- **Encoder Decoder**
- Attention
- Transformers

Pragmatics: Example

(i) A: So can you please come over here again right now

(ii) B: Well, I have to go to Edinburgh today sir

(iii) A: Hmm. How about this Thursday?

What information can we infer about the context in which this (short and insignificant) exchange occurred ?

we can make a great number of detailed **(Pragmatic) inferences** about the nature of the context in which it occurred

Pragmatics: Conversational Structure

(i) A: So can you please come over here again right now

(ii) B: Well, I have to go to Edinburgh today sir

(iii) A: Hmm. How about this Thursday?

Not the end of a conversation (nor the beginning)

Pragmatic knowledge: Strong expectations about the structure of conversations

- **Pairs e.g., request <-> response**
- **Closing/Opening forms**

Pragmatics: Dialog Acts

(i) A: So can you please come over here again right now?
(ii) B: Well, I have to go to Edinburgh today sir
(iii) A: Hmm. How about this Thursday?

Not a Y/N info seeking question like *"can you run for 1h?"*
It is a request for an action

- *A is **requesting** B to come at time of speaking,*
- *B **implies he can't** (or would rather not)*
- *A **repeats the request** for some other time.*

Pragmatic assumptions relying on:

- **mutual knowledge** (B knows that A knows that…)
- **co-operation** (must be a response… triggers inference)
- **topical coherence** (who should do what on Thur?)

Pragmatics: Specific Act (Request)

(i) A: So can you please come over here again right now
(ii) B: Well, I have to go to Edinburgh today sir
(iii) A: Hmm. How about this Thursday?

- *A wants B to come over*
- *A believes it is possible for B to come over*
- *A believes B is not already there*
- *A believes he is not in a position to order B to…*

Pragmatic knowledge: speaker beliefs and intentions underlying the act of requesting

Assumption: A behaving rationally and sincerely

Pragmatics: Deixis

(i) A: So can you please come over here again right now

(ii) B: Well, I have to go to Edinburgh today sir

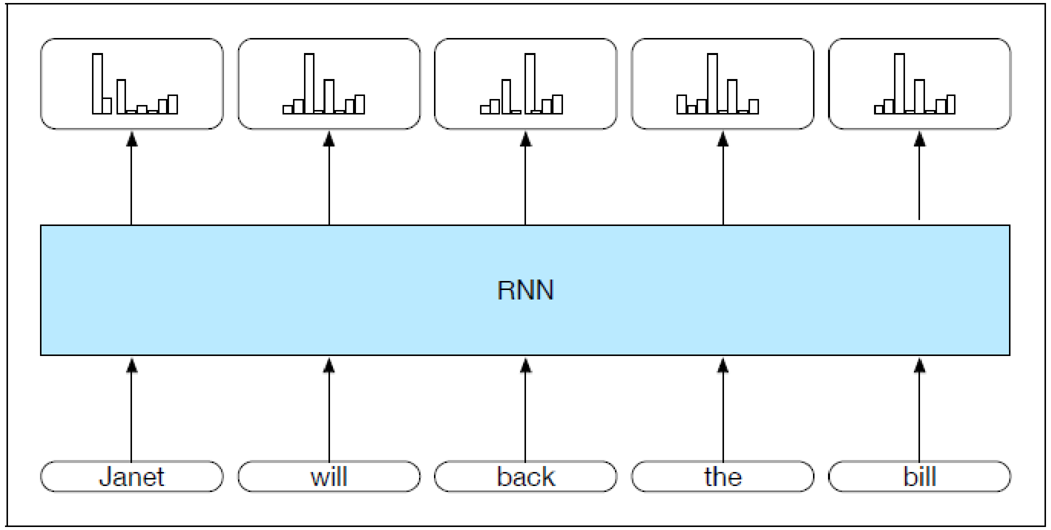
(iii) A: Hmm. How about this Thursday?

- *A assumes B knows where A is*
- *Neither A nor B are in Edinburgh*
- *The day in which the exchange is taking place is not Thur., nor Wed. (or at least, so A believes)*

Pragmatic knowledge: References to space and time wrt space and time of speaking

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.

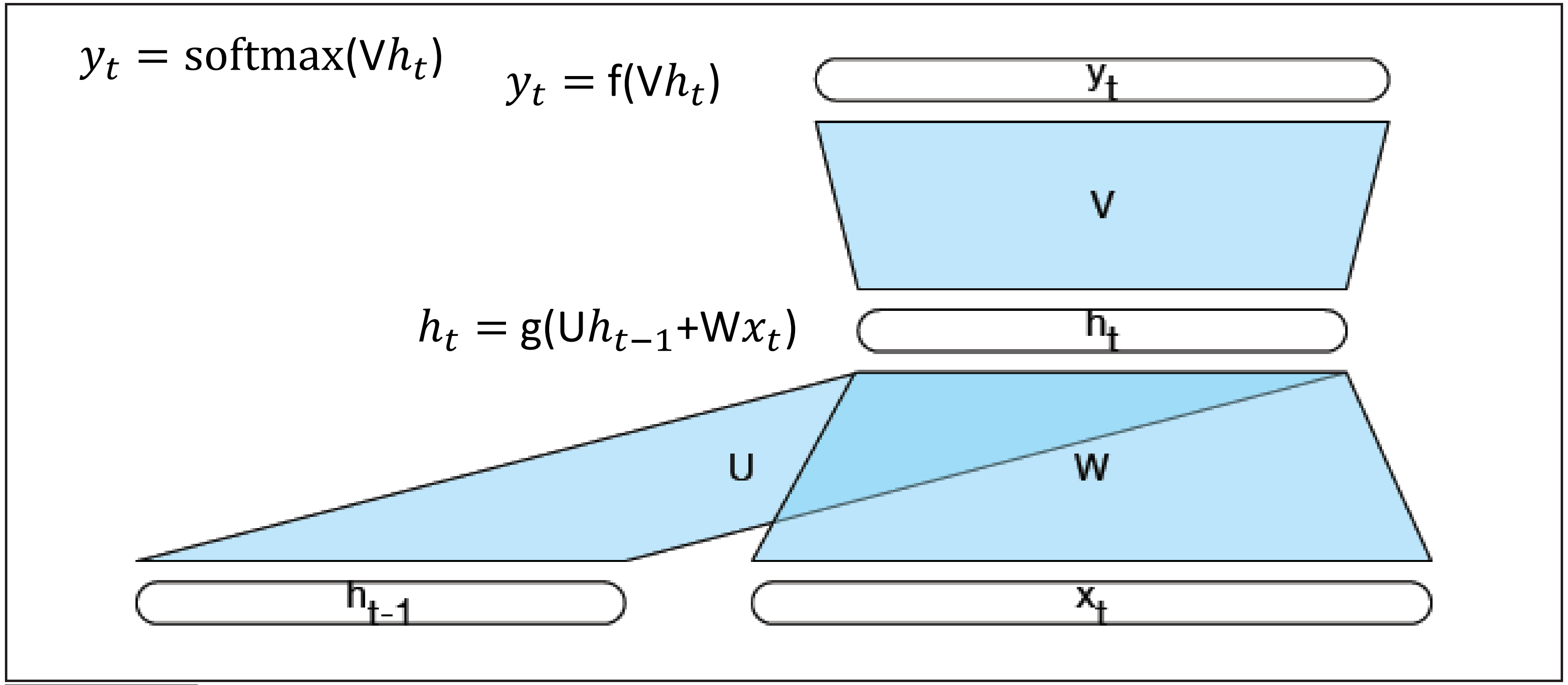
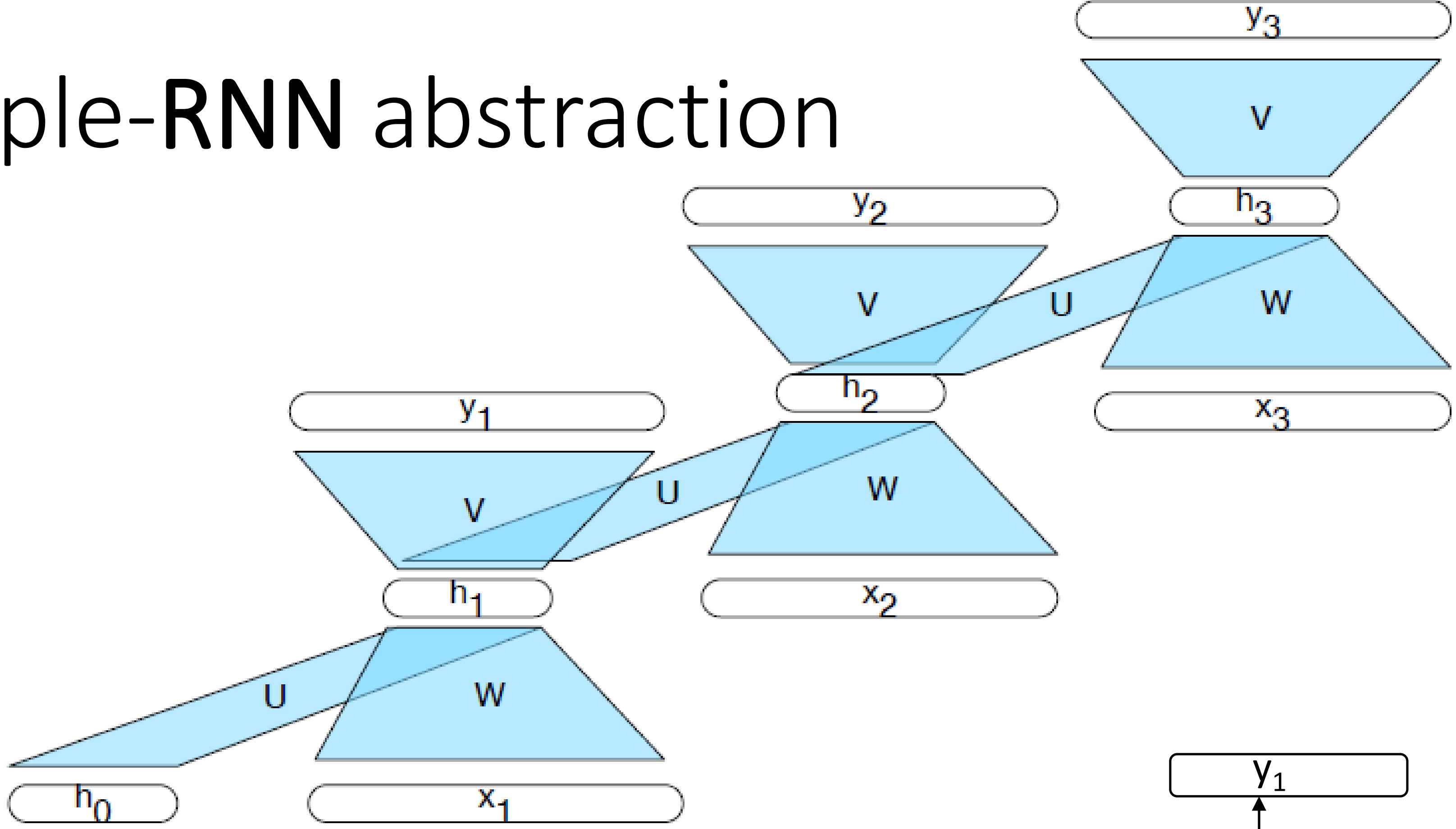
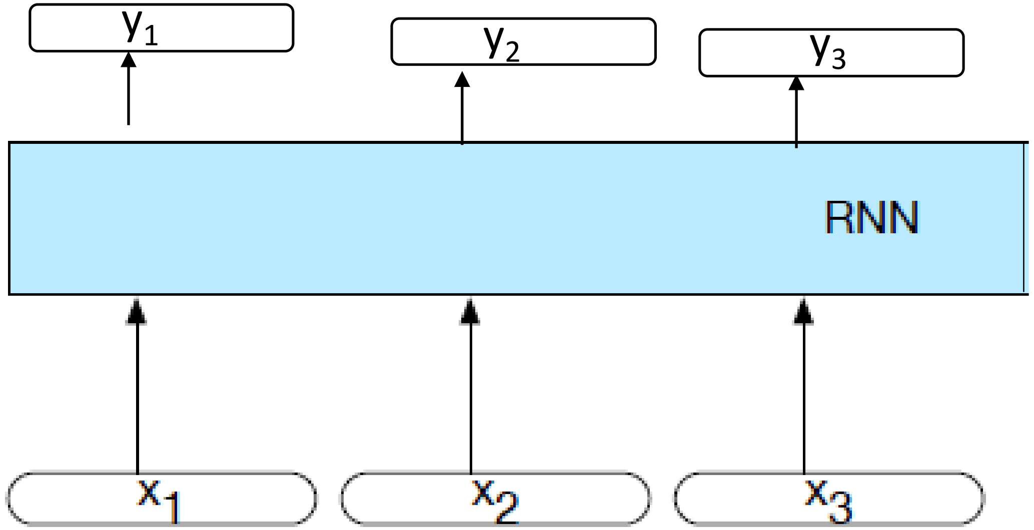


Figure 9.3 Simple recurrent neural network illustrated as a feed-forward network.

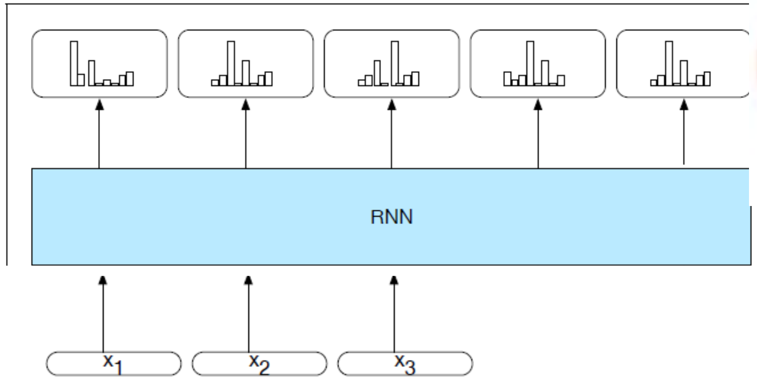
Simple-**RNN** abstraction

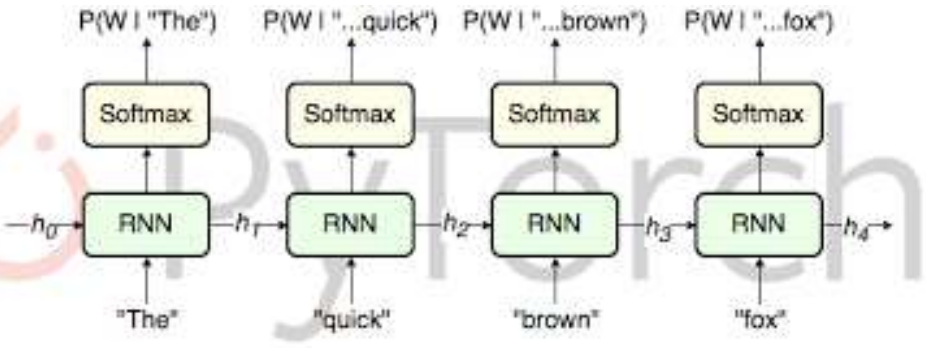




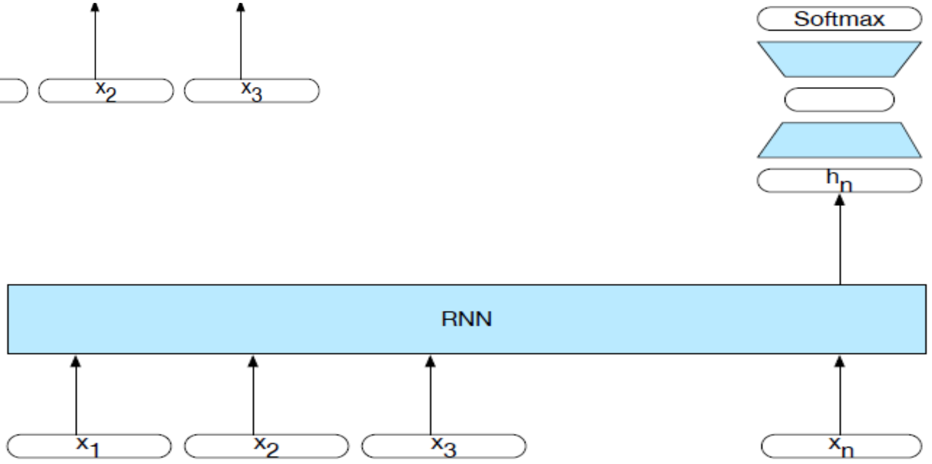
RNN Applications

- Language Modeling

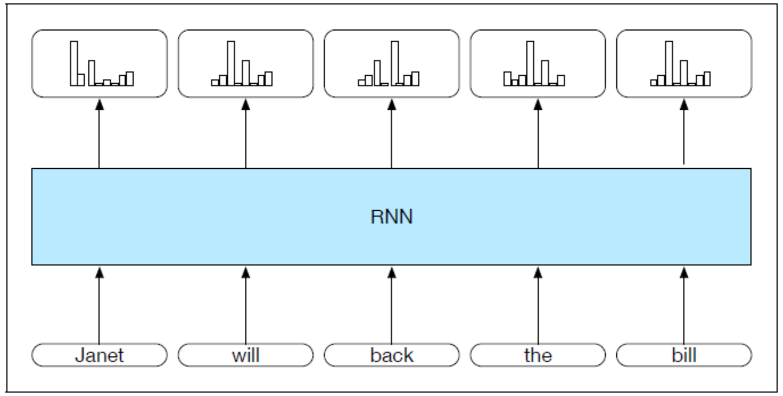




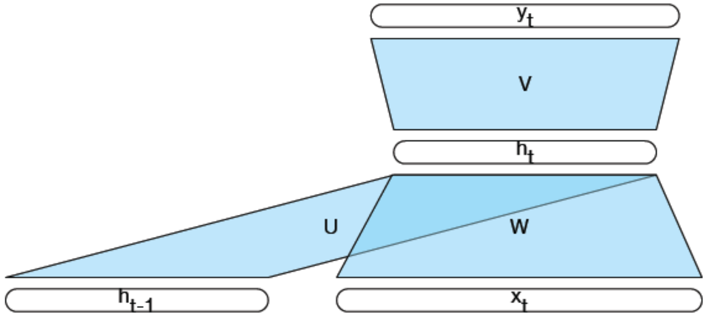
- Sequence Classification (Sentiment, Topic)



- Sequence to Sequence

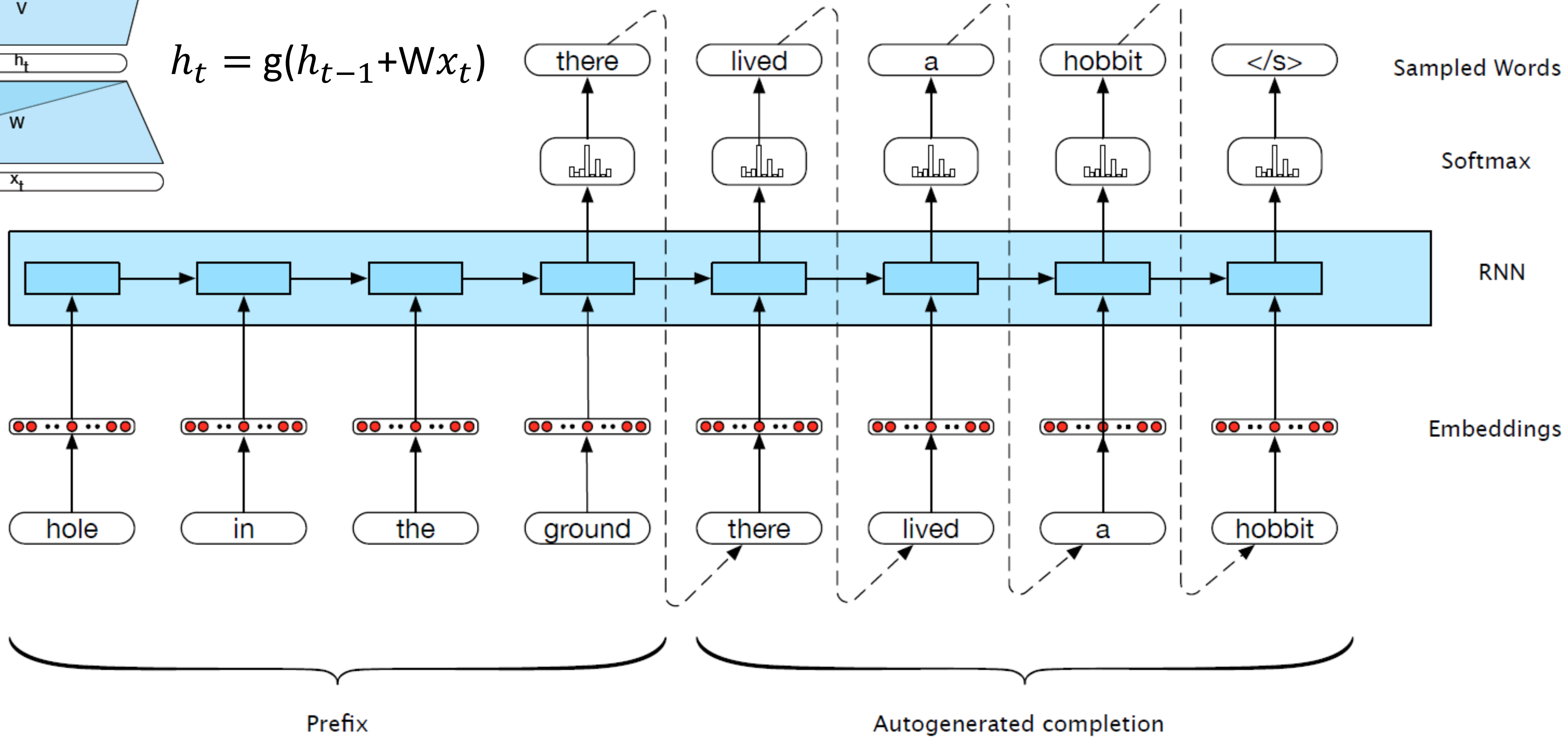


Sentence Completion using an RNN



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

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



- **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*

  ……..

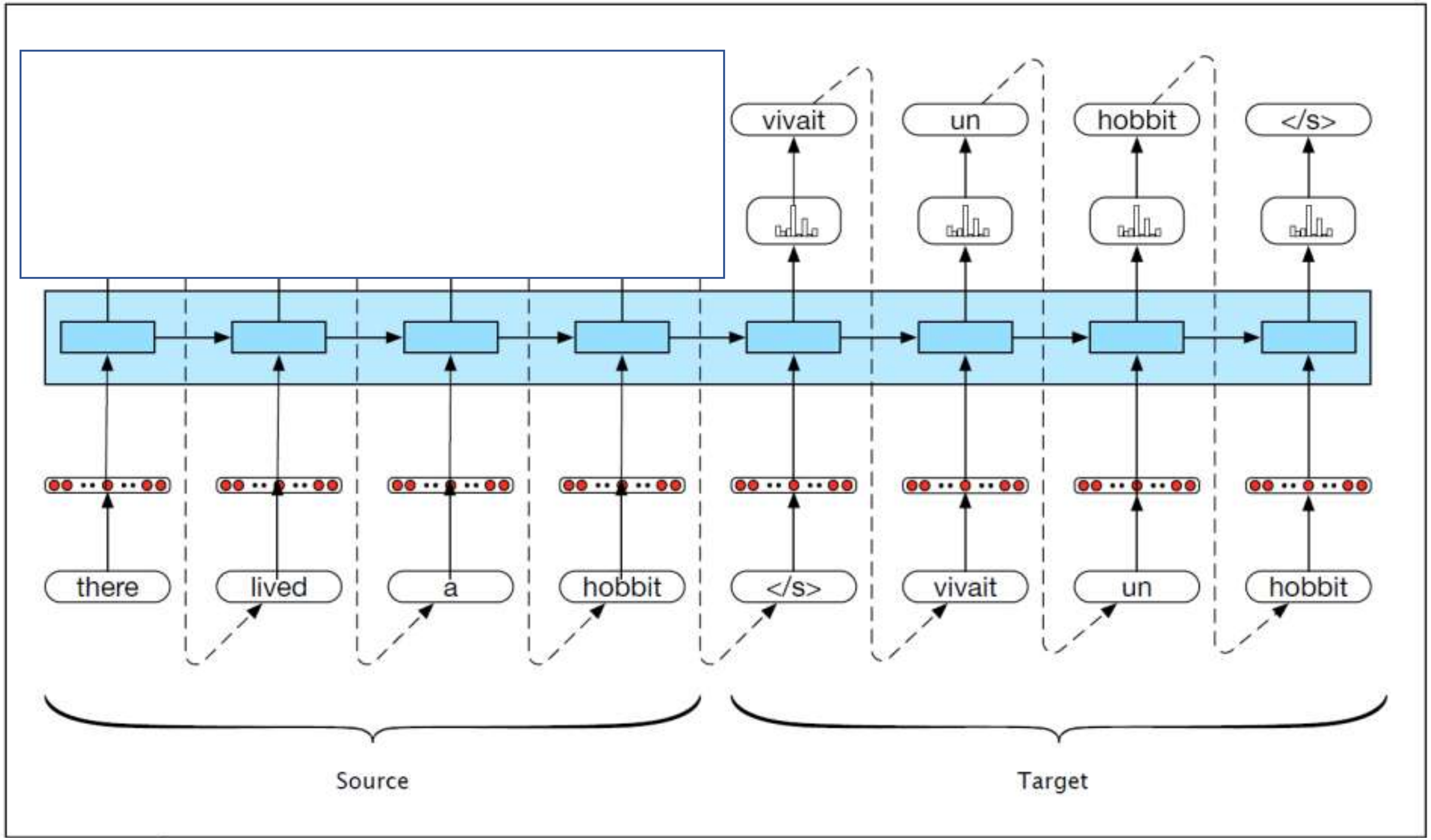
- Build an RNN language model on the concatenation of source and target

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

  ……..

Extending (autoregressive) generation to Machine Translation

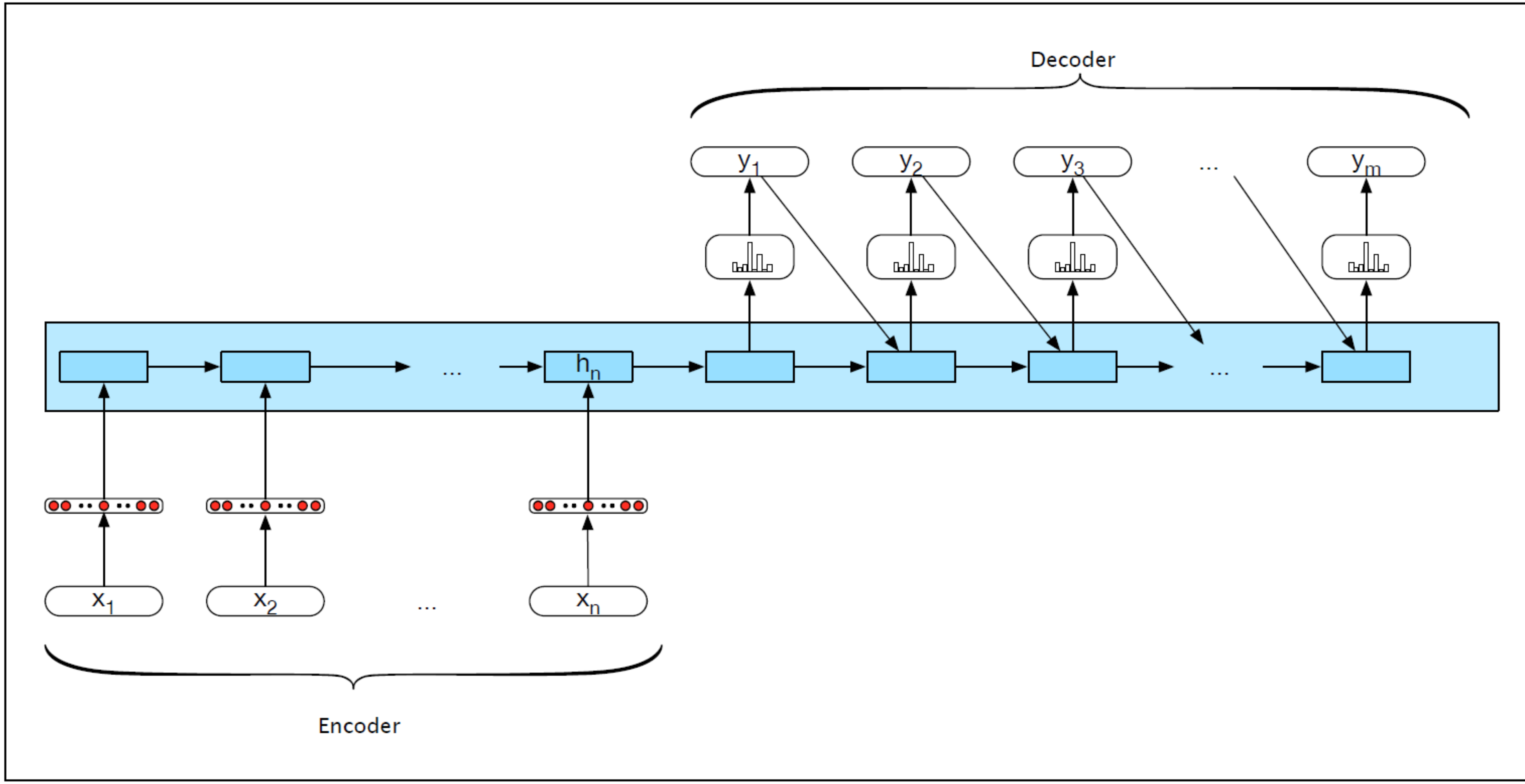
- Translation as Sentence Completion !



(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.

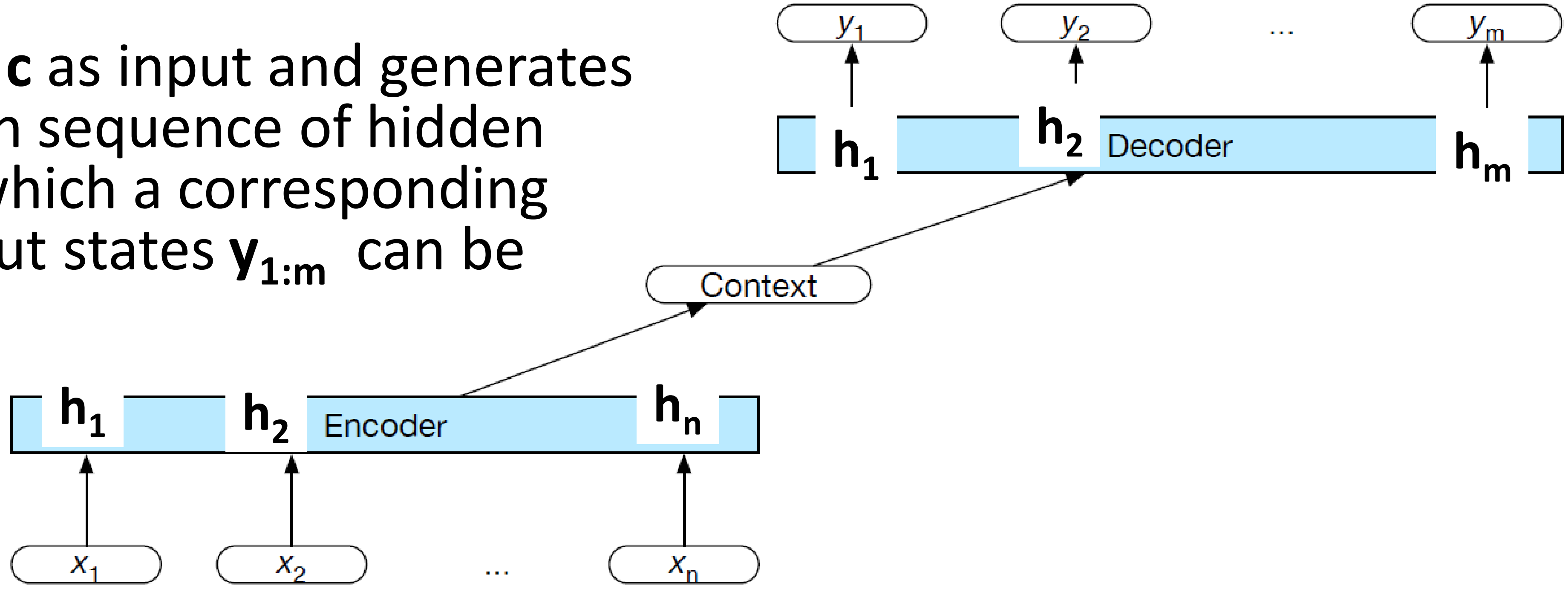


- Encoder generates a contextualized representation of the input (last state).
- Decoder takes that state and autoregressively generates a sequence of outputs

General Encoder Decoder Networks

Abstracting away from these choices

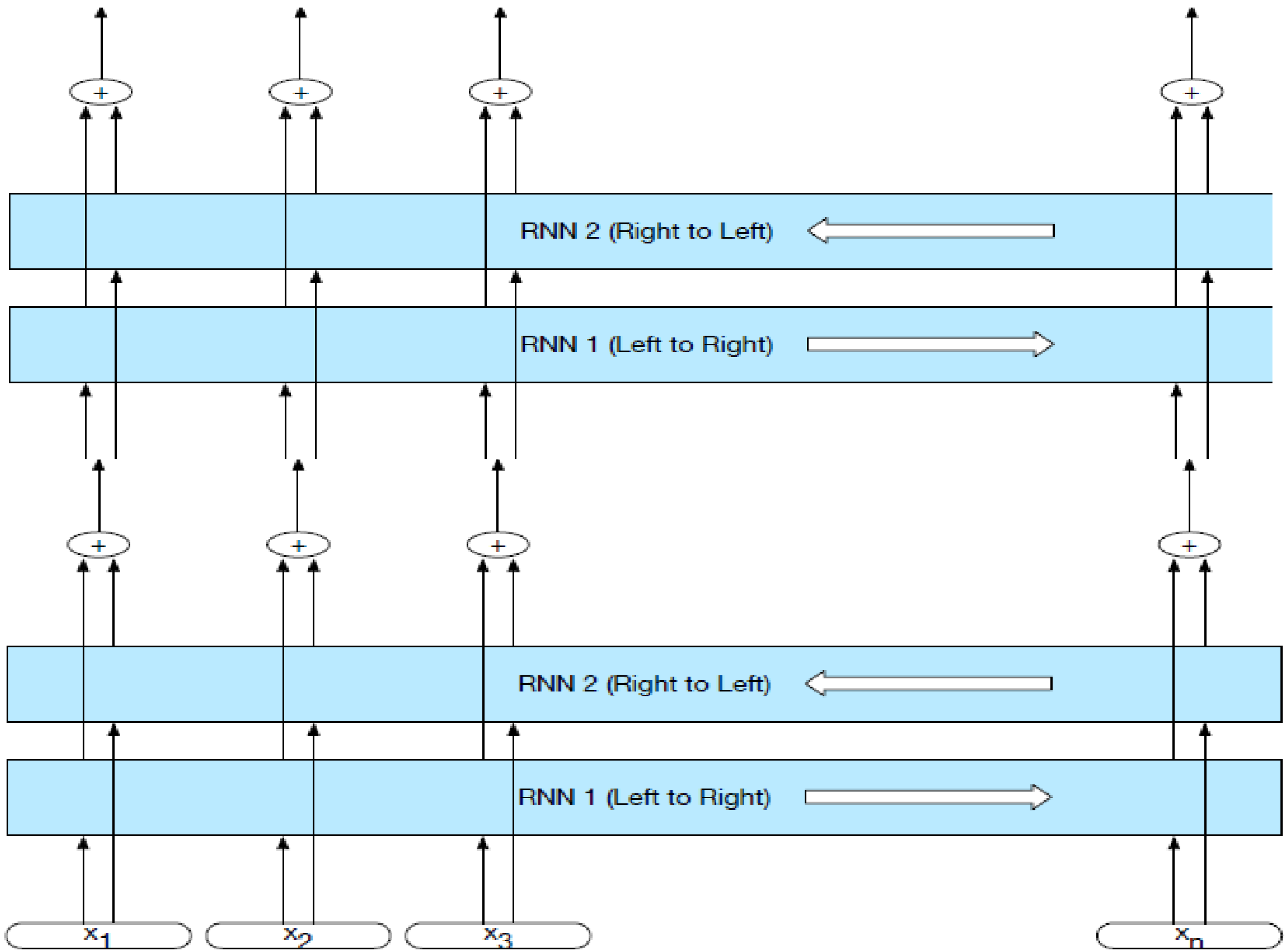
1. **Encoder**: accepts an input sequence, $\mathbf{x}_{1:n}$ and generates a corresponding sequence of contextualized representations, $\mathbf{h}_{1:n}$
2. **Context vector c**: function of $\mathbf{h}_{1:n}$ and conveys the essence of the input to the decoder.
3. **Decoder**: accepts **c** as input 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.



Popular architectural choices: Encoder

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

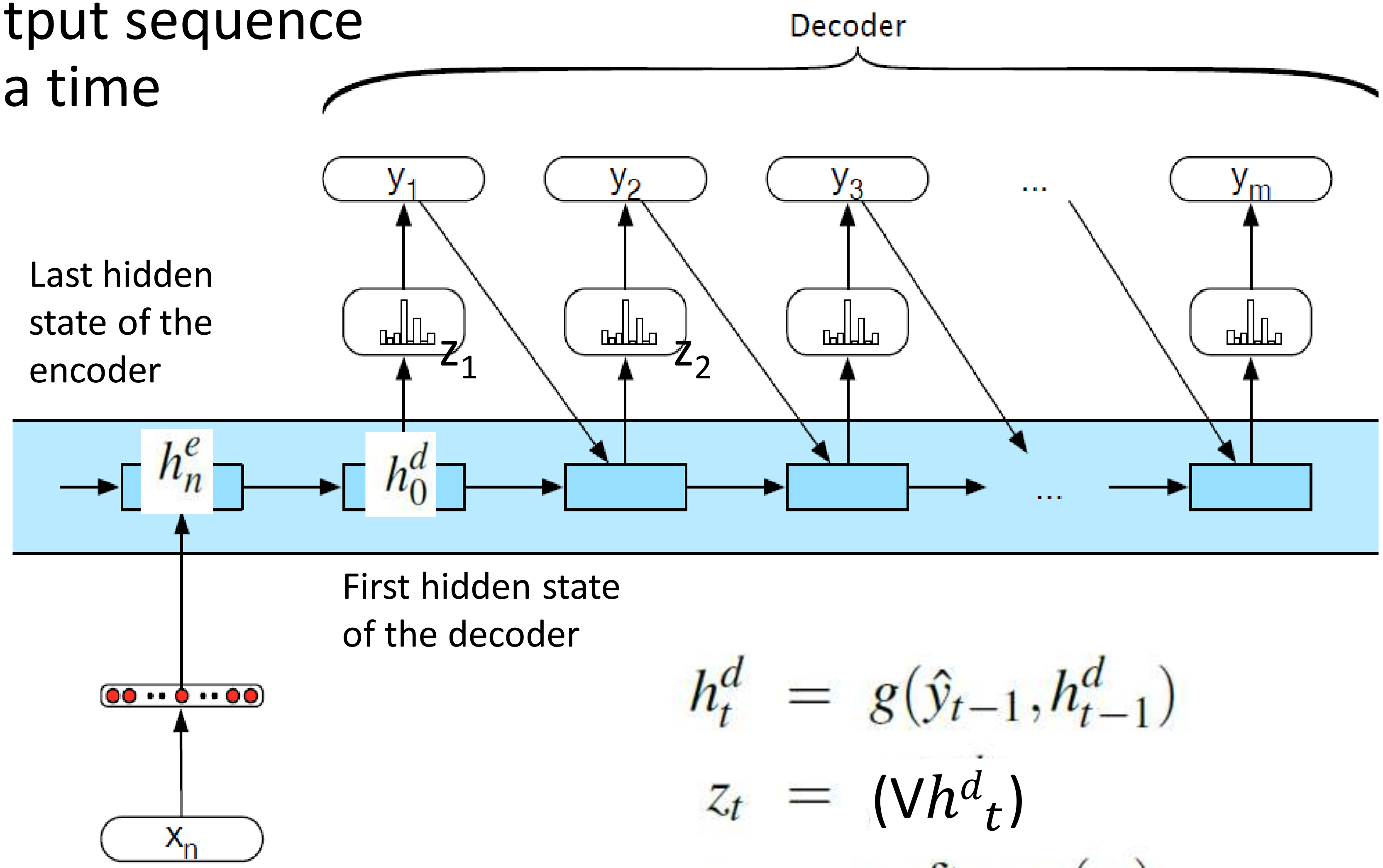
- Contextualized representations for each time step**: hidden states from top layers** from the forward and backward passes



Decoder Basic Design

- produce an output sequence an element at a time

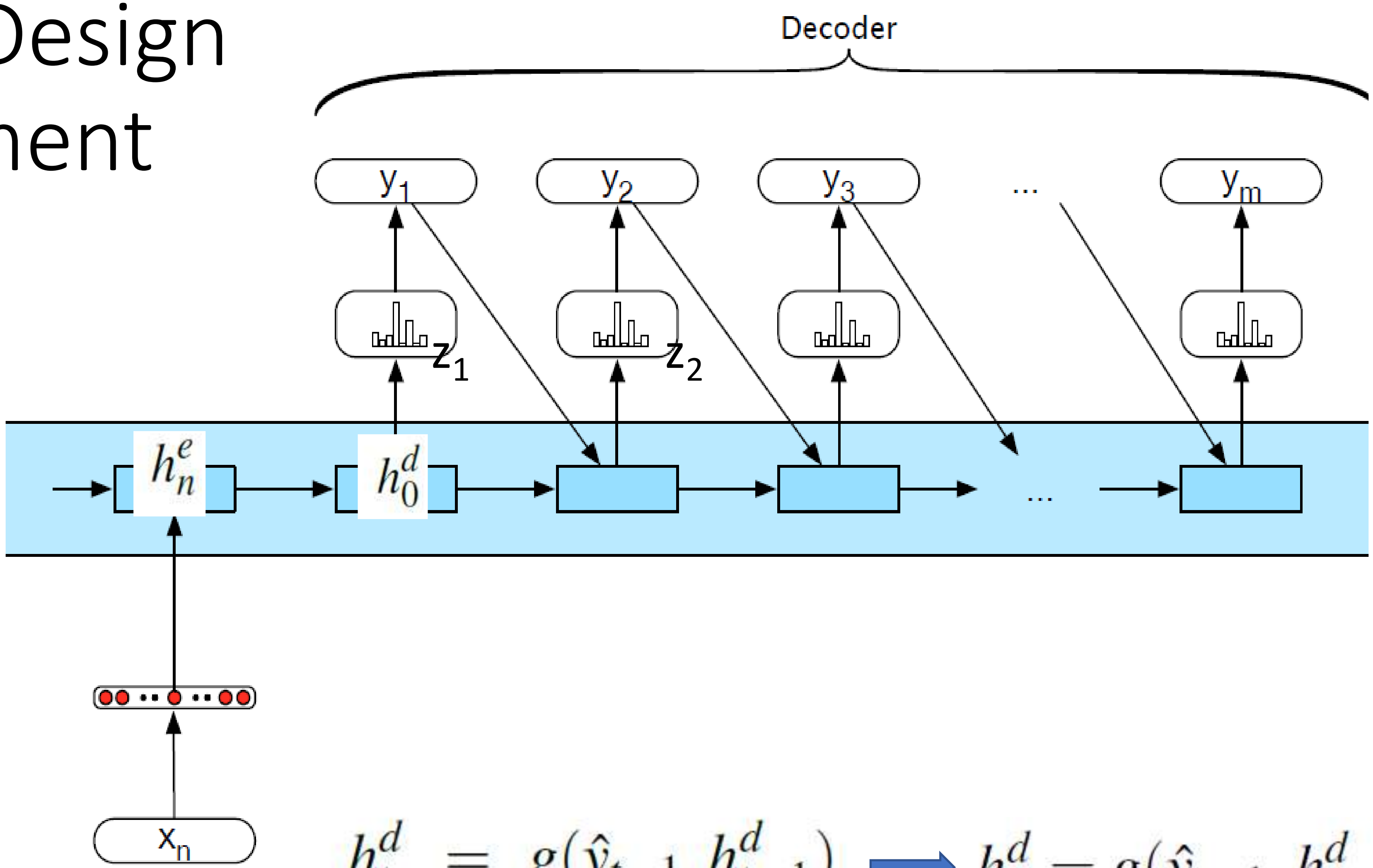
$$c = h_n^e$$
$$h_0^d = c$$



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

Decoder Design Enhancement

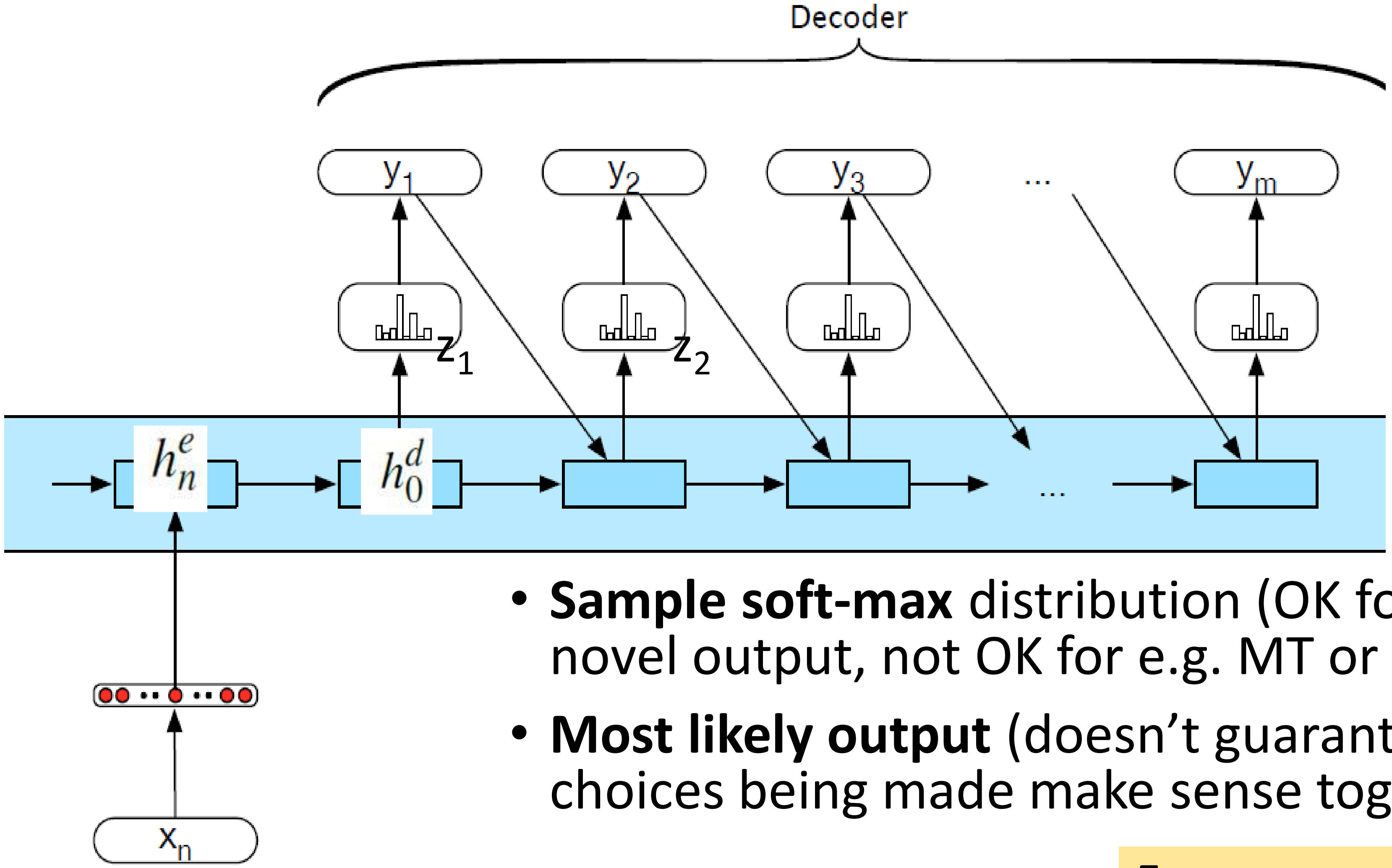
$$c = h_n^e$$
$$h_0^d = c$$



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

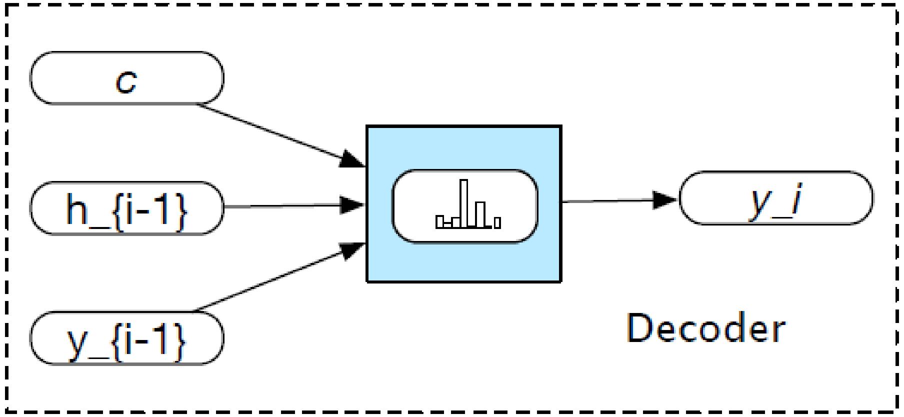
Context available at each step of decoding

Decoder: How output *y* is chosen

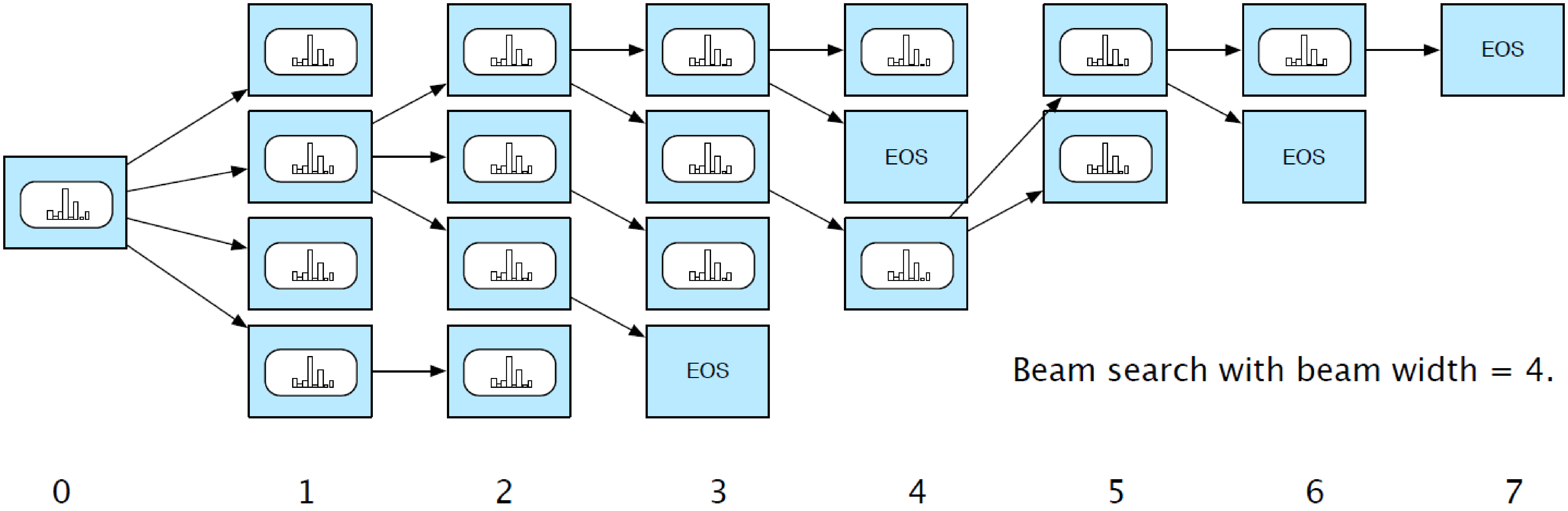


- **Sample soft-max** distribution (OK for generating novel output, not OK for e.g. MT or Summ)
- **Most likely output** (doesn't guarantee individual choices being made make sense together)

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



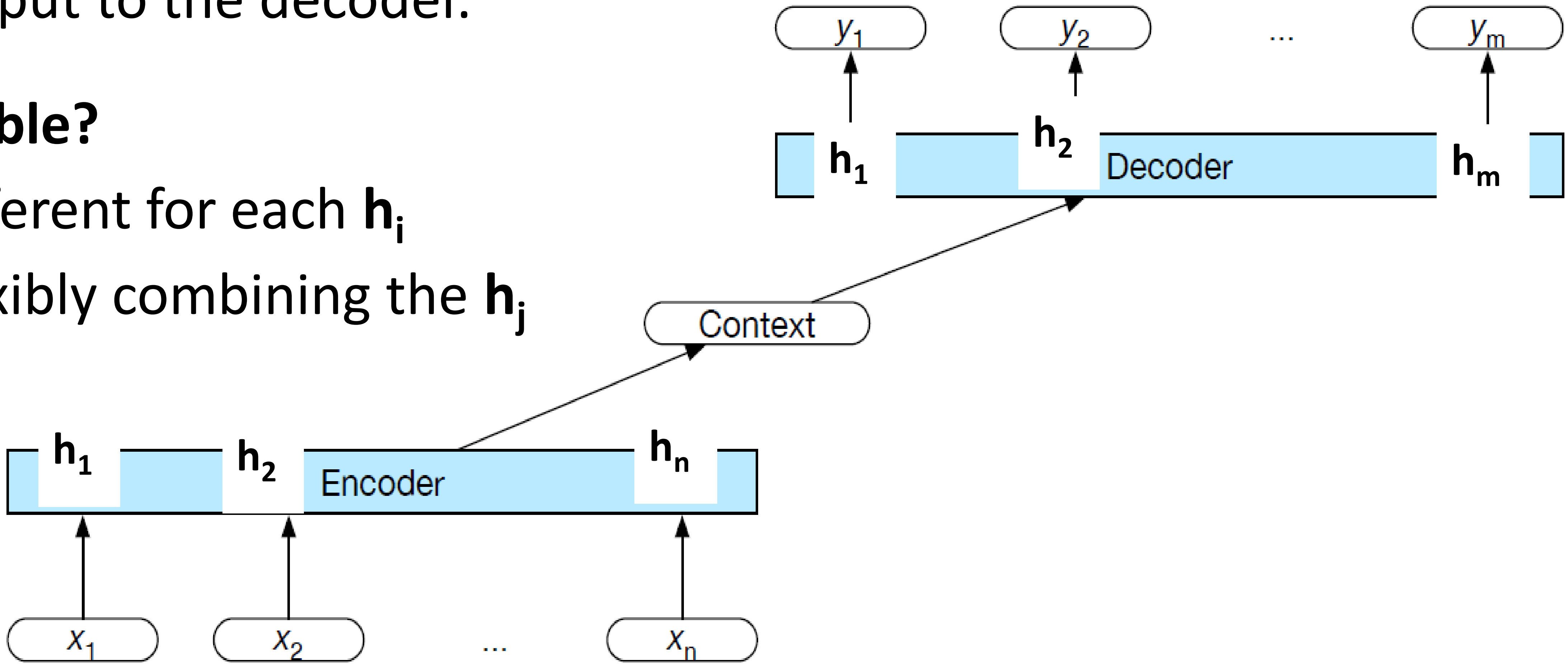
- Encoder Decoder
- **Attention**
- Transformers

Flexible context: Attention

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

Flexible?

- Different for each $\mathbf{h}_i$
- Flexibly combining the $\mathbf{h}_j$



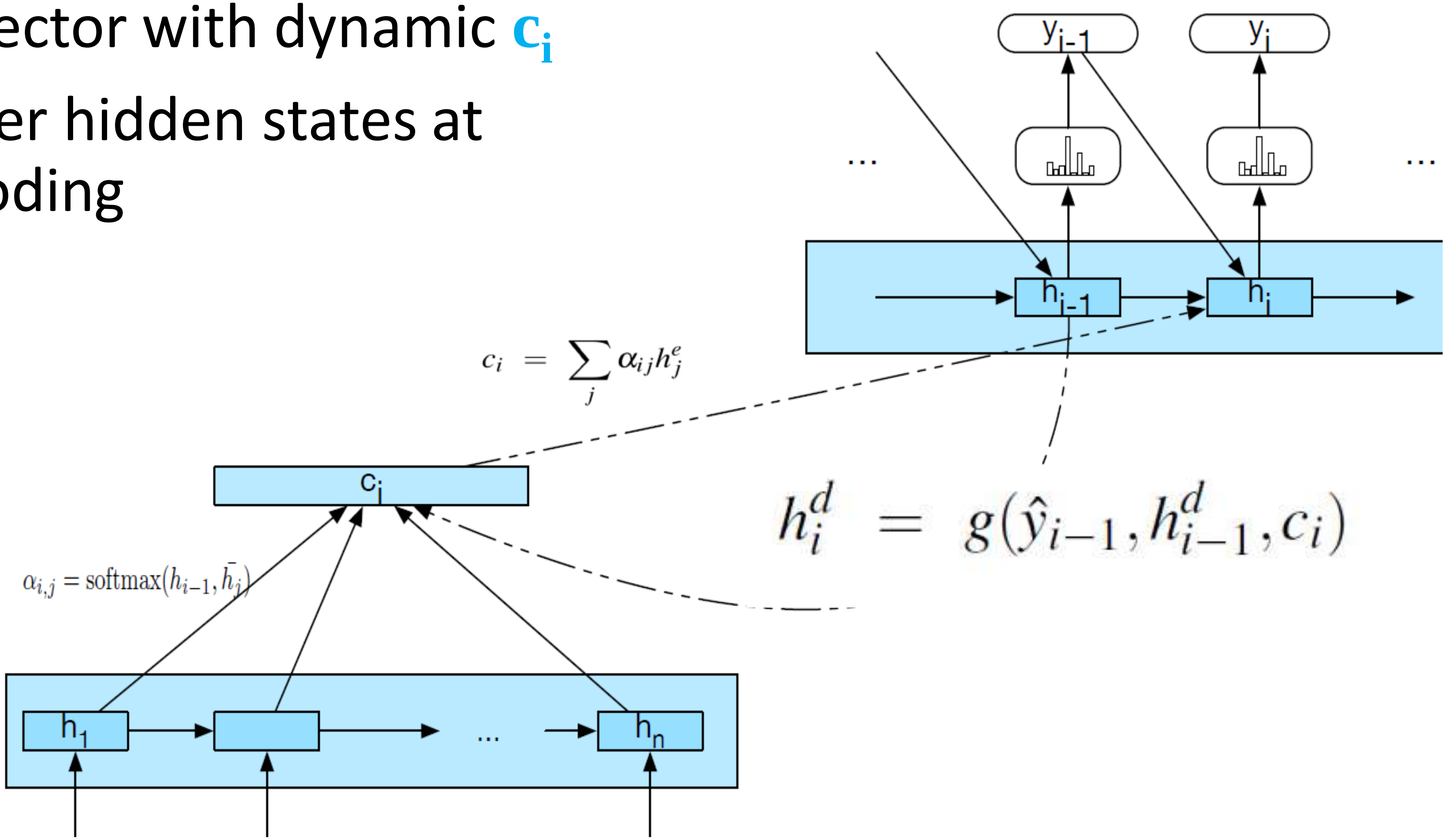
Attention (1): dynamically derived context

- 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 ?

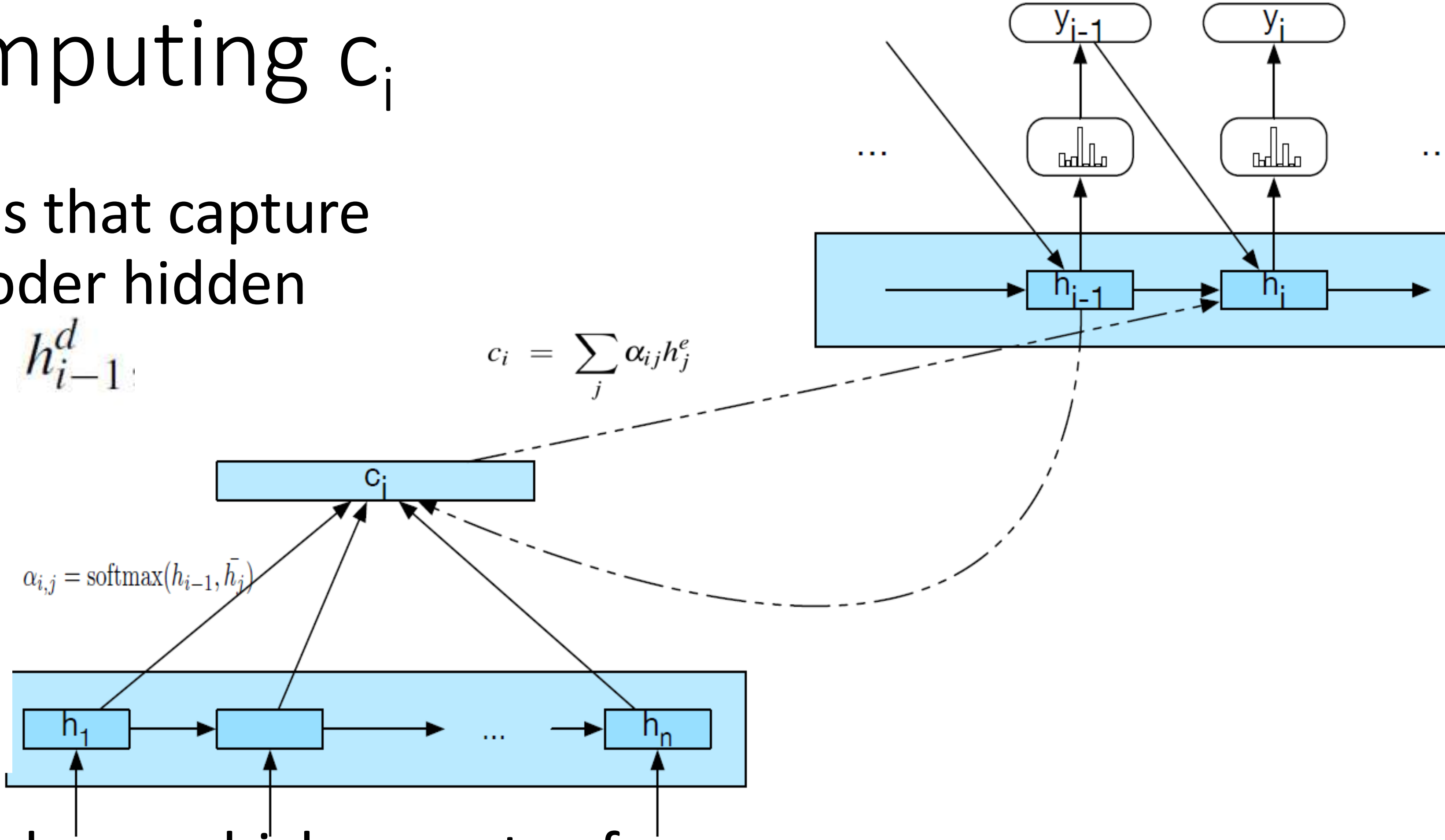
Attention (2): computing c_i

- 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.

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

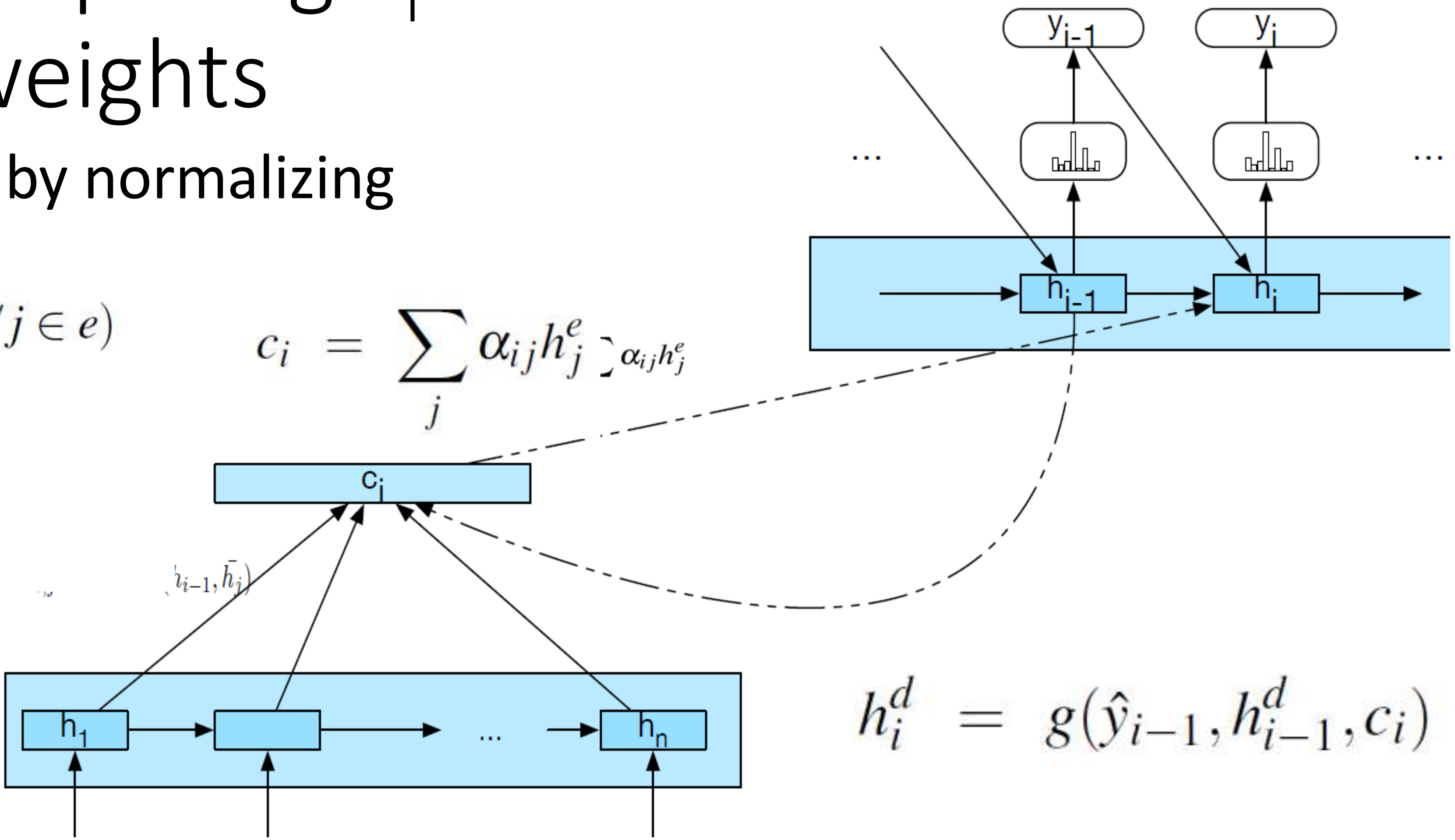
Attention (3): computing c_i From scores to weights

- Create vector of weights by normalizing scores

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

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

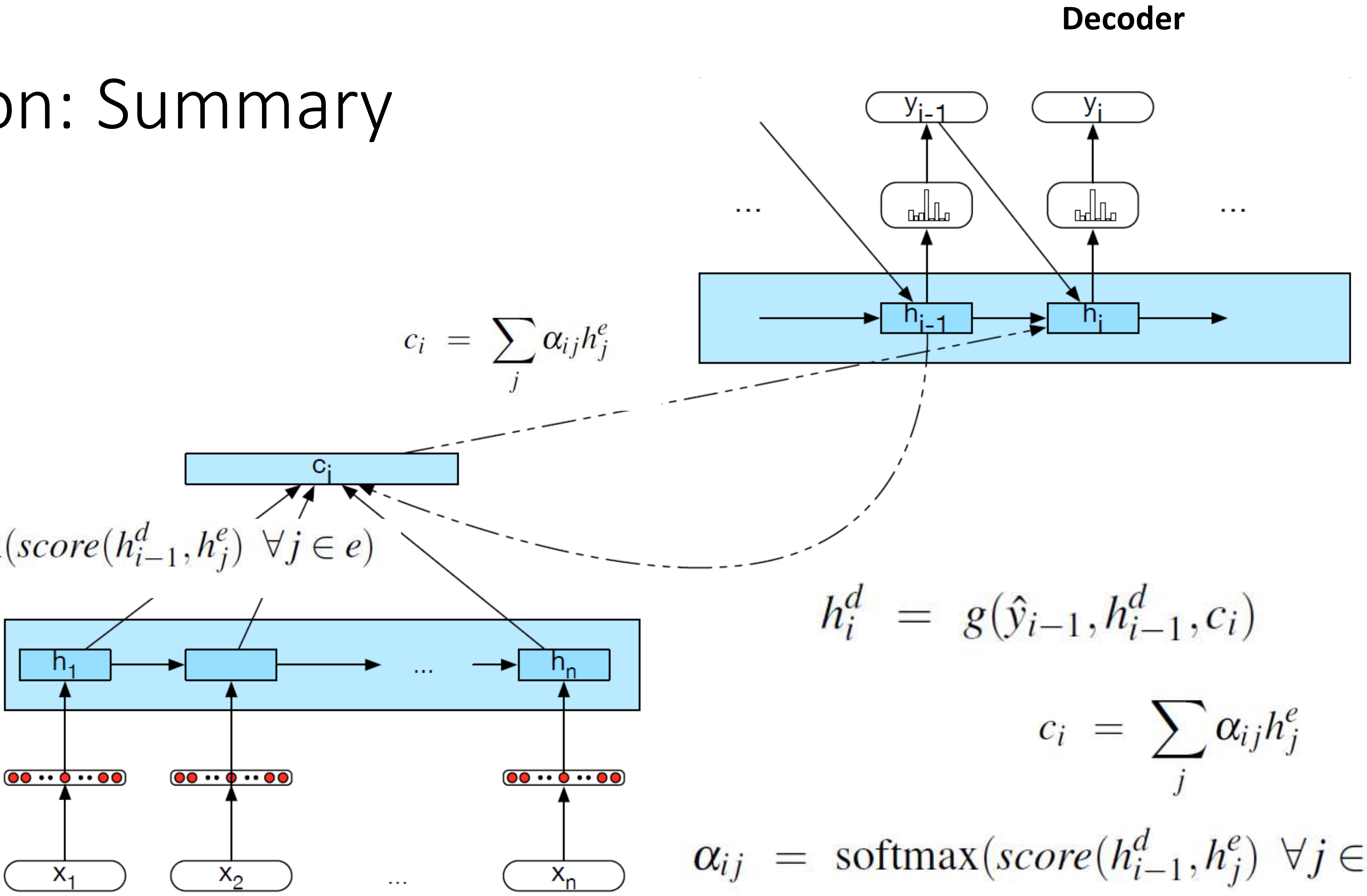
$$c_i = \sum_j \alpha_{ij} h^e_j$$



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

- **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



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

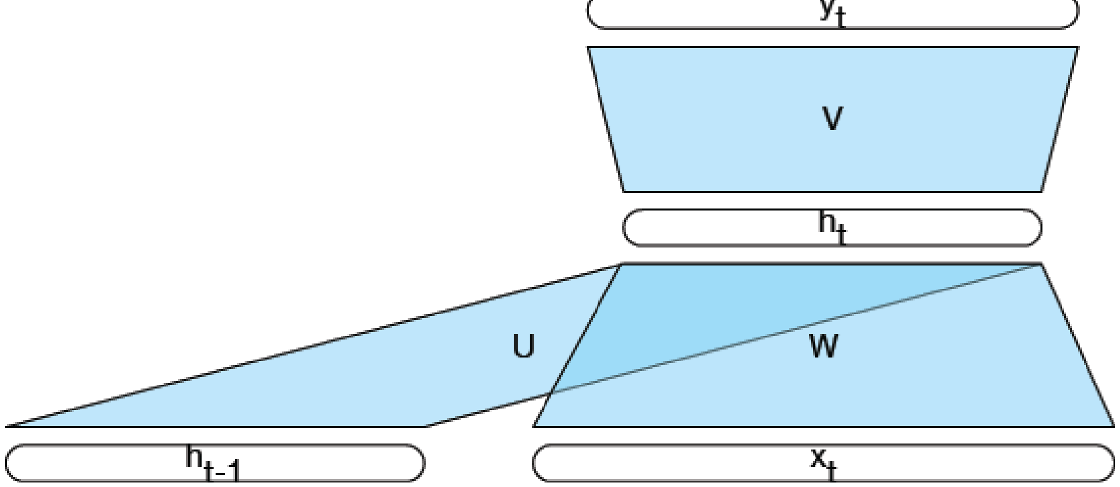
$$\alpha_{ij} = \text{softmax}(score(h_{i-1}^d, h_j^e)\ \forall j \in e)$$

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

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

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

Explain Y. Goldberg different n



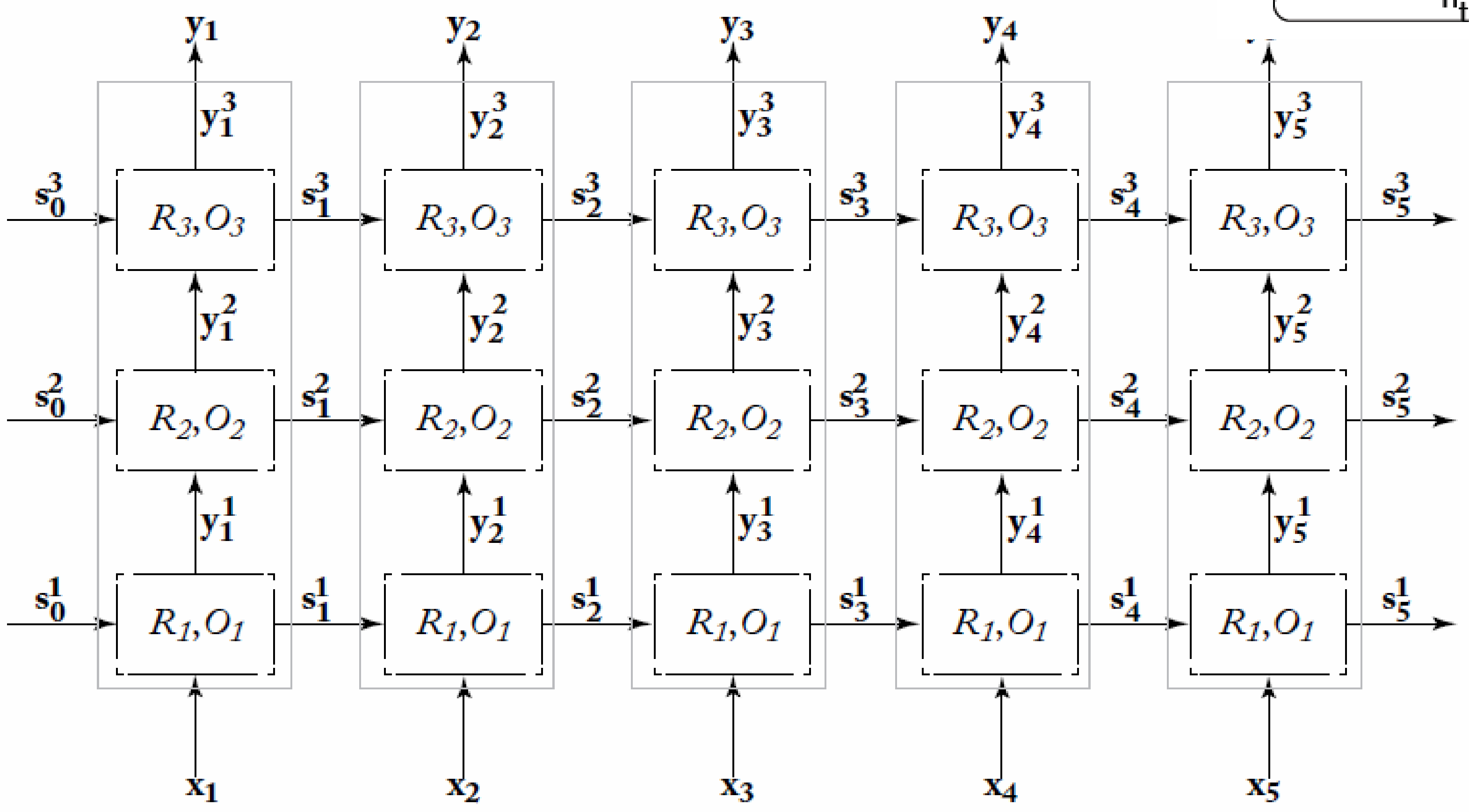


Figure 14.7: A three-layer ("deep") RNN architecture.

Intro to Encoder-Decoder and Attention (Goldberg's notation)

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

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

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

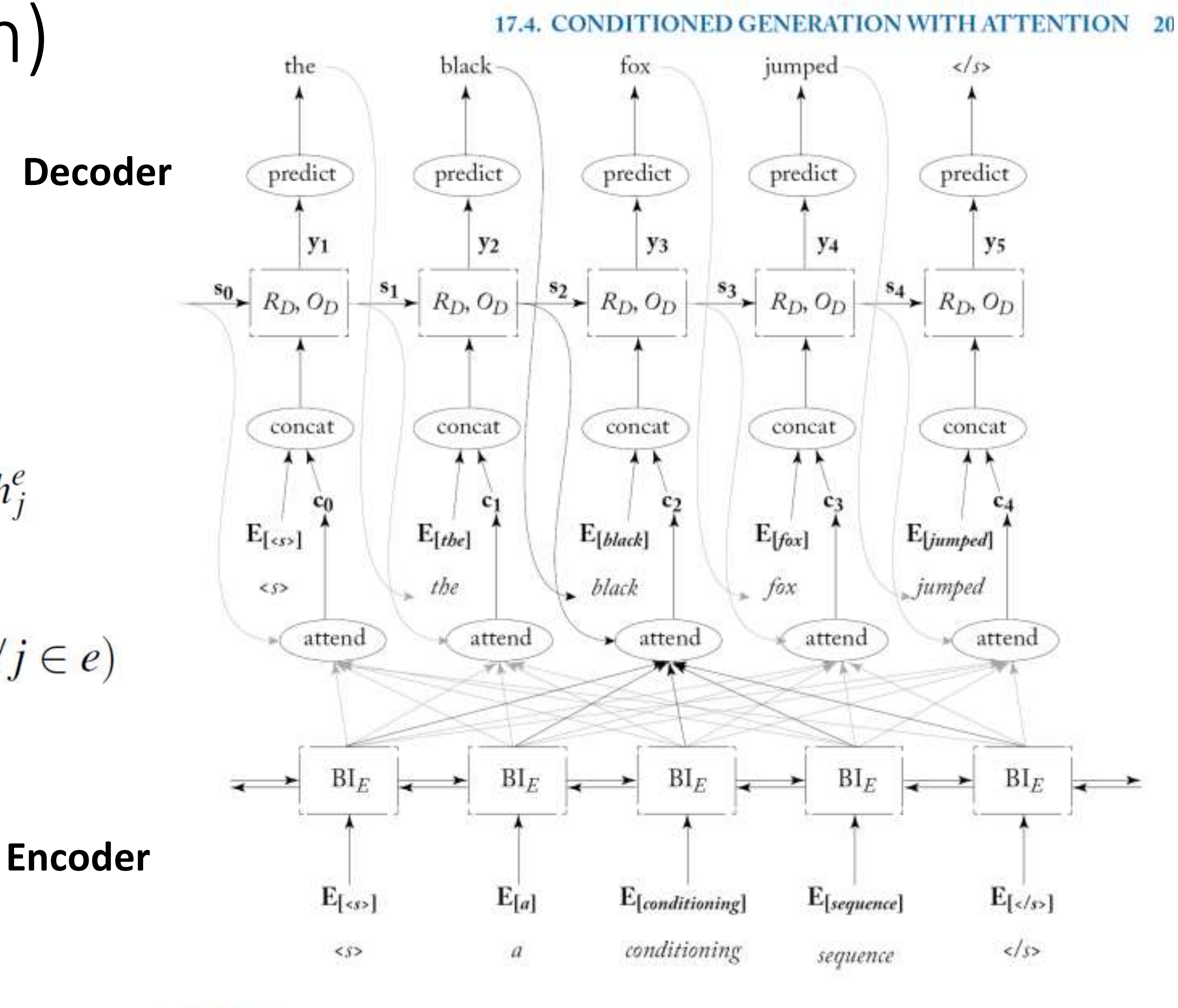


Figure 17.5: Sequence-to-sequence RNN generator with attention.

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

Transformers (Attention is all you need 2017)

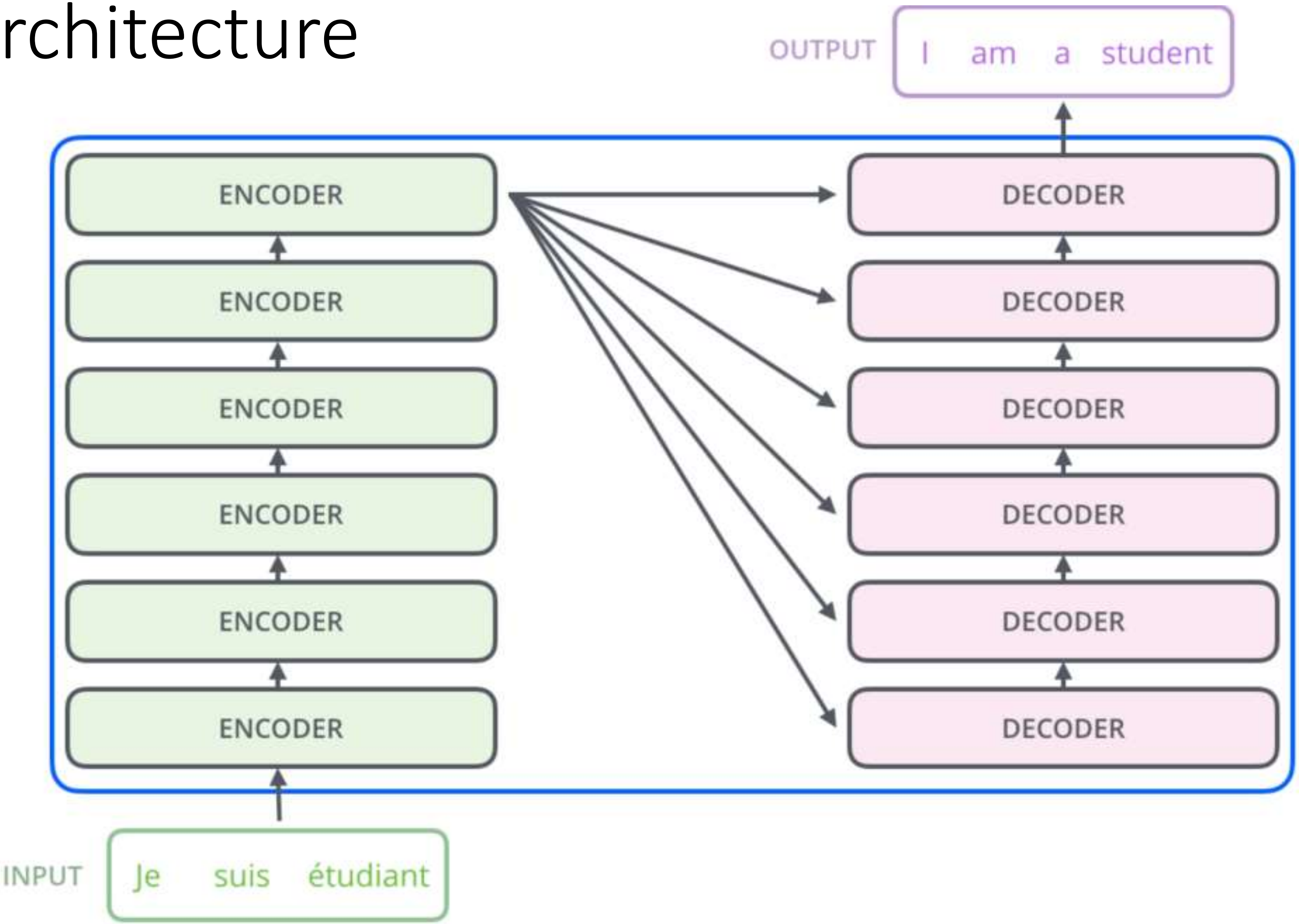
- **Just an introduction:** These are two valuable resources to learn more details on the architecture and implementation
- Also Assignment 4 will help you learn more about Transformers

- http://nlp.seas.harvard.edu/2018/04/03/attention.html

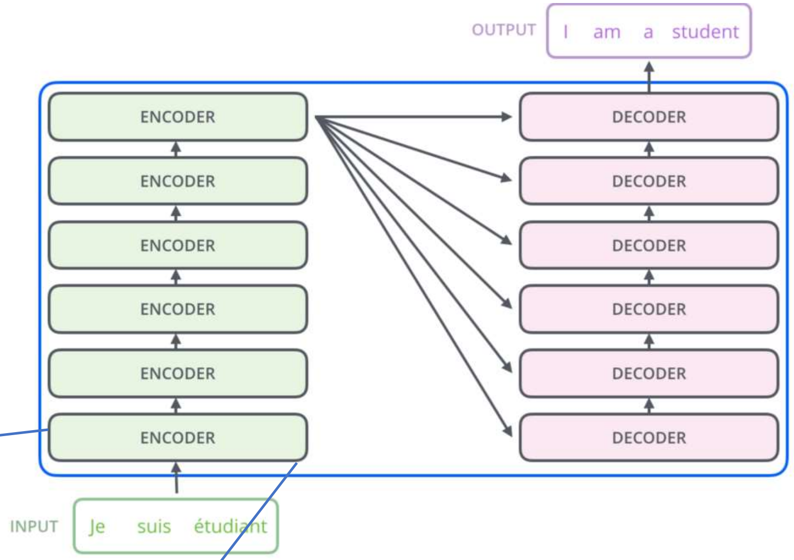
- https://jalammar.github.io/illustrated-transformer/ (slides come from this source)

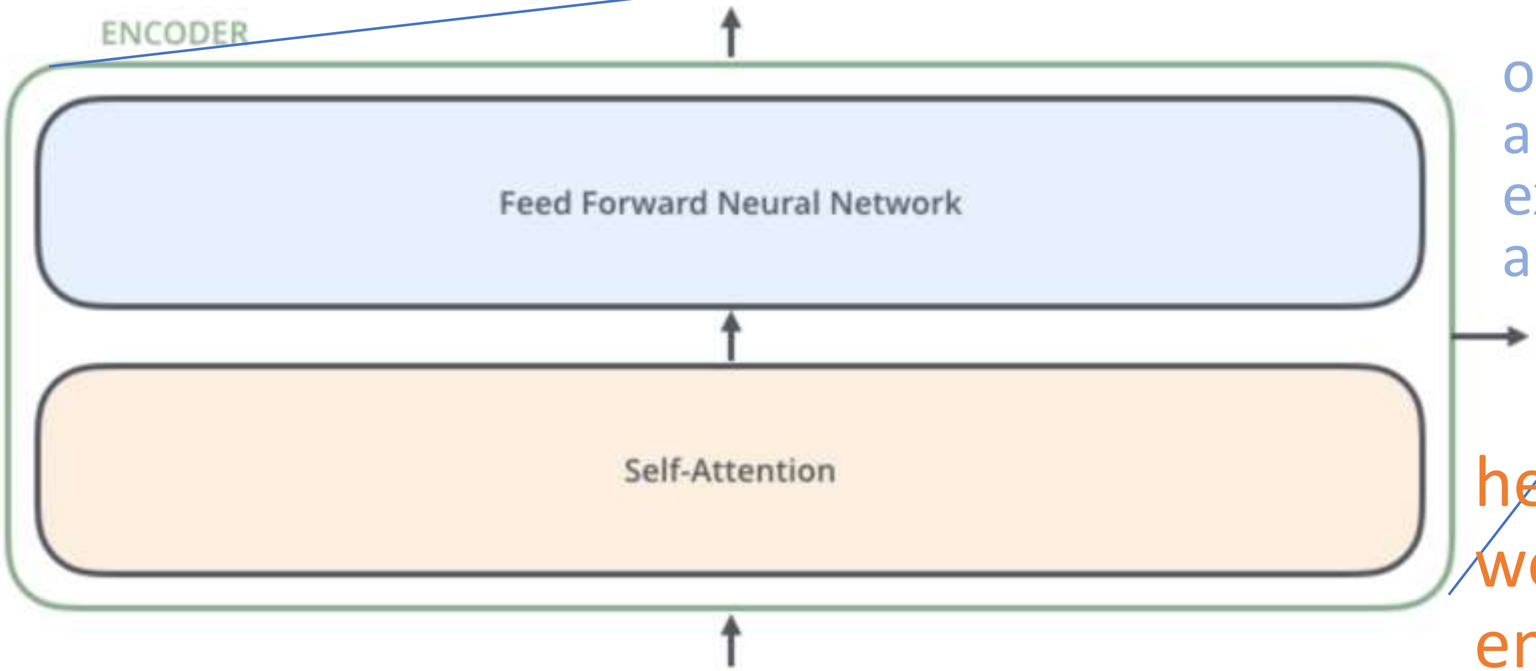
High-level architecture

- 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



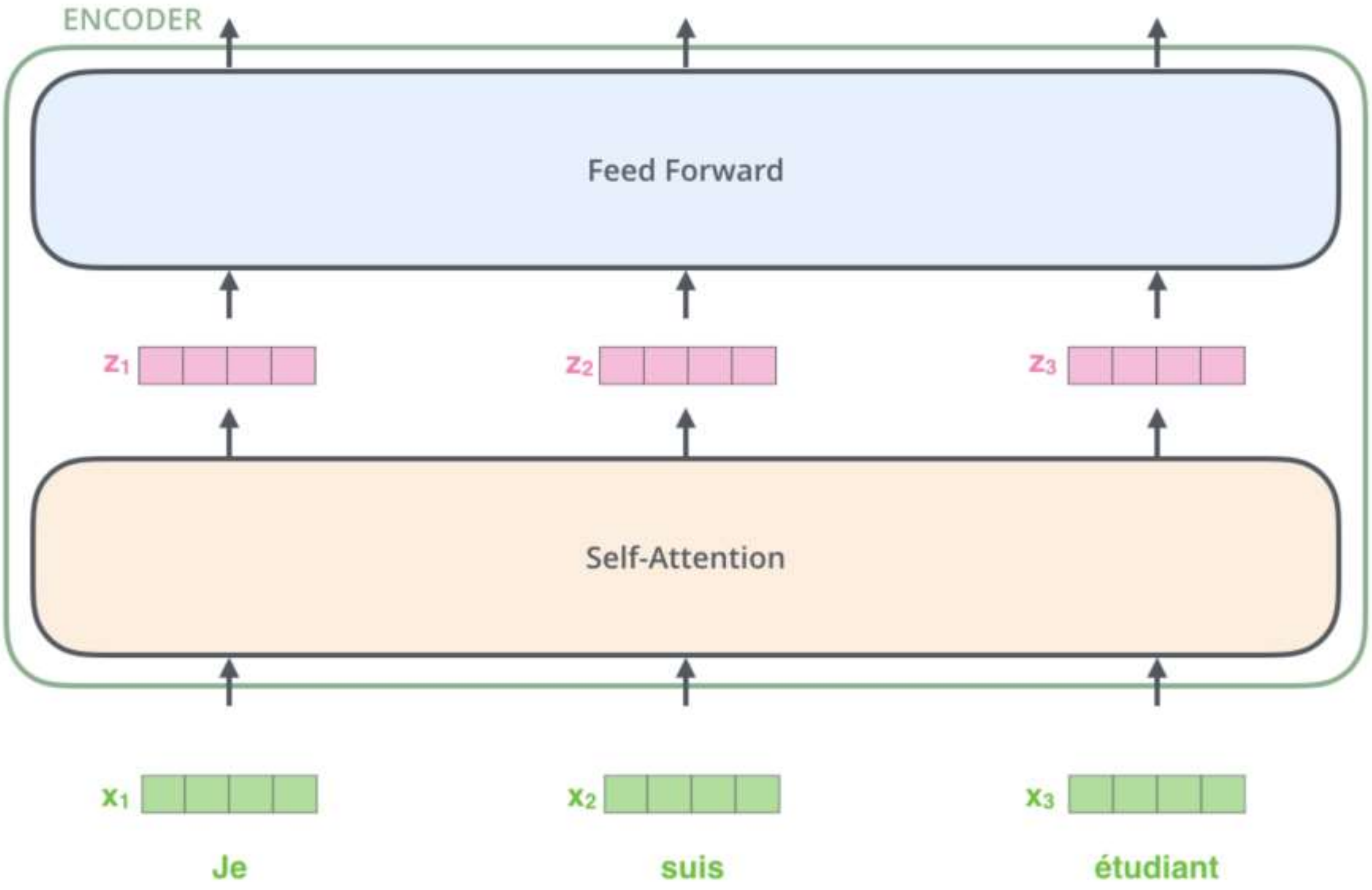


outputs of the self-attention are fed to a feed-forward neural network. The exact same one is independently applied to each position.

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.

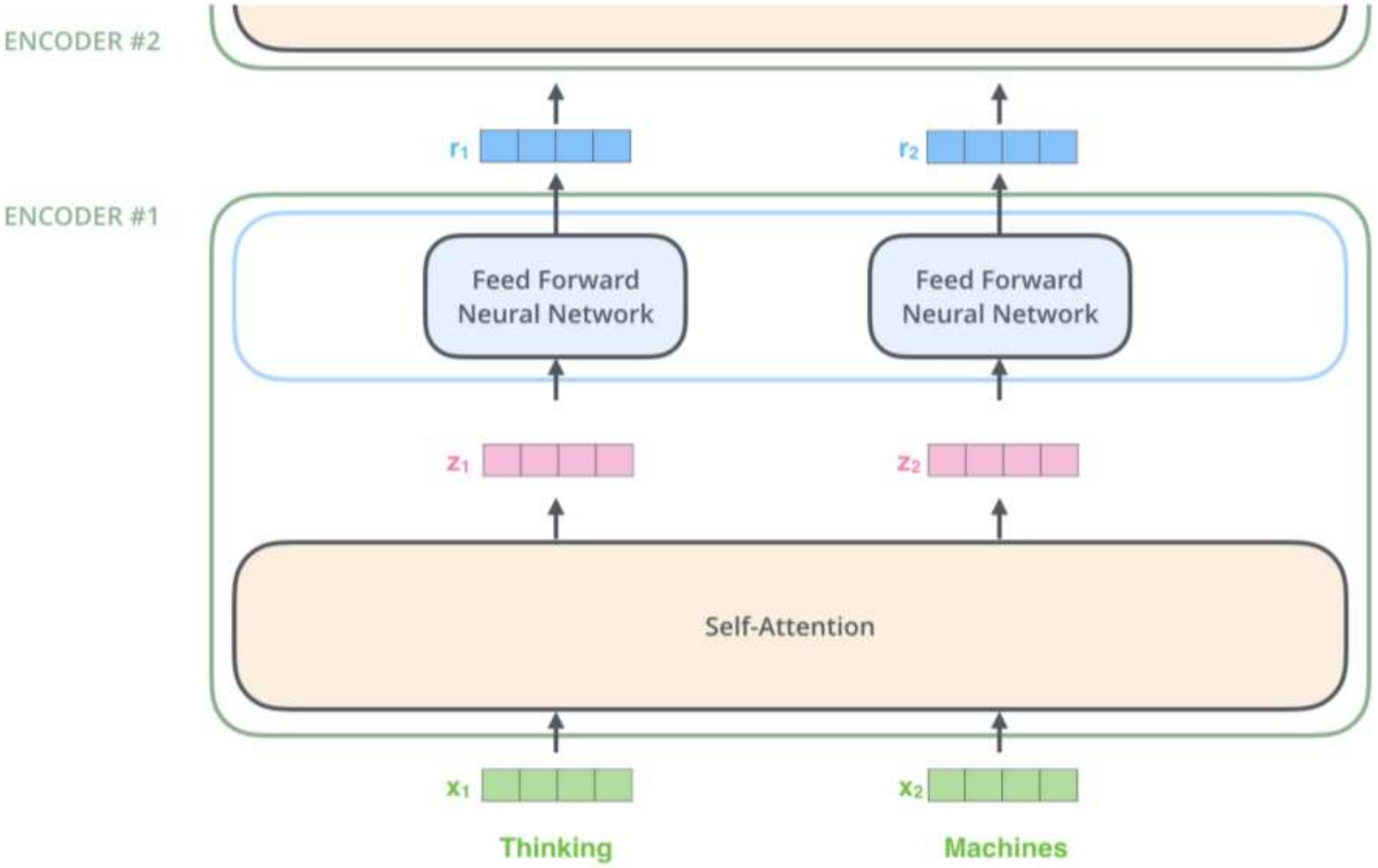
- There are dependencies between these paths in the self-attention layer.
- Feed-forward layer does not have those dependencies => various paths can be executed in parallel !



Word embeddings

Visually clearer on two words

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



Word embeddings

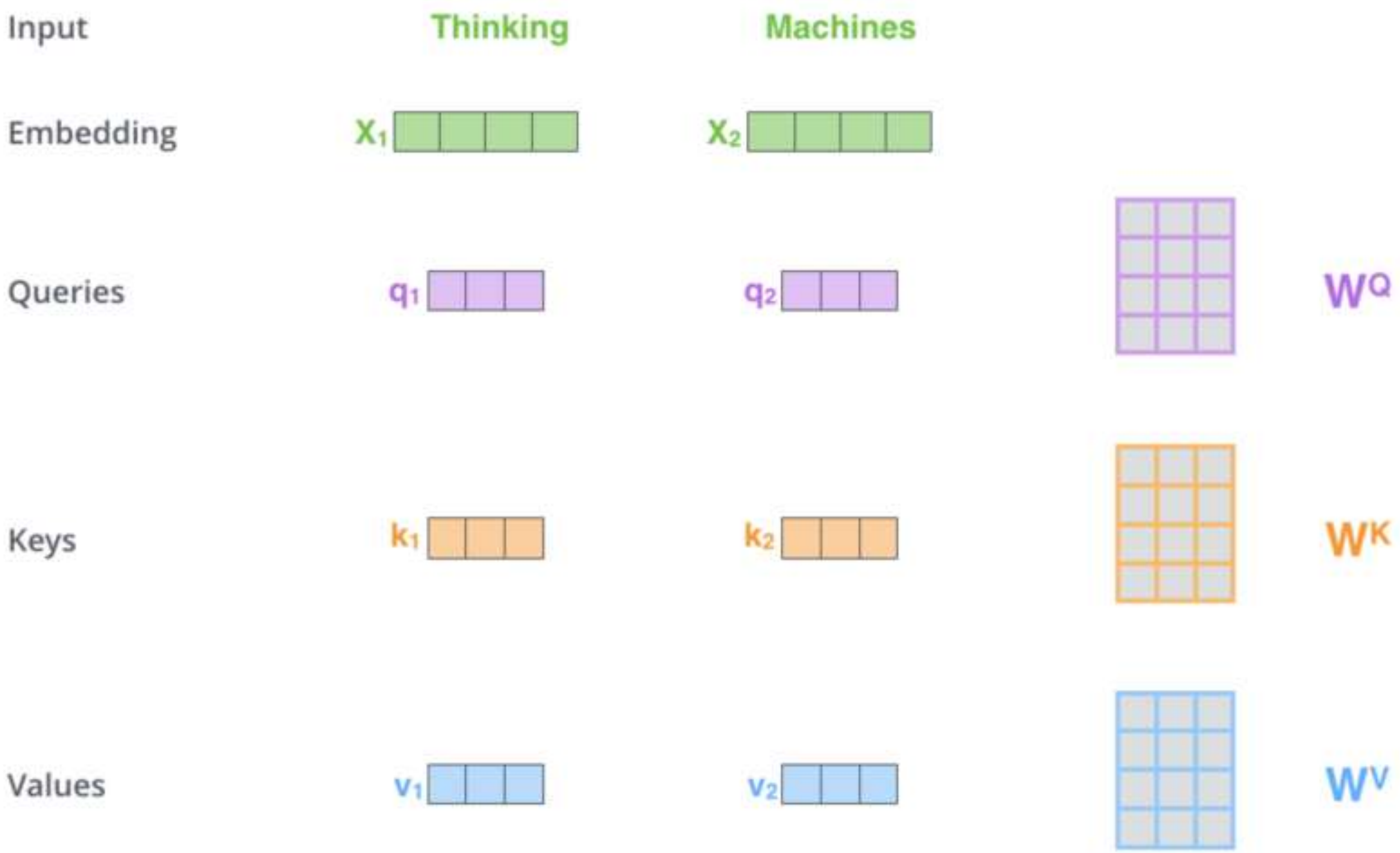
Self-Attention

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).

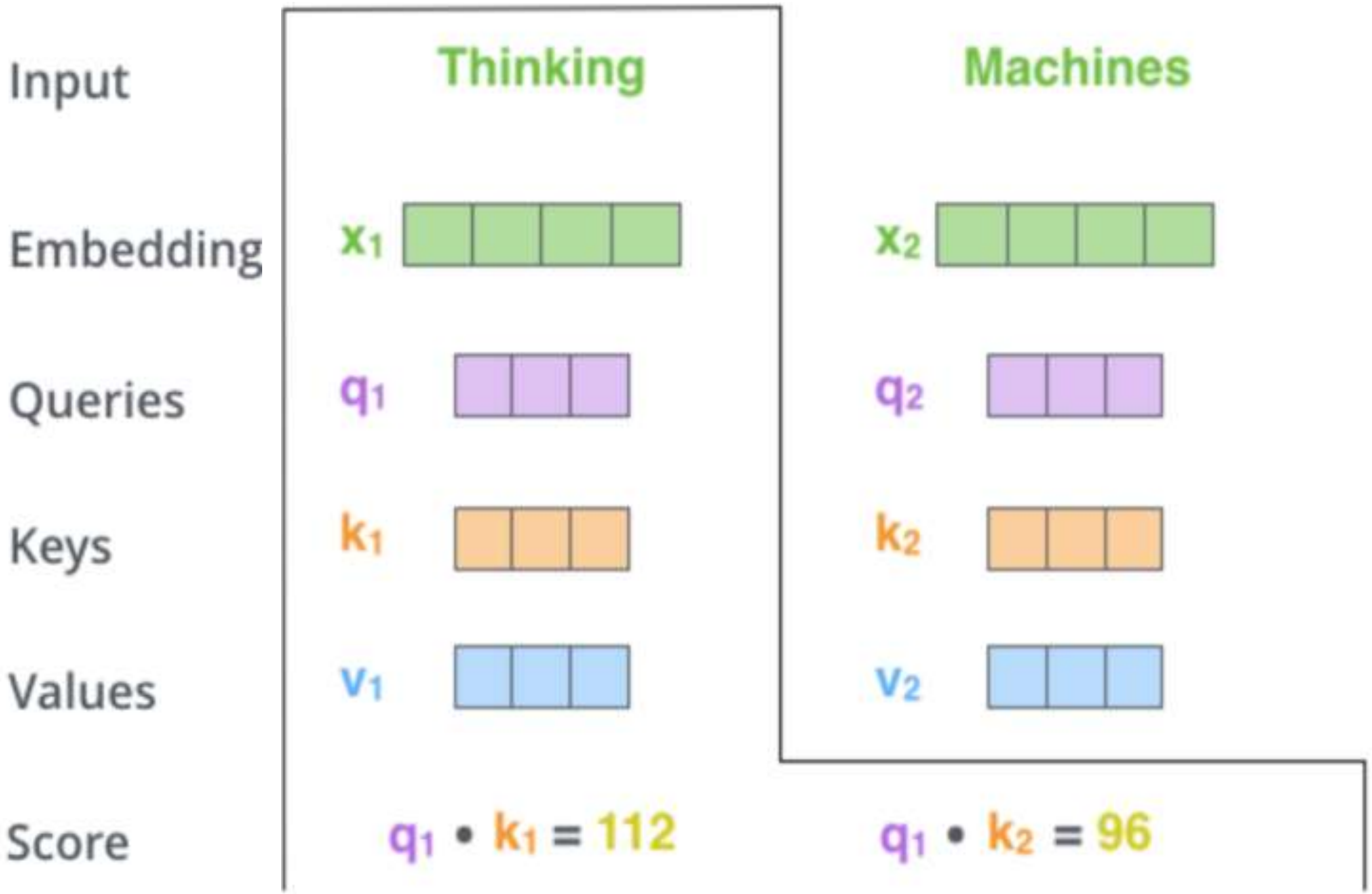
by multiplying the embedding by three matrices that we **trained** during the training process.



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.

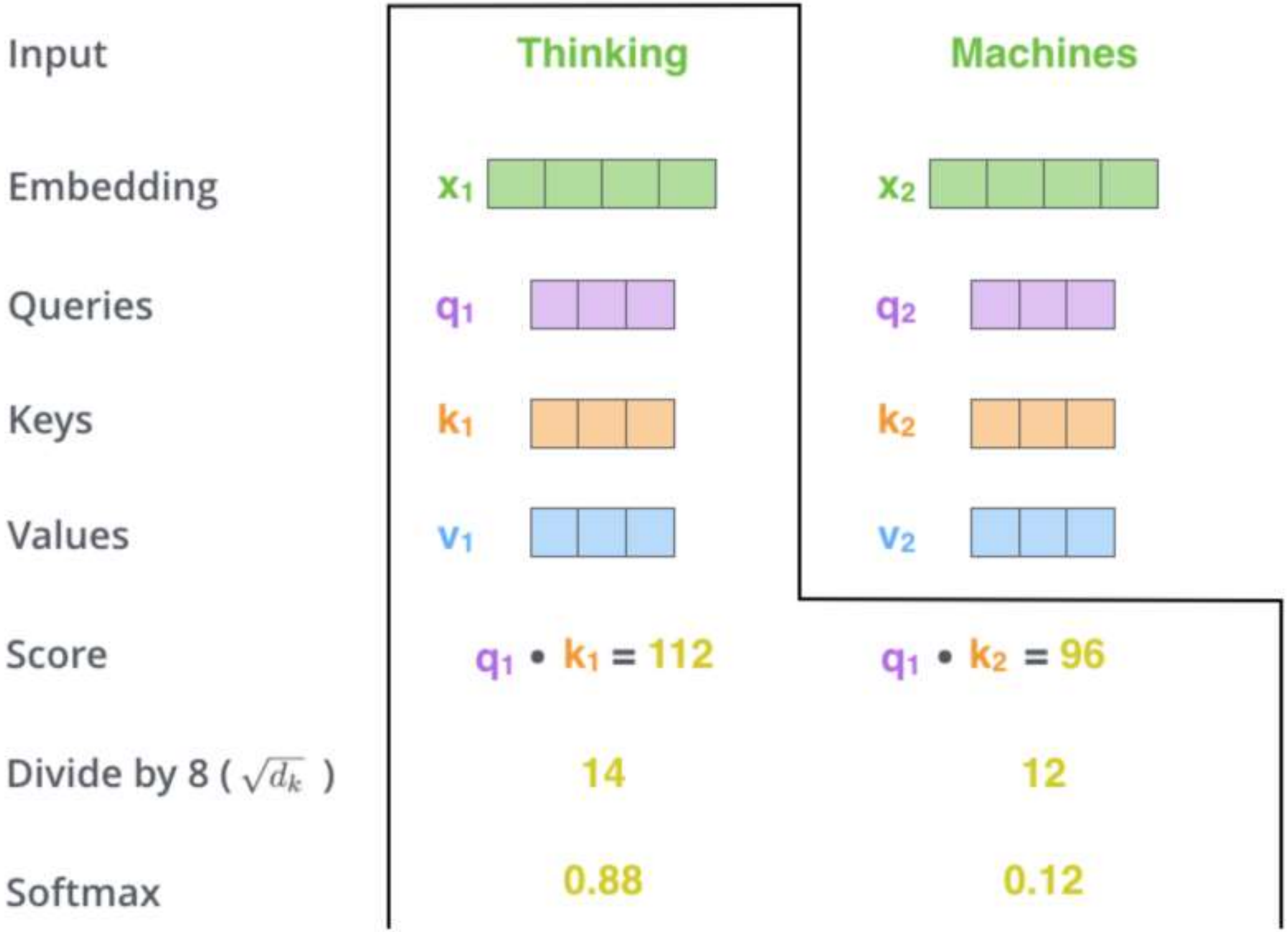
Take dot product of the query vector with the key vector of the respective word we're scoring.



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/