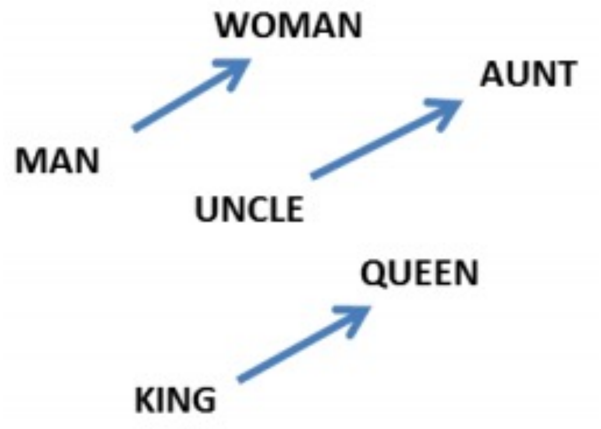
CS 307 Modeling and Learning in Data Science

Today’s goal

- Recall: Image autoencoder
- Simple word embeddings
- Encoder decoder for word embeddings
- Attention
- Transformers

Word embeddings: properties

- Relationships between words correspond to difference between vectors.



$$W(\text{``woman''}) - W(\text{``man''}) \simeq W(\text{``aunt''}) - W(\text{``uncle''})$$

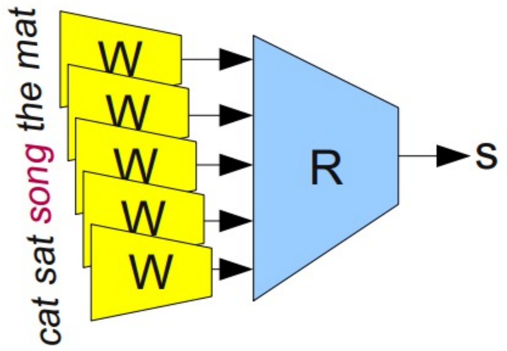
$$W(\text{``woman''}) - W(\text{``man''}) \simeq W(\text{``queen''}) - W(\text{``king''})$$

Word embeddings: questions

- How big should the embedding space be?
  - Trade-offs like any other machine learning problem – greater capacity versus efficiency and overfitting.
- How do we find W?
  - Often as part of a prediction or classification task involving neighboring words.

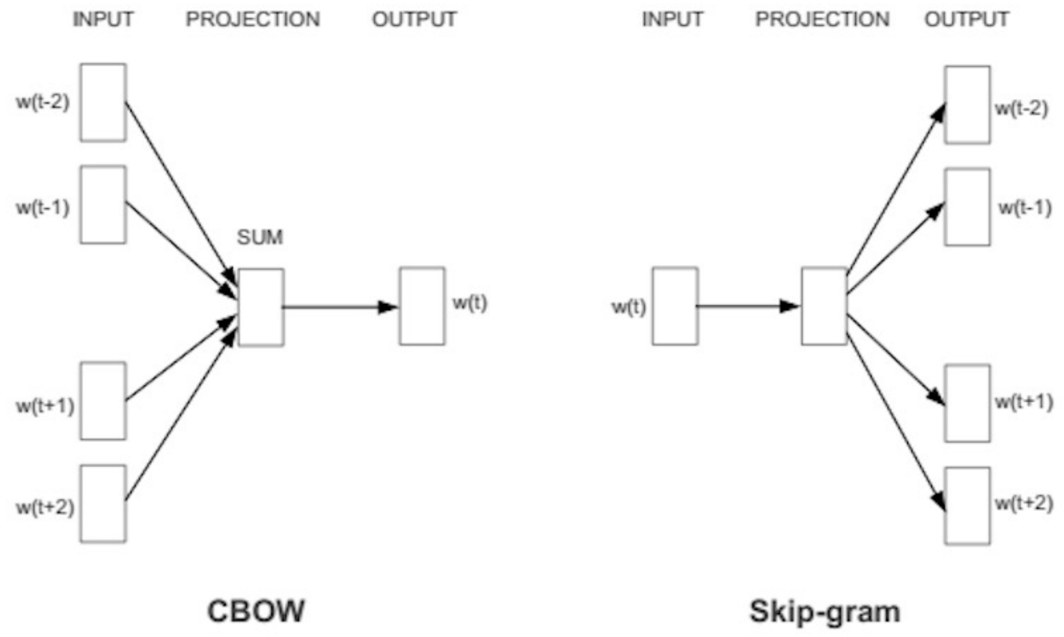
Learning word embeddings

- First attempt:
  - Input data is sets of 5 words from a meaningful sentence. E.g., “one of the best places”. Modify half of them by replacing middle word with a random word. “one of function best places”
  - W is a map (depending on parameters, Q) from words to 50 dim’l vectors. E.g., a look-up table or an RNN.
  - Feed 5 embeddings into a module R to determine ‘valid’ or ‘invalid’
  - Optimize over Q to predict better



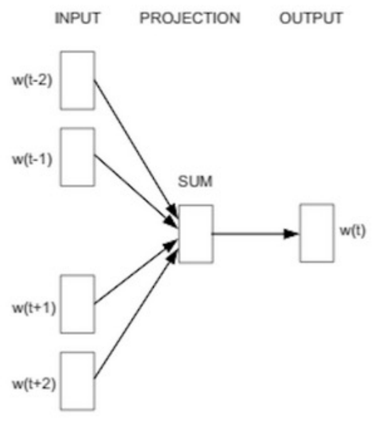
word2vec

- Predict words using context
- Two versions: CBOW (continuous bag of words) and Skip-gram



CBOW

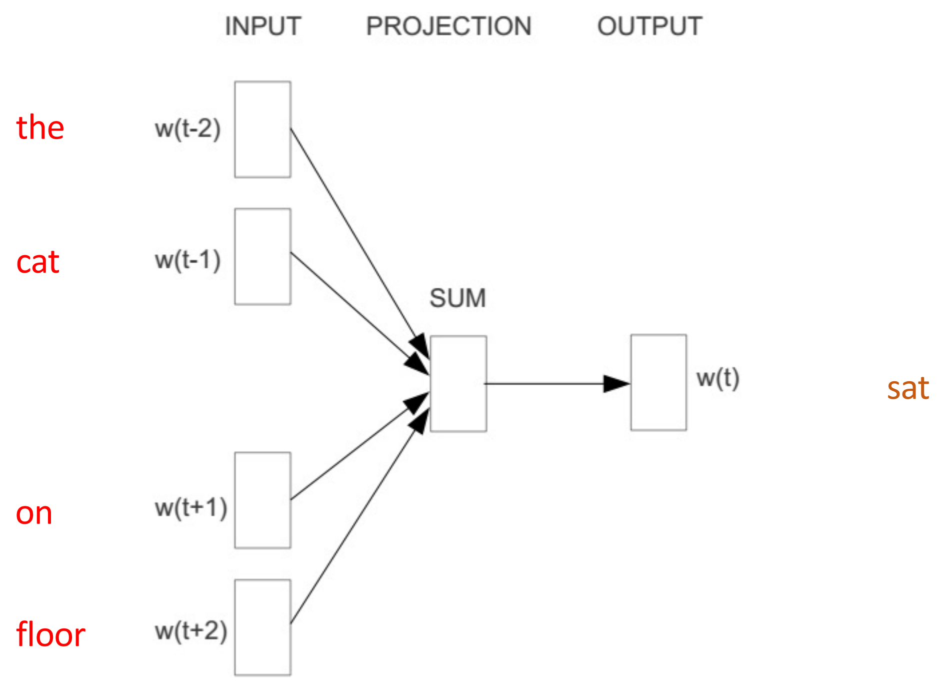
- Bag of words
  - Gets rid of word order. Used in discrete case using counts of words that appear.
- CBOW
  - Takes vector embeddings of n words before target and n words after and adds them (as vectors).
  - Also removes word order, but the vector sum is meaningful enough to deduce missing word.

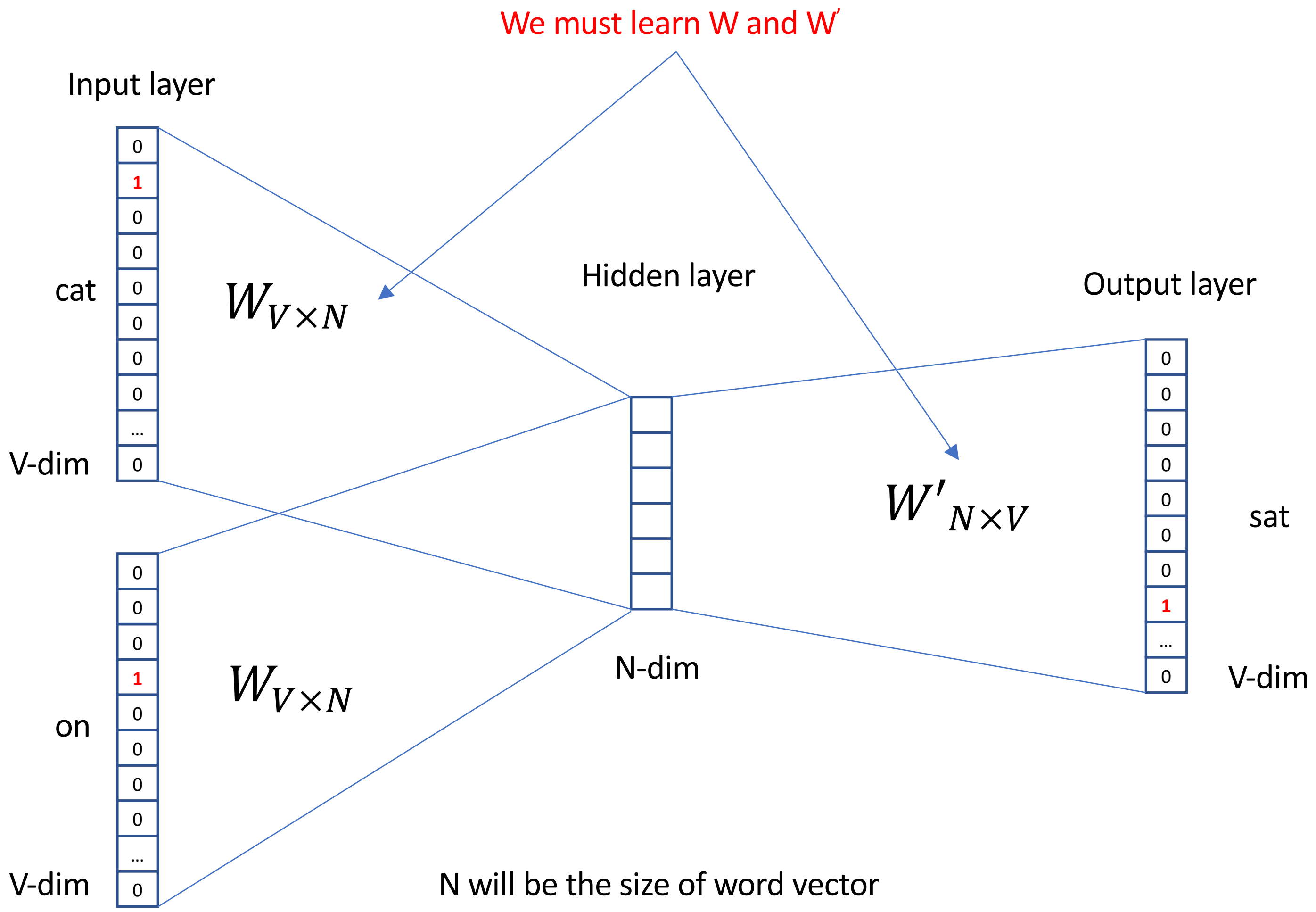


CBOW

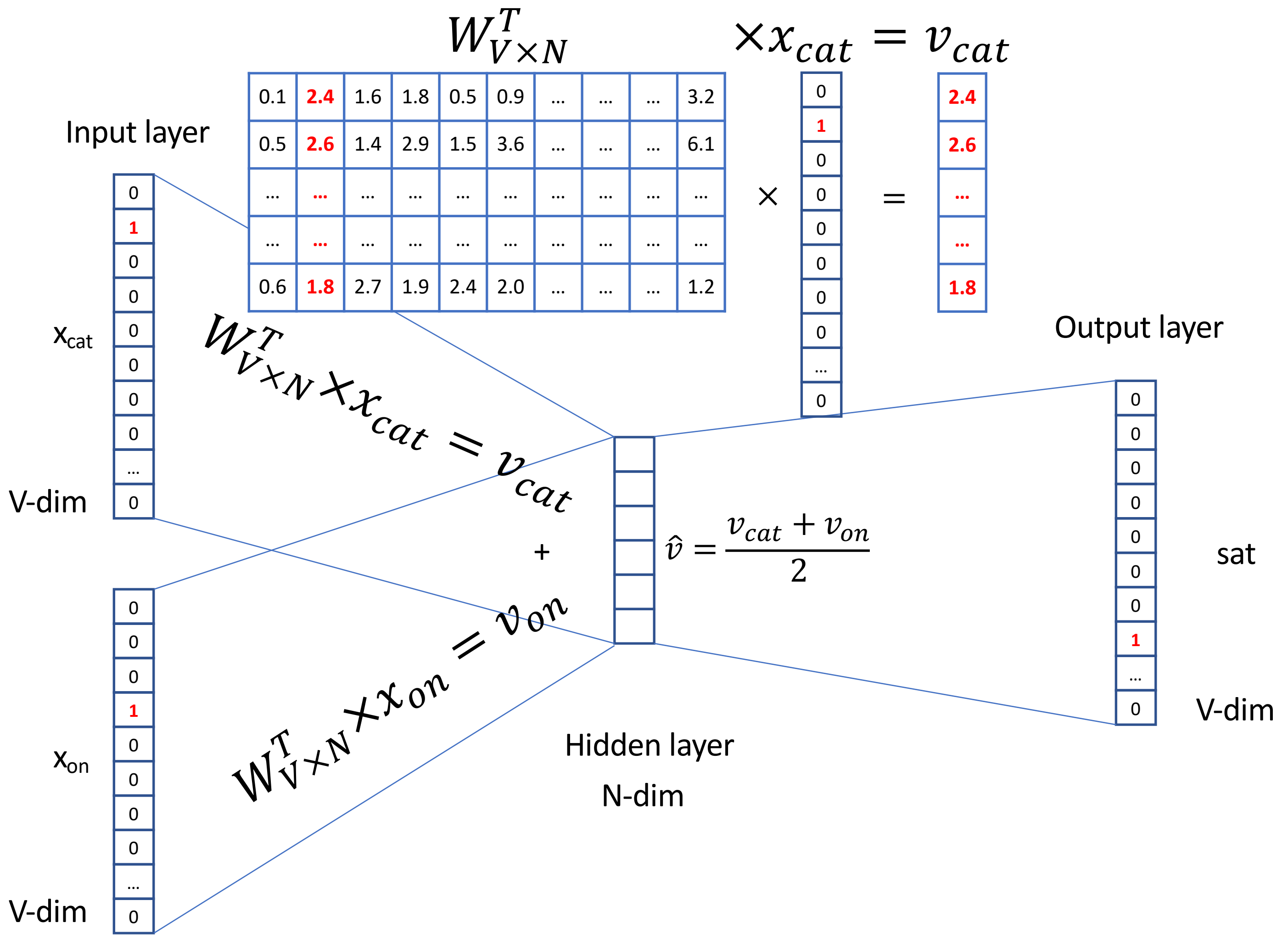
Word2vec – Continuous Bag of Word

- E.g. "The cat sat on floor"
    - Window size = 2



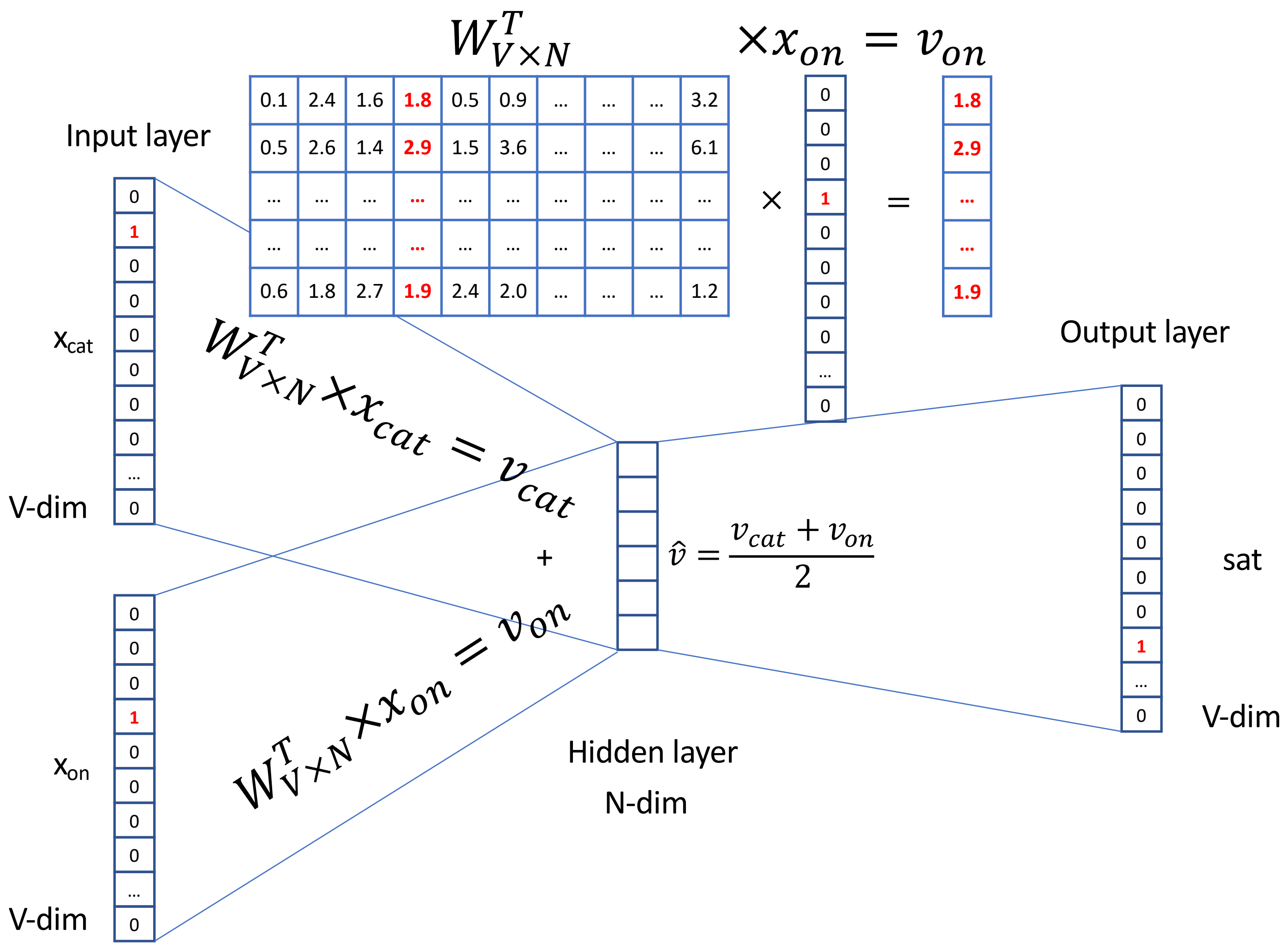


N will be the size of word vector

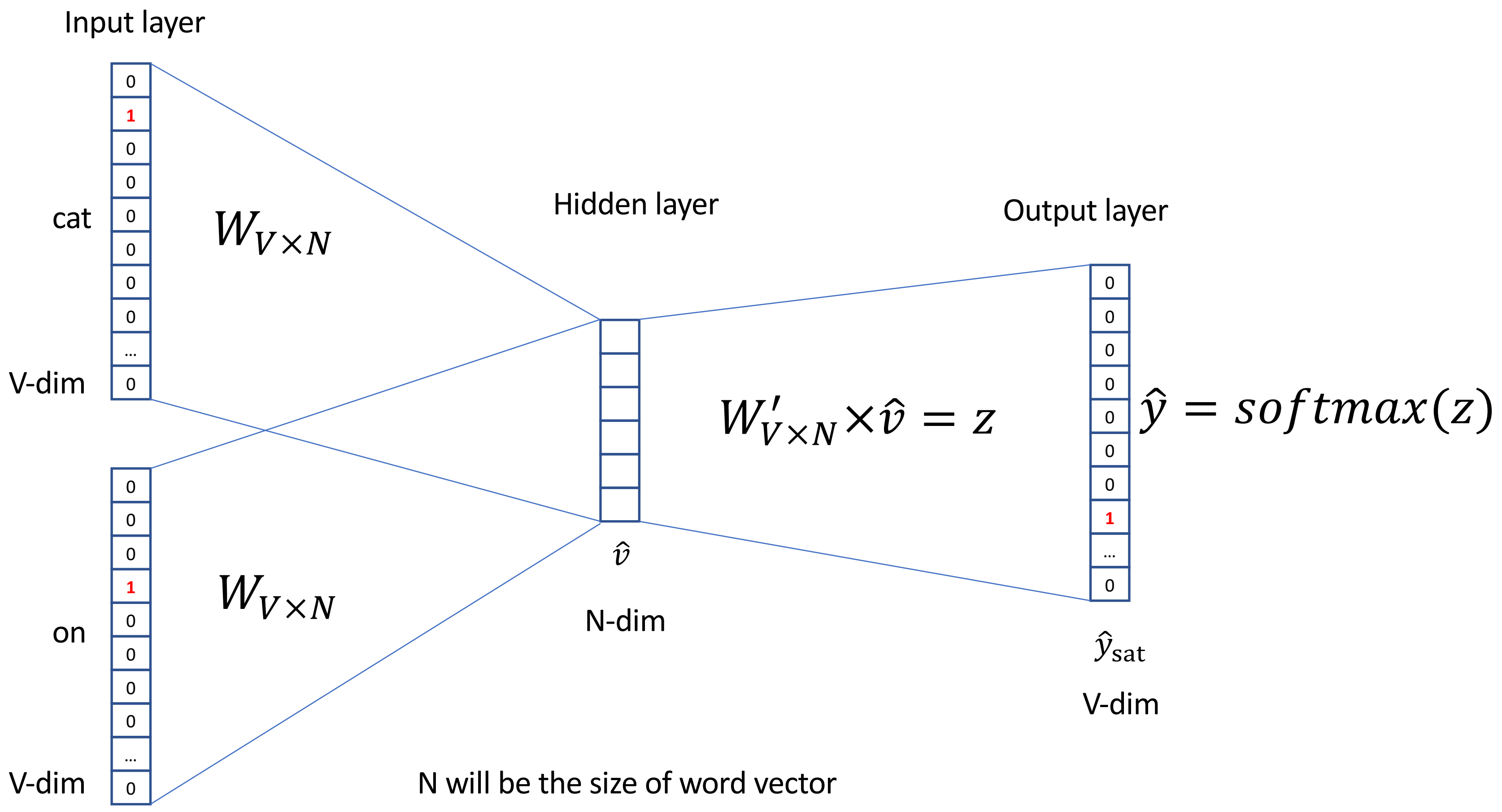


$W_{V\times N}^T \times x_{cat} = v_{cat}$
Input layer
x_{cat}
V-dim
x_{on}
V-dim
$W_{V\times N}^T \times x_{cat} = v_{cat}$
+
$W_{V\times N}^T \times x_{on} = v_{on}$
$\hat{v} = \frac{v_{cat} + v_{on}}{2}$
Hidden layer
N-dim
Output layer
sat
V-dim

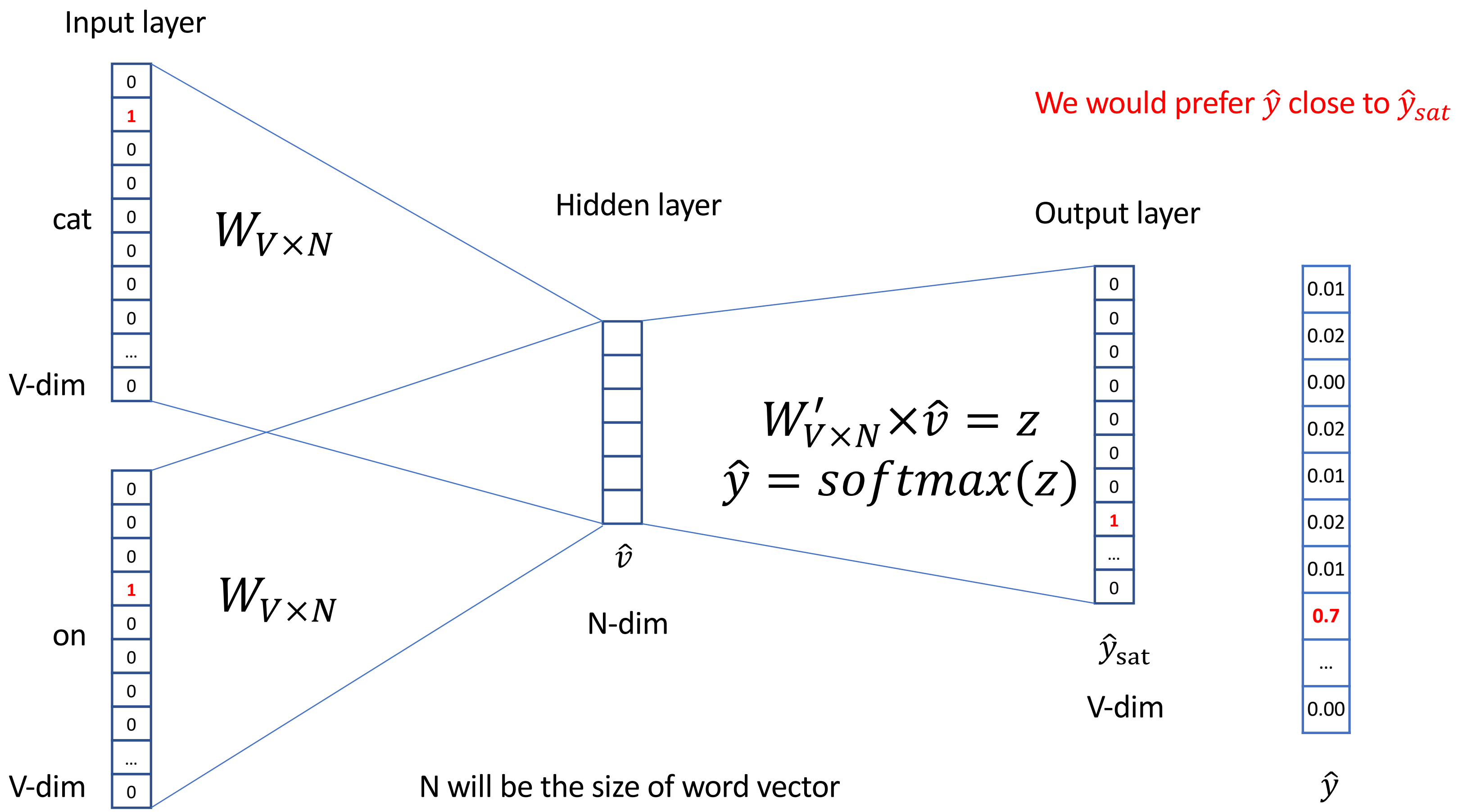
$$\begin{bmatrix} 0.1 & 2.4 & 1.6 & 1.8 & 0.5 & 0.9 & \dots & \dots & \dots & 3.2 \\ 0.5 & 2.6 & 1.4 & 2.9 & 1.5 & 3.6 & \dots & \dots & \dots & 6.1 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0.6 & 1.8 & 2.7 & 1.9 & 2.4 & 2.0 & \dots & \dots & \dots & 1.2 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \dots \\ 0 \end{bmatrix} = \begin{bmatrix} 2.4 \\ 2.6 \\ \dots \\ \dots \\ 1.8 \end{bmatrix}$$



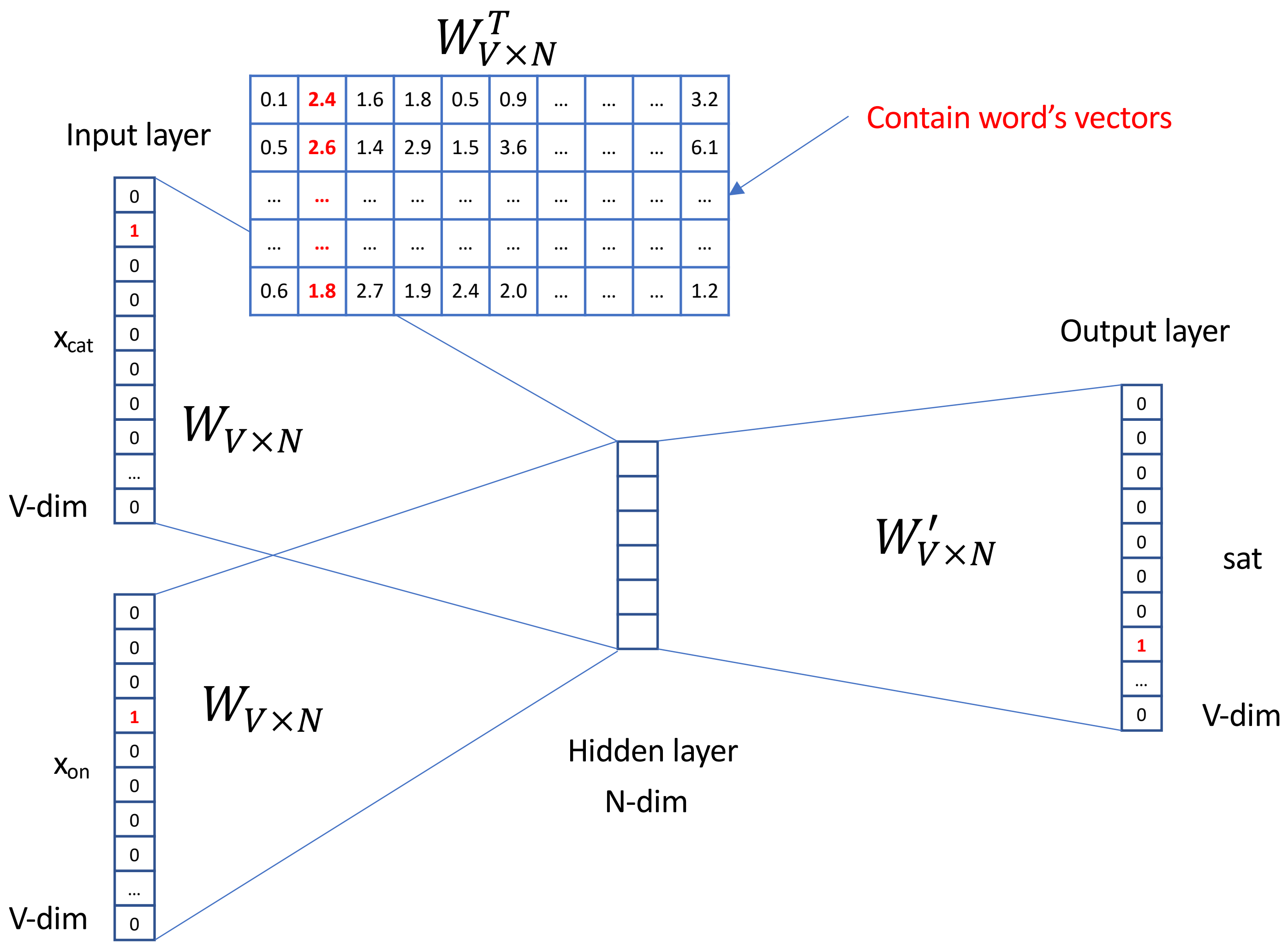
$W_{V\times N}^{T}$
$\times x_{on} = v_{on}$
0.1 2.4 1.6 1.8 0.5 0.9 … … … 3.2
0.5 2.6 1.4 2.9 1.5 3.6 … … … 6.1
0 0 0 1 0 0 0 0 … 0
1.8 2.9 … … 1.9
Input layer
x_{cat}
V-dim
x_{on}
V-dim
$W_{V\times N}^{T} \times x_{cat} = v_{cat}$
+
$W_{V\times N}^{T} \times x_{on} = v_{on}$
$\hat{v} = \frac{v_{cat} + v_{on}}{2}$
Hidden layer
N-dim
Output layer
sat
V-dim



N will be the size of word vector



N will be the size of word vector



We can consider either W or W' as the word's representation. Or even take the average.

Some interesting results

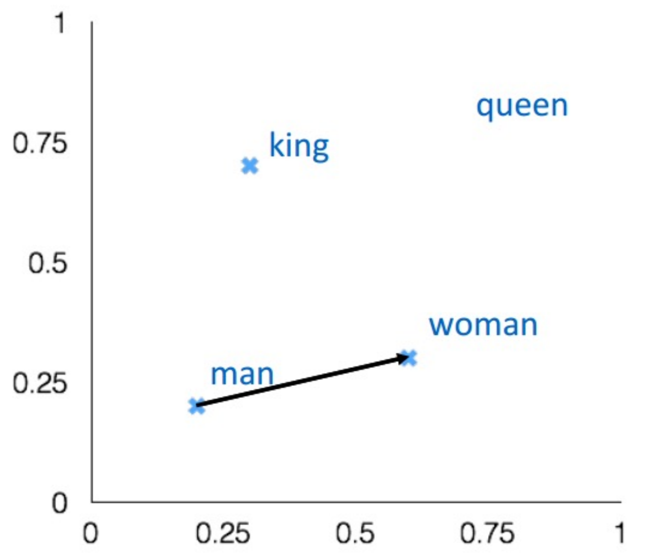
Word Analogies

Test for linear relationships, examined by Mikolov et al. (2014)

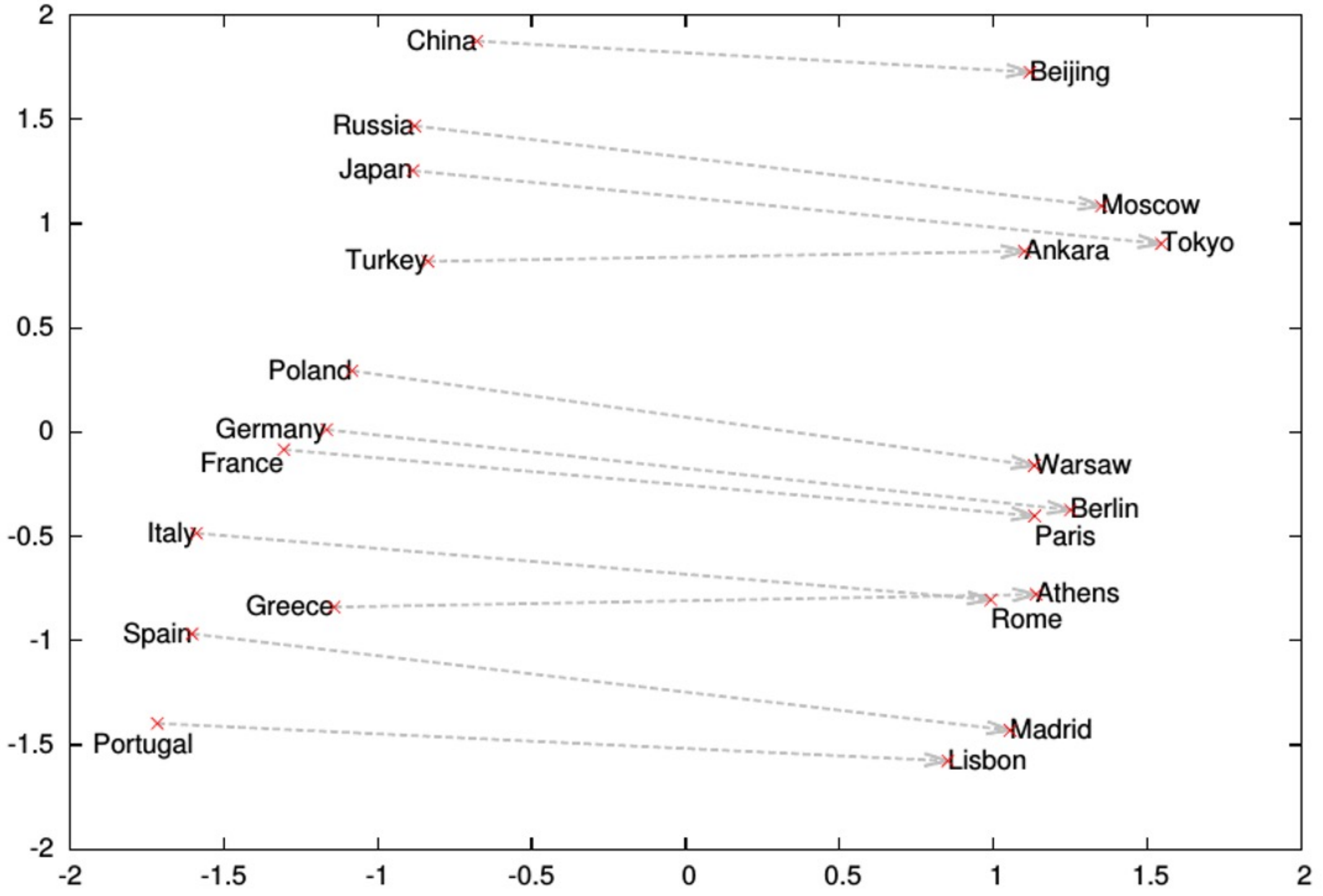
$\boxed{a:b :: c:?}$ $\longrightarrow$ $a = \arg\max_x \frac{}{||w_b - w_a + w_c||}$

man:woman :: king:?

+	king	[0.30 0.70]
-	man	[0.20 0.20]
+	woman	[0.60 0.30]
	queen	[0.70 0.80]

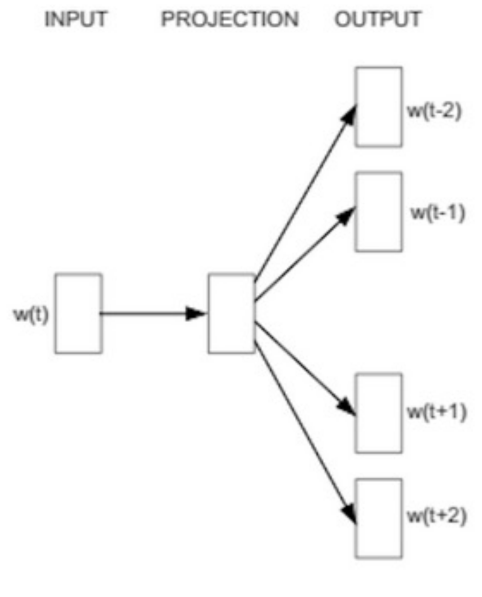


Word analogies



Skip gram

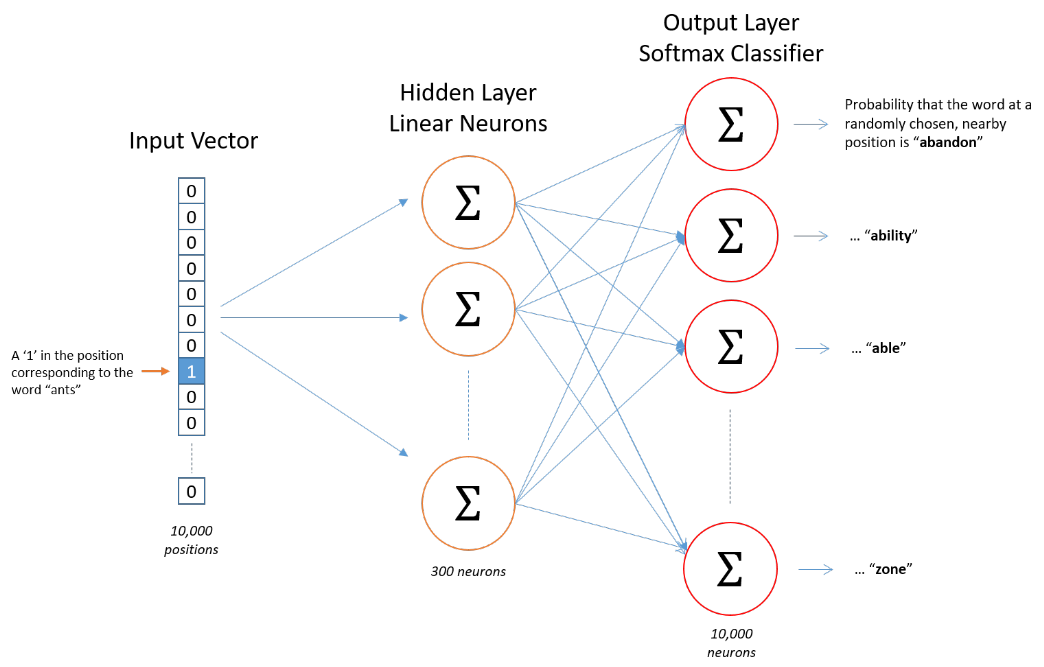
- Skip gram – alternative to CBOW
  - Start with a single word embedding and try to predict the surrounding words.
  - Much less well-defined problem, but works better in practice (scales better).



Skip-gram

Skip gram

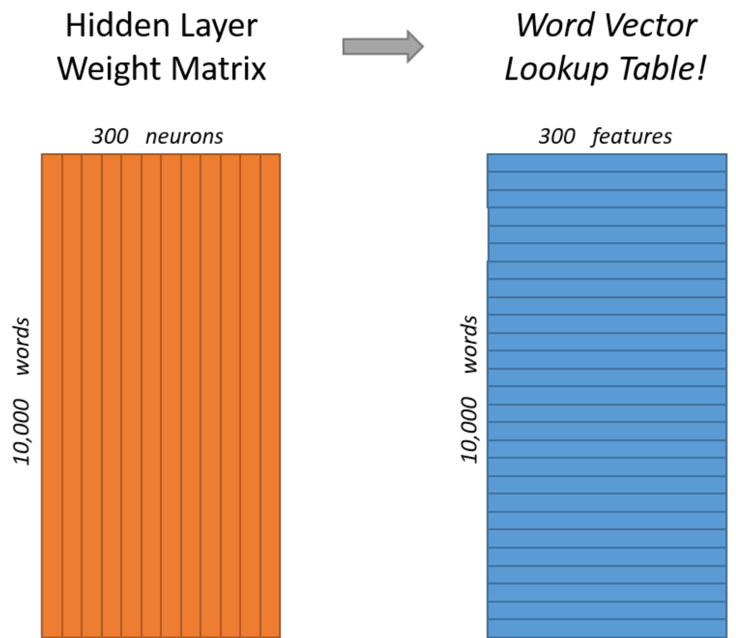
- Map from center word to probability on surrounding words. One input/output unit below.
    - There is no activation function on the hidden layer neurons, but the output neurons use softmax.



Skip gram example

- Vocabulary of 10,000 words.
- Embedding vectors with 300 features.
- So the hidden layer is going to be represented by a weight matrix with 10,000 rows (multiply by vector on the left).

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = \begin{bmatrix} 10 & 12 & 19 \end{bmatrix}$$



Skip gram/CBOW intuition

- Similar “contexts” (that is, what words are likely to appear around them), lead to similar embeddings for two words.
- One way for the network to output similar context predictions for these two words is if *the word vectors are similar*. So, if two words have similar contexts, then the network is motivated to learn similar word vectors for these two words!

Word2vec shortcomings

- **Problem:** 10,000 words and 300 dim embedding gives a large parameter space to learn. And 10K words is minimal for real applications.

- Slow to train, and need lots of data, particularly to learn uncommon words.

Word2vec improvements: word pairs and phrases

- **Idea:** Treat common word pairs or phrases as single "words."
  - E.g., Boston Globe (newspaper) is different from Boston and Globe separately. Embed Boston Globe as a single word/phrase.
- **Method:** make phrases out of words which occur together often relative to the number of individual occurrences. Prefer phrases made of infrequent words in order to avoid making phrases out of common words like "and the" or "this is".
- **Pros/cons:** Increases vocabulary size but decreases training expense.
- **Results:** Led to 3 million "words" trained on 100 billion words from a Google News dataset.

Word2vec improvements: subsample frequent words

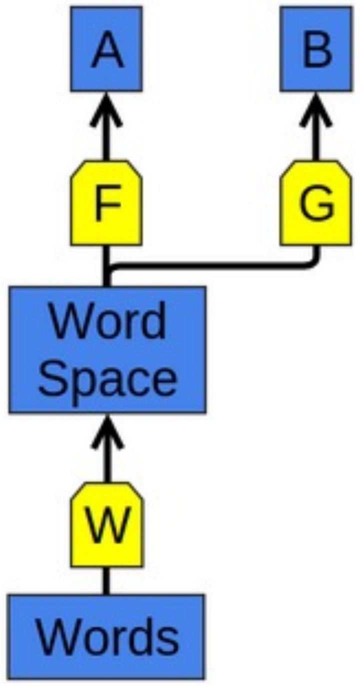
- **Idea:** Subsample frequent words to decrease the number of training examples.
  - The probability that we cut the word is related to the word's frequency. More common words are cut more.
  - Uncommon words (anything < 0.26% of total words) are kept
  - E.g., remove some occurrences of "the."
- **Method:** For each word, cut the word with probability related to the word's frequency.
- **Benefits:** If we have a window size of 10, and we remove a specific instance of "the" from our text:
  - As we train on the remaining words, "the" will not appear in any of their context windows.

Word2vec improvements: selective updates

- **Idea:** Use “Negative Sampling”, which causes each training sample to update only a small percentage of the model’s weights.
- **Observation:** A “correct output” of the network is a one-hot vector. That is, one neuron should output a 1, and *all* of the other thousands of output neurons to output a 0.
- **Method:** With negative sampling, randomly select just a small number of “negative” words (let’s say 5) to update the weights for. (In this context, a “negative” word is one for which we want the network to output a 0 for). We will also still update the weights for our “positive” word.

Word embedding applications

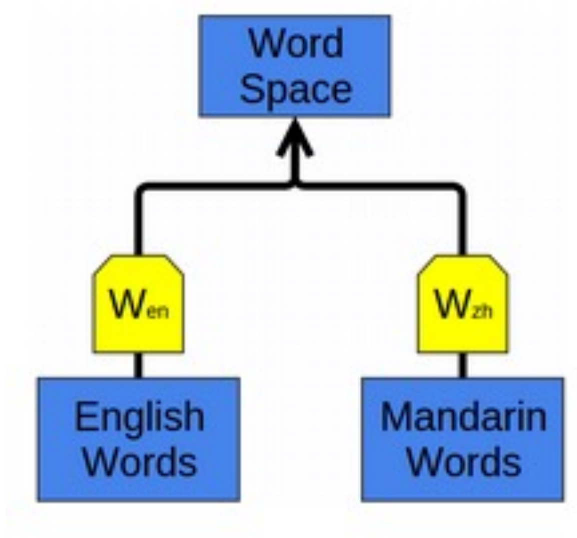
- The use of word representations… has become a key “secret sauce” for the success of many NLP systems in recent years, across tasks including named entity recognition, part-of-speech tagging, parsing, and semantic role labeling. (Luong *et al.* (2013))
- Learning a good representation on a task A and then using it on a task B is one of the major tricks in the Deep Learning toolbox.
  - Pretraining, transfer learning, and multi-task learning.
  - Can allow the representation to learn from more than one kind of data.



W and F learn to perform task A. Later, G can learn to perform B based on W.

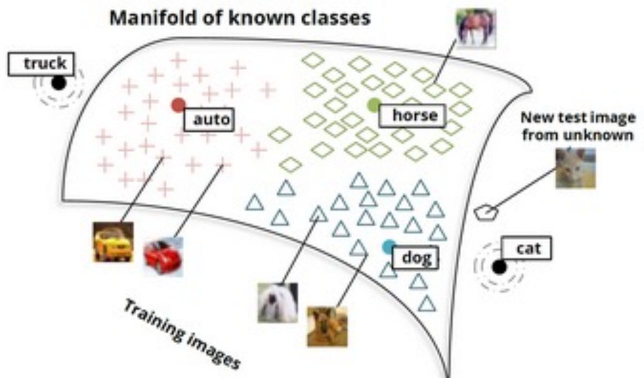
Word embedding applications

- Can learn to map multiple kinds of data into a single representation.
- E.g., bilingual English and Mandarin Chinese word-embedding as in Socher *et al.* (2013a).
- Embed as above, but words that are known as close translations should be close together.
- Words we *didn't know* were translations end up close together!
- Structures of two languages get pulled into alignment.

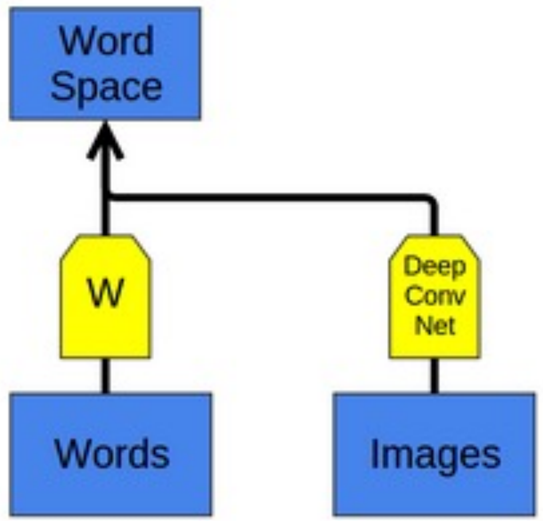


Word embedding applications

- Can apply to get a joint embedding of words and images or other multi-modal data sets.
- New classes map near similar existing classes: e.g., if ‘cat’ is unknown, cat images map near dog.



(Socher *et al.* (2013b))

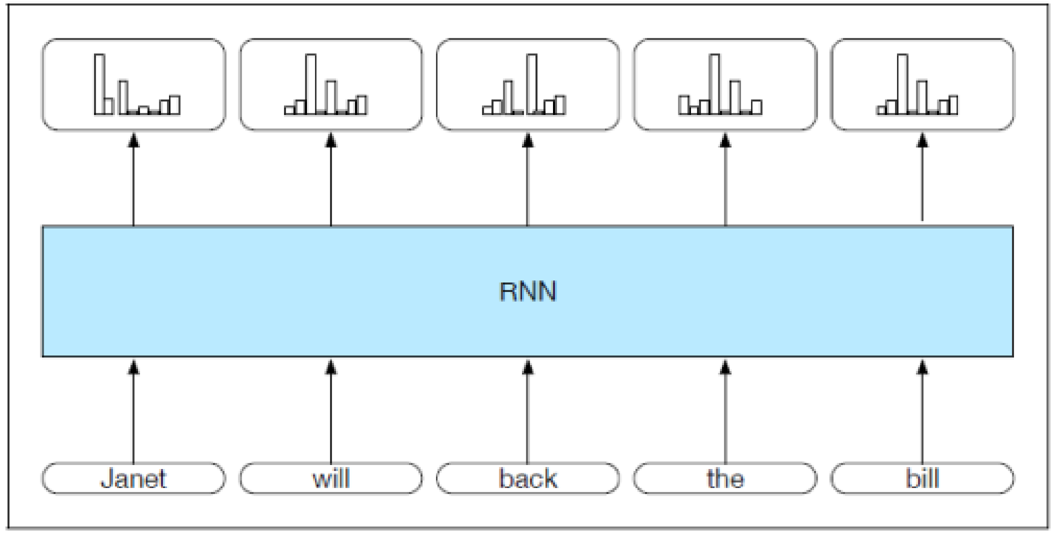


Word Embeddings

- **Encoder Decoder**
- Attention
- Transformers

Encoder-Decoder

- **RNN:** input sequence is transformed into output sequence in a one-to-one fashion



- **Goal:** Develop an architecture capable of generating *contextually appropriate*, *arbitrary length*, output sequences
- **Applications**:
  - Machine translation,
  - Summarization,
  - Question answering,
  - Dialogue modeling.

Simple recurrent neural network illustrated as a feed-forward network

Most significant change: new set of weights, U

- connect the hidden layer from the previous time step to the current hidden layer.
- determine how the network should make use of past context in calculating the output for the current input.

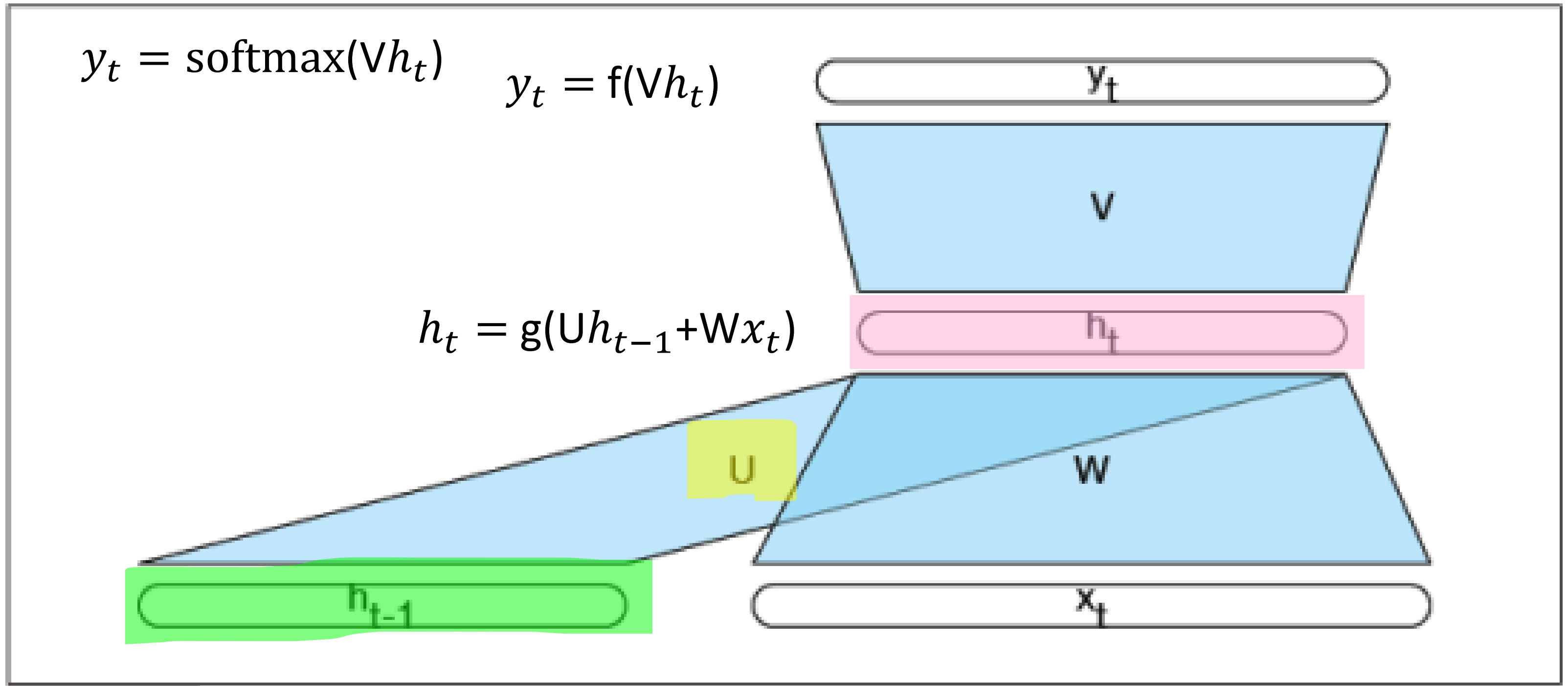
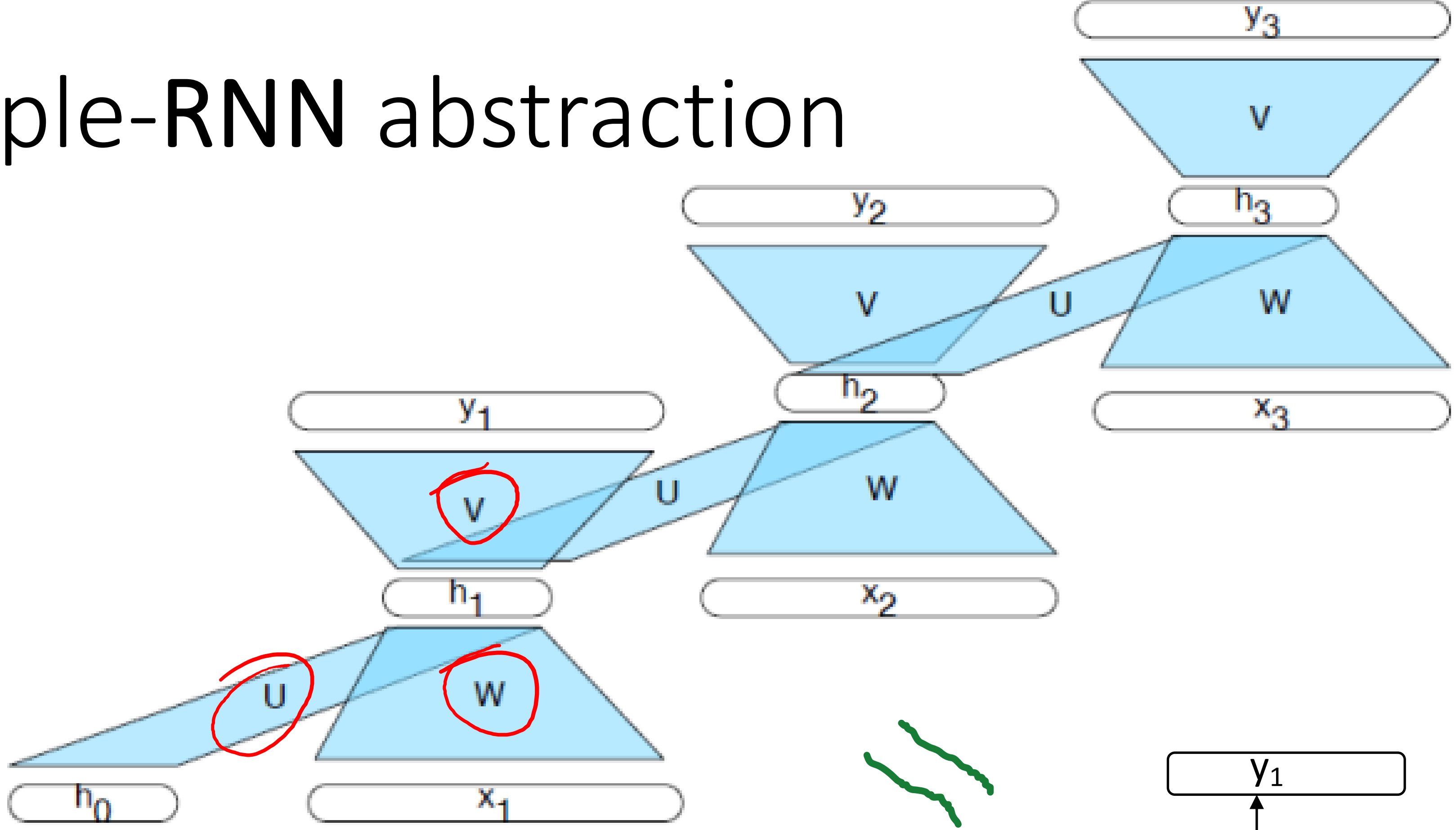
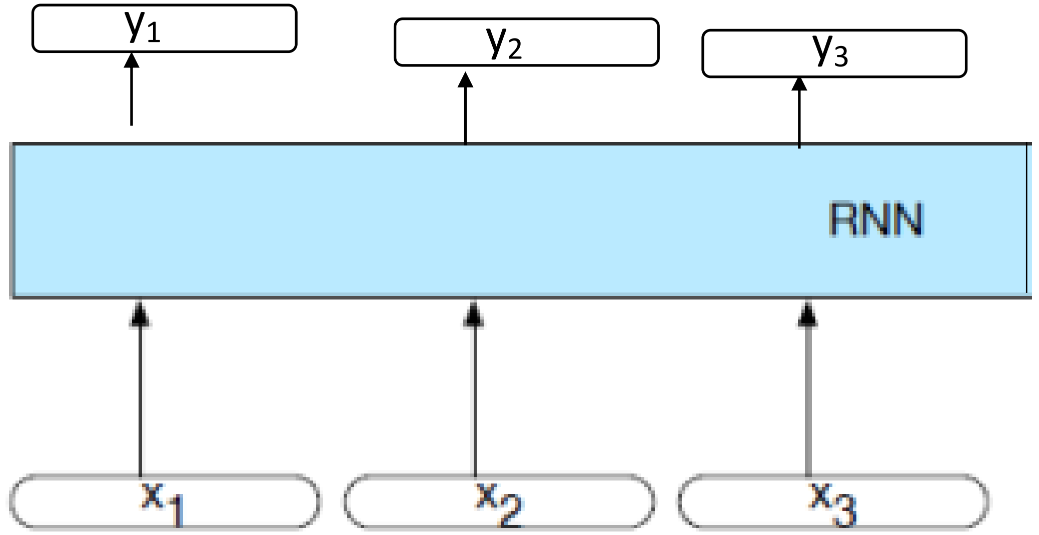


Figure 9.3 Simple recurrent neural network illustrated as a feed-forward network.

Simple-**RNN** abstraction

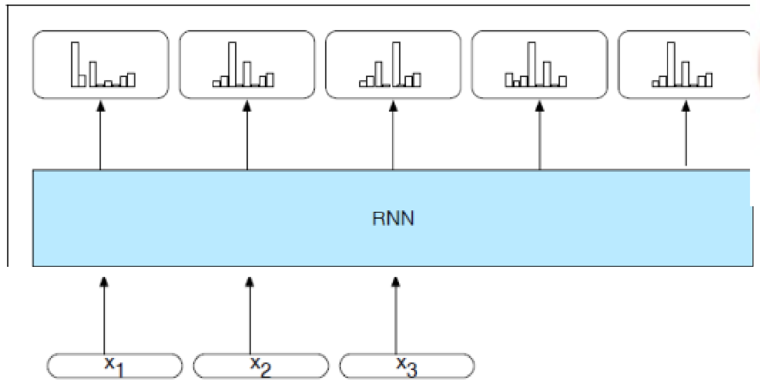


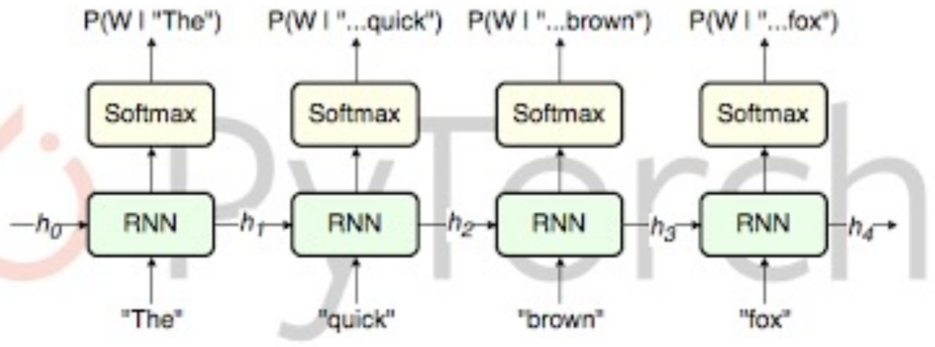
$\approx$



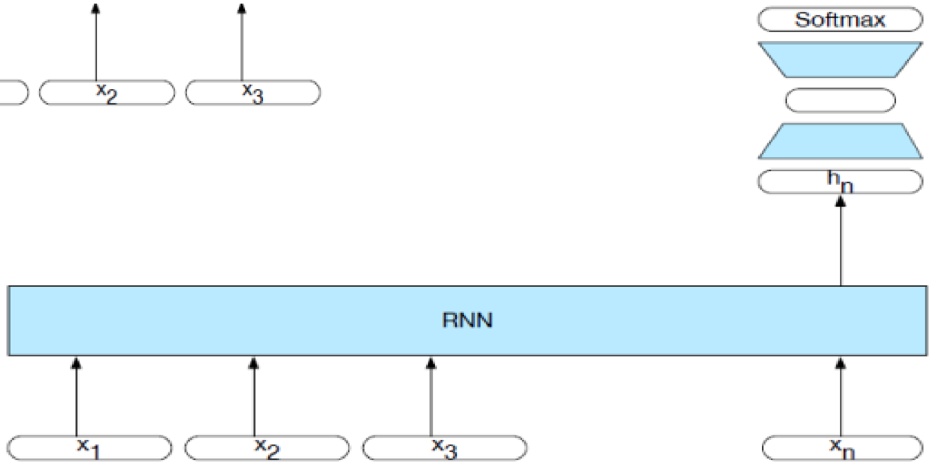
RNN Applications

- Language Modeling

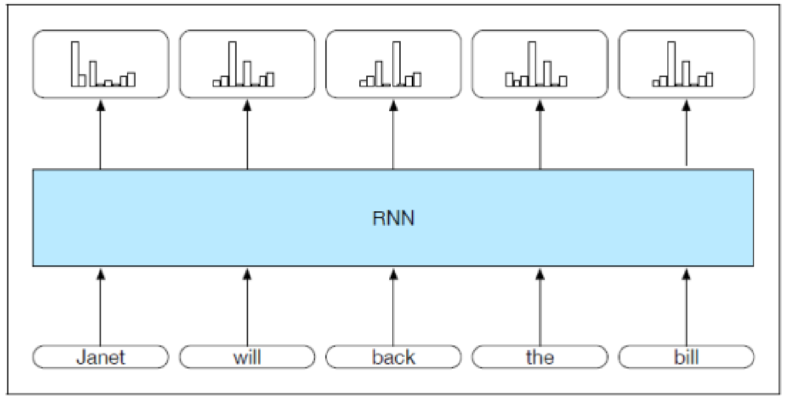




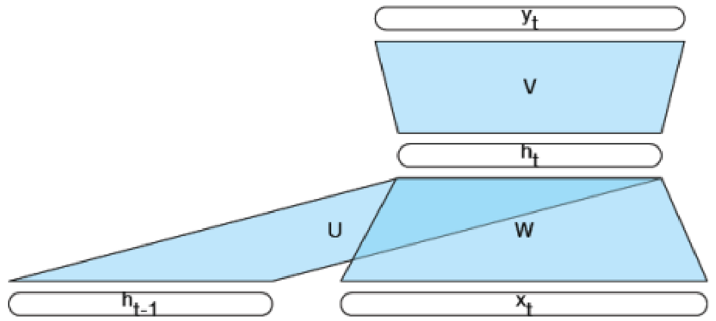
- Sequence Classification (Sentiment, Topic)



- Sequence to Sequence

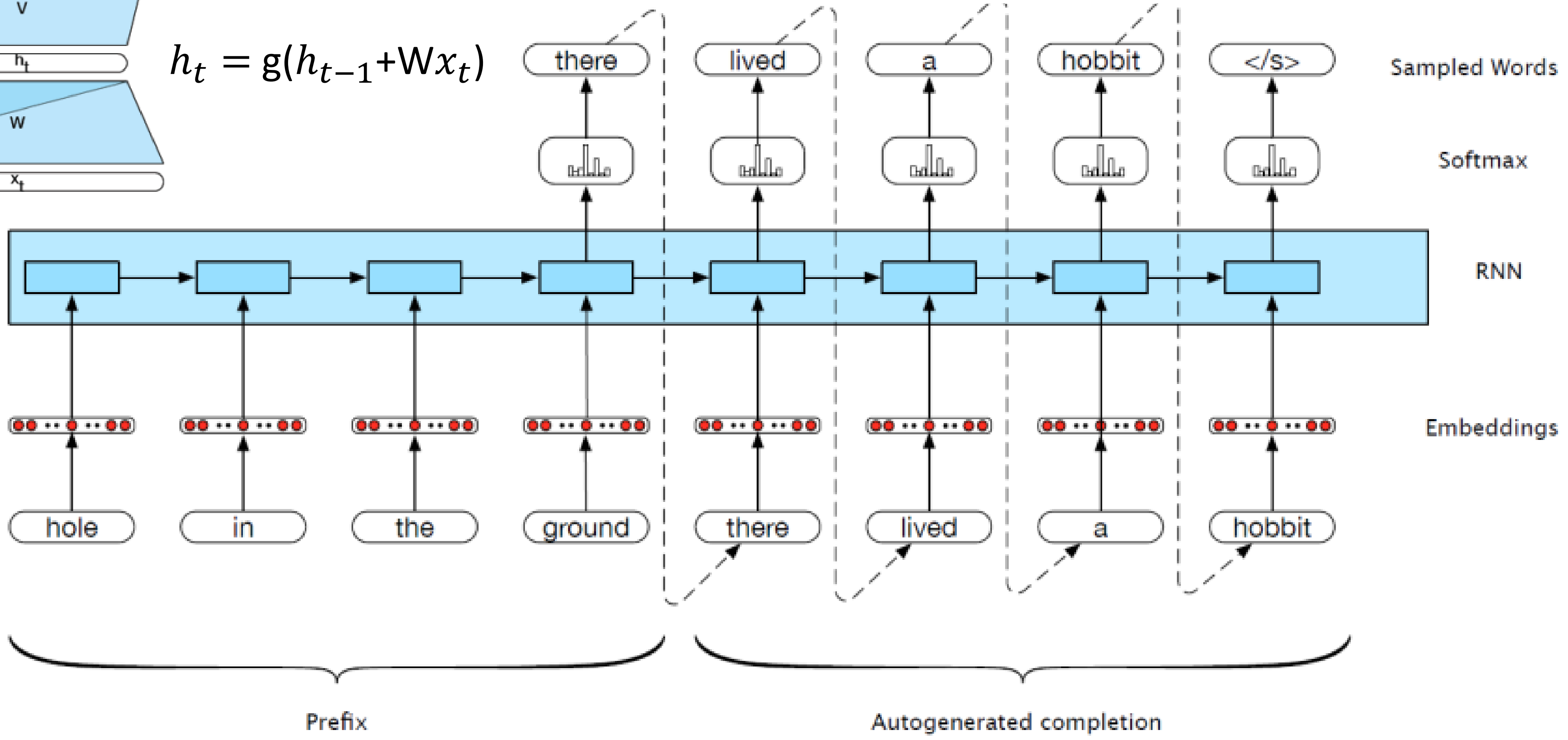


Sentence Completion using an RNN



$$y_t = \text{softmax}(Vh_t)$$

$$h_t = g(h_{t-1} + Wx_t)$$



- **Trained Neural Language Model** can be used to generate novel sequences
- Or to **complete** a given sequence (until end of sentence token <\s> is generated)

Extending (autoregressive) generation to Machine Translation

word generated at each time step is conditioned on word from previous step.

- Training data are parallel text e.g., English / French

  there lived a hobbit *vivait un hobbit*

  ……..

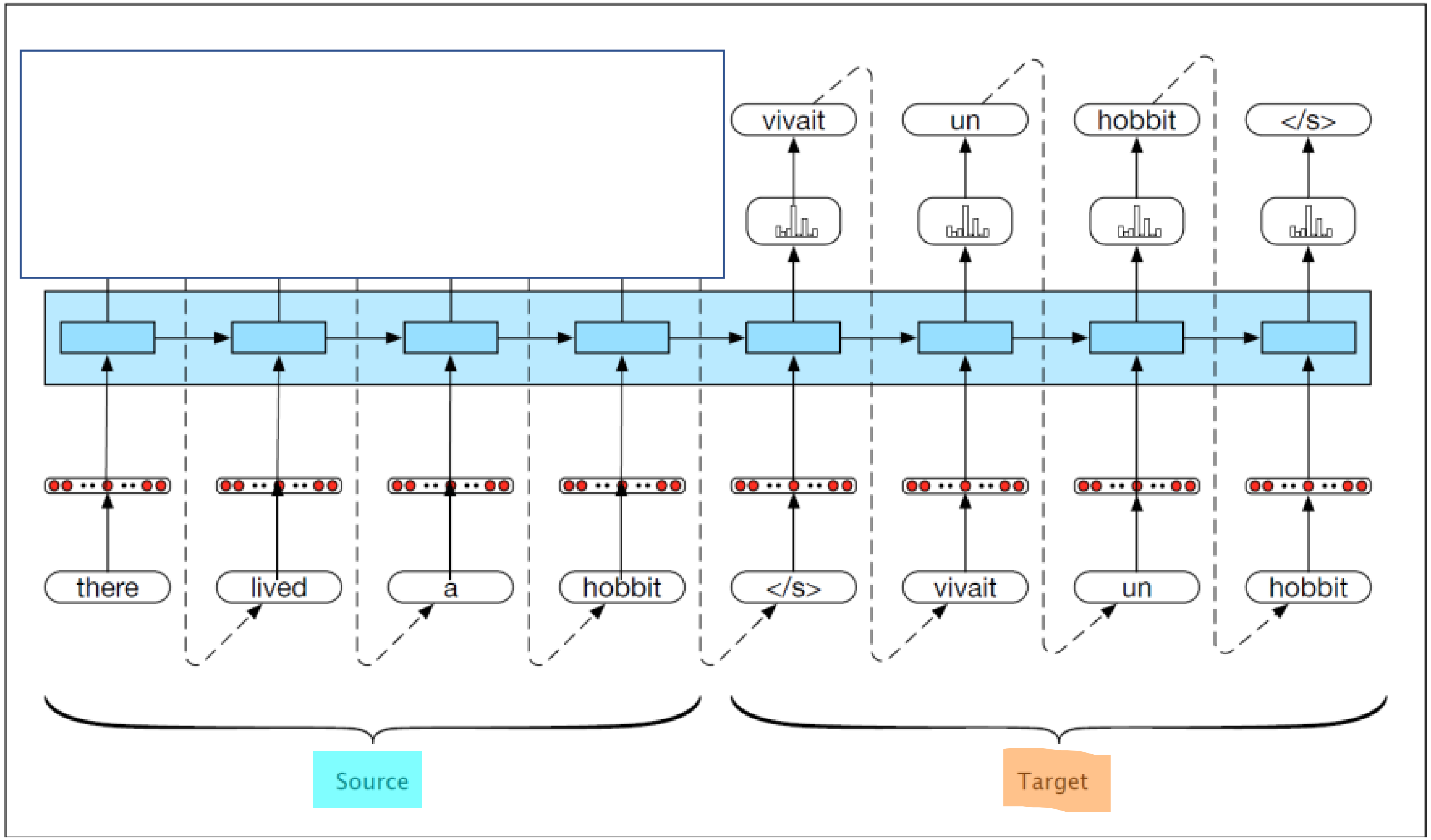
- Build an RNN language model on the concatenation of source and target

  there lived a hobbit <\s> *vivait un hobbit* <\s>

  ……..

Extending (autoregressive) generation to Machine Translation

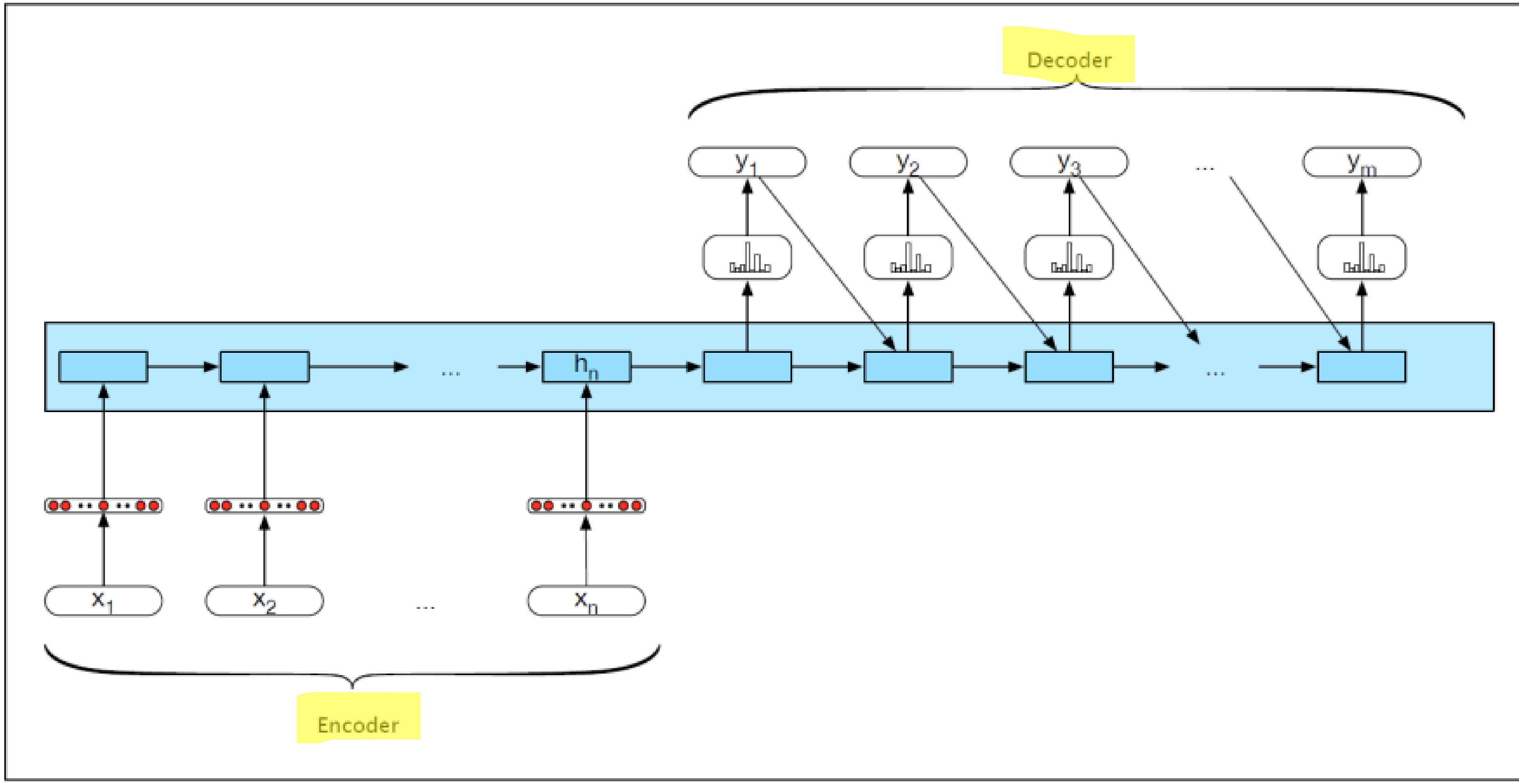
- Translation as Sentence Completion !



(simple) Encoder Decoder Networks

Limiting design choices

- **E** and **D** assumed to have the same internal structure (here RNNs)
- Final state of the **E** is the only context available to **D**
- this context is only available to **D** as its initial hidden state.



- Encoder generates a contextualized representation of the input (last state).
- Decoder takes that state and autoregressively generates a sequence of outputs

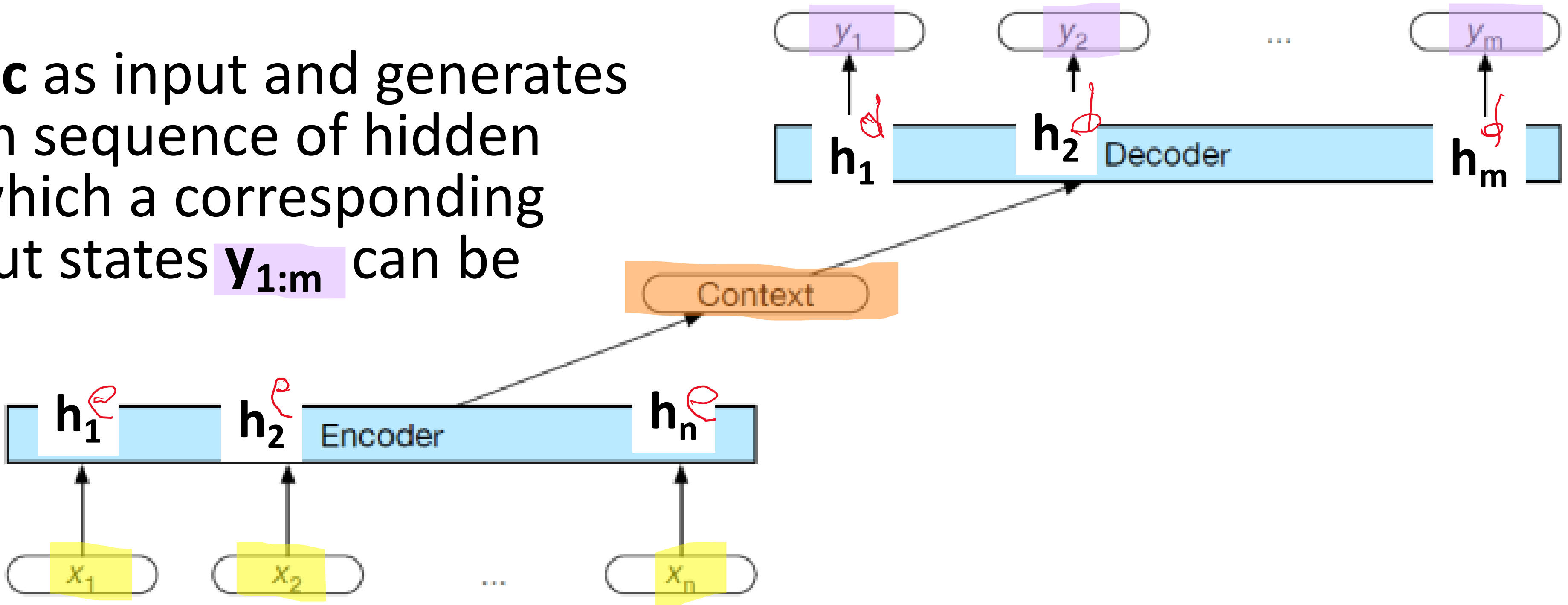
General Encoder Decoder Networks

Abstracting away from these choices

1. **Encoder**: accepts an input sequence, $\mathbf{x}_{1:n}$ and generates a corresponding sequence of contextualized representations, $\mathbf{h}_{1:n}$
2. **Context vector c**: function of $\mathbf{h}_{1:n}$ and conveys the essence of the input to the decoder.
3. **Decoder**: accepts **c** as input and generates an arbitrary length sequence of hidden states $\mathbf{h}_{1:m}$ from which a corresponding sequence of output states $\mathbf{y}_{1:m}$ can be obtained.

h^e_j

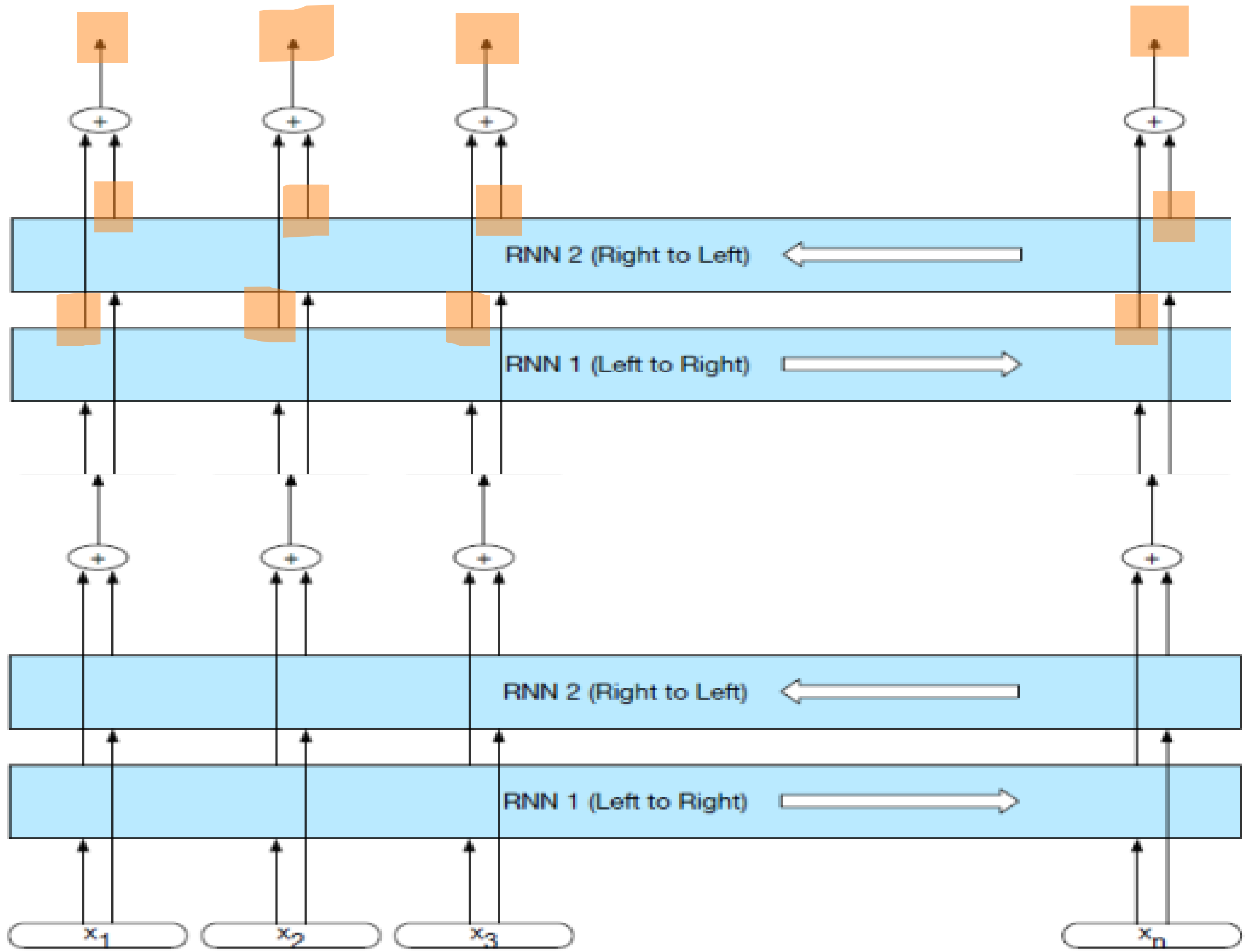
h^d_i



Popular architectural choices: Encoder

Widely used encoder design: **stacked Bi-LSTMs**

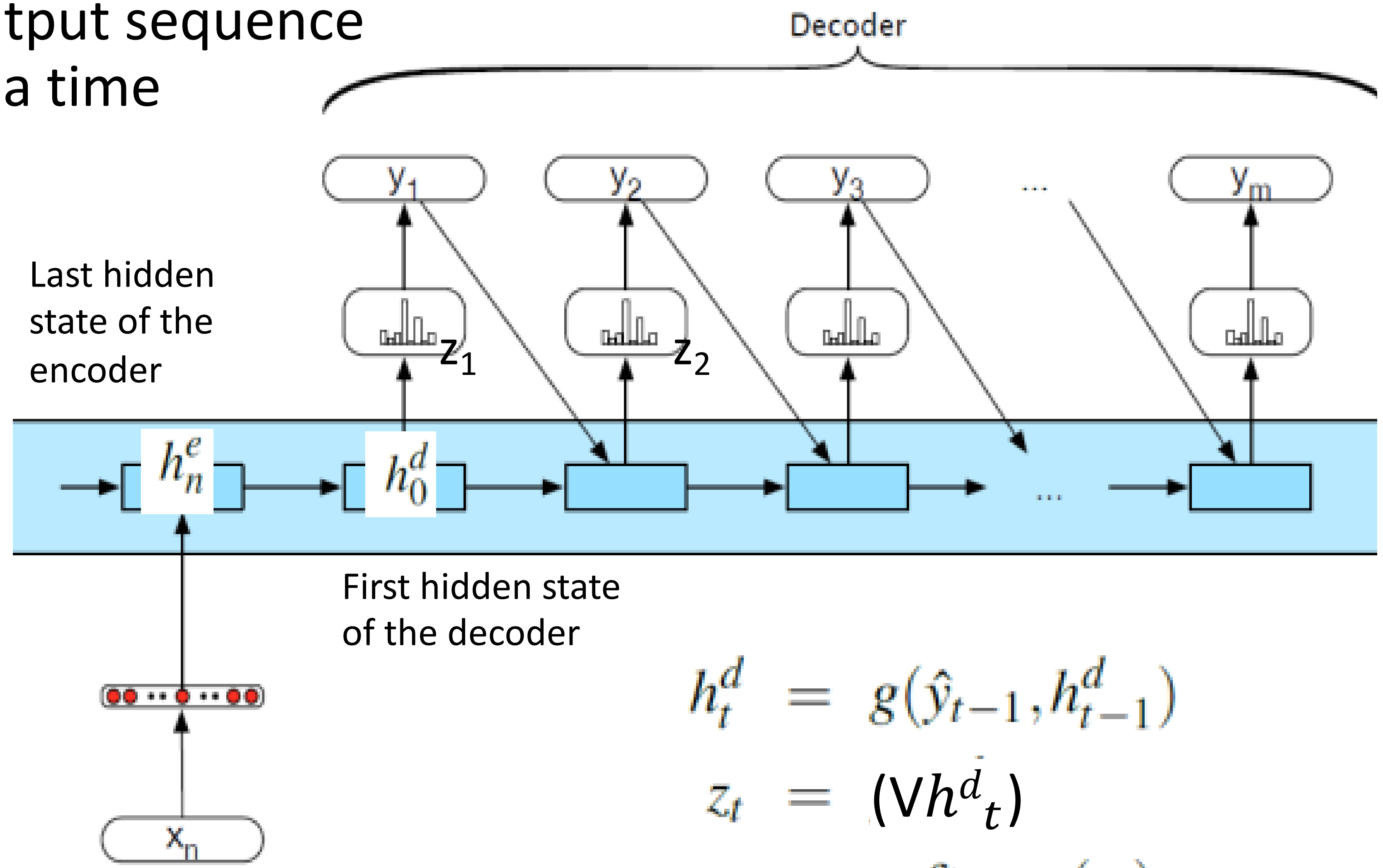
- Contextualized representations for each time step: **hidden states from top layers** from the forward and backward passes



Decoder Basic Design

- produce an output sequence an element at a time

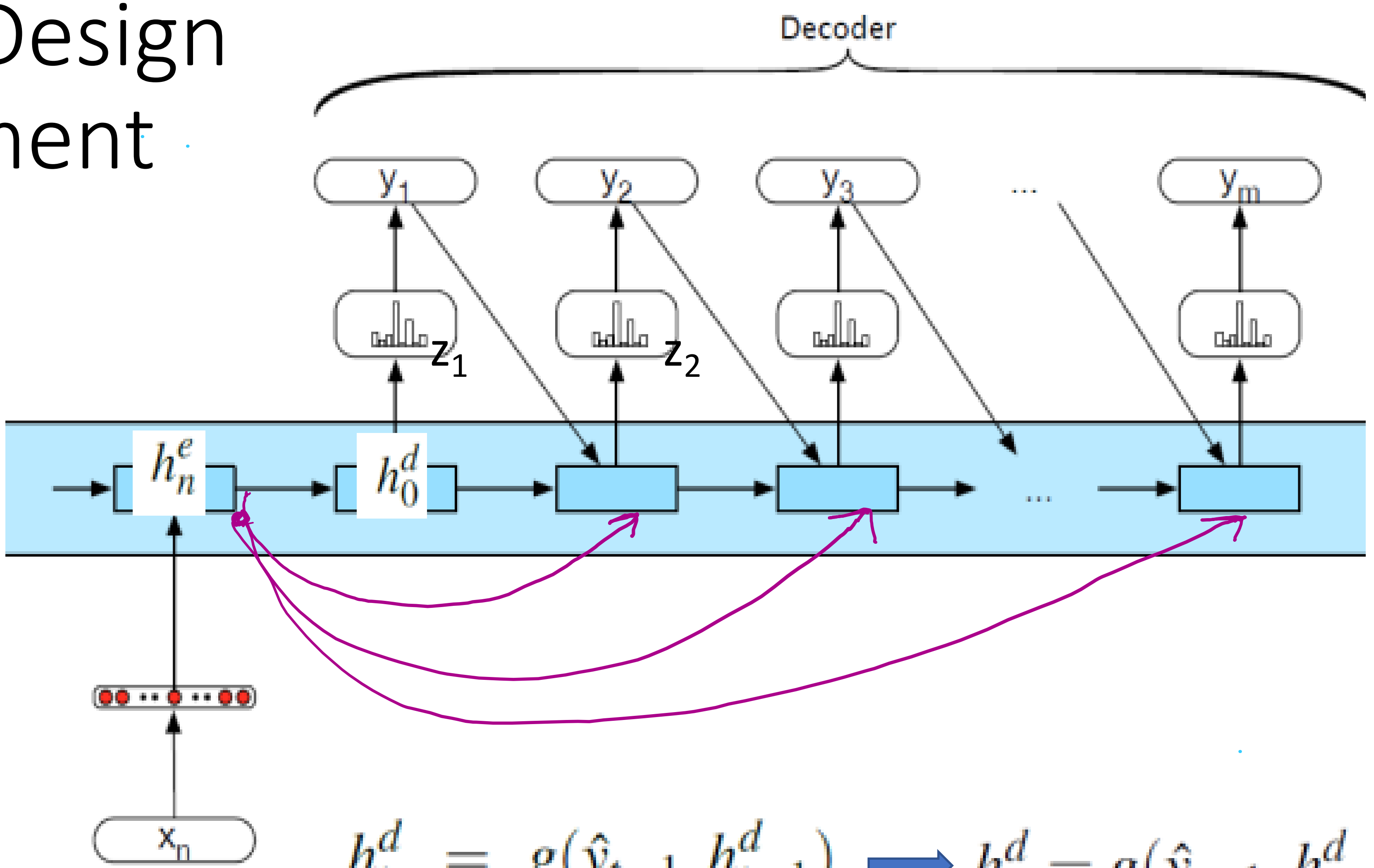
$$c = h_n^e$$
$$h_0^d = c$$


Decoder
y_1
y_2
y_3
...
y_m
z_1
z_2
Last hidden state of the encoder
h_n^e
h_0^d
...
First hidden state of the decoder
x_n

$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d)$$
$$z_t = (Vh^d{}_t)$$
$$y_t = \text{softmax}(z_t)$$

Decoder Design Enhancement

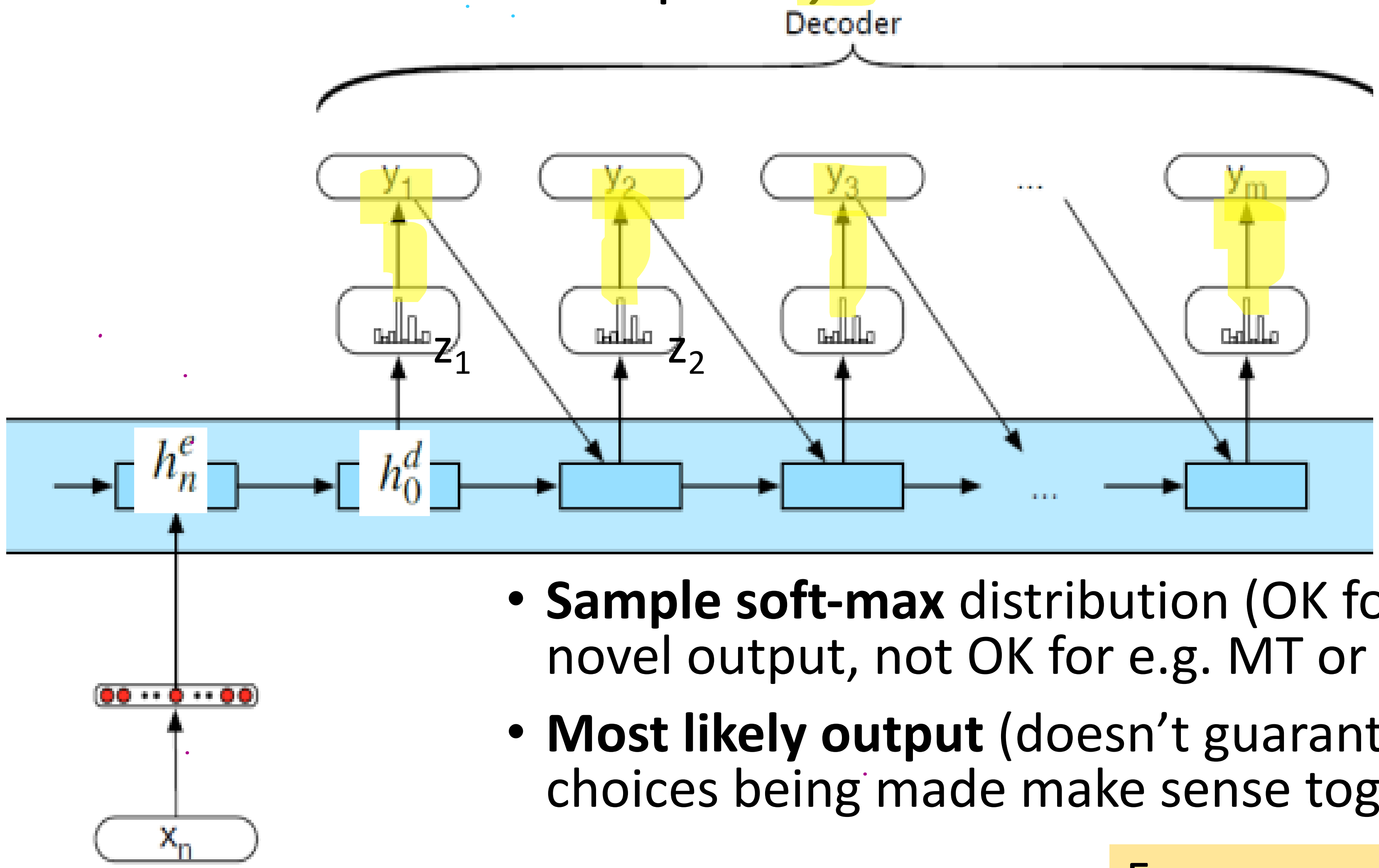
$$c = h_n^e$$
$$h_0^d = c$$



$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d) \Rightarrow h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c)$$
$$z_t = f(h_t^d)$$
$$y_t = \text{softmax}(z_t)$$

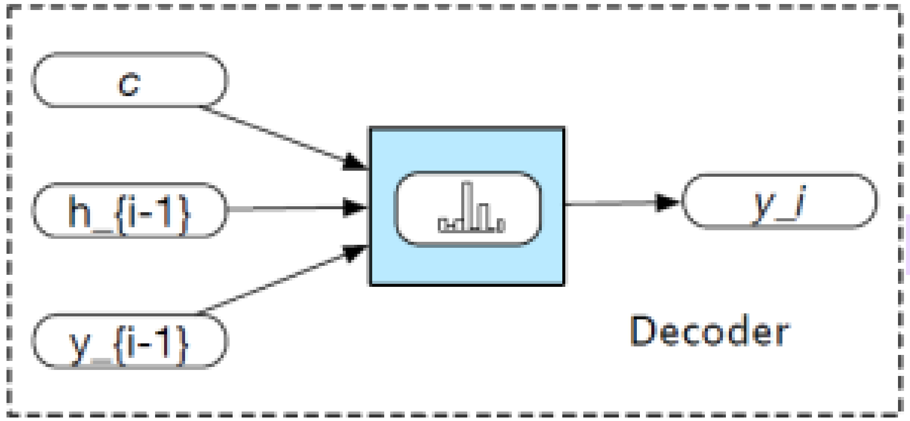
Context available at each step of decoding

Decoder: How output *y* is chosen

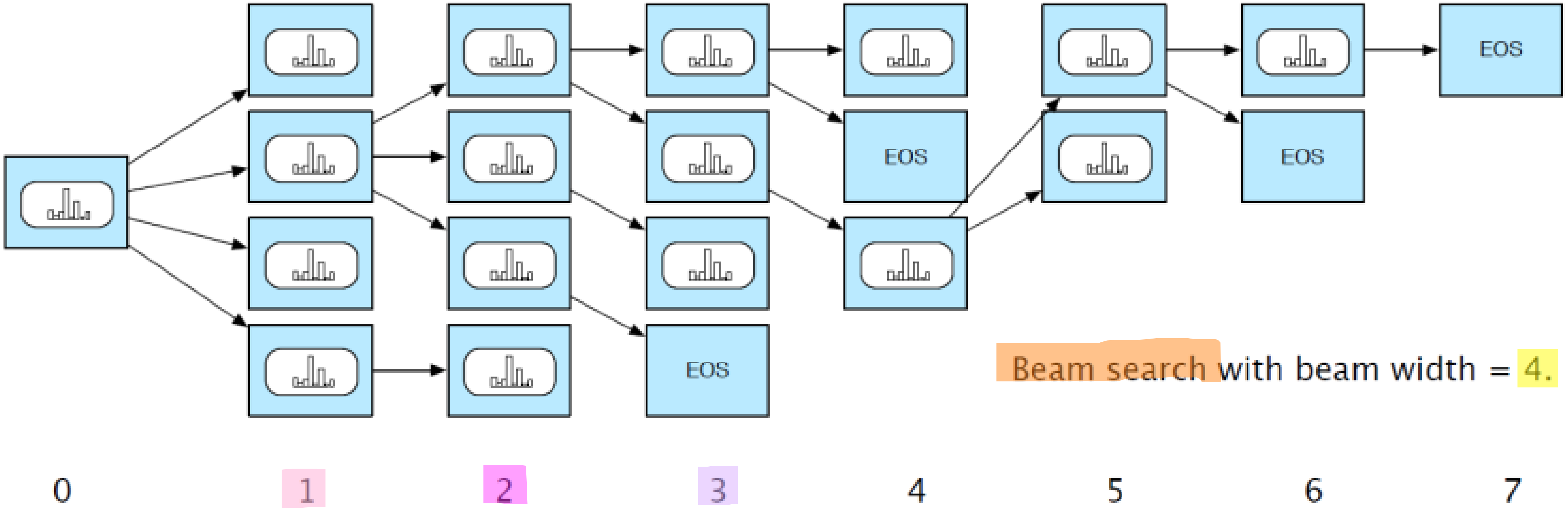


- **Sample soft-max** distribution (OK for generating novel output, not OK for e.g. MT or Summ)
- **Most likely output** (doesn’t guarantee individual choices being made make sense together)

For sequence labeling we used **Viterbi** – here not possible ☹



- 4 most likely "words" decoded from initial state
- Feed each of those in decoder and keep most likely 4 sequences of two words
- Feed most recent word in decoder and keep most likely 4 sequences of three words …….
- When EOS is generated. Stop sequence and reduce Beam by 1



Beam search with beam width = 4.

Word Embeddings

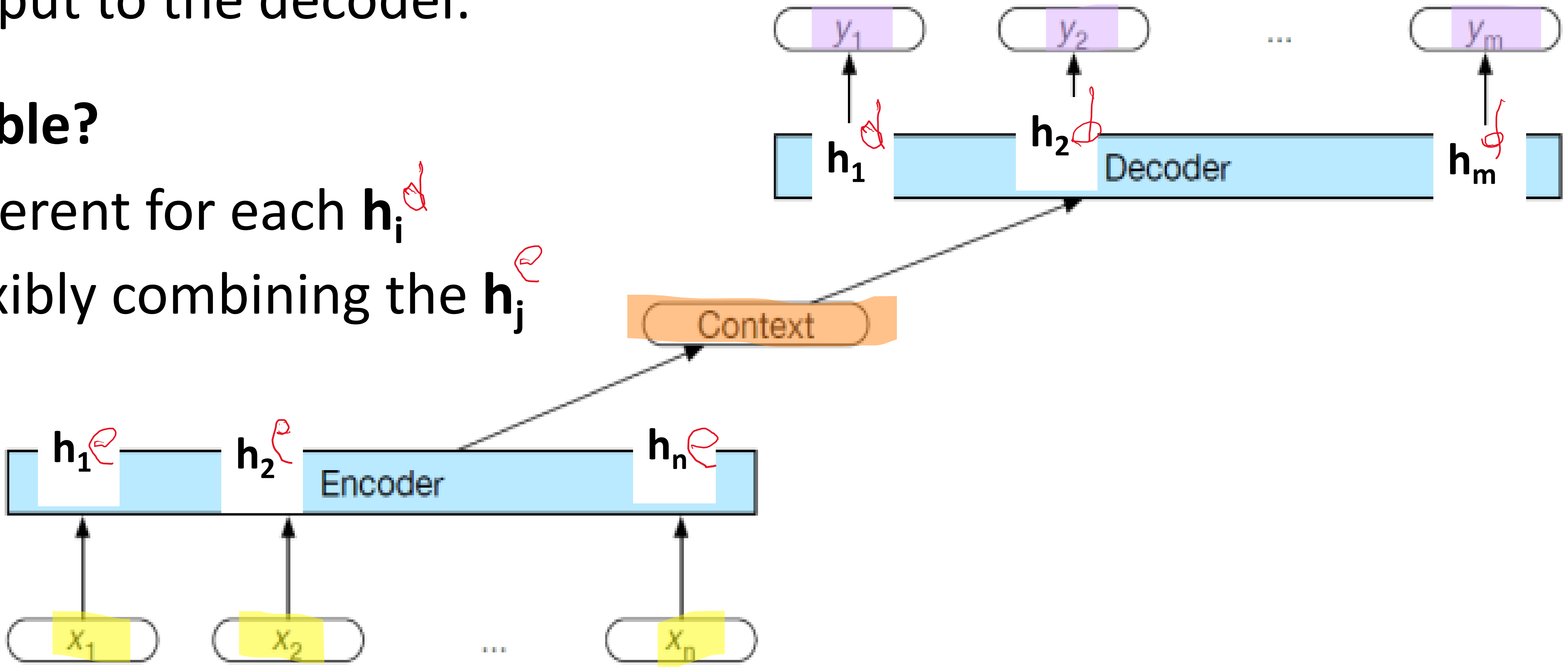
- Encoder Decoder
- **Attention**
- Transformers

Flexible context: Attention

Context vector c: function of $\mathbf{h}^{e}_{1:n}$ and conveys the essence of the input to the decoder.

Flexible?

- Different for each $\mathbf{h}^{d}_{i}$
- Flexibly combining the $\mathbf{h}^{e}_{j}$



Attention (1): dynamically derived context

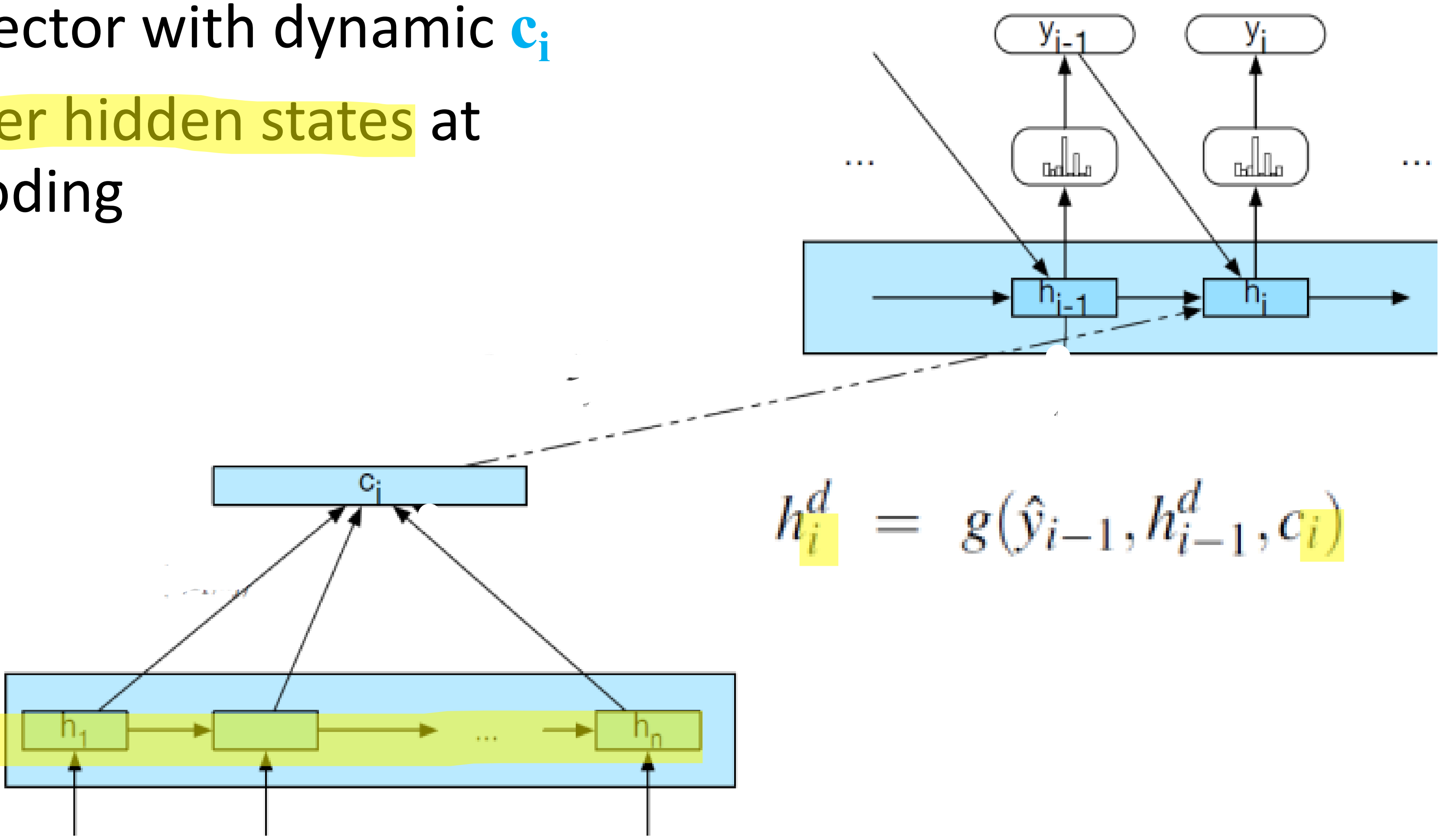
- Replace static context vector with dynamic $\mathbf{c_i}$
- derived from the encoder hidden states at each point i during decoding

Ideas:

- should be a linear combination of those states

$$c_i = \sum_j \alpha_{ij} h_j^e$$

- α_{ij} should depend on ?



$$h_i^d = g(\hat{y}_{i-1}, h_{i-1}^d, c_i)$$

Attention (2): computing c_i

- Compute a vector of scores that capture the relevance of each encoder hidden state to the decoder state h^d_{i-1}

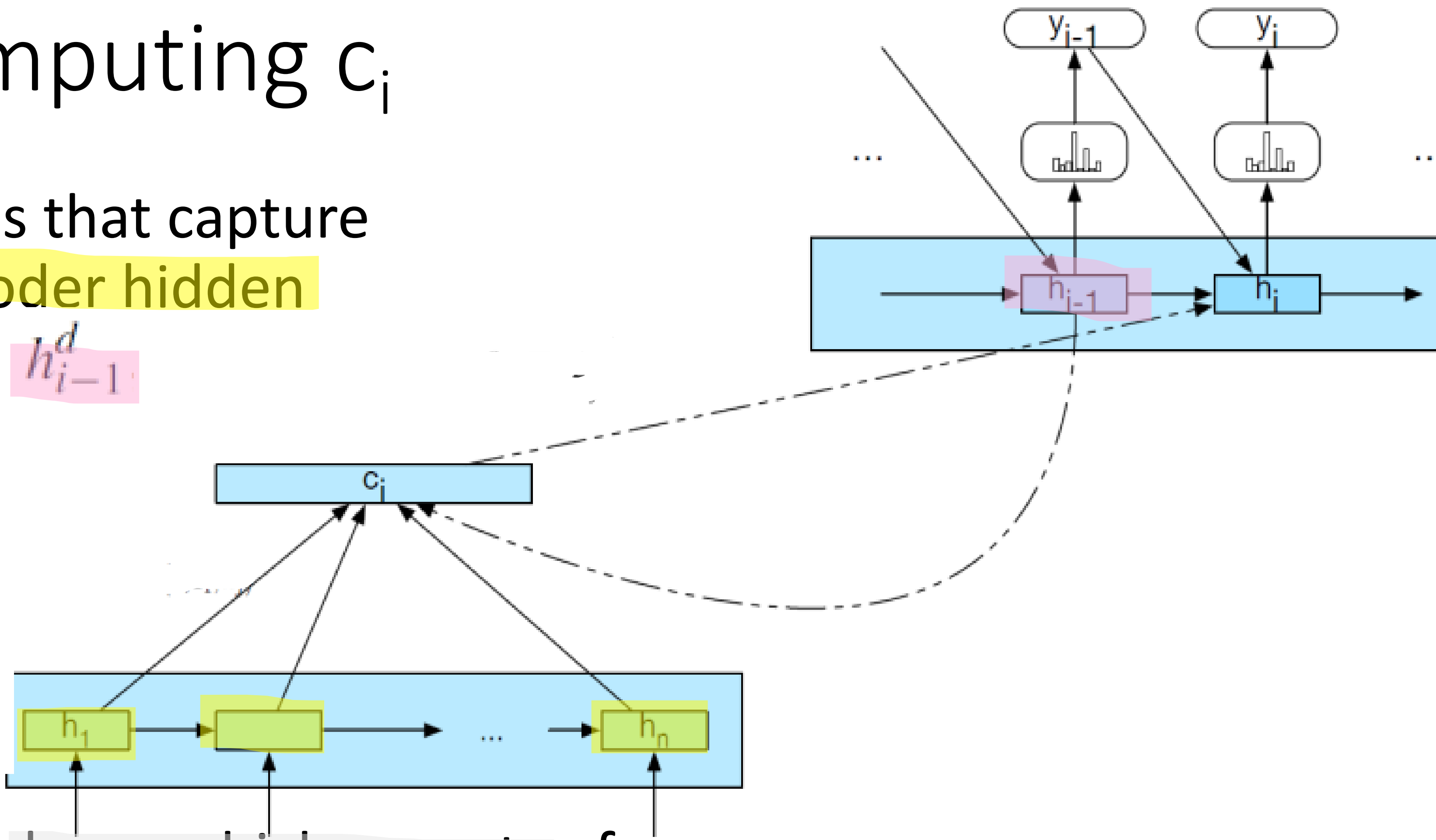
$$score(h^d_{i-1}, h^e_j)$$

- Just the similarity

$$score(h^d_{i-1}, h^e_j) = h^d_{i-1} \cdot h^e_j$$

- Give network the ability to learn which aspects of similarity between the decoder and encoder states are important to the current application.

$$score(h^d_{i-1}, h^e_j) = h^d_{t-1} W_s h^e_j$$



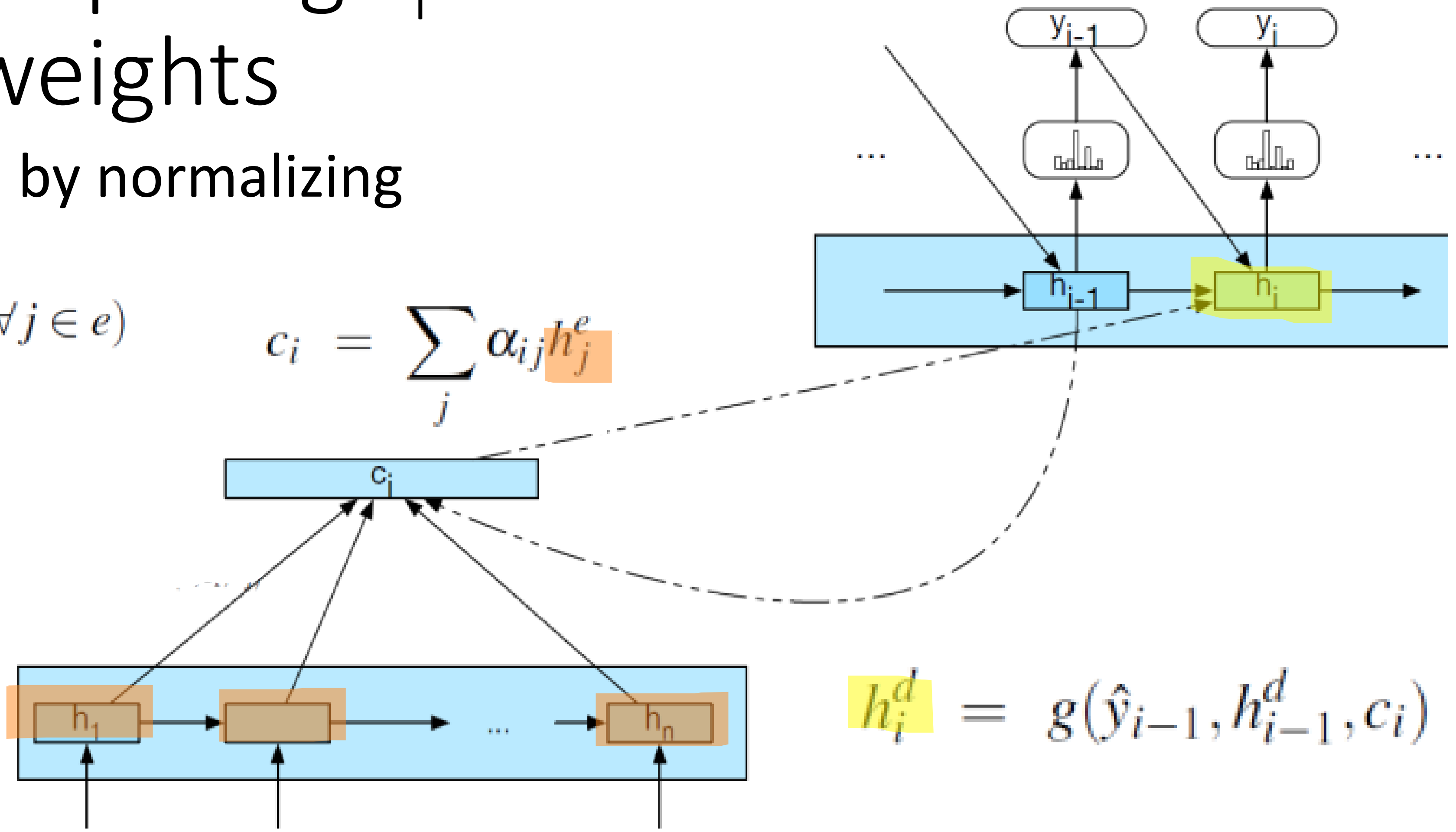
Attention (3): computing c_i From scores to weights

- Create vector of weights by normalizing scores

$$\alpha_{ij} = \text{softmax}(score(h^d_{i-1}, h^e_j) \ \forall j \in e)$$

$$= \frac{exp(score(h^d_{i-1}, h^e_j)}{\sum_k exp(score(h^d_{i-1}, h^e_k))}$$

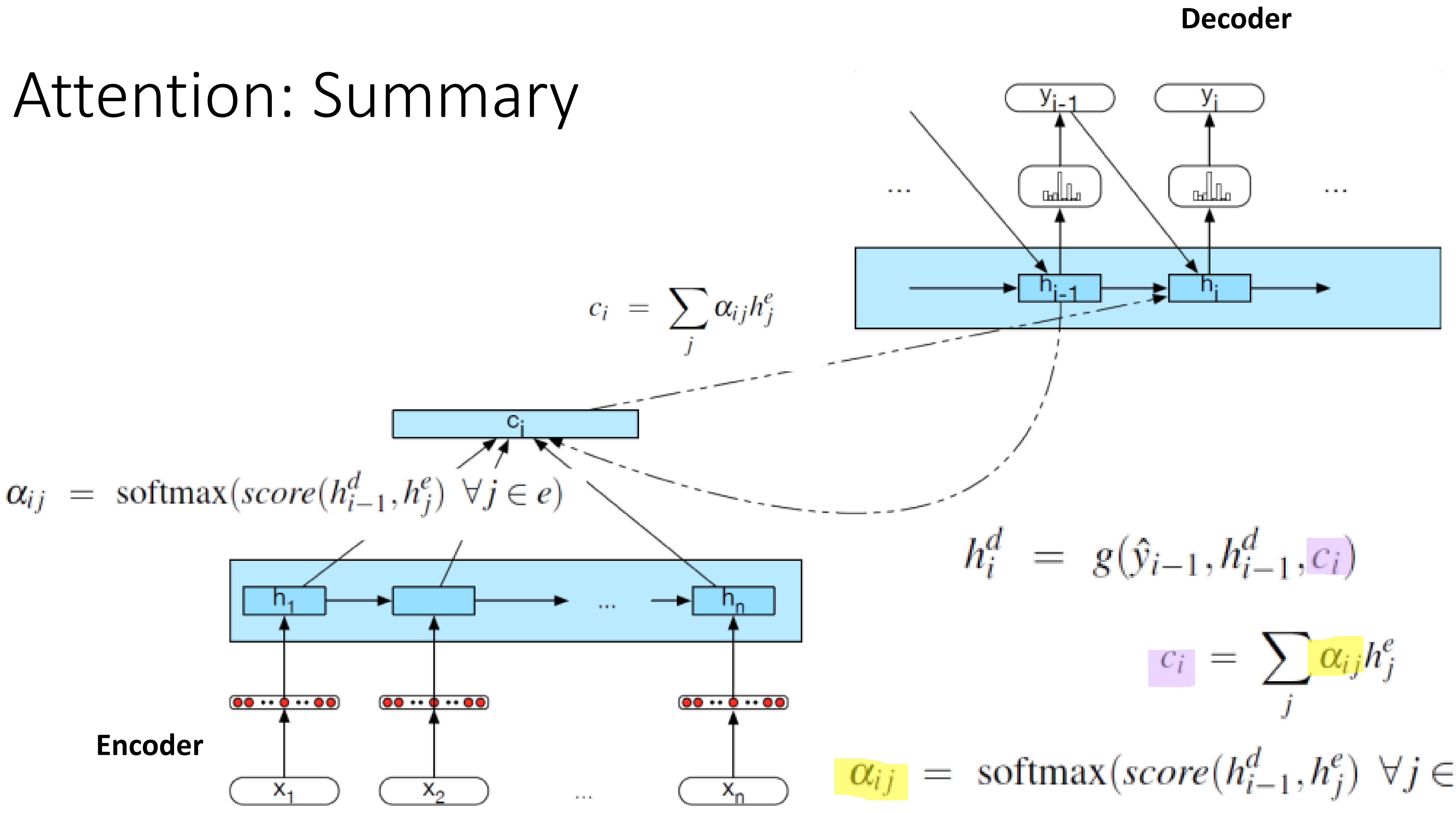
$$c_i = \sum_j \alpha_{ij} h^e_j$$



$$h^d_i = g(\hat{y}_{i-1}, h^d_{i-1}, c_i)$$

- **Goal achieved**: compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states.

Attention: Summary



$$c_i = \sum_j \alpha_{ij} h_j^e$$

$$\alpha_{ij} = \mathrm{softmax}(score(h_{i-1}^d, h_j^e)\ \forall j \in e)$$

$$h_i^d = g(\hat{y}_{i-1}, h_{i-1}^d, c_i)$$

$$c_i = \sum_j \alpha_{ij} h_j^e$$

$$\alpha_{ij} = \mathrm{softmax}(score(h_{i-1}^d, h_j^e)\ \forall j \in e)$$

Word embeddings

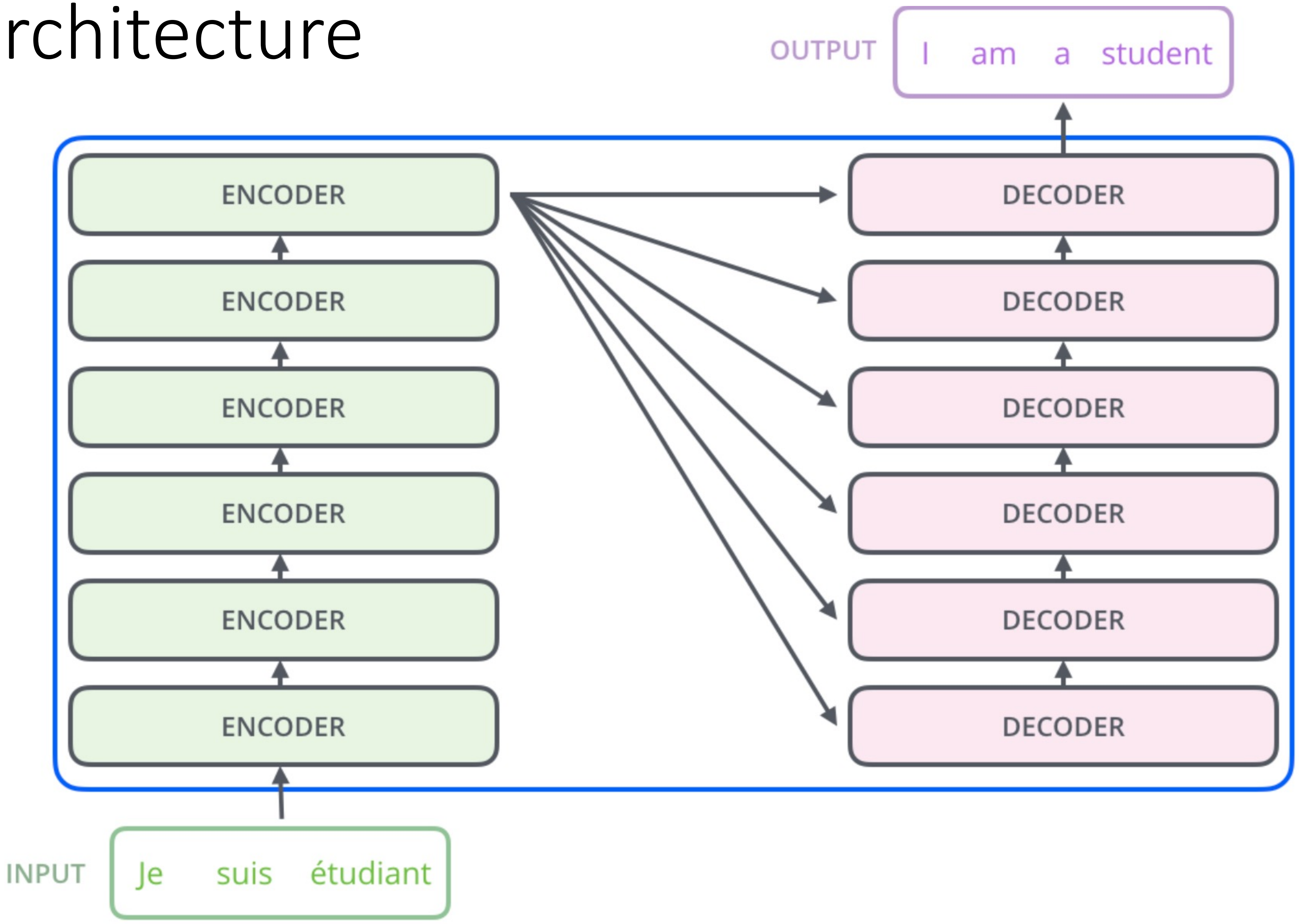
- Encoder Decoder
- Attention
- **Transformers** (self-attention)

Transformers (Attention is all you need 2017)

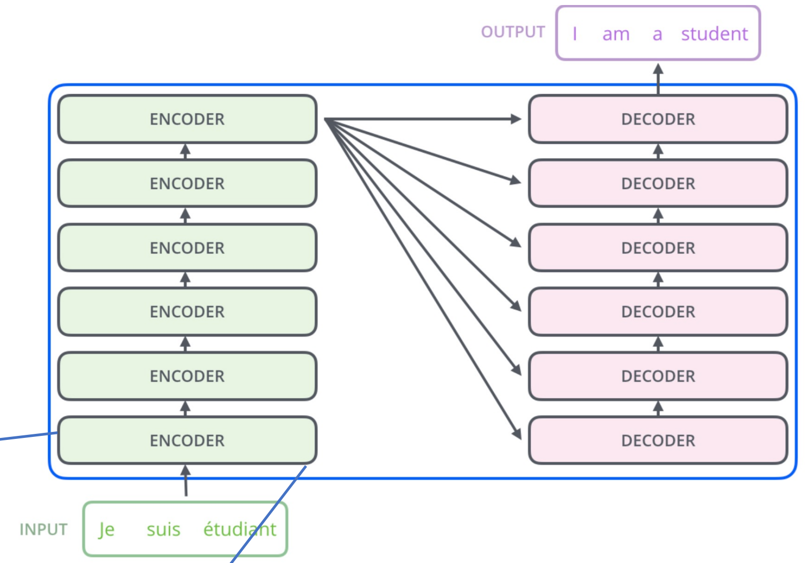
- **Just an introduction:** These are two valuable resources to learn more details on the architecture and implementation

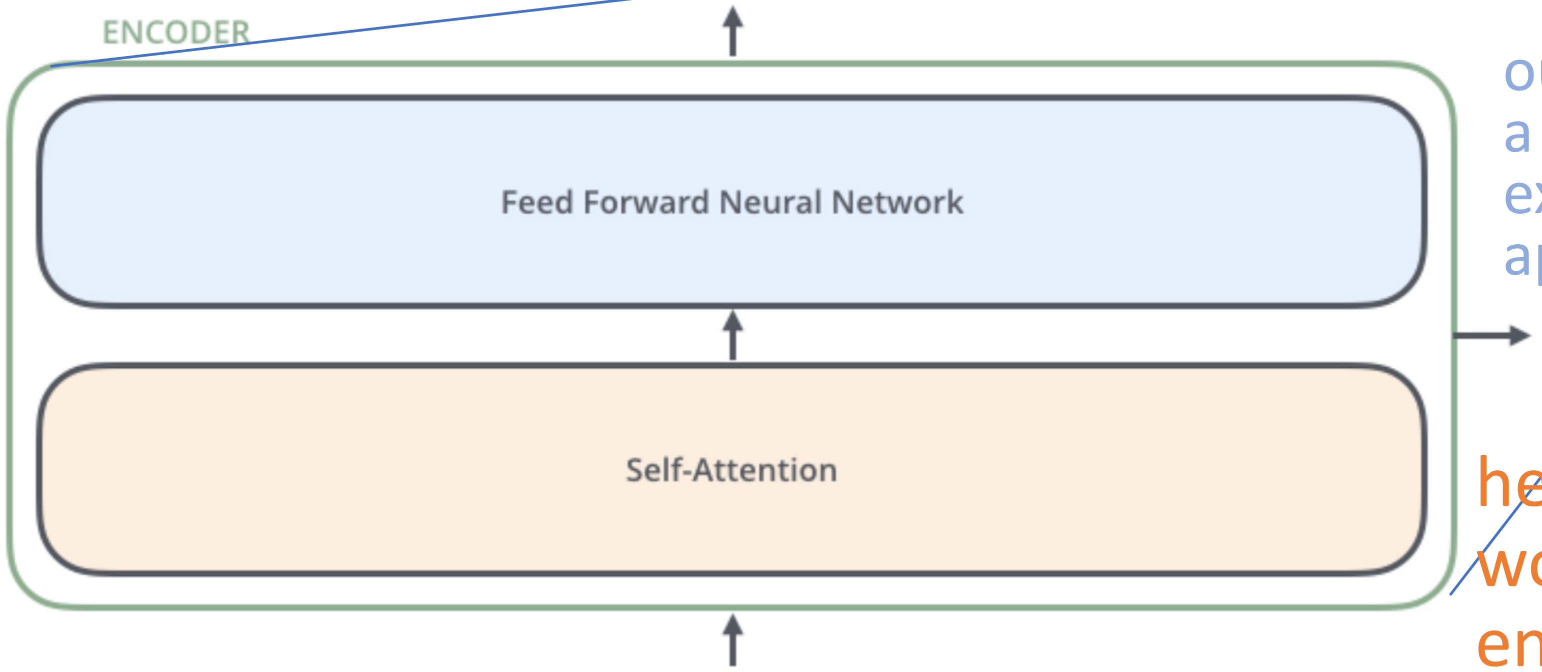
High-level architecture

- Will only look at the ENCODER(s) part in detail



The **encoders** are **all identical in structure** (yet they do not share weights). Each one is broken down into two sub-layers



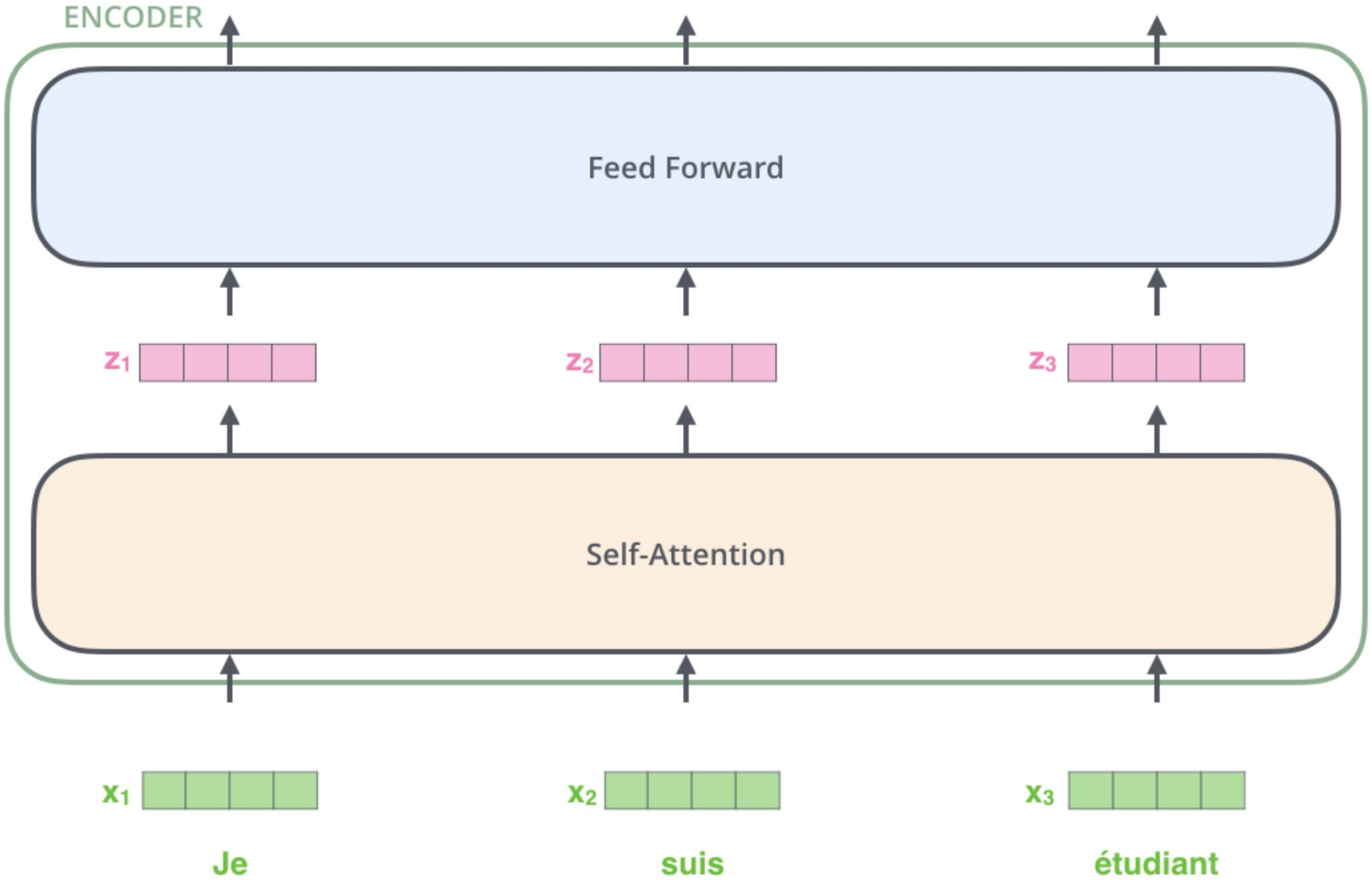


outputs of the self-attention are fed to a feed-forward neural network. The exact same one is independently applied to each position.

helps the encoder look at other words in the input sentence as it encodes a specific word.

Key property of Transformer: word in each position flows through its own path in the encoder.

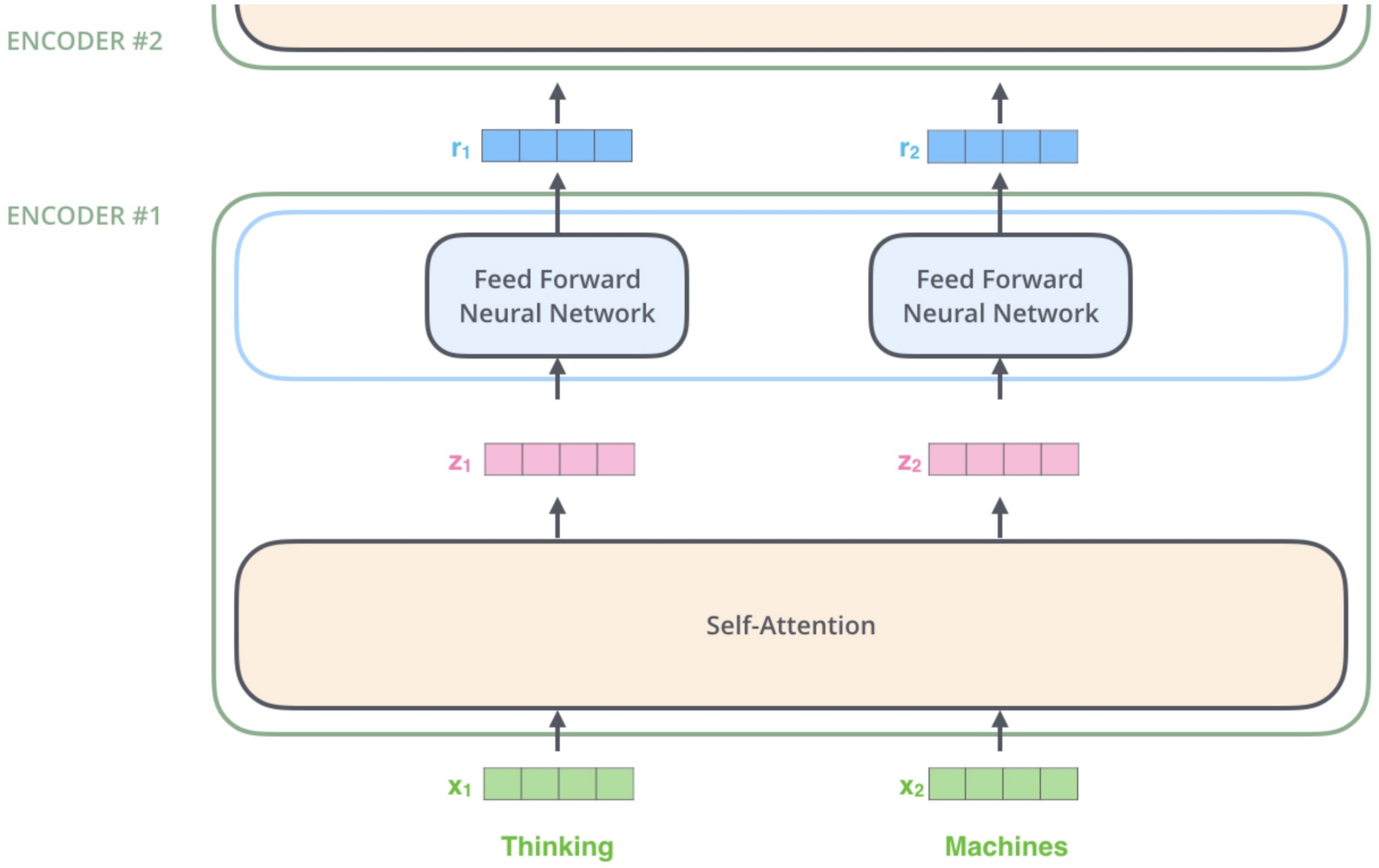
- There are dependencies between these paths in the self-attention layer.
- Feed-forward layer does not have those dependencies => various paths can be executed in parallel !



Word embeddings

Visually clearer on two words

- dependencies in self-attention layer.
- No dependencies in Feed-forward layer



Word embeddings

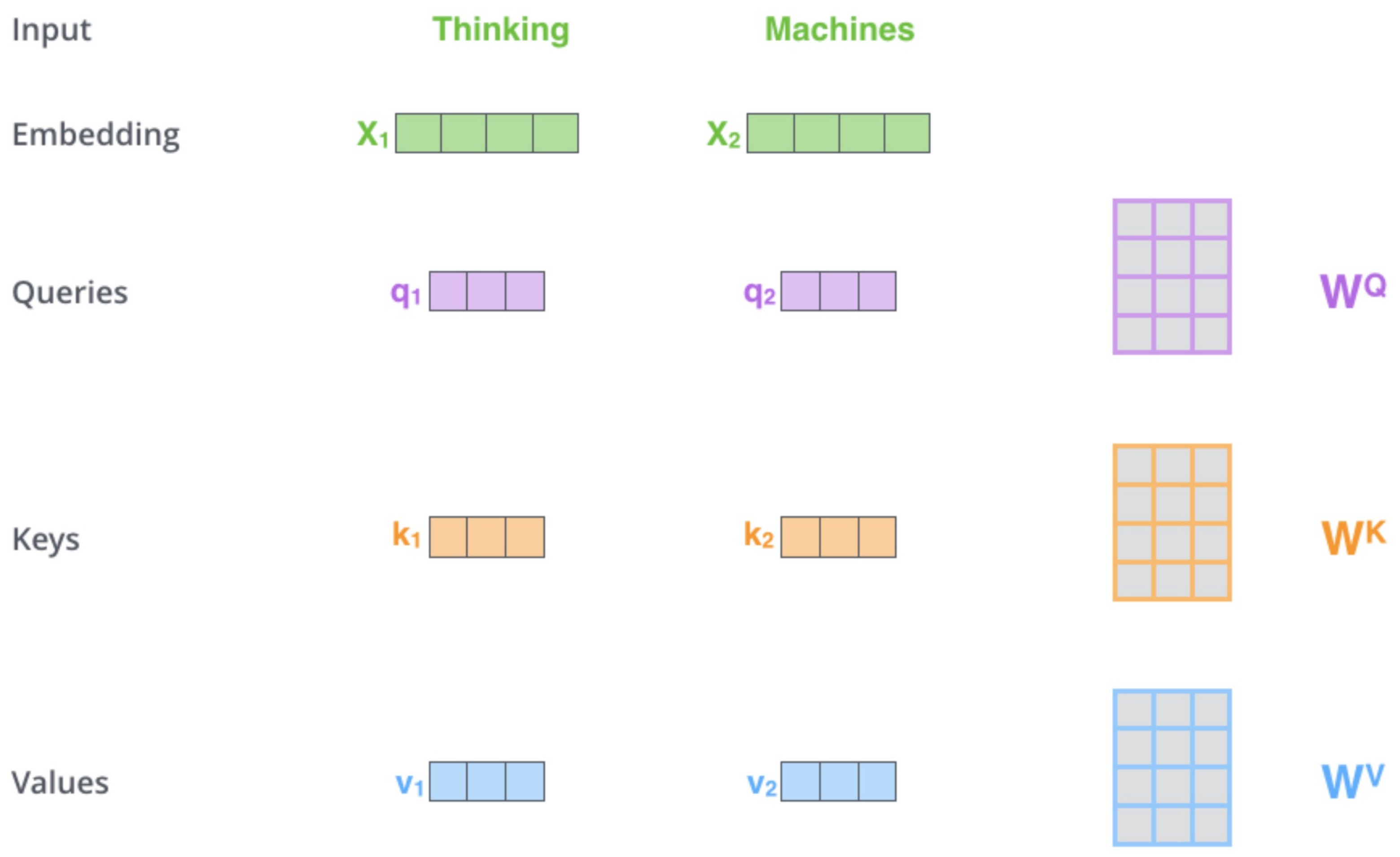
Self-Attention

While processing **each word** it allows to look at other positions in the input sequence for clues to build a better encoding for **this word**.

Step1: create three vectors from each of the encoder's input vectors:

Query, a Key, Value (typically smaller dimension).

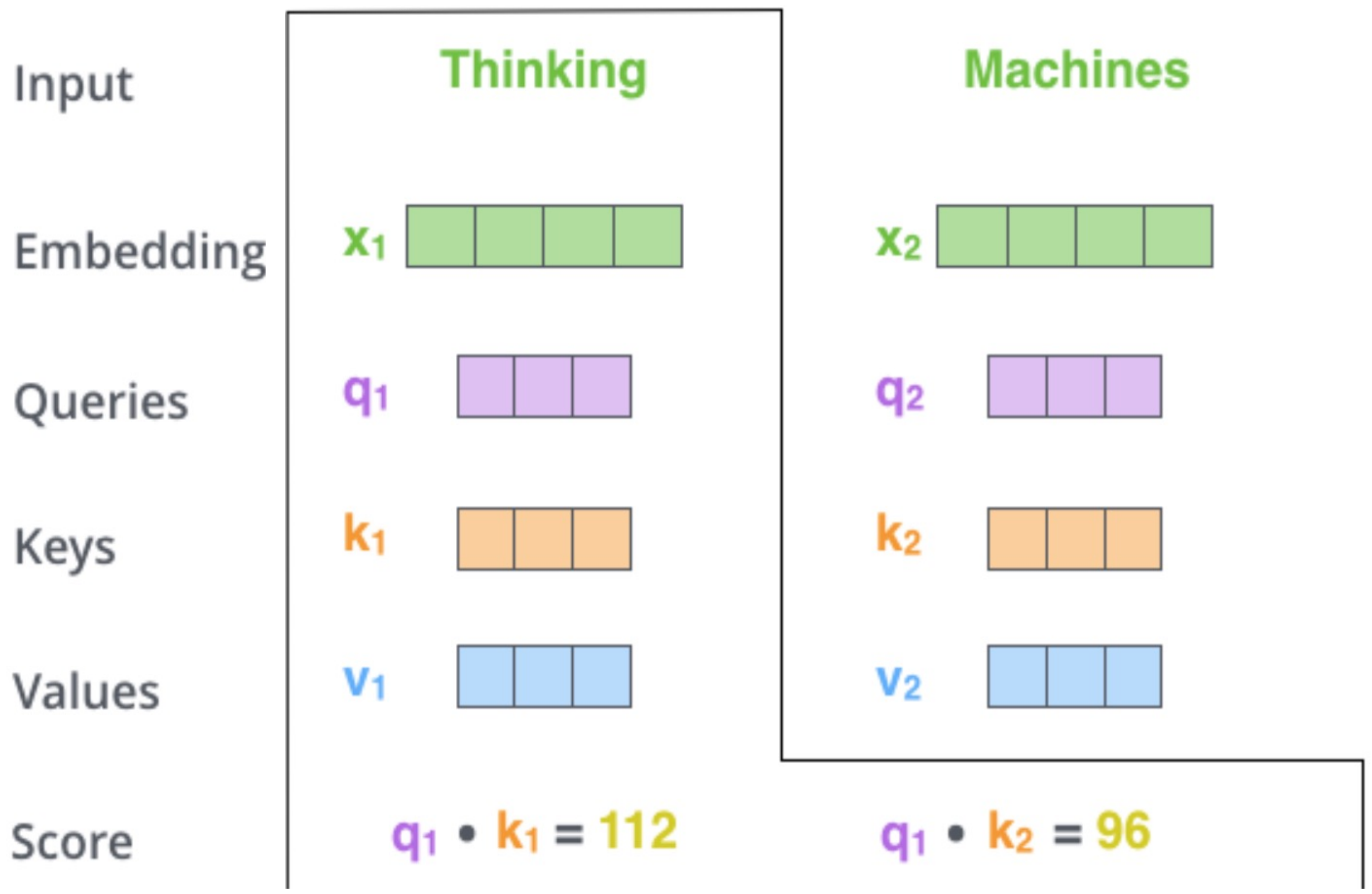
by multiplying the embedding by three matrices that we **trained** during the training process.



Self-Attention

Step 2: calculate a score (like we have seen for regular attention!) how much focus to place on other parts of the input sentence as we encode a word at a certain position.

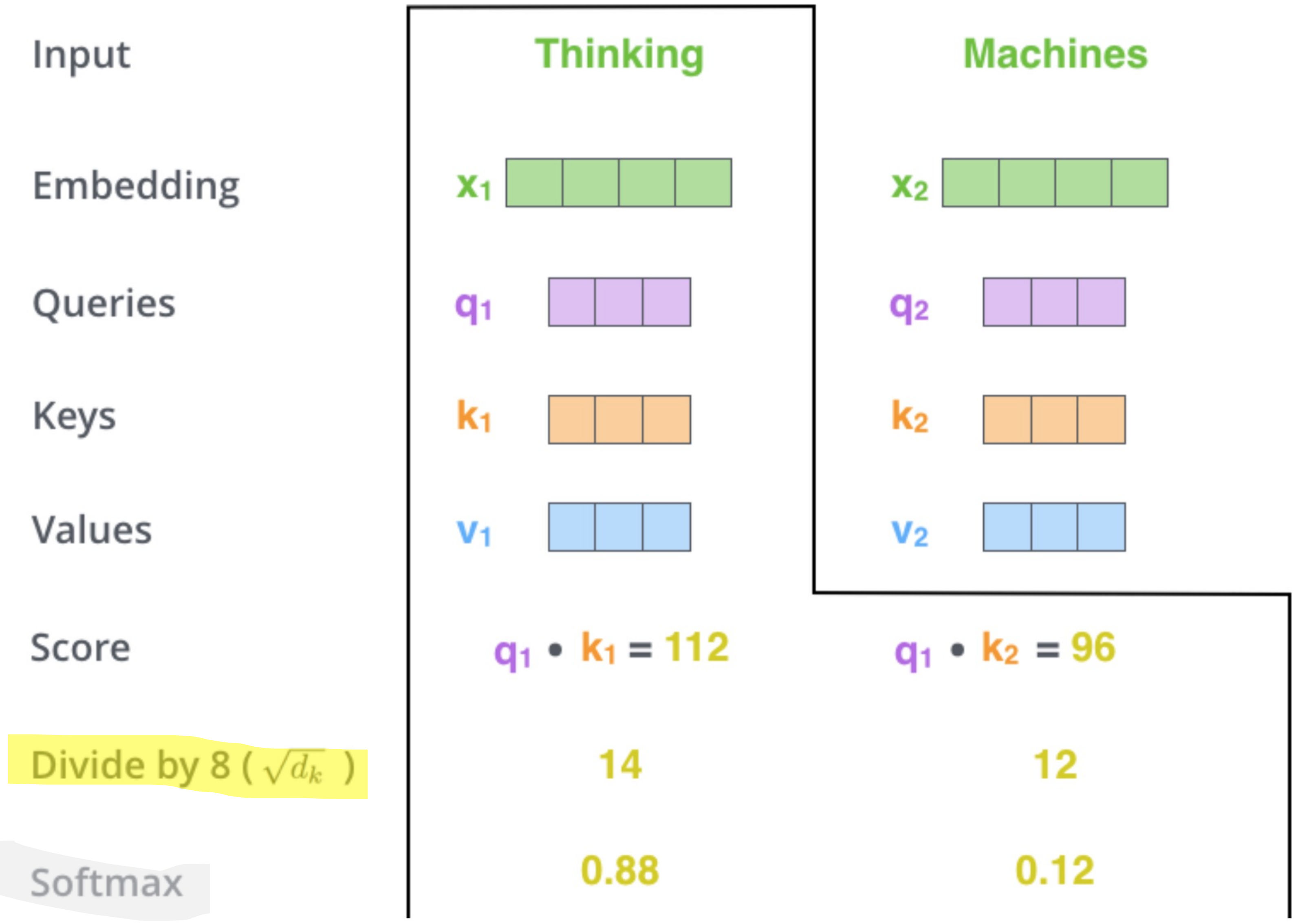
Take dot product of the query vector with the key vector of the respective word we're scoring.



E.g., Processing the self-attention for word "Thinking" in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.

Self Attention

- Step 3 divide scores by the square root of the dimension of the key vectors (more stable gradients).
- Step 4 pass result through a softmax operation. (all positive and add up to 1)



Intuition: softmax score determines how much each word will be expressed at this position.

Self Attention

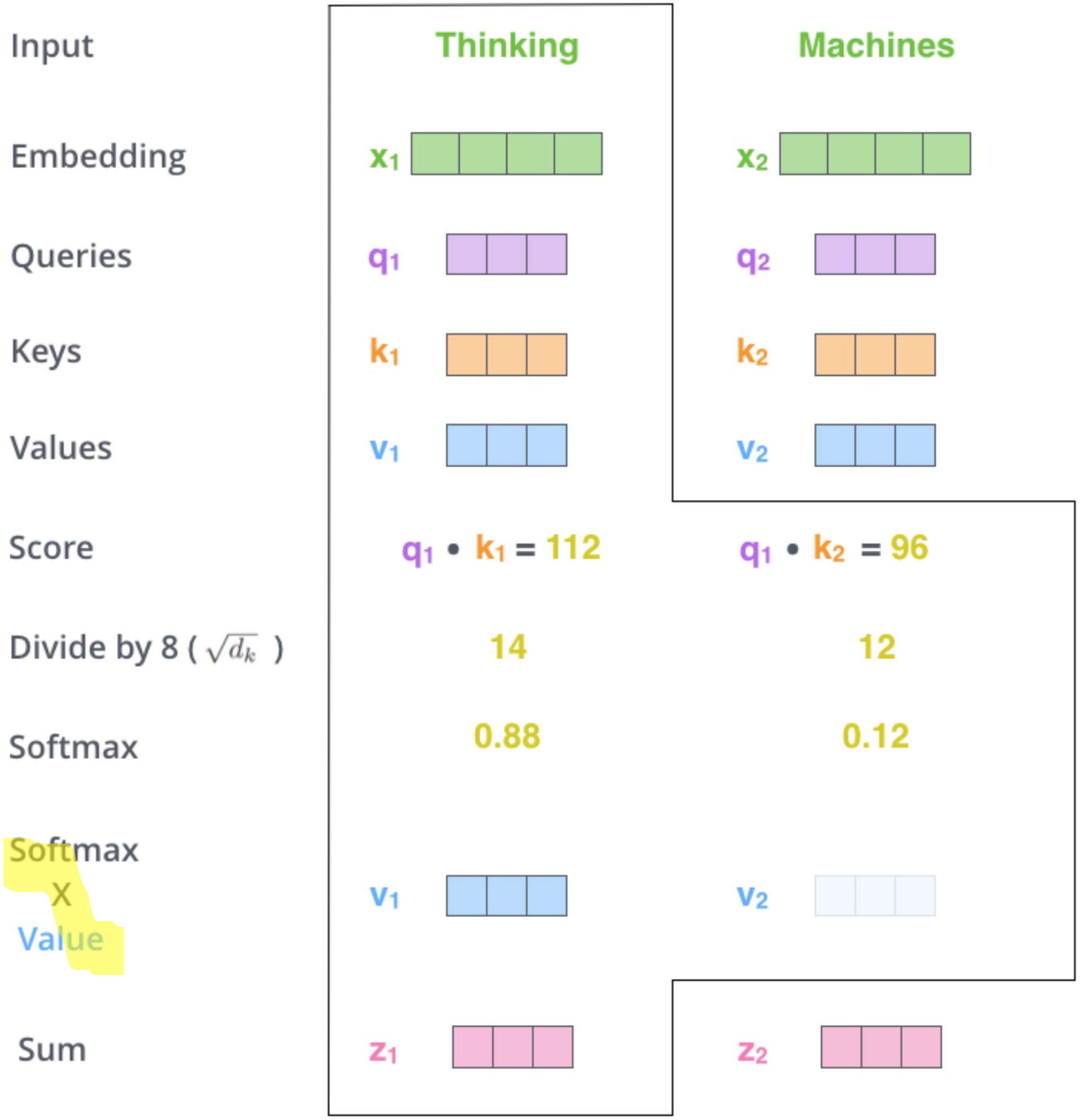
- **Step6** : sum up the weighted value vectors. This produces the output of the self-attention layer at this position

More details:

- What we have seen for a word is done **for all words** (using matrices)
- Need to **encode position** of words
- And improved using a mechanism called "**multi-headed**" attention

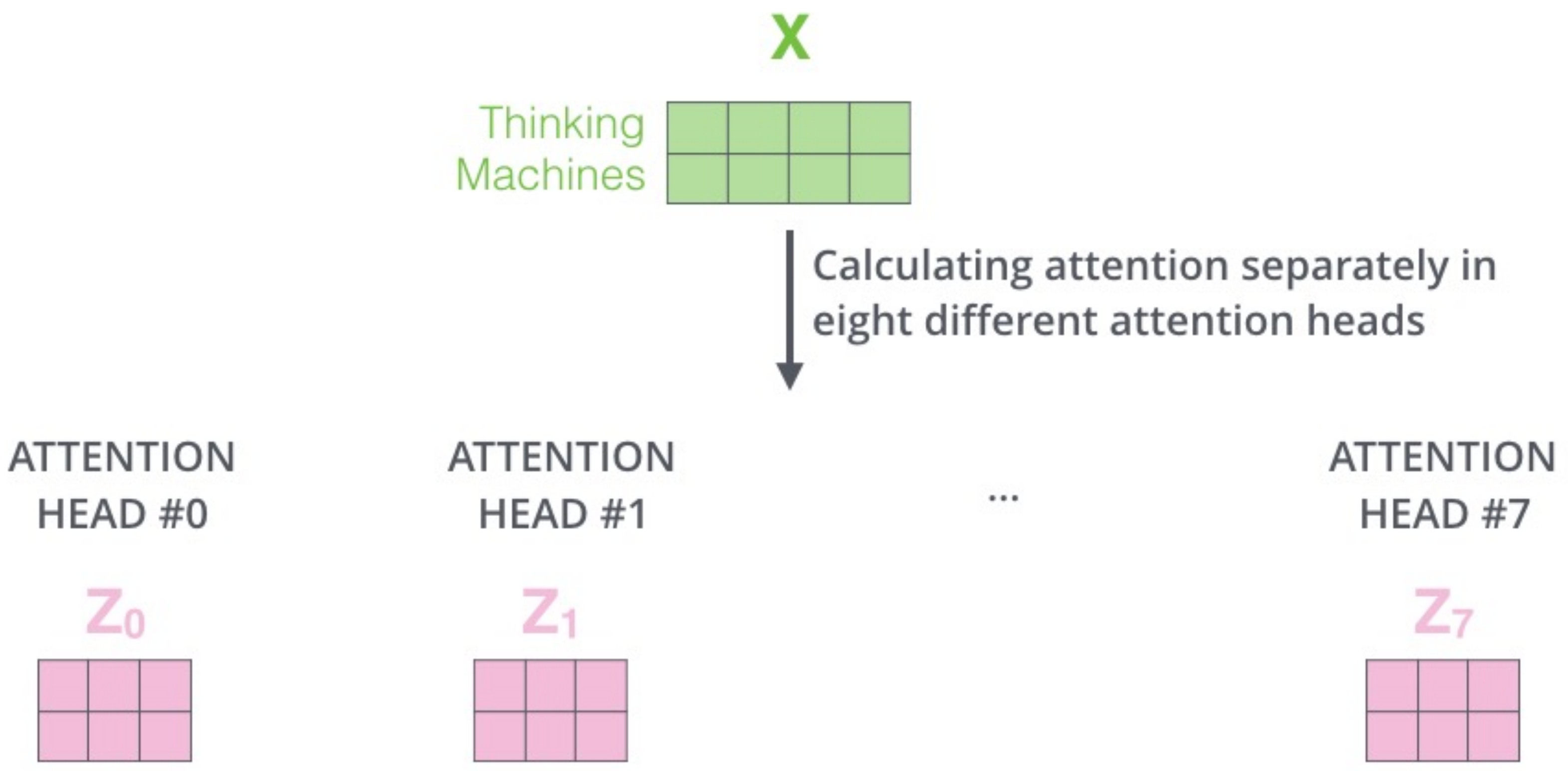
(kind of like multiple filters for CNN)

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



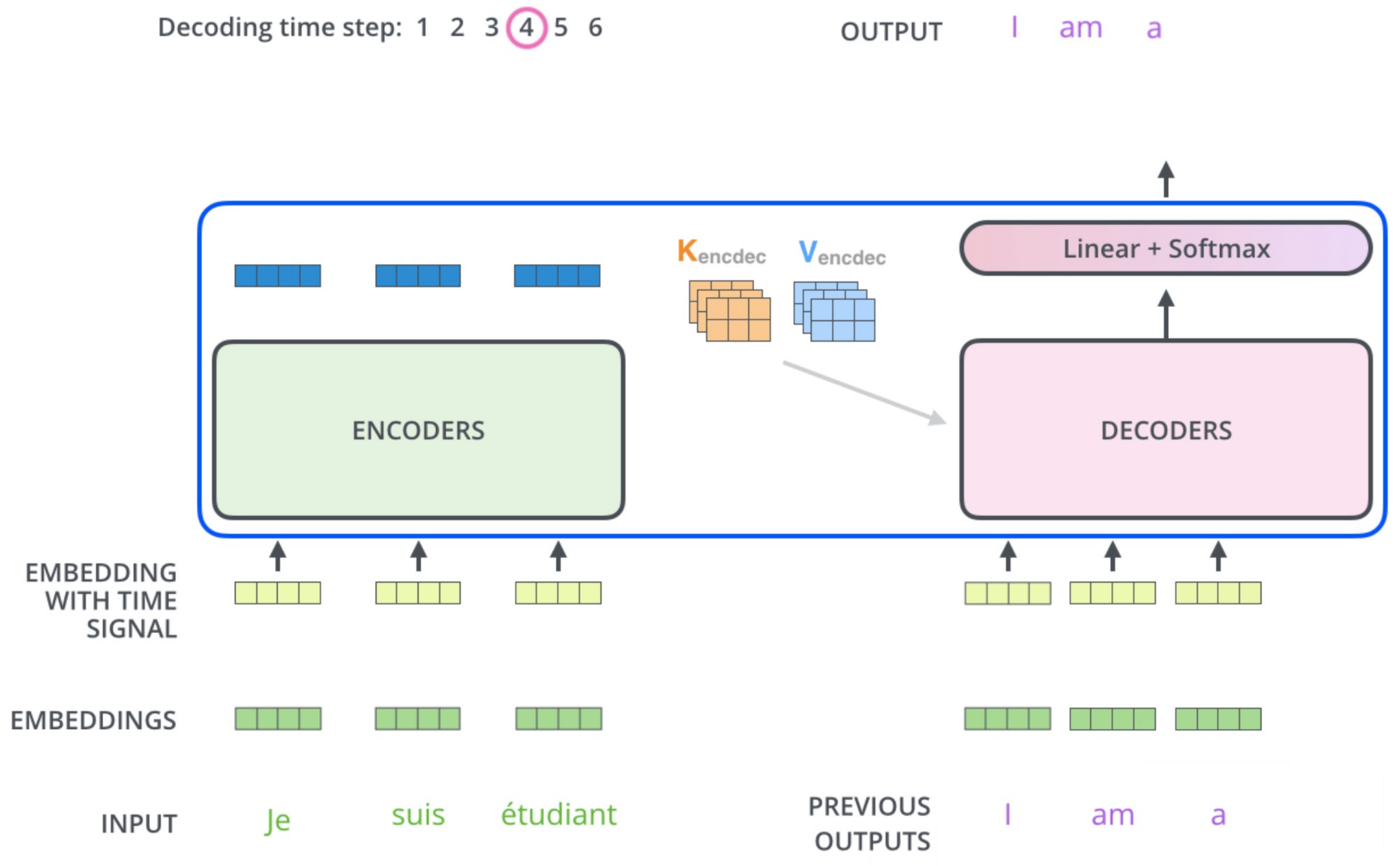
Multiple heads

1. It expands the model's ability to focus on different positions.
2. It gives the attention layer multiple "representation subspaces"



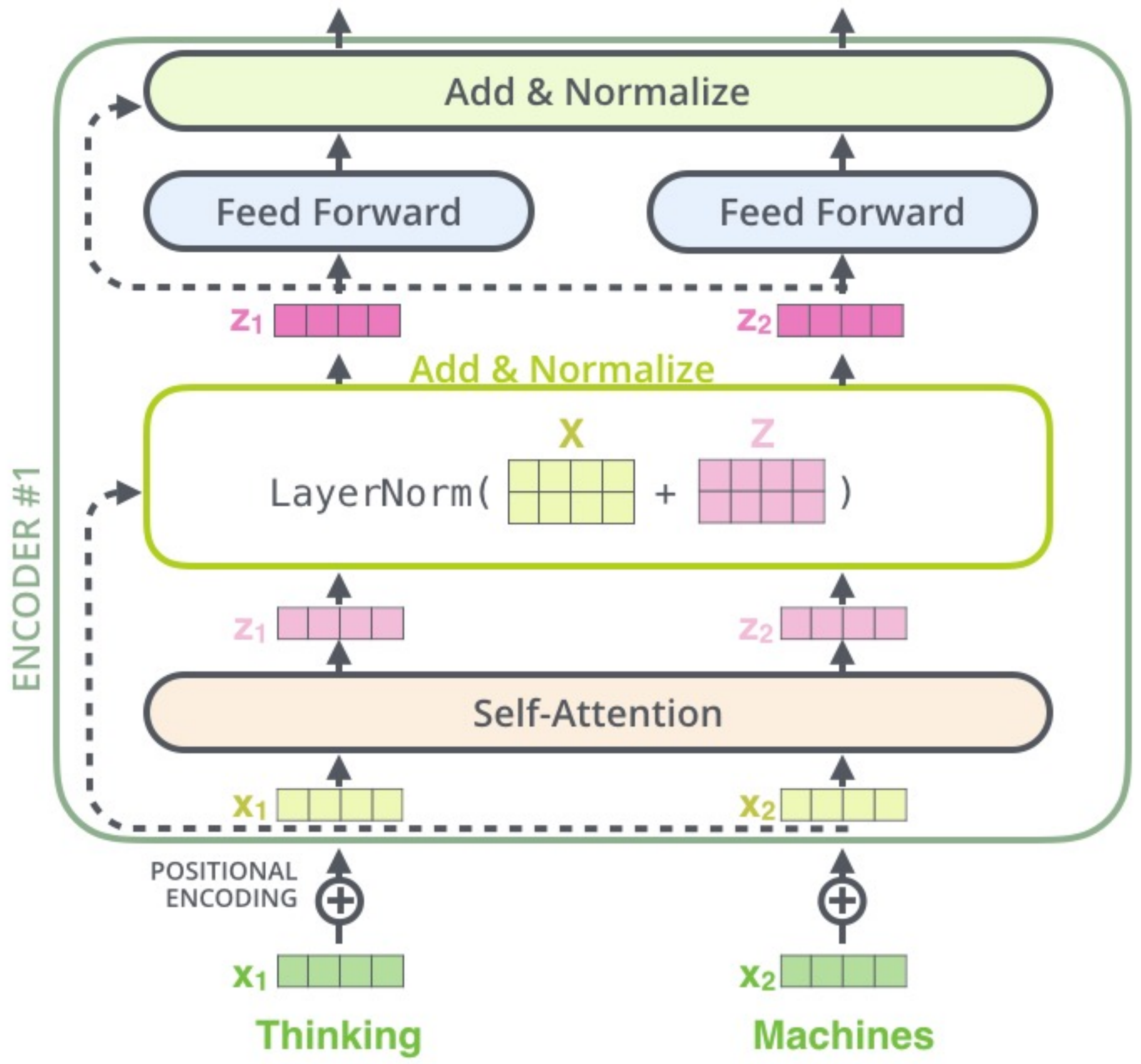
The Decoder Side

- Relies on most of the concepts on the encoder side



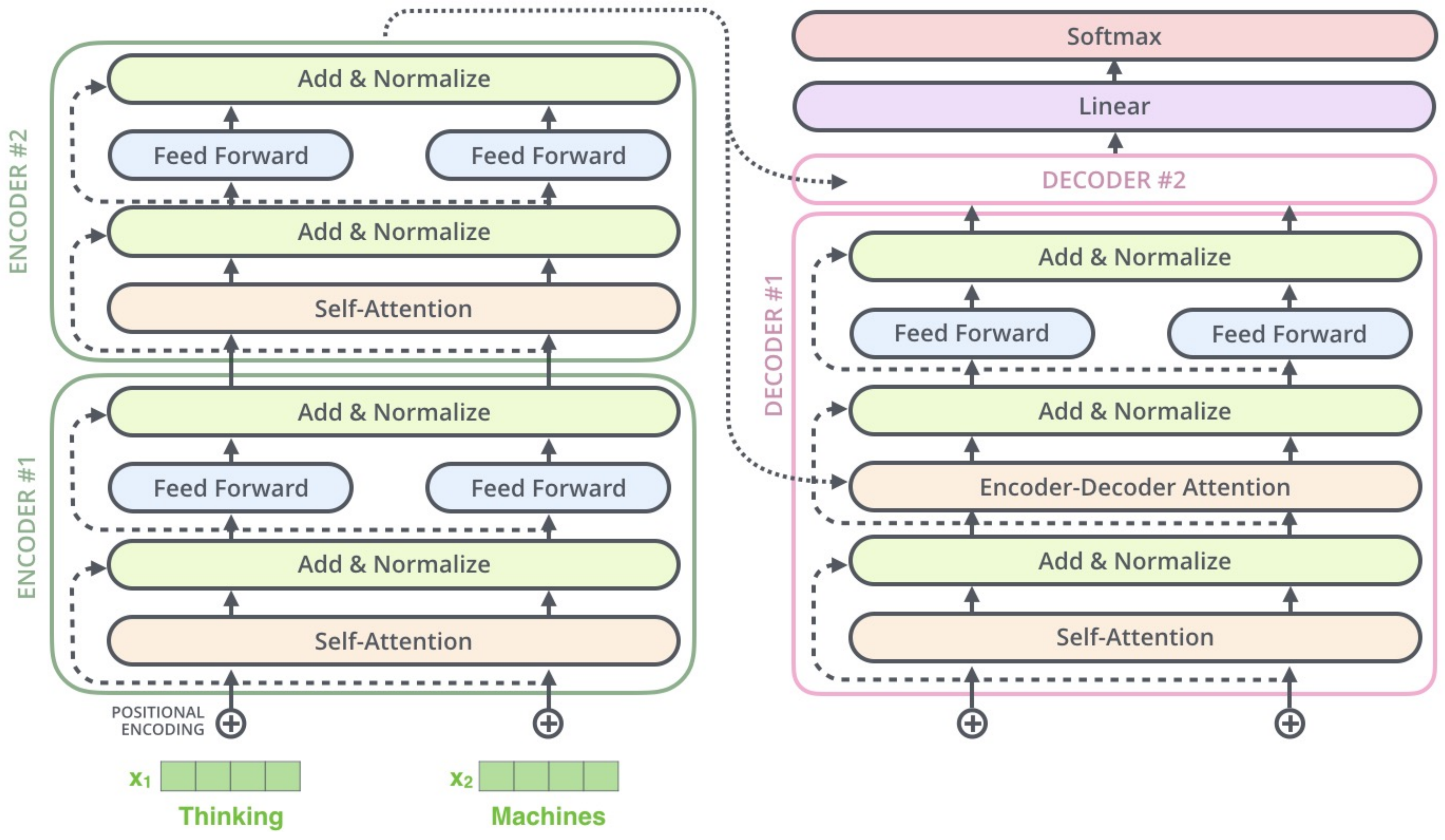
Add and Normalize

In order to regulate the computation, this is a normalization layer so that each feature (column) have the same average and deviation.



The complete transformer

The encoder-decoder attention is just like self attention, except it uses K, V from the top of encoder output, and its own Q



Transformer Results

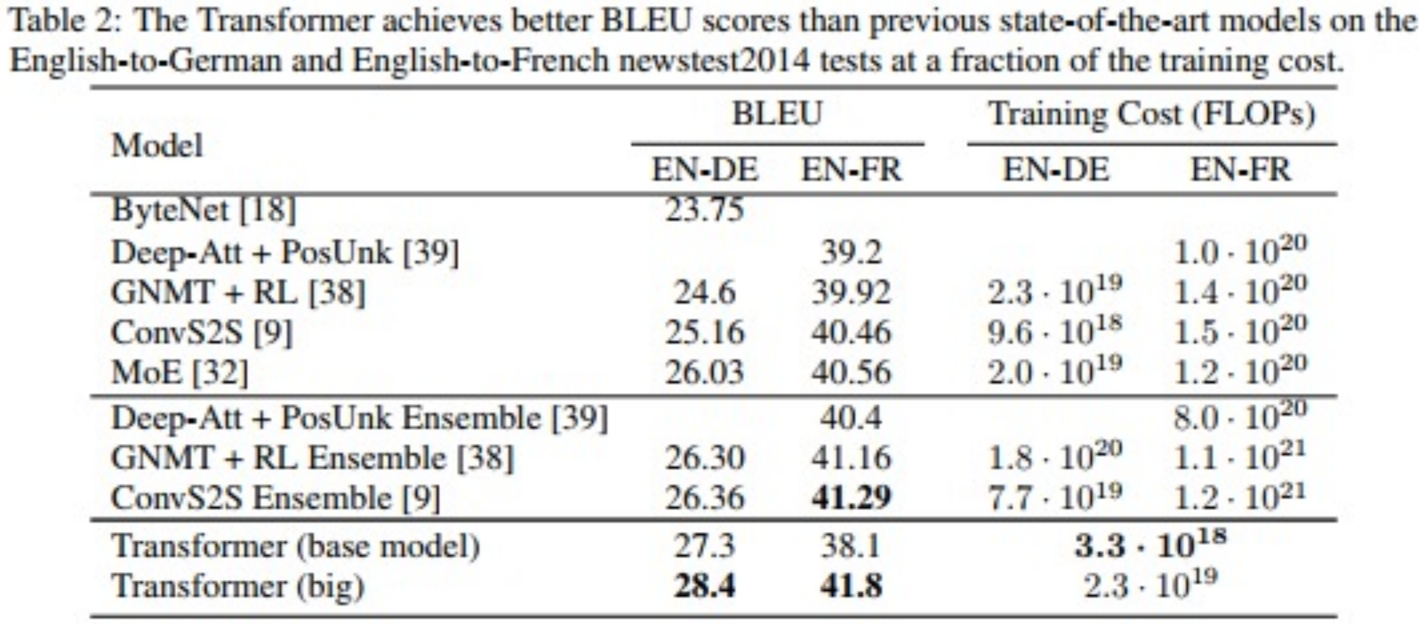


Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Labs

- Generate word embeddings with BOW for spam classification