Intelligent Agents

1d-CNNs LSTMs ELMo Transformers BERT GPT

Ralf Möller
Universität zu Lübeck
Institut für Informationssysteme

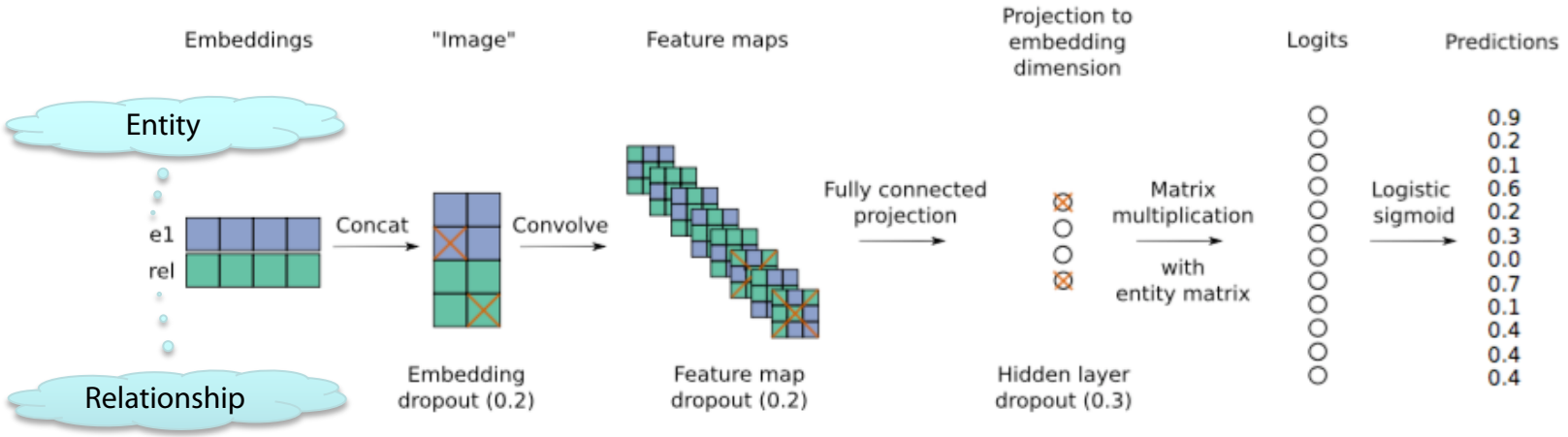
Acknowledgements

- CS546: Machine Learning in NLP (Spring 2020)
  - *http://courses.engr.illinois.edu/cs546/*
  - Julia Hockenmaier http://juliahmr.cs.illinois.edu
  - RNNs, LSTMs, ELMo, Transformers

- Machine Learning (Spring 2020)
  - http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html
  - 李宏毅 (Hung-yi Lee) http://speech.ee.ntu.edu.tw/~tlkagk/
  - ELMo, BERT: http://speech.ee.ntu.edu.tw/~tlkagk/courses/ML_2019/Lecture/BERT%20(v3).pdf

Recap: Embedding Approaches

- ConvE: Uses convolutional network



- Does it really make sense to use 2d-CNN for graph data?
  - Why is it effective to apply 2d filters to data that is embedded into 2d space in a rather arbitrary way?

Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, Sebastian Riedel, Convolutional 2D Knowledge Graph Embeddings. In: Proc. AAAI-18. **2018**.

Recap: Convolution

Input image

Convolution Kernel

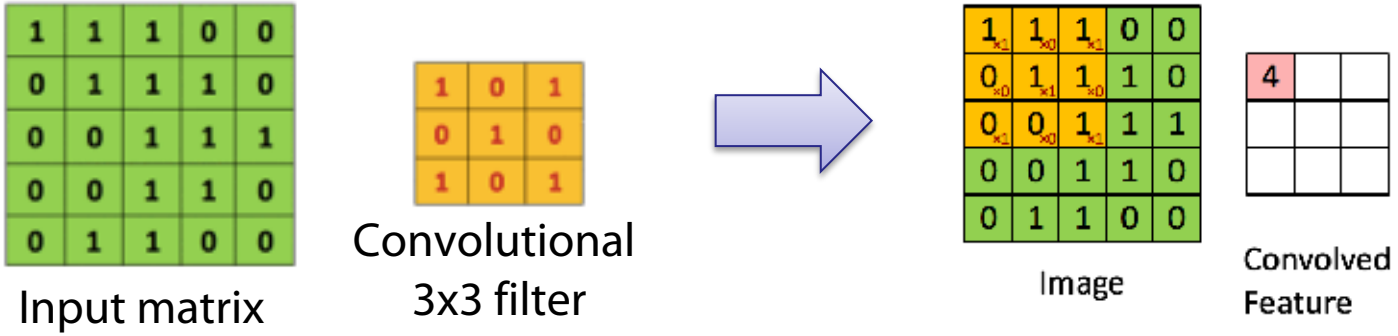
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map

Recap: Convolutional Neural Networks (CNNs)

Main CNN idea for text:
Compute vectors for n-grams and group them afterwards

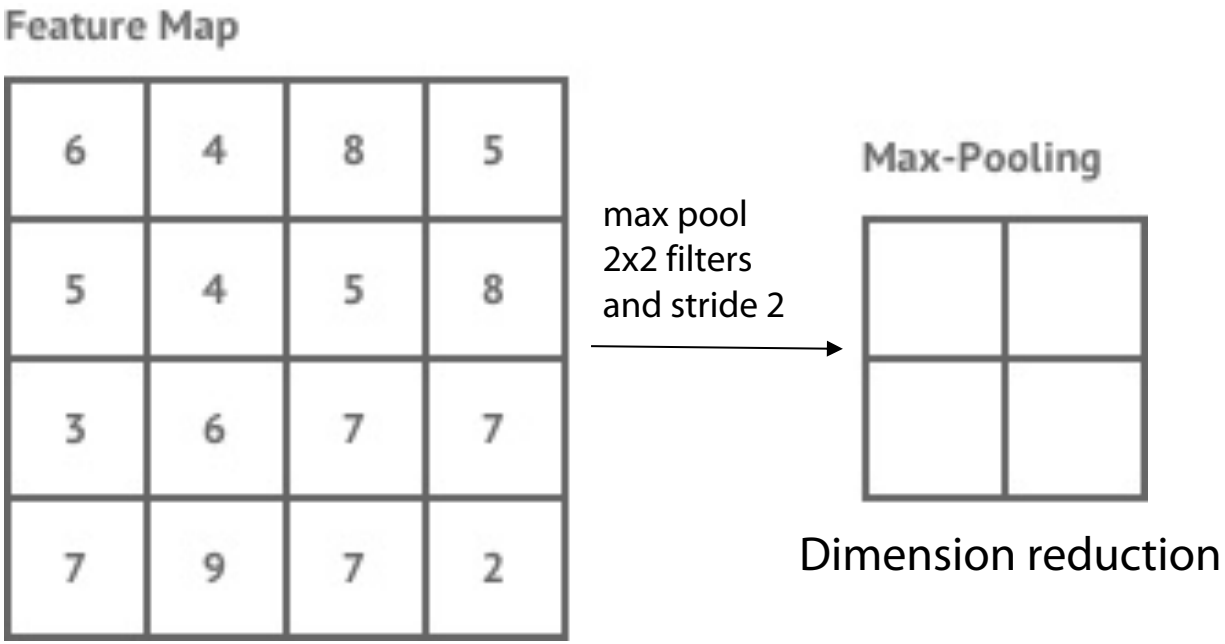
Example: “this takes too long” compute vectors for:
This takes, takes too, too long, this takes too, takes too long, this takes too long



http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Recap: Convolutional Neural Networks (CNNs)

Main CNN idea for text:
Compute vectors for n-grams and **group them afterwards**



https://shafeentejani.github.io/assets/images/pooling.gif

1d-CNNs for text

Text is a (variable-length) sequence of words (word vectors)

We can use a 1d-CNN to slide a window of n tokens across:

- filter size n = 3, stride = 1, no padding

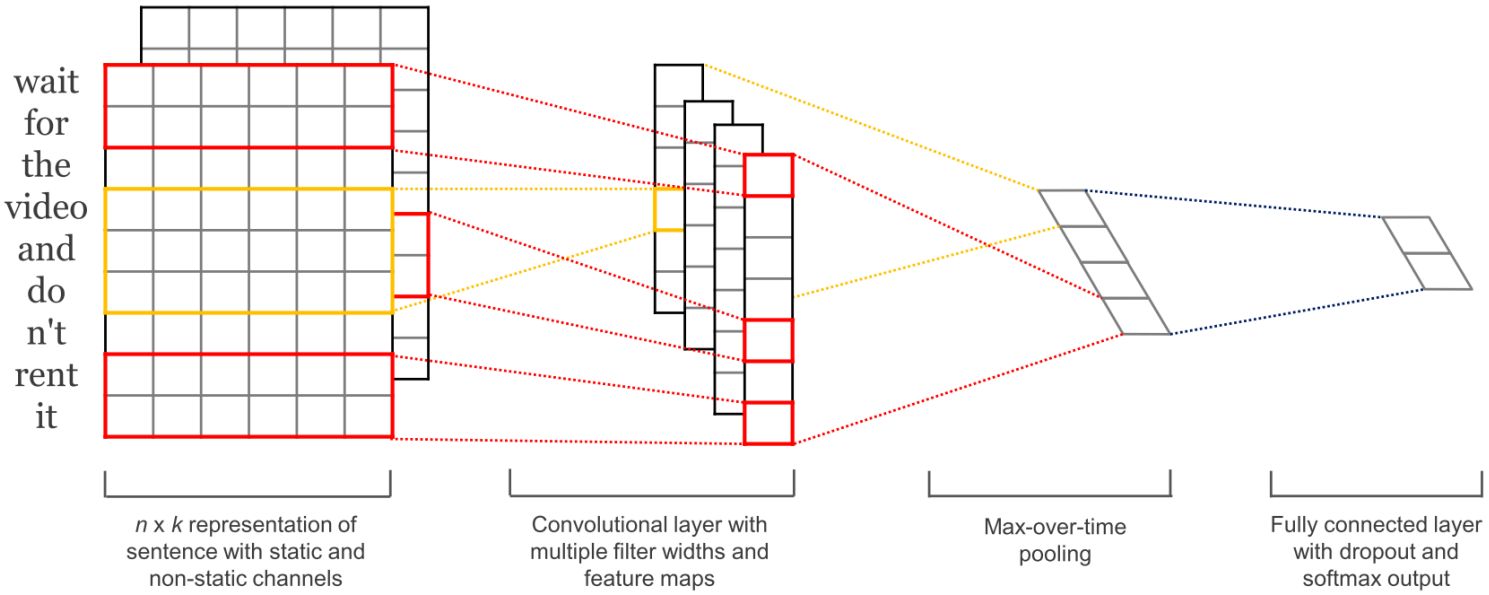
```
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
```

- filter size n = 2, stride = 2, no padding:

```
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
```

CNNs (w/ ReLU and maxpool) can be used for classifying (parts of) the text

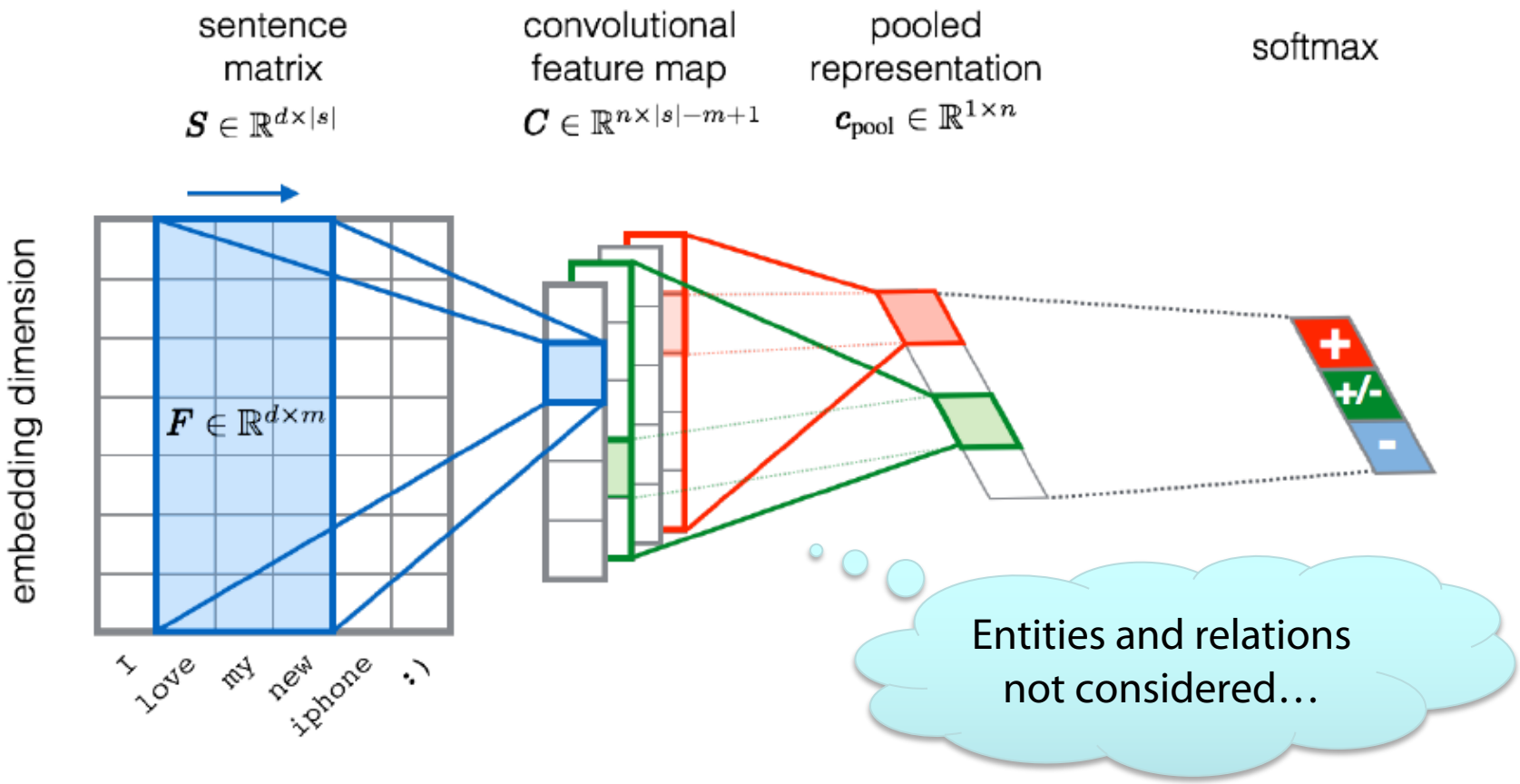
CNN with multiple filters



Kim, Y. "Convolutional Neural Networks for Sentence Classification", EMNLP (2014)

sliding over 3, 4 or 5 words at a time

CNN for text classification



Severyn, Aliaksei, and Alessandro Moschitti. "UNITN: Training Deep Convolutional Neural Network for Twitter Sentiment Classification." *SemEval@ NAACL-HLT*. 2015.

Fasttext (https://fasttext.cc)

- Library for word embeddings and text classification
  - static word embeddings and ngram features
  - that get averaged together in one hidden layer
  - hierarchical softmax output over class labels

- Enriching word vectors with subword information
  - Skipgram model where each word is a sum of character ngram embeddings and its own embedding
  - Each word is deterministically mapped to ngrams

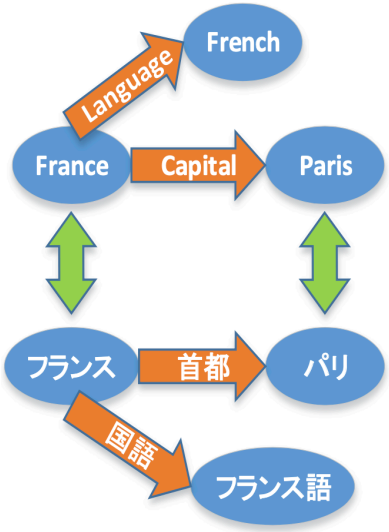
Piotr Bojanowski, Edouard Grave, Armand Joulin, Tomas Mikolov. Enriching Word Vectors with Subword Information. Transactions of the Association for Computational Linguistics, Volume 5. 135-146. **2017.**

Armand Joulin, Edouard Grave, Piotr Bojanowski, Tomas Mikolov, Bag of Tricks for Efficient Text Classification. Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers. 427-431. **2017**.

Alon Jacovi, Oren Sar Shalom, Yoav Goldberg. Understanding Convolutional Neural Networks for Text Classification. In Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP. **2018**.

Multilingual Knowledge Graph Embeddings

- **Multilingual KG Embeddings**

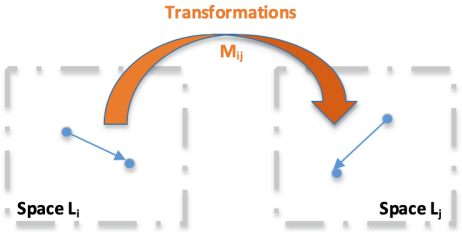


Entities

Paris (0.036, -0.12, …, 0.323)
France (0.138, 0.551, …, 0.222)
…

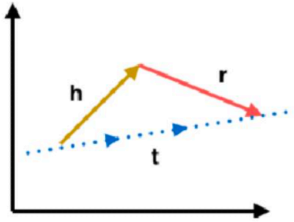
Separated embedding spaces

Semantic Transfer



(Cross-lingual) transforms of embedding spaces

Monolingual Relations



(Monolingual) vector algebraic operations

- **Applications**
  - Knowledge alignment
  - Phrasal translation
  - Causality reasoning
  - Cross-lingual QA
  - etc..

MTransE [Chen et al. 2017a; 2017b]
- Joint learning of structure encoders and an alignment model
- Alignment techniques: Linear transforms (best), vector translations, collocation (minimizing L2 distance)

JAPE [Sun et al. 2017]
+ Logistic-based proximity normalizer for entity attributes

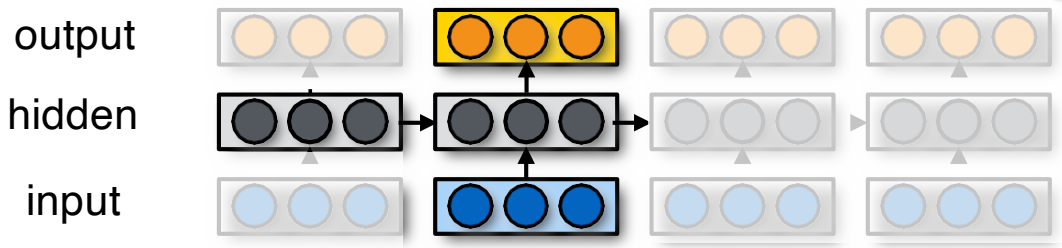
ITransE [Zhu et al. 2017]
- self-training + cross-lingual collocation of entity embeddings

KDCoE [Chen et al., 2018] leverages a weakly aligned multilingual KG for semi-supervised cross-lingual learning using entity descriptions

BootEA [Sun et al., 2018] tries iteratively enlarge the labeled entity pairs based on the bootstrapping strategy

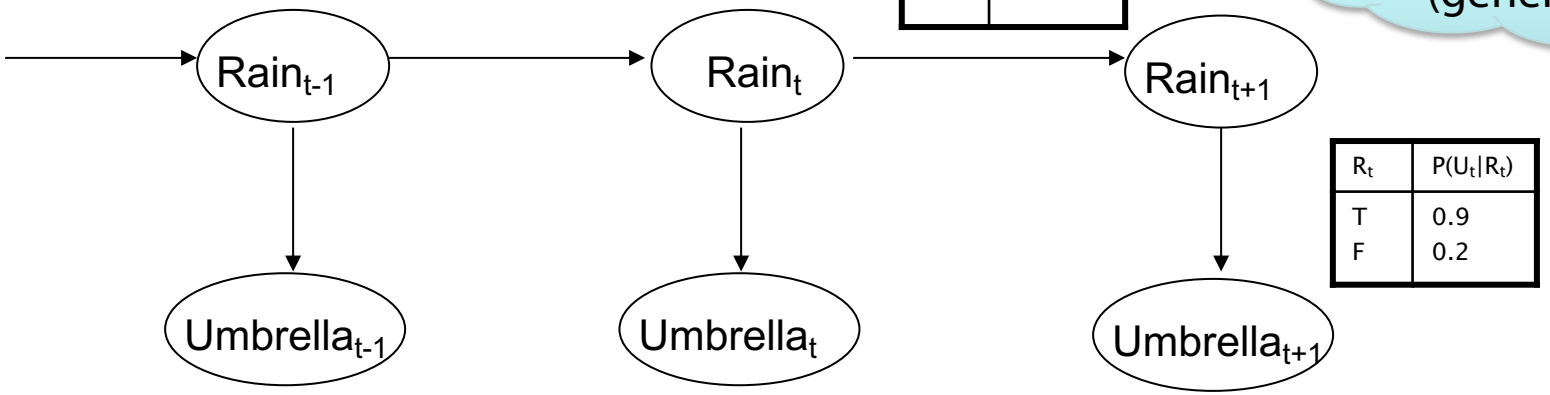
Recursive Networks – Or: Copying the Pattern

- Basic computational network copied per time slice
- Input: previous hidden state, output: next hidden state



Computational model

- Compare with HMM:



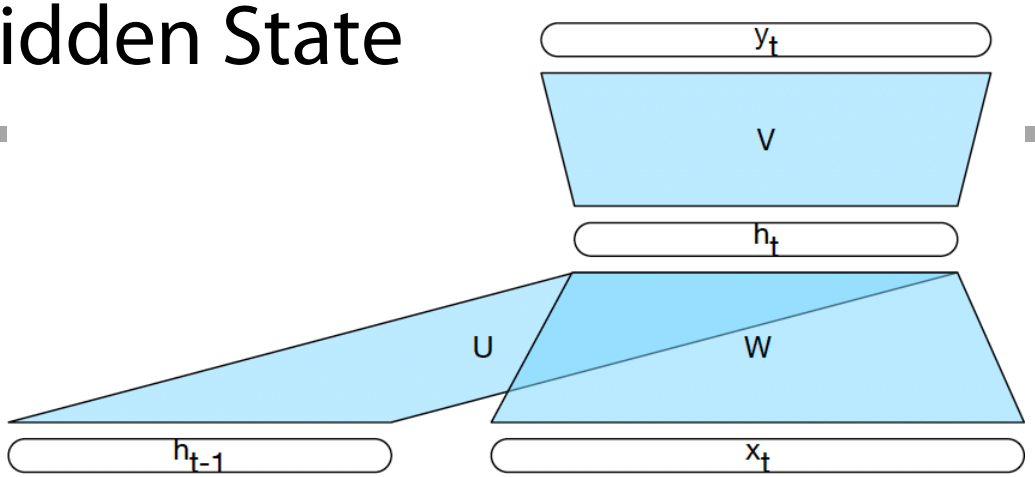
R_{t-1}	$P(R_t \mid R_{t-1})$
T	0.7
F	0.3

R_t	$P(U_t \mid R_t)$
T	0.9
F	0.2

Declarative model (generative)

Computing the Hidden State

- RNN



Computing the hidden state at time t: $\mathbf{h}^{(t)} = g(\mathbf{U}\mathbf{h}^{(t-1)} + \mathbf{W}\mathbf{x}^{(t)})$

The i-the element of $\mathbf{h}_t$: $h_i^{(t)} = g\left(\sum_j U_{ji} h_j^{(t-1)} + \sum_k W_{ki} x_k^{(t)}\right)$

- HMM Filtering

$$P(X_{t+1} / e_{1:t+1}) = \alpha P(e_{t+1} / X_{t+1}) \sum_{X_t} P(X_{t+1} / x_t) P(x_t / e_{1:t})$$

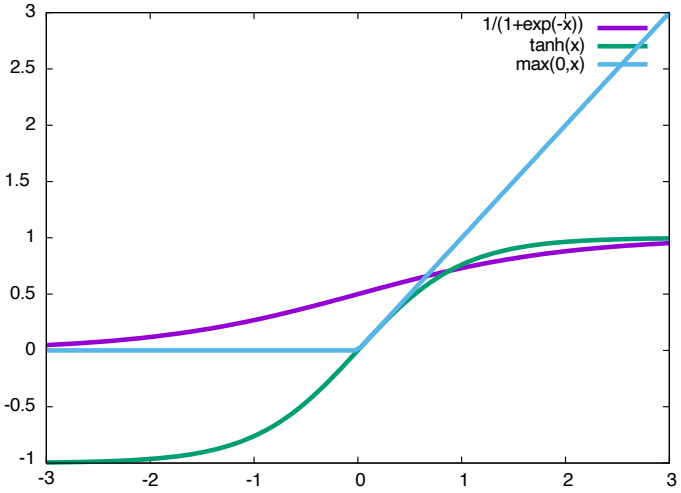
What about smoothing?

Recap: Activation Functions

Sigmoid (logistic function):

$\sigma(x) = 1/(1 + e^{-x})$

Returns values bound above and below in the $0,1$ range



Hyperbolic tangent:

$\tanh(x) = (e^{2x-1})/(e^{2x+1})$

Returns values bound above and below in the $-1, +1$ range

Rectified Linear Unit:

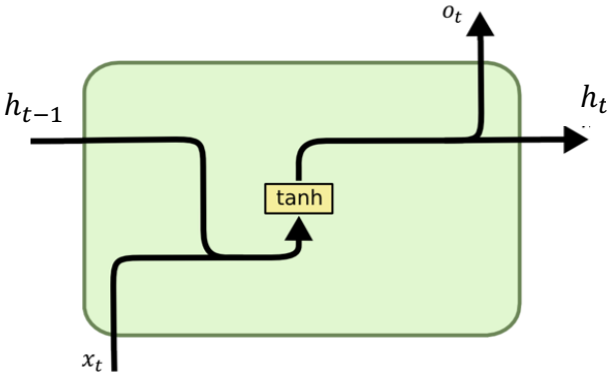
$\mathrm{ReLU}(x) = \max(0, x)$

Returns values bound below in the $0, +\infty$ range

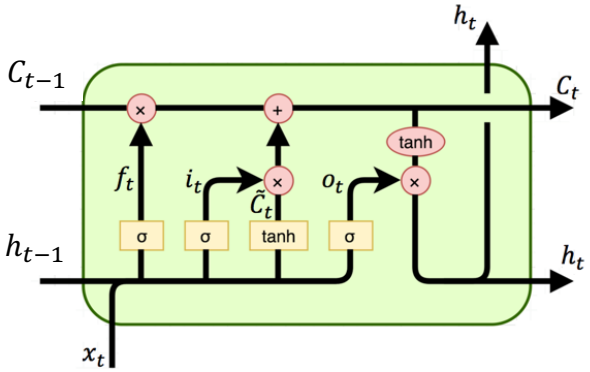
RNN Variants: LSTMs, GRUs

- Long Short Term Memory networks (LSTMs) are RNNs with a more complex architecture to combine the last hidden state with the current input.
- Gated Recurrent Units (GRUs) are a simplification of LSTMs
- Both contain "gates" to control how much of the input or past hidden state to forget or remember

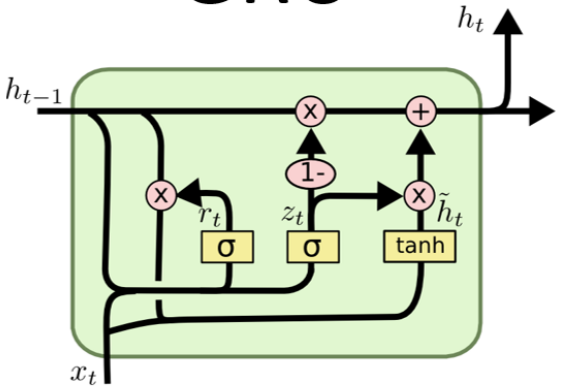
RNN



LSTM



GRU



Gates

- A gate performs element-wise multiplication of
  - the output of a *d*-dimensional sigmoid layer (all elements between 0 and 1), and
  - a *d*-dimensional input vector
- Result: a *d*-dimensional output vector which is like the input, except some dimensions have been (partially) "forgotten"

RNNs for Language Modeling

- If our vocabulary consists of V words, the output layer (at each time step) has V units, one for each word.

- The softmax gives a distribution over the V words for the next word.

- To compute the probability of a string $w_0 w_1 \ldots w_n$ w_{n+1} (where $w_0 = \langle s \rangle$, and $w_{n+1} = \langle \backslash s \rangle$), feed in w_i as input at time step i and compute

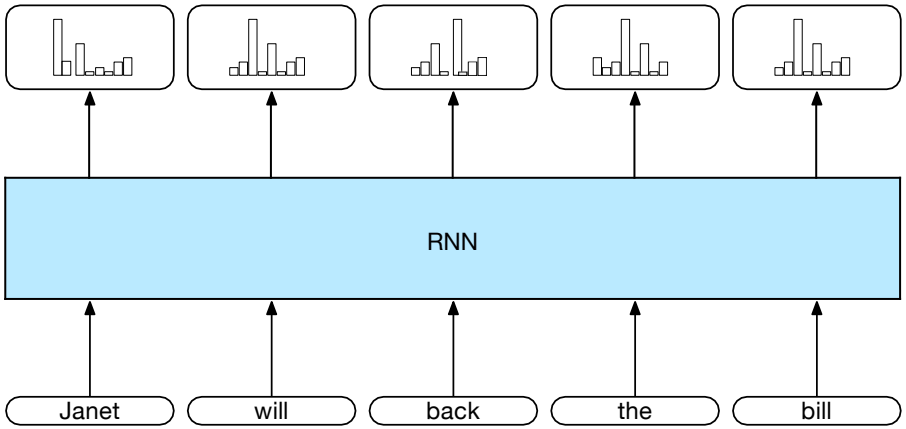
$$\prod_{i=1..n+1} P(w_i | w_0 \ldots w_{i-1})$$

RNNs for Sequence Labeling

- In sequence labeling, we want to assign a label or tag t_i to each word w_i
- Now the output layer gives a distribution over the T possible tags.
- The hidden layer contains information about the previous words and the previous tags.
- To compute the probability of a tag sequence $t_1 \ldots t_n$ for a given string $w_1 \ldots w_n$ feed in w_i (and possibly t_{i-1}) as input at time step i and compute $P(t_i \mid w_1 \ldots w_{i-1}, t_1 \ldots t_{i-1})$

Basic RNNs for Sequence Labeling

Each time step has a distribution over output classes

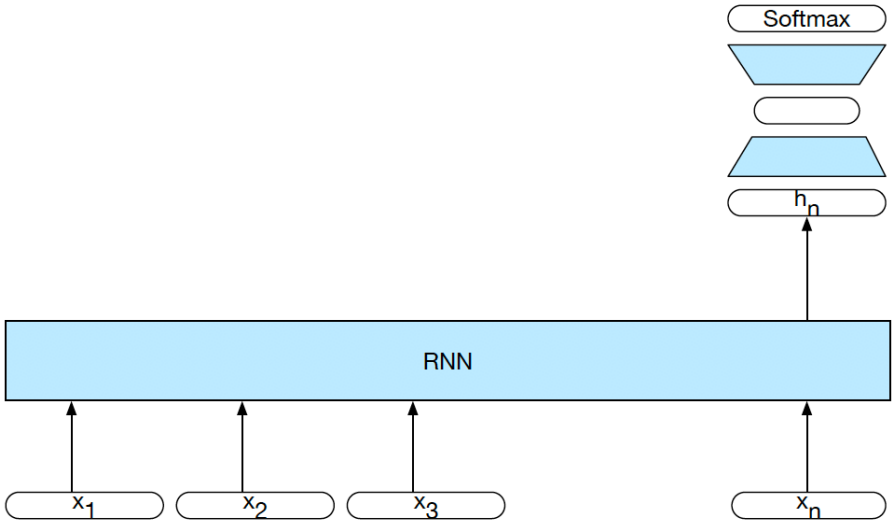


Extension: add a HMM/CRF layer to capture dependencies among labels of adjacent tokens.

RNNs for Sequence Classification

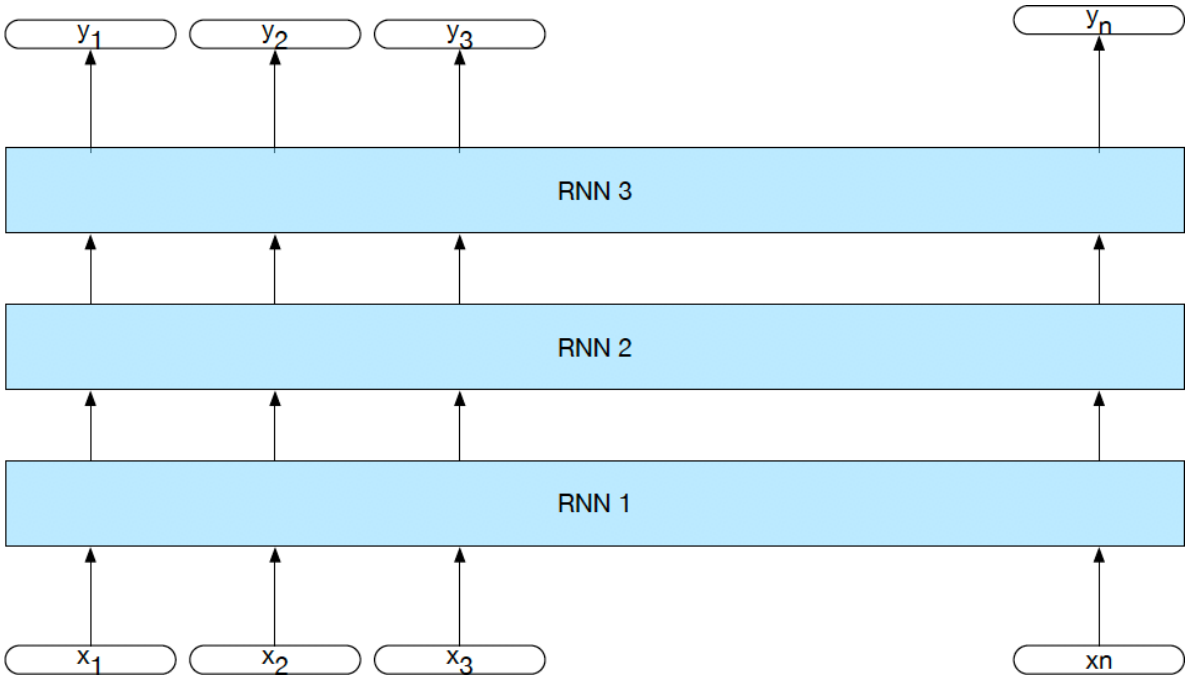
If we just want to assign a label to the entire sequence, we don't need to produce output at each time step, so we can use a simpler architecture.

We can use the hidden state of the last word in the sequence as input to a feedforward net:

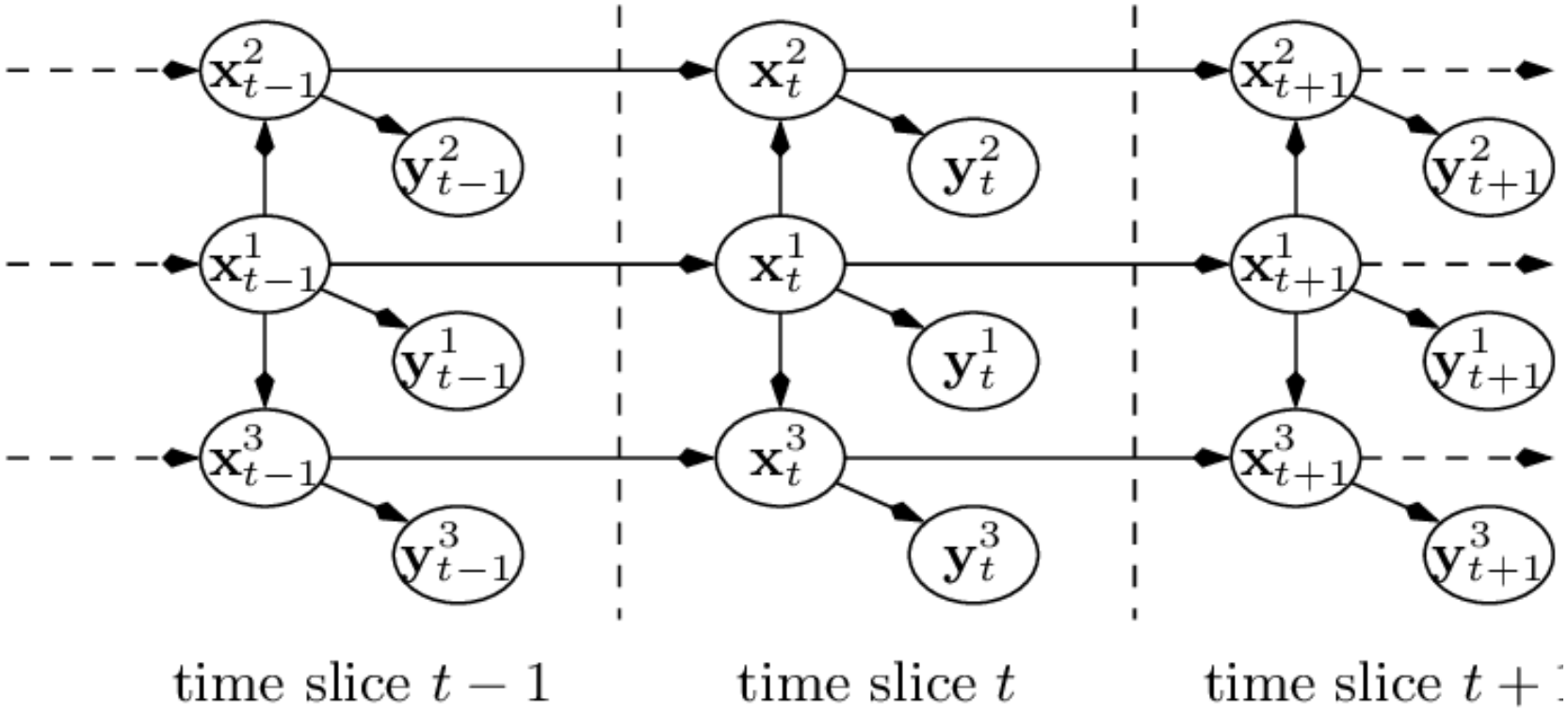


Stacked RNNs

We can create an RNN that has “vertical” depth (at each time step) by stacking multiple RNNs:



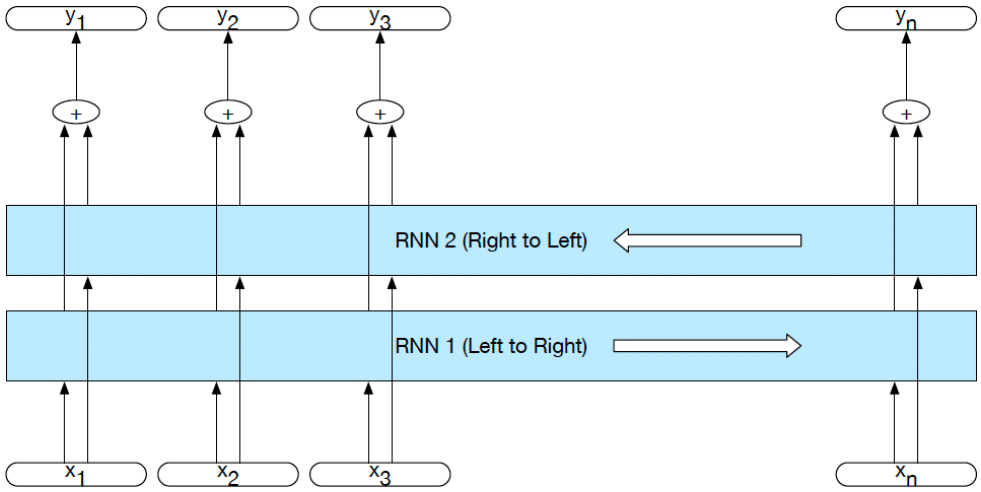
Comparison with Dynamic Baysian Networks



Bidirectional RNNs

Computational specification of smoothing?

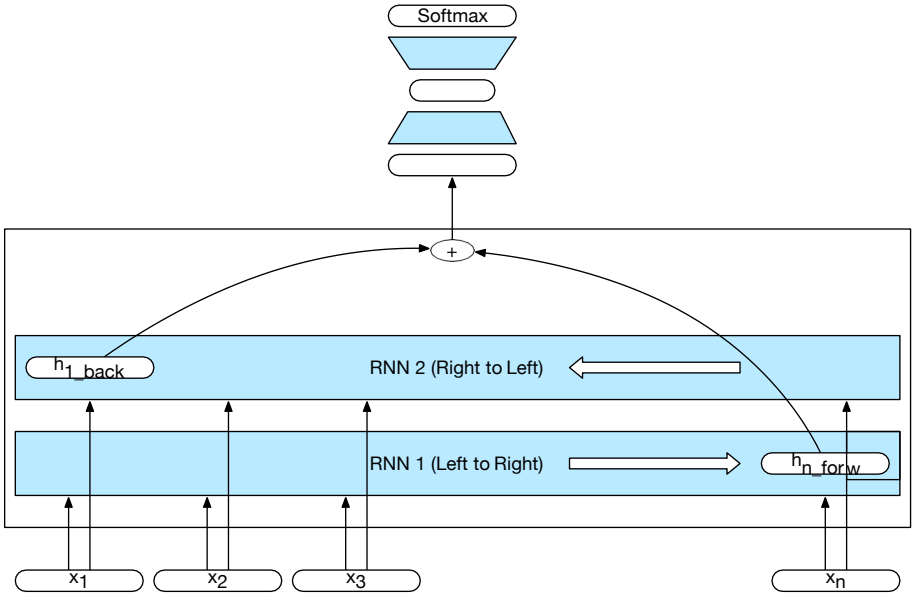
Unless we need to generate a sequence, we can run *two* RNNs over the input sequence — one in the forward direction, and one in the backward direction.
Their hidden states will capture different context information



Hidden state of biRNN: $\mathbf{h}_{\mathbf{bi}}^{(\mathbf{t})} = \mathbf{h}_{\mathbf{fw}}^{(\mathbf{t})} \oplus \mathbf{h}_{\mathbf{bw}}^{(\mathbf{t})}$ where $\oplus$ is typically concatenation (or element-wise addition, multiplication)

Bidirectional RNNs for sequence classification

Combine the hidden state of the last word of the forward RNN and the hidden state of the first word of the backward RNN into a single vector

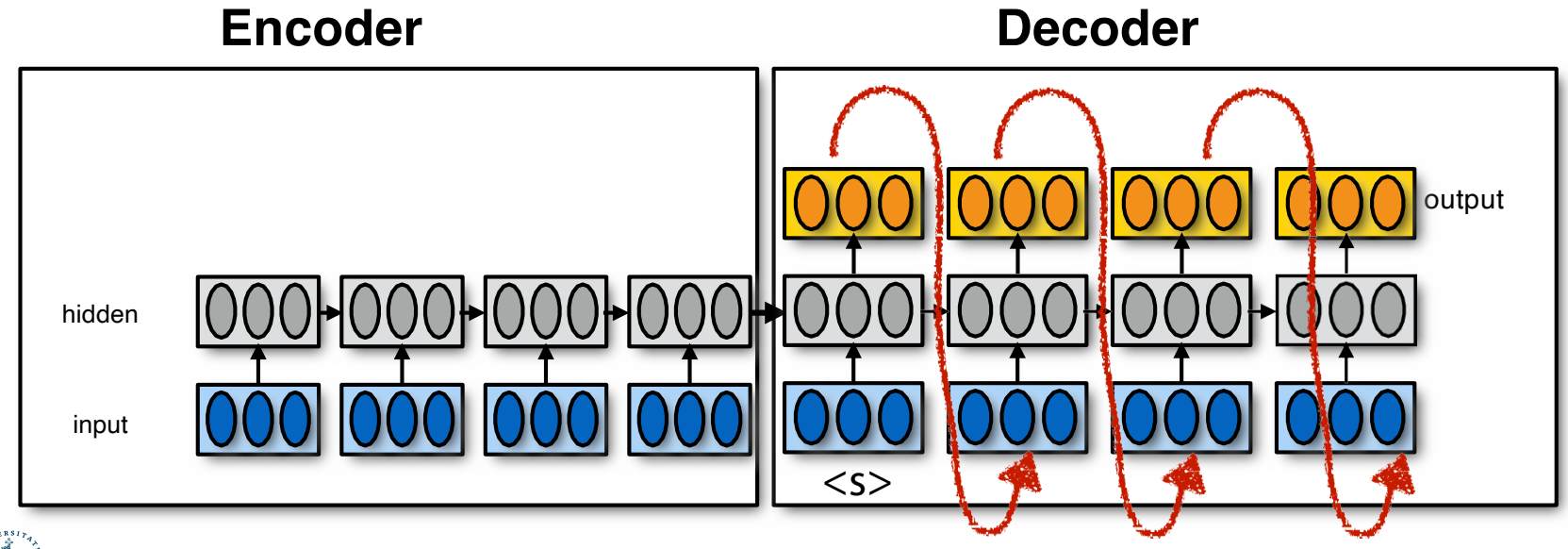


Encoder-Decoder (seq2seq) model

- Task: Read an input sequence and return an output sequence
  - Machine translation: translate source into target language
  - Dialog system/chatbot: generate a response

- Reading the input sequence: RNN Encoder

- Generating the output sequence: RNN Decoder



Encoder-Decoder (seq2seq) Model

Encoder RNN:

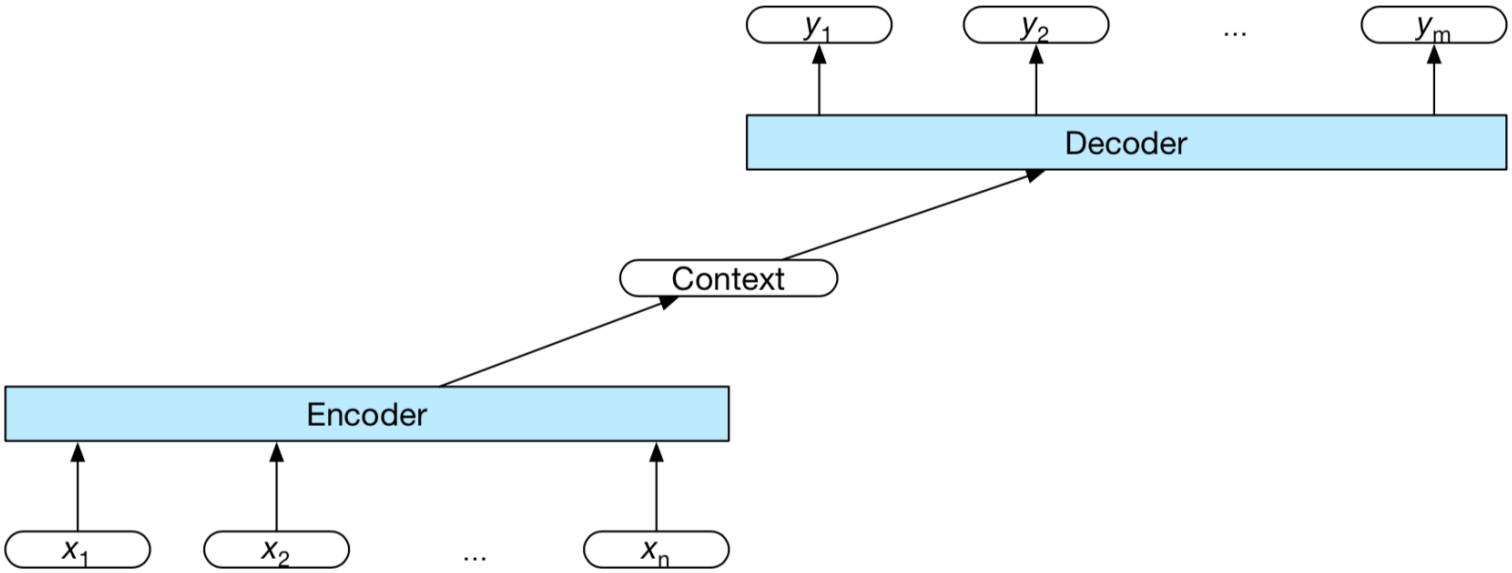
reads in the input sequence
passes its last hidden state to the initial hidden state of the decoder

Decoder RNN:

generates the output sequence
typically uses different parameters from the encoder
may also use different input embeddings

A More General View of seq2seq

In general, any function over the encoder's output can be used as a representation of the context we want to condition the decoder on.

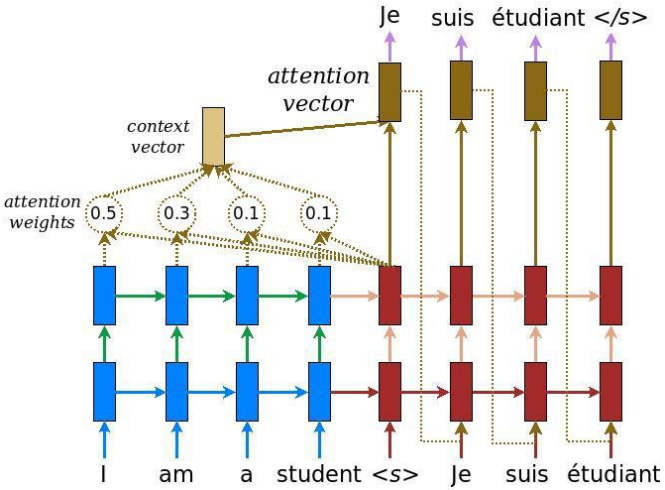


We can feed the context in at any time step during decoding (not just at the beginning).

Attention Mechanisms

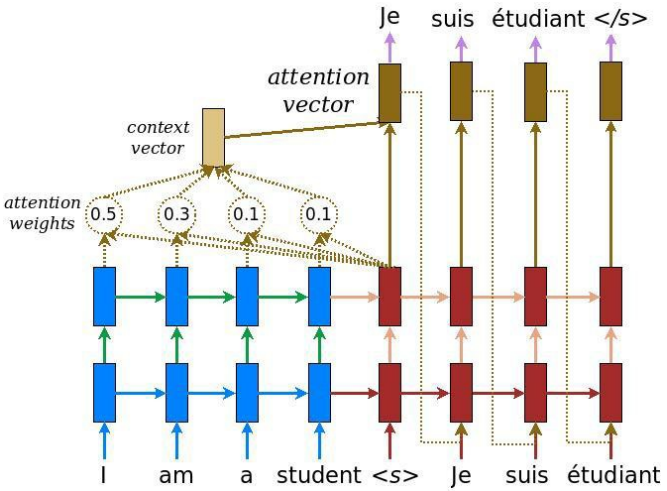
Define a **distribution** $\alpha = (\alpha_{1t}, \ldots, \alpha_{St})$ **over the *S* elements of the input** sequence **that depends on the current output** element t (with $\sum_{s=1..S} \alpha_{st} = 1; \forall_{s \in 1 \ldots S}\, 0 \leq \alpha_{st} \leq 1$)

Use this distribution to compute a **weighted average of the input:** $\sum_{s=1..S} \alpha_{st} o_s$ and feed that into the decoder.



hhttps://www.tensorflow.org/tutorials/text/nmt_with_attention

Attention Mechanisms



$\mathbf{h}_t$: current hidden state of decoder (target)
$\mathbf{h'}_s$: output of the encoder for word s (source)
Attention weights α_{ts}: distribution over $\mathbf{h'}_s$
α_{ts} depends on score($\mathbf{h}_t$, $\mathbf{h'}_s$)
Context vector $\mathbf{c}_t$: weighted average of $\mathbf{h'}_s$
Attention vector $\mathbf{a}_t$: computed by feedforward layer over $\mathbf{c}_t$ and $\mathbf{h}_t$

$$\alpha_{ts} = \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'=1}^{S} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \quad \text{[Attention weights]} \quad (1)$$

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s \quad \text{[Context vector]} \quad (2)$$

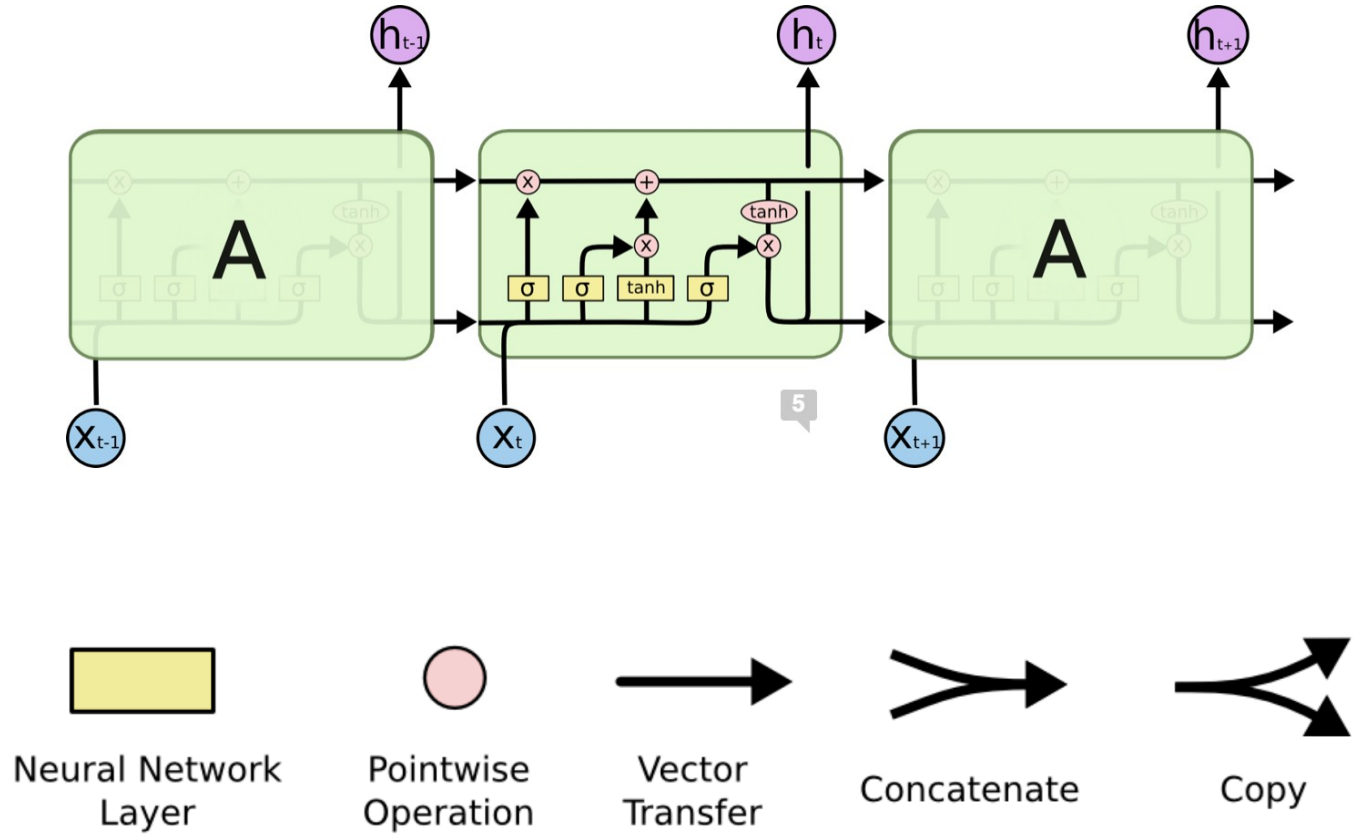
$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W}_{\boldsymbol{c}}[\boldsymbol{c}_t; \boldsymbol{h}_t]) \quad \text{[Attention vector]} \quad (3)$$

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_1} \boldsymbol{h}_t + \boldsymbol{W_2} \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases} \quad (4)$$

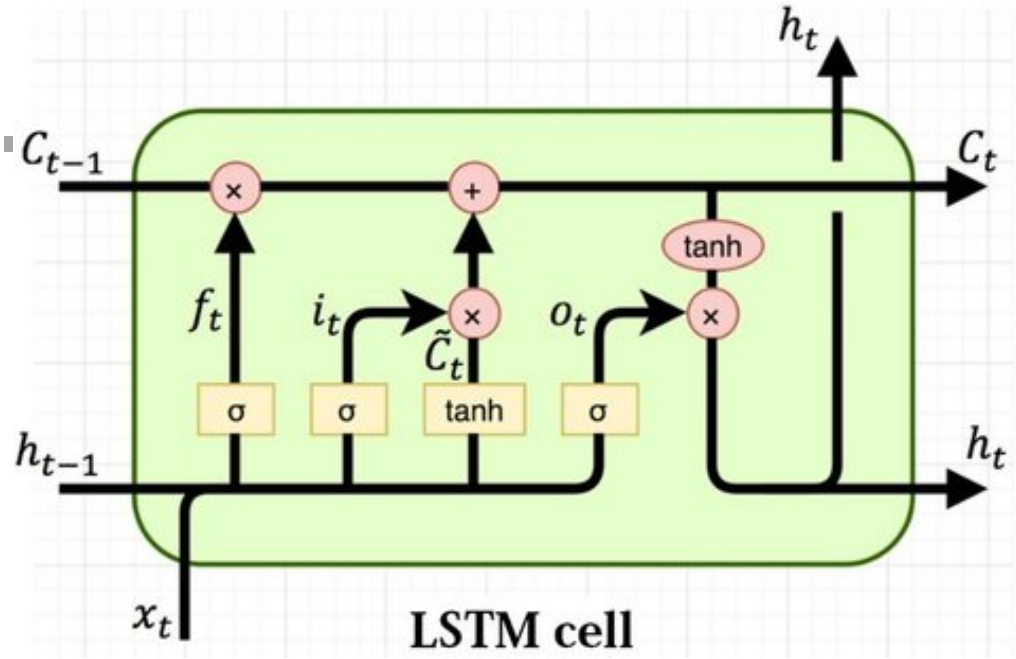
From RNNs to LSTMs

- In simple RNNs, hidden state depends on previous hidden state and on the input:
  - $\mathbf{h}_t = g(W_h[\mathbf{h}_{t-1}, \mathbf{x}_t] + b_h)$ with e.g. *g*=tanh
- Vanishing gradient problem
  - RNNs can't be trained effectively on long sequences
- LSTMs (Long Short-Term Memory networks) to the rescue
  - Additional cell state passed through the network and updated at each time step
  - LSTMs define four different layers (gates) that read in the previous hidden state and current input.

Long Short Term Memory Networks (LSTMs)



https://colah.github.io/posts/2015-08-Understanding-LSTMs/



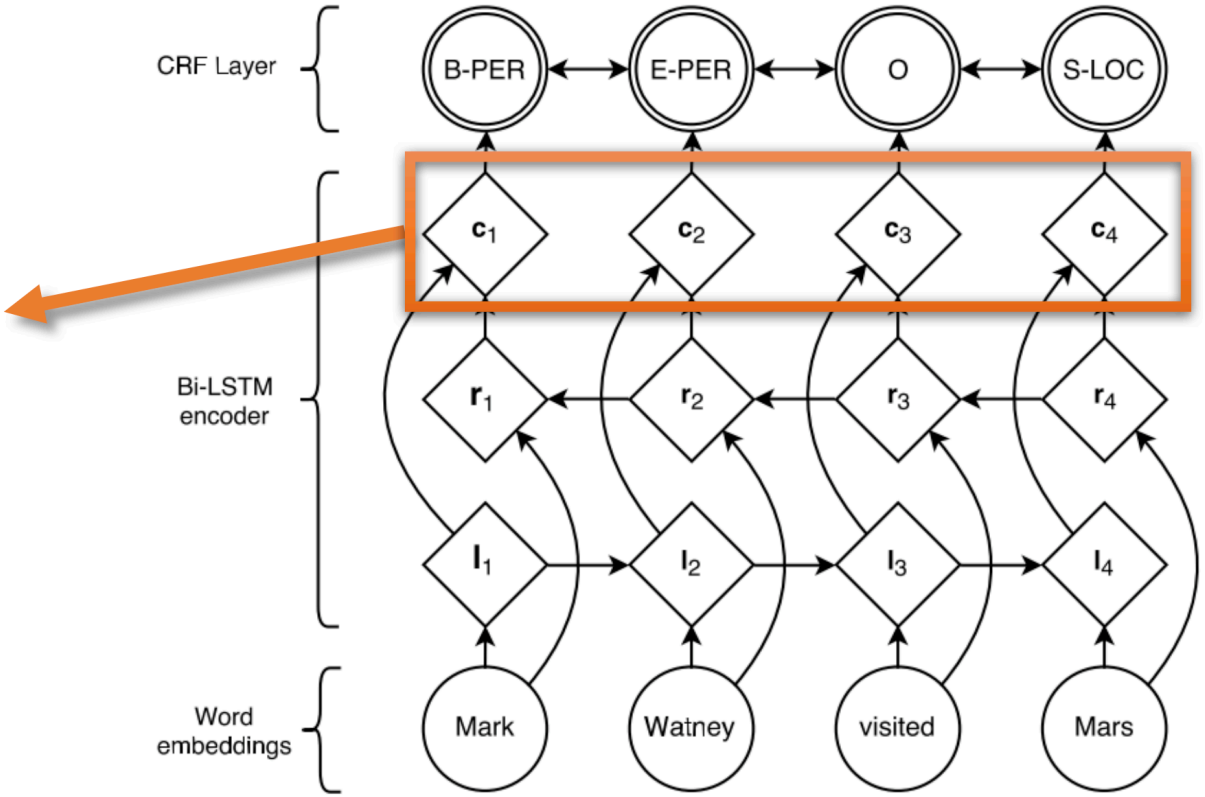
$$i_t = \sigma\left(x_t U^i + h_{t-1} W^i\right)$$
$$f_t = \sigma\left(x_t U^f + h_{t-1} W^f\right)$$
$$o_t = \sigma\left(x_t U^o + h_{t-1} W^o\right)$$
$$\tilde{C}_t = \tanh\left(x_t U^g + h_{t-1} W^g\right)$$
$$C_t = \sigma\left(f_t * C_{t-1} + i_t * \tilde{C}_t\right)$$
$$h_t = \tanh(C_t) * o_t$$

At time t, the LSTM cell reads in
— a *c*-dimensional previous cell state vector $\mathbf{c}_{t-1}$
— an *h*-dimensional previous hidden state vector $\mathbf{h}_{t-1}$
— a *d*-dimensional current input vector

At time t, the LSTM cell returns
— a *c*-dimensional previous cell state vector $\mathbf{c}_t$
— an *h*-dimensional previous hidden state vector $\mathbf{h}_t$
(which may also be passed to an output layer)

Bi-LSTM Encoder w/ HMM/CRF Layer

Contextualized Representations



Embeddings from Language Models

Replace static embeddings (lexicon lookup) with context-dependent embeddings (produced by a deep language model)

=> Each token's representation is a function of the entire input sentence, computed by a deep (multi-layer) bidirectional language model

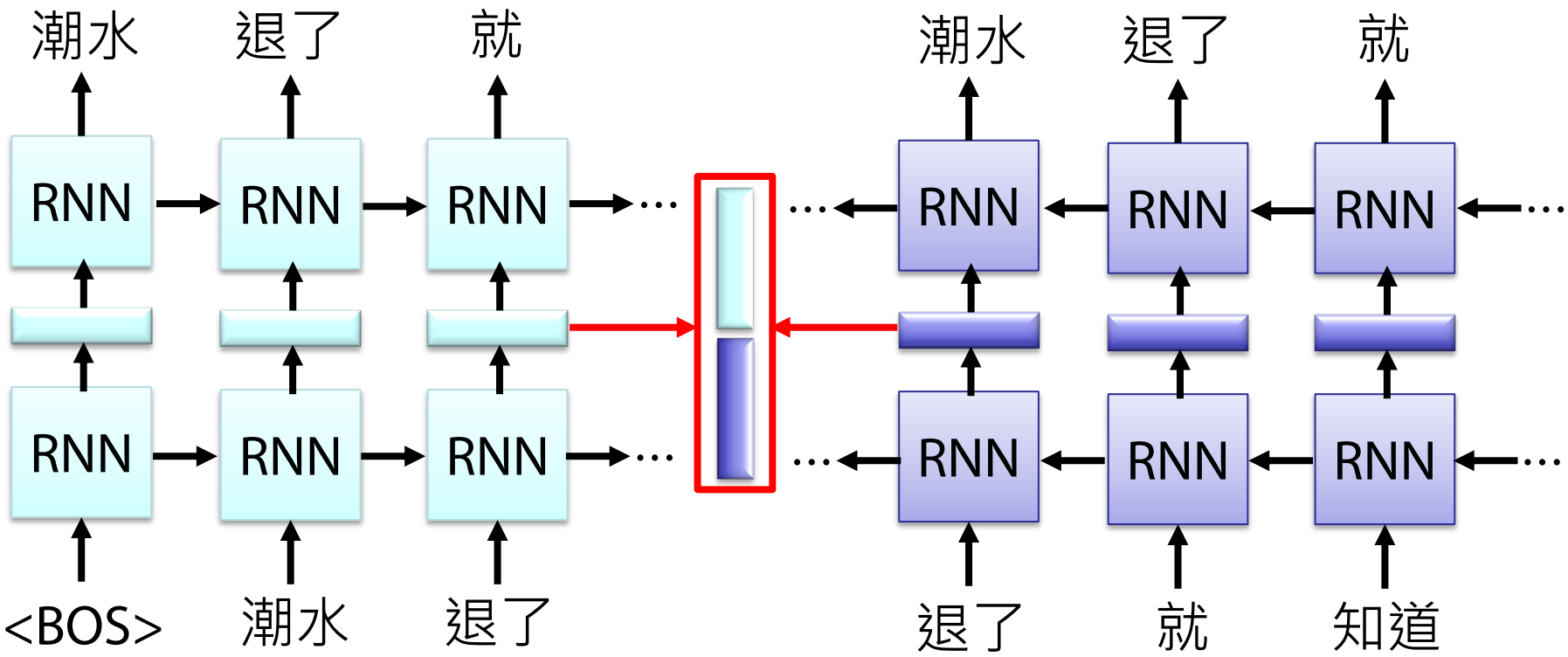
=> Return for each token a (task-dependent) linear combination of its representation across layers.

=> Different layers capture different information

Embeddings from Language Model (ELMO)

- RNN-based language models (trained from lots of sentences)

e.g., given "潮水 退了 就 知道 誰 沒穿 褲子"



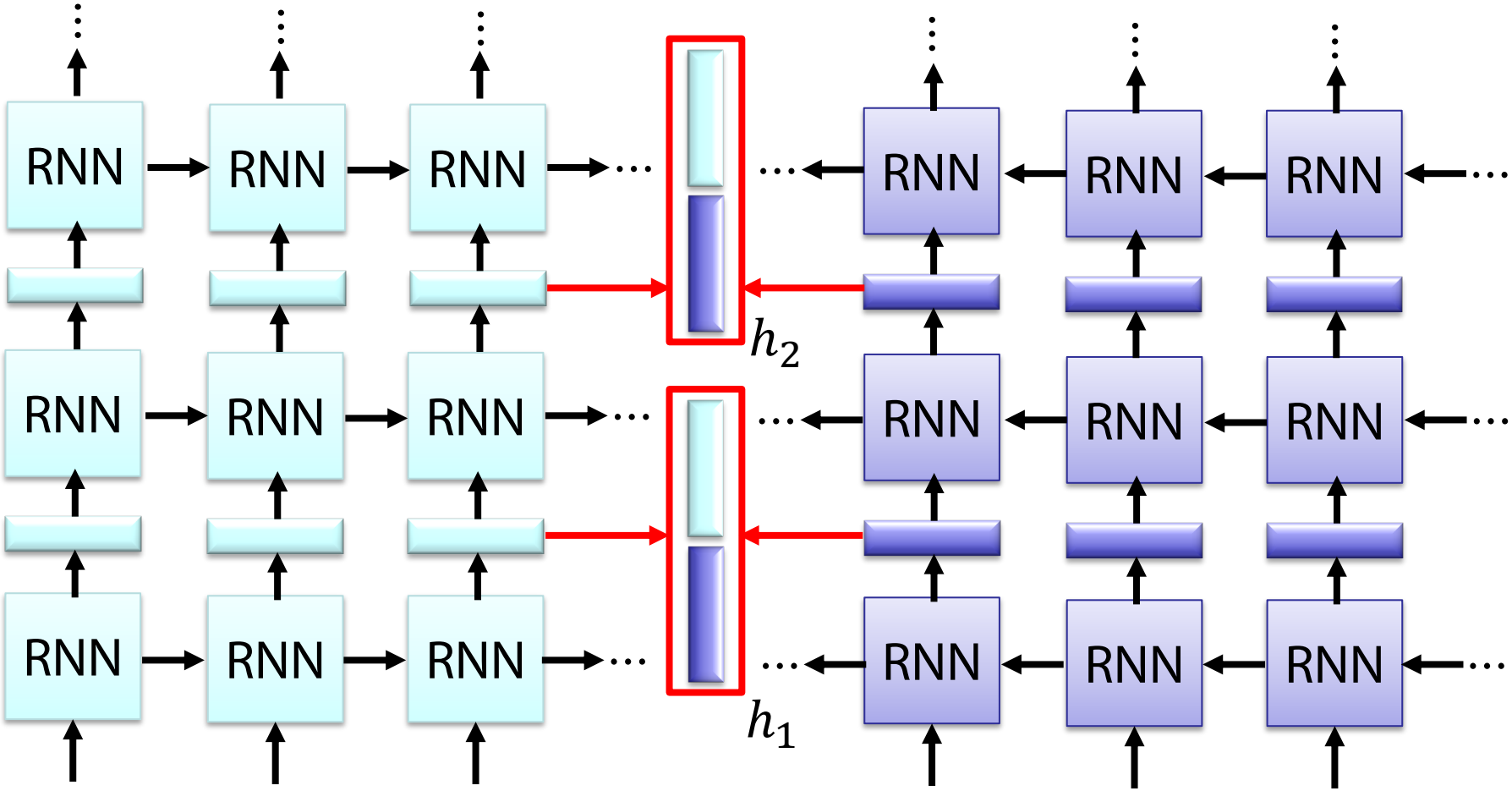
潮水 退了 就 知道 誰 沒穿 褲子 = When the tide goes out, you know who's not wearing pants.

https://arxiv.org/abs/1802.05365

ELMO

Each layer in deep LSTM can generate a latent representation.

Which one should we use???

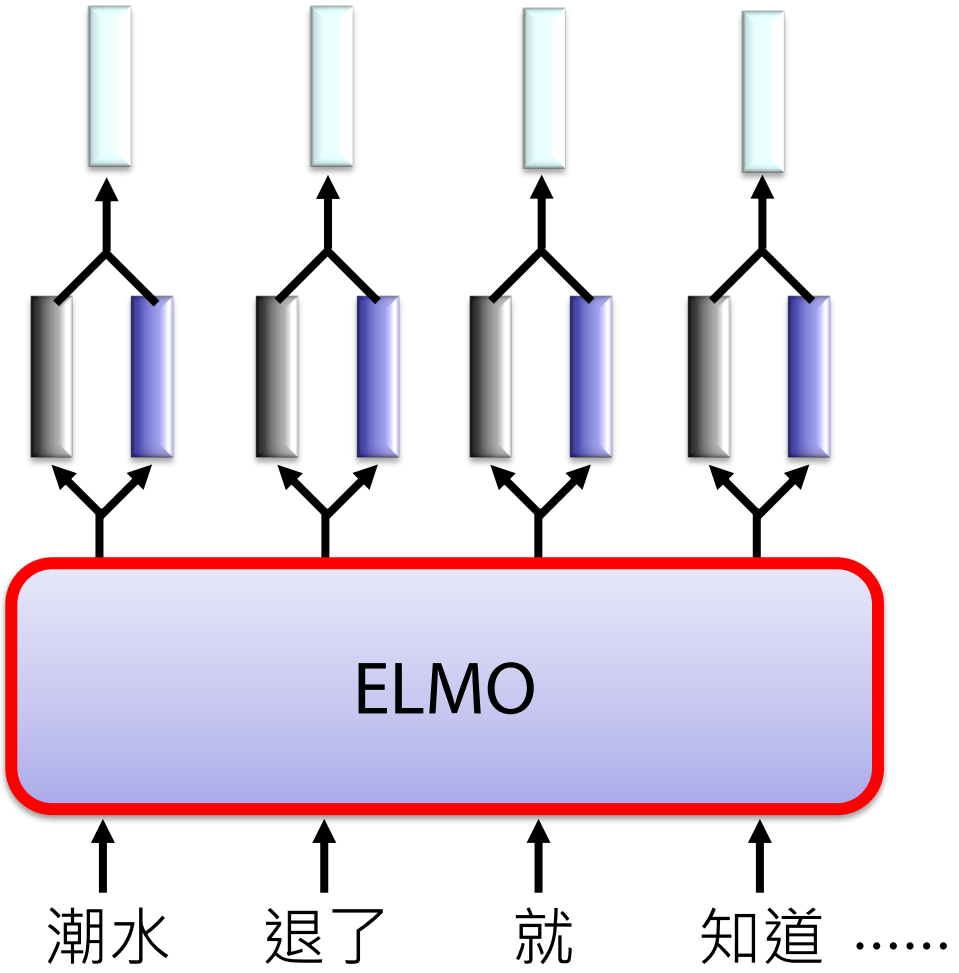


http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

ELMo architecture

- Train a multi-layer bidirectional language model with character convolutions on raw text
- Each layer of this language model network computes a vector representation for each token.
- Freeze the parameters of the language model.
- For each task: train task-dependent softmax weights to combine the layer-wise representations into a single vector for each token *jointly* with a task-specific model that uses those vectors

ELMO



$= \alpha_1 + \alpha_2$

Learned with the down stream tasks

ELMo's bidirectional language models

- The forward LM is a deep LSTM that goes over the sequence from start to end to predict token t_k based on the prefix $t_1 \ldots t_{k-1}$:

$$p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s)$$

- Parameters: token embeddings Θ_x LSTM $\overrightarrow{\Theta}_{LSTM}$ softmax Θ_s

- The backward LM is a deep LSTM that goes over the sequence from end to start to predict token t_k based on the suffix $t_{k+1} \ldots t_N$:

$$p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s)$$

- Train these LMs jointly, with the same parameters for the token representations and the softmax layer (but not for the LSTMs)

$$\sum_{k=1}^{N} \left(\log p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \right)$$

ELMo's token representations

The input token representations are purely character-based: a character CNN, followed by linear projection to reduce dimensionality

"2048 character n-gram convolutional filters with two highway layers, followed by a linear projection to 512 dimensions"

Advantage over using fixed embeddings:
no UNK tokens, any word can be represented

ELMo's token representations

Given a token representation $\mathbf{x}_k$, each layer j of the LSTM language models computes a vector representation $\mathbf{h}_{k,j}$ for every token k.

With L layers, ELMo represents each token as

$$\begin{aligned} R_k &= \{\mathbf{x}_k^{LM}, \overrightarrow{\mathbf{h}}_{k,j}^{LM}, \overleftarrow{\mathbf{h}}_{k,j}^{LM} \mid j = 1, \ldots, L\} \\ &= \{\mathbf{h}_{k,j}^{LM} \mid j = 0, \ldots, L\}, \end{aligned}$$

where $\mathbf{h}_{k,j}^{LM} = [\overrightarrow{\mathbf{h}}_{k,j}^{LM}; \overleftarrow{\mathbf{h}}_{k,j}^{LM}]$ and $\mathbf{h}_{k,0}^{LM} = \mathbf{x}_k$

ELMo learns softmax weights s_j^{task} to collapse these vectors into a single vector and a task-specific scalar γ^{task}

Learning objective:

$$\mathbf{ELMo}_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \mathbf{h}_{k,j}^{LM}.$$

Weighted Softmax

The softmax function is used in various multiclass classification methods, such as multinomial logistic regression (also known as softmax regression)[2]:206–209 [1], multiclass linear discriminant analysis, naive Bayes classifiers, and artificial neural networks.[6] Specifically, in multinomial logistic regression and linear discriminant analysis, the input to the function is the result of K distinct linear functions, and the predicted probability for the j'th class given a sample vector $\mathbf{x}$ and a weighting vector $\mathbf{w}$ is:

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

This can be seen as the composition of K linear functions $\mathbf{x} \mapsto \mathbf{x}^\mathsf{T}\mathbf{w}_1, \ldots, \mathbf{x} \mapsto \mathbf{x}^\mathsf{T}\mathbf{w}_K$ and the softmax function (where $\mathbf{x}^\mathsf{T}\mathbf{w}$ denotes the inner product of $\mathbf{x}$ and $\mathbf{w}$). The operation is equivalent to applying a linear operator defined by $\mathbf{w}$ to vectors $\mathbf{x}$, thus transforming the original, probably highly-dimensional, input to vectors in a K-dimensional space $\mathbb{R}^K$.

[Wikipedia]

How do you use ELMo?

ELMo embeddings can be used as (additional) input to any further language model

- ELMo can be tuned with dropout and L2-regularization (so that all layer weights stay close to each other)
- It often helps to fine-tune the biLMs (train them further) on task-specific raw text

In general: concatenate $\mathbf{ELMo}_k^{task}$ with other embeddings $\mathbf{x}_k$ for token input

Results

ELMo gave improvements on a variety of tasks:

— question answering (SQuAD)
— entailment/natural language inference (SNLI)
— semantic role labeling (SRL)
— coreference resolution (Coref)
— named entity recognition (NER)
— sentiment analysis (SST-5)

TASK	PREVIOUS SOTA		OUR BASELINE	ELMO + BASELINE	INCREASE (ABSOLUTE/ RELATIVE)
SQuAD	Liu et al. (2017)	84.4	81.1	85.8	4.7 / 24.9%
SNLI	Chen et al. (2017)	88.6	88.0	88.7 ± 0.17	0.7 / 5.8%
SRL	He et al. (2017)	81.7	81.4	84.6	3.2 / 17.2%
Coref	Lee et al. (2017)	67.2	67.2	70.4	3.2 / 9.8%
NER	Peters et al. (2017)	91.93 ± 0.19	90.15	92.22 ± 0.10	2.06 / 21%
SST-5	McCann et al. (2017)	53.7	51.4	54.7 ± 0.5	3.3 / 6.8%

Using ELMo at input vs output

Task	Input Only	Input & Output	Output Only
SQuAD	85.1	**85.6**	84.8
SNLI	88.9	**89.5**	88.7
SRL	**84.7**	84.3	80.9

Table 3: Development set performance for SQuAD, SNLI and SRL when including ELMo at different locations in the supervised model.

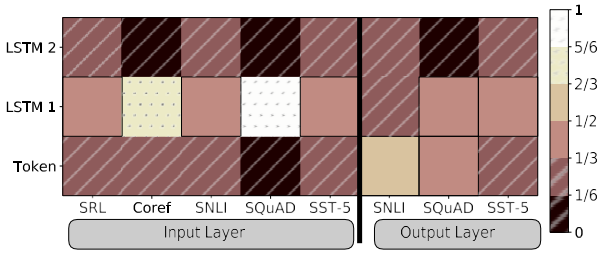


Figure 2: Visualization of softmax normalized biLM layer weights across tasks and ELMo locations. Normalized weights less then 1/3 are hatched with horizontal lines and those greater then 2/3 are speckled.

The supervised models for question-answering, entailment and SRL all use sequence architectures.

— We can concatenate ELMo to the input and/or the output of that network (with different layer weights)

—> Input always helps, Input+output often helps

—> Layer weights differ for each task

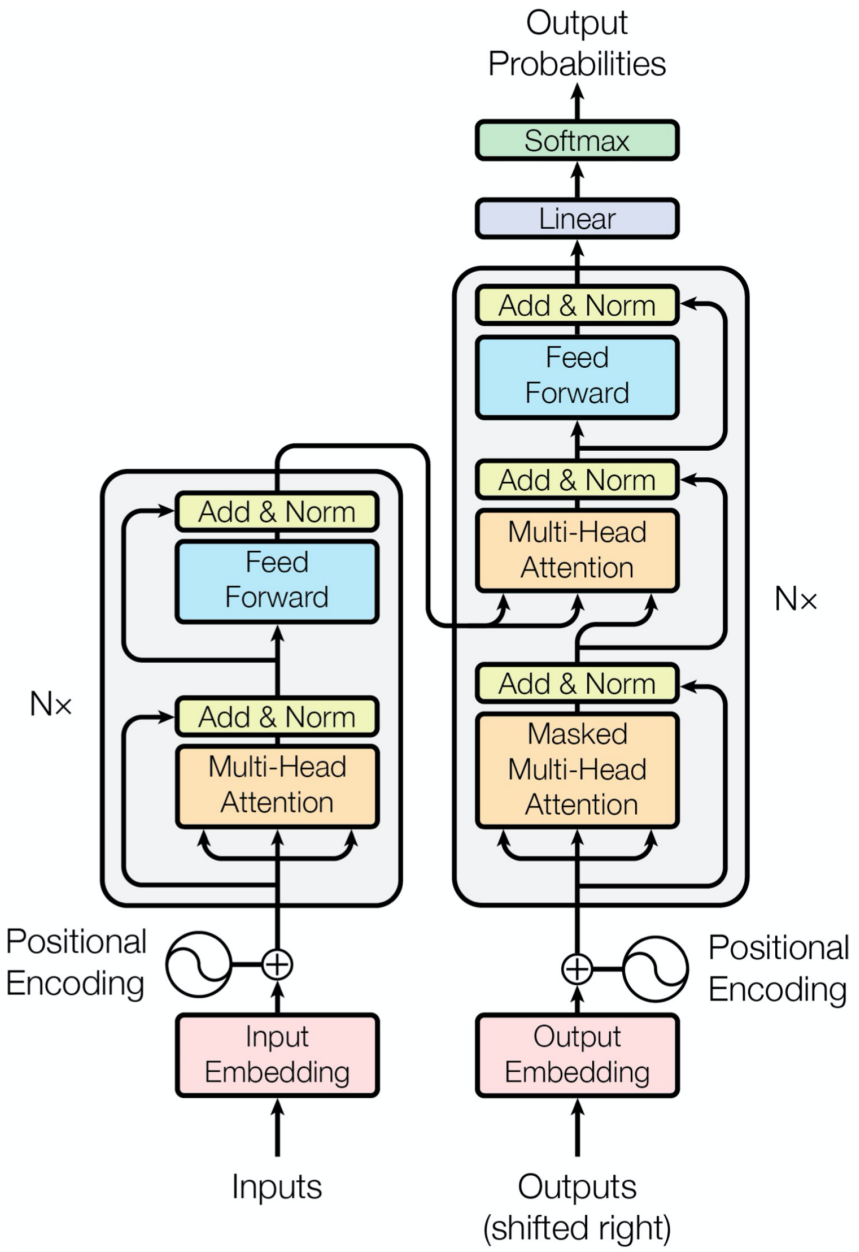
Transformers

Sequence transduction model based on attention (no convolutions or recurrence)

- easier to parallelize than recurrent nets
- faster to train than recurrent nets
- captures more long-range dependencies than CNNs with fewer parameters

Transformers use stacked self-attention and pointwise, fully-connected layers for the encoder and decoder

Transformer Architecture

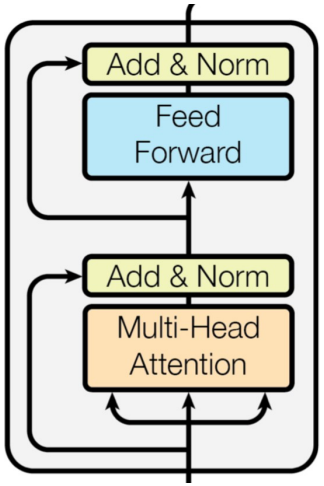


Encoder

A stack of N=6 identical layers
All layers and sublayers are 512-dimensional

Each layer consists of two sublayers
— one multi-headed self attention layer
— one position-wise fully connected layer

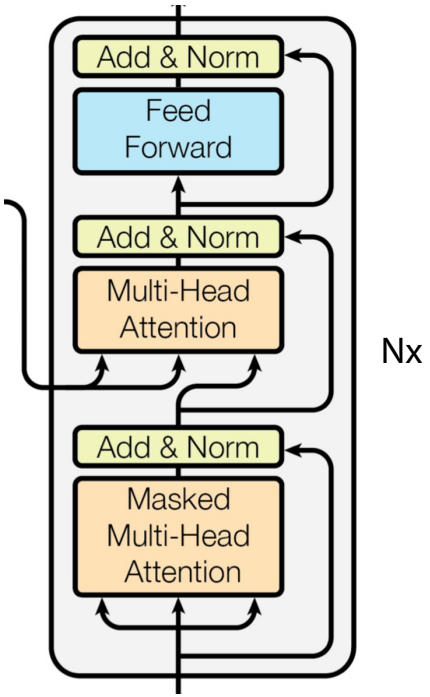
Each sublayer has a residual connection and is normalized:
LayerNorm(x + Sublayer(x))



Decoder

A stack of N=6 identical layers
All layers and sublayers are 512-d

Each layer consists of **three** sublayers
- one multi-headed self attention layer over decoder output (ignoring future tokens)
- one multi-headed attention layer over encoder output
- one position-wise fully connected layer



Each sublayer has a residual connection and is normalized:
LayerNorm(x + Sublayer(x))

Attention mechanisms

Compute a probability distribution $\alpha = (\alpha_{1t}, \ldots, \alpha_{St})$ over the *encoder's* hidden states $\mathbf{h}^{(s)}$ that depends on the *decoder's* current $\mathbf{h}^{(t)}$

$$\alpha_{ts} = \frac{\exp(s(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}))}{\sum_{s'} \exp(s(\mathbf{h}^{(t)}, \mathbf{h}^{(s')}))}$$

Compute a weighted avg. of the *encoder's* $\mathbf{h}^{(s)}$: $\mathbf{c}^{(t)} = \sum_{s=1..S} \alpha_{ts}\mathbf{h}^{(s)}$

that gets then used with $\mathbf{h}^{(t)}$, e.g. in $\mathbf{o}^{(t)} = \tanh(W_1\mathbf{h}^{(t)} + W_2\mathbf{c}^{(t)})$

- **Hard attention** (degenerate case, non-differentiable):
  is a one-hot vector

- **Soft attention** (general case): is not a one-hot
  - $s(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}) = \mathbf{h}^{(t)} \cdot \mathbf{h}^{(s)}$ is the dot product (no learned parameters)
  - $s(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}) = (\mathbf{h}^{(t)})^T W \mathbf{h}^{(s)}$ (learn a bilinear matrix W)
  - $s(\mathbf{h}^{(t)}, \mathbf{h}^{(s)}) = \mathbf{v}^T \tanh(W_1\mathbf{h}^{(t)} + W_2\mathbf{h}^{(s)})$ concat. hidden states

Self-Attention

Attention so far (in seq2seq architectures):

In the *decoder* (which has access to the complete input sequence), compute attention weights over *encoder* positions that depend on each decoder position

Self-attention:

If the *encoder* has access to the complete input sequence, we can also compute attention weights over *encoder* positions that depend on each *encoder* position

self-attention:
encoder

For each ~~*decoder*~~ position t,

compute an attention weight for each *encoder* position s
renormalize these weights (that depend on t) w/ softmax
to obtain a new weighted avg. of the input sequence vectors

Self-attention

Given T k-dimensional *input* vectors $\mathbf{x}^{(1)}\ldots\mathbf{x}^{(i)}\ldots\mathbf{x}^{(T)}$,
compute T k-dimensional *output* vectors $\mathbf{y}^{(1)}\ldots\mathbf{y}^{(i)}\ldots\mathbf{y}^{(T)}$
where each $\mathbf{y}^{(i)}$ is a weighted average of the input vectors, and
where the weights w_{ij} depend on $\mathbf{y}^{(i)}$ and $\mathbf{x}^{(j)}$

$$\mathbf{y}^{(i)} = \sum_{j=1..T} w_{ij}\mathbf{x}^{(j)}$$

Computing weights w_{ij} naively:
use dot product: $w'_{ij} = \sum_k x_k^{(i)} x_k^{(j)}$

followed by softmax: $w_{ij} = \frac{\exp(w_i{}'_j)}{\sum_j \exp(w'_{ij})}$

Queries, keys, values

- Let's add learnable parameters (kxk weight matrices), and turn each vector $\mathbf{x}^{(i)}$ into three versions:
  - **Query** vector $\mathbf{q}^{(i)} = \mathbf{W}_q \mathbf{x}^{(i)}$
  - **Key** vector: $\mathbf{k}^{(i)} = \mathbf{W}_k \mathbf{x}^{(i)}$
  - **Value** vector: $\mathbf{v}^{(i)} = \mathbf{W}_v \mathbf{x}^{(i)}$

- The **attention weight of the *j*-th position** to compute the **new output or the *i*-th position** depends on the **query of i** and the **key of j**:

$$w_j^{(i)} = \frac{\exp(\mathbf{q}^{(i)}\mathbf{k}^{(j)})}{\sum_j \exp(\mathbf{q}^{(i)}\mathbf{k}^{(j)})} = \frac{\exp(\sum_l q_l^{(i)} k_l^{(j)})}{\sum_j \exp(\sum_l q_l^{(i)} k_l^{(j)})}$$

The **new output vector for the i-th position** depends on the **attention weights** and **value** vectors of all **input positions j**:

$$\mathbf{y}^{(i)} = \sum_{j=1..T} w_j^{(i)} \mathbf{v}^{(j)}$$

Scaling attention weights

Value of dot product grows with vector dimension k

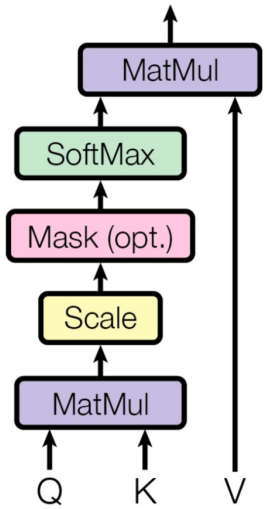
To scale back the dot product, divide by $\sqrt{k}$:

$$w_j^{(i)} = \frac{\exp(\mathbf{q}^{(i)}\mathbf{k}^{(j)})/\sqrt{(k)}}{\sum_j (\exp(\mathbf{q}^{(i)}\mathbf{k}^{(j)})/\sqrt{k})}$$

Scaled Dot-Product Attention

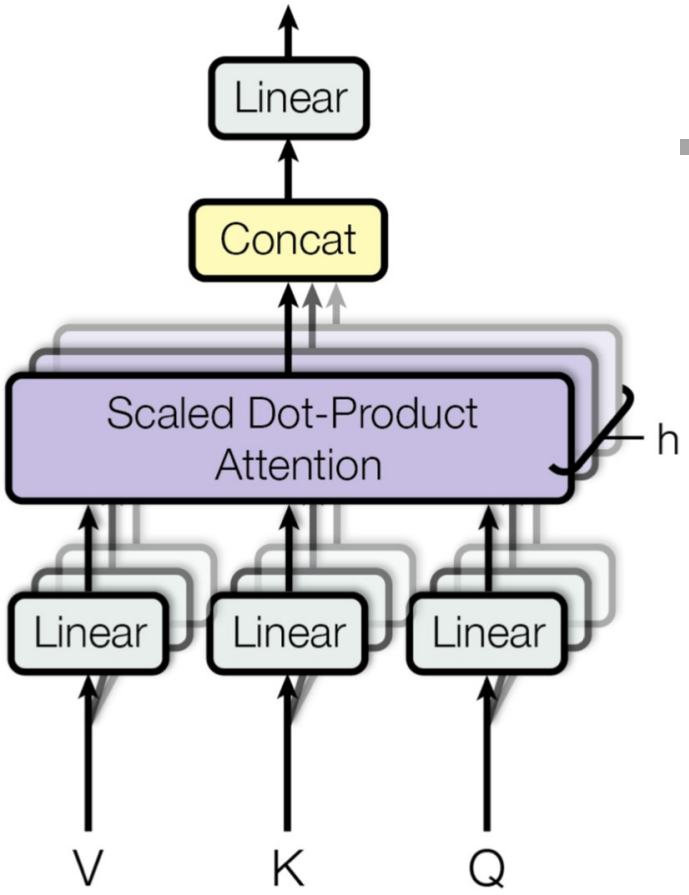
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention



Multi-Head attention

- Learn *h* different linear projections of Q,K,V
- Compute attention separately on each of these *h* versions
- Concatenate and project the resultant vectors to a lower dimensionality.
- Each attention head can use low dimensionality



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Position-wise feedforward nets

We train a feedforward net for each layer that only reads in input for its token
(two linear transformations with ReLU in between)

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Input and output: 512 dimensions
Internal layer: 2048 dimensions

Parameters differ from layer to layer
(but are shared across positions)
(cf. 1x1 convolutions)

Positional Encoding

How does this model capture sequence order?

Positional embeddings have the same dimensionality as word embeddings (512) and are added in.

Fixed representations: each dimension is a sinuoid (a sine or cosine function with a different frequency)

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

A word can have multiple senses.

Have you paid that money to the bank yet ?
It is safest to deposit your money in the bank .

The victim was found lying dead on the river bank .
They stood on the river bank to fish.

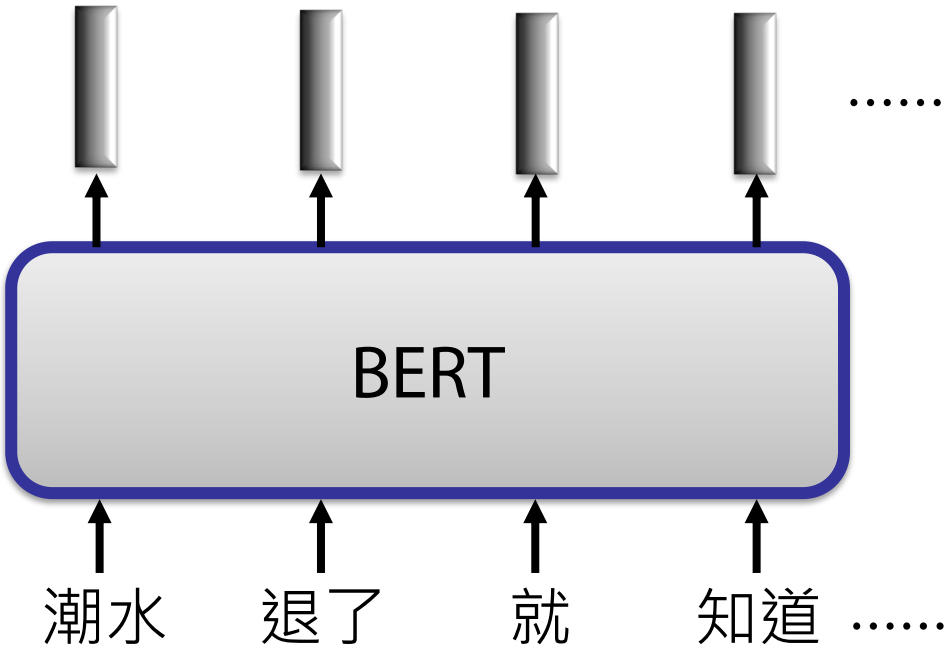
The hospital has its own blood bank.

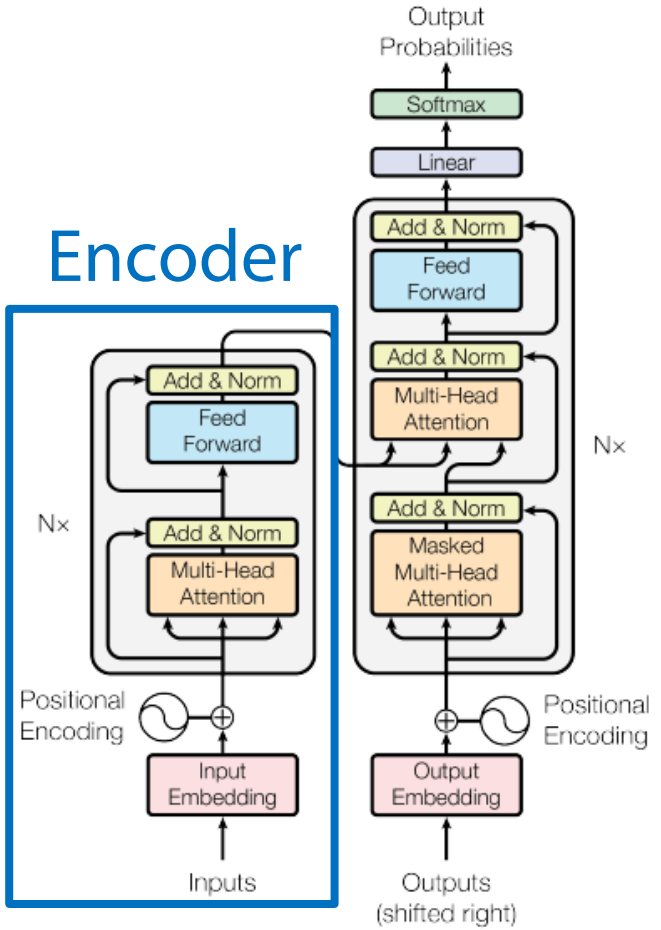
The third sense or not?

Bidirectional Encoder Representations from Transformers (BERT)

- BERT = Encoder of Transformer

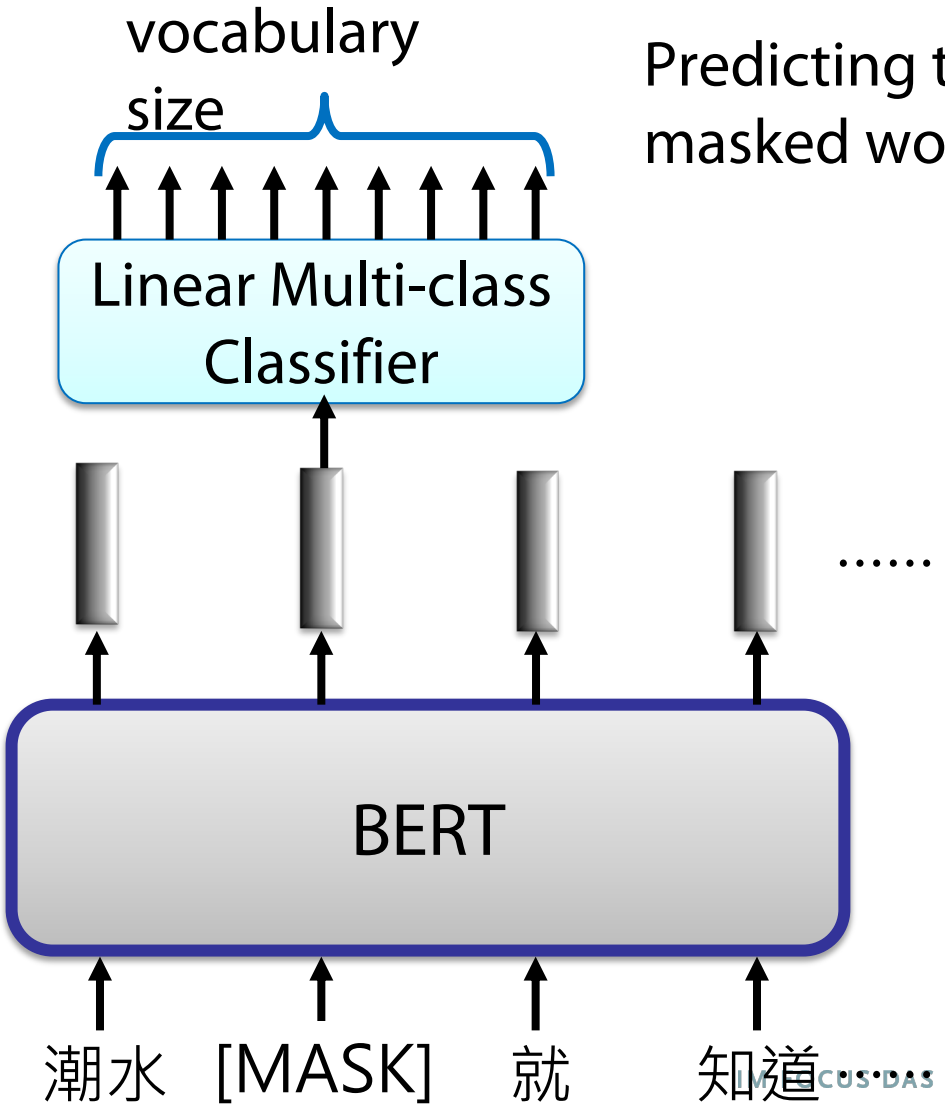
Learned from a large amount of text without annotation





Training of BERT

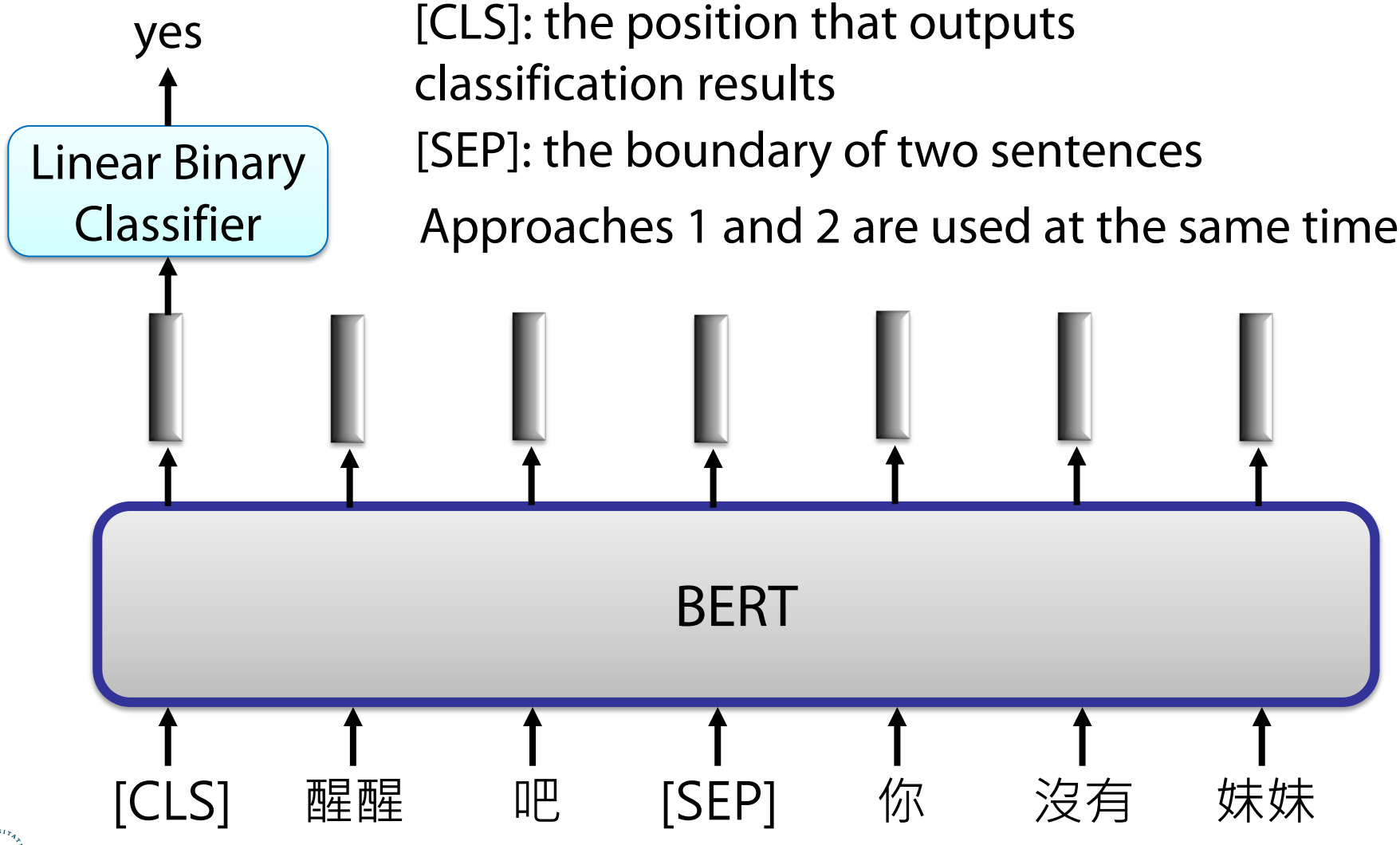
- Approach 1: Masked LM



http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

Training of BERT

Approach 2: Next Sentence Prediction



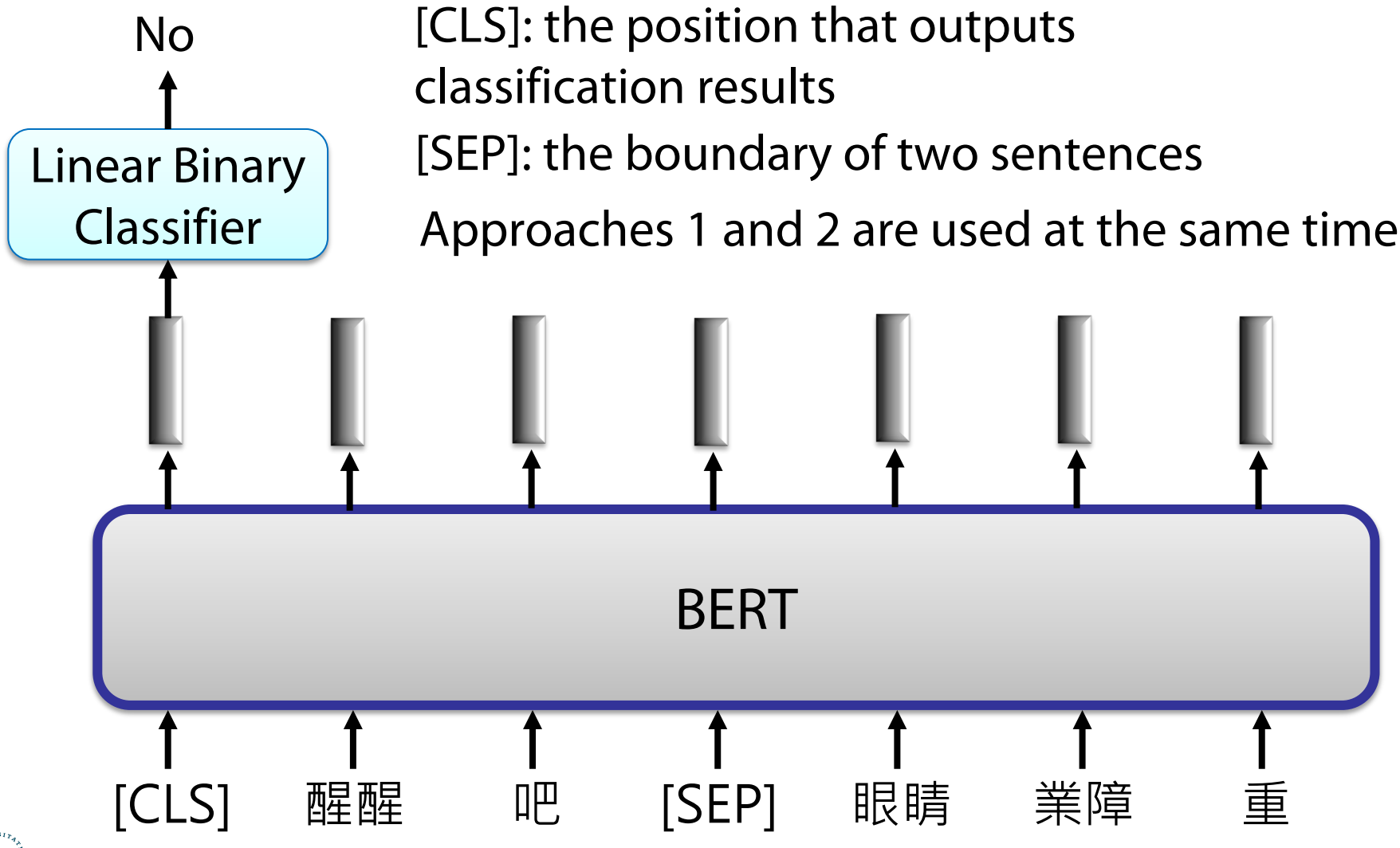
[CLS]: the position that outputs classification results

[SEP]: the boundary of two sentences

Approaches 1 and 2 are used at the same time.

Training of BERT

Approach 2: Next Sentence Prediction



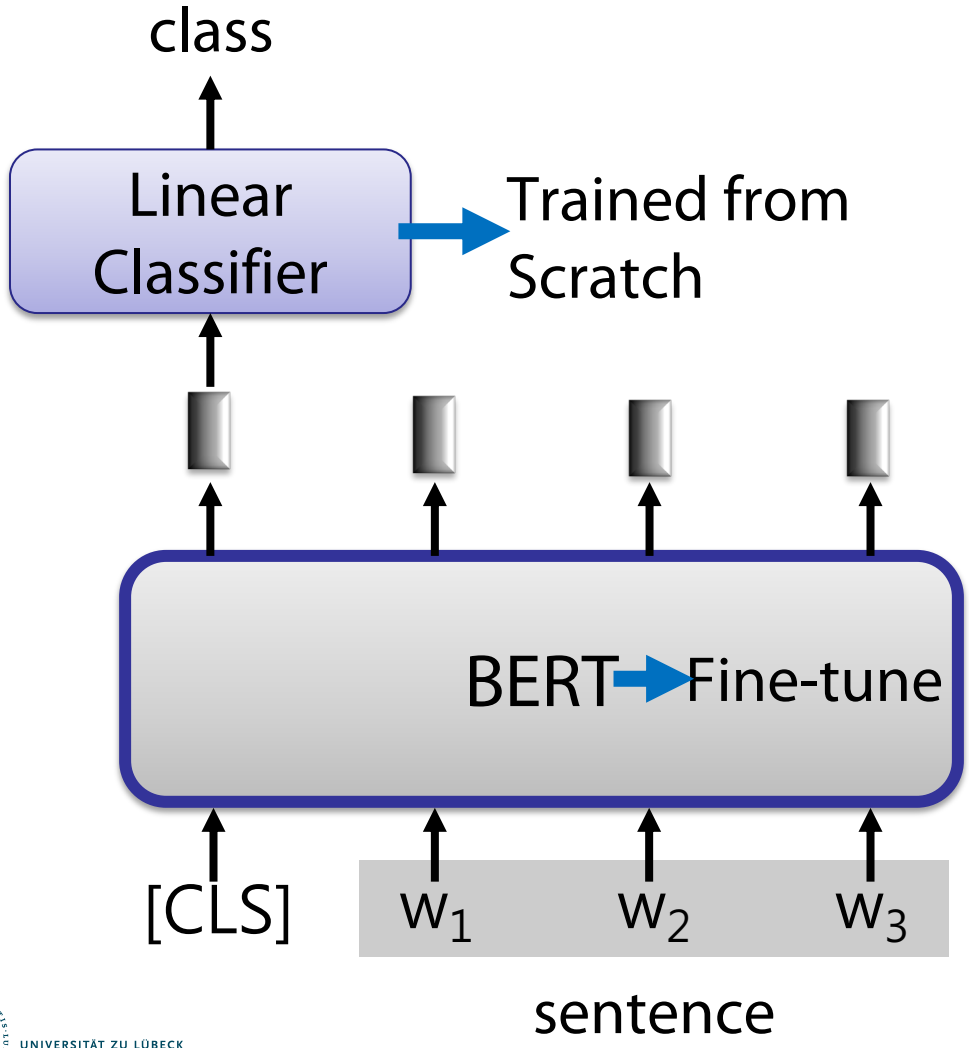
[CLS]: the position that outputs classification results

[SEP]: the boundary of two sentences

Approaches 1 and 2 are used at the same time.

http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

How to use BERT – Case 1

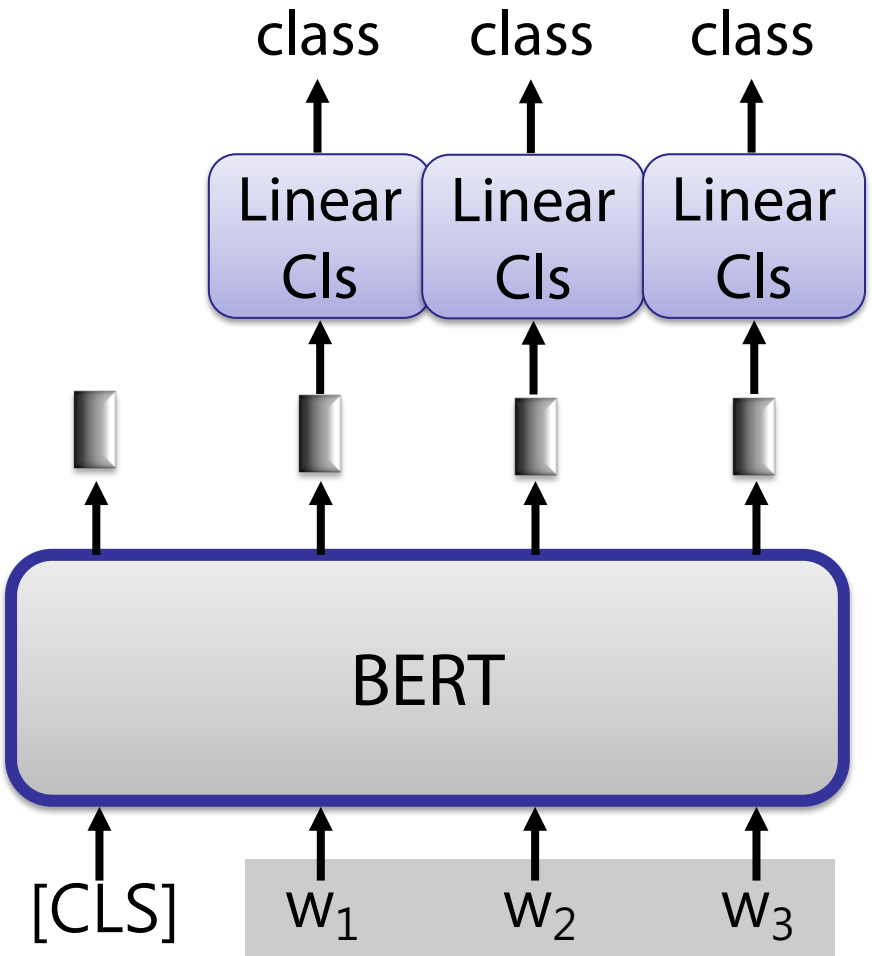


Input: single sentence, output: class

Example:
Sentiment analysis (our HW),
Document Classification

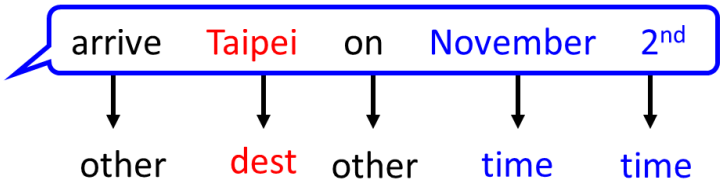
http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

How to use BERT – Case 2



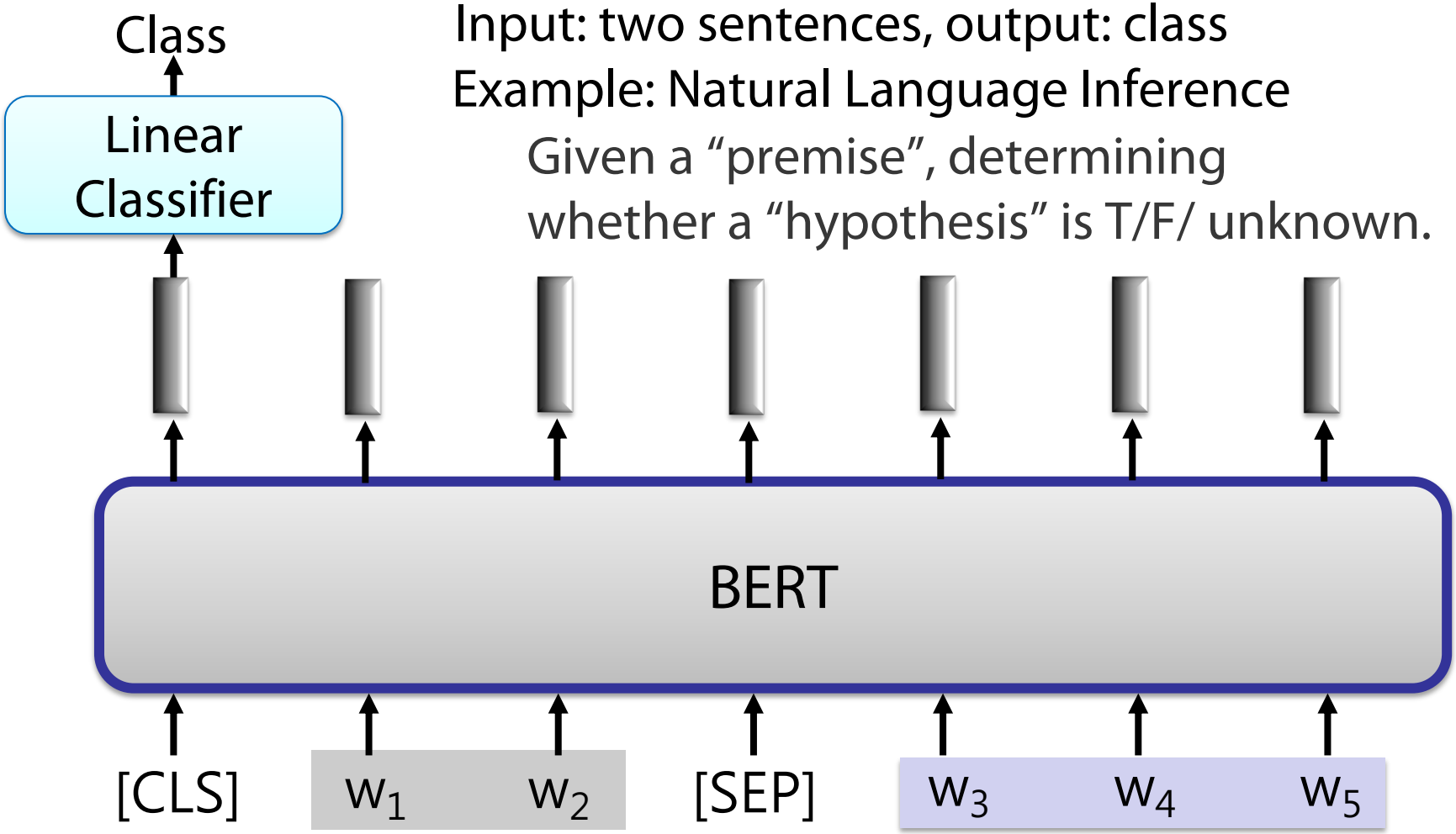
Input: single sentence, output: class of each word

Example: Slot filling



http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

How to use BERT – Case 3



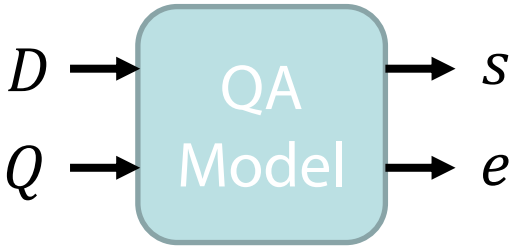
http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

How to use BERT – Case 4

- Extraction-based Question Answering (QA) (E.g. SQuAD)

Document: $D = \{d_1, d_2, \cdots, d_N\}$

Query: $Q = \{q_1, q_2, \cdots, q_N\}$



output: two integers (s, e)

Answer: $A = \{q_s, \cdots, q_e\}$

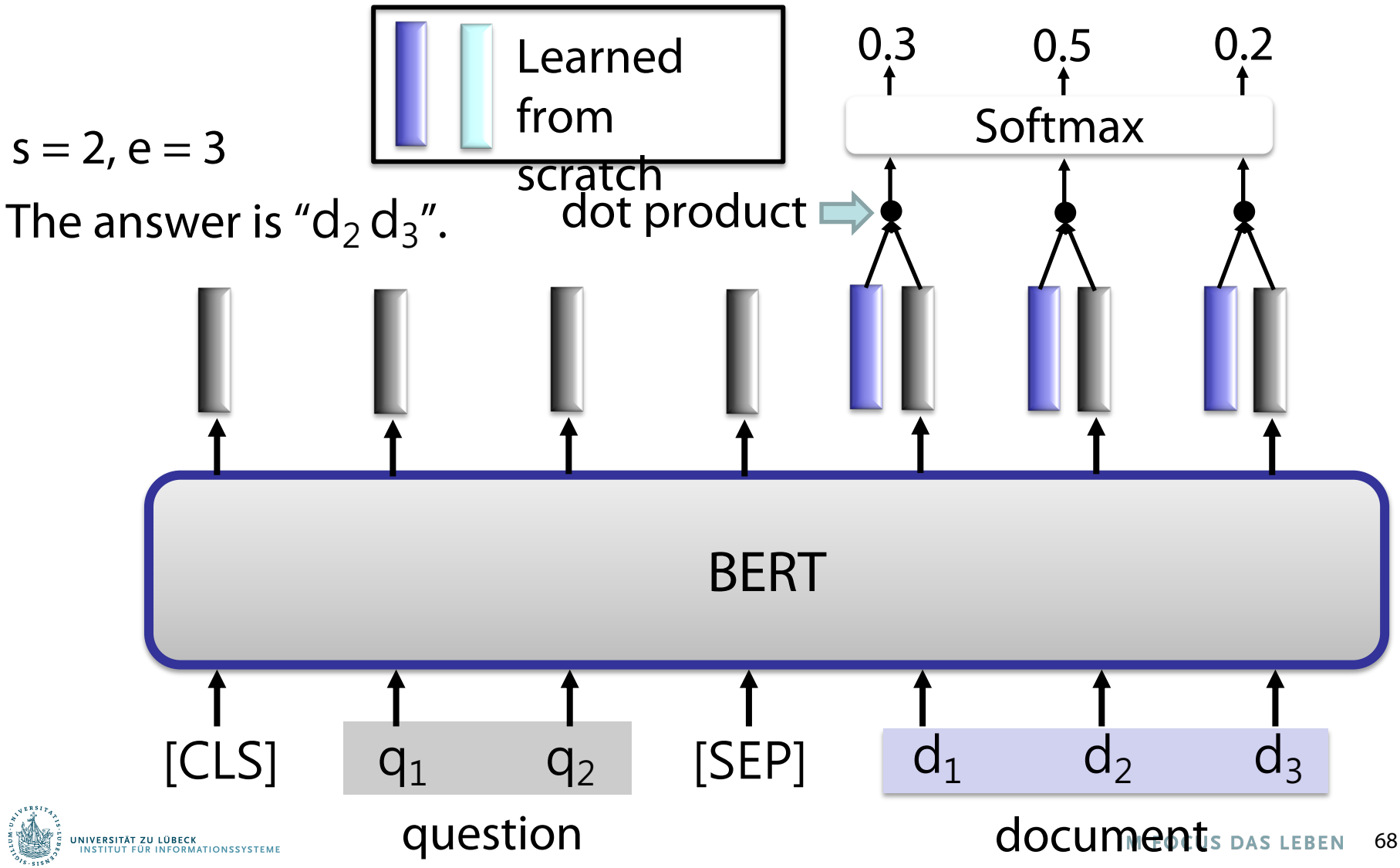
In meteorology, precipitation is any product of the condensation of 17 spheric water vapor that falls under gravity. The main forms of precipitation include drizzle, rain, sleet, snow, graupel and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals within a cloud. Short, intense periods of rain 77 atte 79 cations are called "showers".

What causes precipitation to fall?
gravity $s = 17, e = 17$

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?
graupel

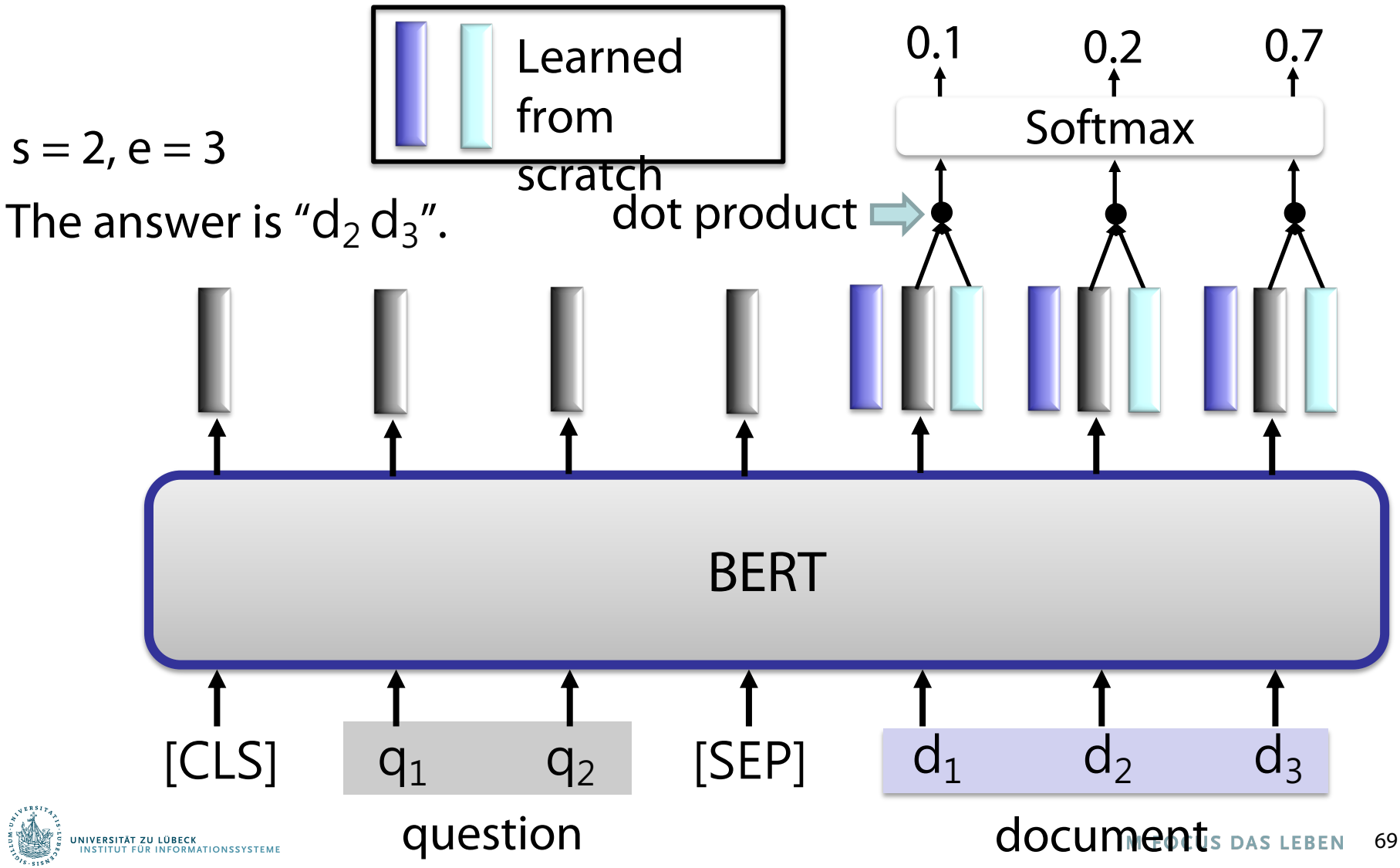
Where do water droplets collide with ice crystals to form precipitation?
within a cloud $s = 77, e = 79$

How to use BERT – Case 4



http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

How to use BERT – Case 4



http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

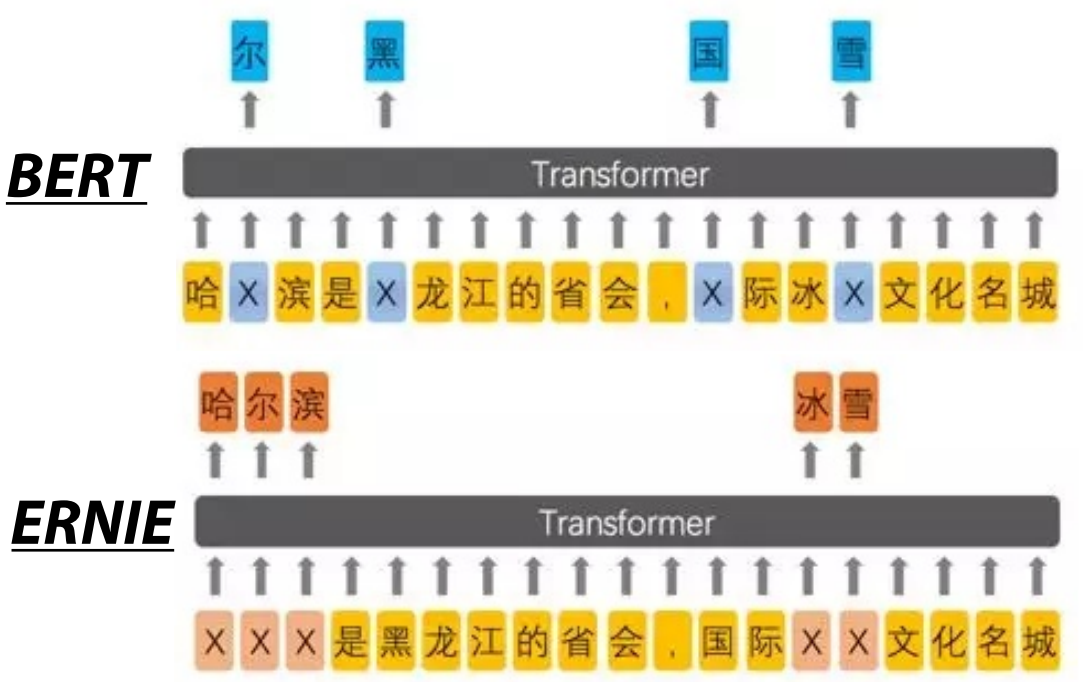
Rank	Model	EM	F1
	Human Performance *Stanford University* (Rajpurkar & Jia et al. '18)	86.831	89.452
1 Mar 20, 2019	BERT + DAE + AoA (ensemble) *Joint Laboratory of HIT and iFLYTEK Research*	**87.147**	**89.474**
2 Mar 15, 2019	BERT + ConvLSTM + MTL + Verifier (ensemble) *Layer 6 AI*	86.730	89.286
3 Mar 05, 2019	BERT + N-Gram Masking + Synthetic Self-Training (ensemble) *Google AI Language* https://github.com/google-research/bert	86.673	89.147
4 May 21, 2019	XLNet (single model) *XLNet Team*	86.346	89.133
5 Apr 13, 2019	SemBERT(ensemble) *Shanghai Jiao Tong University*	86.166	88.886

SQuAD 2.0

http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

Enhanced Representation through Knowledge Integration (ERNIE)

- Designed for Chinese



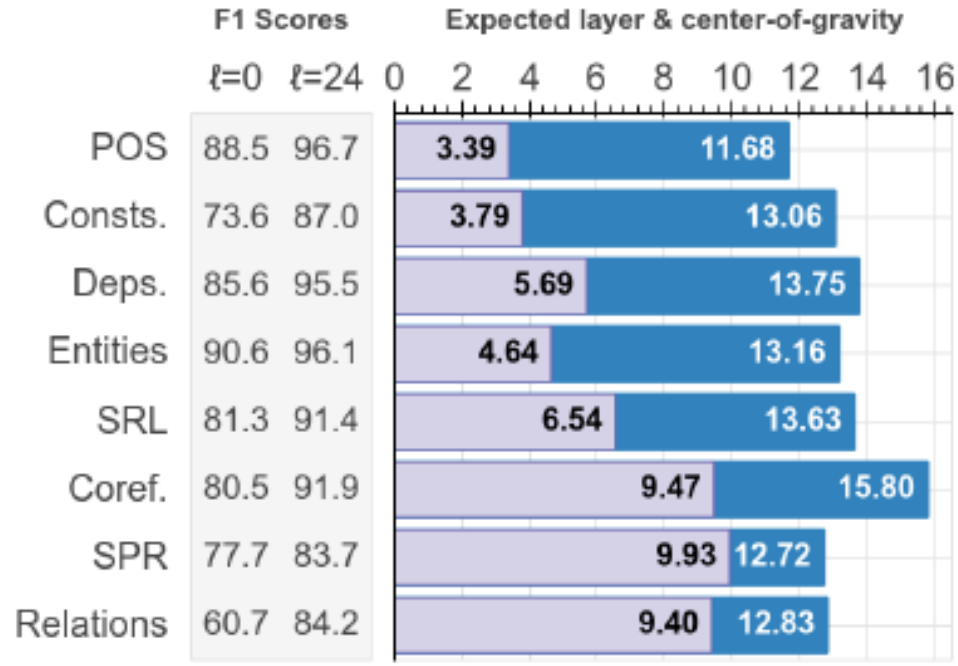
Source of image: https://zhuanlan.zhihu.com/p/59436589

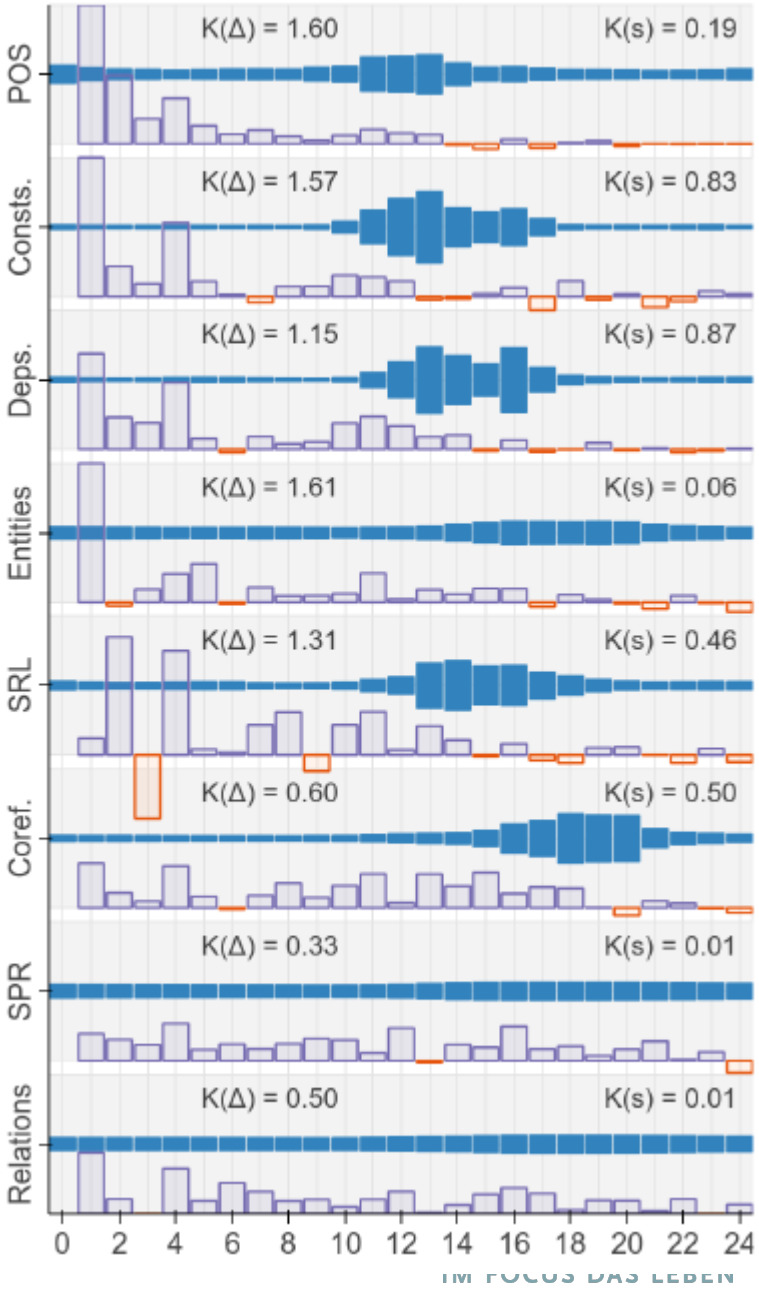
https://arxiv.org/abs/1904.09223

What does BERT learn?

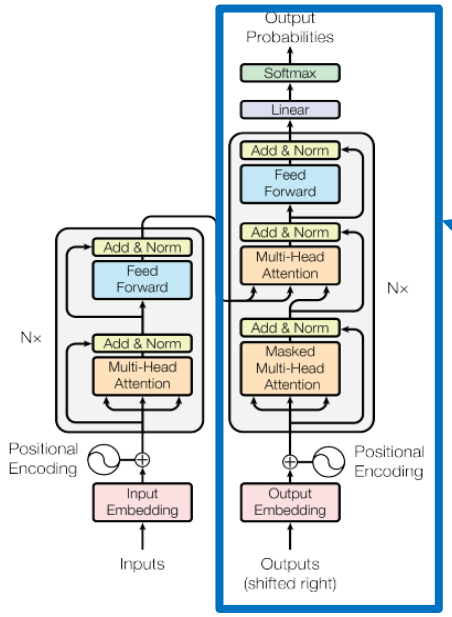
https://arxiv.org/abs/1905.05950

https://openreview.net/pdf?id=SJzSgnRcKX





Generative Pre-Training (GPT)



Transformer Decoder

BERT (340M)

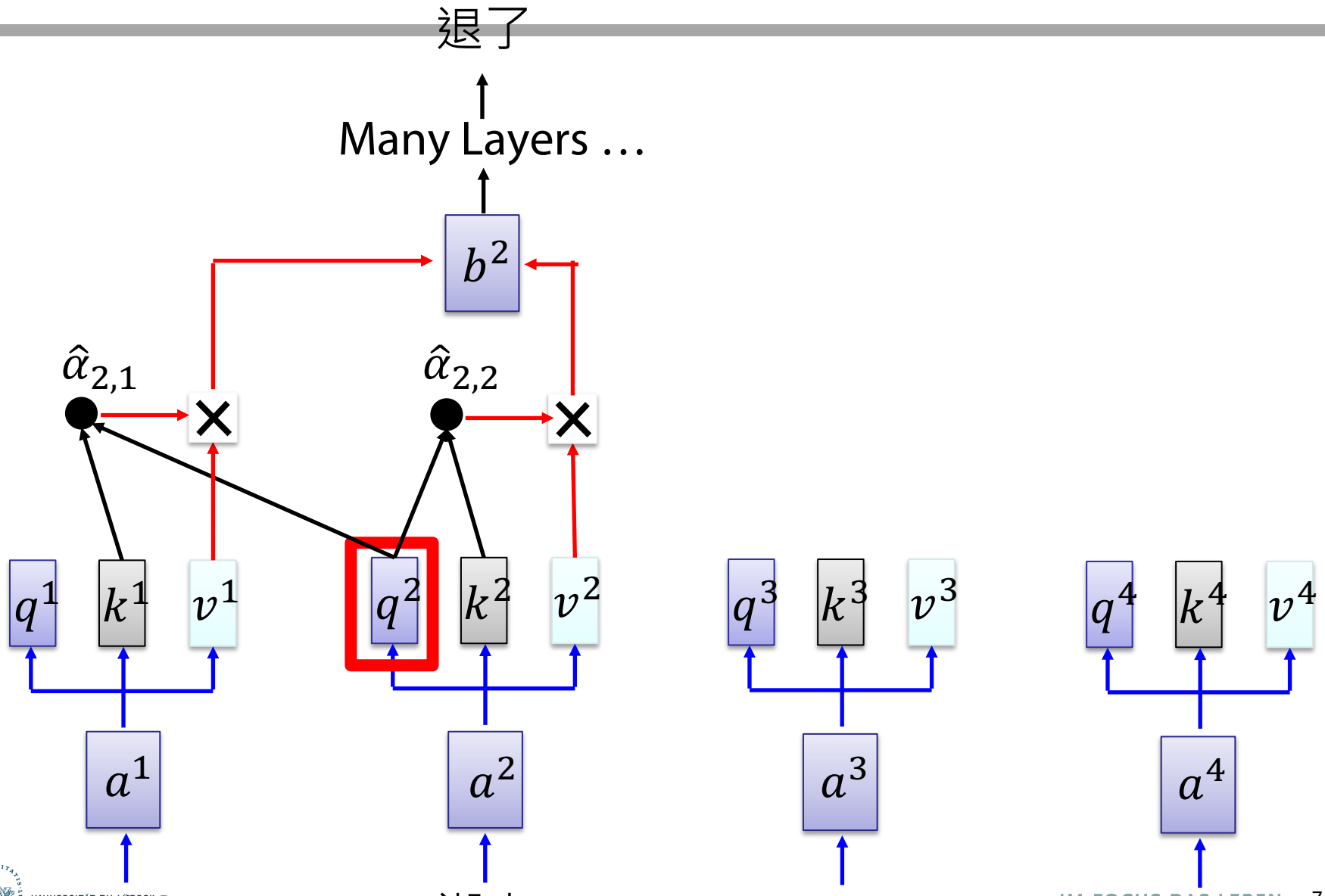
ELMO (94M)

GPT-2 (1542M)

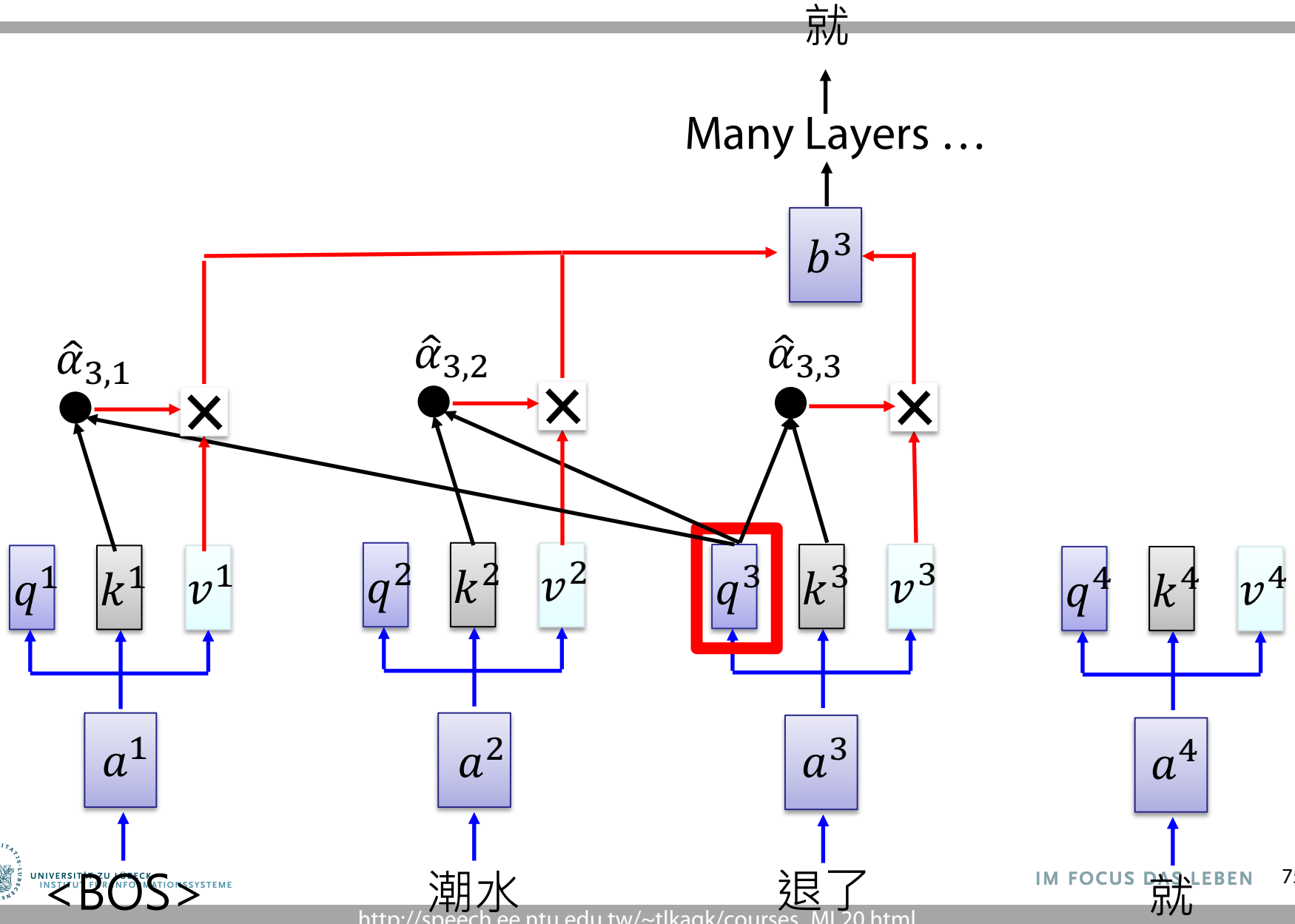
Source of image: https://huaban.com/pins/1714071707/

https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Generative Pre-Training (GPT)



Generative Pre-Training (GPT)

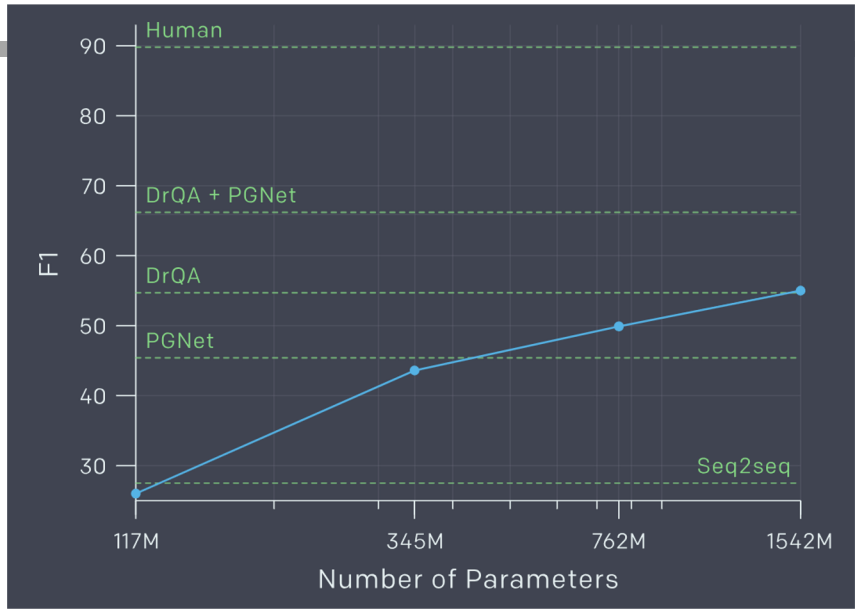


http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

Zero-shot Learning?

- ***Reading Comprehension***

$d_1, d_2, \cdots, d_N,$
"Q:", $q_1, q_2, \cdots, q_N,$
"A:"



- ***Summarization*** $d_1, d_2, \cdots, d_N$,"TL;DR:"

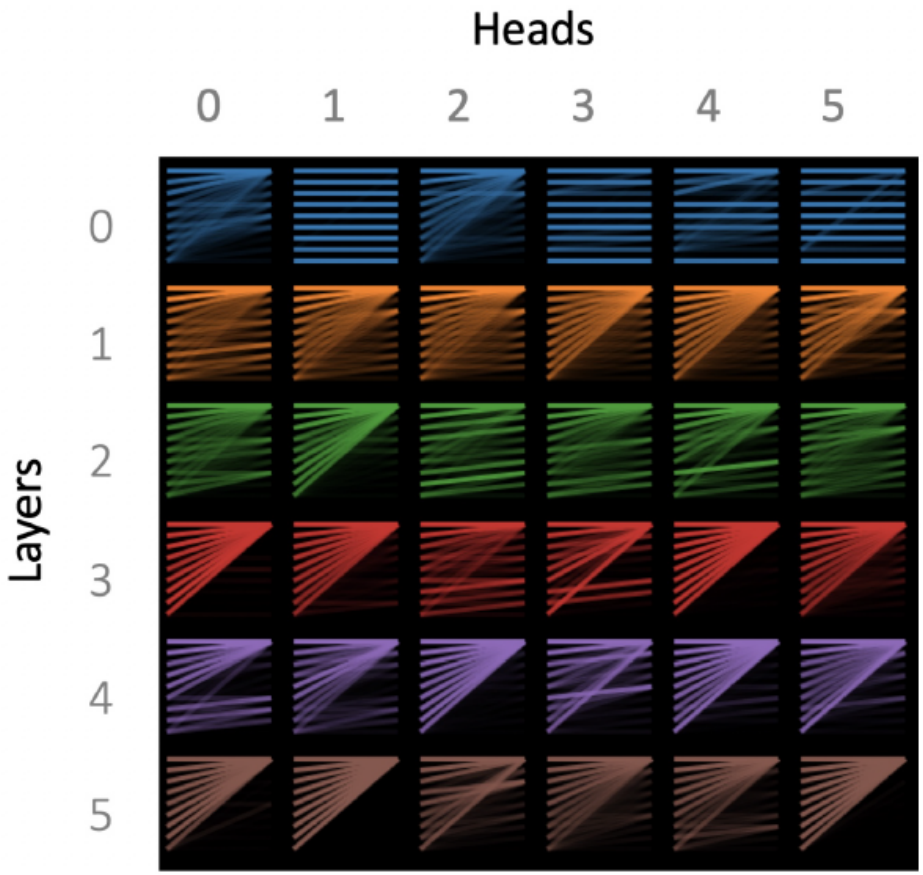
- ***Translation***

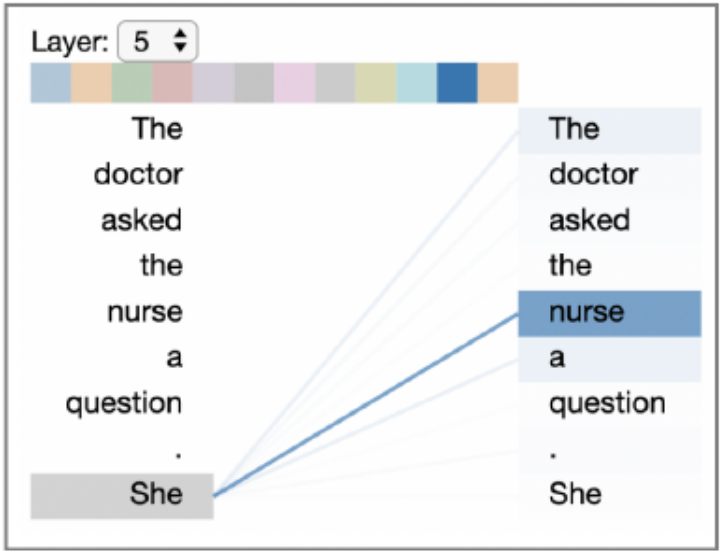
English sentence 1	=	French sentence 1
English sentence 2	=	French sentence 2
English sentence 3	=	

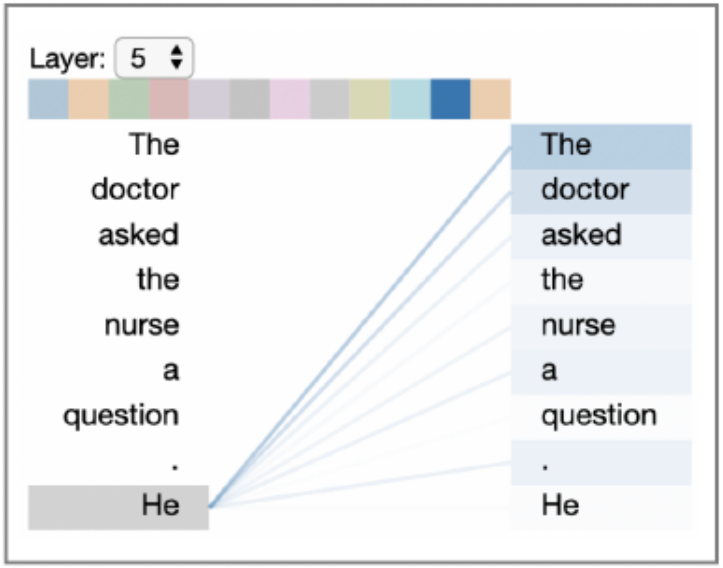
http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html

Visualization

https://arxiv.org/abs/1904.02679
(The results below are from GPT-2)







BERT as a Markov Random Field Language Model

- Show that BERT (Devlin et al., 2018) is a Markov random field language model
- Gives way to a natural procedure to sample sentences from BERT
  - Can produce high quality, fluent generations
  - Generates sentences that are more diverse but of slightly worse quality

Alex Wang, Kyunghyun Cho. BERT has a Mouth, and It Must Speak: BERT as a Markov Random Field Language Model. Volume:
In Proc. of the Workshop on Methods for Optimizing and Evaluating Neural Language Generation, June **2019**.
https://arxiv.org/abs/1902.04094