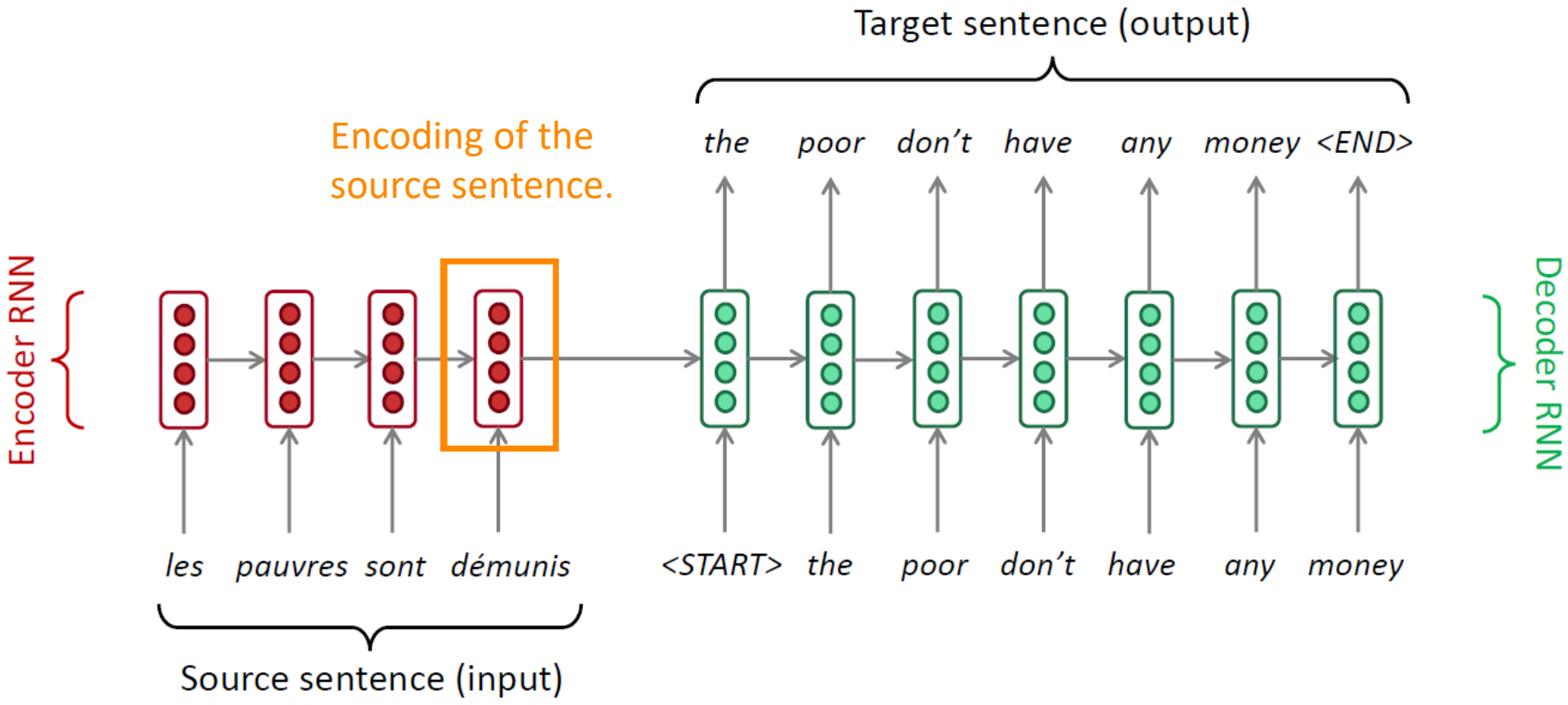
# COMP4901K/Math4824B
# Machine Learning for Natural Language Processing

## Lecture 16: Attention Models

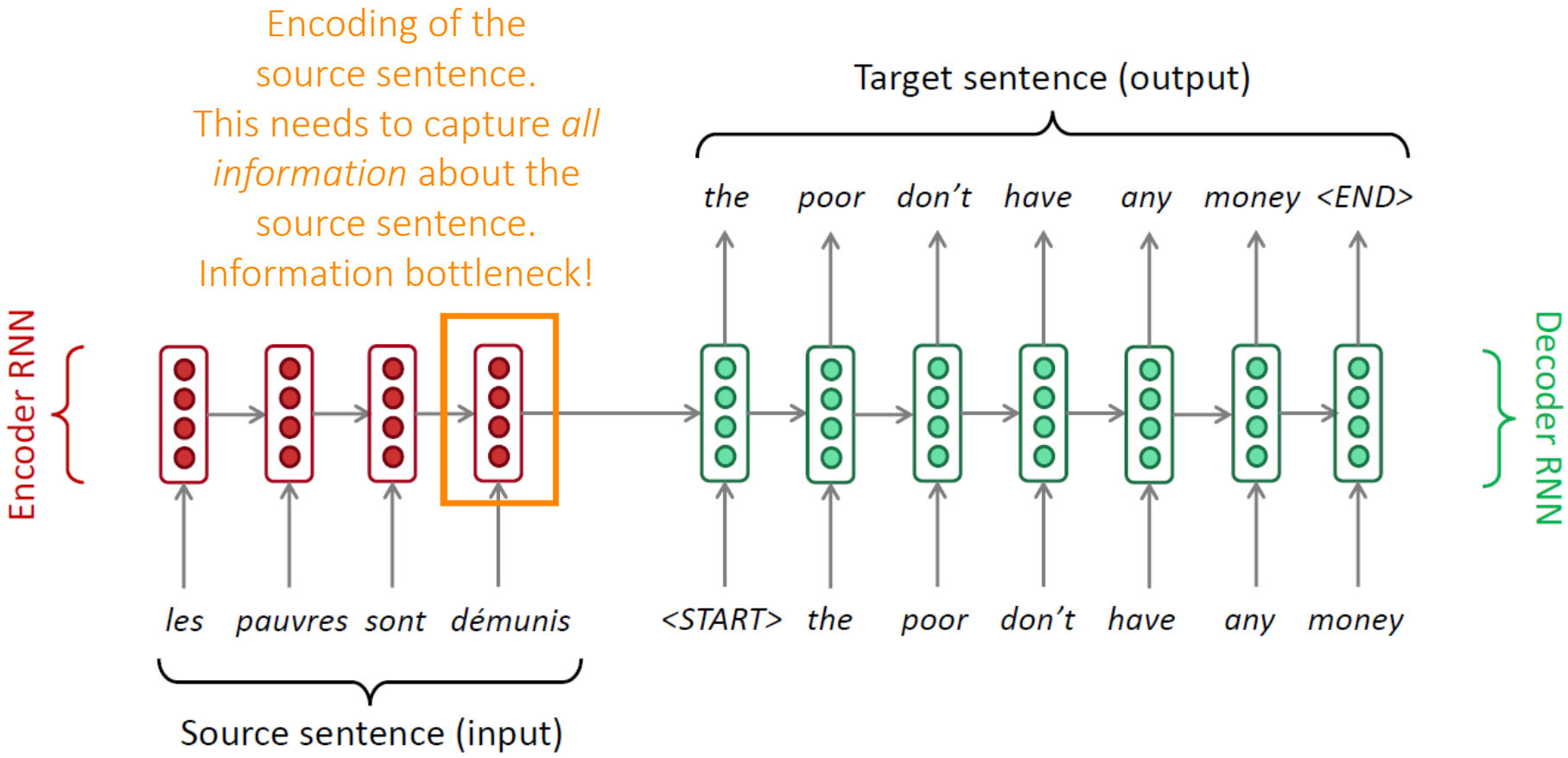
Instructor: Yangqiu Song

Slides credits: Richard Socher, Vagelis Hristidis, Abigail See, Elena Voita

# Sequence-to-sequence: the bottleneck problem



# Sequence-to-sequence: the bottleneck problem



# Attention

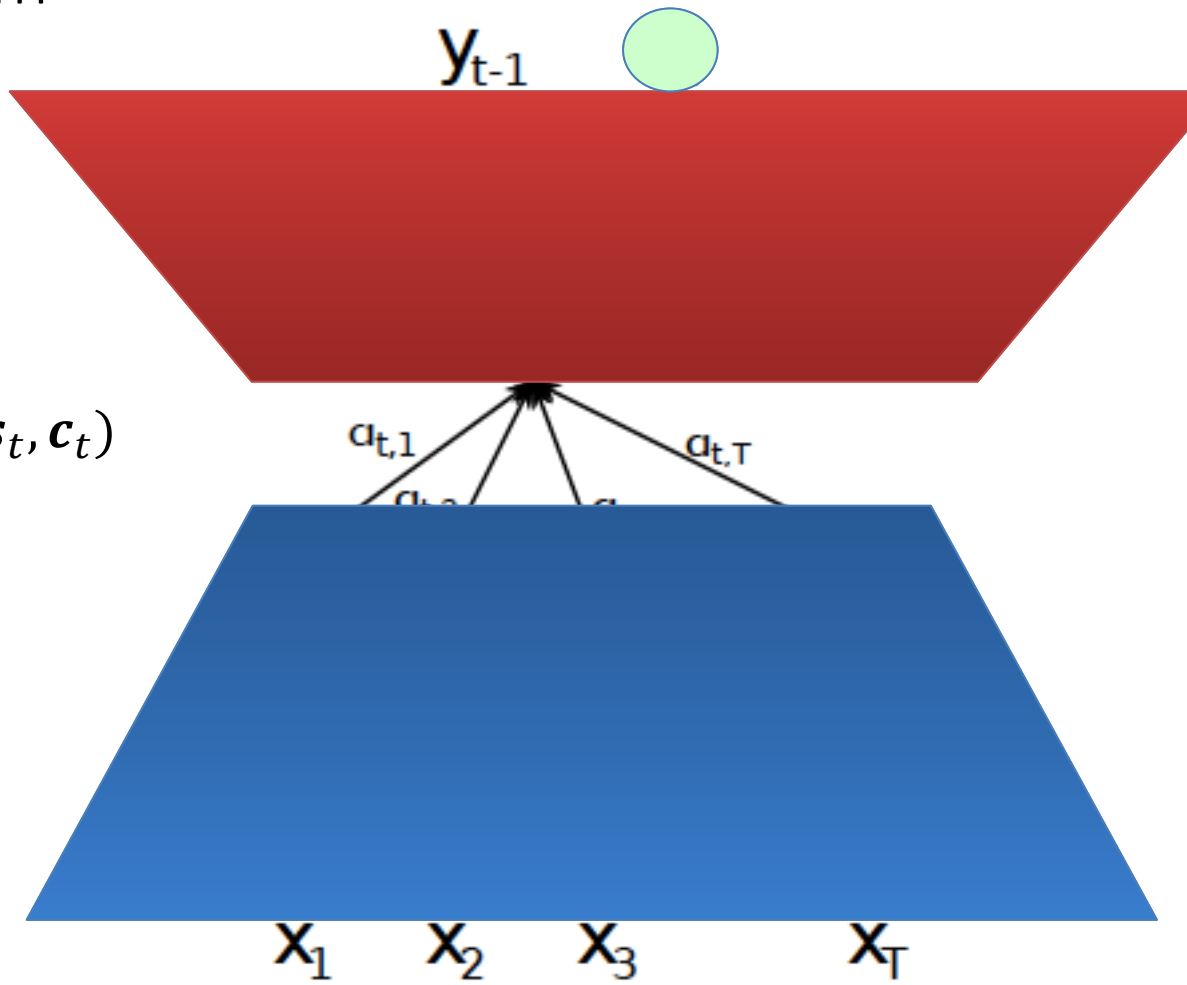
- **Attention** provides a solution to the bottleneck problem.

- Core idea: on each step of the decoder, *focus on a particular part* of the source sequence

# Jointly Learning to Align and Translate

- Attention mechanism

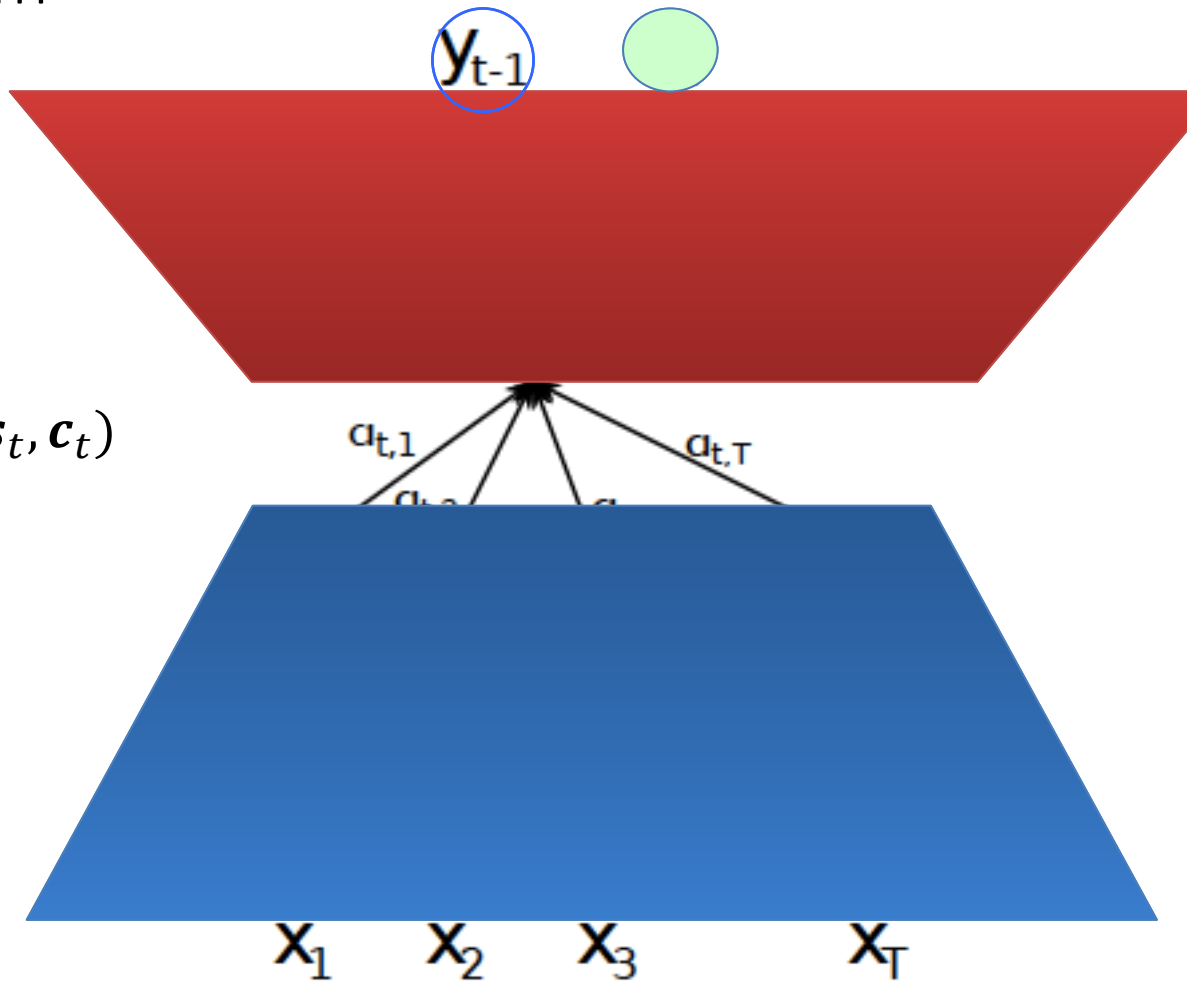
$$p(y_t|y_{t-1}, \dots, y_1, \boldsymbol{x}) = f(y_{t-1}, \boldsymbol{s}_t, \boldsymbol{c}_t)$$



# Jointly Learning to Align and Translate

- Attention mechanism

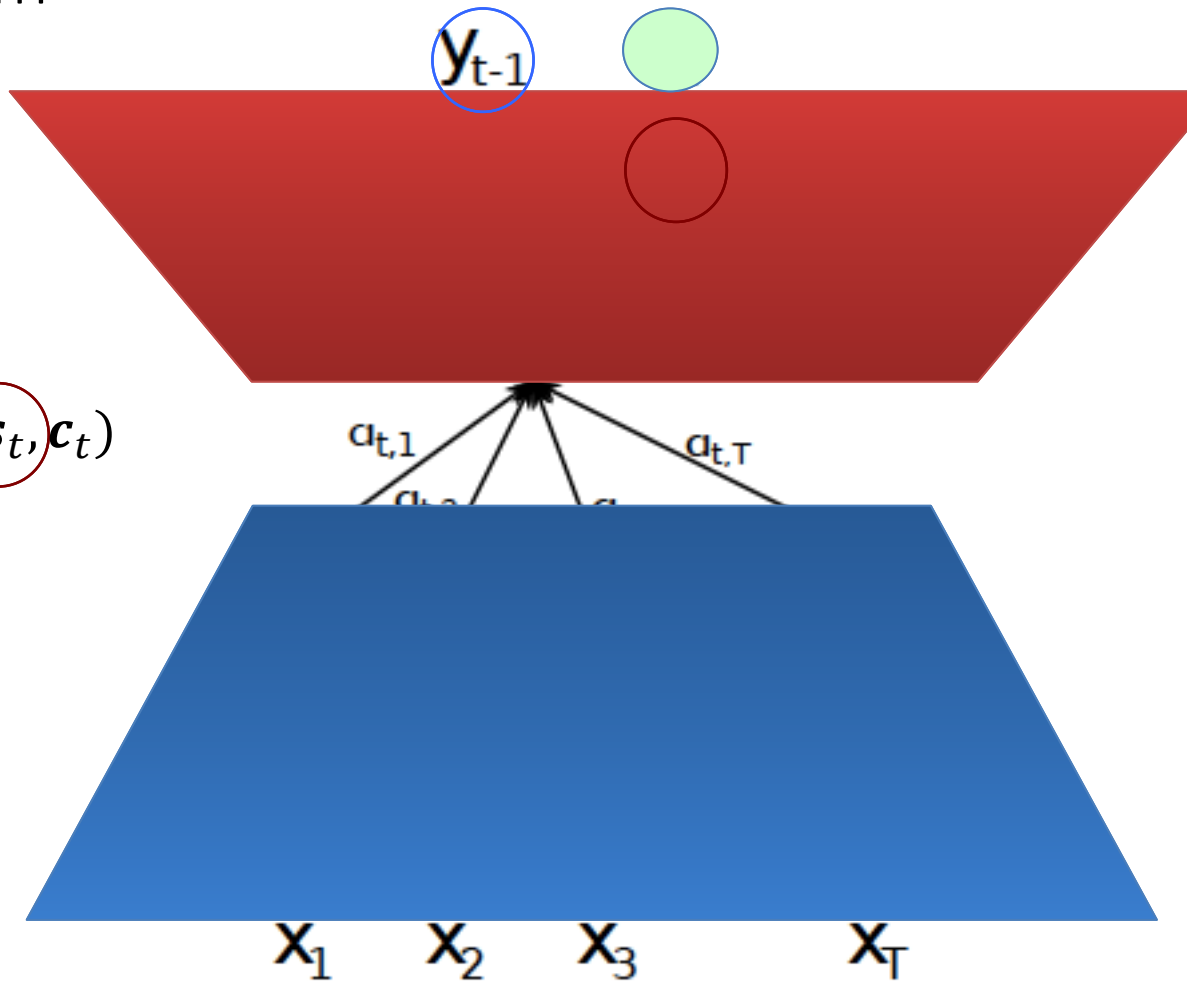
$$p(y_t|y_{t-1}, \dots, y_1, \boldsymbol{x}) = f(y_{t-1}, \boldsymbol{s}_t, \boldsymbol{c}_t)$$



# Jointly Learning to Align and Translate

- Attention mechanism

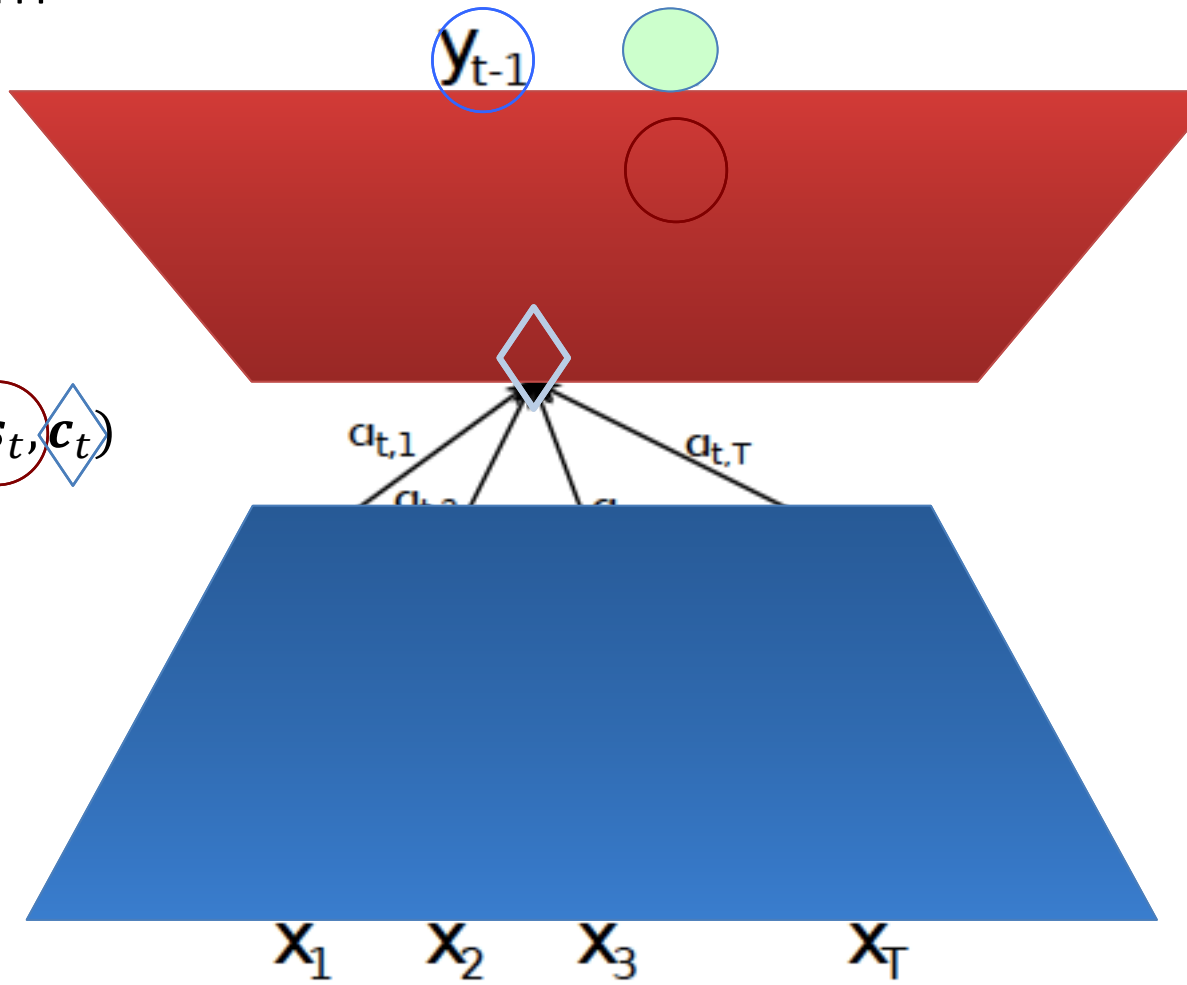
$$p(y_t|y_{t-1},\dots,y_1,\boldsymbol{x}) = f(y_{t-1},\boldsymbol{s}_t,\boldsymbol{c}_t)$$



# Jointly Learning to Align and Translate

- Attention mechanism

$$p(y_t|y_{t-1},\dots,y_1,\boldsymbol{x}) = f(y_{t-1},\boldsymbol{s}_t,\boldsymbol{c}_t)$$

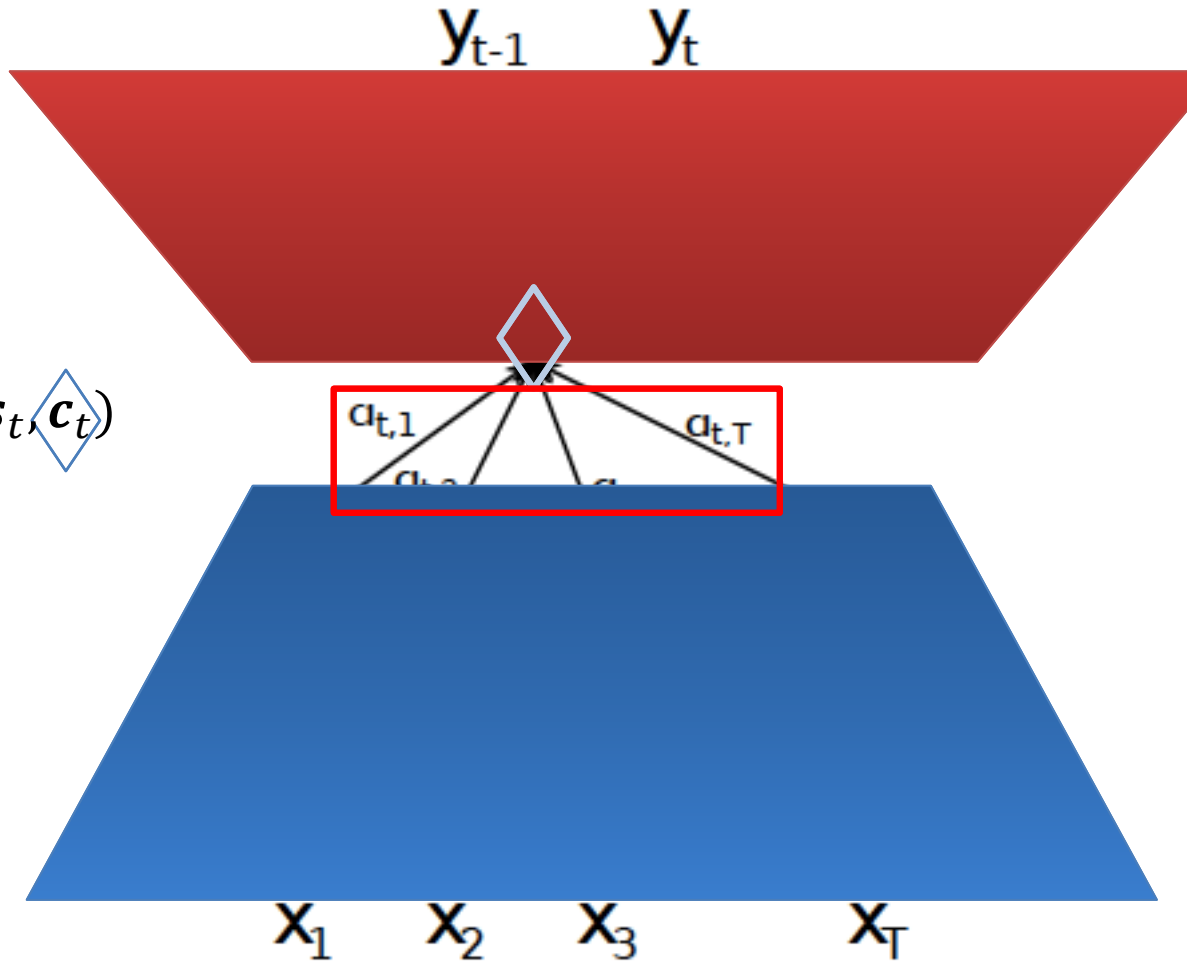


# Jointly Learning to Align and Translate

- Attention mechanism

$$p(y_t|y_{t-1},\dots,y_1,\boldsymbol{x}) = f(y_{t-1},\boldsymbol{s}_t,\boldsymbol{c}_t)$$

$$\boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j}\boldsymbol{h}_j$$



# Jointly Learning to Align and Translate

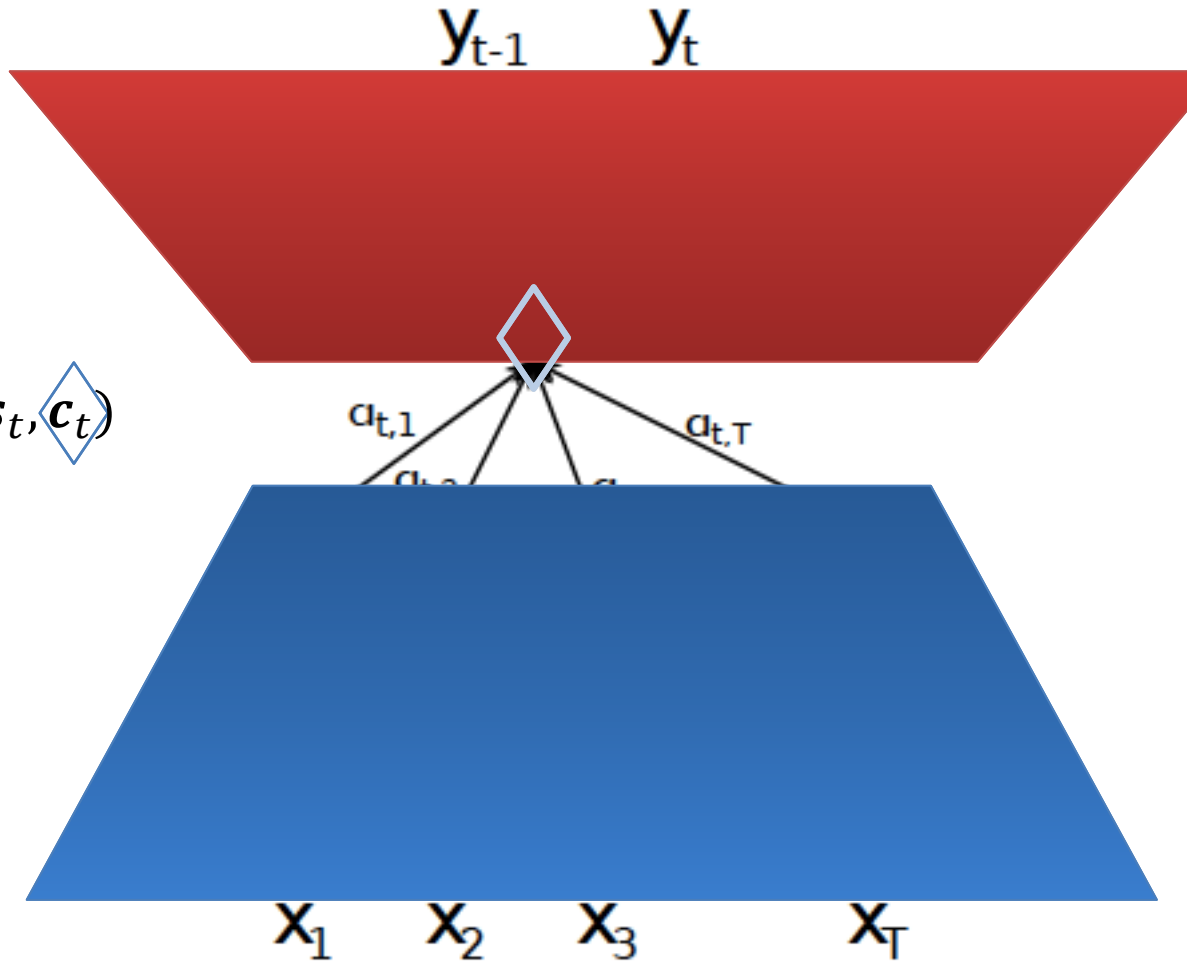
- Attention mechanism

$$p(y_t|y_{t-1},\dots,y_1,\boldsymbol{x}) = f(y_{t-1},\boldsymbol{s}_t,\boldsymbol{c}_t)$$

$$\boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j}\boldsymbol{h}_j$$

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T}\exp(e_{t,k})}$$

$$e_{t,j} = a(\boldsymbol{s}_{t-1},\boldsymbol{h}_j)$$



# Jointly Learning to Align and Translate

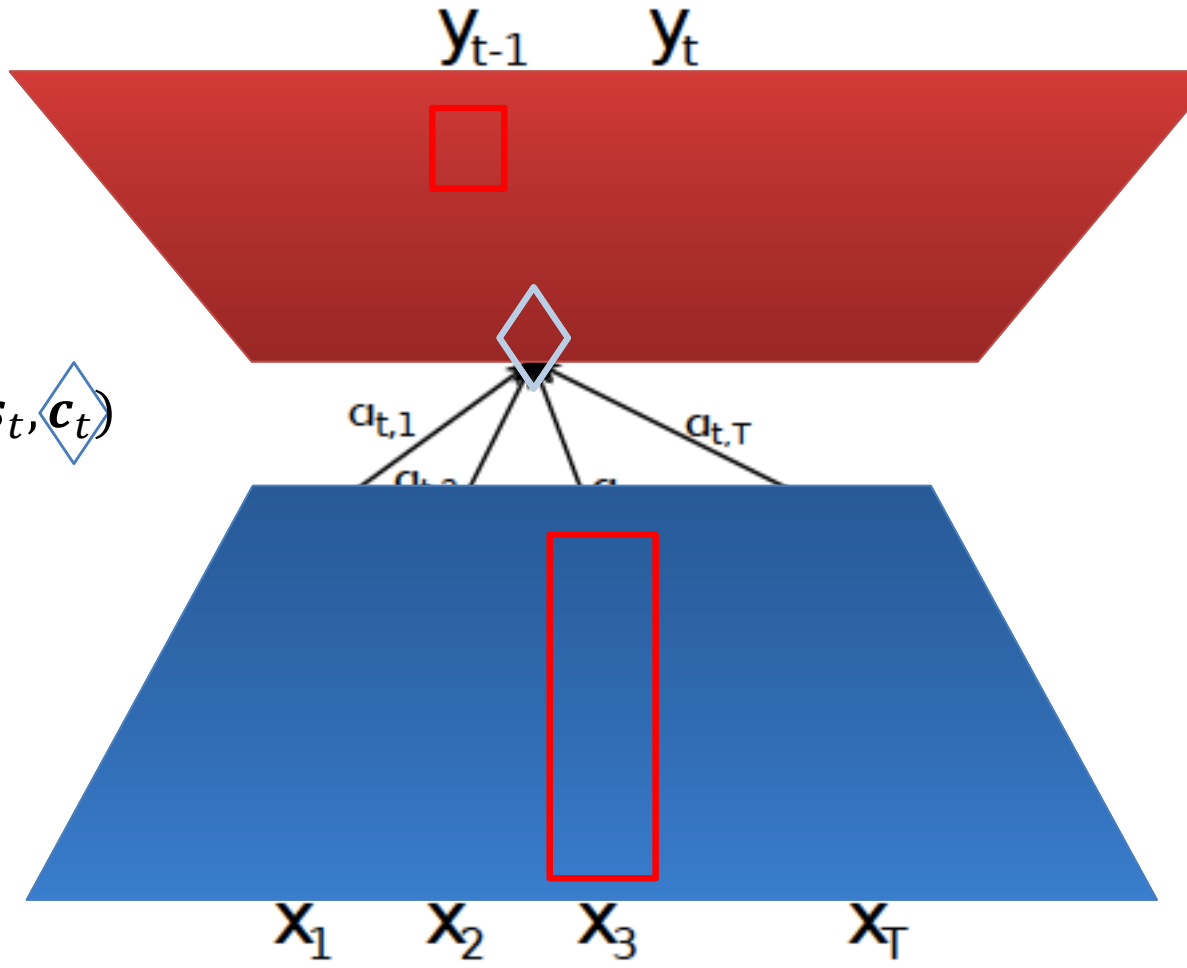
- Attention mechanism

$$p(y_t|y_{t-1},\dots,y_1,\boldsymbol{x}) = f(y_{t-1},\boldsymbol{s}_t,\boldsymbol{c}_t)$$

$$\boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j}\boldsymbol{h}_j$$

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T}\exp(e_{t,k})}$$

$$e_{t,j} = a(\boldsymbol{s}_{t-1},\boldsymbol{h}_j)$$



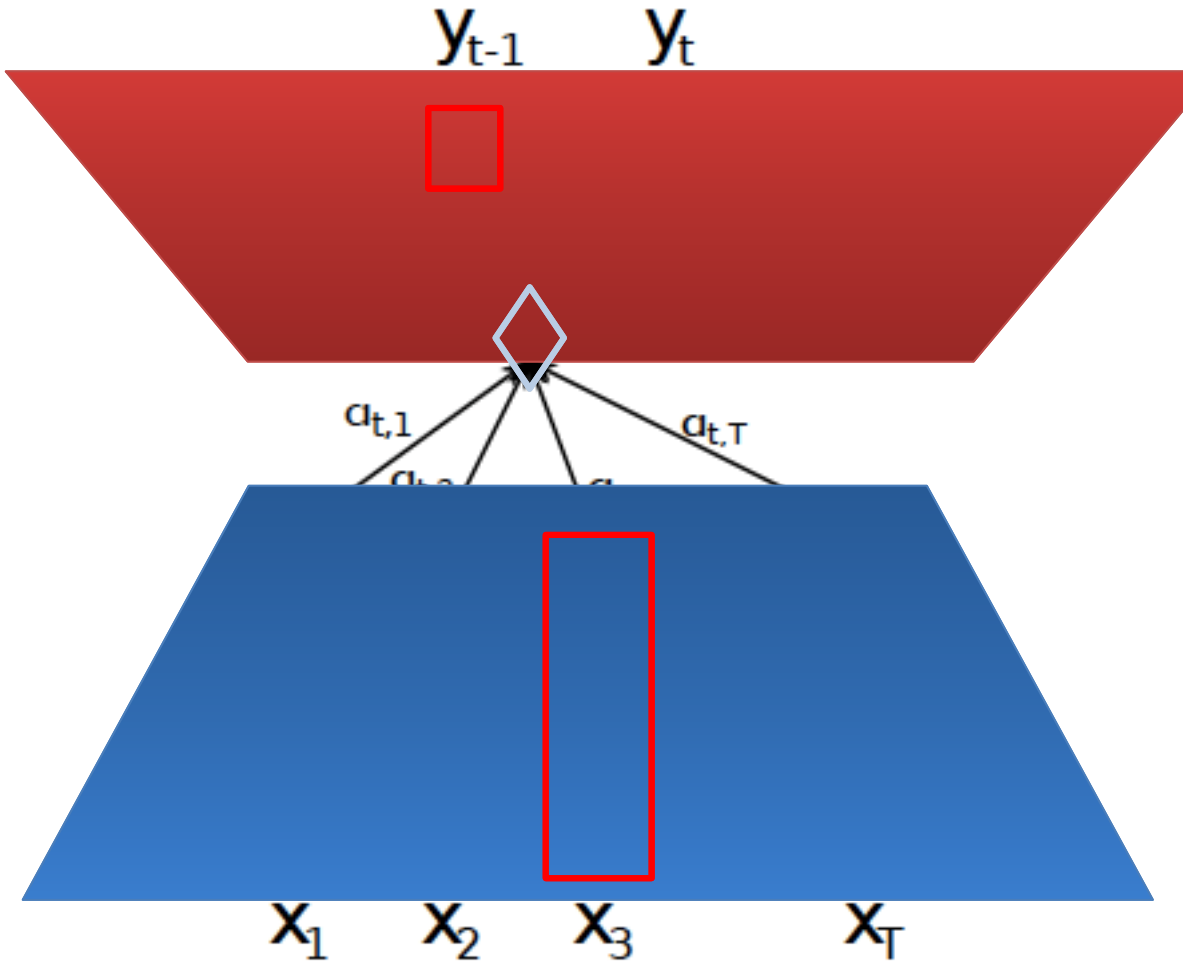
# Attention variants

$$p(y_t|y_{t-1},\dots,y_1,\boldsymbol{x}) = f(y_{t-1},\boldsymbol{s}_t,\boldsymbol{c}_t)$$

$$\boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j}\boldsymbol{h}_j$$

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T}\exp(e_{t,k})}$$

$$e_{t,j} = a(\boldsymbol{s}_{t-1},\boldsymbol{h}_j)$$




Basic dot-product attention

$$e_{tj} = s_{t-1}^{T} h_j$$

# Attention variants

$$p(y_t|y_{t-1},\dots,y_1,\boldsymbol{x}) = f(y_{t-1},\boldsymbol{s}_t,\boldsymbol{c}_t)$$

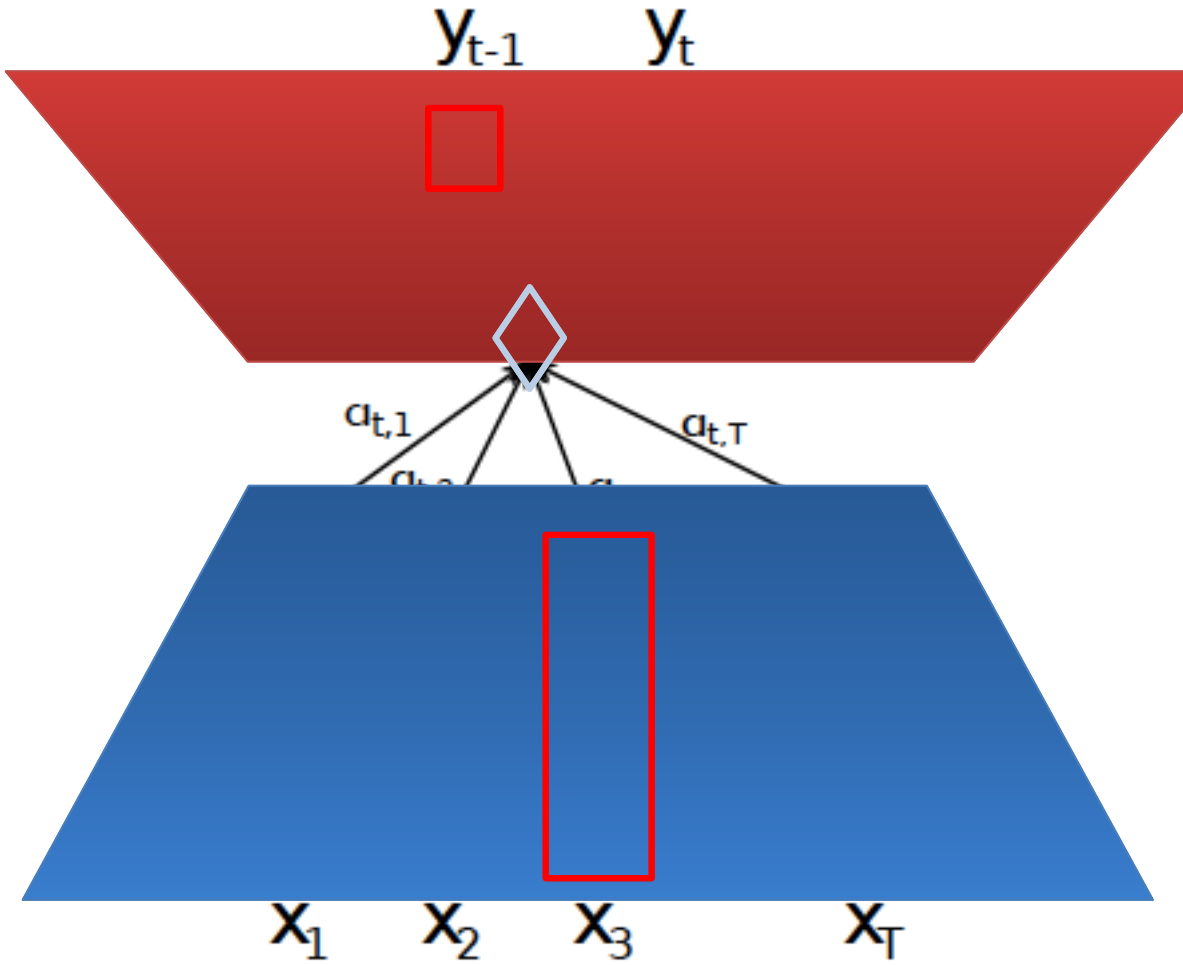
$$\boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j}\boldsymbol{h}_j$$

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T}\exp(e_{t,k})}$$

$$e_{t,j} = a(\boldsymbol{s}_{t-1},\boldsymbol{h}_j)$$

Multiplicative attention .
(bilinear mode)

$$e_{tj} = s_{t-1}^{T} W h_j$$



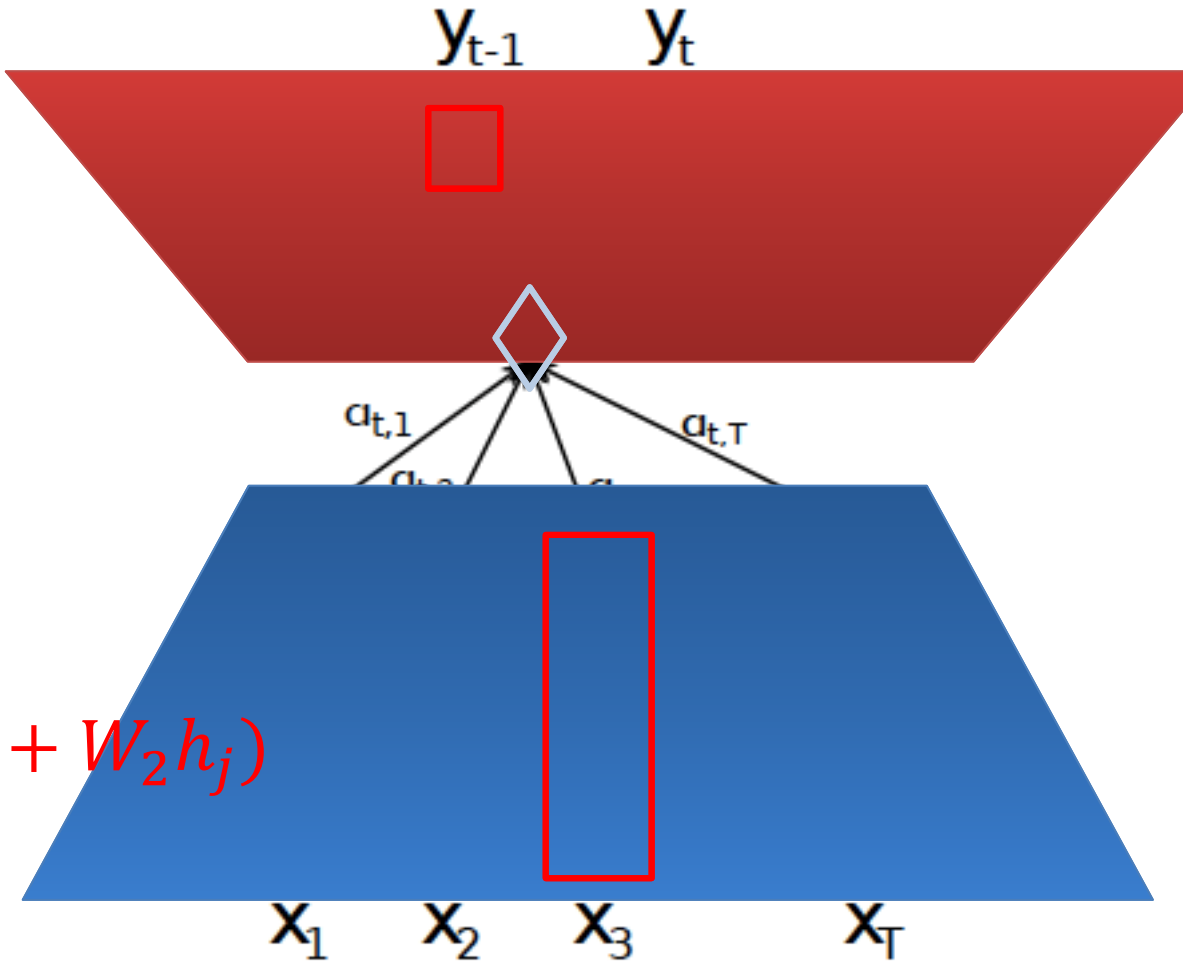
# Attention variants

$$p(y_t|y_{t-1},\dots,y_1,\boldsymbol{x}) = f(y_{t-1},\boldsymbol{s}_t,\boldsymbol{c}_t)$$

$$\boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j}\boldsymbol{h}_j$$

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T}\exp(e_{t,k})}$$

$$e_{t,j} = a(\boldsymbol{s}_{t-1},\boldsymbol{h}_j)$$



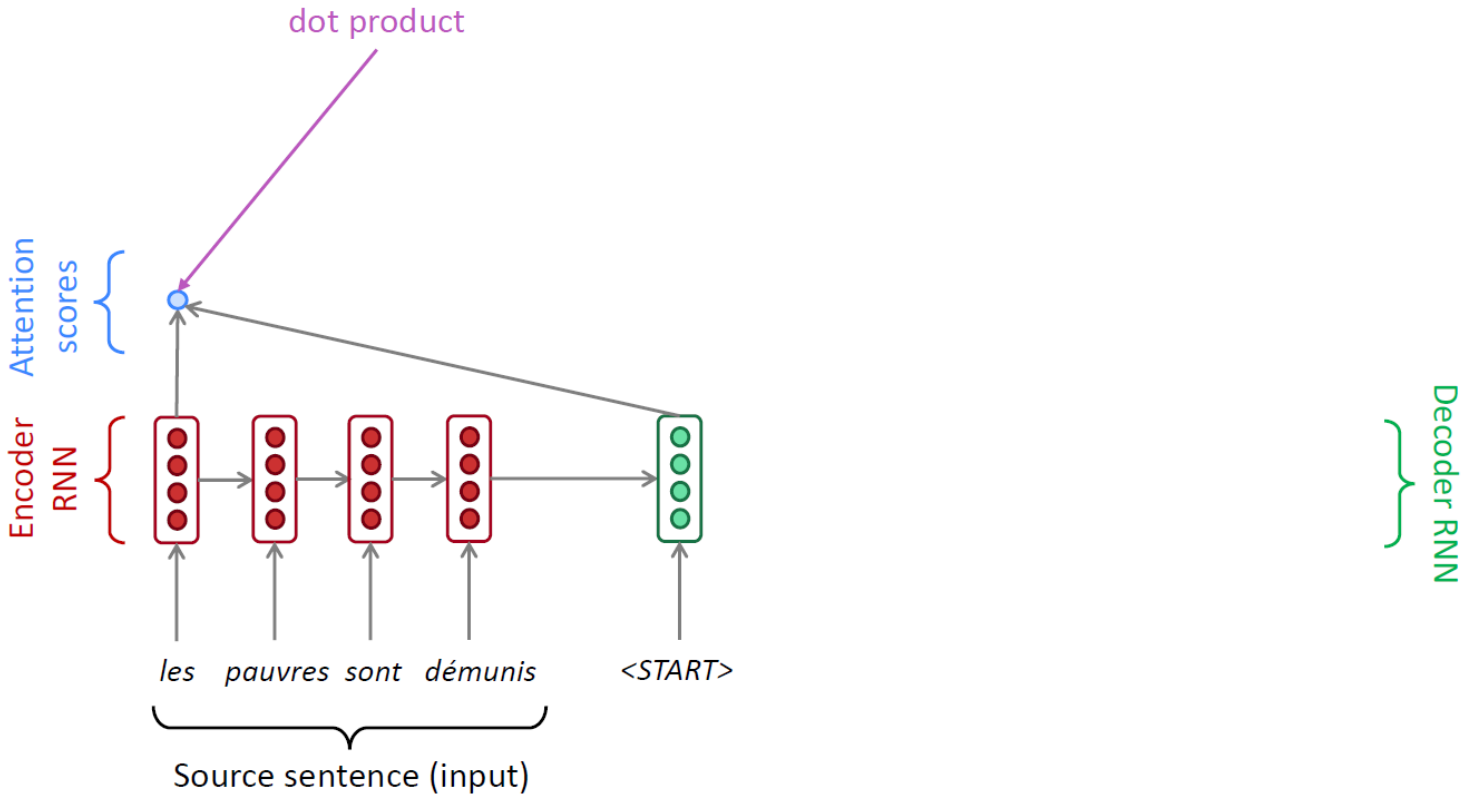

Additive attention

$$e_{tj} = v^T \tanh(W_1 s_{t-1}^T + W_2 h_j)$$

# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})}$$

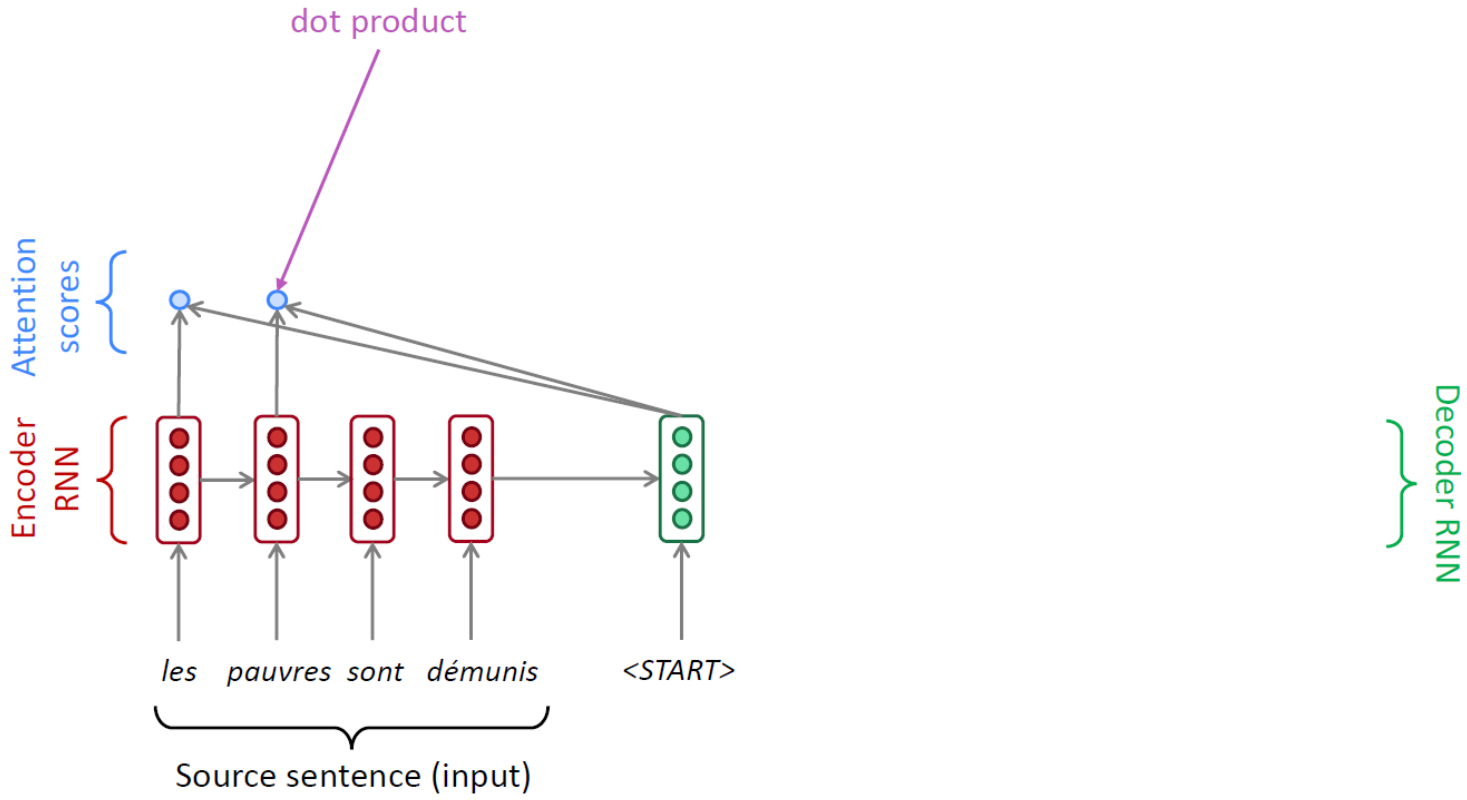
$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})}$$

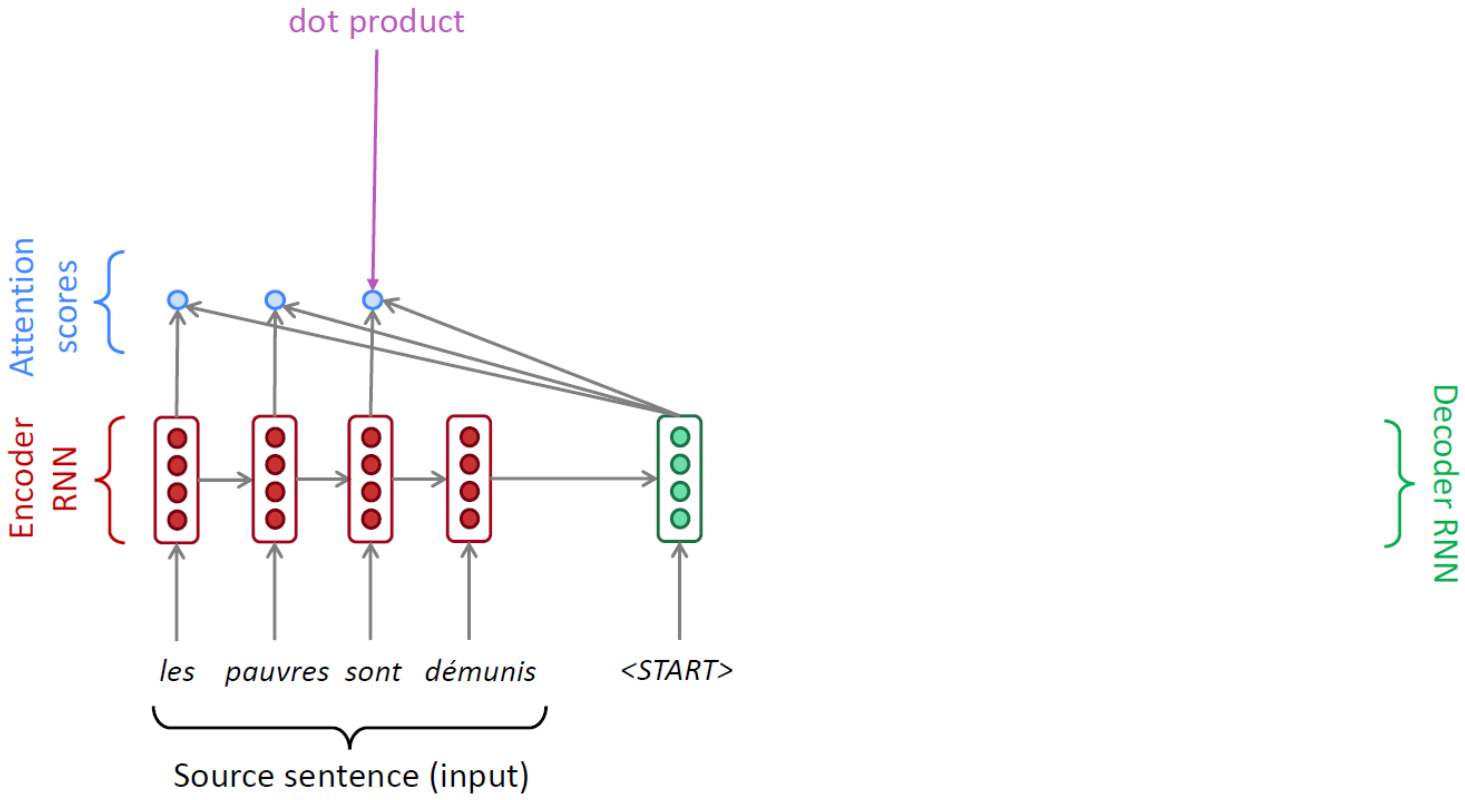
$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})}$$

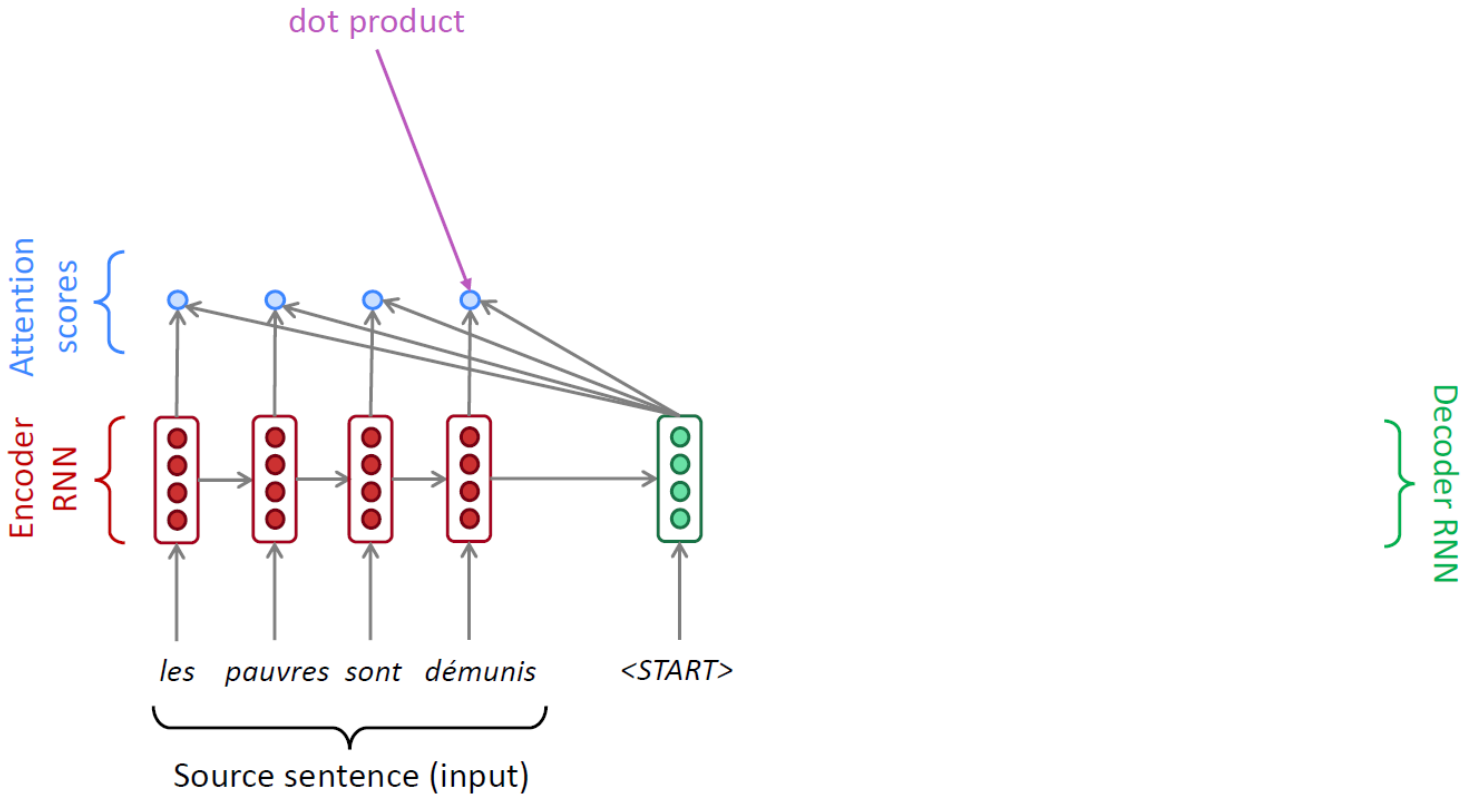
$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})}$$

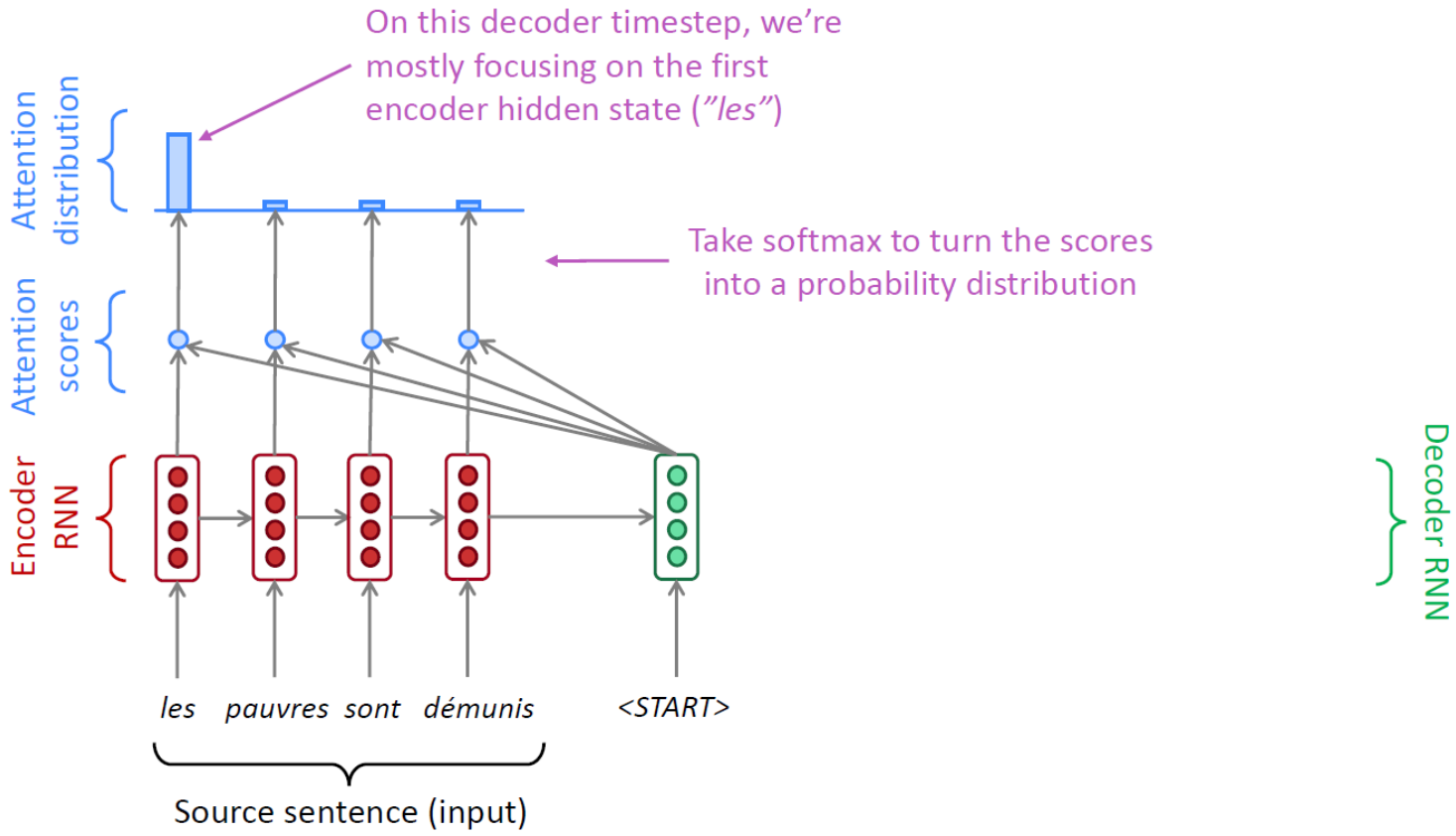
$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})}$$

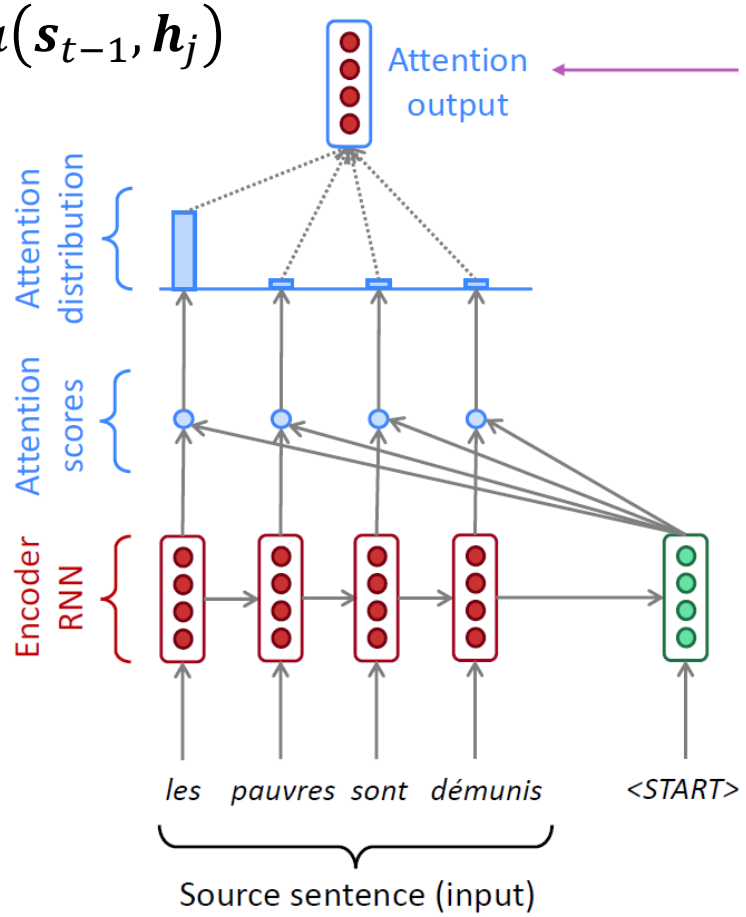
$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})} \qquad \boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j} \boldsymbol{h}_j$$

$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$




Use the attention distribution to take a **weighted sum** of the encoder hidden states.

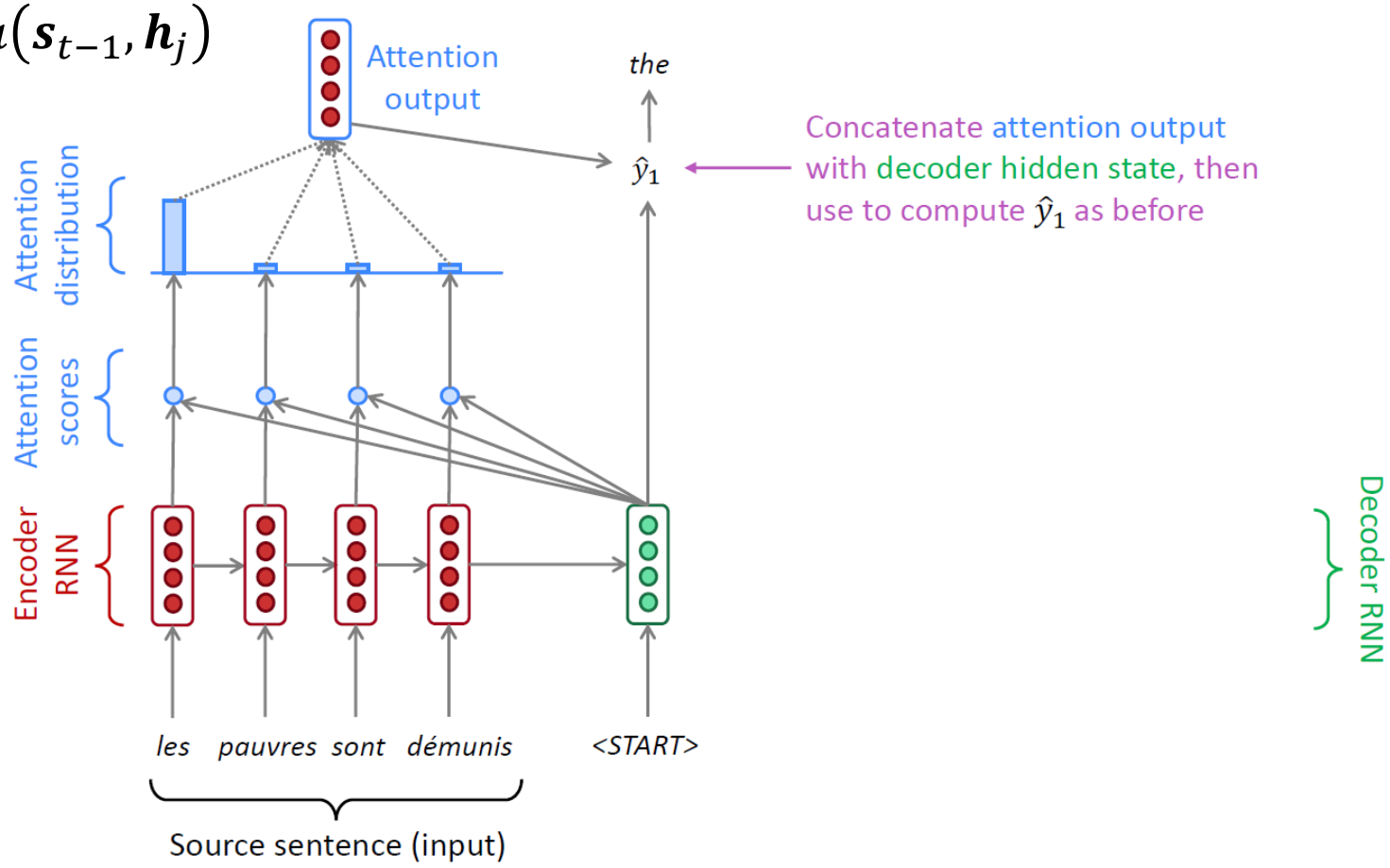
The attention output mostly contains information the hidden states that received high attention.

Decoder RNN

# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})} \qquad \boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j} \boldsymbol{h}_j \qquad p(y_t | y_{t-1}, \dots, y_1, \boldsymbol{x}) = f(y_{t-1}, \boldsymbol{s}_t, \boldsymbol{c}_t)$$

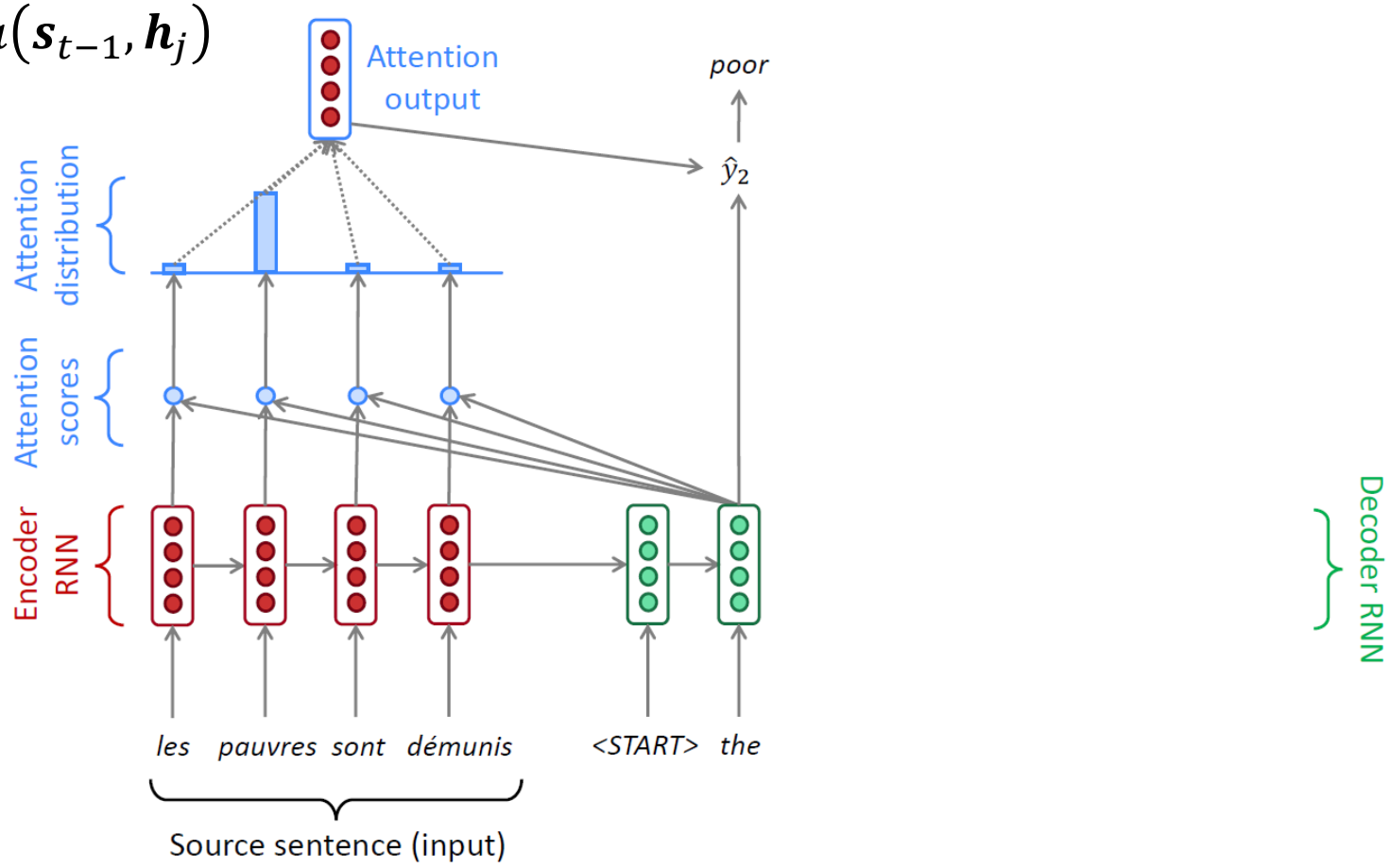
$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})} \qquad \boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j} \boldsymbol{h}_j \qquad p(y_t | y_{t-1}, \dots, y_1, \boldsymbol{x}) = f(y_{t-1}, \boldsymbol{s}_t, \boldsymbol{c}_t)$$

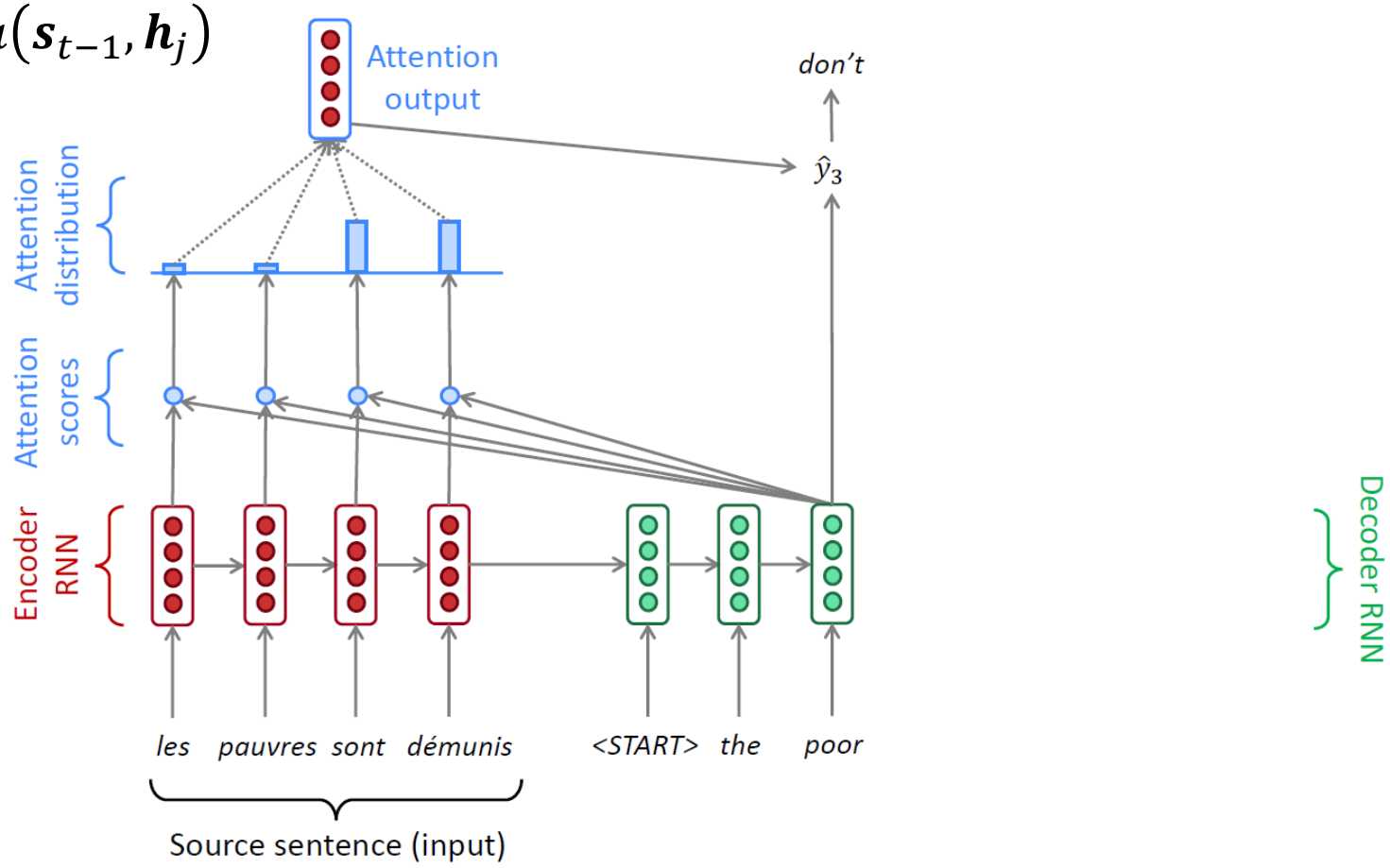
$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})} \qquad \boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j} \boldsymbol{h}_j \qquad p(y_t | y_{t-1}, \dots, y_1, \boldsymbol{x}) = f(y_{t-1}, \boldsymbol{s}_t, \boldsymbol{c}_t)$$

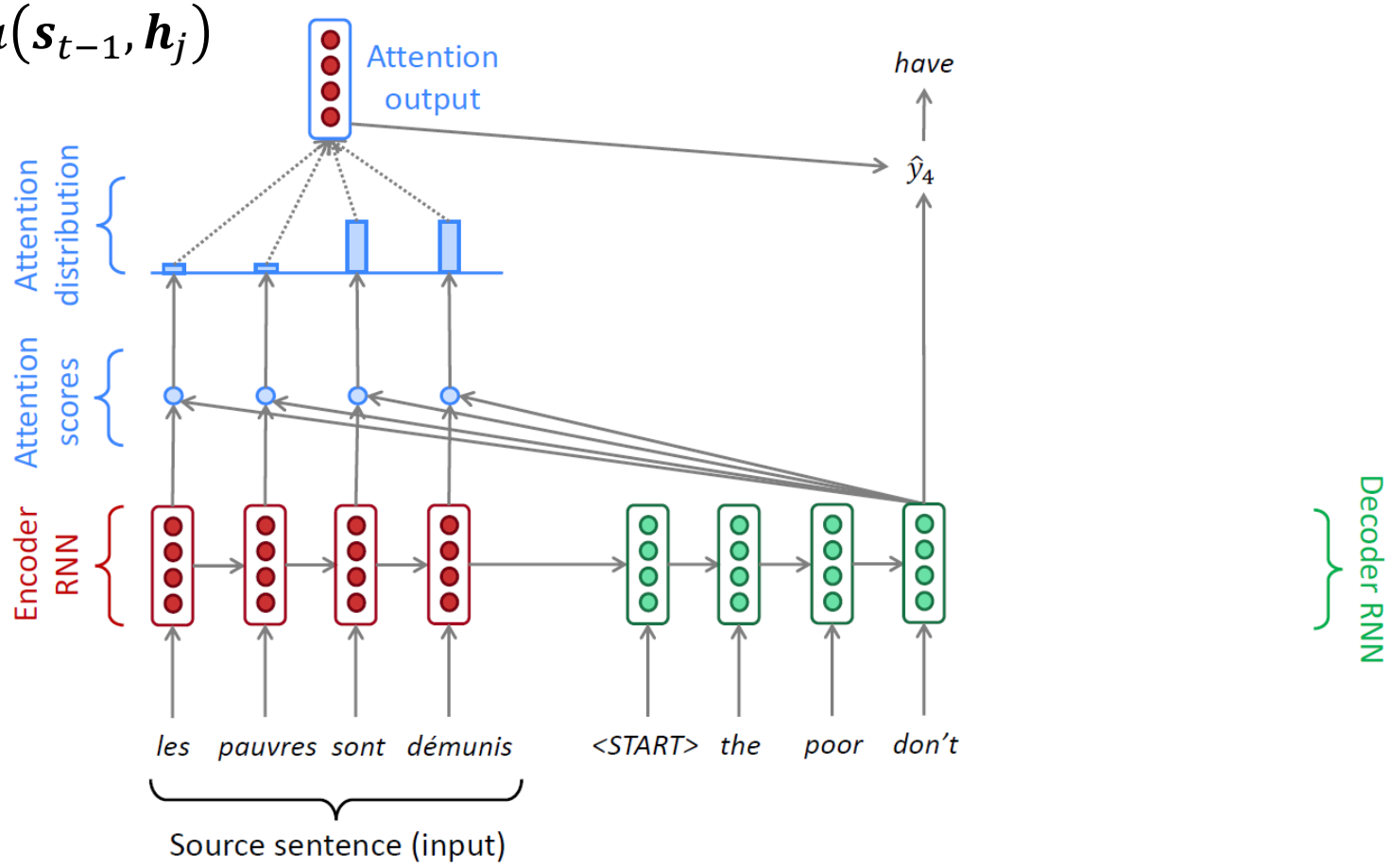
$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})} \qquad \boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j} \boldsymbol{h}_j \qquad p(y_t | y_{t-1}, \dots, y_1, \boldsymbol{x}) = f(y_{t-1}, \boldsymbol{s}_t, \boldsymbol{c}_t)$$

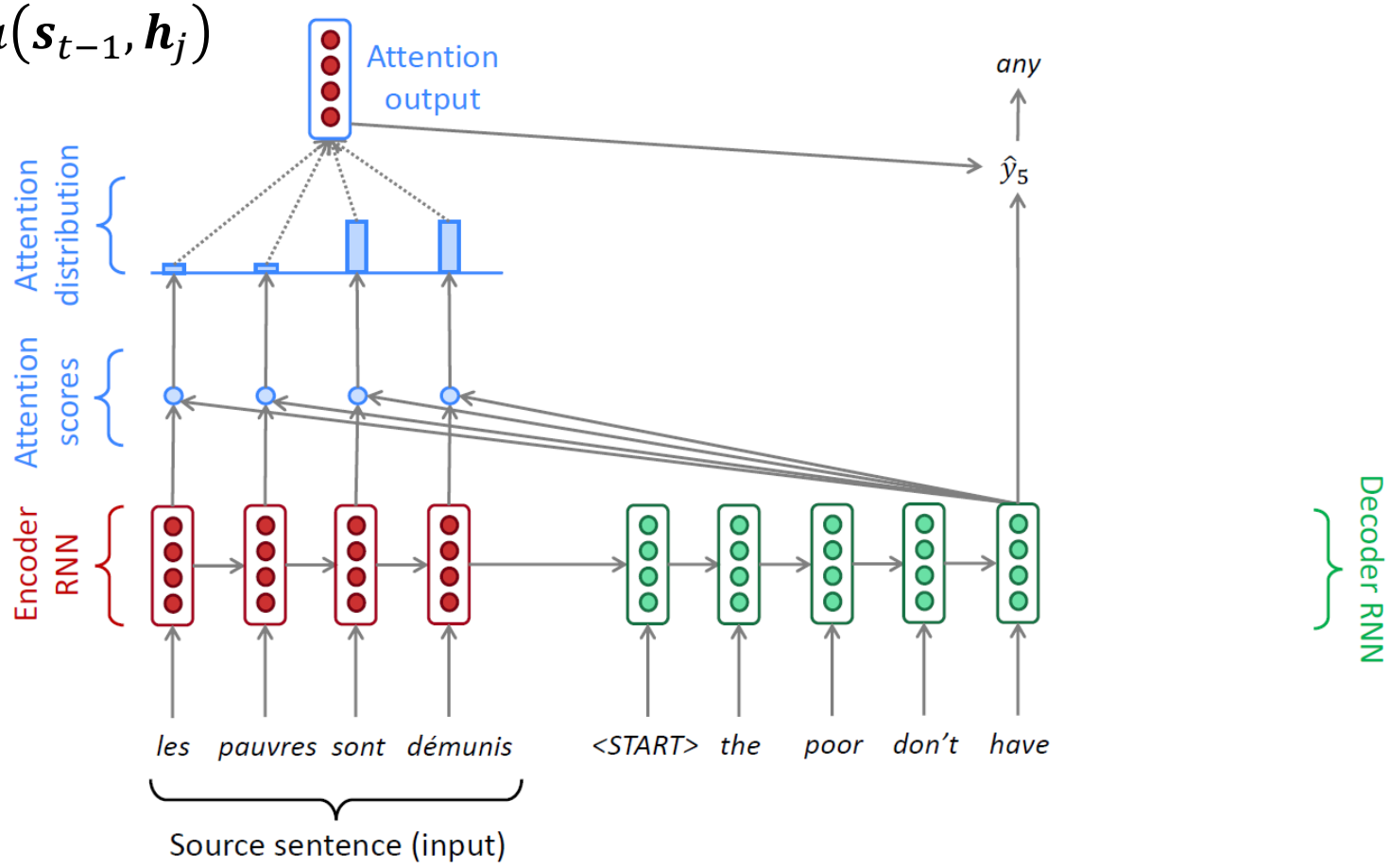
$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# A Running Example

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})} \qquad \boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j} \boldsymbol{h}_j \qquad p(y_t | y_{t-1}, \dots, y_1, \boldsymbol{x}) = f(y_{t-1}, \boldsymbol{s}_t, \boldsymbol{c}_t)$$

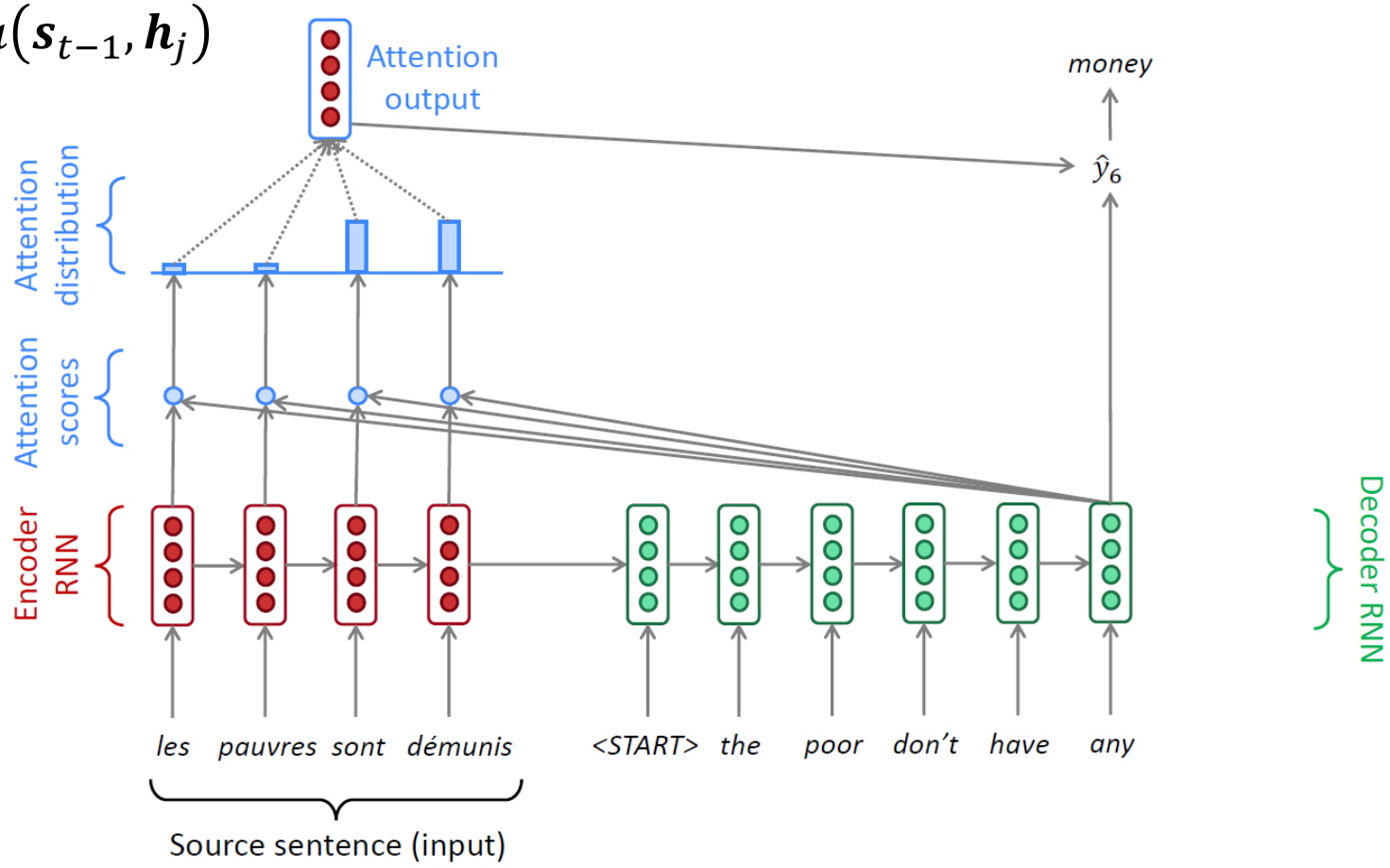
$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# A Running Example

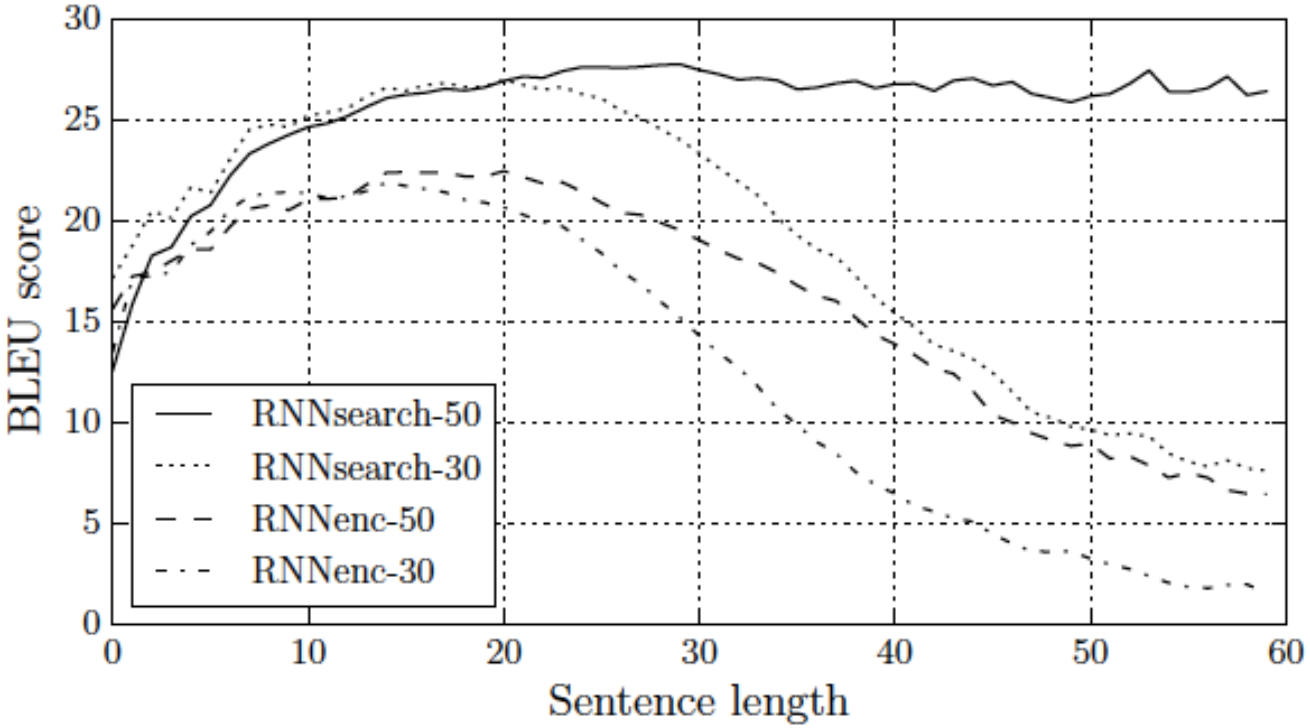
$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})} \qquad \boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{t,j} \boldsymbol{h}_j \qquad p(y_t | y_{t-1}, \dots, y_1, \boldsymbol{x}) = f(y_{t-1}, \boldsymbol{s}_t, \boldsymbol{c}_t)$$

$$e_{t,j} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$$



# Jointly Learning to Align and Translate

- No performance drop for long sentences




"We train each model twice: first with the sentences of length up to 30 words (RNNencdec-30, RNNsearch-30) and then with the sentences of length up to 50 word (RNNencdec-50, RNNsearch-50)."

RNNencdec: Cho et al. 2014 https://www.aclweb.org/anthology/D14-1179
RNNsearch: Bahdanauet al, ICLR 2015, https://arxiv.org/pdf/1409.0473.pdf

# OpenNMT

- http://opennmt.net/

# Google Neural Machine Translation (Wu et al. (2016))

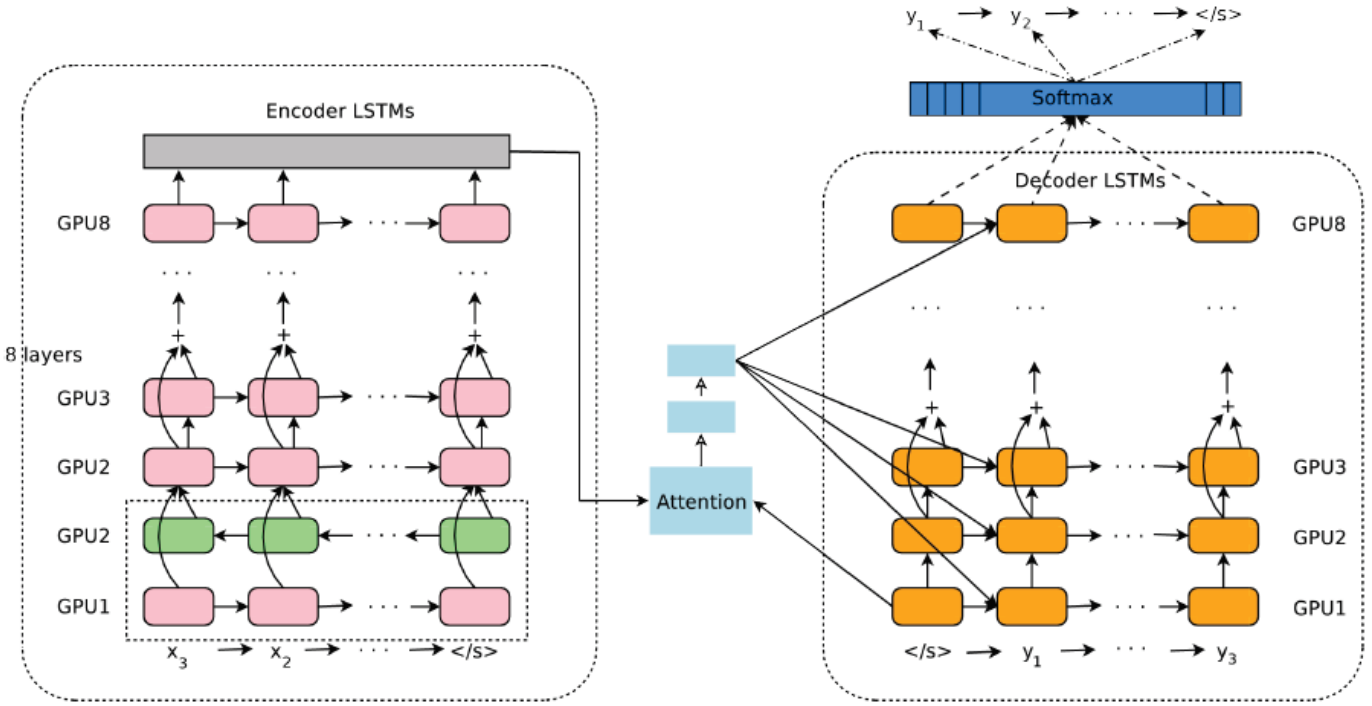
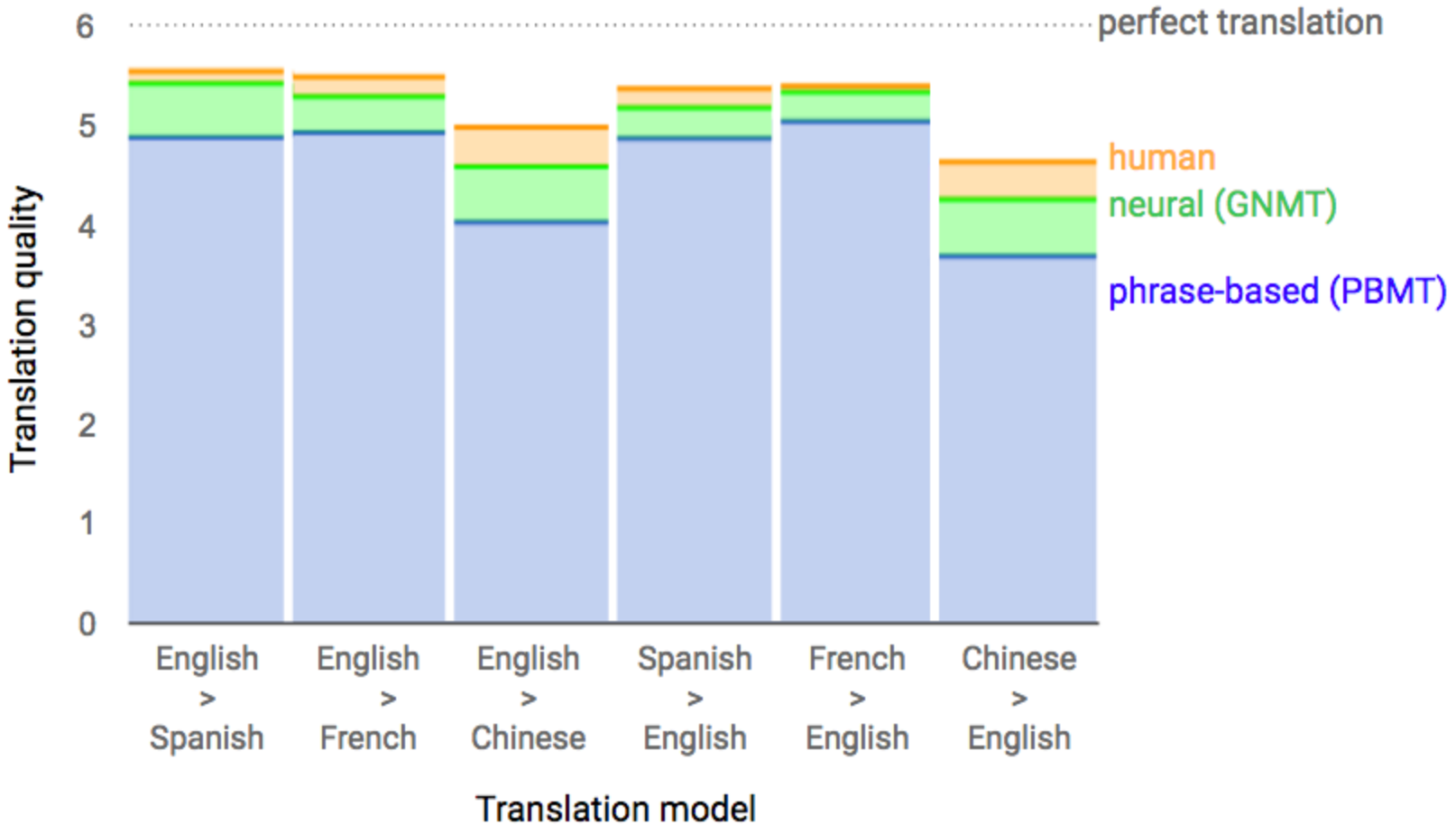


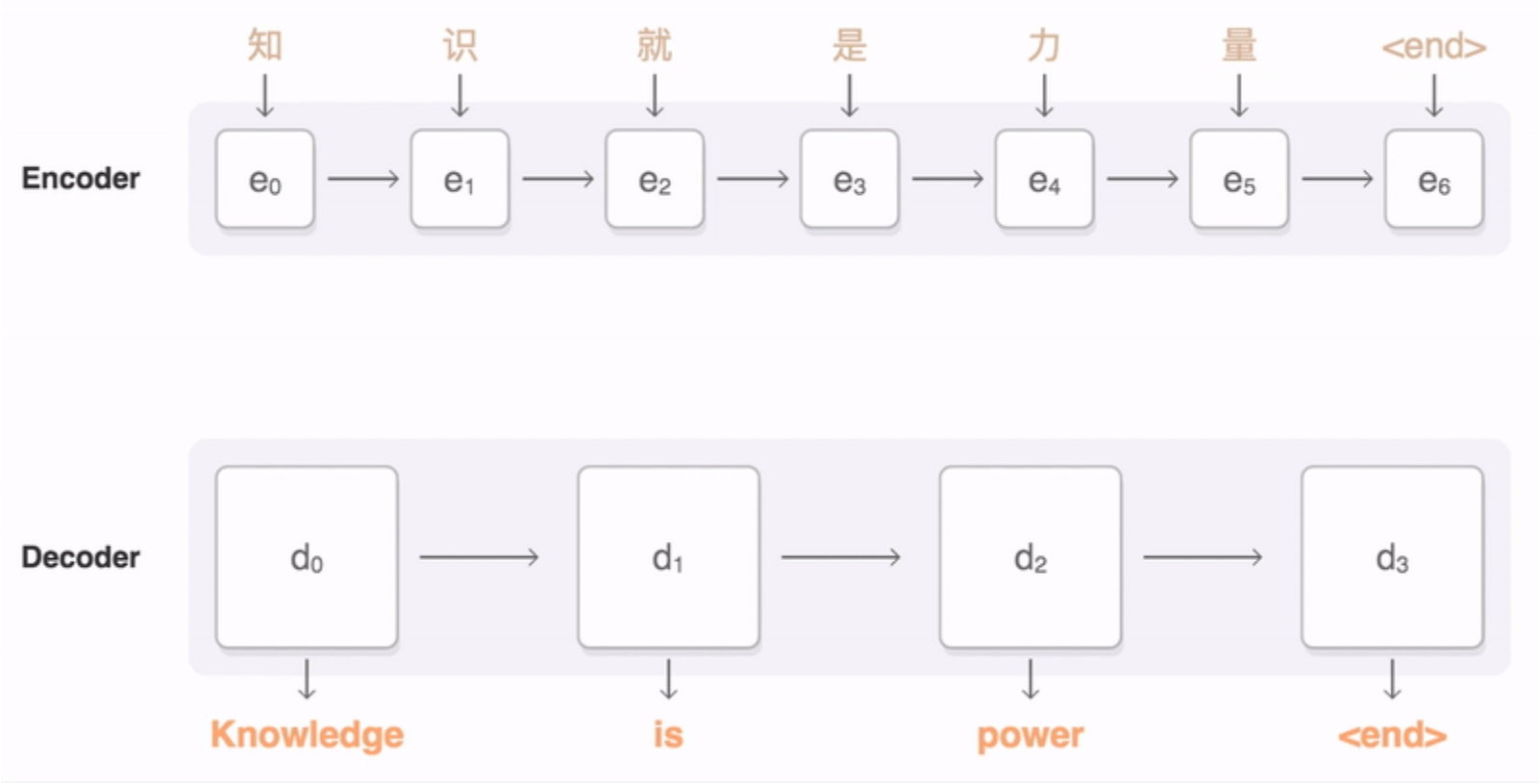

Figure 1: The model architecture of GNMT, Google's Neural Machine Translation system. On the left is the encoder network, on the right is the decoder network, in the middle is the attention module. The bottom encoder layer is bi-directional: the pink nodes gather information from left to right while the green nodes gather information from right to left. The other layers of the encoder are uni-directional. Residual connections start from the layer third from the bottom in the encoder and decoder. The model is partitioned into multiple GPUs to speed up training. In our setup, we have 8 encoder LSTM layers (1 bi-directional layer and 7 uni-directional layers), and 8 decoder layers. With this setting, one model replica is partitioned 8-ways and is placed on 8 different GPUs typically belonging to one host machine. During training, the bottom bi-directional encoder layers compute in parallel first. Once both finish, the uni-directional encoder layers can start computing, each on a separate GPU. To retain as much parallelism as possible during running the decoder layers, we use the bottom decoder layer output only for obtaining recurrent attention context, which is sent directly to all the remaining decoder layers. The softmax layer is also partitioned and placed on multiple GPUs. Depending on the output vocabulary size we either have them run on the same GPUs as the encoder and decoder networks, or have them run on a separate set of dedicated GPUs.

# Google Neural Machine Translation (Wu et al. (2016))




https://ai.googleblog.com/2016/09/a-neural-network-for-machine.html

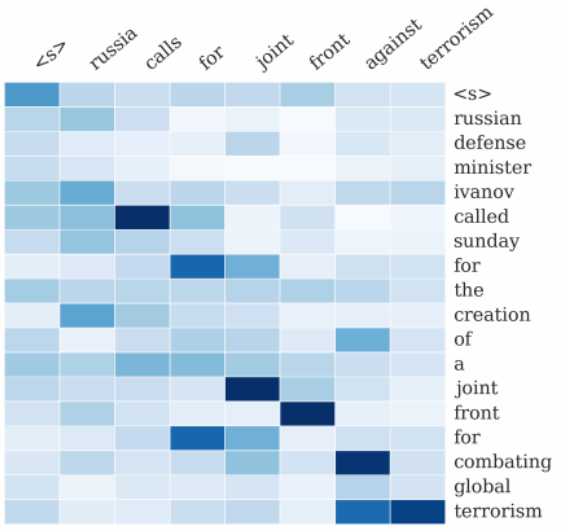
# Google Neural Machine Translation (Wu et al. (2016))




https://ai.googleblog.com/2016/09/a-neural-network-for-machine.html

# Summary

- Attention significantly improves NMT performance
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with vanishing gradient problem
  - Provides shortcut to faraway states
- Attention provides some interpretability
  - By inspecting attention distribution, we can see what the decoder was focusing on
  - We get alignment for free!
  - This is cool because we never explicitly trained an alignment system
  - The network just learned alignment by itself



# Sequence-to-sequence is versatile!

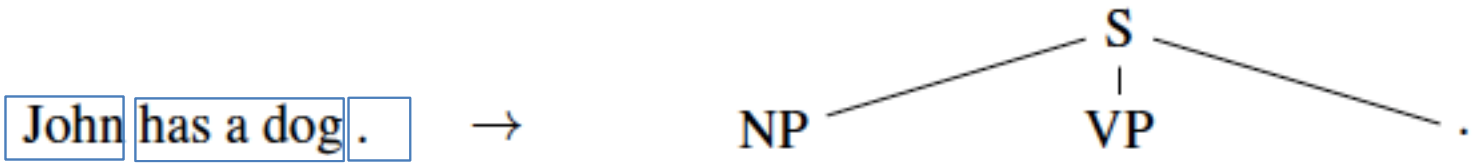
- Sequence-to-sequence is useful for *more than just MT*
- Many NLP tasks can be phrased as sequence-to-sequence:
  - Summarization (long text → short text)
  - Dialogue (previous utterances → next utterance)
  - Parsing (input text → output parse as sequence)
  - Code generation (natural language → Python code)

# Grammar as a Foreign Language

- Vinyals et al., 2015

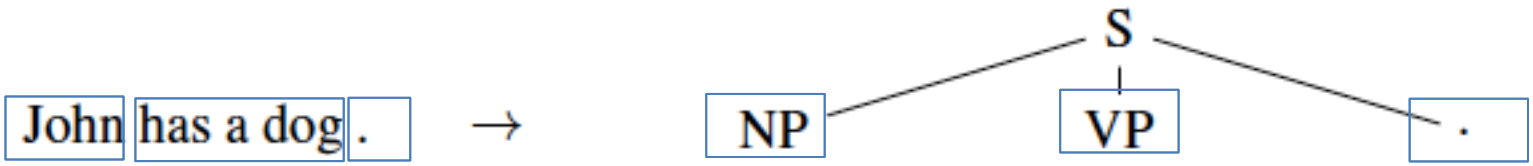
# Grammar as a Foreign Language

Parsing tree



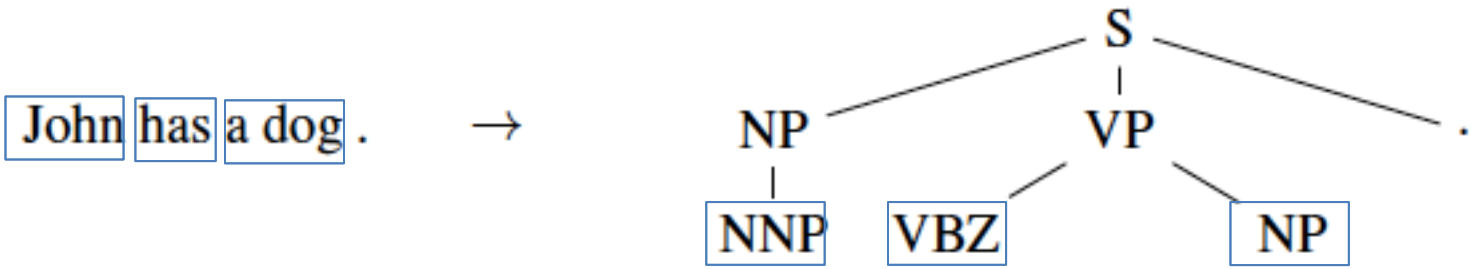
# Grammar as a Foreign Language

Parsing tree



# Grammar as a Foreign Language

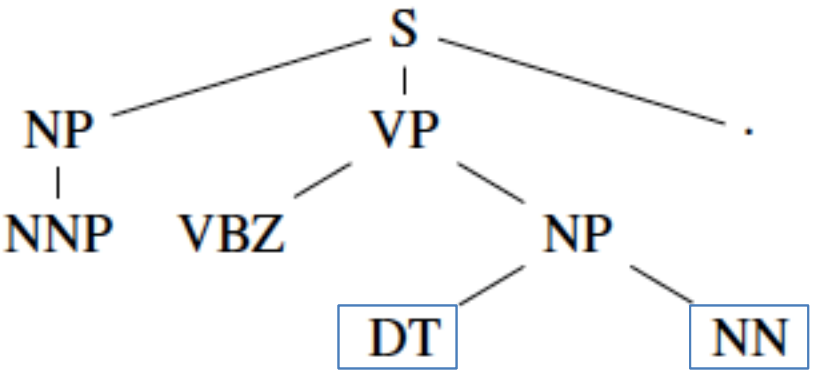
Parsing tree



# Grammar as a Foreign Language

Parsing tree

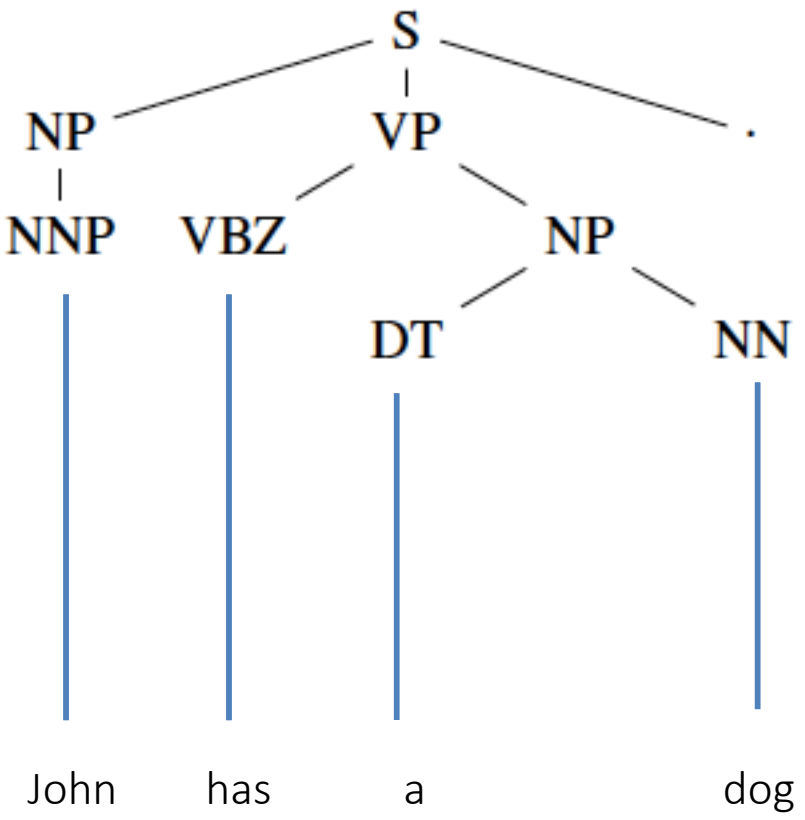
John has a dog . →



# Grammar as a Foreign Language

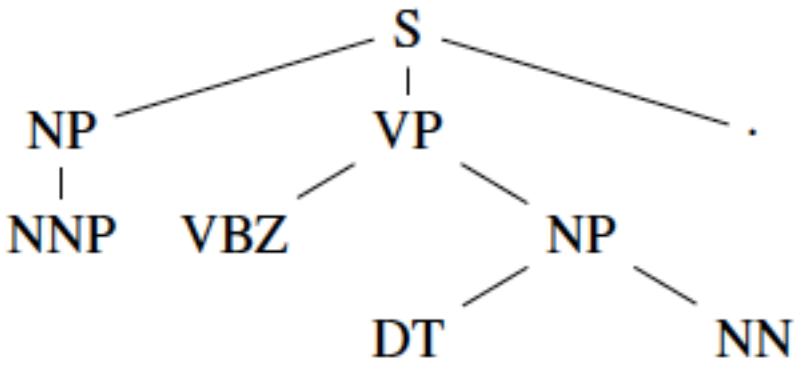
Parsing tree

John has a dog . →



# Grammar as a Foreign Language

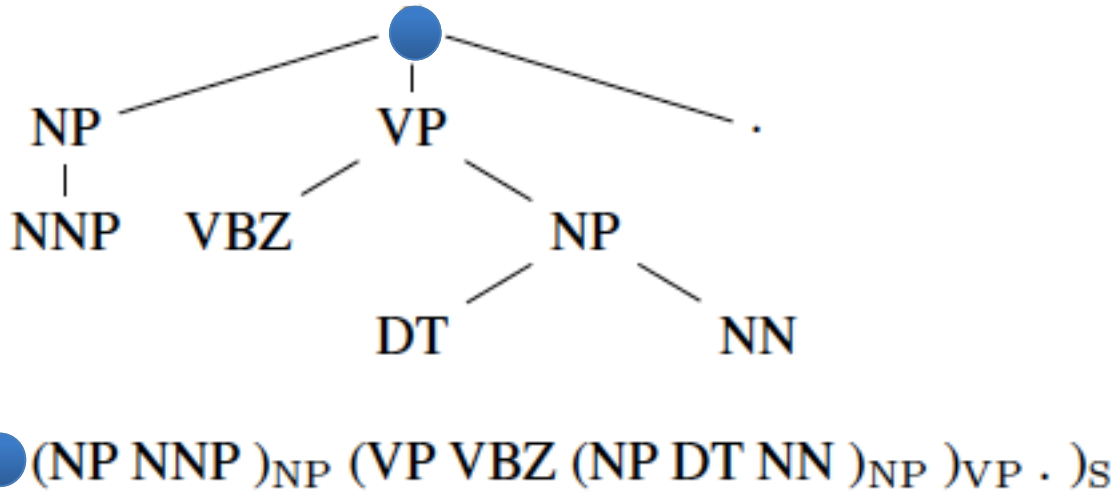
Converting tree to sequence




$$(\text{S (NP NNP )}_{\text{NP}} \text{ (VP VBZ (NP DT NN )}_{\text{NP}} \text{ )}_{\text{VP}} \text{ . )}_{\text{S}}$$

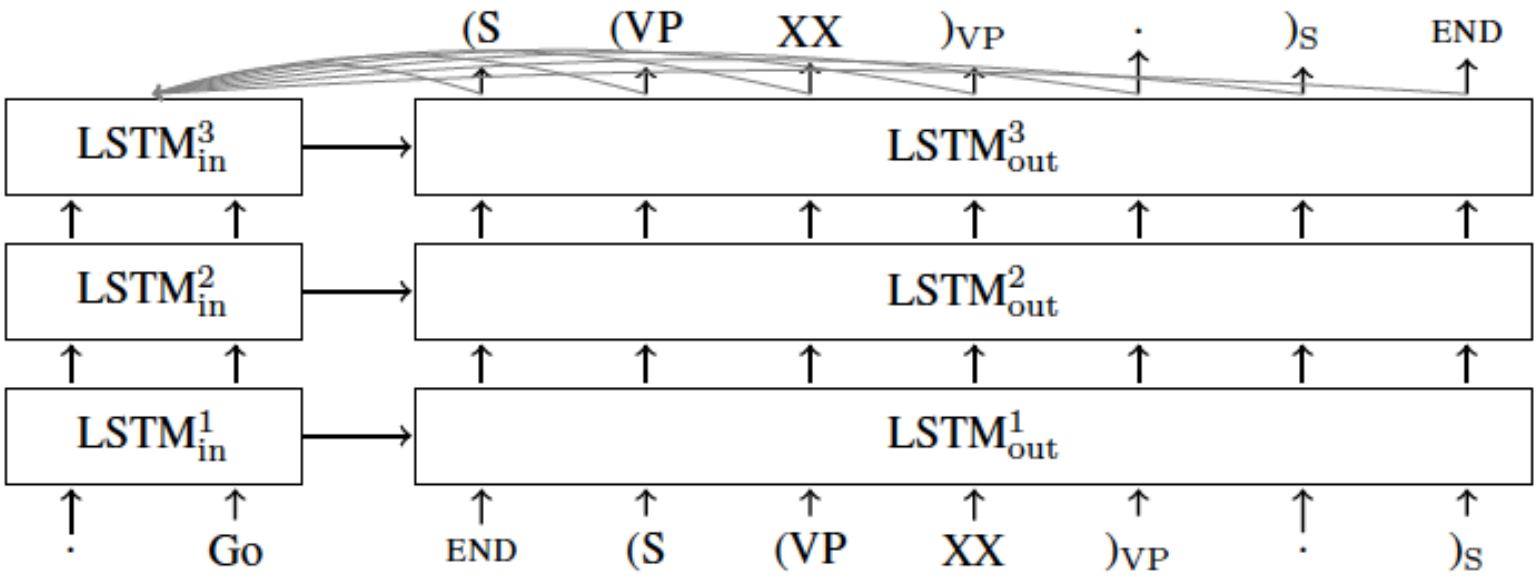
# Grammar as a Foreign Language

Converting tree to sequence



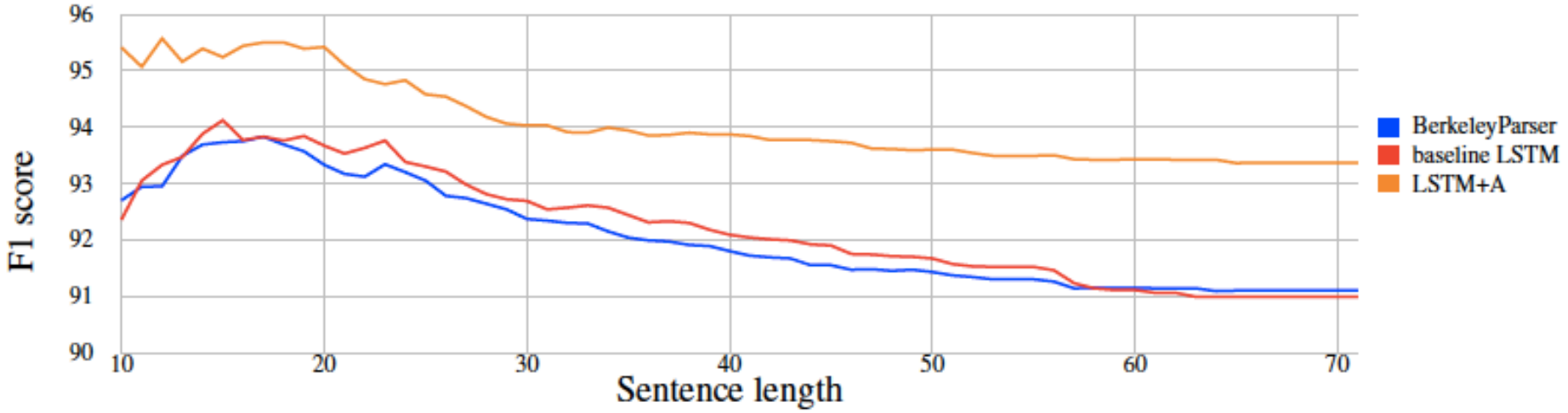
# Grammar as a Foreign Language

Model



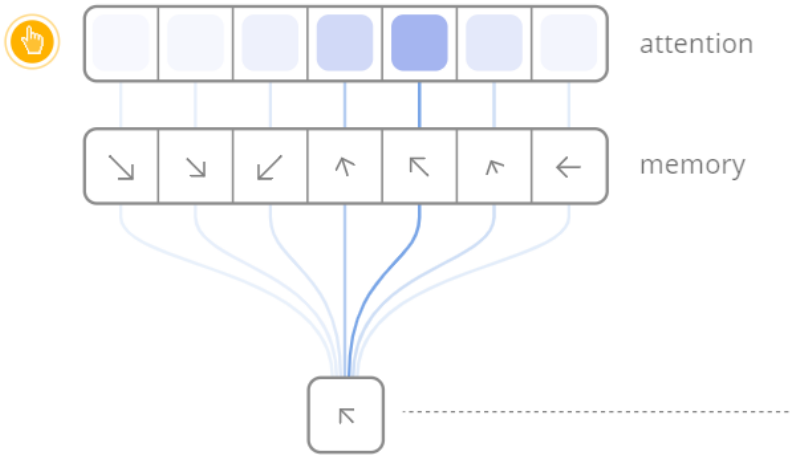
# Grammar as a Foreign Language

## Results



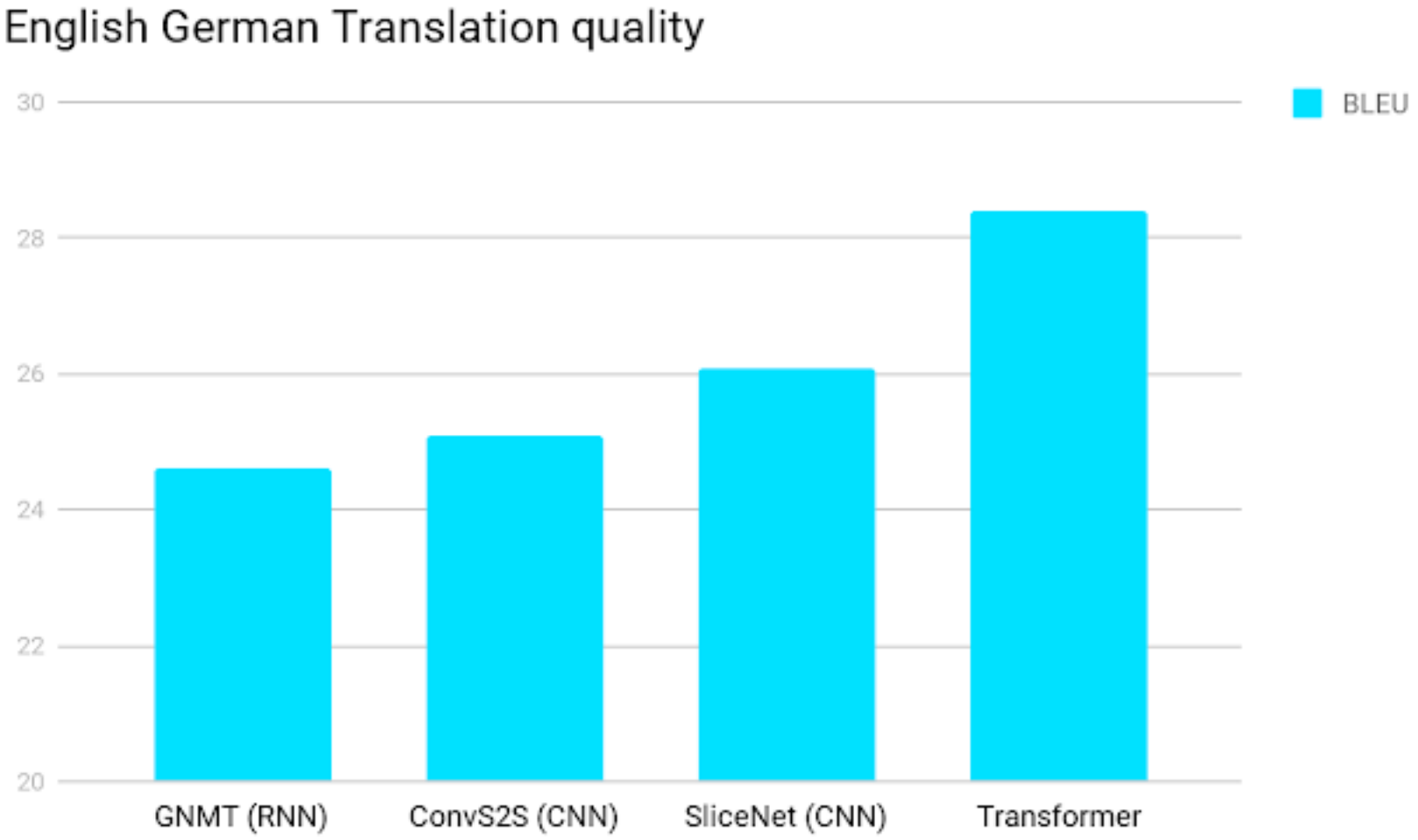
# Attention is a *general* Deep Learning technique

- More general definition of attention
  - Given a set of vector *values*, and a vector *query*, **attention** is a technique to compute a weighted sum of the values, dependent on the query.
  - We sometimes say that the query *attends to* the values
  - For example, in the seq2seq + attention model, each decoder hidden state *attends to* the encoder hidden states




https://distill.pub/2016/augmented-rnns/

# Attention is all you need? – Transformer Networks




https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

# Problems with RNNs = Motivation for Transformers

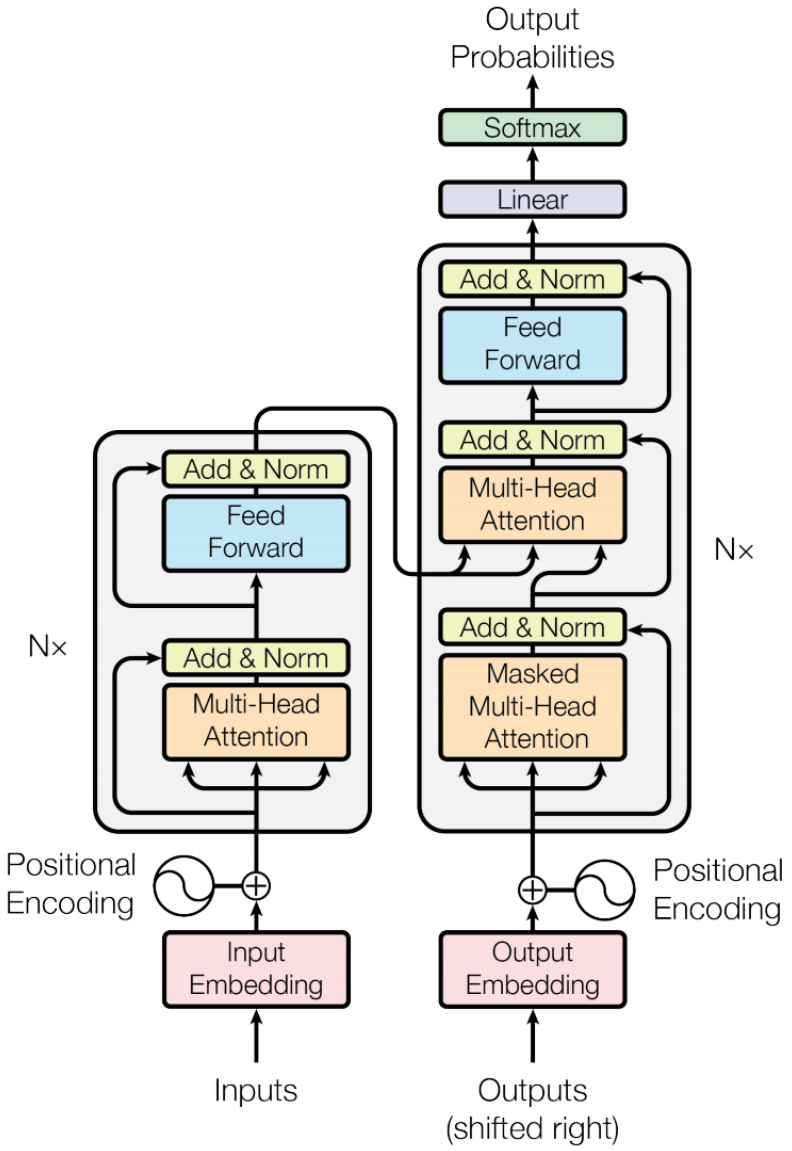
- Recurrent models typically factor computation along the symbol positions of the input and output sequences
  - Sequential computation prevents parallelization
  - Critical at longer sequence lengths, as memory constraints limit batching across examples

- Despite advanced RNNs like LSTMs, RNNs still need attention mechanism to deal with long range dependencies – path length for codependent computation between states grows with sequence

- But if attention gives us access to any state… maybe we don't need the RNN?

# Transformer Overview

- Sequence-to-sequence
- Encoder-Decoder
- Task: machine translation with parallel corpus
- Predict each translated word
- Final cost/error function is standard cross-entropy error on top of a softmax classifier




https://arxiv.org/pdf/1706.03762.pdf

# Transformer Basics

- Let’s define the basic building blocks of transformer networks first: new attention layers!

# Dot-Product Attention (Extending our previous def.)

- Inputs: a query q and a set of key-value (k-v) pairs to an output
  - Query, keys, values, and output are all vectors
- Output is weighted sum of values, where
  - Weight of each value is computed by an inner product of query and corresponding key
  - Queries and keys have same dimensionality

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

# Dot-Product Attention – Matrix notation

- When we have multiple queries q, we stack them in a matrix Q:

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

- Becomes: $A(Q, K, V) = softmax(QK^T)V$

$[|Q| \times d_k] \times [d_k \times |K|] \times [|K| \times d_v]$

softmax row-wise

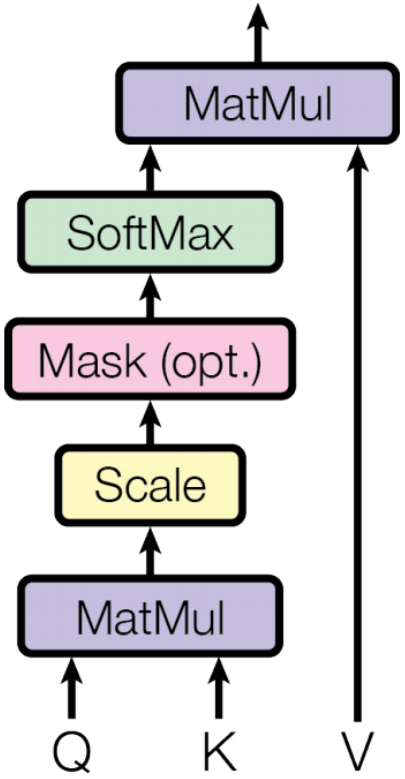
$= [|Q| \times d_v]$

# Scaled Dot-Product Attention

- Problem: As $d_k$ gets large, the variance of $q^T k$ increases → some values inside the softmax get large → the softmax gets very peaked → hence its gradient gets smaller.

- Solution: Scale by length of query/key vectors:

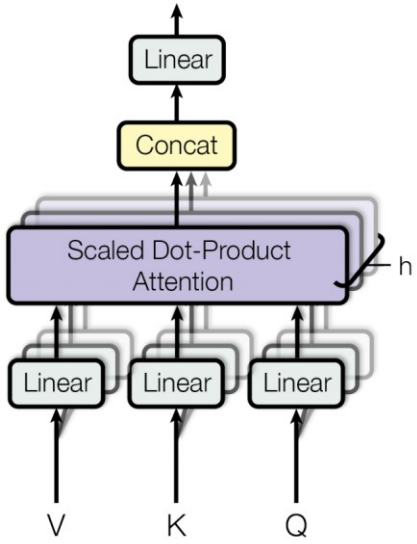
$$A(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



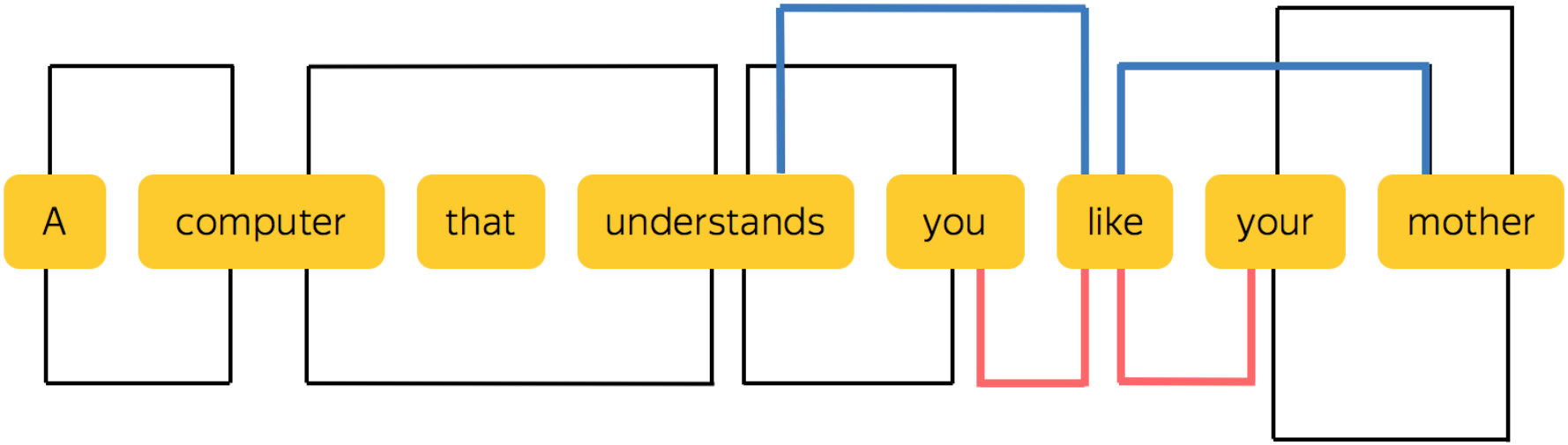
# Self-attention and Multi-head attention

- The input word vectors could be the queries, keys and values
  - In other words: the word vectors themselves select each other
  - Word vector stack = Q = K = V
- Problem: Only one way for words to interact with one-another
- Solution: Multi-head attention
  - First map Q, K, V into h many lower dimensional spaces via W matrices
  - Then apply attention, then concatenate outputs and pipe through linear layer

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\textbf{where } \text{head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



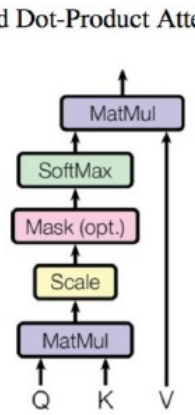
# Self-Attention

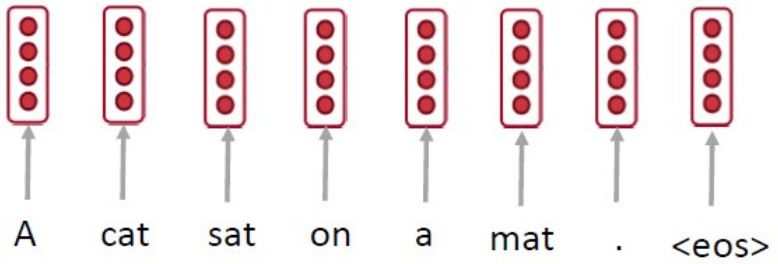



(example and picture from David Talbot)

# Self-attention: A Running Example

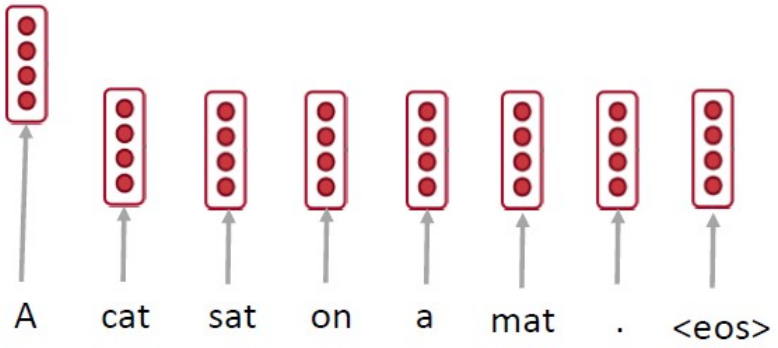
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

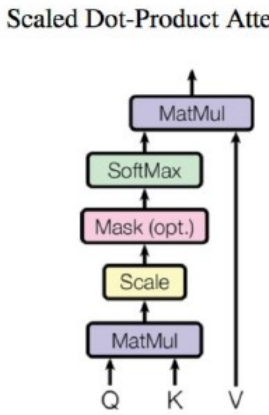


# Self-attention: A Running Example

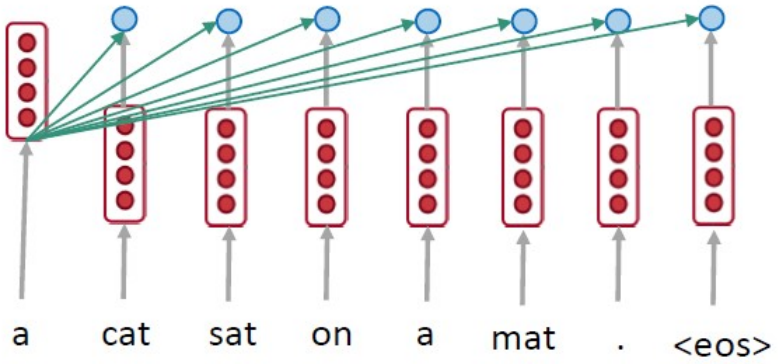
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

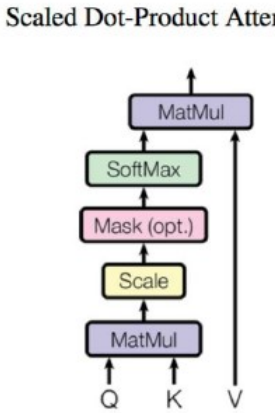


# Self-attention: A Running Example

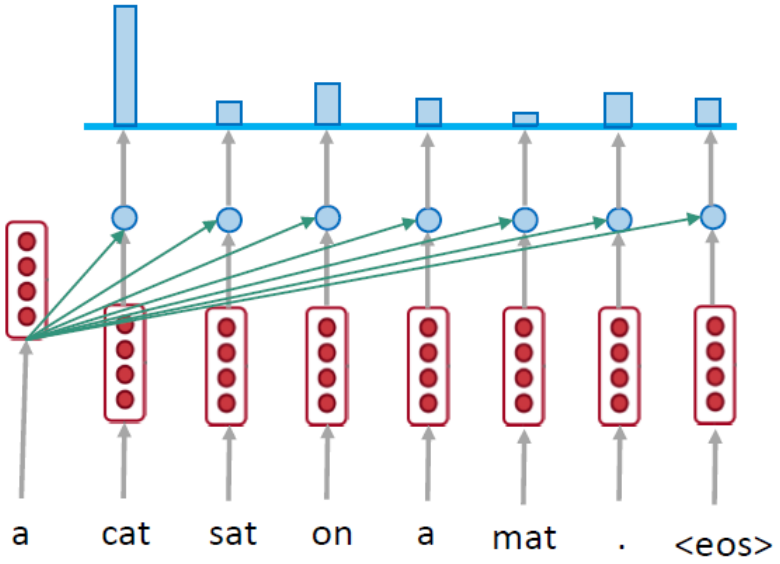
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

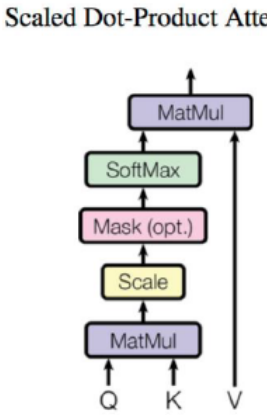


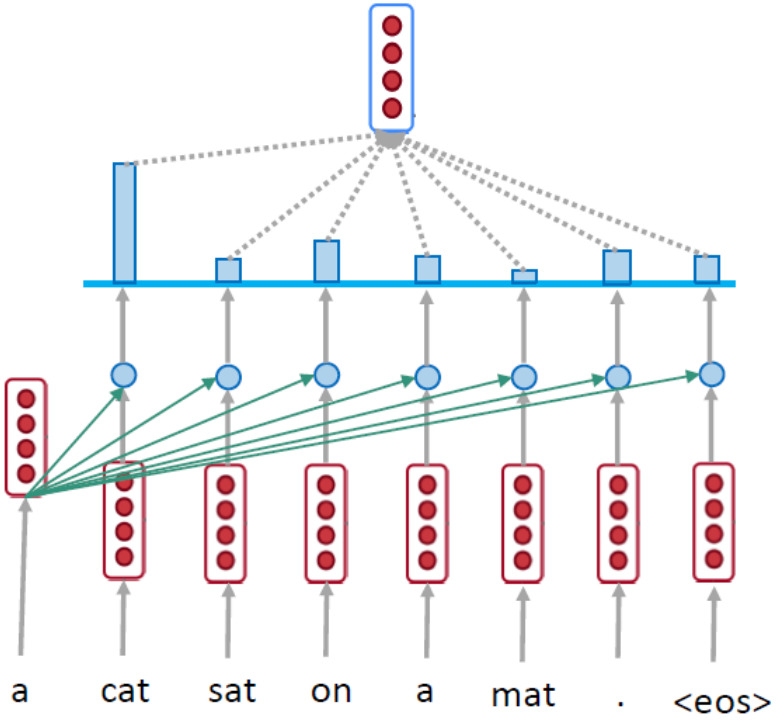
# Self-attention: A Running Example

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



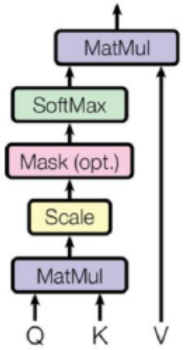



# Self-attention: A Running Example



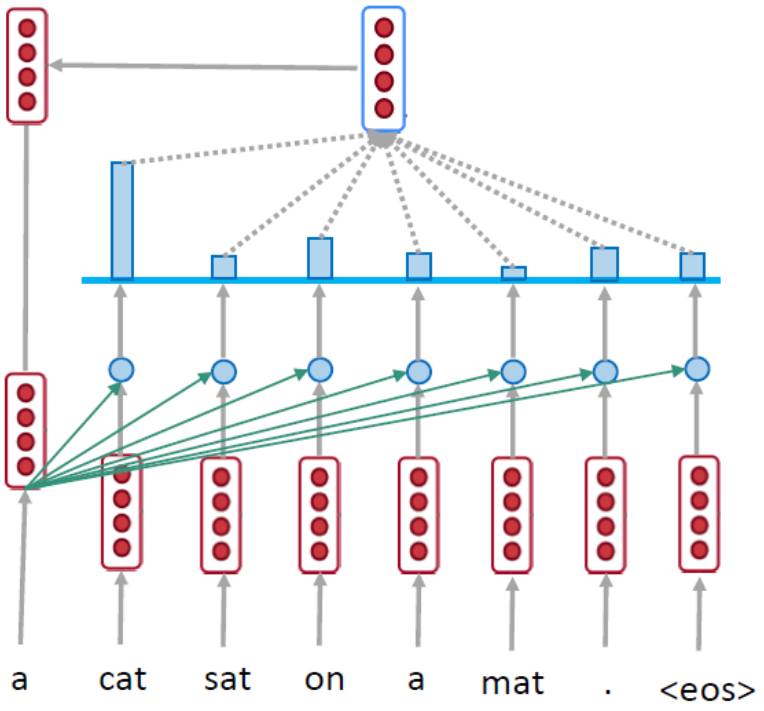

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention



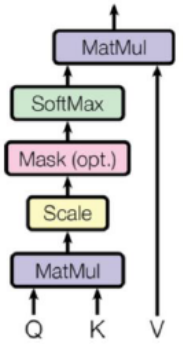
# Self-attention: A Running Example

update representation for the word "a"



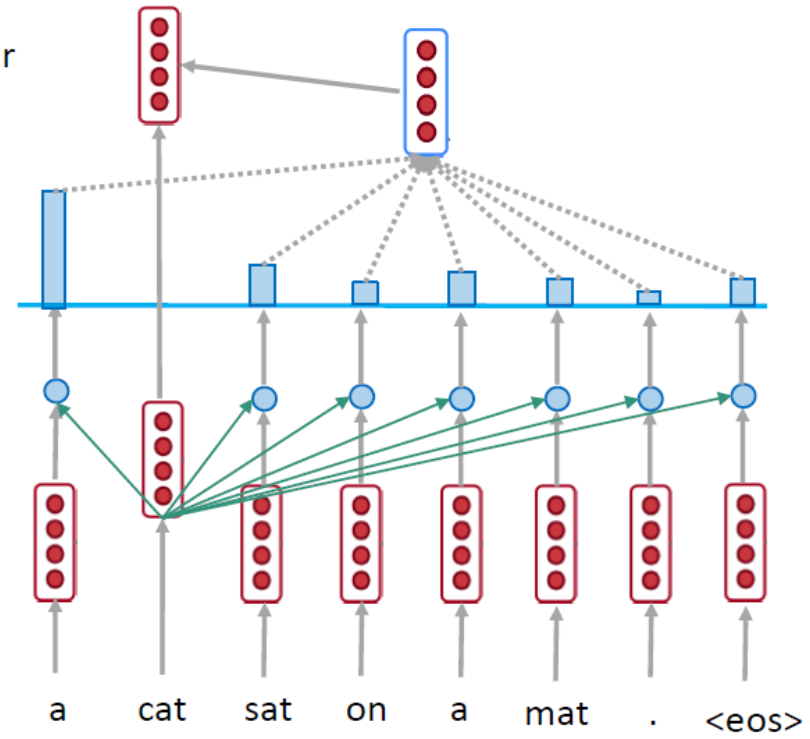

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention



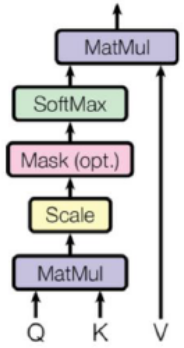
# Self-attention: A Running Example

update representation for the word "cat"



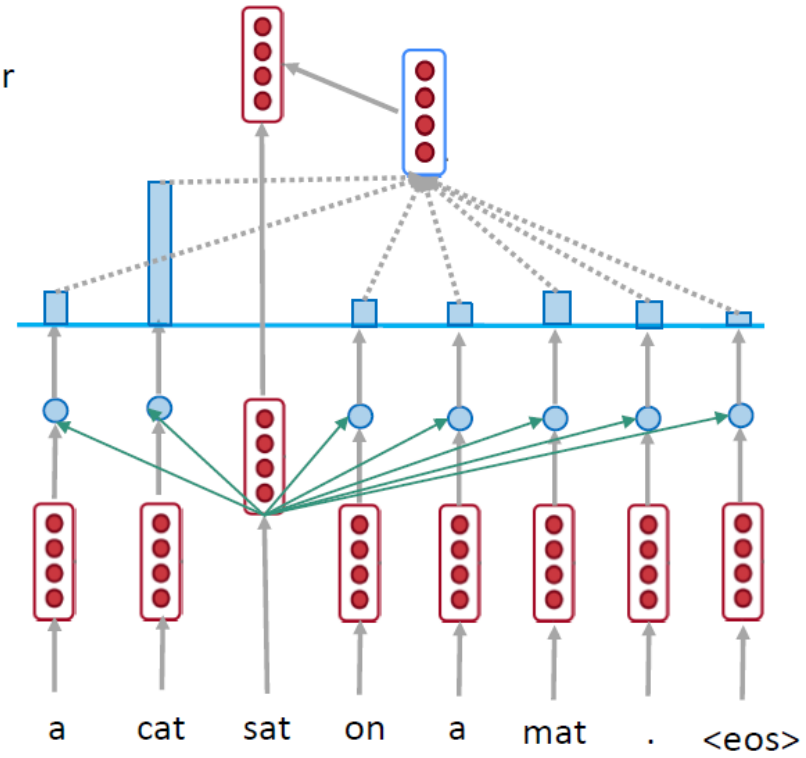

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention



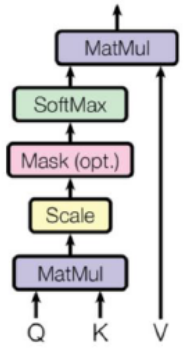
# Self-attention: A Running Example

update representation for the word "sat"



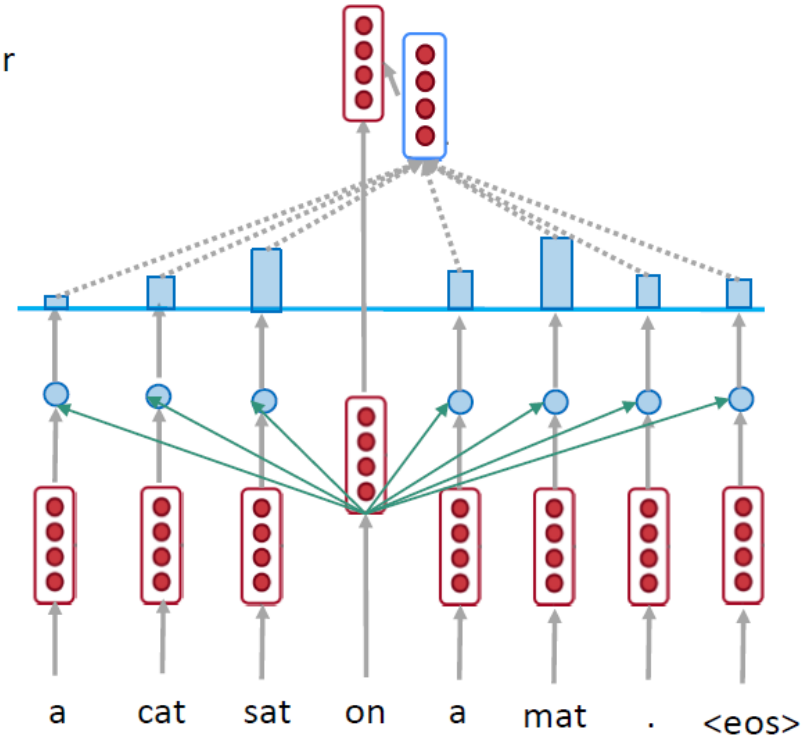

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention

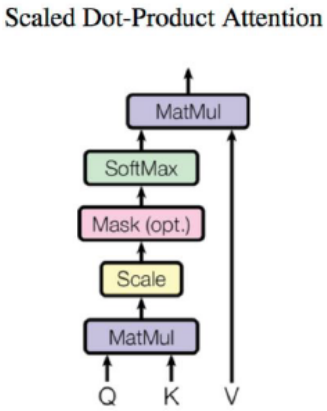


# Self-attention: A Running Example

update representation for the word "on"

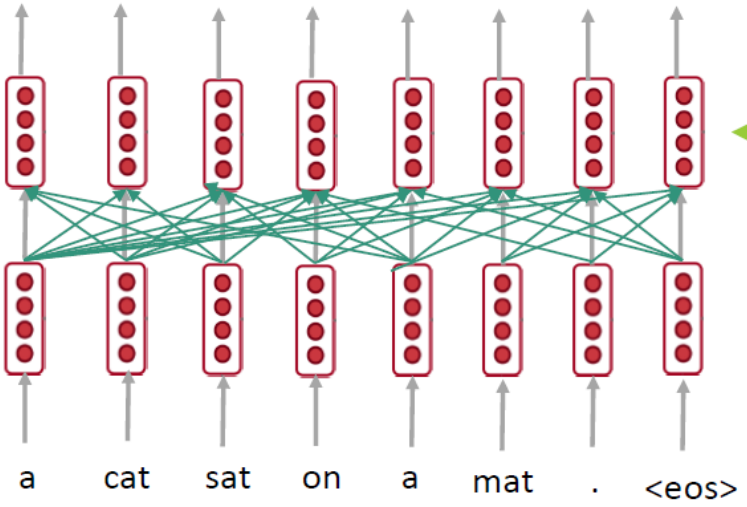



$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$



# Self-attention: A Running Example

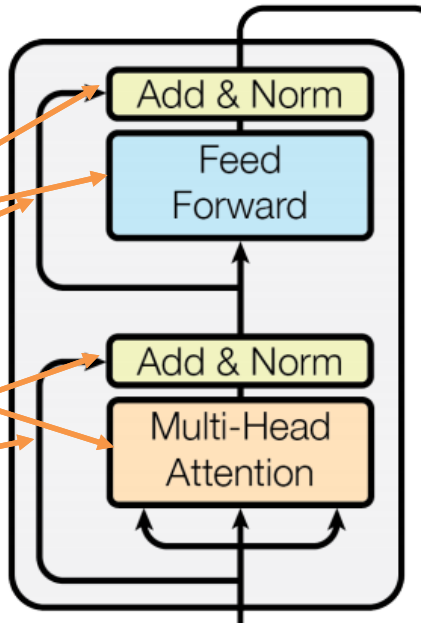
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$




Updated representations for each word (one layer)

# Complete transformer block

- Each block has two "sublayers"
  - 1. Multihead attention
  - 2. 2 layer feed-forward Nnet (with relu)

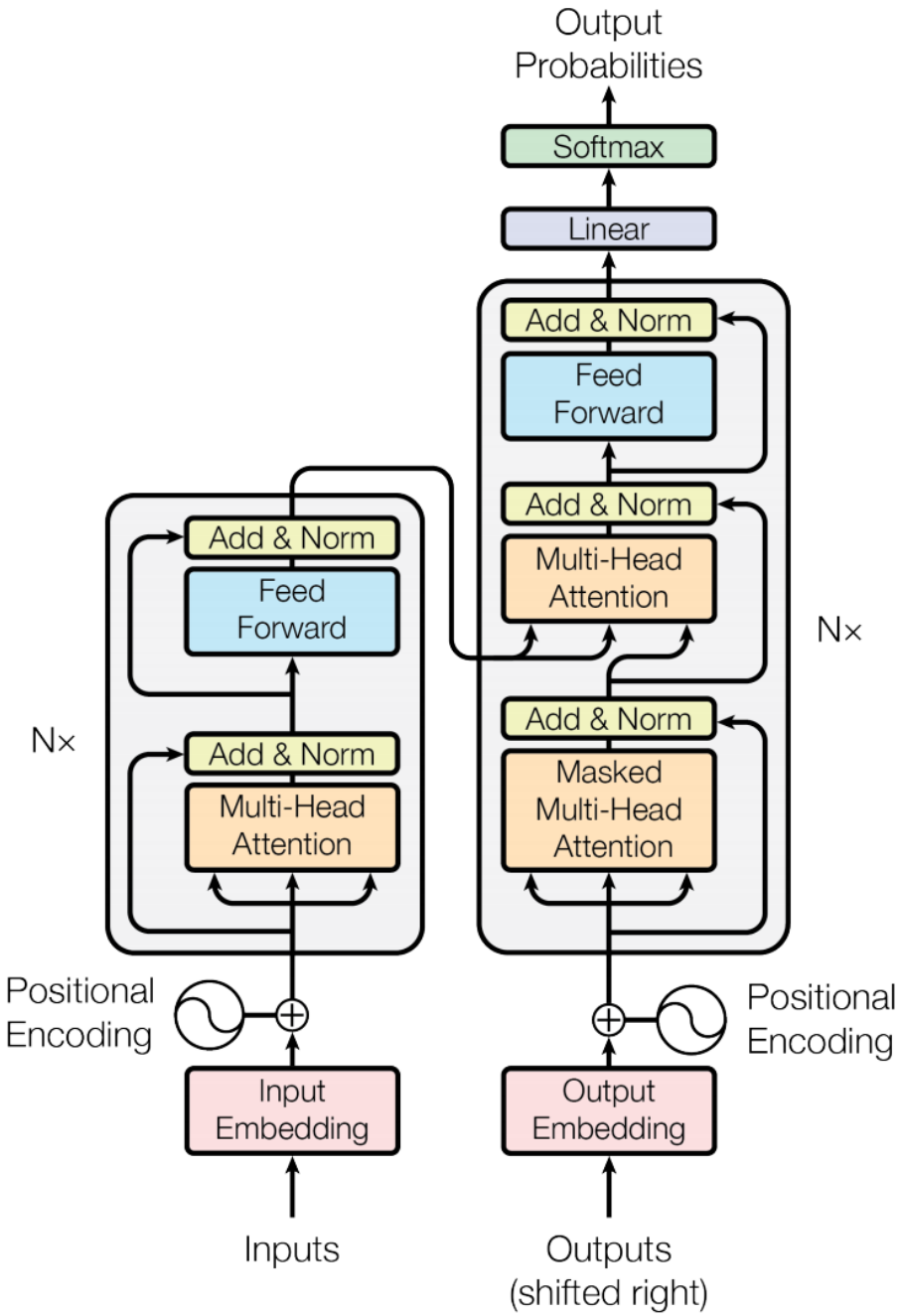



- Each of these two steps also has:
  - Residual (short-circuit) connection and LayerNorm:
  - LayerNorm(x + Sublayer(x))
    - Layernorm changes input to have mean 0 and variance 1, per layer and per training point (and adds two more parameters)

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2} \qquad h_i = f\left(\frac{g_i}{\sigma_i}\left(a_i - \mu_i\right) + b_i\right)$$

Layer Normalization by Ba, Kiros and Hinton, https://arxiv.org/pdf/1607.06450.pdf

Blocks are repeated
N=6 times

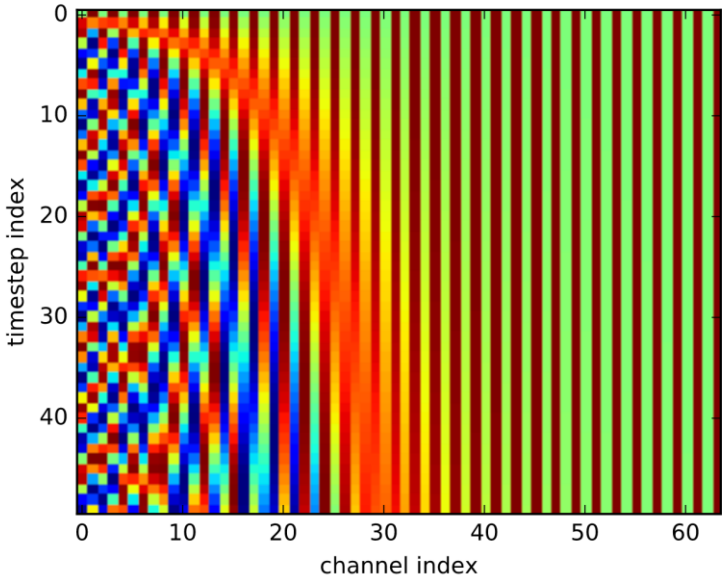


# Encoder Input

- Actual word representations are byte-pair encodings
  - Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural Machine Translation of Rare Words with Subword Units. ACL 2016.
- Added is a positional encoding so same words at different locations have different overall representations:

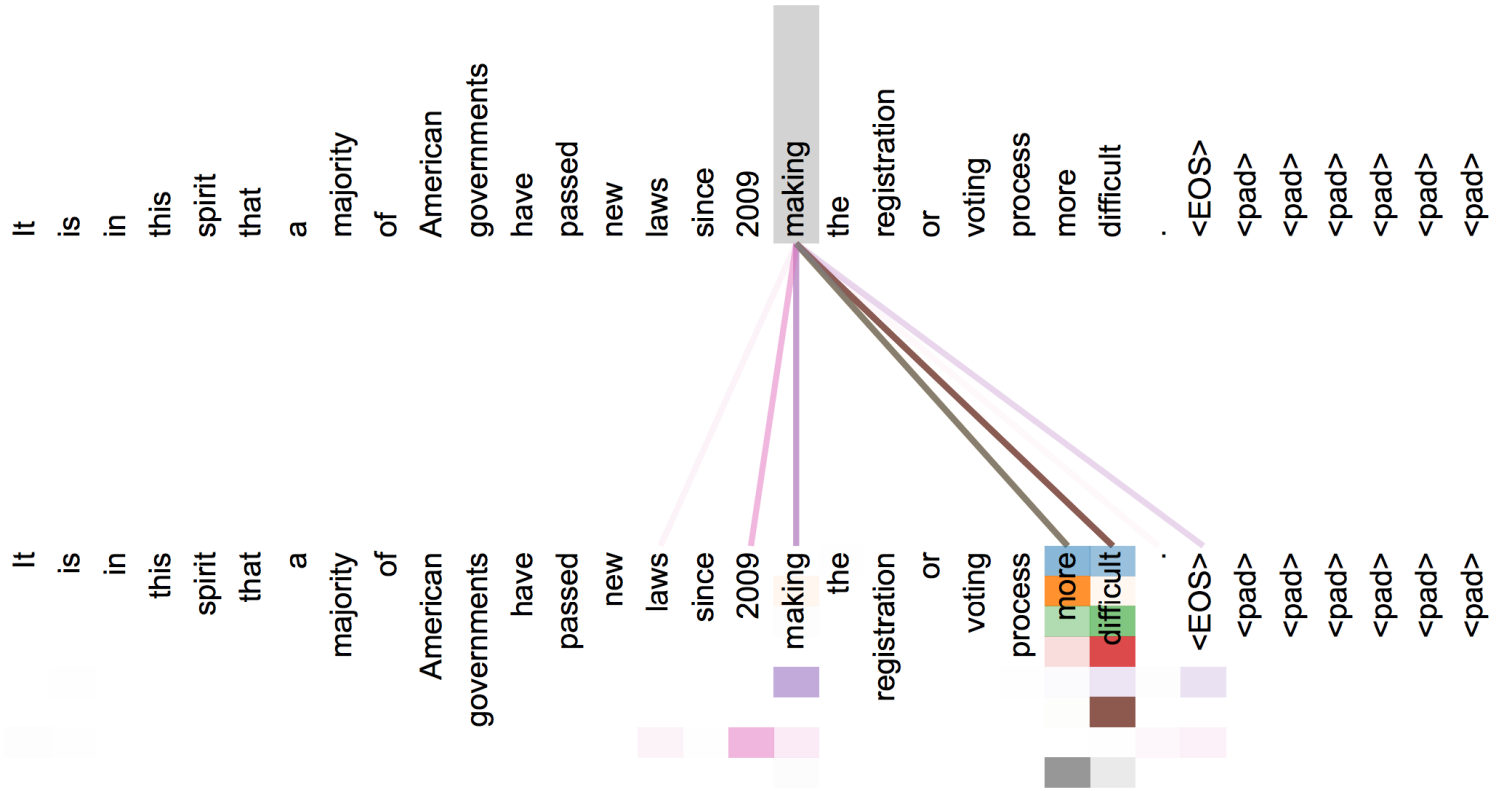
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

*pos* is the position of a word
*i* is the dimension index



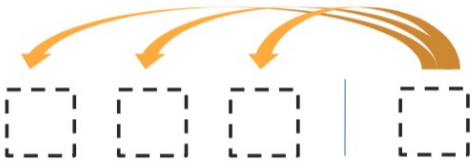
# Self Attention Visualization in Layer 5

- Words start to pay attention to other words in sensible ways



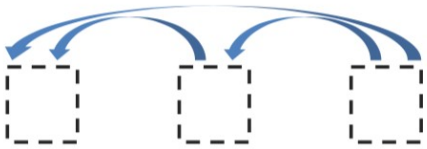
# Transformer Decoder

- 2 sublayer changes in decoder
- Masked decoder self-attention
  - Only depends on previous words
- Encoder-Decoder Attention
  - Queries come from previous decoder layer and keys and values come from output of encoder

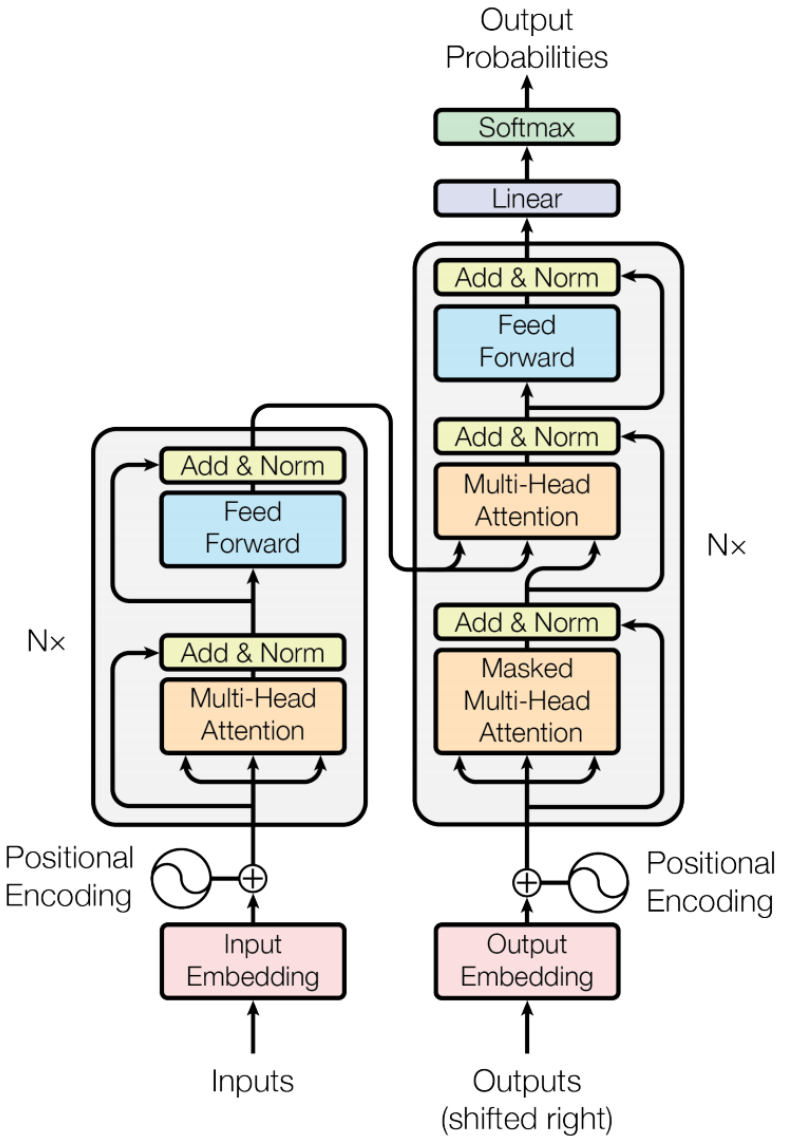


Encoder-Decoder Attention

Encoder Self-Attention



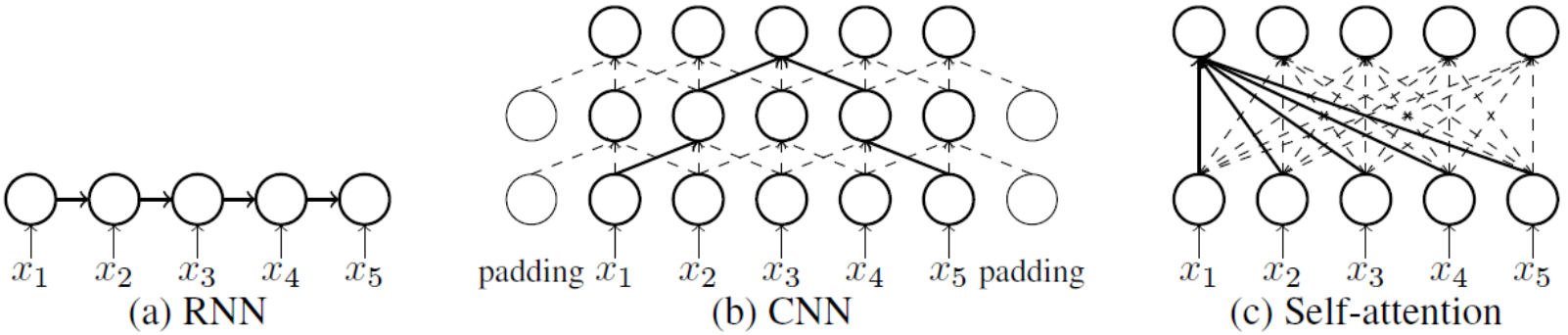
MaskedDecoder Self-Attention



# Advantages

- No recurrence: parallel encoding
- Fast training: both encoder and decoder are parallel
- No long range problem: O(1) for all tokens direct connections
- Three attentions: the model does not have to remember too much
- Multi-head attention allows to pay attention to different aspects

- Why self-attention and CNN is better than RNN on NMT is still under investigation




A comparison of RNN, CNN, and self-attention
http://aclweb.org/anthology/D18-1458

# Transformer Networks Visualization

https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

# Google's BERT

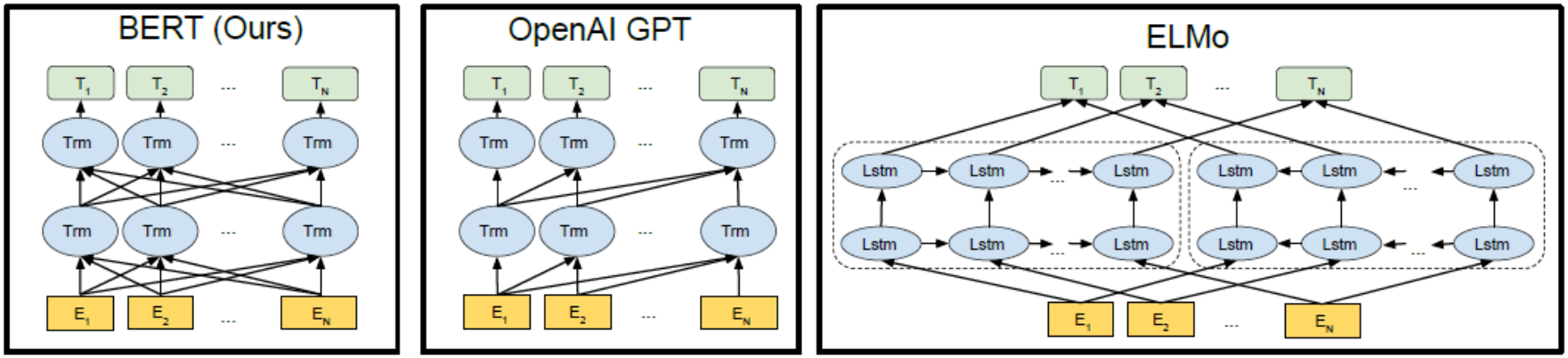



Figure 1: Differences in pre-training model architectures. BERT uses a bidirectional Transformer. OpenAI GPT uses a left-to-right Transformer. ELMo uses the concatenation of independently trained left-to-right and right-to-left LSTM to generate features for downstream tasks. Among three, only BERT representations are jointly conditioned on both left and right context in all layers.

- Training of BERT-BASE was performed on 4 Cloud TPUs in Pod configuration (16 TPU chips total).
- Training of BERT-LARGE was performed on 16 Cloud TPUs (64 TPU chips total).
- Each pre-training took 4 days to complete.

https://ai.googleblog.com/2018/11/open-sourcing-bert-state-of-art-pre.html
https://arxiv.org/pdf/1810.04805.pdf