# Session 8: Attention

**Korbinian Riedhammer**

# …welcome back from the break!

- RNN/LSTM help with several sequence problems:
  - *many-to-one:* apply linear layer on last output/hidden state
  - *many-to-many:* apply linear layer at each time step
  - *one-to-many:* yes, you can just keep sampling from an RNN :-)
  - …how about many-to-many with $M \neq N$ ?

# Today's Menu

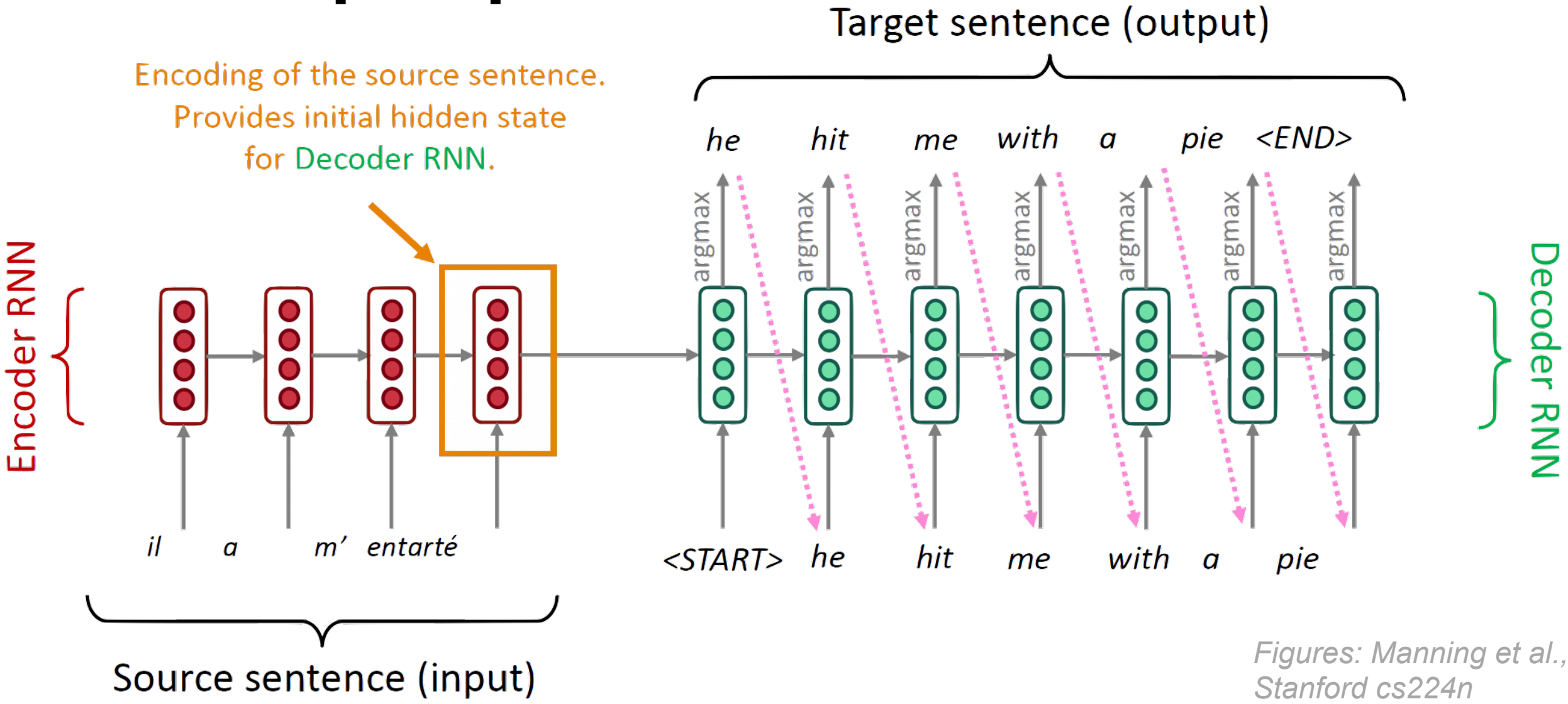
- Neural Machine Translation
    - encoder/decoder architecture
    - sampling from decoders: beam search
- Attention: a better modeling of context
- Self-Attention: getting rid of recurrence
- Transformer architecture

# Neural Machine Translation

- Data: (huge) “parallel corpus”, e.g. European parliament
- Sequence-to-sequence problem, with
  - complex dependencies (word order, sex/gender, …)
  - m:n relations: phrase translations may have different lengths
  - …all trained “end to end” with two RNNs


**encoder RNN** **decoder RNN**

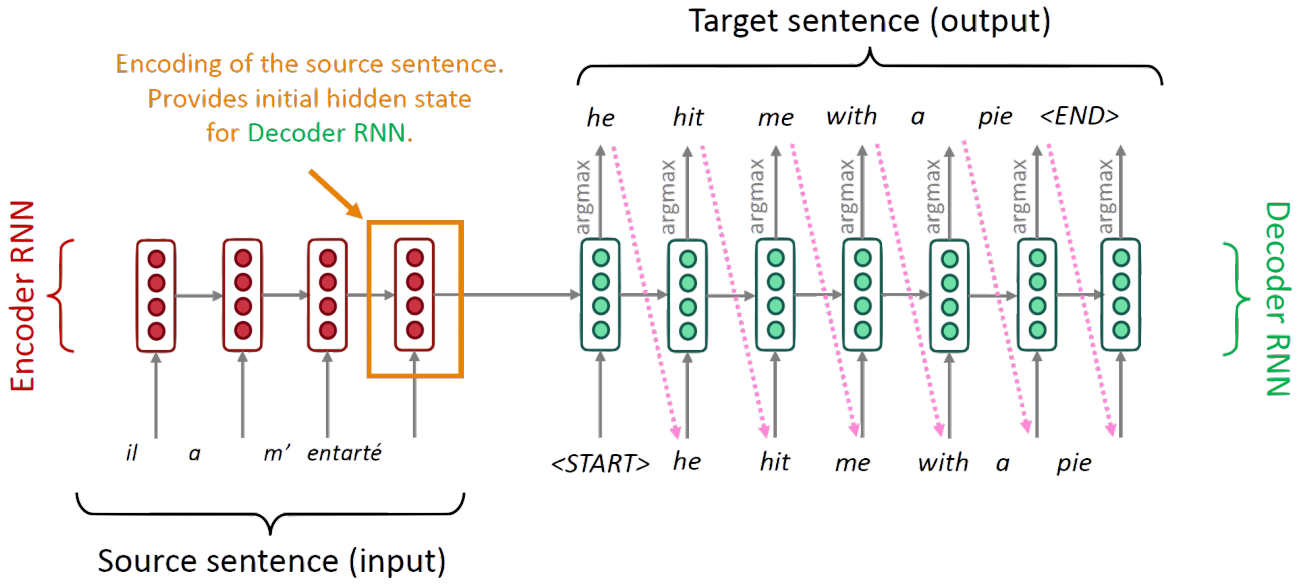
# NMT seq2seq Architecture




*Figures: Manning et al., Stanford cs224n*

# NMT seq2seq Architecture

- Encoder RNN *consumes* input and produces an overall encoding
- Decoder RNN uses encoding as initial hidden state, and *generates* the target sentence




- "end-to-end": complete task modeled as one large network
  - at training time: apply Teacher-Forcing on Decoder outputs
  - at test time: use output at $t$ as input at $t+1$

*Figures: Manning et al., Stanford cs224n*

# Seq2Seq is versatile!

- Summarization: long text → short text
- Dialog: previous utterance → next utterance
- Parsing: input text → parse tree
- Code generation: natural language → python
- Speech recognition: spoken word → written word
  (“Listen-Attend-Spell”, https://arxiv.org/abs/1508.01211)

# Encoder/Decoder

## Training

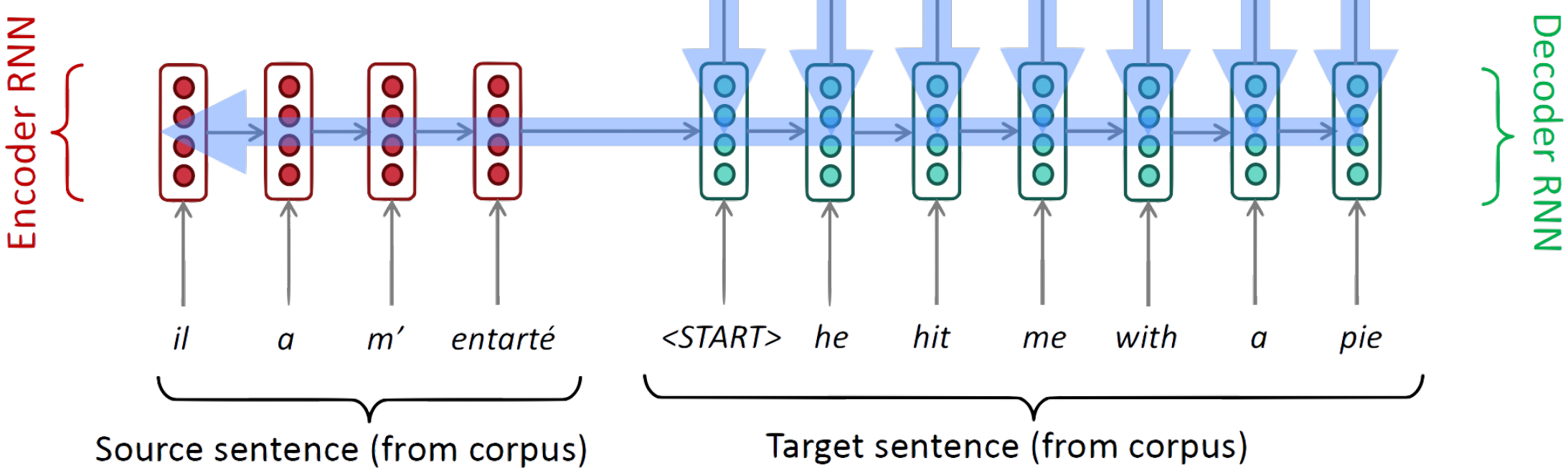
$$J = \frac{1}{T}\sum_{t=1}^{T} J_t \quad = \quad J_1 + J_2 + J_3 + J_4 + J_5 + J_6 + J_7$$

= negative log prob of "he"

= negative log prob of "with"

= negative log prob of <END>

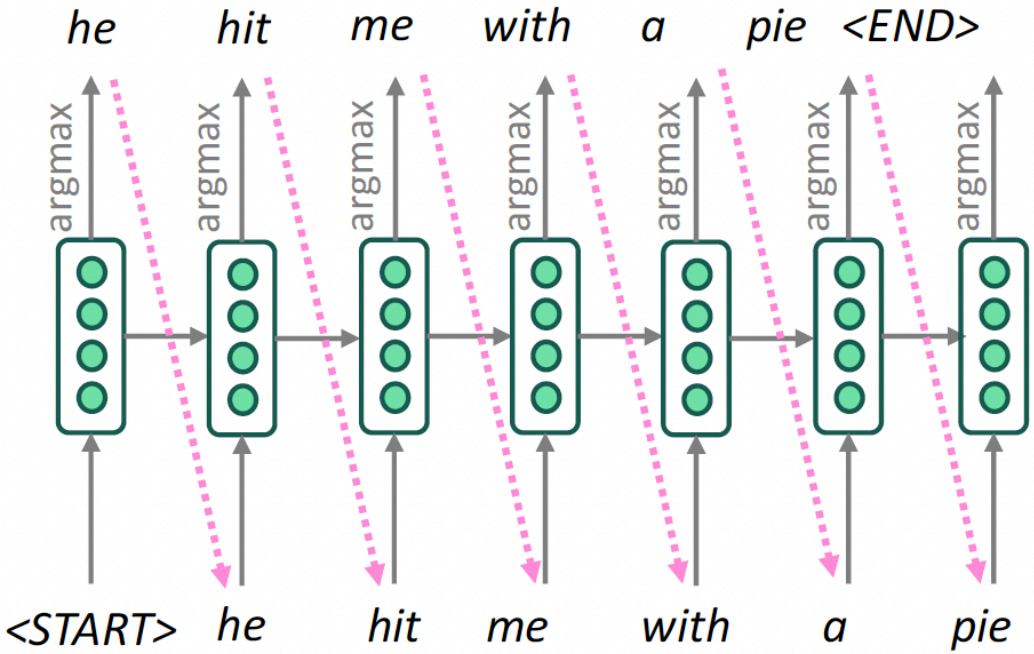
Back-Propagation operates "end-to-end"!




*Figs.: Manning et al., Stanford cs224n*

# Sampling the Decoder

- Initialize hidden state with last state of decoder
- Use special *start* and *end* symbols
- Greedy sampling:
  - Use output at time $t$ as input to time $t + 1$
  - Terminate on observing *end* token
  - …or on exceeding target length




*Figures: Manning et al., Stanford cs224n*

# Problems with Greedy Decoding

- Early decisions can “spoil” the best solution
    - frequently, the correct token is *not* ranked 1st
    - how to recover from wrong decisions?

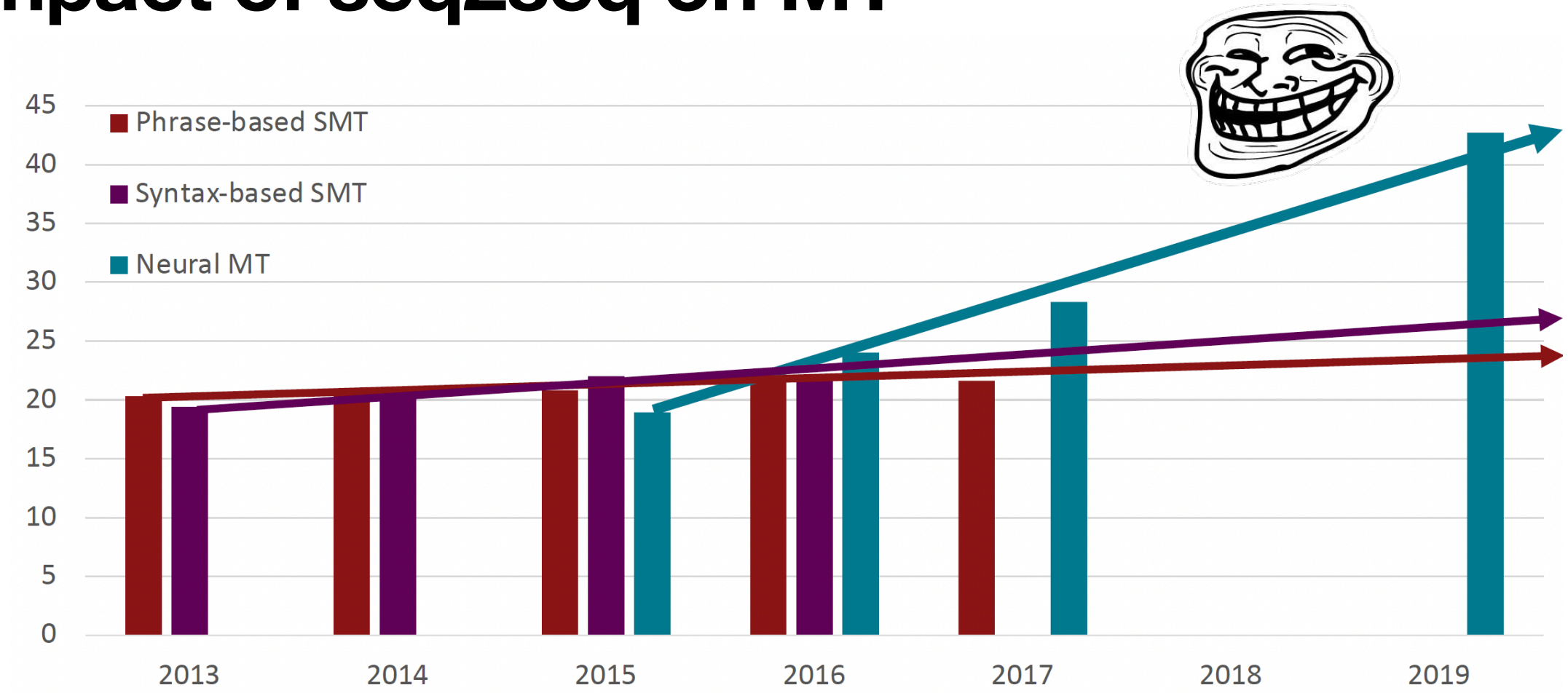
# Solution: Beam Search Decoding

- Core idea: instead of just going with the current best hypothesis, keep track of $k$ most probable partial results ("hypotheses")
  - Well-studied in AI (path finding) and speech recognition
  - The larger the $k$ (the "beam size"), the more paths needed be be kept active…which requires memory and compute time
  - When reaching *end*-symbol, keep exploring others and re-rank at the end (normalizing for length)

# Pros & Cons NMT

- Better performance then statistical MT , also in terms of
  - fluency
  - context utilization
  - phrase similarities
- Single neural network to train (statMT often complex system combination)
- …thus: less engineering effort
- *But:* less interpretable (and hard to debug), difficult to control

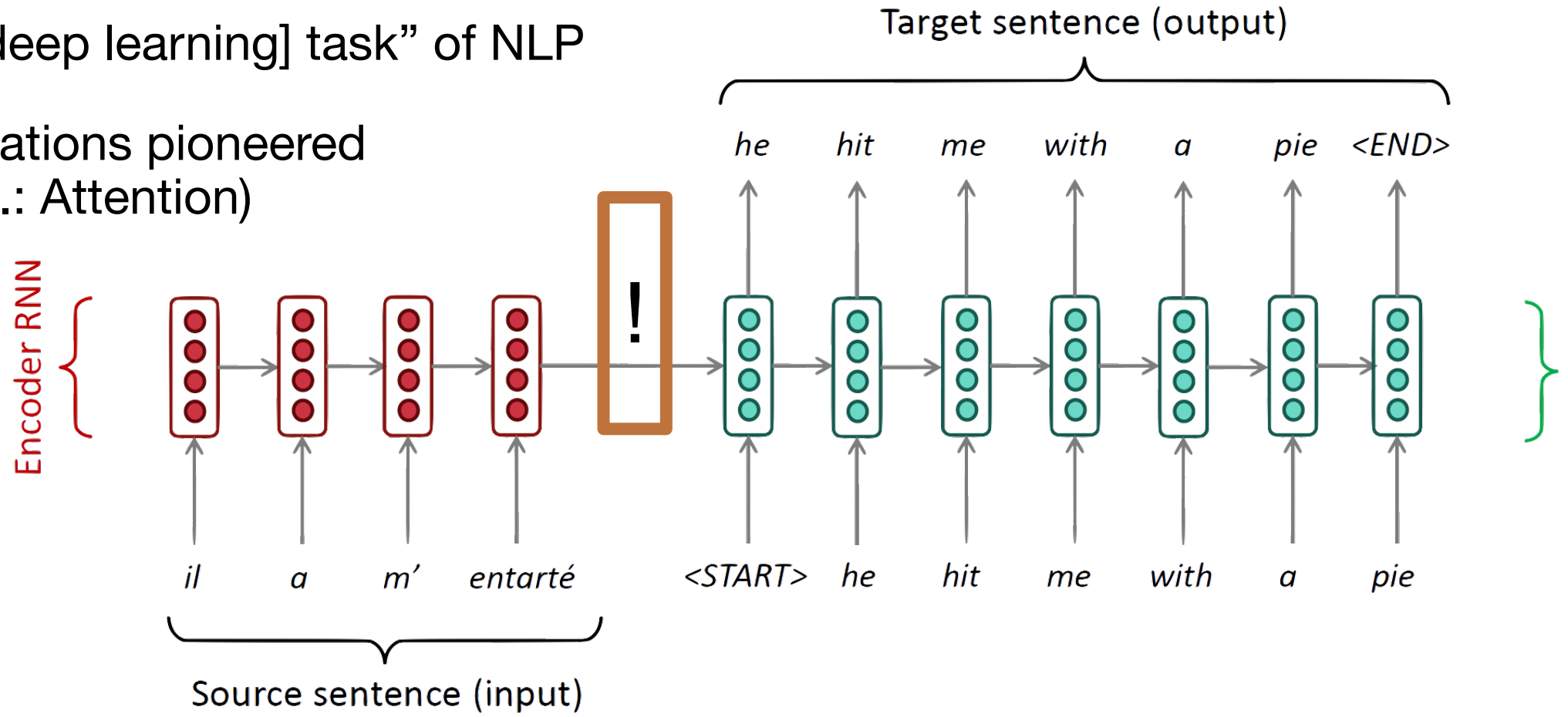
# Impact of seq2seq on MT




http://www.meta-net.eu/events/meta-forum-2016/slides/09_sennrich.pdf

# Where's the key issue in NMT (or: seq2seq…)

- "Flagship [deep learning] task" of NLP
- Many innovations pioneered in NMT (e.g.: Attention)

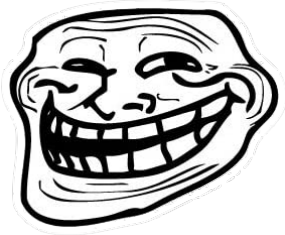



*Figures: Manning et al., Stanford cs224n*

**Problems with this architecture?**

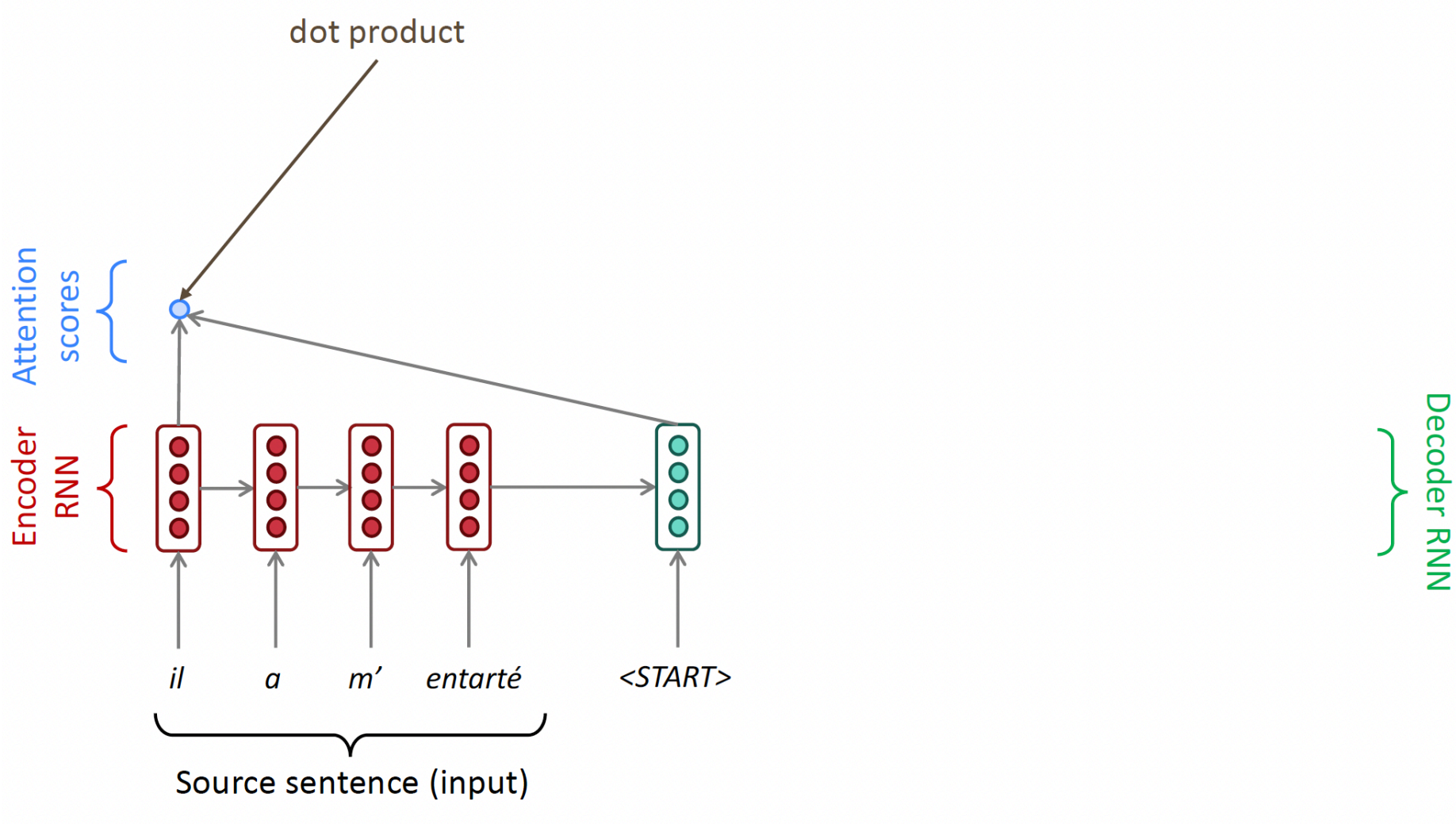
# Enter: Attention

- Bottleneck: single last hidden state is to encode all context



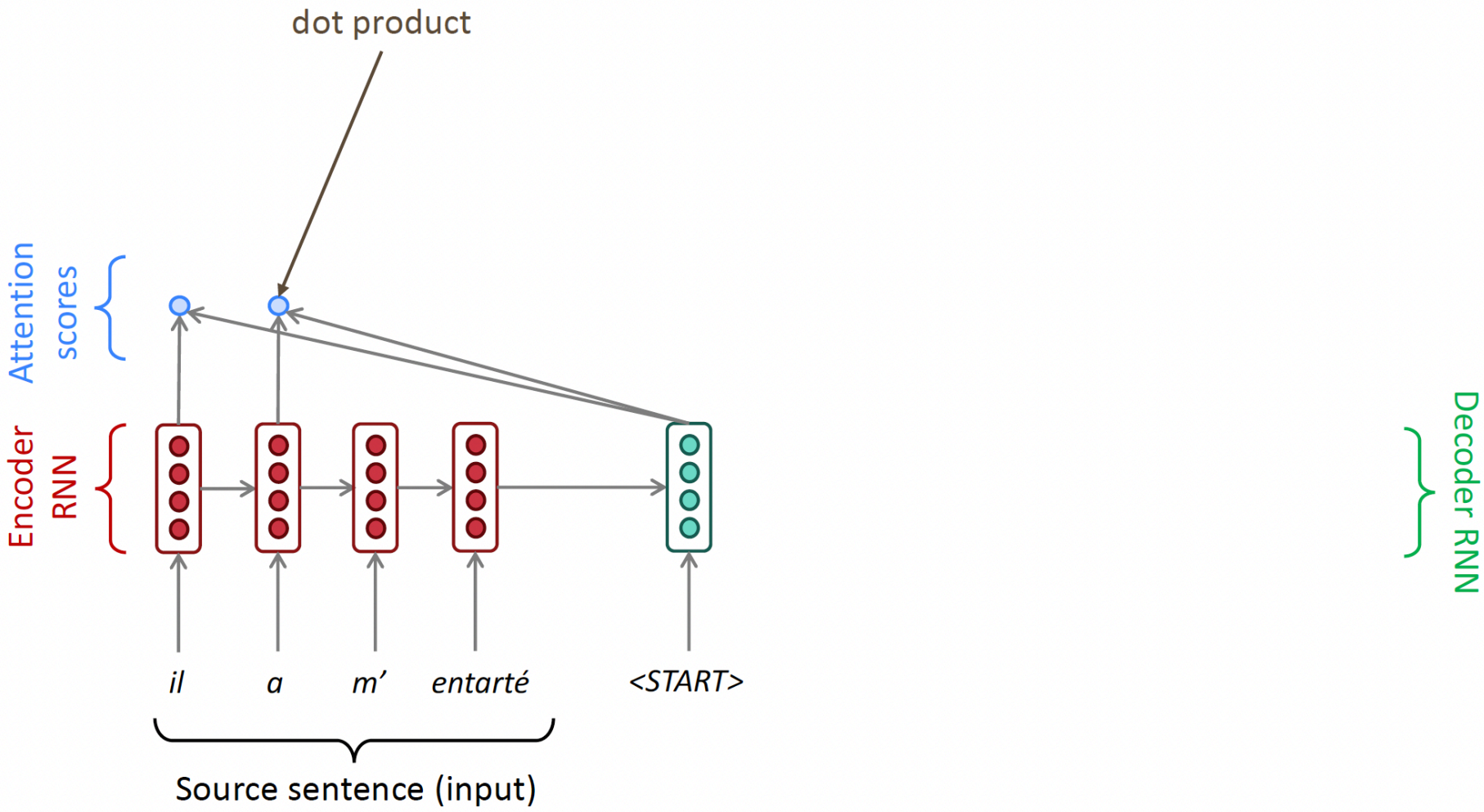
- Core idea: at each step at the decoder, use a direct connection to the encoder (states) to focus on particular parts of the input sequence

# Attention: visually



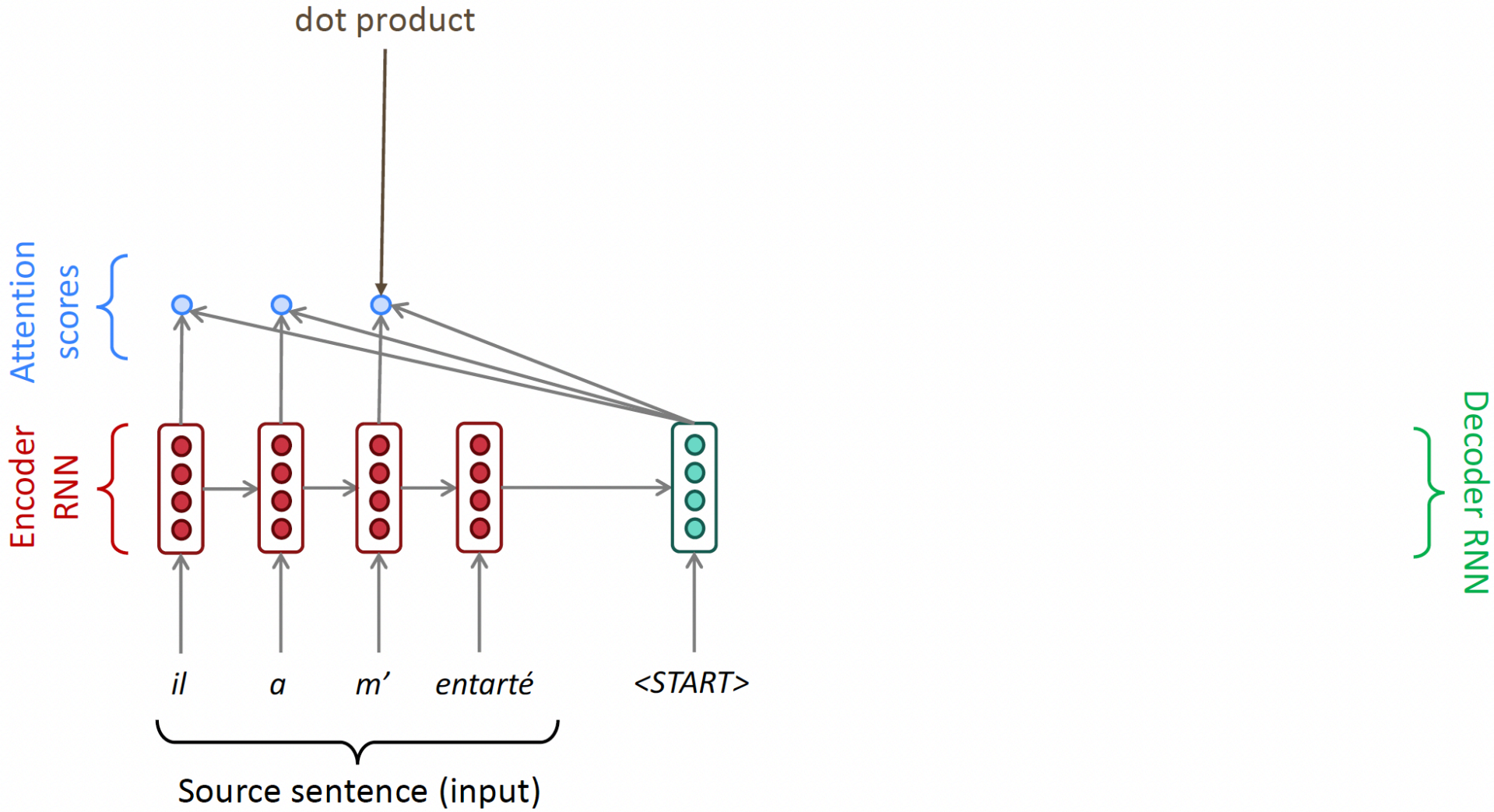

*Figures: Manning et al., Stanford cs224n*

# Attention: visually



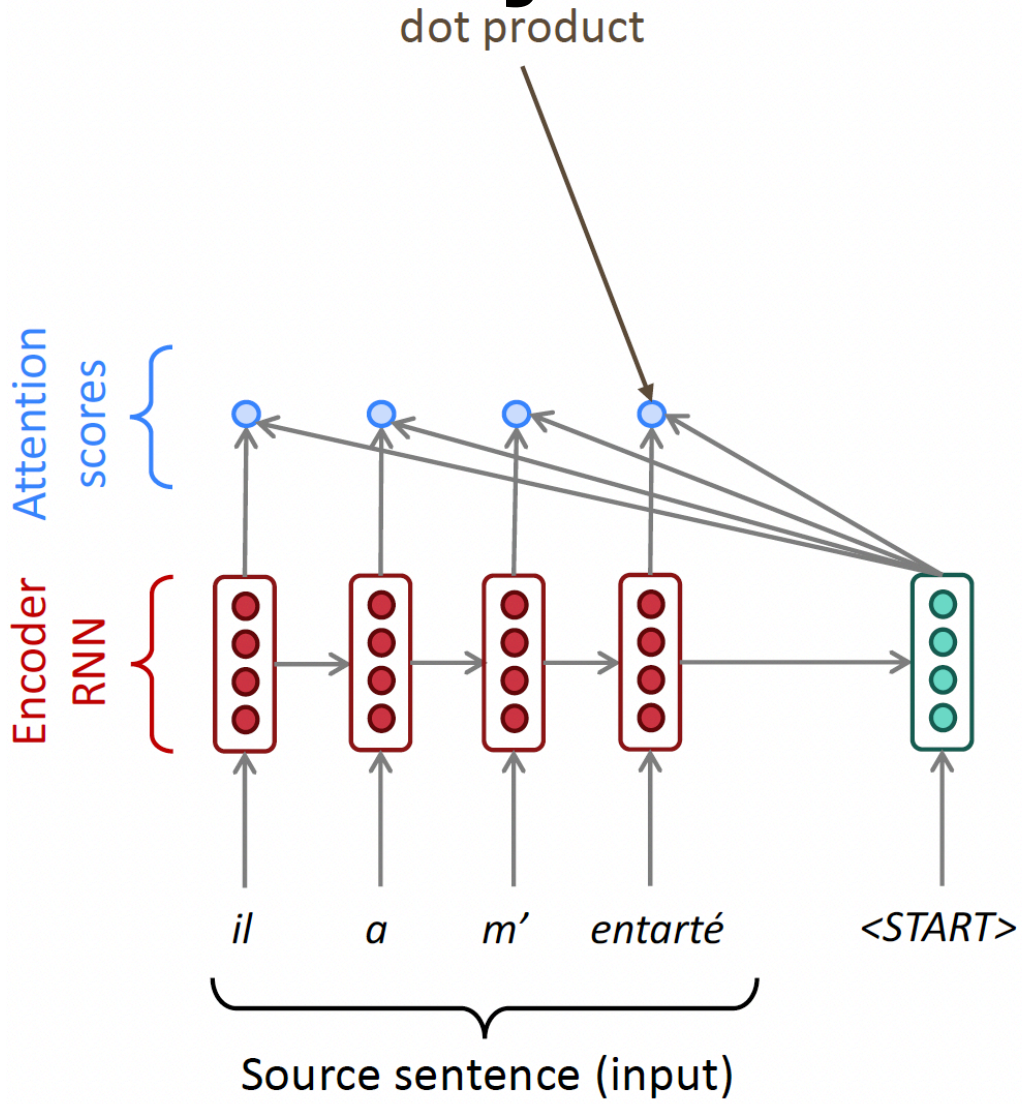

*Figures: Manning et al., Stanford cs224n*

# Attention: visually



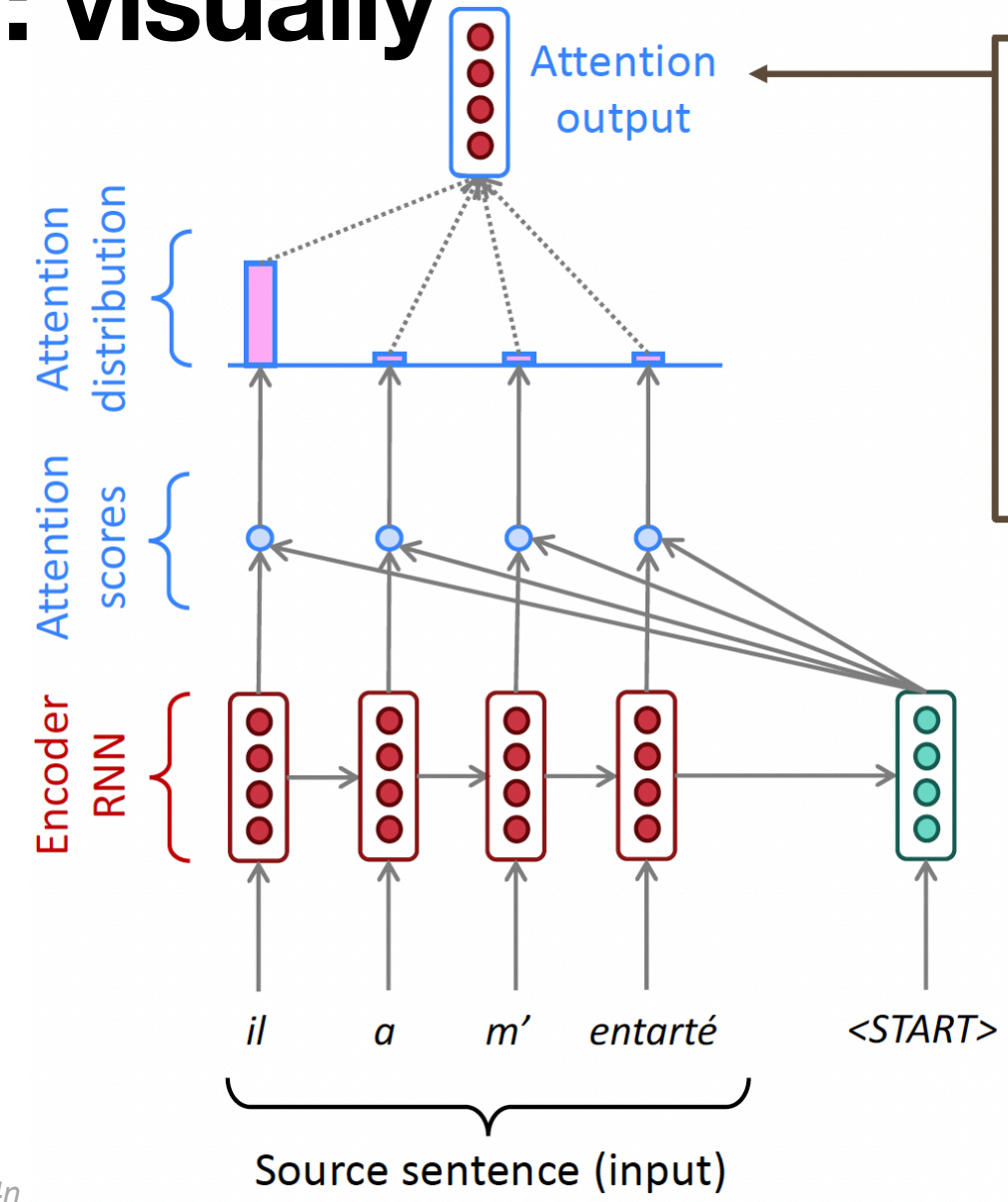

*Figures: Manning et al., Stanford cs224n*

# Attention: visually




*Figures: Manning et al., Stanford cs224n*

# Attention: visually

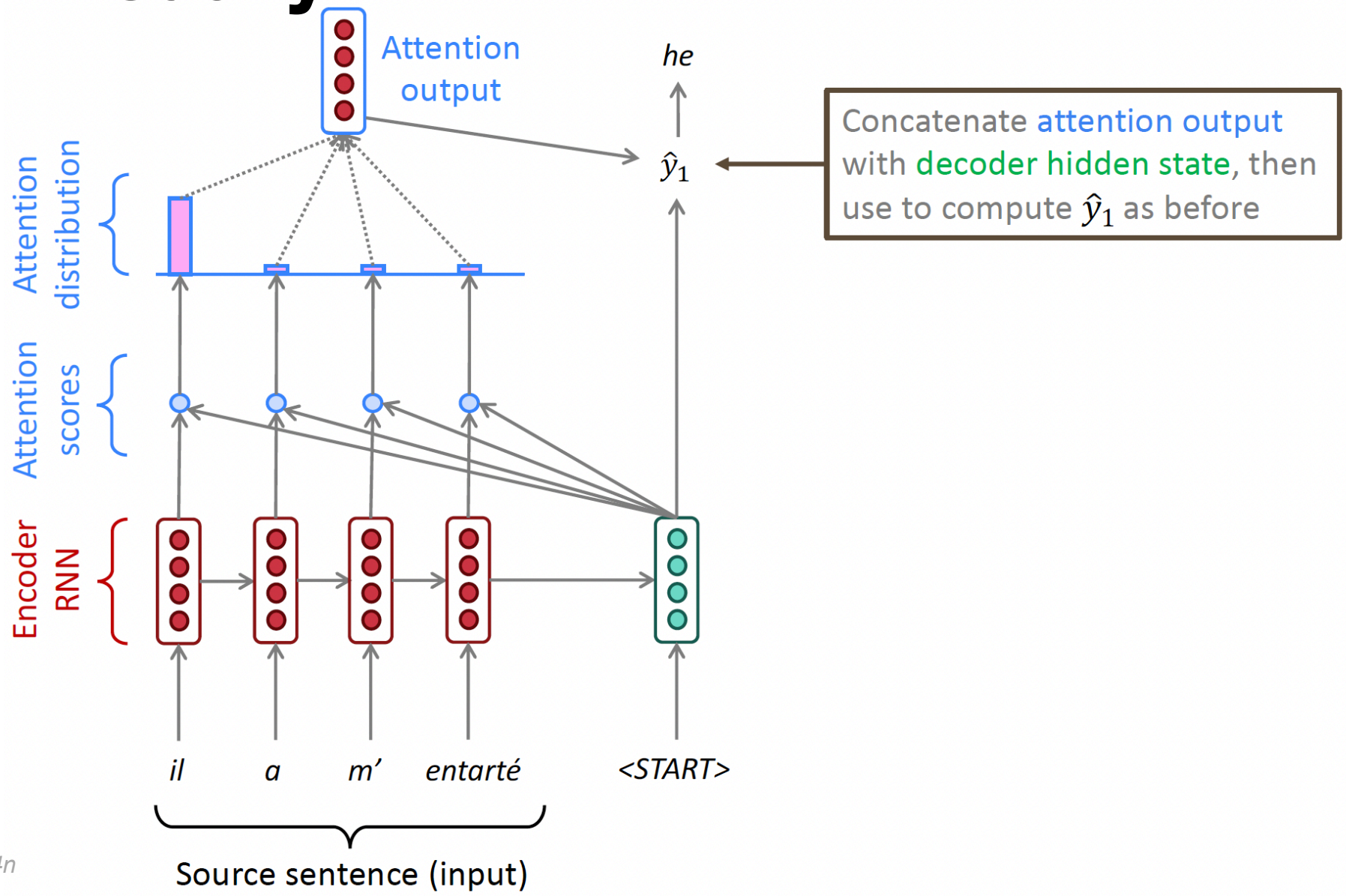



Use the attention distribution to take a **weighted sum** of the encoder hidden states.

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

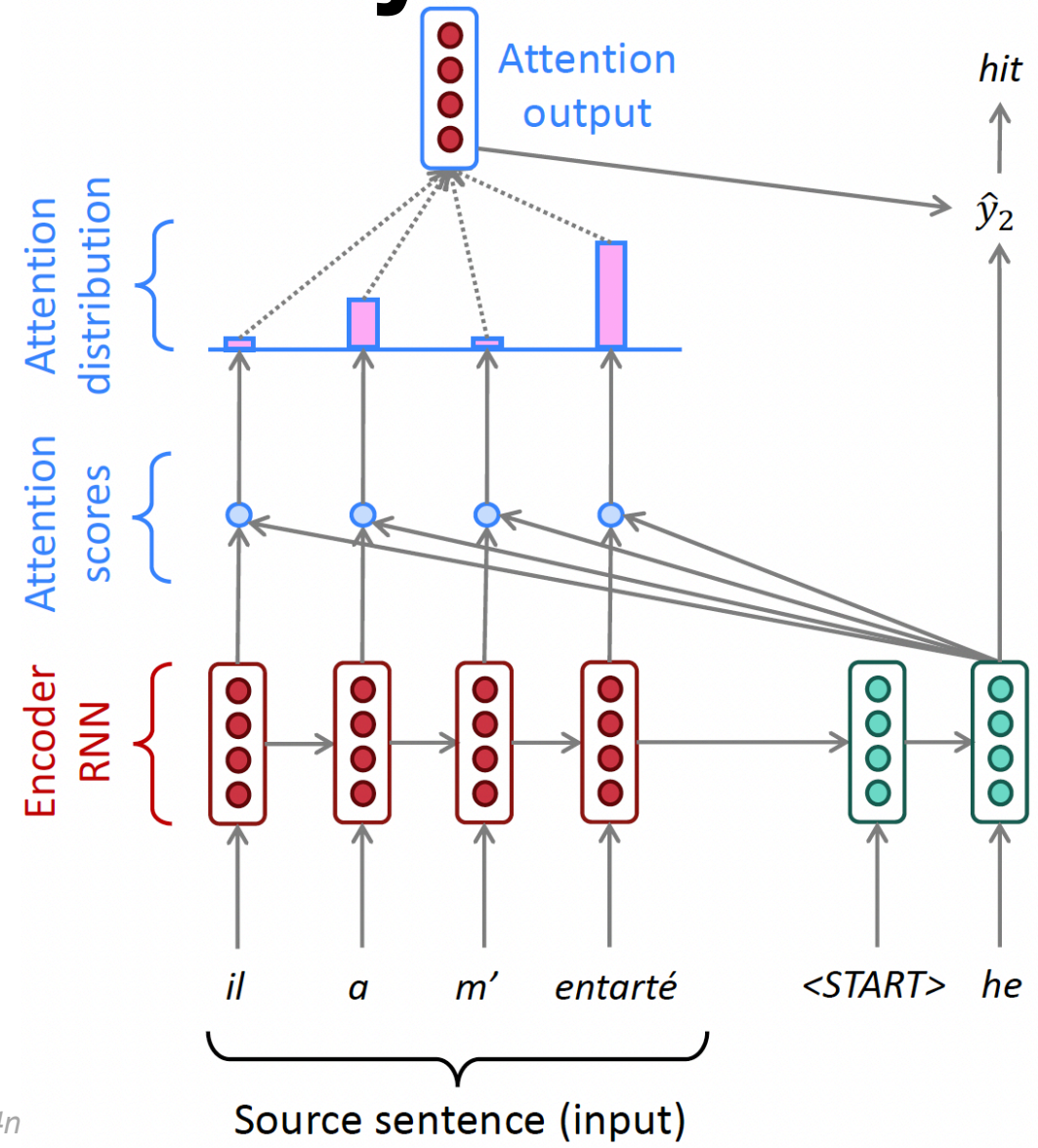
*Figures: Manning et al., Stanford cs224n*

# Attention: visually




*Figures: Manning et al., Stanford cs224n*

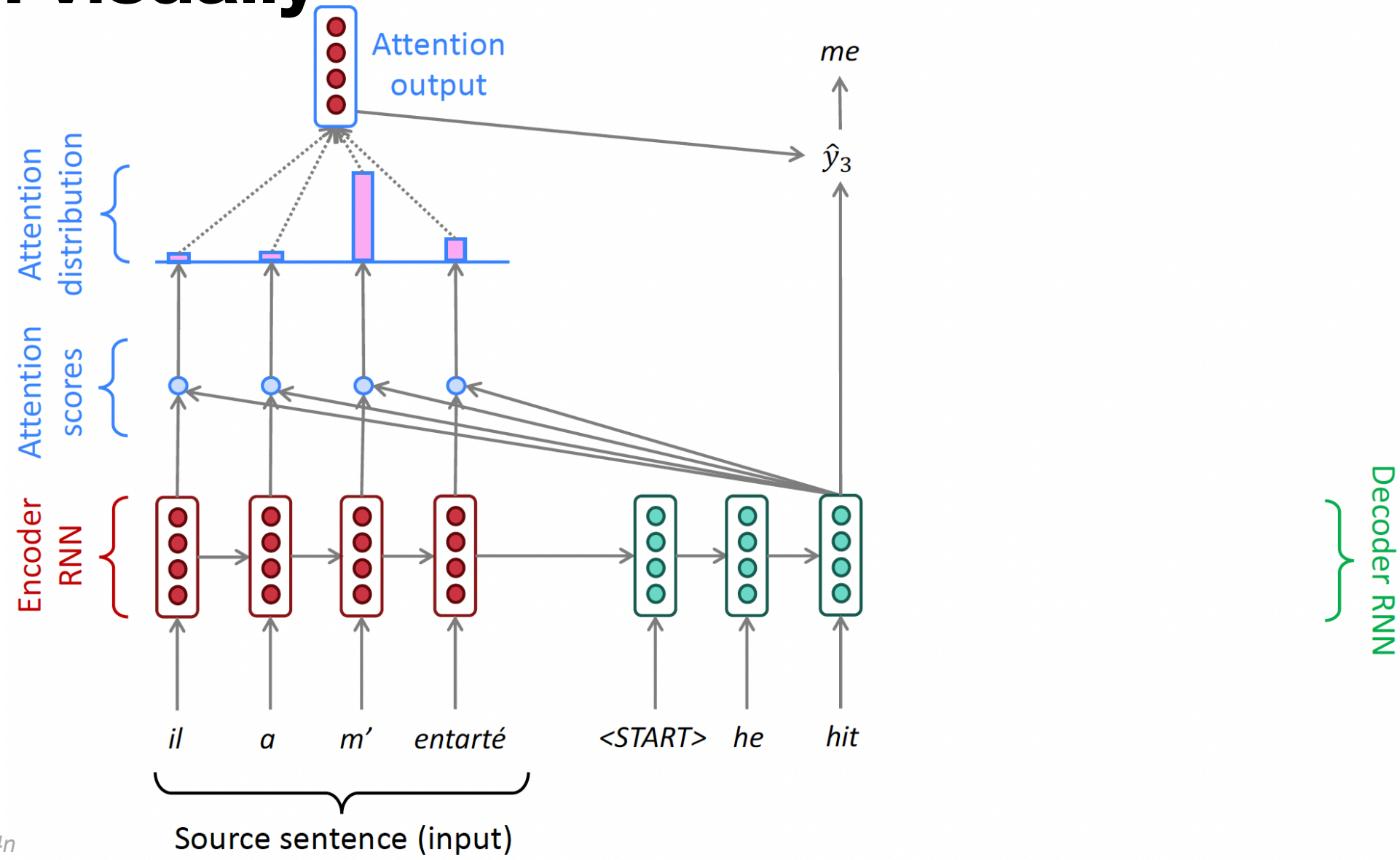
# Attention: visually




Sometimes we take the attention output from the previous step, and also feed it into the decoder (along with the usual decoder input).

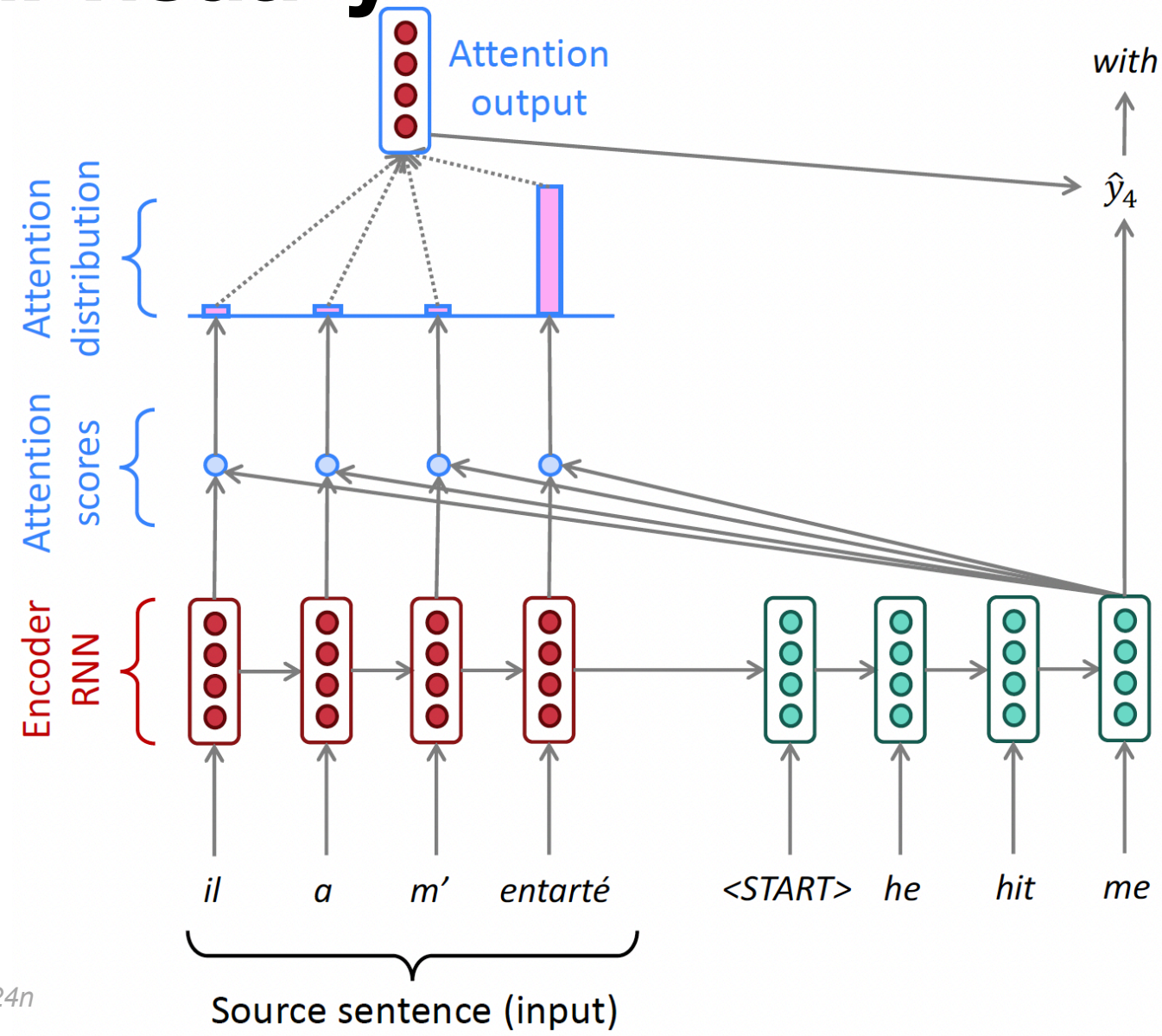
*Figures: Manning et al., Stanford cs224n*

# Attention: visually



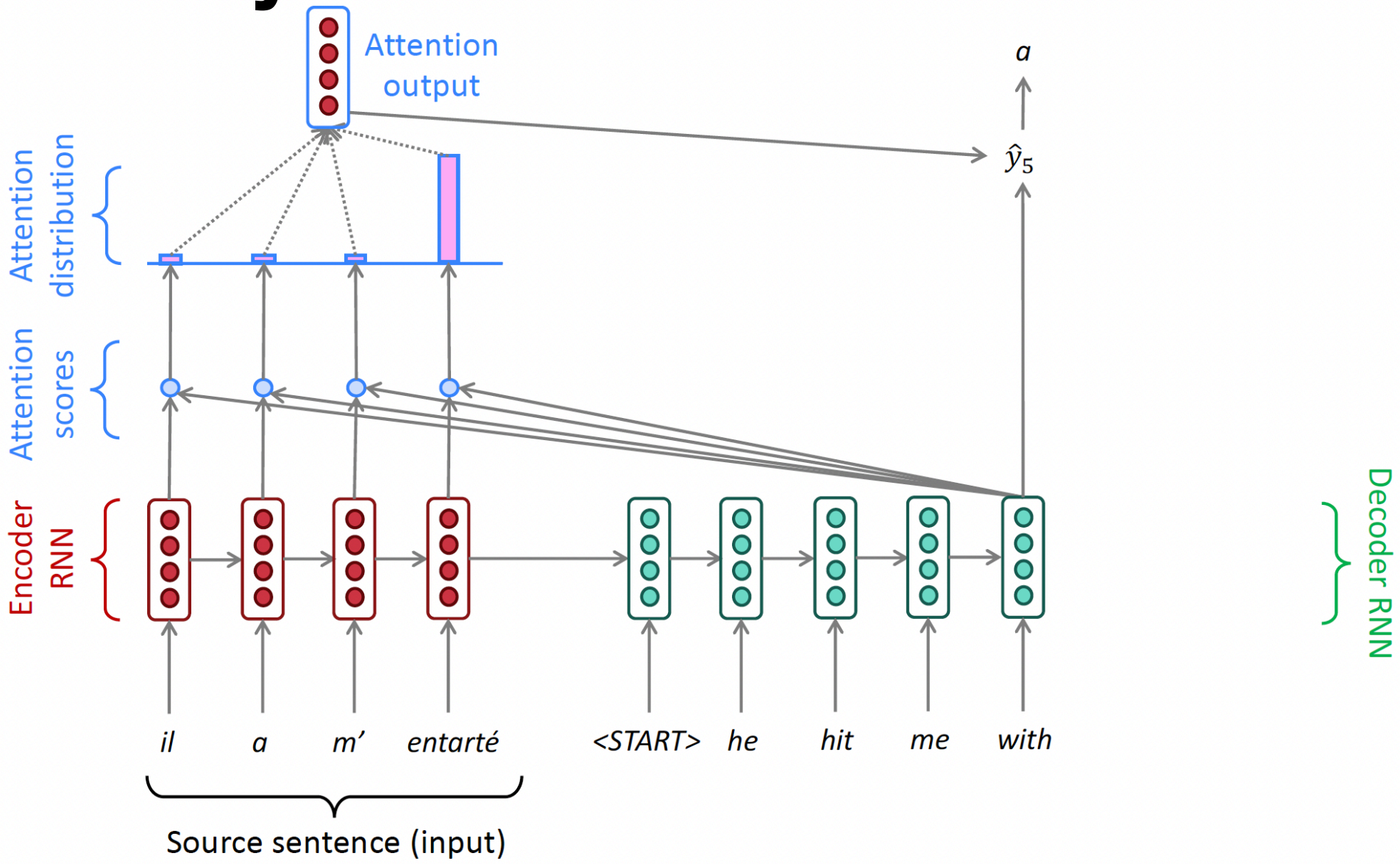

*Figures: Manning et al., Stanford cs224n*

# Attention: visually



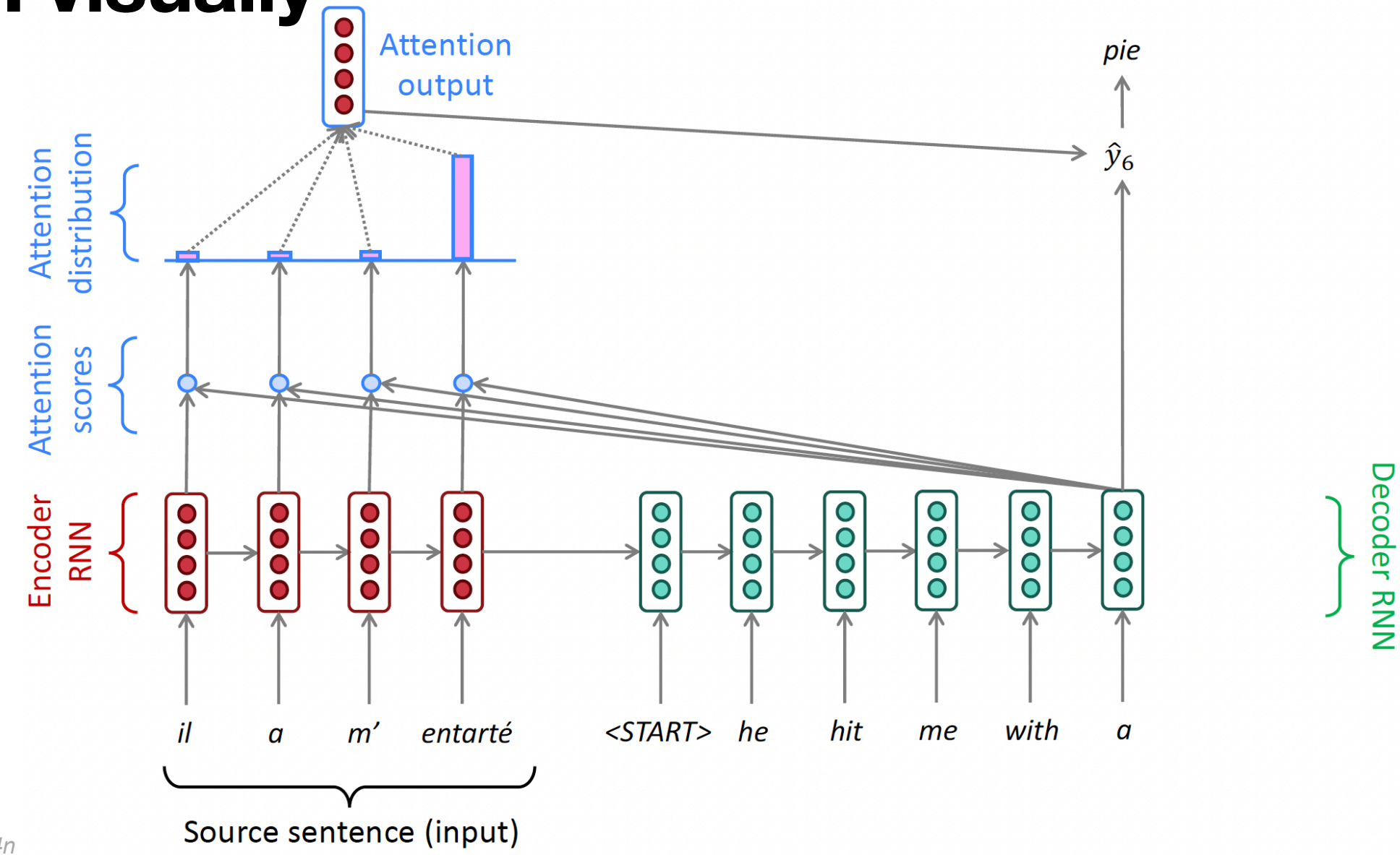

Figures: Manning et al., Stanford cs224n

# Attention: visually




*Figures: Manning et al., Stanford cs224n*

# Attention: visually




Figures: Manning et al., Stanford cs224n

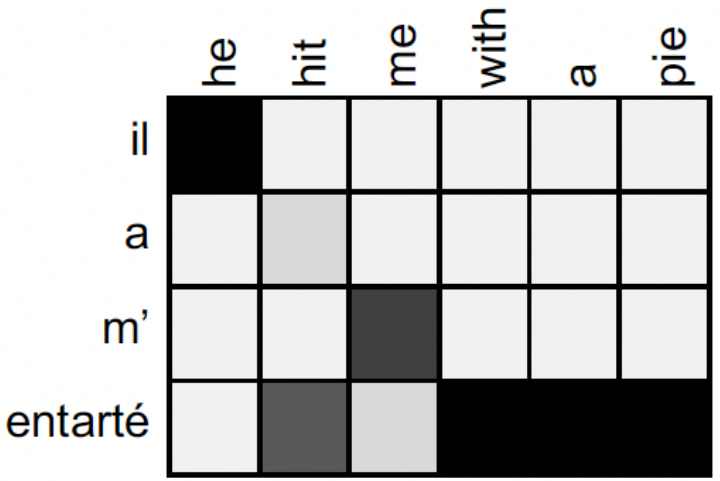
# Attention: the math

- $N$ inputs, hidden state dimension $H$
- Encoder hidden states: $h_1, \ldots, h_N \in \mathbb{R}^H$
- Decoder hidden states: $s_t \in \mathbb{R}^H$
- Attention scores $\boldsymbol{e}_t = [s_t^T h_1, \ldots, s_t^T h_N] \in \mathbb{R}^N$
- Attention distribution: $\alpha_t = \text{softmax}(\boldsymbol{e}_t) \in \mathbb{R}^N$
- Attention output: $\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_t^{(i)} h_i \in \mathbb{R}^H$
- Concatenate $[a_t; s_t] \in \mathbb{R}^{2H}$

Vanilla dot-product attention

# Benefits of Attention

- Key contribution to NMT performance
- Solves bottleneck: decoder can now “look” at complete sequence
- Helps with VG through residual-like connections
- Provides “explainability”:
  - high attention value = high impact to decision
  - soft multi-alignment




*Figures: Manning et al., Stanford cs224n*

# Attention is General Purpose DL

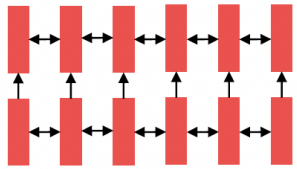
- Given a set of vector *values* and a vector *query*, attention is a technique to compute a weighted sum of the values, dependent on the query
    - "the query *attends to* the values"
    - weighted sum is a selective summary of the information
- Attention allows to obtain fixed-size representations of arbitrary set of representations (*values*) based on some other representation (*query*)

# The many variants of attention…

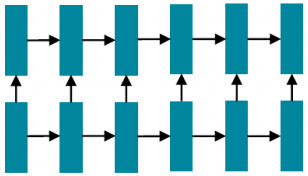
- Basic dot-product (Bahdanau et al. 2015)
- Multiplicative attention: $e_i = s^T W h_i \in \mathbb{R}$ , where W is learned
- Additive attention: $e_i = v^T \tanh(W_1 h_i + W_2 s) \in \mathbb{R}$
- …and many others

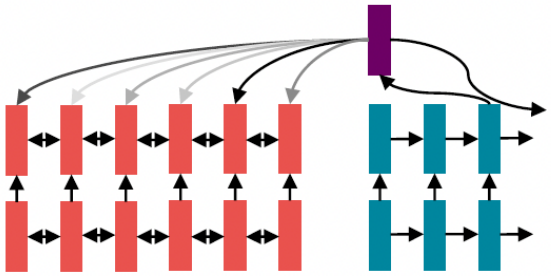
# By 2016, the SOTA was

- encode sentences with a bLSTM



- Define some output (sentiment, summary, …)



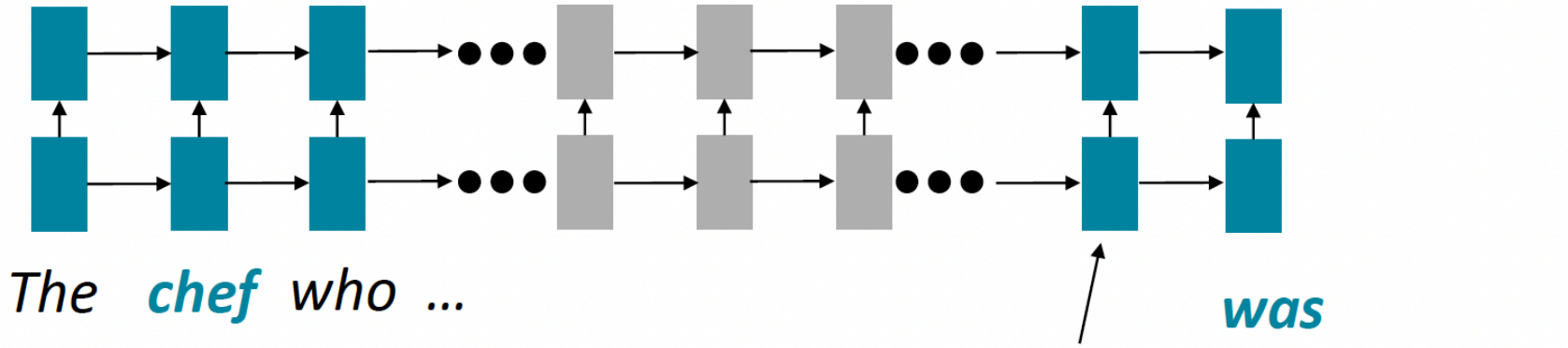


- Add attention to allow flexible memory/data access

**So everything is solved, right?**

# Issues with RNNs

- Unrolled left-to-right (or vice-versa), ie. context is built-up using linear locality
- Problem: RNNs take O(seq-len) steps for distant word pairs to “interact” (slow! gradient!)

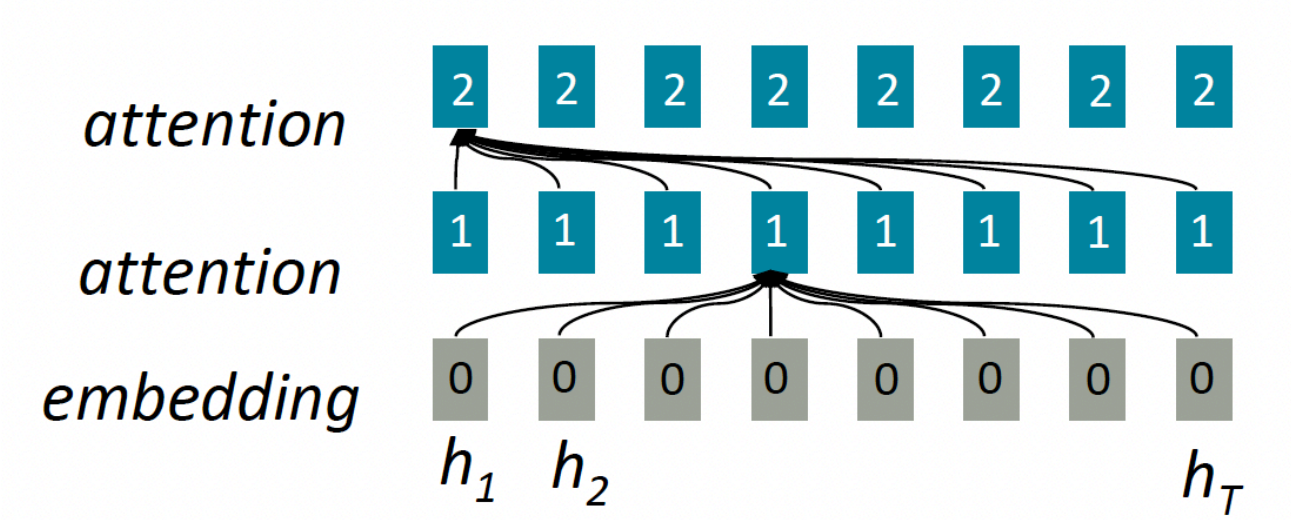



*Figures: Manning et al., Stanford cs224n*

Figures: Manning et al., Stanford cs224n

# If not recurrence, **how about attention?**

- Recall: Attention treats each word's representation as a query to access and incorporate information of a set of values
  - previously: decoder attends to encoder
  - how about this: values attend to each other within the sequence?




All words attend to all words in previous layer; most arrows here are omitted

# Self-Attention

- Recall: Attention operates on queries, keys and values
  - Queries $q_1, \ldots, q_T$; $q_i \in \mathbb{R}^d$
  - Keys $k_1, \ldots, k_T$; $k_i \in \mathbb{R}^d$
  - Values $v_1, \ldots, v_T$;
- Self-attention: $q$, $k$ and
  - …dot product:

$$e_{ij} = q_i^\top k_j$$

Compute **key-query** affinities

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

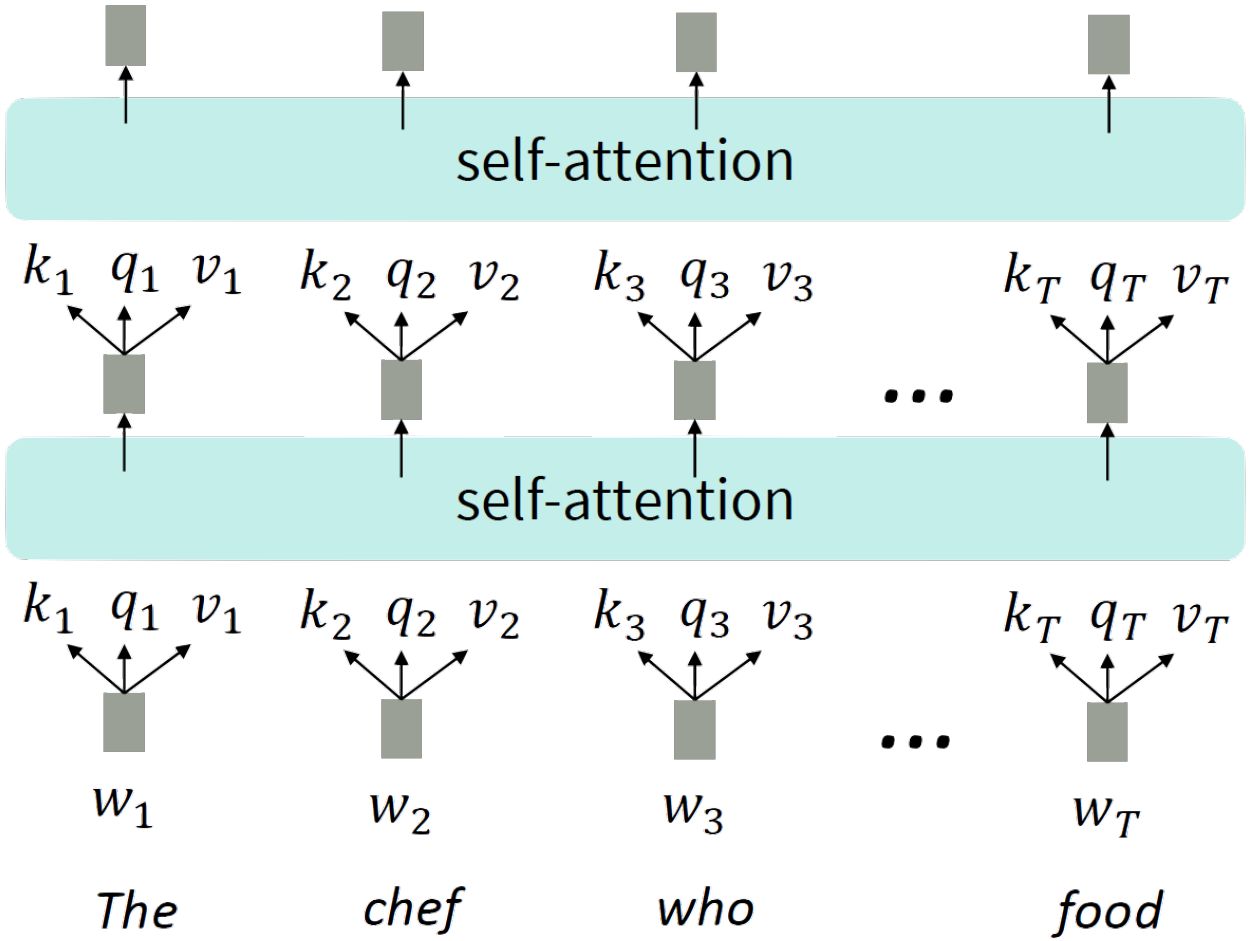
Compute attention weights from affinities (softmax)

$$\text{output}_i = \sum_j \alpha_{ij} v_j$$

Compute outputs as weighted sum of **values**

# Self-Attention as a Building Block

- Self-attention blocks can be stacked
- No free lunch :-(
  1. no notion of order
  2. (only) matrix multiplications, all weighted averages…
  3. for decoders: prevent looking into the future



# “Fixing” Self-Attention (1)

- Sequence order
  - Introduce *position vectors* (aka positional encoding)

    $p_i \in \mathbb{R}^d; \quad i \in 1,\ldots,T$

  - Add to original value, key a$v_i = \tilde{v}_i + p_i$ rs (at input layer)

$$q_i = \tilde{q}_i + p_i$$
$$k_i = \tilde{k}_i + p_i$$

# “Fixing” Self-Attention (1)

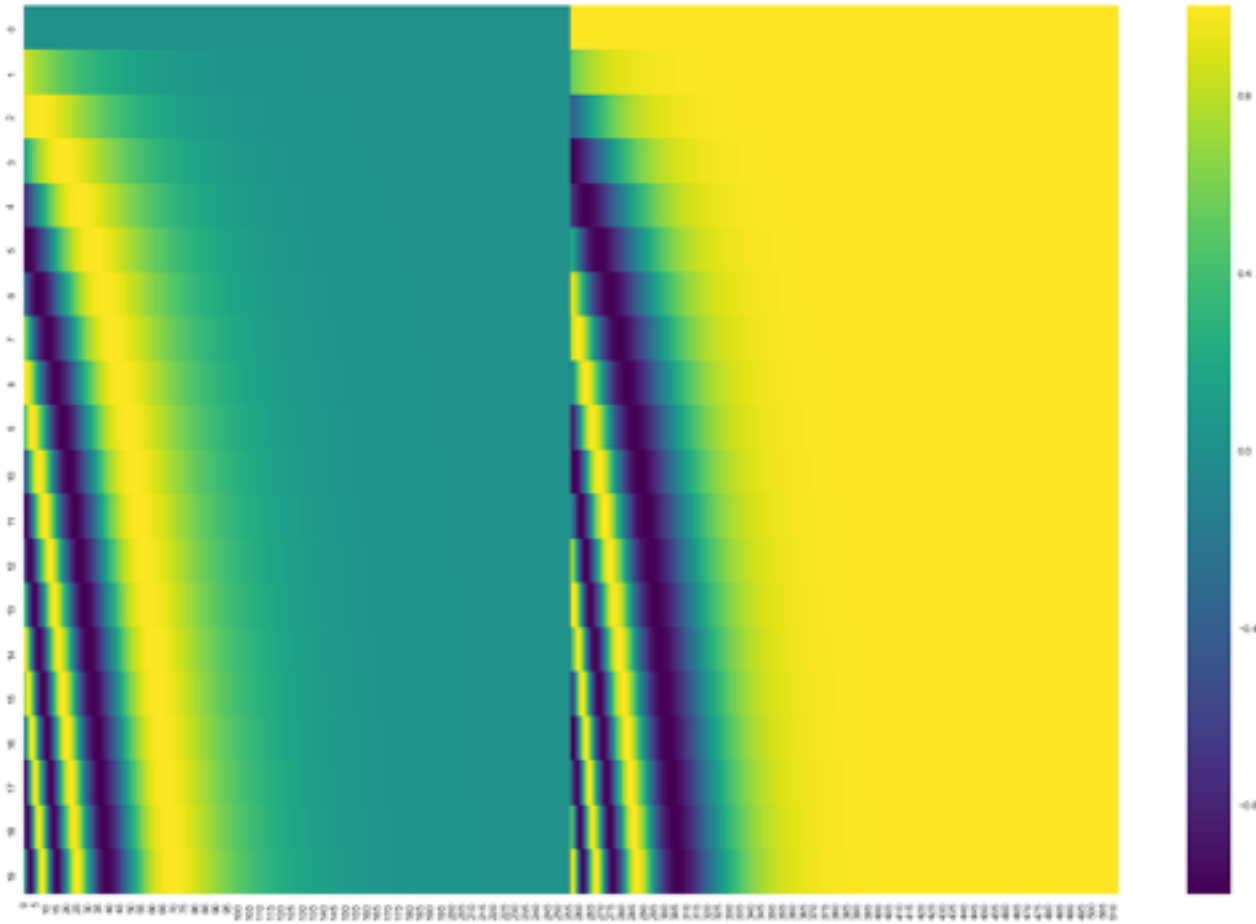
## Sources for positional embeddings

- Should allow meaningful distances between embedding vectors
    - Vectors follow a specific pattern/ formula
    - Sinusoidal pattern
    - Sequence number
    - Learned
    - Left out altogether
    - …

# “Fixing” Self-Attention (1)

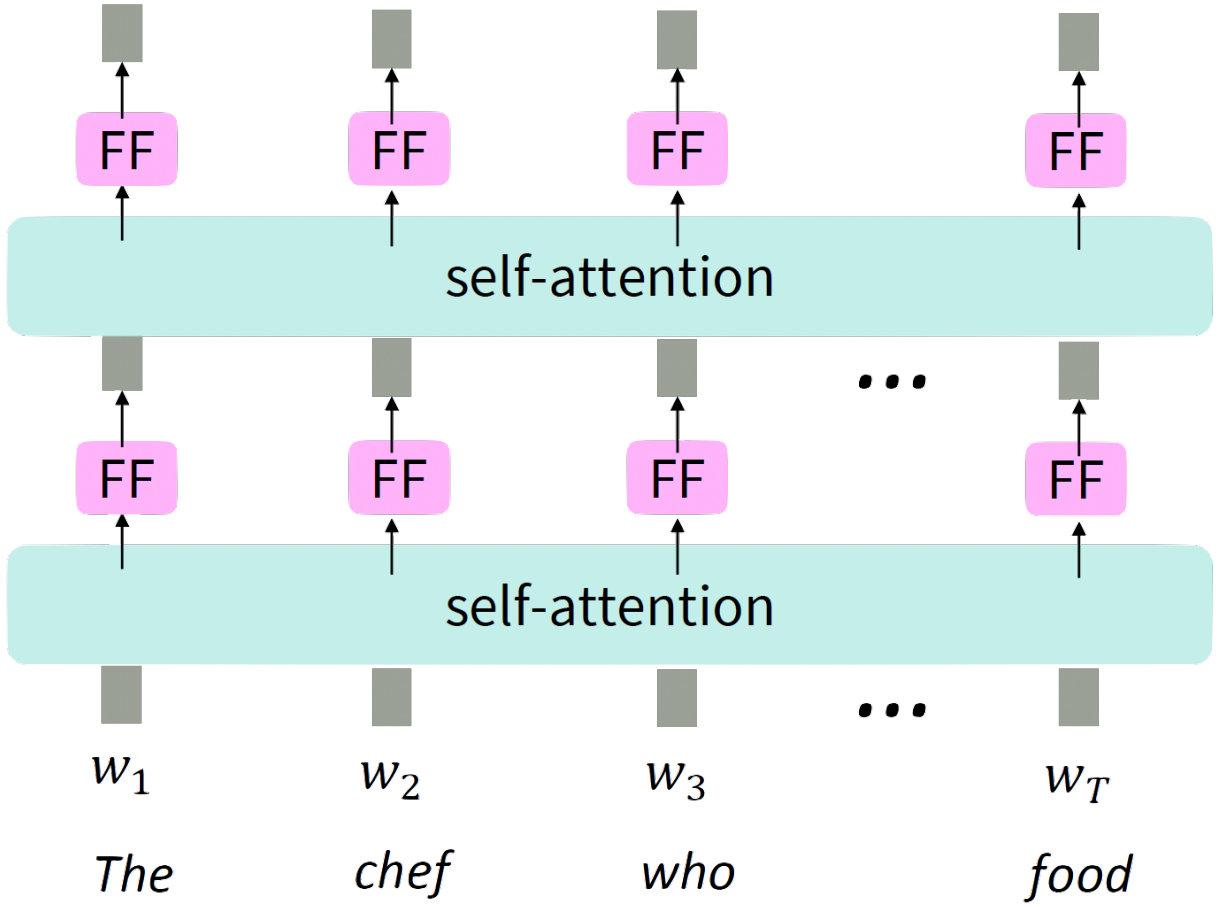
## Most common pattern, proposed in Vaswani et al., 2017

- $\overrightarrow{x} = \sqrt{d_{model}}\overrightarrow{x} + PE$
- $PE_{(pos,2i)} = \sin(pos 10000^{\frac{2i}{d_{model}}})$
- $PE_{(pos,2i+1)} = \cos(pos 10000^{\frac{2i}{d_{model}}})$
- $\cos(x) = sin(x + \frac{\pi}{2})$



# “Fixing” Self-Attention (2)

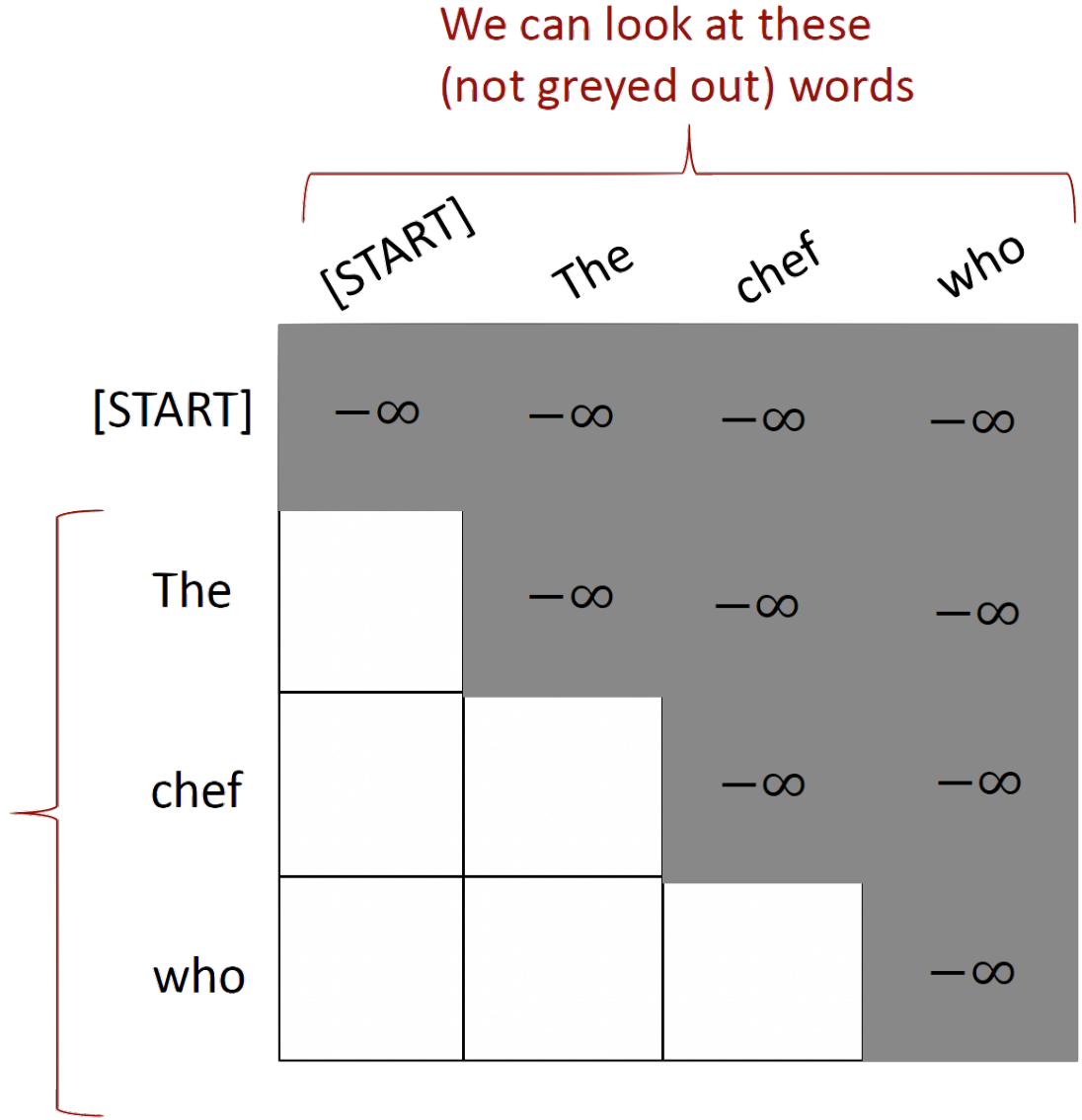
- Linear combinations…
  - add nonlinearity!
  - eg. Linear(Relu(Linear .)




Intuition: the FF network processes the result of attention

# “Fixing” Self-Attention (3)

- For decoders, restrict visibility of future values
- “manually” computing keys and queries too inconvenient
- For parallelization, **mask out attention** to future values

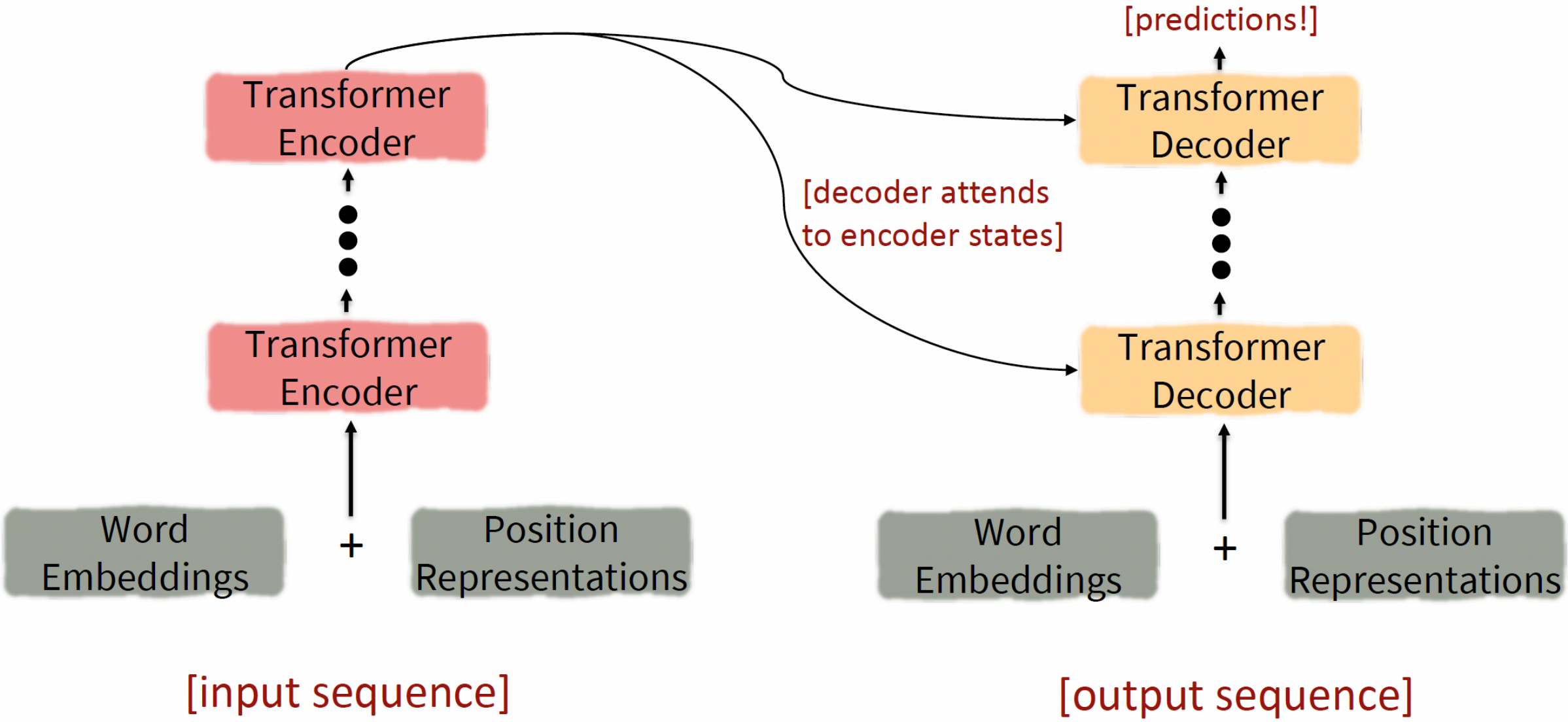



$$e_{ij} = \begin{cases} q_i^\top k_j, j < i \\ -\infty, j \geq i \end{cases}$$

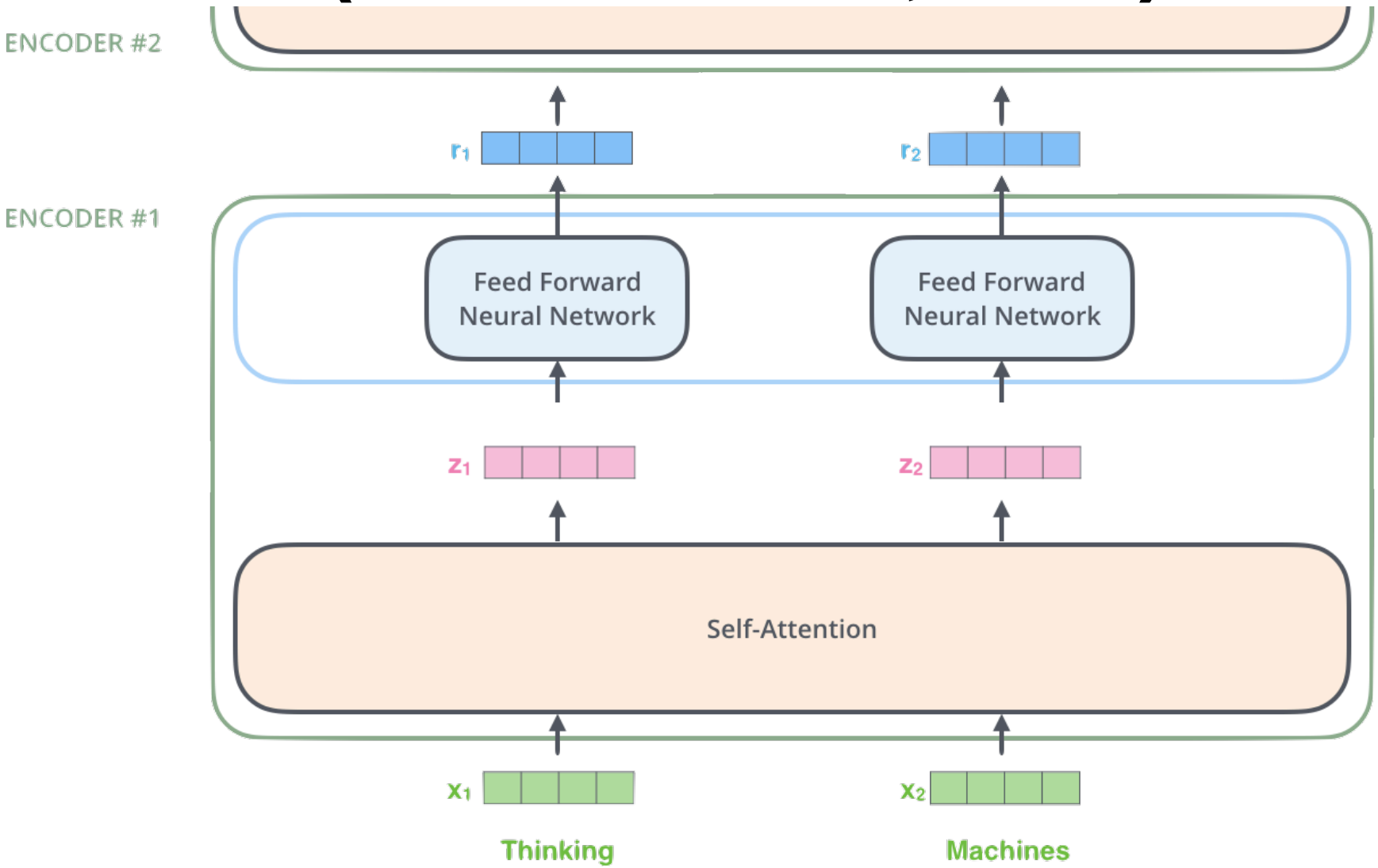
# We got the building blocks:

- Self-attention:
    - recurrence-free (fast!) and spanning the whole sequence
- Position encodings
    - re-introduce sequence order to key, query and values
- Masking
    - to allow parallel computations while “not looking into the future”

# Transformer (Vaswani et al., 2017)

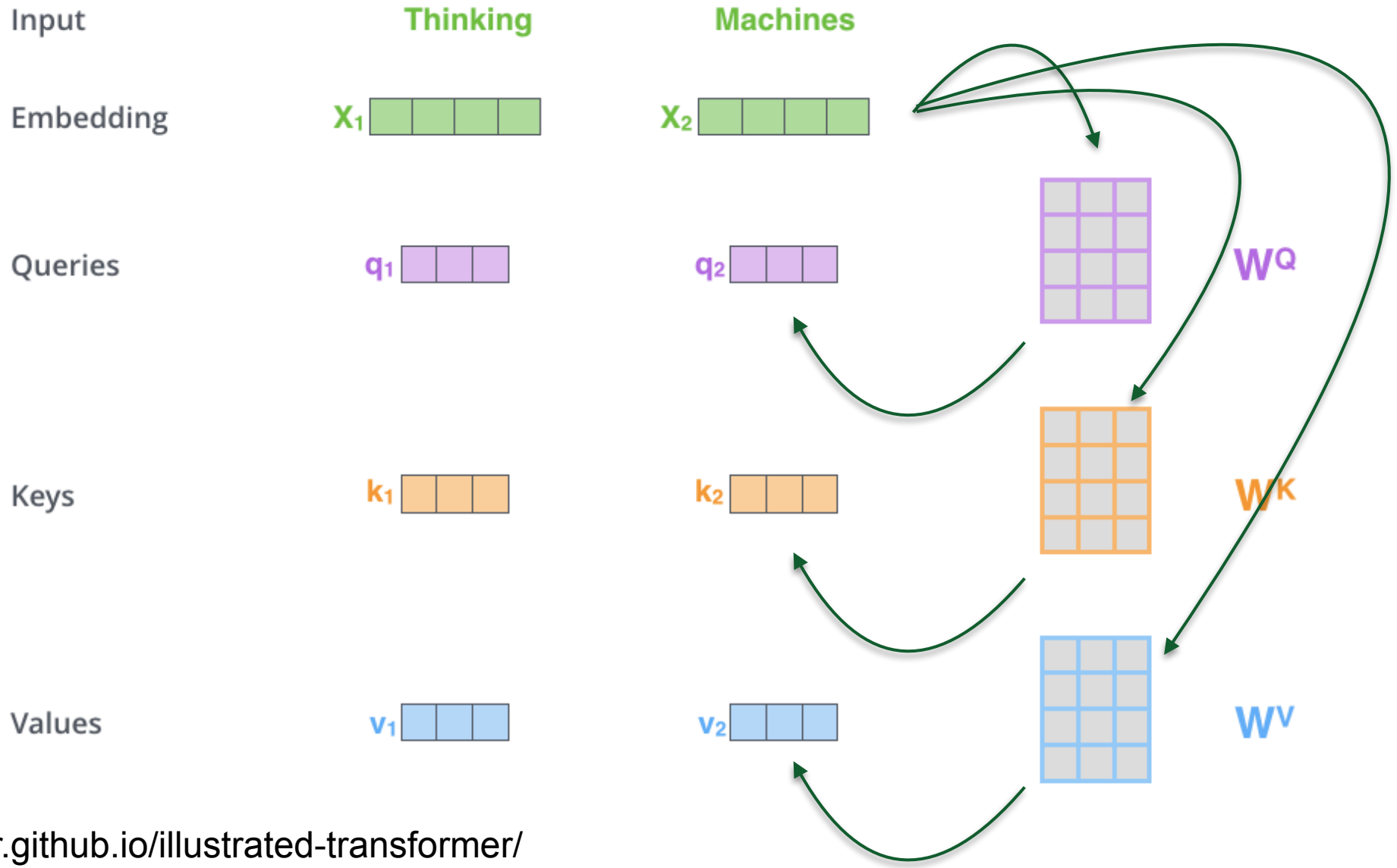


# Transformer (Vaswani et al., 2017)




https://jalammar.github.io/illustrated-transformer/

# Key-Query-Value: visually



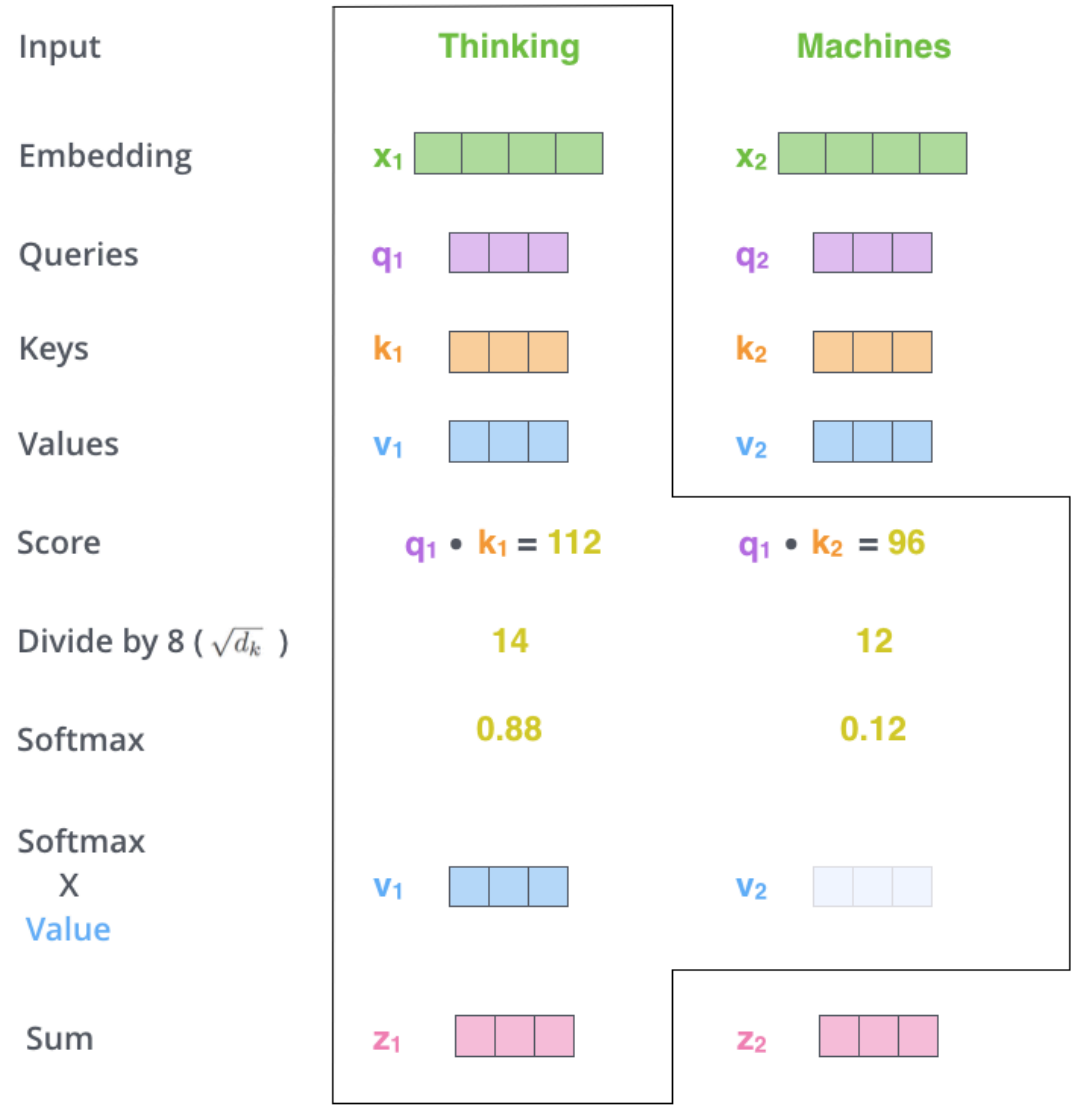

https://jalammar.github.io/illustrated-transformer/

# Key-Query-Value: the math

- Let $x_1, \ldots, x_T$ be the input vectors to the Transformer encoder
- Then we introduce (learnable!) matrices *K*, *Q* and *V* to compute
  - $k_i = Kx_i$ where *K* is the key matrix
  - $q_i = Qx_i$ where *Q* is the key matrix
  - $v_i = Vx_i$ where *V* is the key matrix

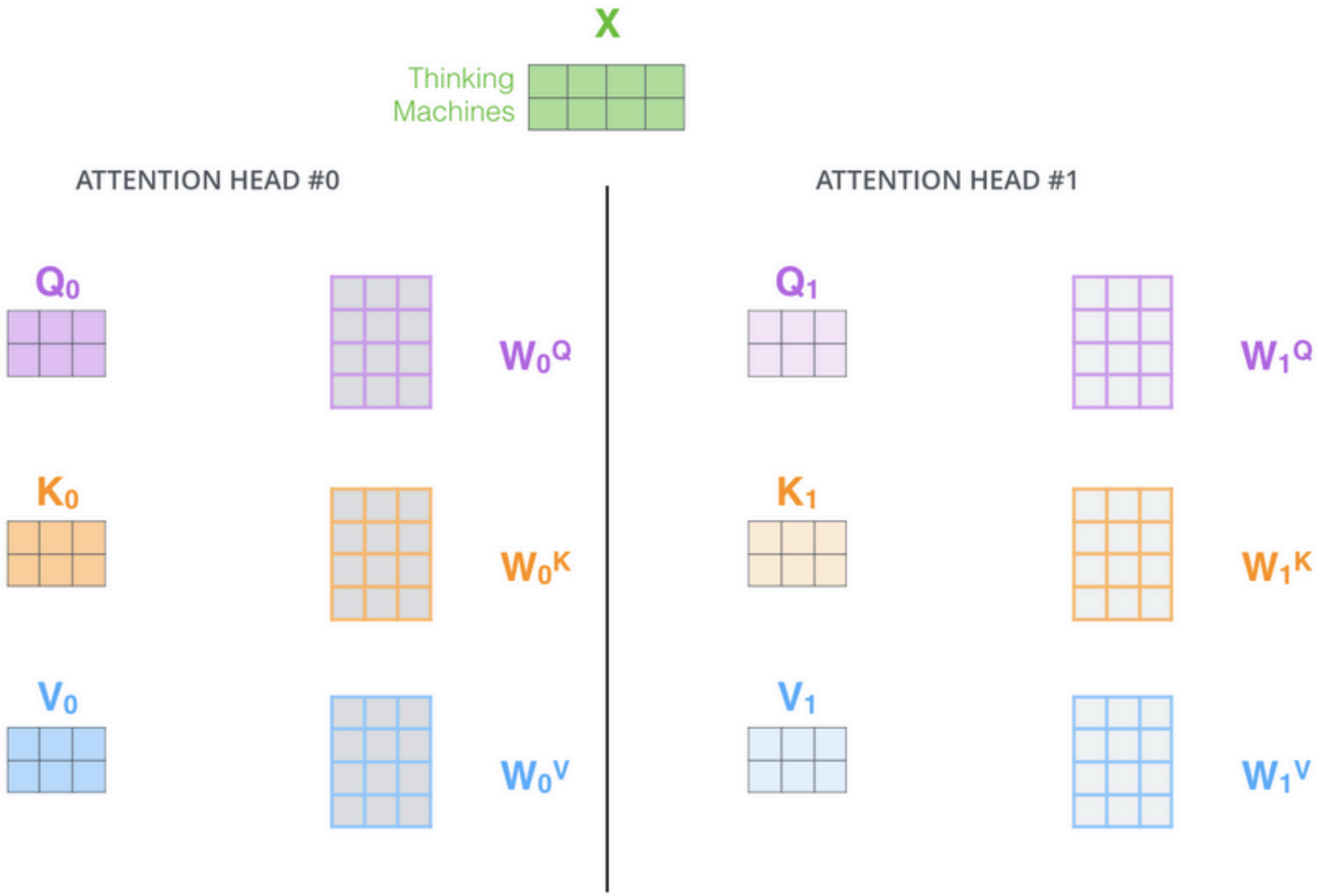
# Scaled Dot-Product, Softmax, Sum

- Let $x_1, \ldots, x_T$ be the input vectors to the Transformer encoder
- Then we introduce (learnable!) matrices *K*, *Q* and *V* to compute
  - $k_i = Kx_i$ where *K* is the key matrix
  - $q_i = Qx_i$ where *Q* is the key matrix
  - $v_i = Vx_i$ where *V* is the key matrix




https://jalammar.github.io/illustrated-transformer/

# Attention – Multi-head attention

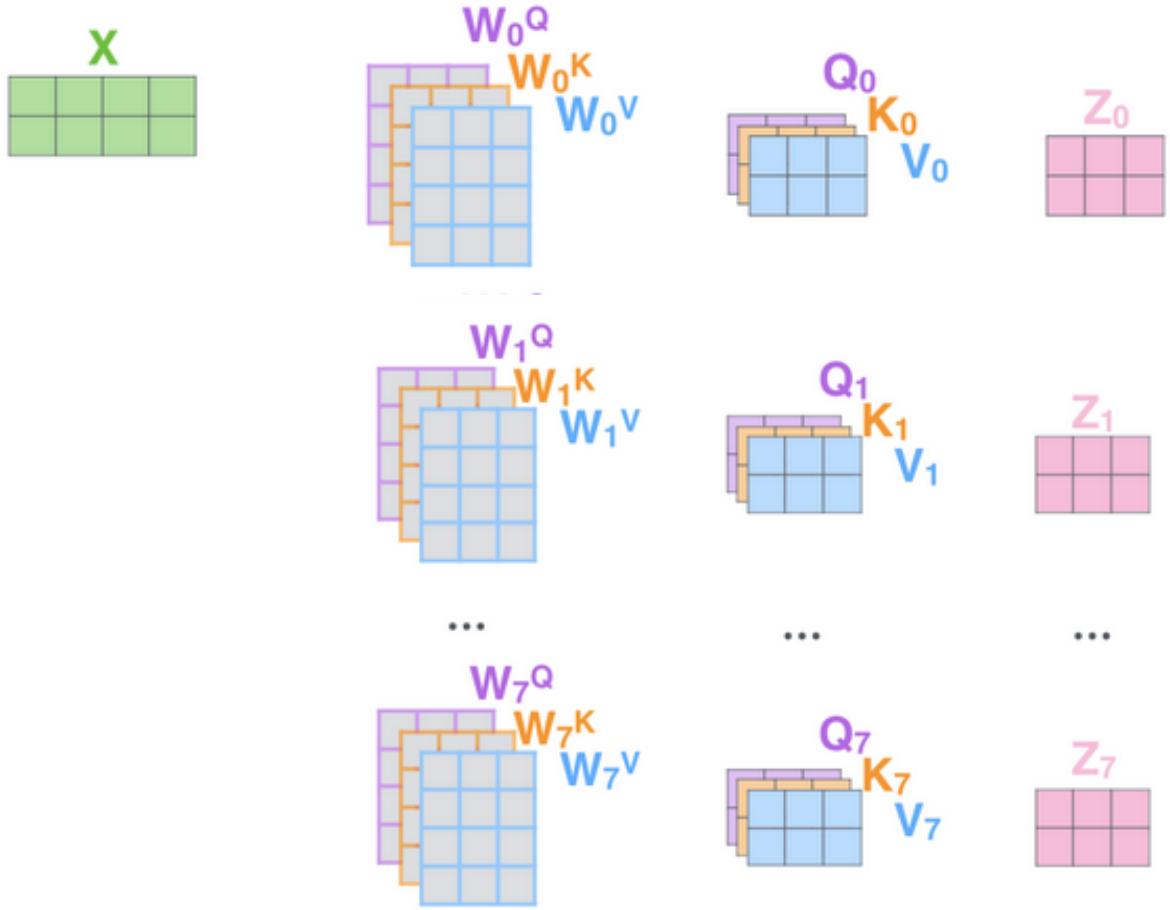



https://jalammar.github.io/illustrated-transformer/

# Attention – Multi-head attention

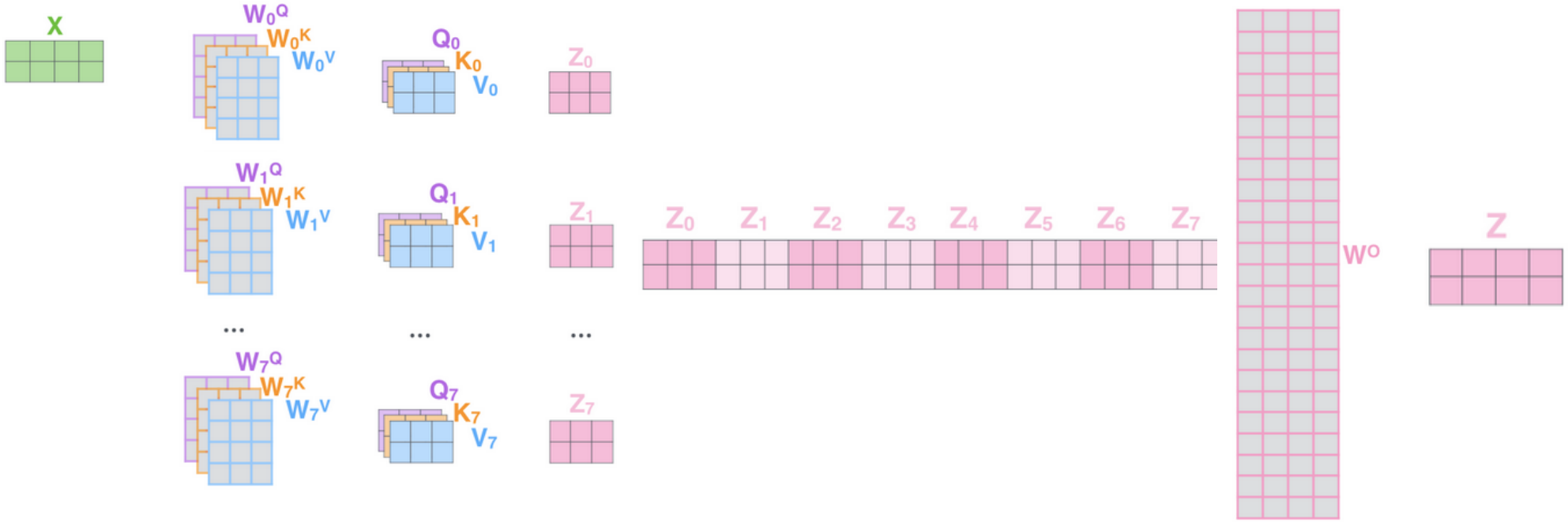
- Transformers make use of multiple attention heads per transformer block
- Multiple ‘attentive’ views on the same concept
- This leads to multiple learnable, key, query and value projections
- All **“heads”** have their own, separately calculated attention output
- The attention outputs will be concatenated and then be multiplied with a shared weight matrix

# Attention – Multi-head attention



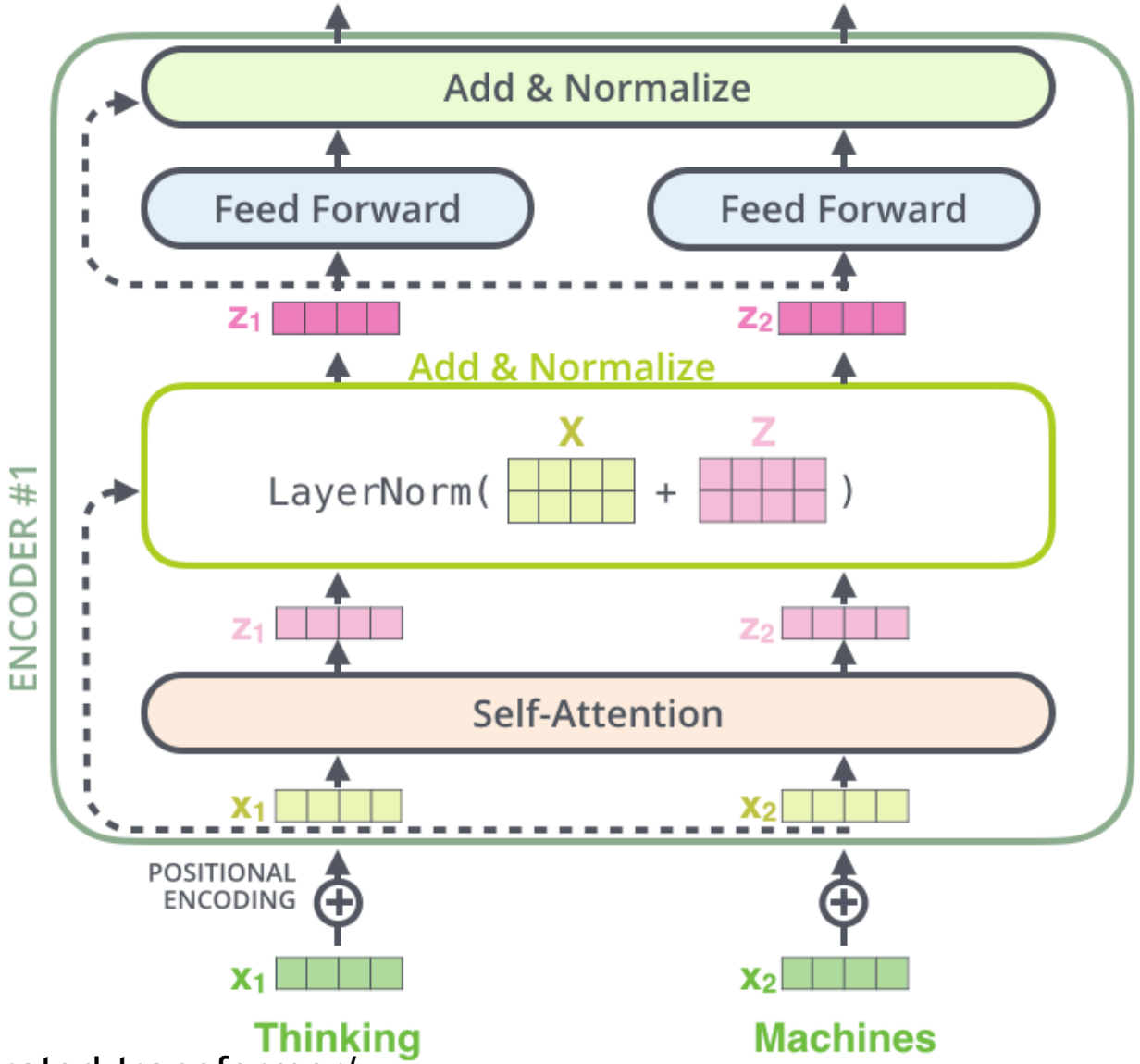

https://jalammar.github.io/illustrated-transformer/

# Attention – Multi-head attention



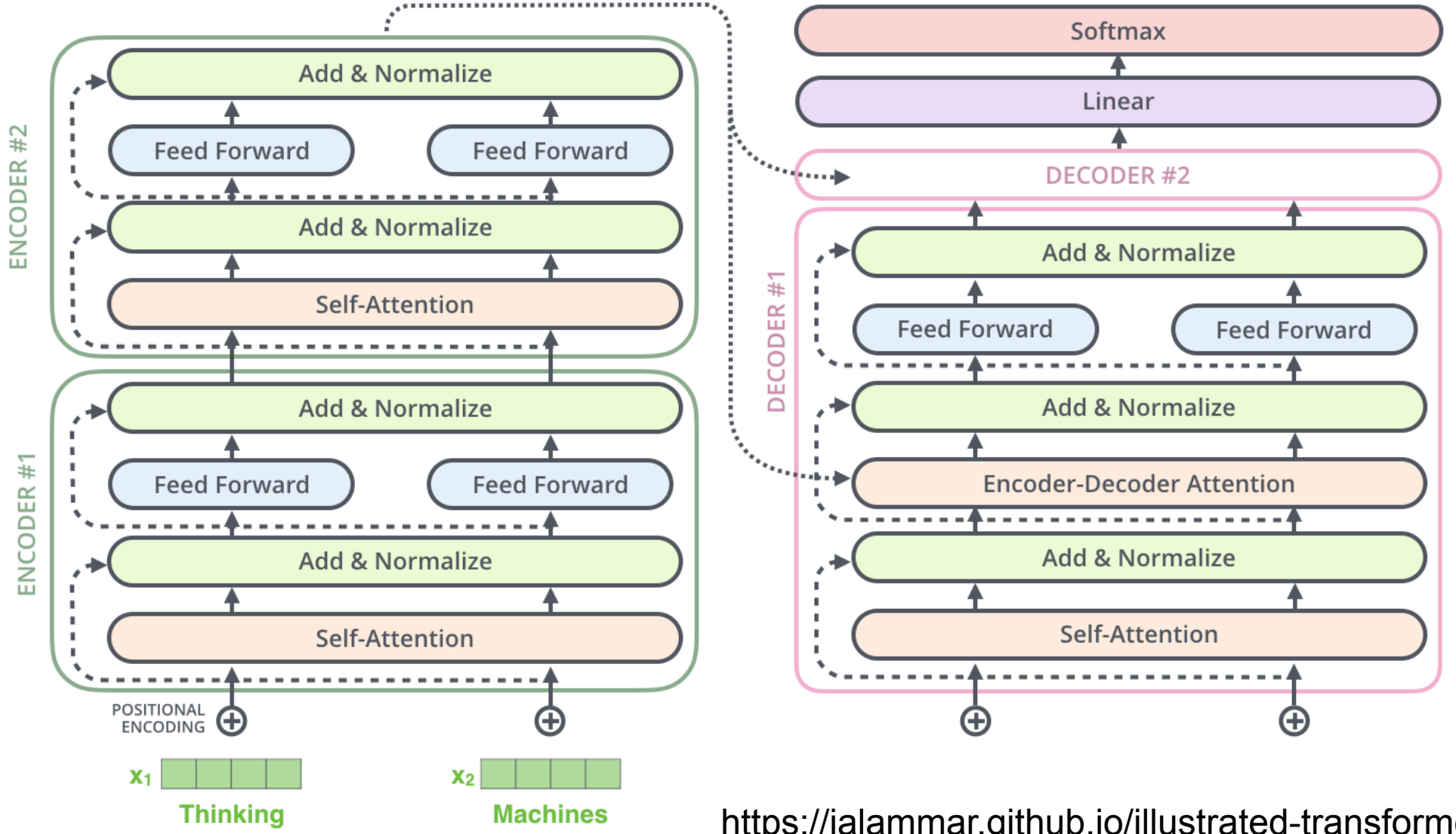

https://jalammar.github.io/illustrated-transformer/

# Residuals and LayerNorm



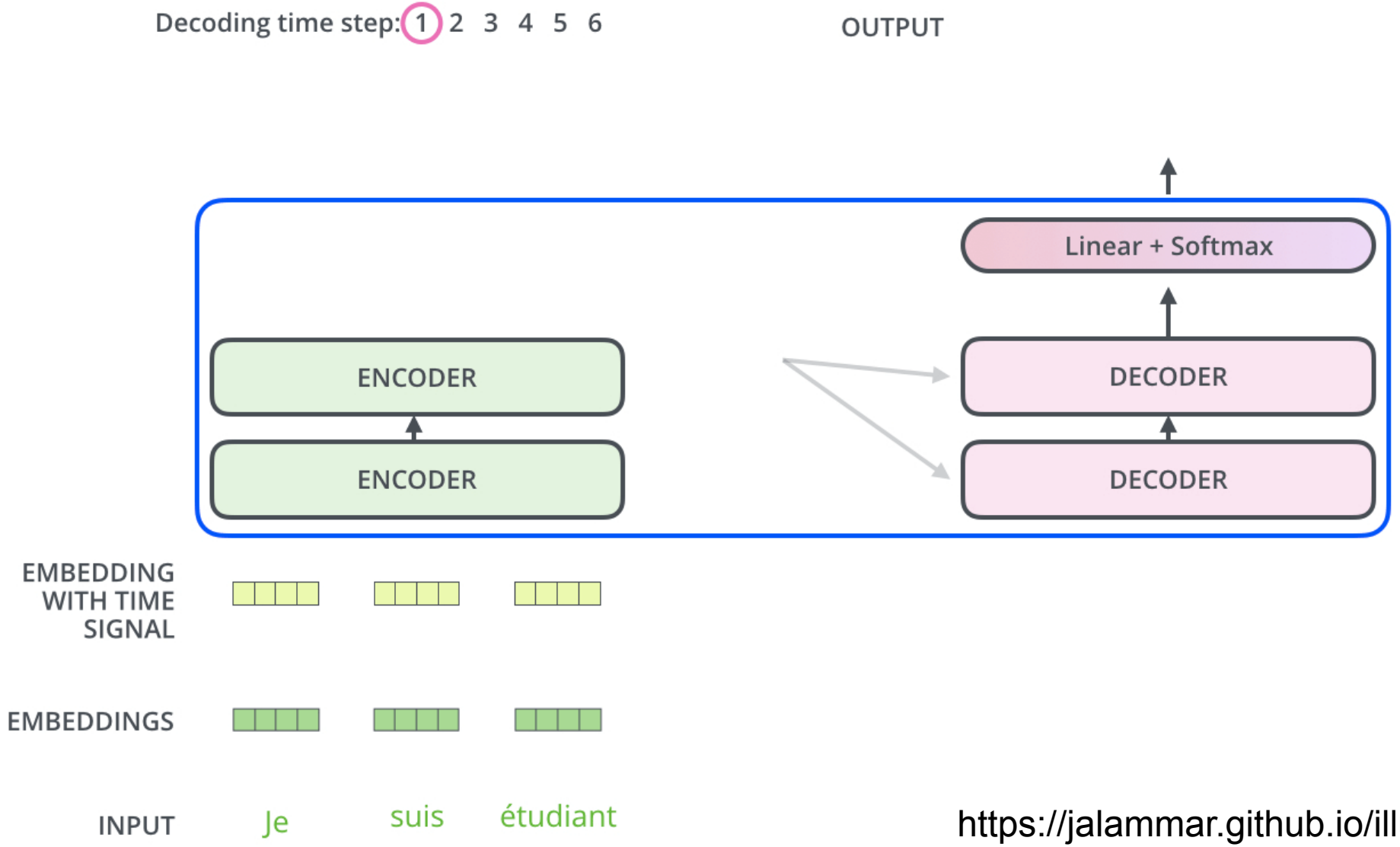

https://jalammar.github.io/illustrated-transformer/

# Decoder



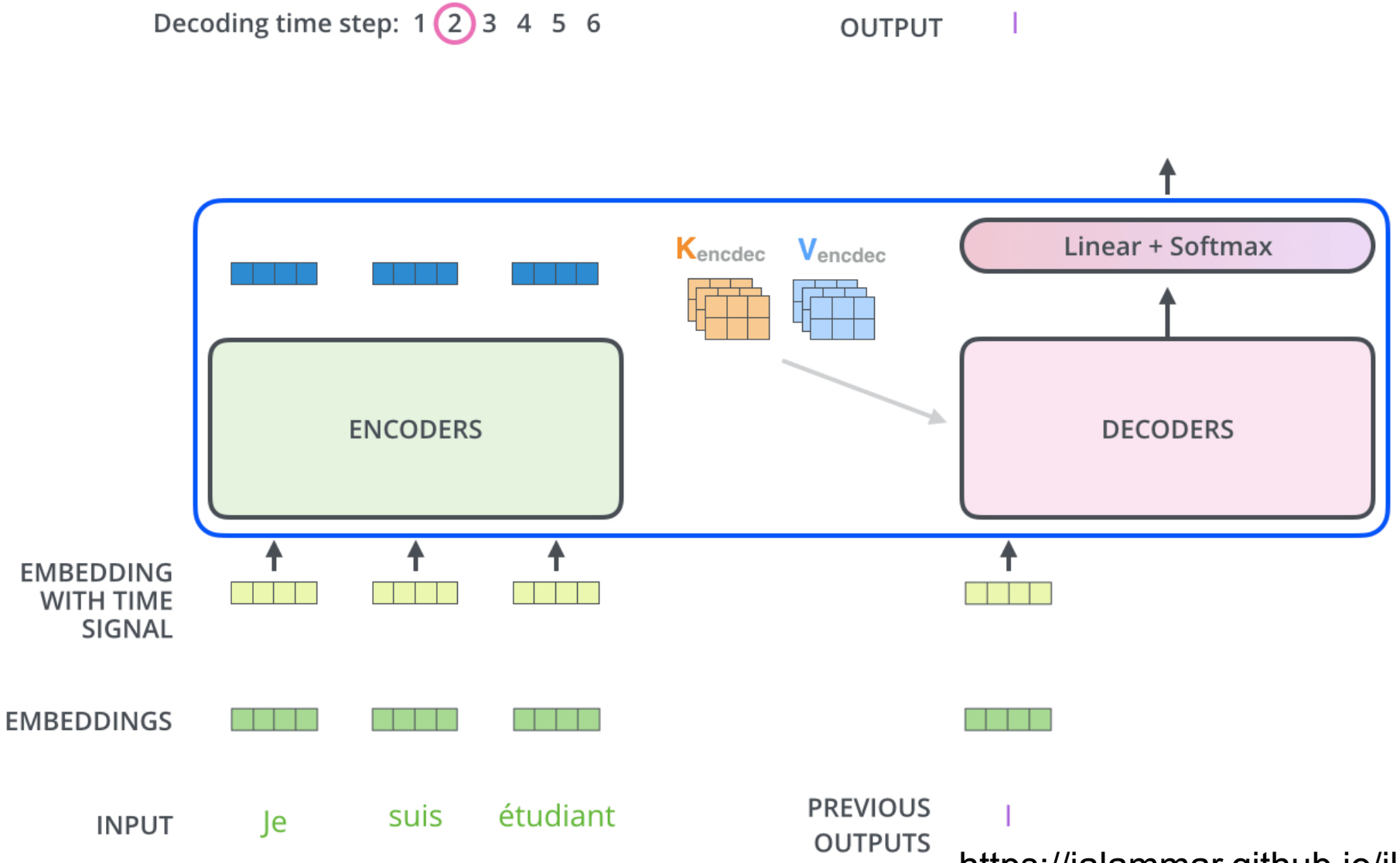

https://jalammar.github.io/illustrated-transformer/

# Transformer: Animated




https://jalammar.github.io/illustrated-transformer/

# Transformer: Animated




https://jalammar.github.io/illustrated-transformer/

# What's special about the Transformer?

- No recurrency (but positional encodings)
- Highly parallelizeable (hey, it's foremost matrix multiplications)
- (It can be pretrained and generalizes well)

# Summary

- Attention is a great mechanism to (directly) access information from across the whole sequence
    - Helps with VG, for similar reasons like residuals
- Self-attention (where k, q, v are computed from x) is a great way to encode a sequence without recurrence
- Transformers are a special setup of self-attention, scaled dot product, residuals and layernorm.
- Transformers are the current state-of-the-art
    - …if you have enough data to train them :-)