weight matrix W with bias term b. Output: probability distribution $\hat{y}$ after applying softmax. Layer: input layer, hidden layer, output layer.

forward propagation:

① linear transformation from input layer to hidden layer:

$$h = W^T x + b$$

② activation function:

$$a = f(h)$$

where $f(\cdot)$ is a activation function(ReLU, tanh, sigmoid, etc.)

③ softmax unit:

$$\hat{y} = \text{softmax}(a) = -\log\left(\frac{e^{s_y}}{\sum_c e^{s_c}}\right)$$

where s_c is the score value of the c-th unit, s_y is the score value of the correct label, the term $\frac{e^{s_y}}{\sum_c e^{s_c}}$ represents the normalized probabilities of each class.

backpropagation: the error is propagated from the last layer back to each parameter.

$$\frac{dL}{dW} = \frac{dL}{da} \cdot \frac{da}{dh} \cdot \frac{dh}{dW}$$

$$\frac{dL}{db} = \frac{dL}{da} \cdot \frac{da}{dh} \cdot \frac{dh}{db}$$

If we have multiple linear layer, the term $\frac{dL}{da} \cdot \frac{da}{dh}$ for each layer's error is denoted to δ. To compute the error gradient in layer δ^h, we have this equation: $\delta^h = \delta^{h+1} \cdot \frac{\partial \delta^{h+1}}{\partial a^{(h)}} \cdot f'(h)$. In other words, the downstream gradient is equal to the product of upstream gradient and local gradient.

For different nodes in computational graph, we have different intuitions:

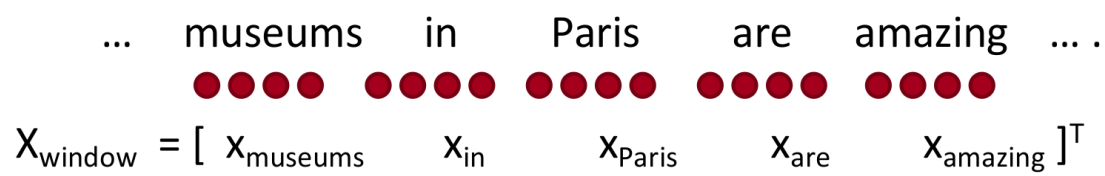
+ distributes the upstream gradient, e.g. $z = x + 2y$. The gradients with respect to x and $2y$ is equal.

max routes the upstream gradient, e.g. $z = \max(x, 2y)$. If $x > 2y$, the upstream gradient only propagates x instead of $2y$.

× switches the upstream gradient, e.g. $z = xy$. the gradient with respect x to is y, while with respect y to is x.

application: name entity recognition, classify a center word w in the context of this sentence with the window size l.

e.g.



Language Model: Assign probability given a piece of text. In practice, given a piece of text with t words $x^{(1)}, x^{(2)}, \cdots, x^{(t-1)}, x^{(t)}$, the model tries to predict the next word $x^{(t+1)}$ according to the probability distribution.

$$P(x^{(t+1)}|x^{(t)}, x^{(t-1)}, \cdots, x^{(2)}, x^{(1)})$$

If we only consider the first n-1 words(i.e. n-grams), the equation is defined as:

$$P(x^{(t+1)}|x^{(t)}, x^{(t-1)}, \cdots, x^{(t-n+3)}, x^{(t-n+2)})$$

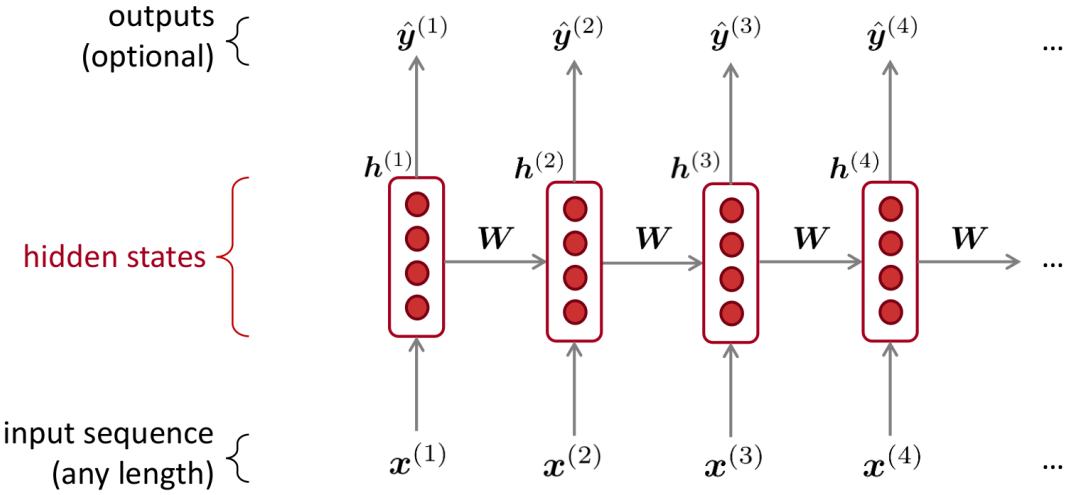
To calculate this equation, we use count-based algorithm to get n-gram probability.

$$P(x^{(t+1)}|x^{(t)}, x^{(t-1)}, \cdots, x^{(t-n+3)}, x^{(t-n+2)})$$
$$= \frac{count(x^{(t+1)}, x^{(t)}, \cdots, x^{(t-n+3)}, x^{(t-n+2)})}{count(x^{(t)}, x^{(t-1)}, \cdots, x^{(t-n+3)}, x^{(t-n+2)})}$$

Count-based model would have sparsity problems. One is n-gram never occurs in our corpus, thus we assign a small probability for smoothing. Another is (n-1)-gram never occurs in the corpus. In this case, we can backoff the (n-1)-gram to smaller gram.

Recurrent Neural Network: language model has drawbacks that cannot handle any length text. In other words, if the predicting word needs a very long dependency, fixed window size is not capable to do the task. Thus, RNN is introduced to process any length input.

architecture:



The input is processed in sequence: $x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}, \cdots$. Each hidden state $h^{(t)}$ is calculated by the linearly transformed version of $x^{(t)}$ by a matrix W_{hx} plus the previous hidden state $h^{(t-1)}$ after transformation of a weight matrix W_{hh}. Finally, we activate $h^{(t)}$ by a non-linearity.

$$h^{(t)} = \sigma(W_{hx}x^{(t)} + W_{hh}h^{(t-1)} + b)$$

The calculation of the output $\hat{y}^{(t)}$ of each time-step.

$$\hat{y}^{(t)} = \text{softmax}(W_{yh}h^{(t)})$$

error evaluation:

For each time step t, we use cross-entropy loss for evaluation:

$$J^{(t)}(\theta) = -\sum_{w \in V} y_w^{(t)} \log \hat{y}_w^{(t)} = -\log \hat{y}_{x_{t+1}}^{(t)}$$

The overall loss:

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{y}_{x_{t+1}}^{(t)}$$

perplexity:

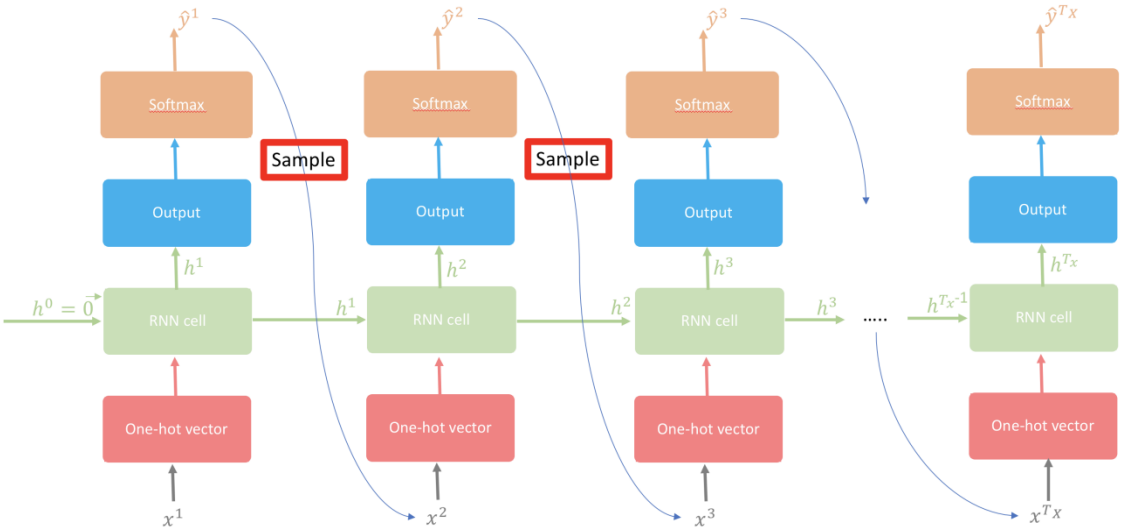
$$\text{perplexity} = \exp(J(\theta))$$

backpropagation: The gradient with respect to each weight matrix and bias term is cumulative.

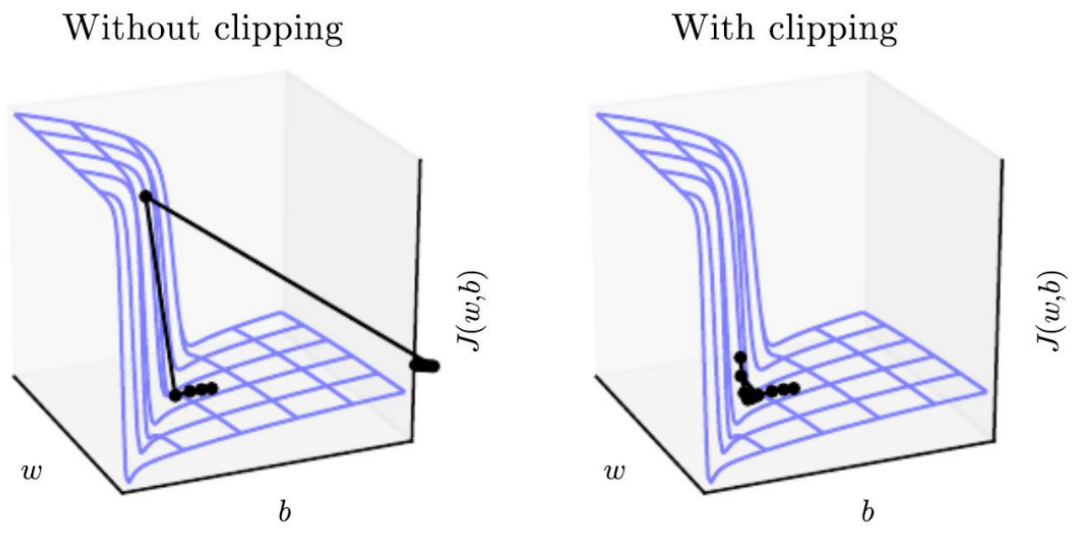
Suppose our objective is to compute t-th error $J^{(t)}(\theta)$'s partial derivative with respect to W.

$$\frac{\partial J^{(t)}(\theta)}{\partial W} = \sum_{i=1}^{t} \frac{\partial J^{(t)}(\theta)}{\partial W}\Big|_{(i)}$$
$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}(\theta)}{\partial h^{(i)}} \frac{\partial h^{(i)}}{\partial W}$$
$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}(\theta)}{\partial h^{(i)}} x^{(i)}$$
$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}(\theta)}{\partial h^{(t)}} \prod_{j=i}^{t-1} \left(\frac{\partial h^{(j+1)}}{\partial h^{(j)}} \right) x^{(i)}$$

sampling: When testing our RNN language model, for each output, we randomly sample from the probability distribution to get $\hat{y}^{(t)}$. Then the output is sent to the next step as the input, which is $x^{(t+1)}$.



gradient explosion and gradient vanishing: As the error backpropagates to the previous step, if the cumulative term $\prod_{j=i}^{t-1} \left(\frac{\partial h^{(j+1)}}{\partial h^{(j)}} \right)$ is larger or smaller than 1, the gradient will explode or vanish. To address gradient explosion, we can clip the gradient which is larger than a threshold to a constant value. In terms of gradient vanishing, an alternative RNN variant model is introduced.



Gated Recurrent Unit(GRU):

update gate:

$$u^{(t)} = \sigma(W_u h^{(t-1)} + U_u x^{(t)} + b_u)$$

reset gate:

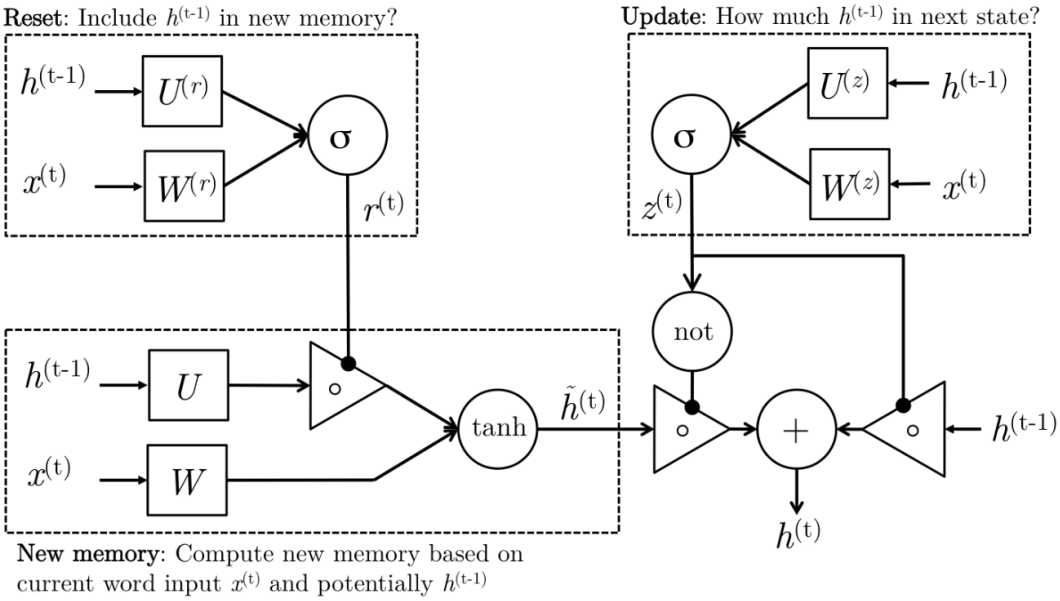
$$r^{(t)} = \sigma(W_r h^{(t-1)} + U_r x^{(t)} + b_r)$$

candidate hidden state:

$$\tilde{h}^{(t)} = \tanh(W_h(r^{(t)} \circ h^{(t-1)}) + U_h x^{(t)} + b_h)$$

next hidden state:

$$h^{(t)} = (1 - u^{(t)}) \circ h^{(t-1)} + u^{(t)} \circ \tilde{h}^{(t)}$$



Long Short-Term Memory(LSTM):

forget gate:

$$f^{(t)} = \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f)$$

input gate:

$$i^{(t)} = \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i)$$

output gate:

$$o^{(t)} = \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o)$$

candidate cell state:

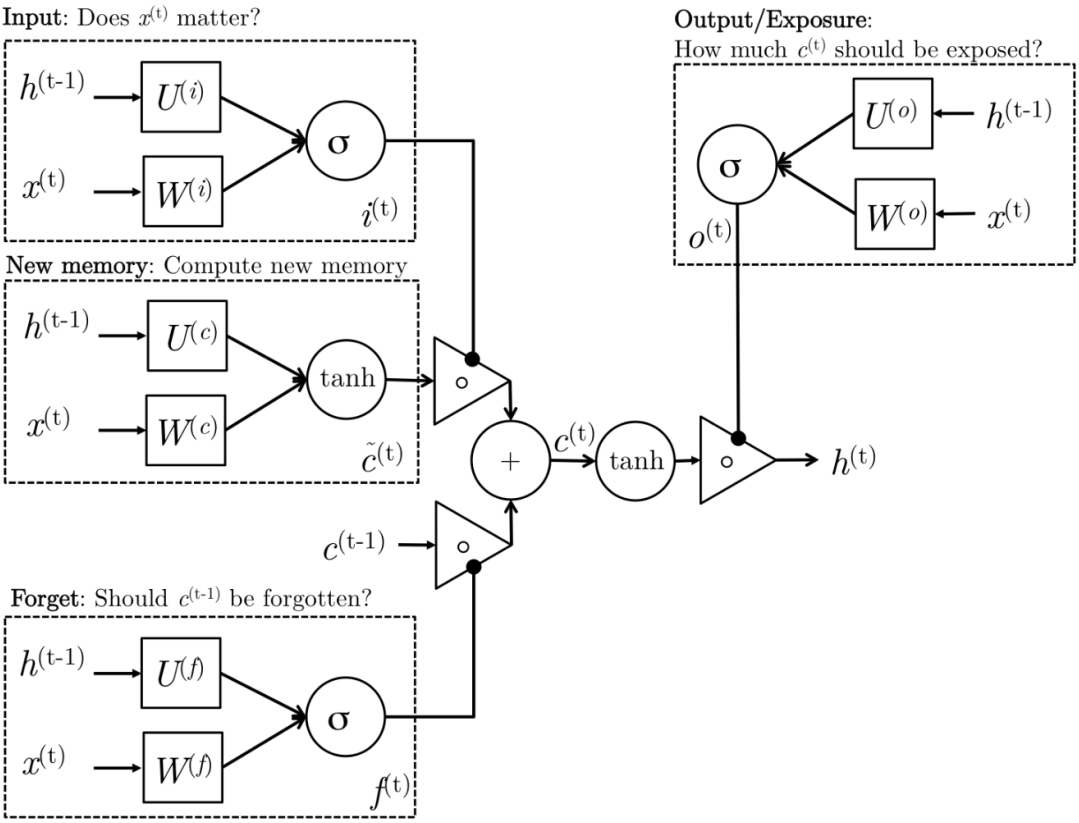
$$\tilde{c}^{(t)} = \tanh(W_c h^{(t-1)} + U_c x^{(t)} + b_c)$$

next cell state:

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

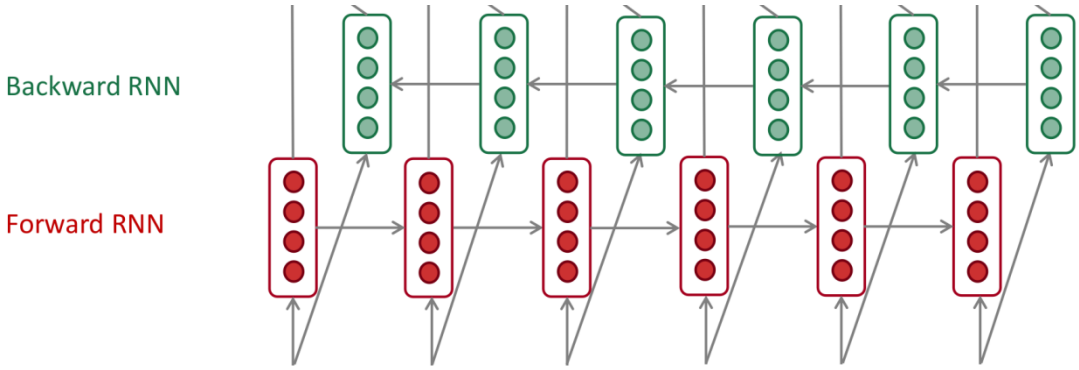
next hidden state:

$$h^{(t)} = o^{(t)} \circ \tanh c^{(t)}$$

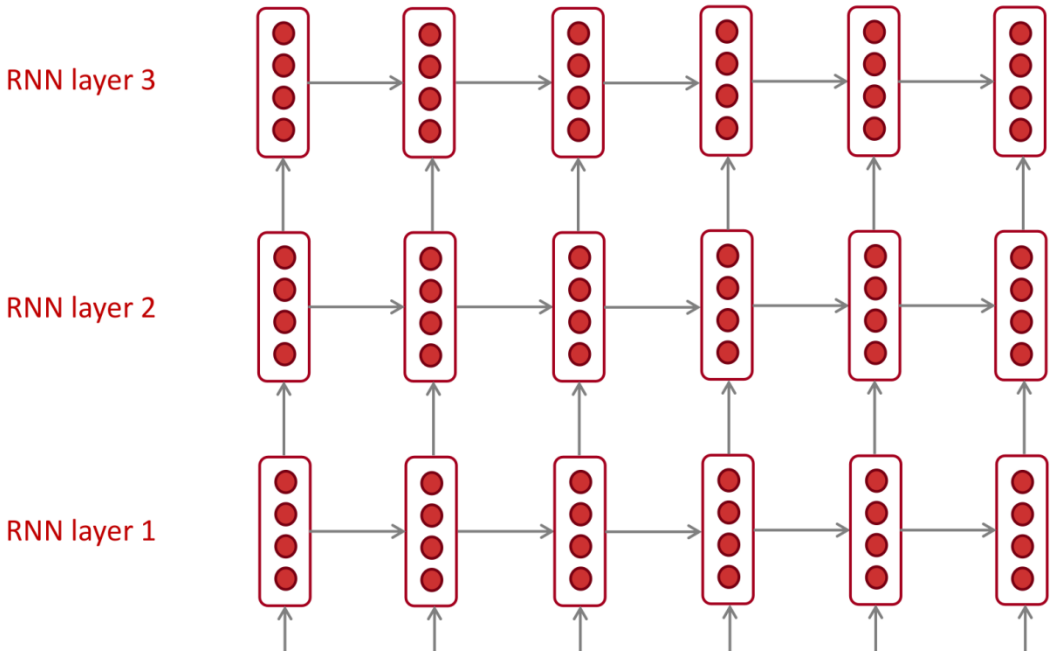


Bidirectional Recurrent Neural Network(Bi-RNN): In a certain context, the word that the model will predict is not only dependent on the previous text, but also the former text. So we represent hidden state $h^{(t)}$of Bi-RNN to the concatenation of forward hidden state $\overrightarrow{h}^{(t)}$ and backward hidden state $\overleftarrow{h}^{(t)}$.

$$h^{(t)} = [\overrightarrow{h}^{(t)}; \overleftarrow{h}^{(t)}]$$



Multiple RNN: In some cases, what we want is to represent a more complex representations or features of a sentence or a piece of text, rather than a single-layer RNN, we stack 2 or 3 of them to complicate our model.



application: sentiment classification, natural language generation

Convolutional Neural Network(CNN): Similar to vision, CNN in NLP also uses a filter to extract features in a fixed size(kernel size), but there are only one dimension to be convolved.

padding: pad 0s across channel at the beginning and at the end of a sentence to make sure the length of the sentence is the same after convolution.

Ø	0.0	0.0	0.0	0.0
tentative	0.2	0.1	−0.3	0.4
deal	0.5	0.2	−0.3	−0.1
reached	−0.1	−0.3	−0.2	0.4
to	0.3	−0.3	0.1	0.1
keep	0.2	−0.3	0.4	0.2
government	0.1	0.2	−0.1	−0.1
open	−0.4	−0.4	0.2	0.3
Ø	0.0	0.0	0.0	0.0

Ø,t,d	−0.6	0.2	1.4
t,d,r	−1.0	1.6	−1.0
d,r,t	−0.5	−0.1	0.8
r,t,k	−3.6	0.3	0.3
t,k,g	−0.2	0.1	1.2
k,g,o	0.3	0.6	0.9
g,o,Ø	−0.5	−0.9	0.1

Apply 3 **filters** of size 3

3	1	2	−3
−1	2	1	−3
1	1	−1	1

1	0	0	1
1	0	−1	−1
0	1	0	1

1	−1	2	−1
1	0	−1	3
0	2	2	1

Could also use (zero)
padding = 2
Also called “wide convolution”

pooling: extract the most important features in the convolved sentence representation.

max pooling: select the maximum value in a receptive field. One variant is k-max pooling, which selects the first k maximum value.

Ø	0.0	0.0	0.0	0.0
tentative	0.2	0.1	−0.3	0.4
deal	0.5	0.2	−0.3	−0.1
reached	−0.1	−0.3	−0.2	0.4
to	0.3	−0.3	0.1	0.1
keep	0.2	−0.3	0.4	0.2
government	0.1	0.2	−0.1	−0.1
open	−0.4	−0.4	0.2	0.3
Ø	0.0	0.0	0.0	0.0

Ø,t,d	−0.6	0.2	1.4
t,d,r	−1.0	1.6	−1.0
d,r,t	−0.5	−0.1	0.8
r,t,k	−3.6	0.3	0.3
t,k,g	−0.2	0.1	1.2
k,g,o	0.3	0.6	0.9
g,o,Ø	−0.5	−0.9	0.1

2-max p	−0.2	1.6	1.4
	0.3	0.6	1.2

Apply 3 **filters** of size 3

	3	1	2	−3
	−1	2	1	−3
18	1	1	−1	1

1	0	0	1
1	0	−1	−1
0	1	0	1

1	−1	2	−1
1	0	−1	3
0	2	2	1

average pooling: compute the average value within a certain receptive field.

In CNN, the layer is often stacked into a huge number, resulting in high training error. The solution is applying residual block and batch normalization. To efficiently train our model, an 1×1 convolution is used to reduce dimensionality.

Quasi-RNN: RNN is a sequential model which is hard for parallelism, but CNN is, so we design a pseudo-recurrence model which performs convolutional operation to the whole sequence.

- Convolutions for parallelism across time:

$$\mathbf{z}_t = \tanh(\mathbf{W}_z^1 \mathbf{x}_{t-1} + \mathbf{W}_z^2 \mathbf{x}_t)$$
$$\mathbf{f}_t = \sigma(\mathbf{W}_f^1 \mathbf{x}_{t-1} + \mathbf{W}_f^2 \mathbf{x}_t)$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_o^1 \mathbf{x}_{t-1} + \mathbf{W}_o^2 \mathbf{x}_t).$$

→

$$\mathbf{Z} = \tanh(\mathbf{W}_z * \mathbf{X})$$
$$\mathbf{F} = \sigma(\mathbf{W}_f * \mathbf{X})$$
$$\mathbf{O} = \sigma(\mathbf{W}_o * \mathbf{X}),$$

Convolutions compute candidate, forget & output gates

- Element-wise gated pseudo-recurrence for parallelism across channels is **done in pooling layer**: $\mathbf{h}_t = \mathbf{f}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}_t) \odot \mathbf{z}_t,$

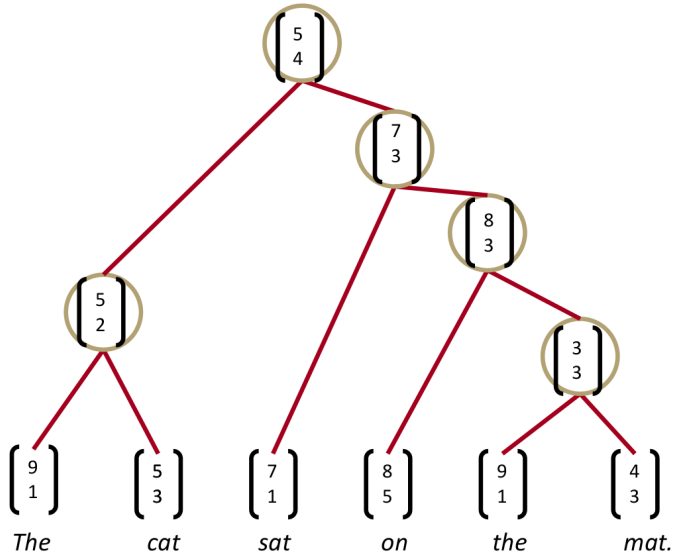
Tree RNN: The motivation of Tree RNN is a sentence or a phrase has perfect nested hierarchy and recursive structure. So instead of represent a single word, our goal is represent phrases in the word space where those with the similar meaning are close each other. e.g.:

(i) single layer RNN: For each recursion, the model computes each pair of words or phrases by a simple fully connected neural network to get the corresponding parent node(phrase) representation and a score to evaluate how plausible the new node would be.

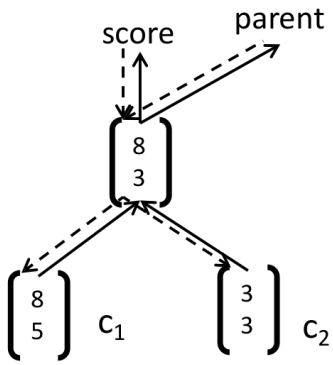
$$p = \tanh(W\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} + b)$$

$$s = U^T p$$

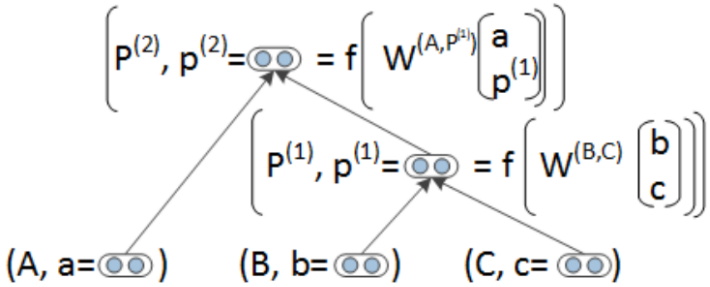
By doing so in a several iterations, we can construct a tree-like structure where the root node is the representation of a whole sentence.



When backpropagation, the error gradient flows from parent node plus its own score error down to child nodes.



(ii) Syntactically Untied RNN(SU-RNN): The naive RNN uses a same matrix W to combine all words together, thus it is less powerful than use multiple matrices to do the same task. SU-RNN allows us to use different matrices based on the syntactic categories of the inputs. e.g. If two words or phrases belong to name phrase, then corresponding matrix W_N will concatenate two words(or phrases) together.

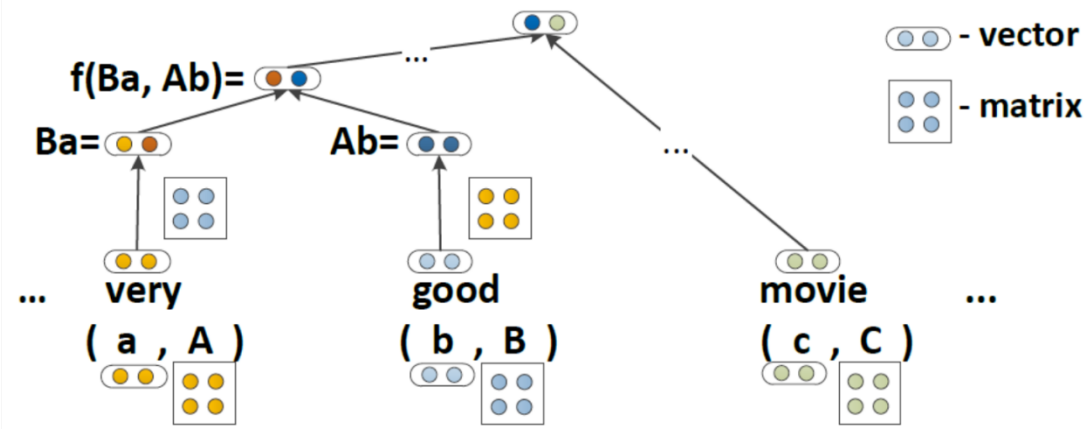


(iii) Matrix-Vector RNN(MV-RNN): As we can see in the model previously, two words(phrases) do not interact with each other. In other words, the upper and lower part of matrix W only has the impact on c_1 and c_2 respectively. For example, if we want to know the extent of a word to scale another word("very" to scale "good"), the naive matrix-vector multiplication is not enough to represent that. Hence, we have a new model MV-RNN in which each node has not only a word vector but a matrix whose function is modify other words.

For two nodes, suppose we have two word vectors a and b, and two matrices A and B, the new node's vector and matrix representation is:

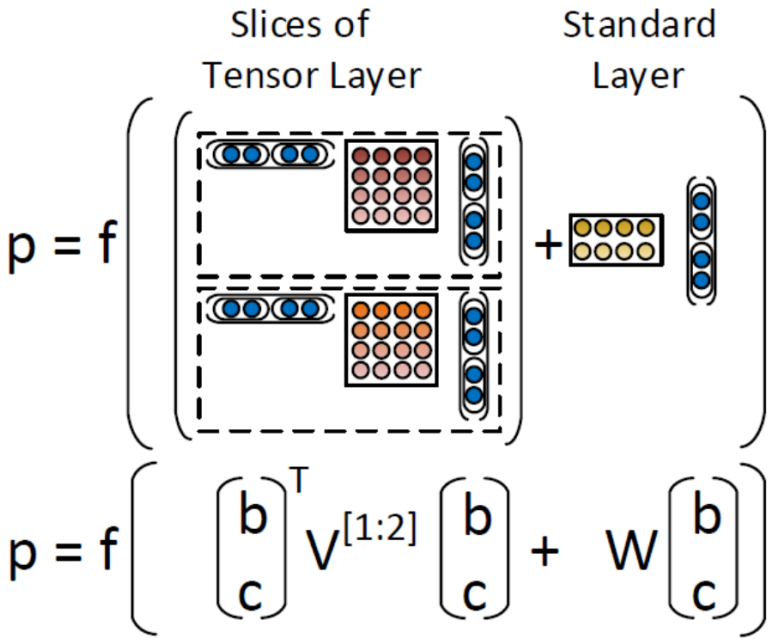
$$p = \tanh(W_V \begin{bmatrix} Ba \\ Ab \end{bmatrix})$$

$$P = W_M \begin{bmatrix} A \\ B \end{bmatrix}$$



However, MV-RNN needs additional matrix for each node, which leads to more space and computation.

(iv) Recursive Neural Tensor Network(RNTN): This model uses a 3-D tensor V to allow two word or phrase vectors to interact multiplicatively. It not only has fewer parameters than MV-RNN, but is a powerful model that can capture the tone of a sentence if negating one word from positive(negative) to negative(positive).



3. Parsing

motivation: want to know the structure of a sentence by partitioning into different parts(name phrase, verb phrase, prepositional phrase, etc.), each of which is again composed by other parts.

dependency parsing: construct a dependency list, in which part of words are dependent on other words, i.e. an adjective is the modifier of a noun or adverb.

transition-based parsing: the process of parsing is a transition from one state to another state. Initially, the parser has a stack σ which contains ROOT symbol, a buffer β with the input sentence, and dependency arcs A. There are three transitions available.

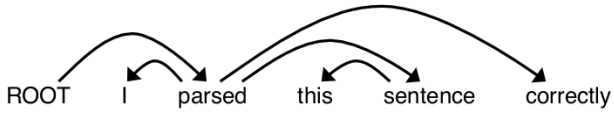
SHIFT: shift one word from the top of buffer to the top of stack.

LEFT-ARC: add a dependency arc($w_i \rightarrow w_j$), where w_i is the top of the stack and w_j is the second top of the stack, then remove word w_j.

RIGHT-ARC: add a dependency arc($w_j \rightarrow w_i$), where w_i is the top of the stack and w_j is the second top of the stack, then remove word w_i.

Iterate the above three transitions until the buffer is empty and there are only one word(ROOT) is the stack.

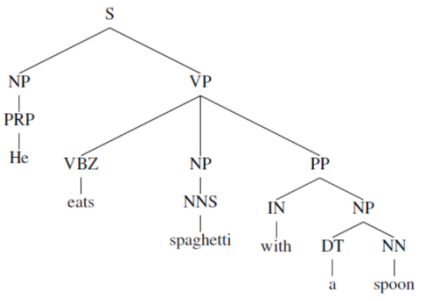
Suppose we have a dependency parsing tree:

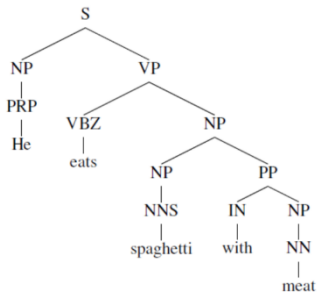


The process of parsing is:

Stack	Buffer	New Dependency	Transition
[ROOT]	[I, parsed, this, sentence, correctly]		Initial Configuration
[ROOT, I]	[parsed, this, sentence, correctly]		SHIFT
[ROOT, I, parsed]	[this, sentence, correctly]		SHIFT
[ROOT, parsed]	[this, sentence, correctly]	parsed→I	LEFT-ARC
[ROOT, parsed, this]	[sentence, correctly]		SHIFT
[ROOT, parsed, this, sentence]	[correctly]		SHIFT
ROOT, parsed, sentence]	[correctly]	sentence→this	LEFT-ARC
[ROOT, parsed]	[correctly]	parsed→sentence	RIGHT-ARC
[ROOT, parsed, correctly]	[]		SHIFT
[ROOT, parsed]	[]	parsed→correctly	RIGHT-ARC
[ROOT]	[]	ROOT→parsed	RIGHT-ARC

constituency parsing: construct parsing tree based on the compositionality of a sentence, which can be a name phrase, verb phrase, etc.



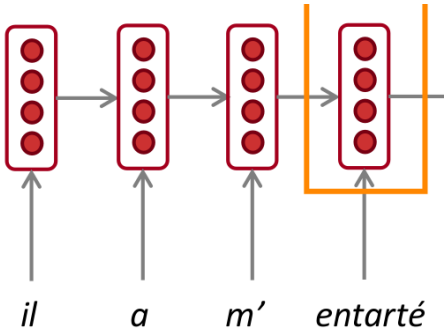


3. Neural Translation

Seq2Seq model: two components: encoder, which encodes a sequence of source language to a feature vector by an

RNN(or LSTM); decoder: which uses a language model to generate target language.

encoder: Produce a context vector which feeds a bunch of embedded word vector to an RNN and the last hidden state(concatenation of first and last hidden state, if Bi-RNN).



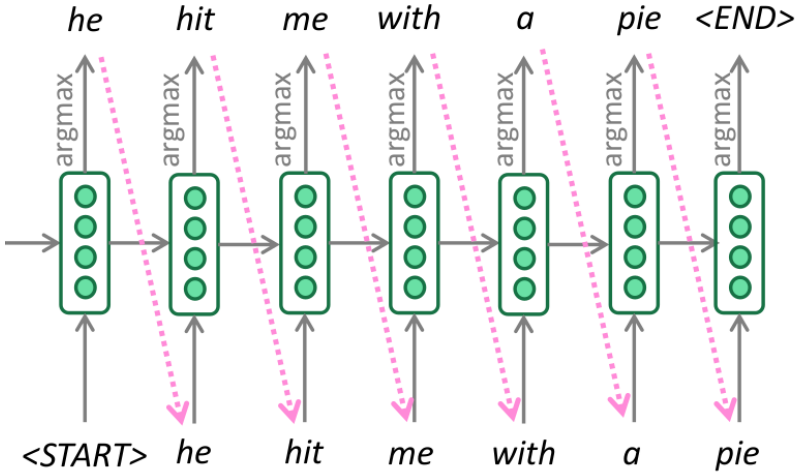
decoder: The feature vector produced by encoder is fed into the decoder as the initial hidden state, then decoder generates target language words step by step. The decoding mechanism can be different based on situations.

(i) greedy decoding: For each time step, choose the word with the largest probability, but it is harmful if decoding undo decisions(i.e. one word mistakenly generates).

(ii) exhaustive search: we try to compute all possibilities of target sequences and choose the best one. This method always finds the best target language, but the complexity($O(V^T)$) is too expensive.

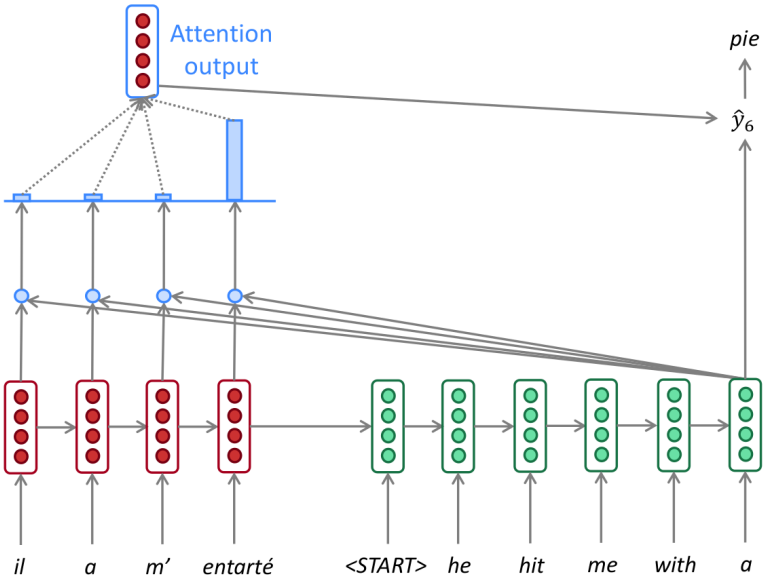
(iii) beam search: For each step, choose the most likely k partial translation sequences and iterate them over and over again until in the last step, we generate the most likely one.

(iv) sampling: For each time we choose a word as the target word, rather than picking the most likely word, we sample from the probability distribution, which allows us to get more possibilities of a target sequence.



attention: In the analog of human translation, which only consider part of information of a long context and translate them, machine translation also takes this approach that pays attention to certain words or phrases and generate translation.

Specifically, whenever a decoder computes the hidden state of the current time step h_t^{dec}, take this hidden state vector to get attention score $\alpha_{t,i}$ by doing dot product with each encoder hidden state h_i^{enc}. After that, we compute the weighted encoder hidden state $a_t = \sum_i^m \alpha_{t,i} h_i^{enc}$. Finally, is concatenated with h_t^{dec} to predict the next target word.



character-level representation: represent a word vector in character level, i.e., each word is represented by several character embeddings.

byte pair encoding: A word segmentation algorithm which compresses the total embedding storage. Concretely, we start with a vocabulary of characters, then compute the frequency of each n-gram character pairs, choose the most frequent one to become the new pair.

Dictionary

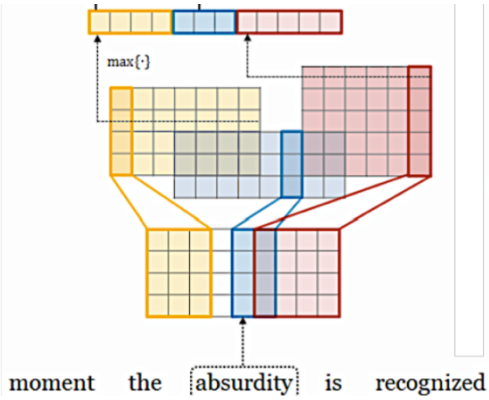
5 **lo** w
2 **lo** w e r
6 n e w est
3 w i d est

Vocabulary

l, o, w, e, r, n, w, s, t, i, d, es, est, **lo**

word-piece/sentence-piece model: Rather than counting the number of n-grams, this model uses greedy approximation to maximizing the language model likelihood to choose that pieces. In other words, the algorithm splits a word into different pieces, choose one that can maximally reduce overall perplexity as the encoding.

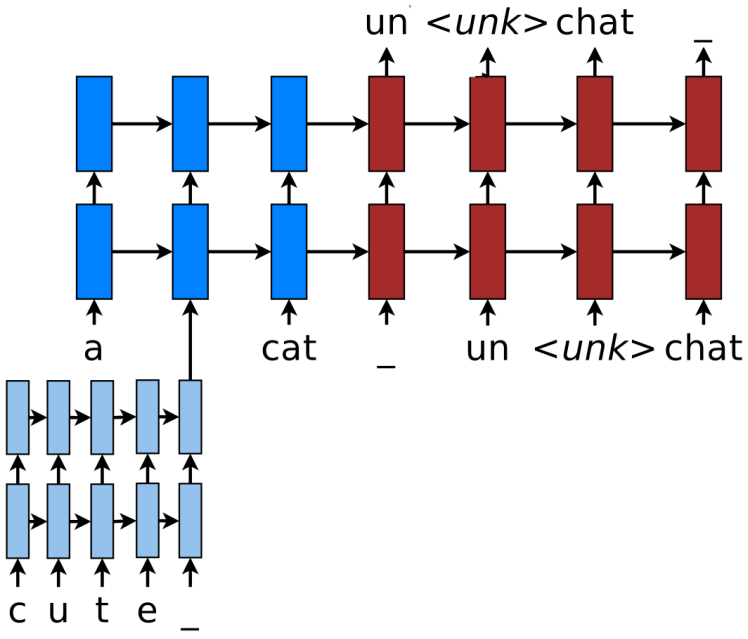
CNN model: convolution over characters to generate word embeddings.



FastText model: represent a word as character n-grams augmented with boundary symbol and a whole word embedding, and then sum them together.

where = <wh, whe, her, ere, re>, <where>

hybrid NMT: combine word-level and character-level model together. In most cases, the translation is based on word level, but in some case where there is a unknown word, we choose character-level embedding to represent that out-of-vocab word. This technique can be done in both encoder and decoder.



BLEU score: an evaluation metric for machine translation.

For each n-gram, we calculate modified precision p_n. Suppose there is a candidate translation $\mathbf{c}$ and k reference translations $\mathbf{r}_1, \mathbf{r}_2 \cdots, \mathbf{r}_k$.

$$p_n = \frac{\sum_{n-garm \in \mathbf{c}} \min\left(\max_{i=1,2,\cdots k} \text{Count}_{\mathbf{r}_i}(n-\text{gram}), \text{Count}_{\mathbf{c}}(n-\text{gram})\right)}{\sum_{n-garm \in \mathbf{c}} \text{Count}_{\mathbf{c}}(n-\text{gram})}$$

where $\sum_{n-garm \in \mathbf{c}} \text{Count}_{\mathbf{c}}(n-\text{gram})$ is the number of n-grams in candidate translation $\mathbf{c}$, $\max_{i=1,2,\cdots k} \text{Count}_{\mathbf{r}_i}(n-\text{gram})$ is the maximum number of n-grams occurring in reference translations $\mathbf{r}_1, \mathbf{r}_2 \cdots, \mathbf{r}_k$.

For those short translations, BLEU has a mechanism called brevity penalty BP.

$$BP = \begin{cases} 1 & \text{if } c \geq r^* \\ \exp(1 - \frac{r^*}{c}) & \text{otherwise} \end{cases}$$

In the above equation, c is the length of candidate translation and r^* is the length of reference translation that is closest to length c(choose shorter one if it is a tie).

Finally, the BLEU score is the weighted sum of precision of all n-gram.

$$\text{BLEU} = BP \times \exp(\sum_{i=1}^{n} \lambda_i \log p_i)$$

where is λ_i the weight factor for each n-gram.

4. Question Answering

what they do: given a piece of text and several questions related, ask the model to answer these questions by extracting the most relevant information.

Stanford Attentive Reader:

Encoding layer: The model use a single-layer bidirectional GRU that takes each word in the passage as input and outputs the concatenated version of each hidden state $\tilde{\mathbf{p}}_i = [\vec{\mathbf{h}}_i, \overleftarrow{\mathbf{h}}_i]$. In terms of question words, we only output the

last hidden state $\mathbf{q}$ in both directions.

Attention layer: The next step is use attention mechanism to compute the attention score α_i for every passage word representation $\tilde{\mathbf{p}}_i$ to question representation $\mathbf{q}$.

$$\alpha_i = \text{softmax}(\mathbf{q}^T \mathbf{W}_s \tilde{\mathbf{p}}_i)$$

Then calculate the weighted sum of passage words embedding $\mathbf{o}$.

$$\mathbf{o} = \sum_i \alpha_i \tilde{\mathbf{p}}_i$$

Prediction layer: the final softmax layer is to compute the prediction for each word in the passage.

BiDAF:

Embedding layer: the incoming context and query words are firstly tokenized, then are subjected to embedding process on three levels of granularity: word-level, character-level and context-level.

Attention layer:

Similarity Matrix: Let c_i and q_j be contextual embedding of i-th context word and j-th query word. The first step of attention mechanism is build a similarity matrix S, in which each element S_i is computed by:

$$S_{ij} = w_{sim}^T [c_i; q_j; c_i \circ q_j]$$

Context-to-Query Attention: For each context word, we compute the attended query word representation to reflect how a context word is relevant to query words.

$$\alpha^i = \text{softmax}(S_{i,:})$$

$$a_i = \sum_{j=1}^{M} \alpha_j^i q_j$$

Query-to-Context Attention: Take the maximum value of each row that concludes the most relevant word in query words of each context word.

$$m_i = \max_j S_{ij}$$

Then apply softmax to get the probability distribution β.

$$\beta = \text{softmax}(m)$$

Finally, use β to take a weighted sum of context word vector c'.

$$c' = \sum_{i=1}^{N} \beta_i c_i$$

Merge: merge the above vector together to get b_i:

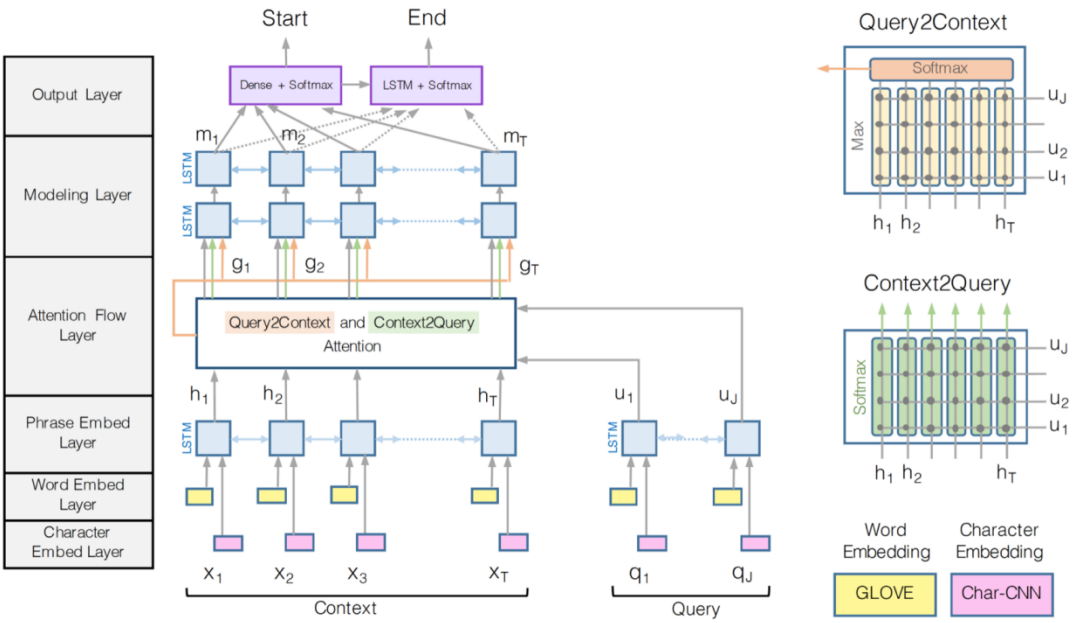
$$b_i = [b_i; c_i; c_i \circ a_i; c_i \circ c_i']$$

Modeling Layer: use a two-layer bidirectional LSTM to obtain a representation for every context word.

$$h_i^{(1)} = \text{LSTM}^{(1)}(b_i)$$
$$h_i^{(2)} = \text{LSTM}^{(2)}(h_i^{(1)})$$

Output Layer: concatenate b_i with $h_i^{(1)}$ and b_i with $h_i^{(2)}$, then compute softmax scores for each context word to predict how likely this word is the start or end of the answer.

$$p_i^{start} = \text{softmax}(w_{start}^T [b_i; h_i^{(1)}])$$
$$p_i^{end} = \text{softmax}(w_{end}^T [b_i; h_i^{(2)}])$$



5. Coreference Resolution

motivation: To fully understand human languages, one important point is understand what a name phrase or pronoun refers to.

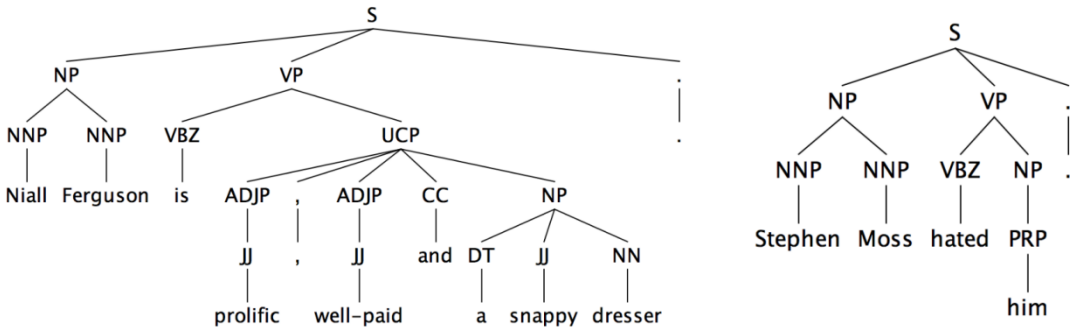
e.g.

> Barack Obama nominated Hillary Rodham Clinton as his secretary of state on Monday. He chose her because she had foreign affairs experience as a former First Lady.

step: mention detection, which is the easy part of coreference resolution, mainly detects the span of text that refers to some entity; cluster these mentions is the hard step for coreference resolution, which clusters referring mention and referent together.

mention detection: Since those mentions are composed of pronoun, name entity or name phrase, some models like part-of-speech tagger, name entity recognizer and parser can be used handle that, but it is more convenient to use a simple classifier to classify each word.

rule-based model(Hobbs Algorithm): a baseline algorithm that starting from a name phrase, the algorithm uses path traversal based on dependency parser and syntactic rules to resolute coreference.



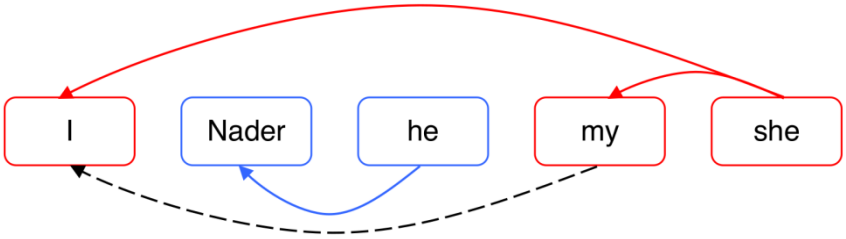
mention pair: train a binary classifier that assigns every pair of candidate mentions a probability being coreferent, where the probability of positive examples are close to 1 and negative examples close to 0. Thus the objective

function we will optimize is:

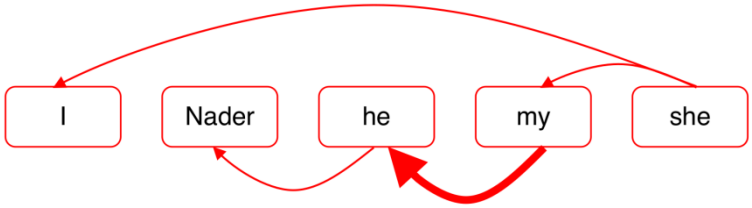
$$J(\theta) = -\sum_{i=2}^{N}\sum_{j=1}^{i-1} y_{ij} \log p(m_i, m_j)$$

where N is the number of mentions in the document, $p(m_i, m_j)$ is the probability of mention pair (m_i, m_j) being coreferent and y_{ij} is equal to 1 if it is a positive example and 0 otherwise.

At test time, we only choose pairs as coreference links that the probability is above a threshold we set before. For those mentions are not linked directly, they are coreferent due to transitivity.



However, mention pair has some drawbacks. When coreference is incorrectly linked, since the transitive closure, it turns out the mention clustering will be totally wrong. Besides, when we have a very long document, it is hard to train a classifier to predict two mentions being coreferent.



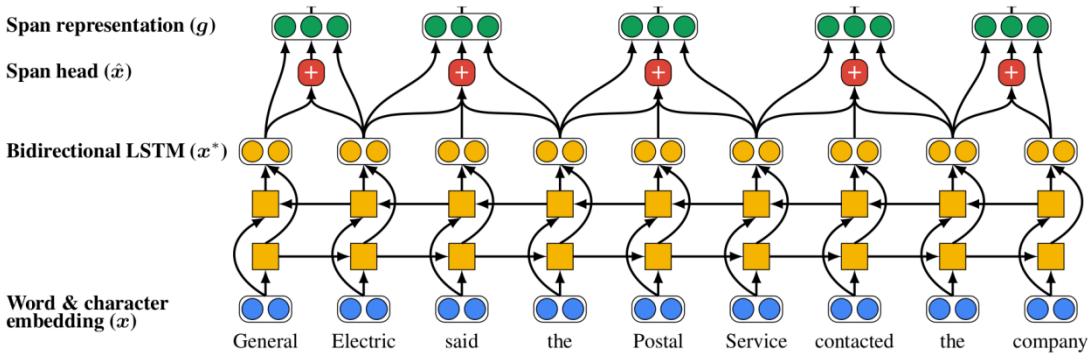
mention ranking: rather than linking mentions between two positive examples, mention ranking model assign each mention with highest score among all candidate antecedents. Thus we have loss function:

$$J(\theta) = \sum_{i=2}^{N} -\log\left(\sum_{j=1}^{i-1} \mathbf{1}(y_{ij} = 1)p(m_i, m_j)\right)$$

To compute the probability, three different approaches can be used: feature-based model, neural-based model and end-to-end model(LSTM, attention, etc.).

end-to-end model:

For each word in the document, embed them by a embedding matrix. Then feed these vectors into a Bi-LSTM RNN to get the hidden state vector x^*. To represent a coreference word or phrase, the model outputs several span representations g for coreference resolution. Concretely the span representation is: $g_i = [x^*_{start(i)}, x^*_{end(i)}, \hat{x}_i, \phi(i)]$, where $x^*_{start(i)}$ and $x^*_{end(i)}$ are hidden state of start and end of the span, $\hat{x}_i$ is weighted sum of span representation and $\phi(i)$ is additional feature vector.

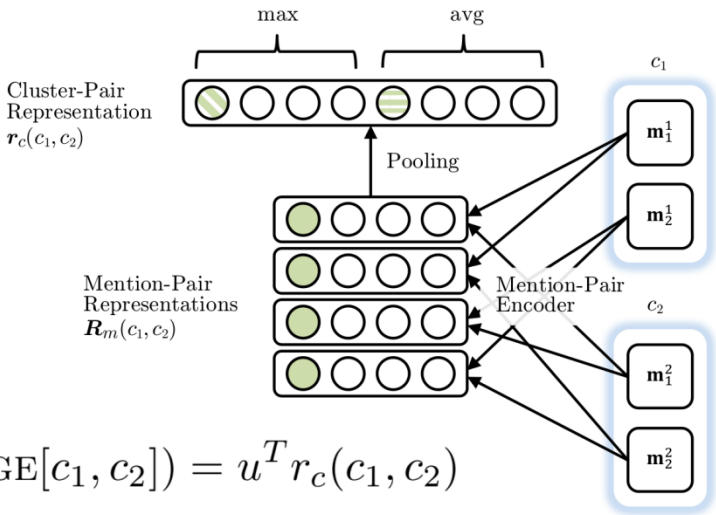


Lastly, the coreferent score $s(i,j)$ of mention and is denoted to:

$$s(i,j) = s_m(i) + s_m(j) + s_a(i,j)$$

where $s_m(i) = w_m \cdot \text{FFNN}_m(g_i)$ which indicates the probability score of span i being mention and $s_a(i,j) = w_m \cdot \text{FFNN}_m([g_i, g_j, g_i \circ g_j, \phi(i,j)])$ which implies whether mention i and mention j is coreferent.

cluster-based model: use clustering algorithm to cluster mentions being coreferent.

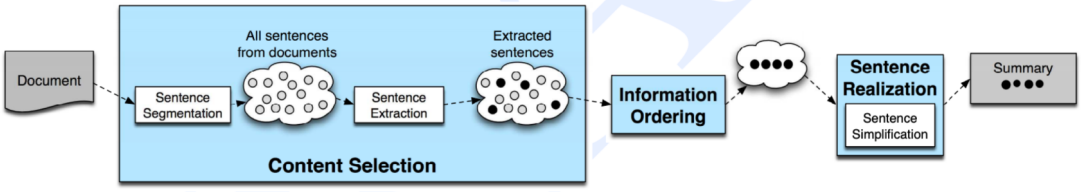


6. Natural Language Generation

motivation: Want to generate new text that are not extracted from document or corpus. Its relevant fields are diverse including machine translation, summarization, dialogue, story writing, question answering, image captioning, etc.

summarization: In NLG, summarization is not just extract original text from the text. Instead, it paraphrases these text to human-understandable summarization.

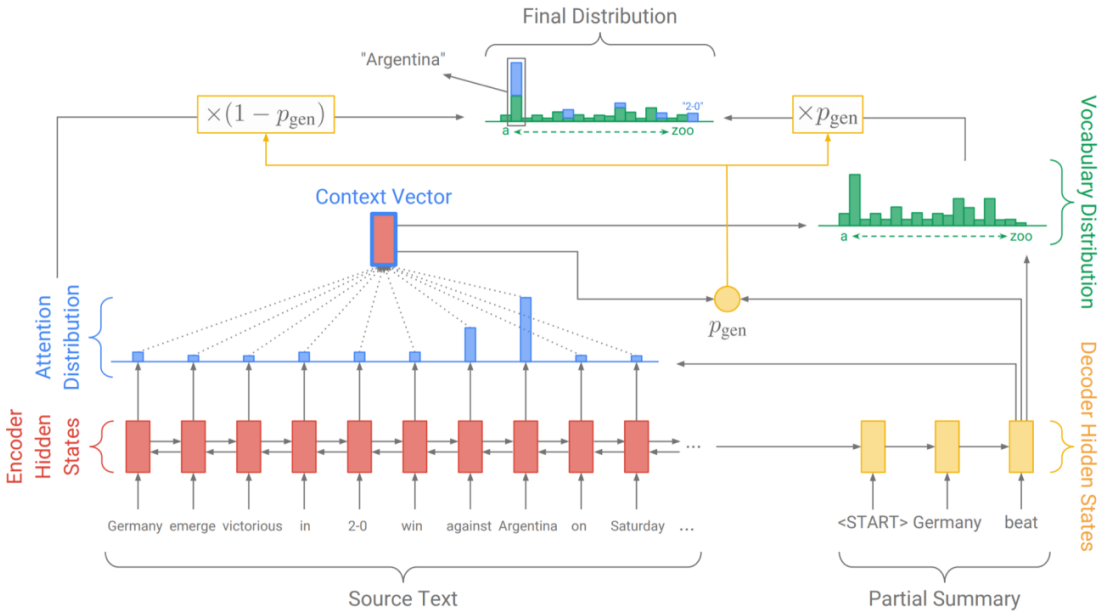
pre-neural summarization: pipeline: content selection(choose some relevant sentences), information ordering(order those selected sentences), sentence realization(e.g. simplification, coherency).



neural summarization: seq2seq + attention to write an abstractive summary, subsequent model also uses multi-head attention, reinforcement learning, and so on, to improve model's performance. However, seq2seq + attention mechanism is bad for copying a rare word from text, so some enhanced model hybrids copying mechanism with

seq2seq + attention that allows model to extract or generate summarization. In particular, each decoder hidden state is taken a dot product with attended context vector to generate the probability of generating the next word p_{gen}. In this case, the total probability is a mixture of generation distribution plus copying distribution.

$$P(w) = p_{gen}P_{vocab}(w) + (1 - p_{gen})\sum_i a_i^t$$



bottom-up summarization: Neural summarization with copying mechanism is sometimes bad, because of copying too much and bad content selection. Bottom-up approach has an additional stage(content selection stage) that use a sequence tagging model to tag words as include or not include, and only those words tagged include have privilege to be attended.

ROUGE: an evaluation for summarization, different from BLEU, ROUGE has no brevity penalty and is based on recall and separated for each n-gram.

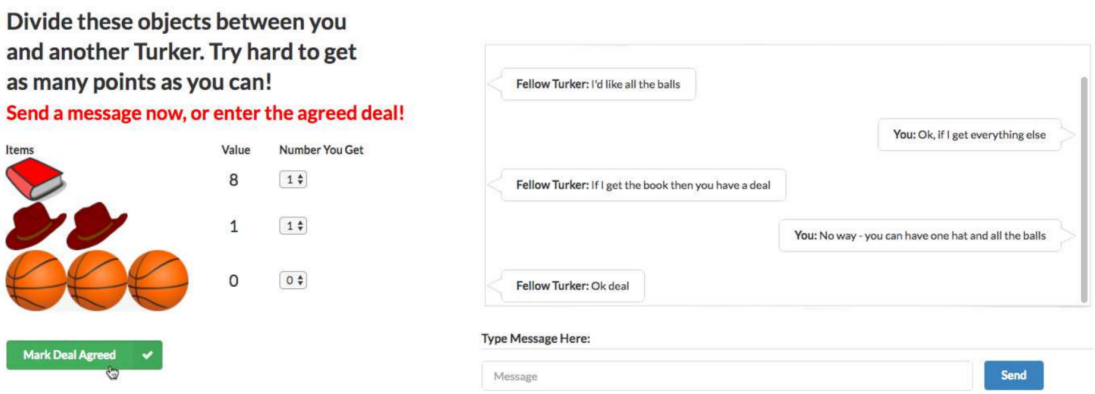
Suppose the candidate summary is referred as $\mathbf{c}$, the reference summaries are $\mathbf{r}_1, \mathbf{r}_2 \cdots, \mathbf{r}_k$. The method for computing n-gram ROUGE:

$$\text{ROUGE}(n - \text{gram}) = \frac{\sum_{s \in \{r_1, r_2 \cdots, r_k\}} \sum_{n-\text{gram} \in s} \text{Count}_{\text{match}}(n - \text{gram})}{\sum_{s \in \{r_1, r_2 \cdots, r_k\}} \sum_{n-\text{gram} \in s} \text{Count}(n - \text{gram})}$$

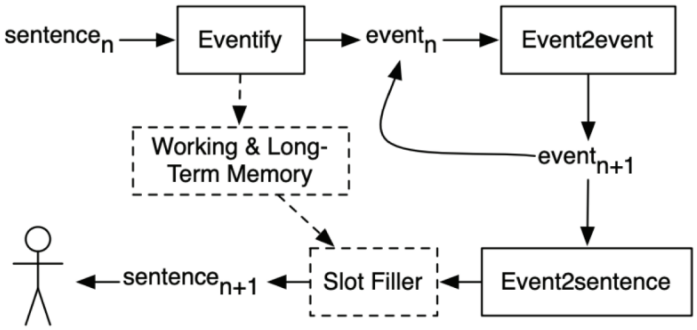
dialogue: answer human's questions, chat with people, etc. based on some queries. The model is seq2seq based, but has many serious problems(like generate boring, irrelevant, repetitive out-of-context responses).

To solve these issues, one can upweight rare words during beam search, sample from the distribution that is written by human, block repetitive n-grams during test time, define a specific training objective for a problem to optimize them.

negotiation dialogue: two agents negotiate something via natural language until they reach an agreement, and they have different value functions.



storytelling: generate a story-like paragraph which has plot, character and ending. They can be generated from an image, writing prompt, or story continuation based on a few sentences. The challenge of storytelling is the generated story is not coherent, sensical and lack of plot. To handle that, some models introduce event2event story generation that generates a story event by event, not sentence by sentence.

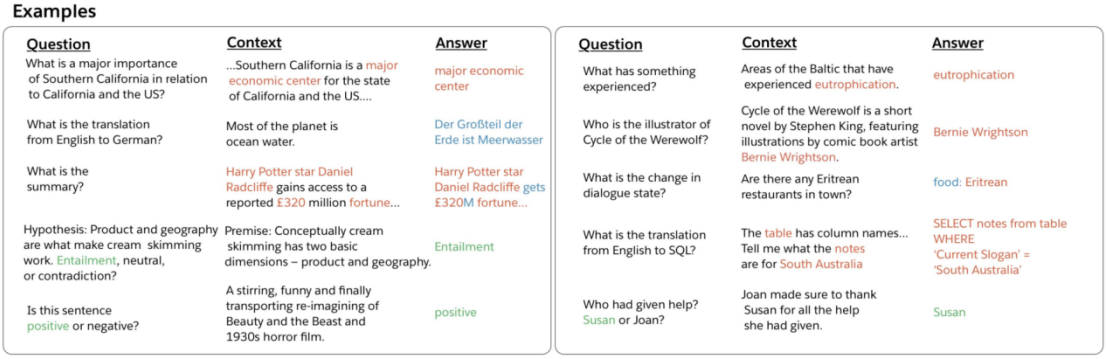


During story generation, the model keeps track of events, entities, characters, etc. for a better story plot.
Applications for storytelling: recipe generation, poetry generation, code generation, ...

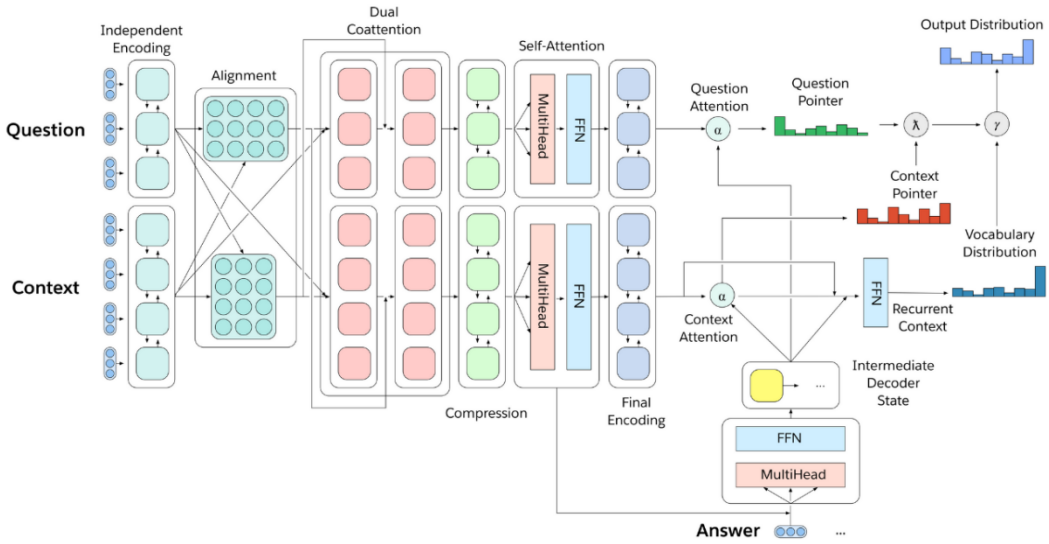
7. Advanced Topic

multitask learning: The motivation of multitask learning is the limit of single-task learning which is performing well for a specific task(machine learning, question answering, summarization, etc.), but it is not general to process multiple tasks. So if we train a large model that can handle multiple NLP tasks, it is more easier to adapt new tasks.
multitask learning as question answering: ask our model to do what task we want to, i.e., given a piece of context c and question q, then output the answer a for that question.

$$a = \text{model}(c, q)$$



Specifically, the answer is generated one word at a time by extracting from context, question or from external vocabulary, and it is crucial to switch pointer in choosing between those three options.



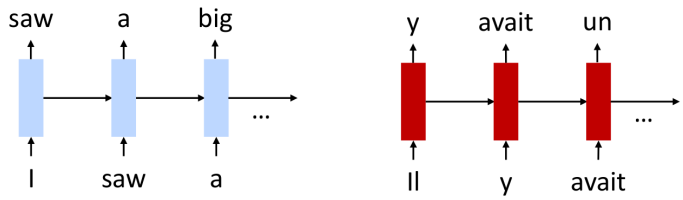
two switches: γ decides whether copy or select word from external vocabulary, λ and decides whether to copy from context or question.

training strategy: the more difficult tasks are trained at first and iterated more epochs.

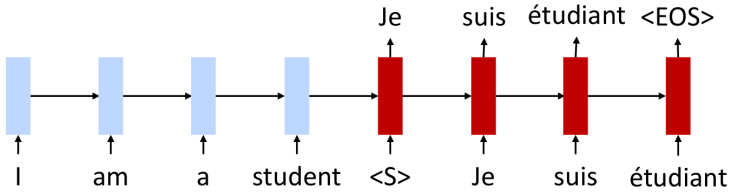
unsupervised machine translation: If some language data are less labeled, i.e. lack of <source, target> pair, we need a model that can automatically label unlabeled data.

pre-training: pretrain two separate language models(encoder and decoder), and then train jointly on labeled bilingual data.

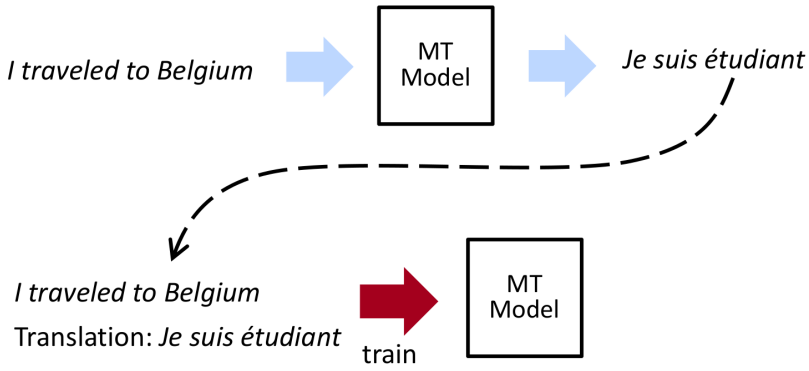
1. Separately Train Encoder and Decoder as Language Models



2. Then Train Jointly on Bilingual Data

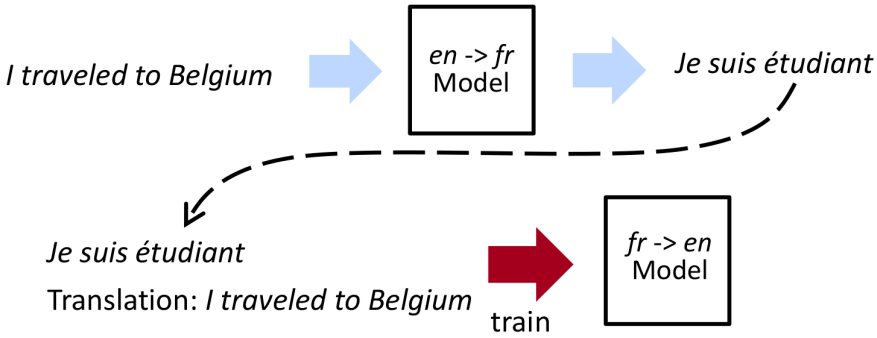


self-training: label unlabeled data to get noisy training examples.

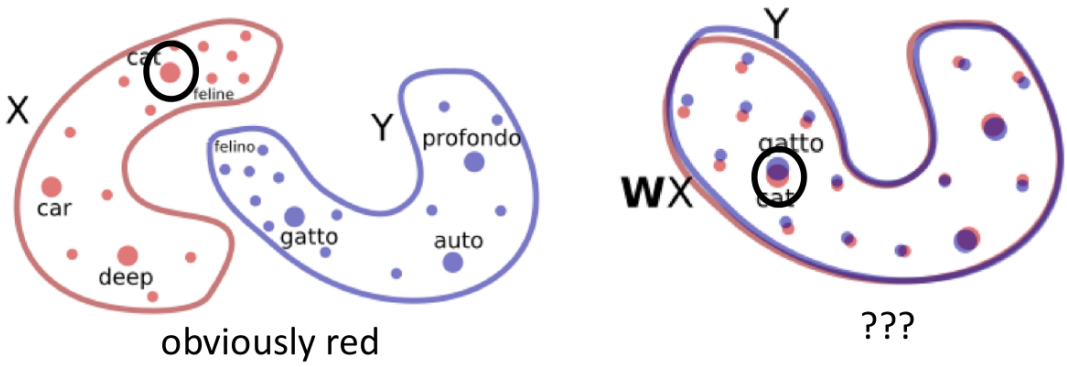


Not effective since the model has already seen this example.

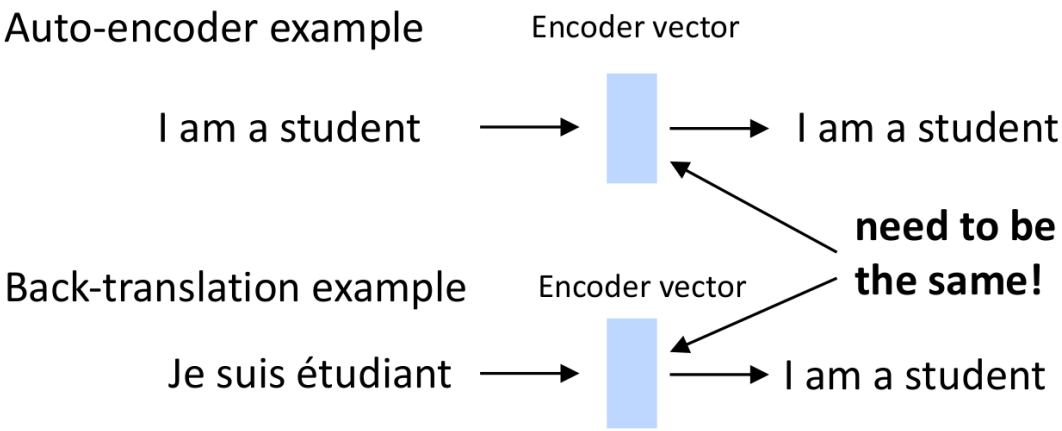
back-translation: two separate machine translation models going in opposite directions(e.g. English to French, French to English).



unsupervised word translation: run word2vec on monolingual data to get two word embeddings X and Y. Then use adversarial method to learn a transformed matrix W to linearly transform X to Y and train a discriminator to differentiate transformed embedding WX and original embedding Y. Our goal is to get a matrix W that fools discriminator.



Hence, the encoder vector of auto-encoder and back-translation is almost the same, so it can be cross-lingual.


Auto-encoder example
Encoder vector
I am a student
I am a student
need to be
the same!
Back-translation example
Encoder vector
Je suis étudiant
I am a student