

# **Understanding Large Language Models**

Carsten Eickhoff, Michael Franke and Polina Tsvilodub

**Session 03: LSTMs & Transformers**

# Main learning goals

## 1. Recap

- backpropagation through time (BTTP)
- vanishing (and exploding ) gradients

## 2. Long Short-Term Memory

- intuition, architecture, variants

## 3. Bi-LSTM

- model architecture, stacked Bi-LSTMs, ELMo

## 4. Attention

- intuition, architecture, early transparency

## 5. Self-Attention

- contextualized word embeddings, architecture

## 6. Transformers

- transformer encoders, multi-headed self-attention, decoders, BERT

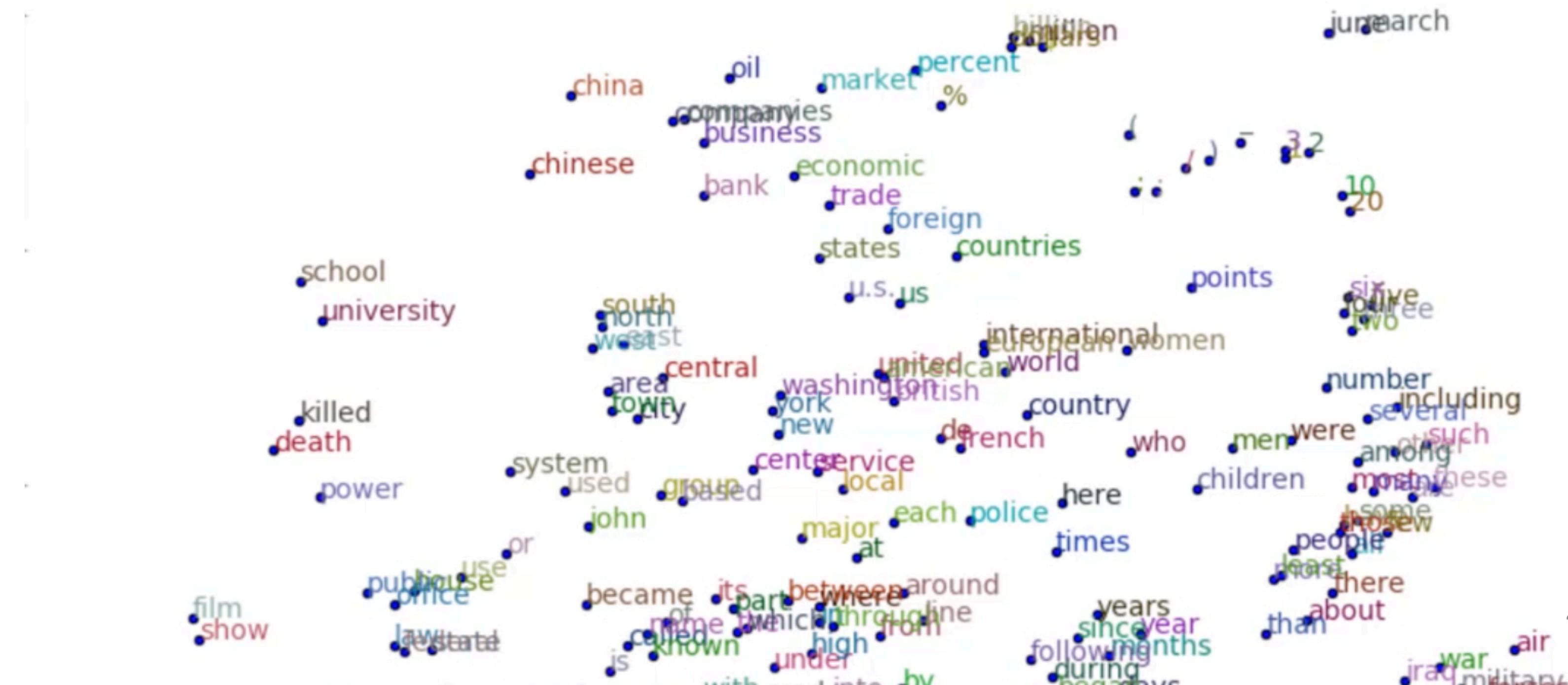


**Recap**

# Recap: Embeddings

Distributed Representations: dense vectors (aka embeddings) that aim to convey the meaning of tokens:

- “word embeddings” refer to when you have type-based representations
  - “contextualized embeddings” refer to when you have token-based representations



## Recap: Language Modeling

An **auto-regressive LM** is one that only has access to the previous tokens (and the outputs become the inputs).

Evaluation: Perplexity

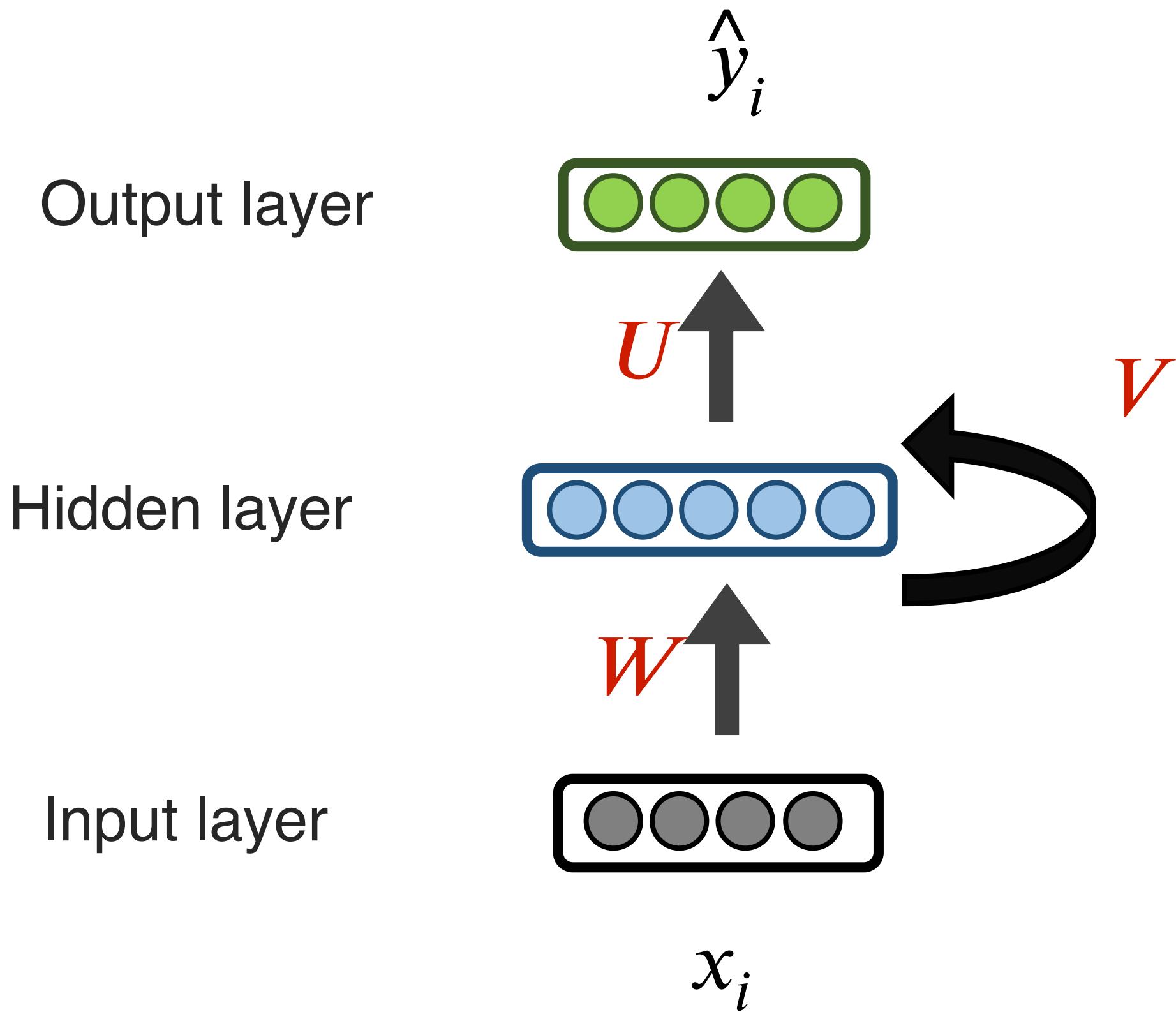
A **masked LM** can peek ahead, too. It “masks” a word within the context (e.g., the center word).

Evaluation: downstream NLP tasks that use the learned embeddings.

**Both of these can produce useful word embeddings!**

# Recap: RNNs

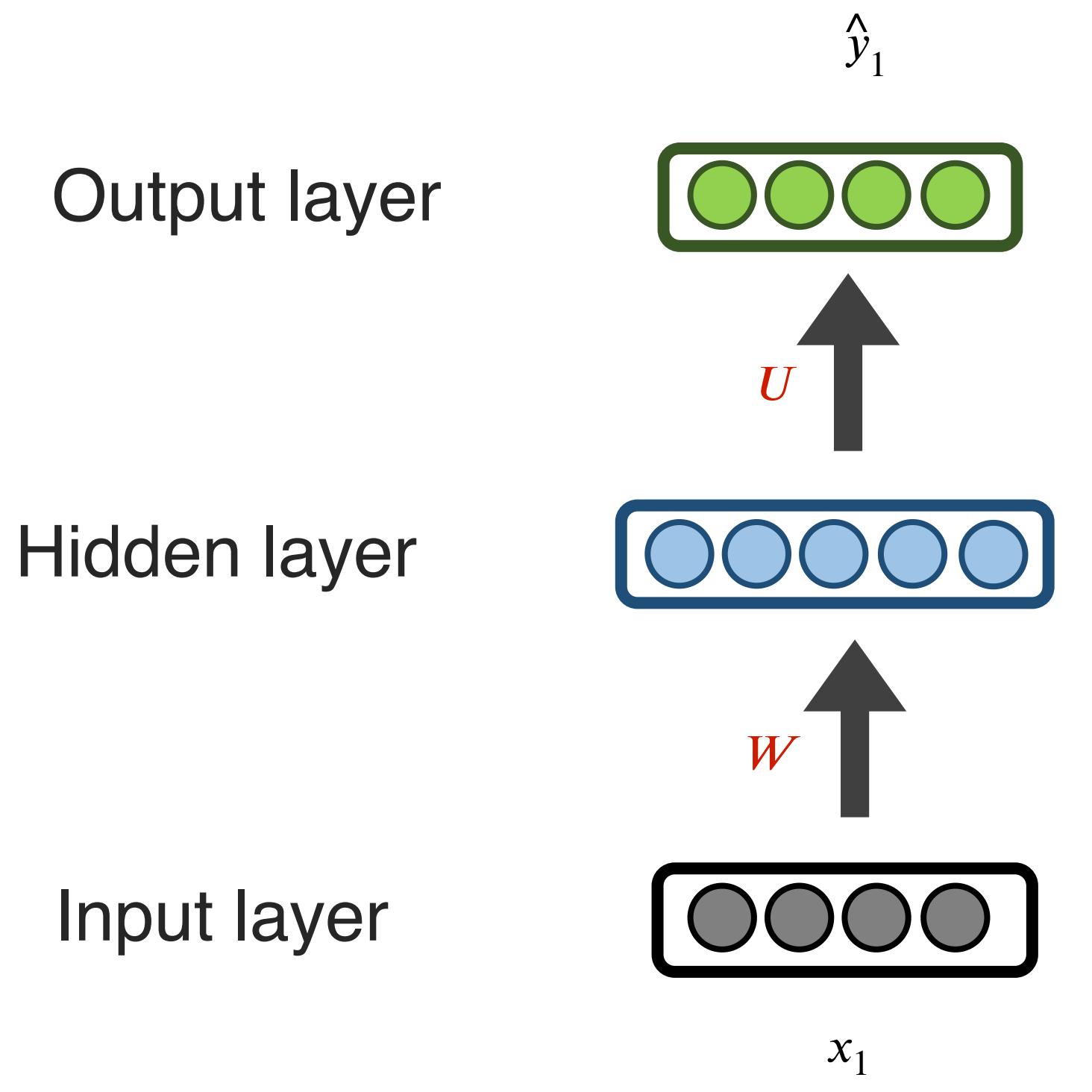
- RNNs help capture more context while avoiding sparsity, storage, and compute issues!
- The hidden layer is what we care about. It represents the word's "meaning".



# Training Process

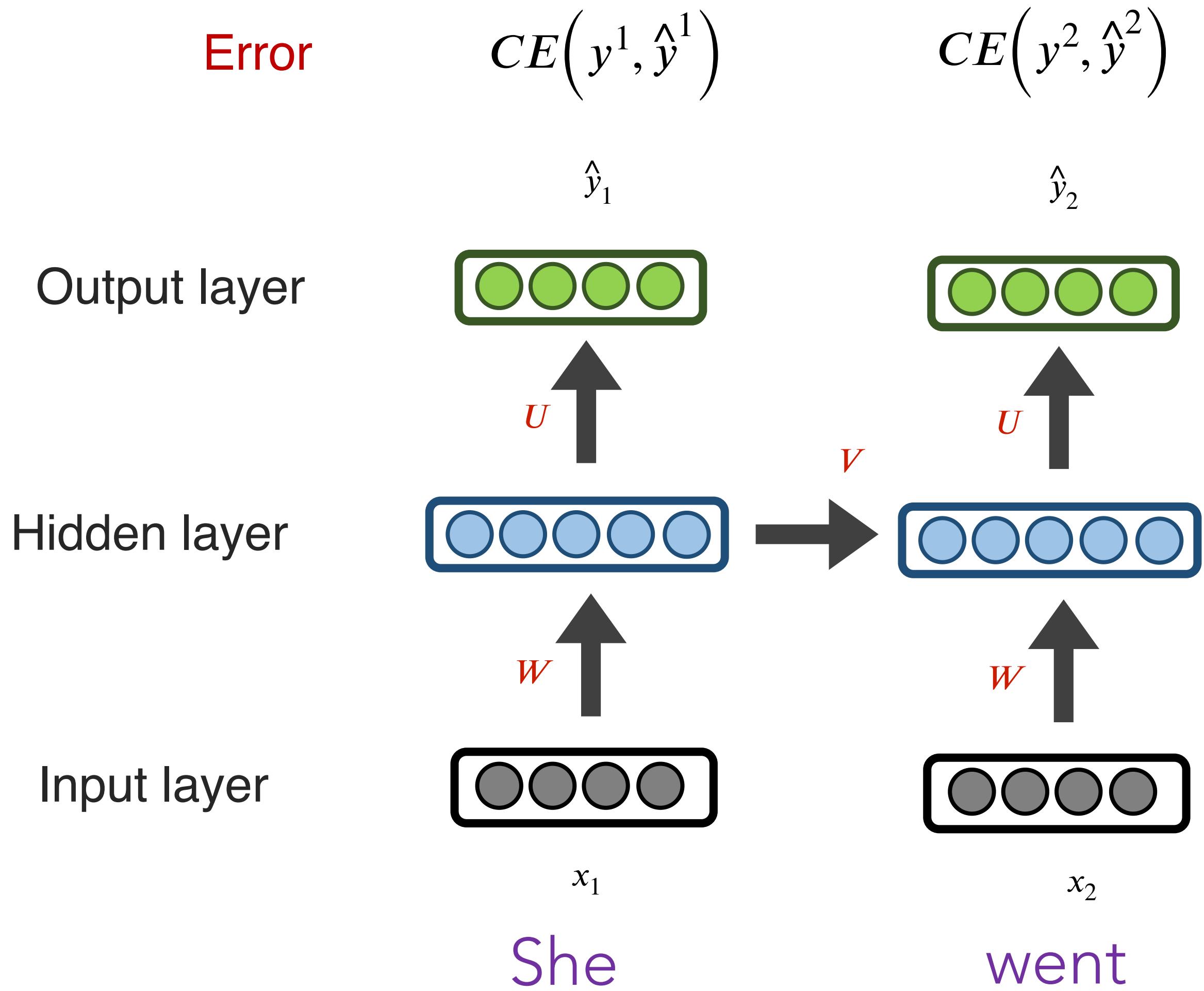
$$CE\left(y^i, \hat{y}^i\right) = - \sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$

Error  $CE\left(y^1, \hat{y}^1\right)$



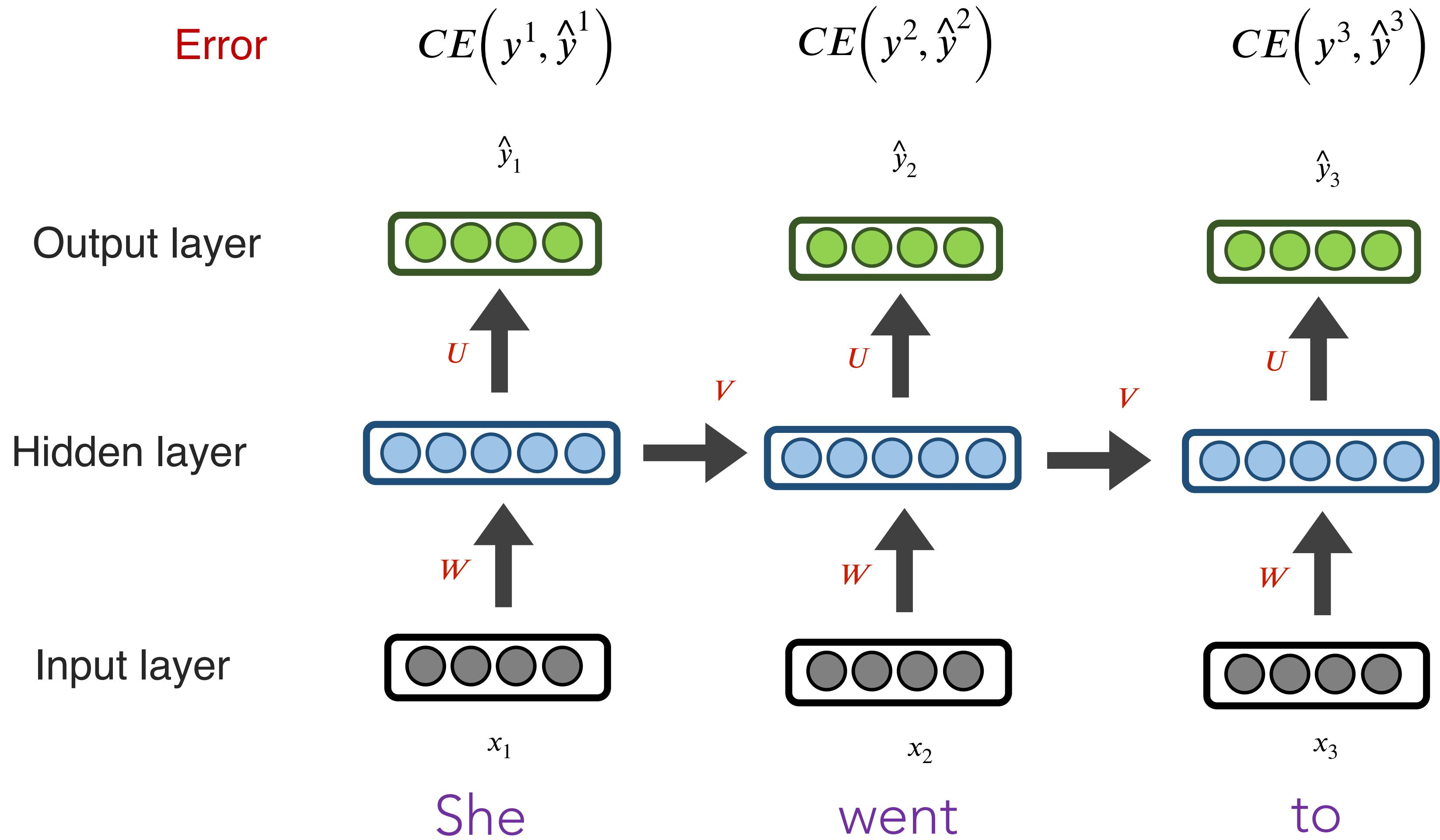
# Training Process

$$CE\left(y^i, \hat{y}^i\right) = - \sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$



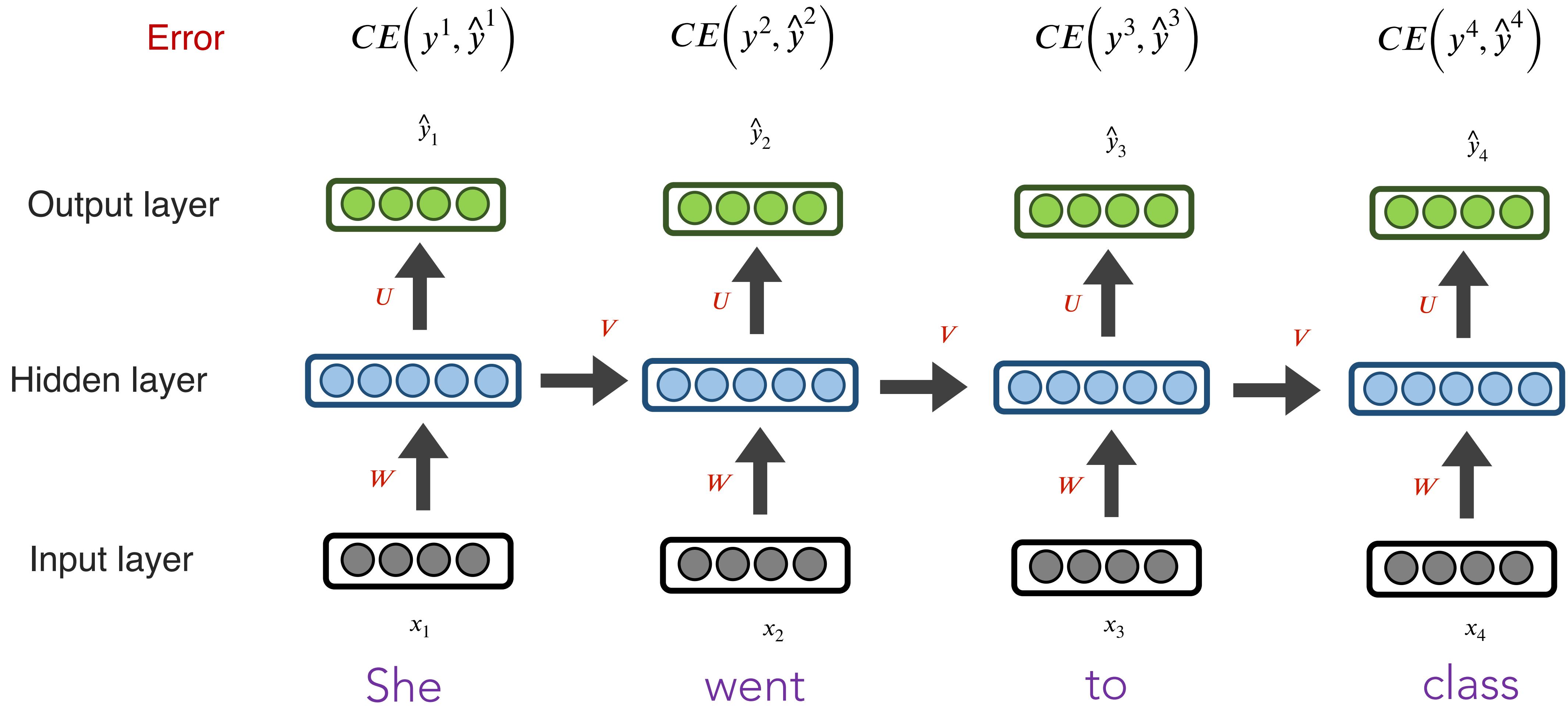
# Training Process

$$CE(y^i, \hat{y}^i) = - \sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$



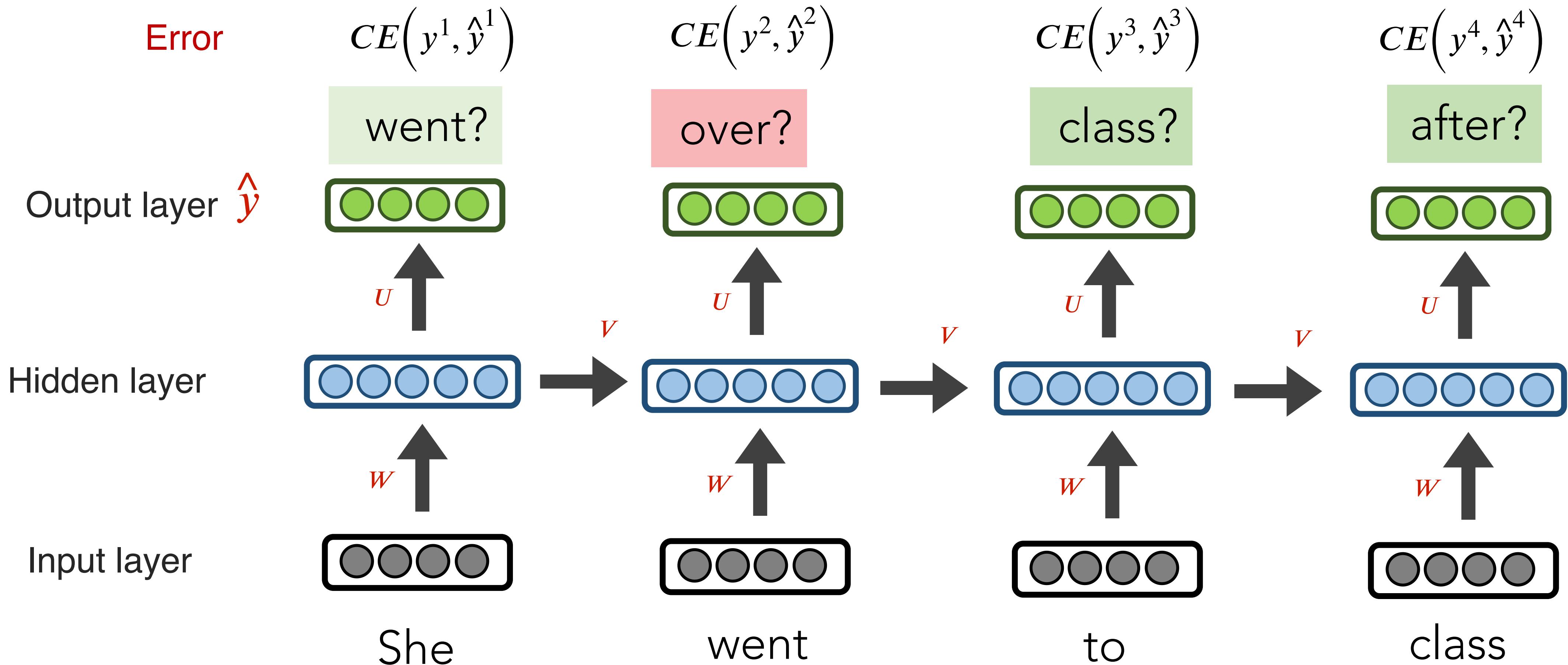
# Training Process

$$CE\left(y^i, \hat{y}^i\right) = - \sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$



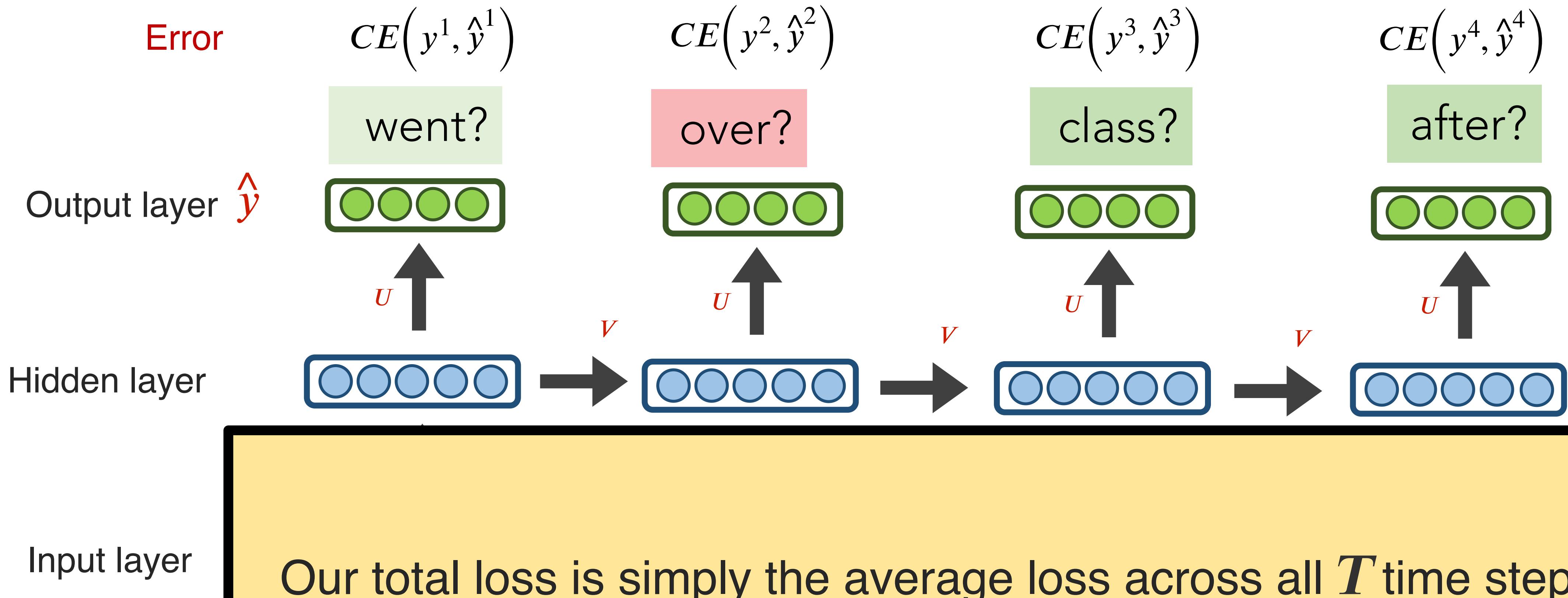
# Training Process

$$CE(y^i, \hat{y}^i) = - \sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$



# Training Process

$$CE(y^i, \hat{y}^i) = - \sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$



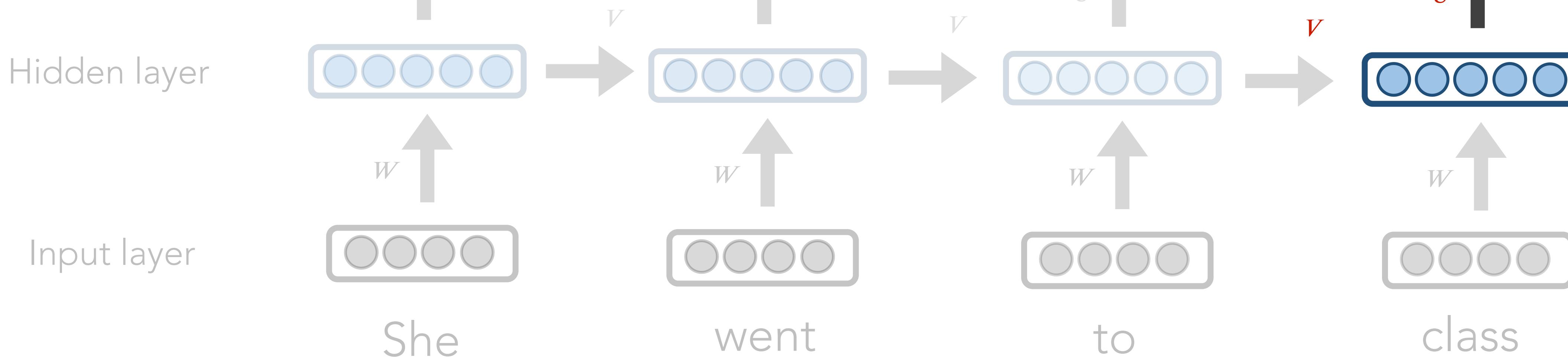
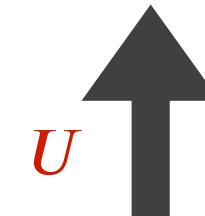
# Training Details

To update our weights (e.g.  $V$ ), we calculate the gradient of our loss w.r.t. the repeated weight matrix (e.g.,  $\frac{\partial L}{\partial V}$ ).

Using the chain rule, we trace the derivative all the way back to the beginning, while summing the results.

$$CE(y^4, \hat{y}^4)$$

after?

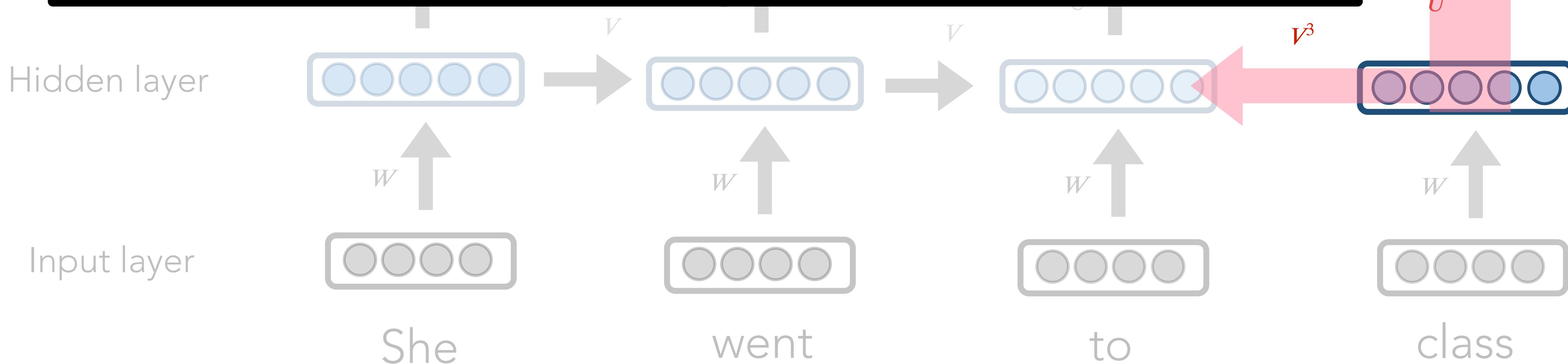


# Training Details

$$\frac{\partial L}{\partial V}$$

To update our weights (e.g.  $V$ ), we calculate the gradient of our loss w.r.t. the repeated weight matrix (e.g.,  $\frac{\partial L}{\partial V}$ ).

Using the chain rule, we trace the derivative all the way back to the beginning, while summing the results.

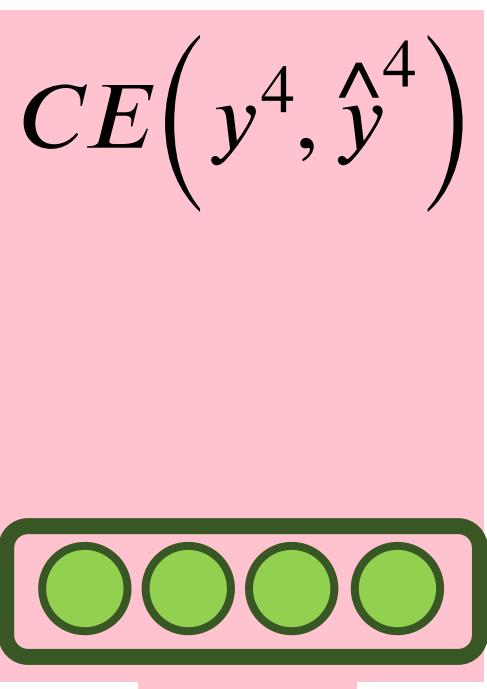


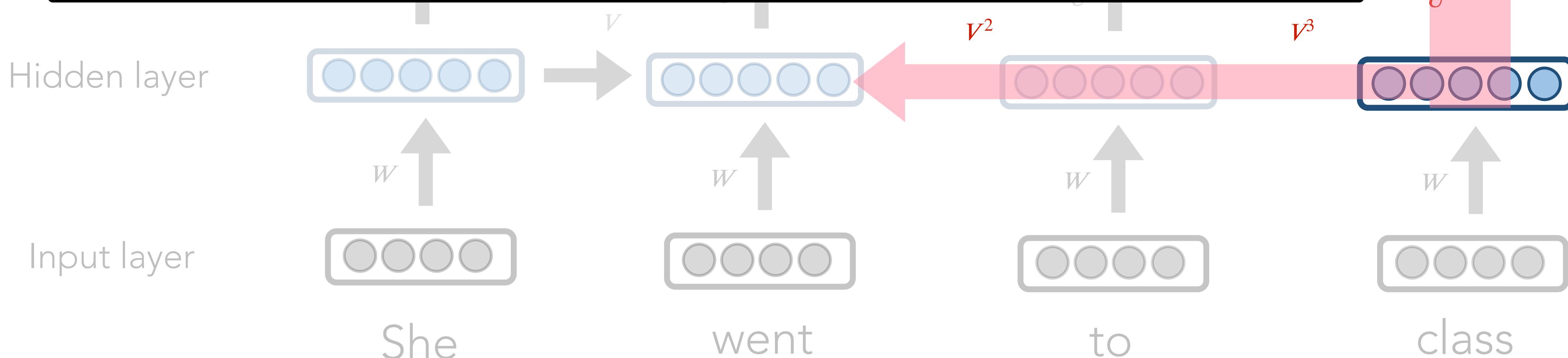
# Training Details

$$\frac{\partial L}{\partial V}$$

To update our weights (e.g.  $V$ ), we calculate the gradient of our loss w.r.t. the repeated weight matrix (e.g.,  $\frac{\partial L}{\partial V}$ ).

Using the chain rule, we trace the derivative all the way back to the beginning, while summing the results.

$$CE(y^4, \hat{y}^4)$$




# Training Details

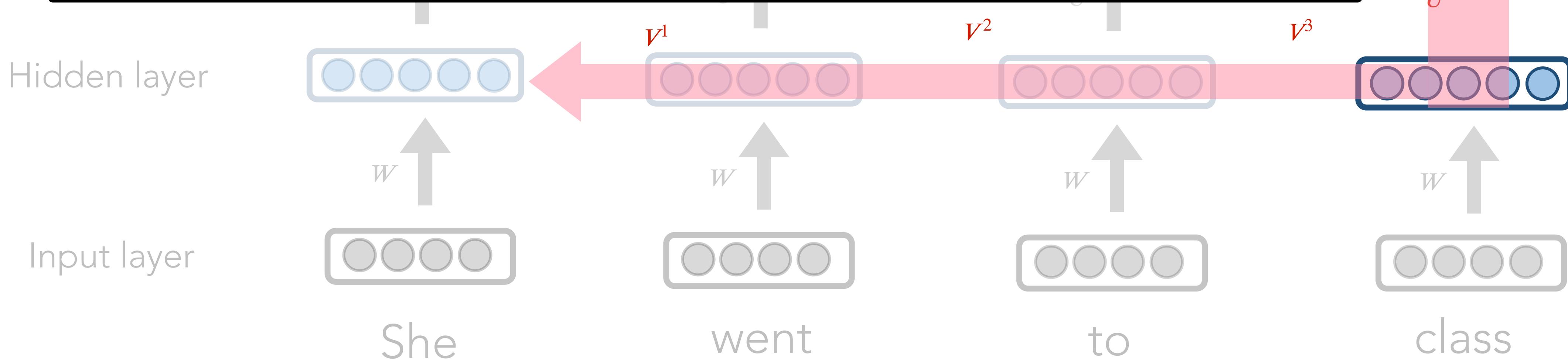
$$\frac{\partial L}{\partial V}$$

To update our weights (e.g.  $V$ ), we calculate the gradient of our loss w.r.t. the repeated weight matrix (e.g.,  $\frac{\partial L}{\partial V}$ ).

Using the chain rule, we trace the derivative all the way back to the beginning, while summing the results.

$$CE(y^4, \hat{y}^4)$$

CE( $y^4, \hat{y}^4$ )



## Training Details

- This backpropagation through time (BPTT) process is expensive
- Instead of updating after every time step, we tend to do so every  $T$  steps (e.g., every sentence or paragraph)
- This is not equivalent to using only a window size  $T$  (a la n-grams) because we still have “infinite memory”

## Examples: Training on Harry Potter



“Sorry,” Harry shouted, panicking—“I’ll leave those brooms in London, are they?”

“No idea,” said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry’s shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn’t felt it seemed. He reached the teams too.

# Examples: Training on Recipes

Title: CHOCOLATE RANCH BARBECUE

Categories: Game, Casseroles, Cookies, Cookies

Yield: 6 Servings

2 tb Parmesan cheese --- chopped

1 c Coconut milk

3 Eggs, beaten

Place each pasta over layers of lumps. Shape mixture into the moderate oven and simmer until firm. Serve hot in bodied fresh, mustard, orange and cheese.

Combine the cheese and salt together the dough in a large skillet; add the ingredients and stir in the chocolate and pepper.



# RNN Summary

## Strengths

- Can handle infinite-length sequences (not just a fixed-window)
- Has a “memory” of the context (thanks to the hidden layer’s recurrent loop)
- Same weights used for all inputs, so positionality is not wonky/overwritten (like FFNN)

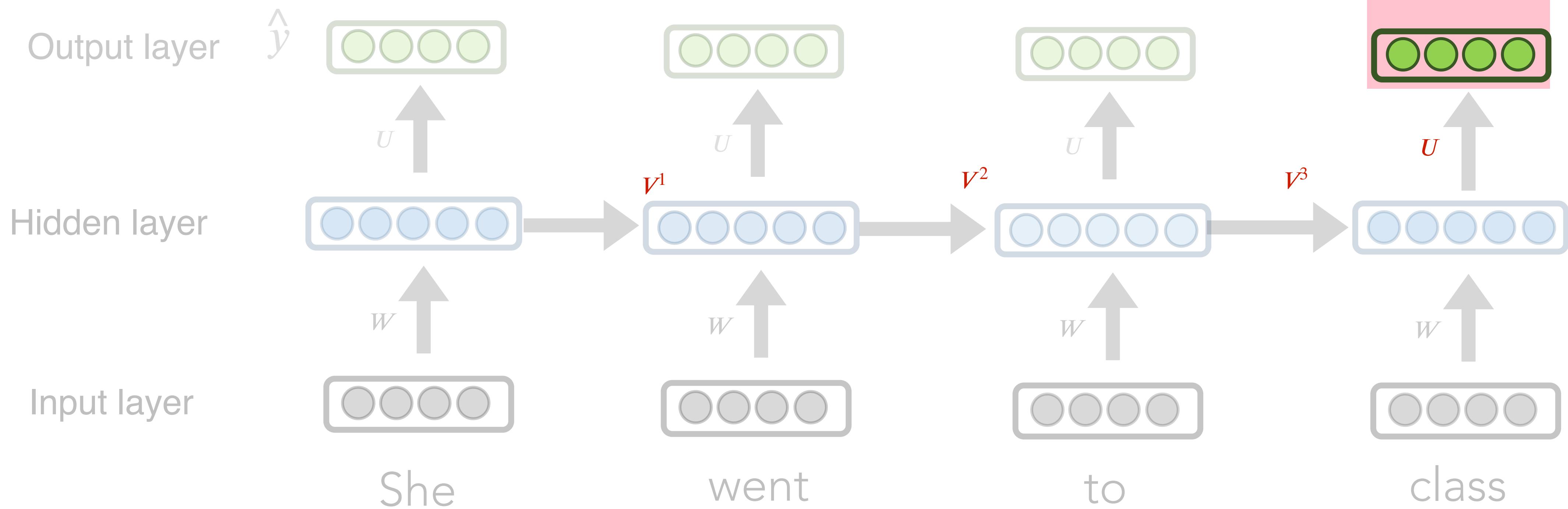
## Issues?

- Slow to train (BPTT)
- Due to “infinite sequence”, gradients can easily vanish or explode
- Has trouble actually making use of long-range context

# Vanishing Gradients

$$\frac{\partial L^4}{\partial V^1}$$

$$\frac{\partial L^4}{\partial V^1} = ?$$

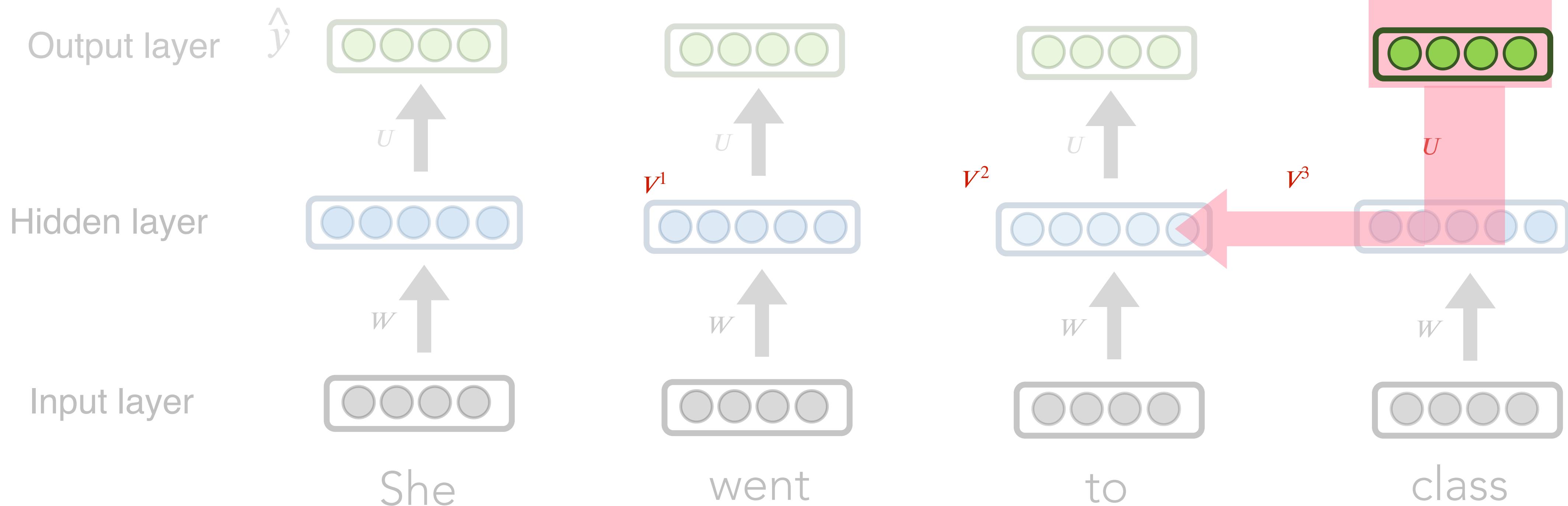


# Vanishing Gradients

$$\frac{\partial L^4}{\partial V^1}$$

$$\frac{\partial L^4}{\partial V^1} = \frac{\partial L^4}{\partial V^3}$$

$$CE(y^4, \hat{y}^4)$$

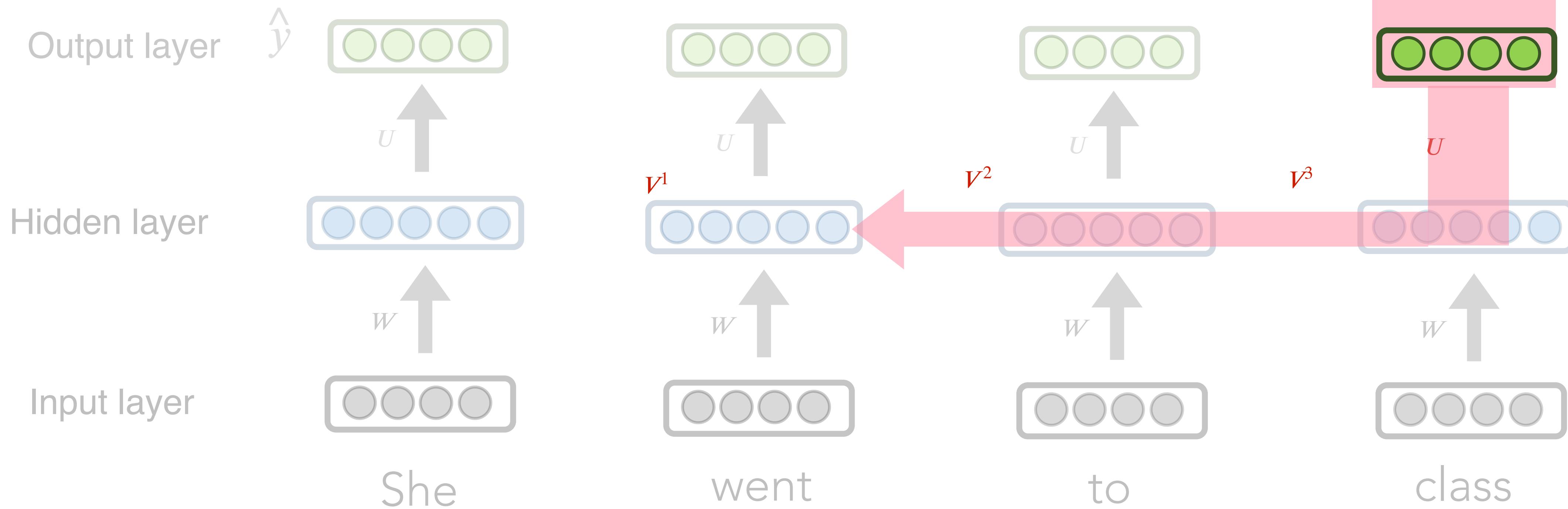


# Vanishing Gradients

$$\frac{\partial L^4}{\partial V^1}$$

$$\frac{\partial L^4}{\partial V^1} = \frac{\partial L^4}{\partial V^3} \frac{\partial V^3}{\partial V^2}$$

$$CE(y^4, \hat{y}^4)$$

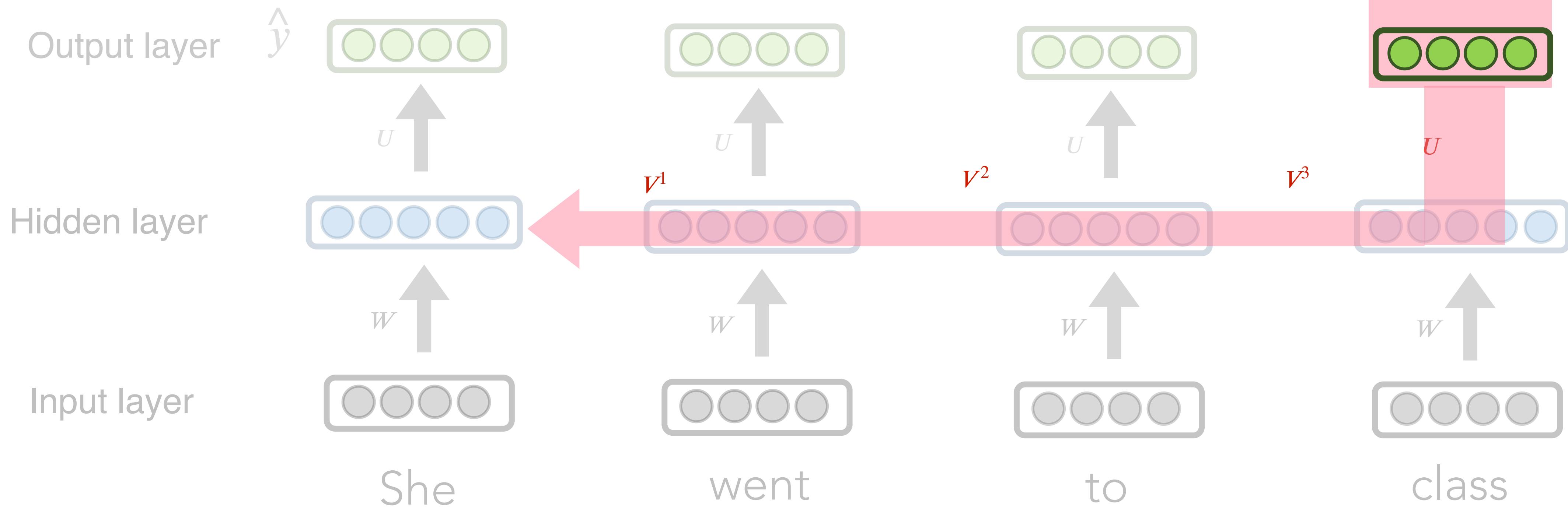


# Vanishing Gradients

$$\frac{\partial L^4}{\partial V^1}$$

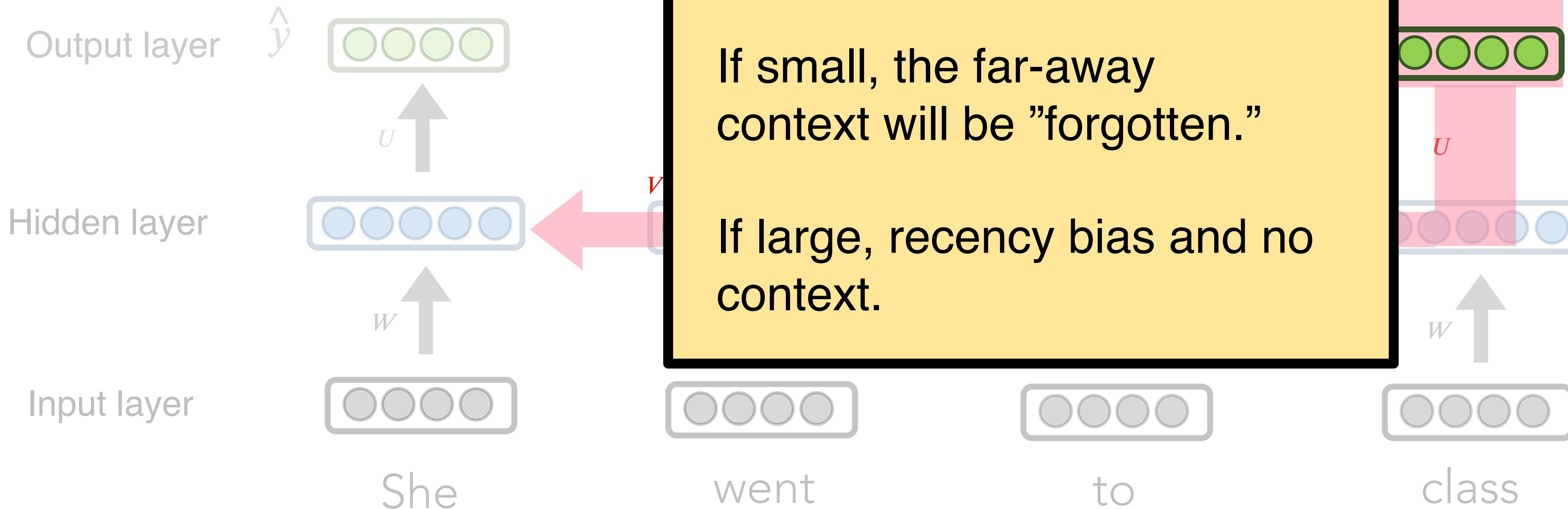
$$\frac{\partial L^4}{\partial V^1} = \frac{\partial L^4}{\partial V^3} \frac{\partial V^3}{\partial V^2} \frac{\partial V^2}{\partial V^1}$$

$$CE(y^4, \hat{y}^4)$$



# Vanishing Gradients

$$\frac{\partial L^4}{\partial V^1} = \frac{\partial L^4}{\partial V^3} \frac{\partial V^3}{\partial V^2} \frac{\partial V^2}{\partial V^1}$$





**LSTM**

# LSTM Motivation

- A type of RNN that is designed to better handle long-range dependencies
- In "vanilla" RNNs, the hidden state is perpetually being rewritten
- In addition to a traditional hidden state  $h$ , let's have a dedicated memory cell  $c$  for long-term events. More power to relay sequence information.

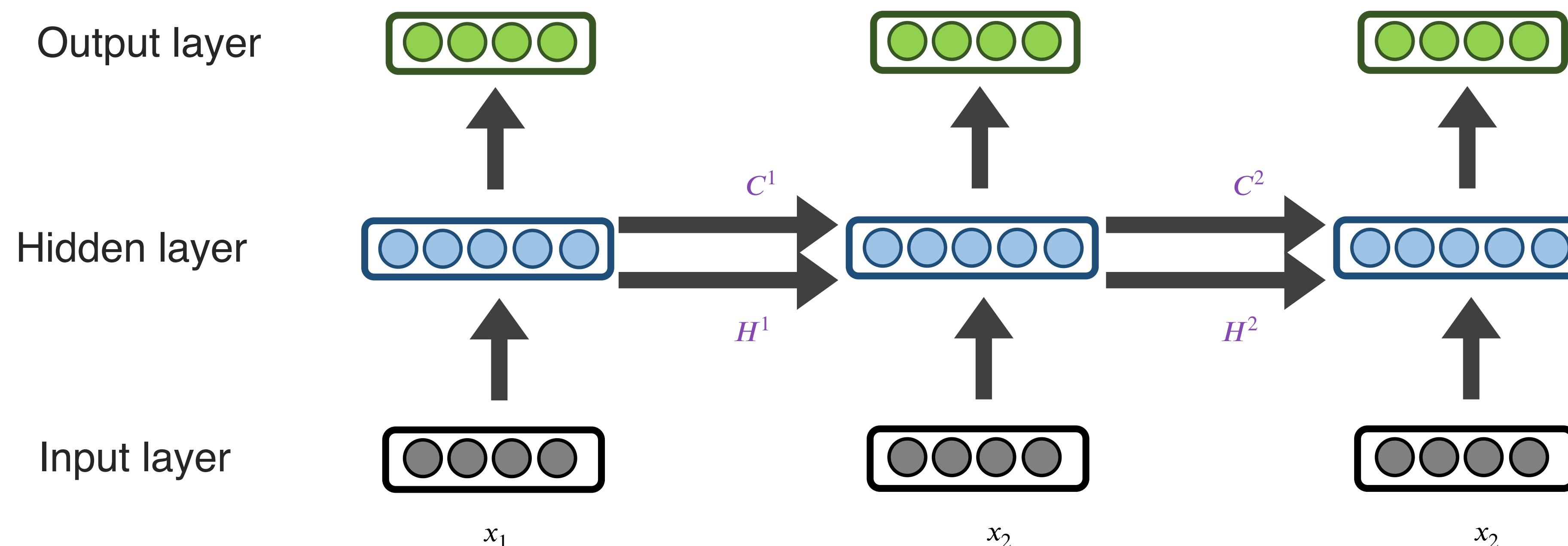
# LSTM Intuition

At each time step  $t$ , we have a hidden state  $h^t$  and cell state  $c^t$ :

- Both are vectors of length n
- cell state  $c^t$  stores long-term information
- At each time step  $t$ , the LSTM erases, writes, and reads information from the cell  $c^t$ 
  - $c^t$  never undergoes a nonlinear activation though, just  $-$  and  $+$
  - $+$  of two things does not modify the gradient; simply adds the gradients

# LSTM Intuition

$C$  and  $H$  relay long- and short-term memory to the hidden layer, respectively. Inside the hidden layer, there are many weights.



# LSTM Architecture

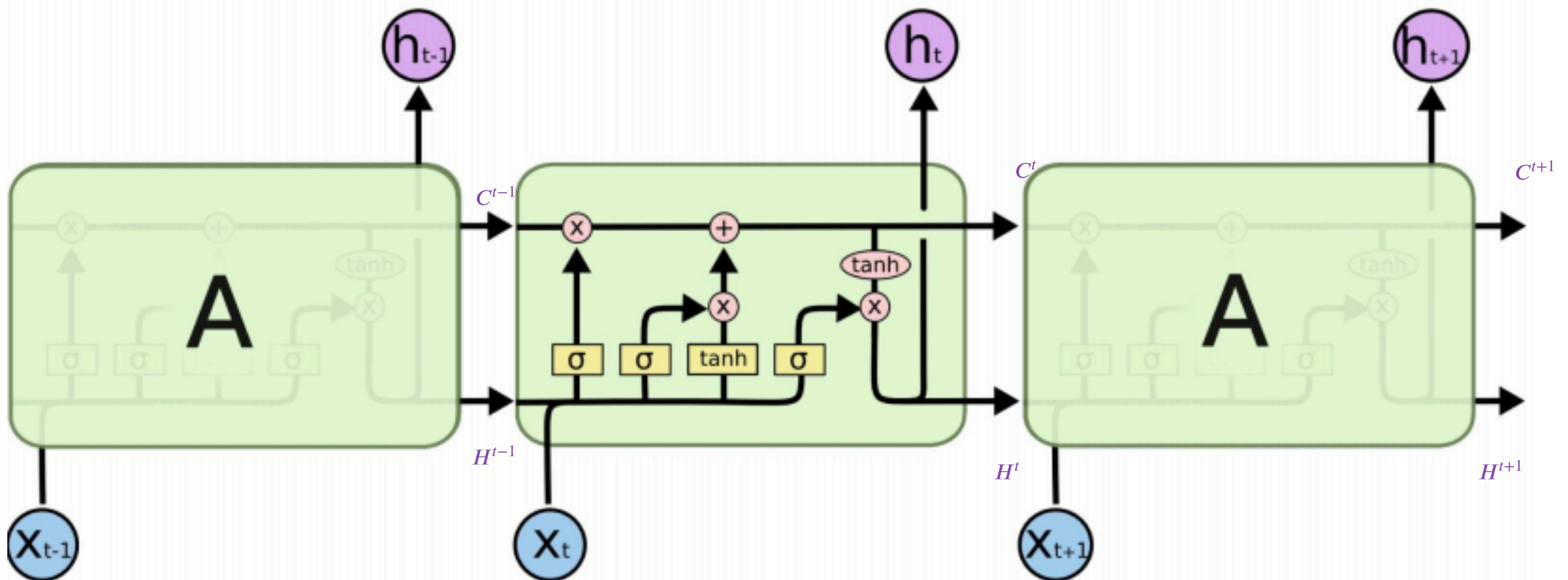
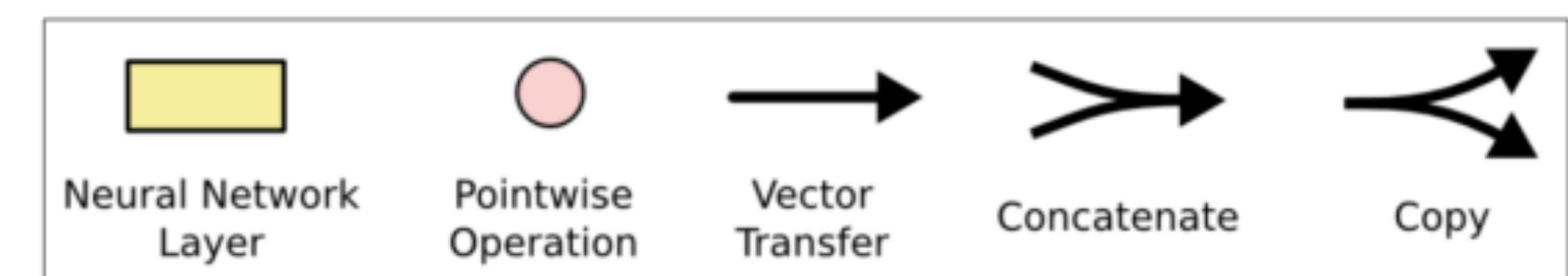


Diagram: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>



# LSTM Architecture

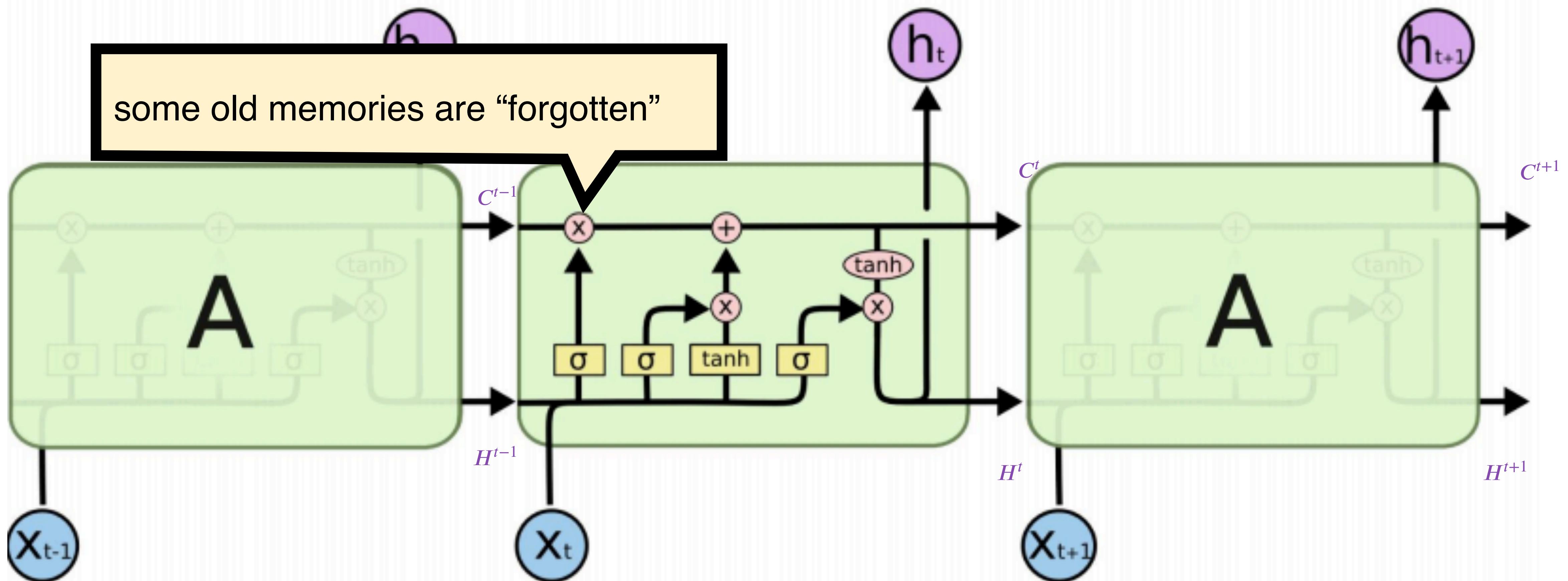
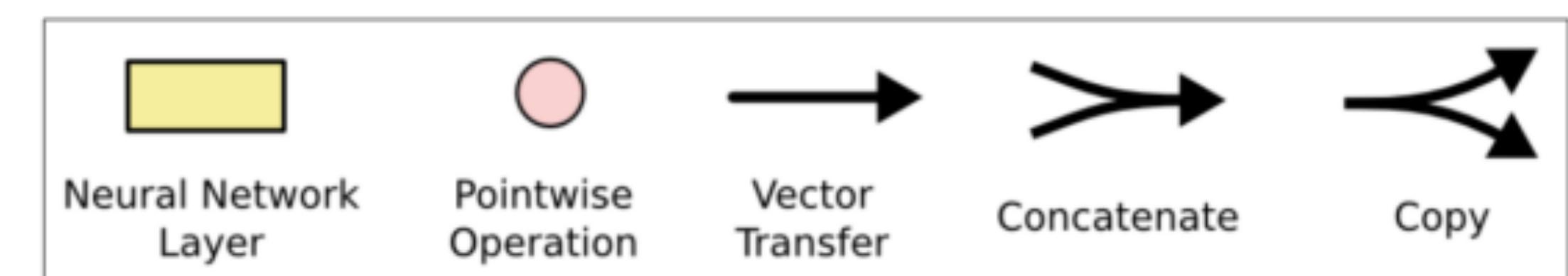


Diagram: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>



# LSTM Architecture

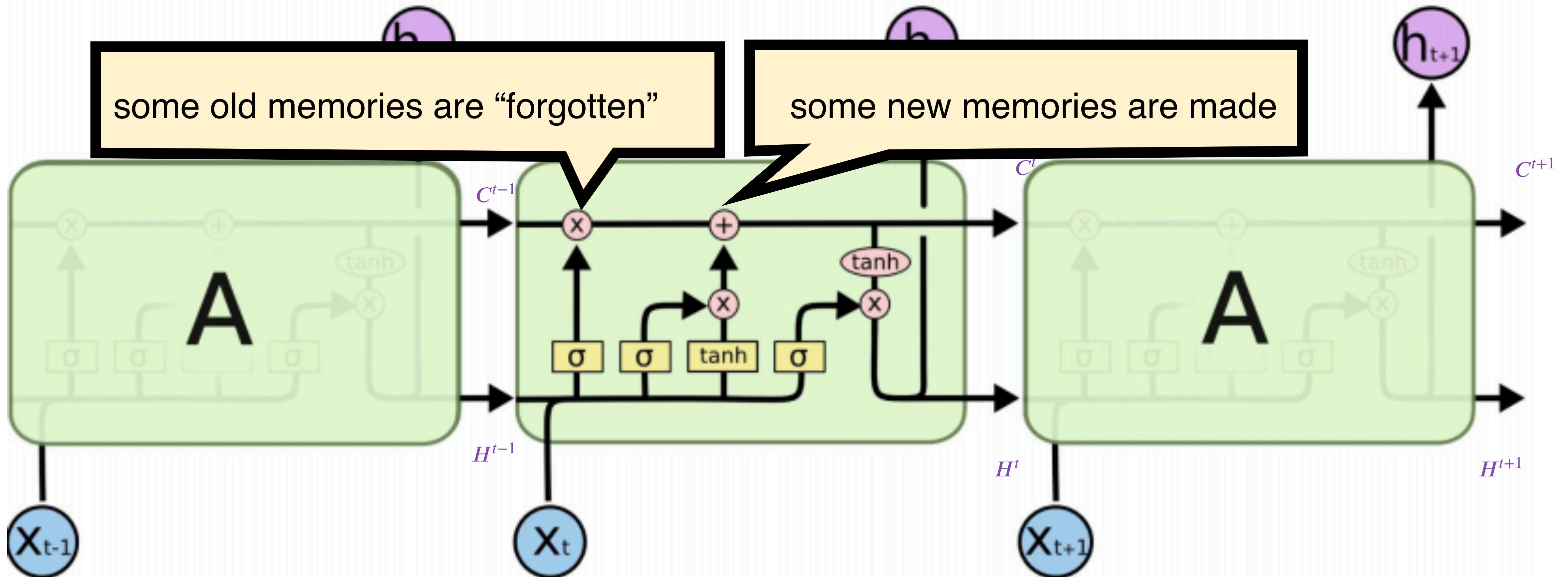
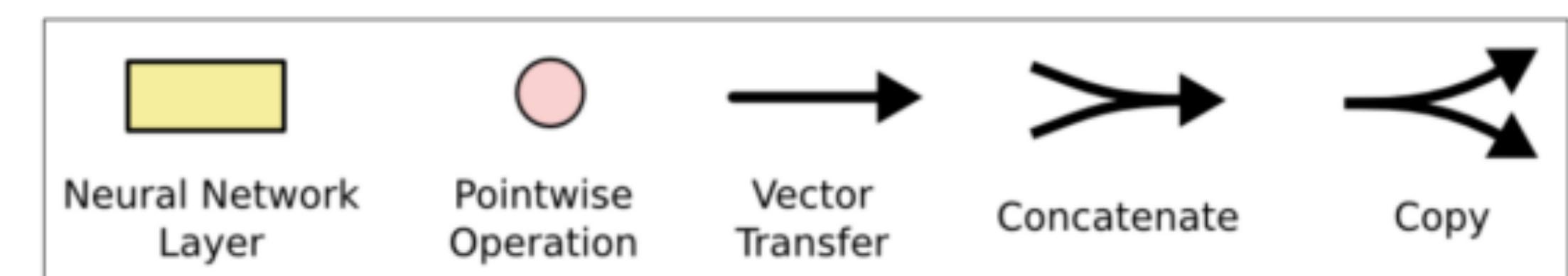


Diagram: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>



# LSTM Architecture

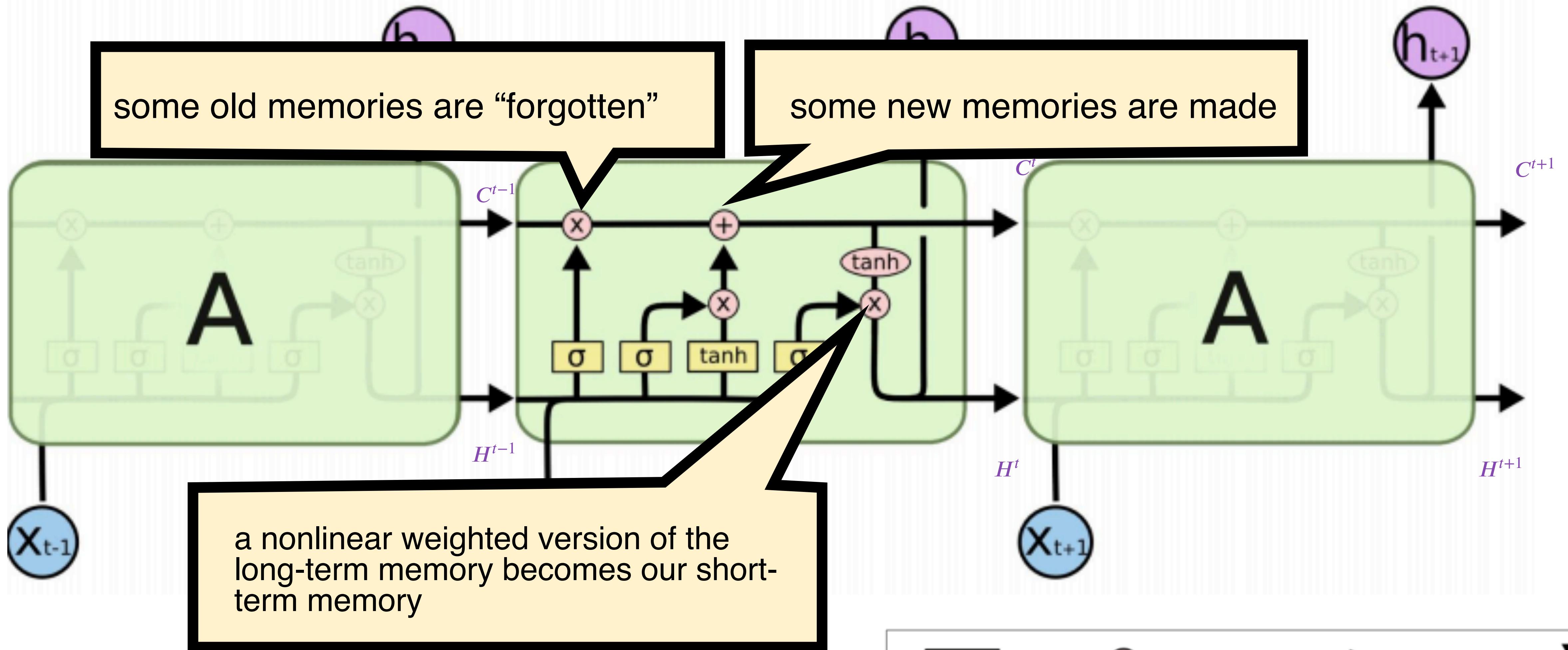
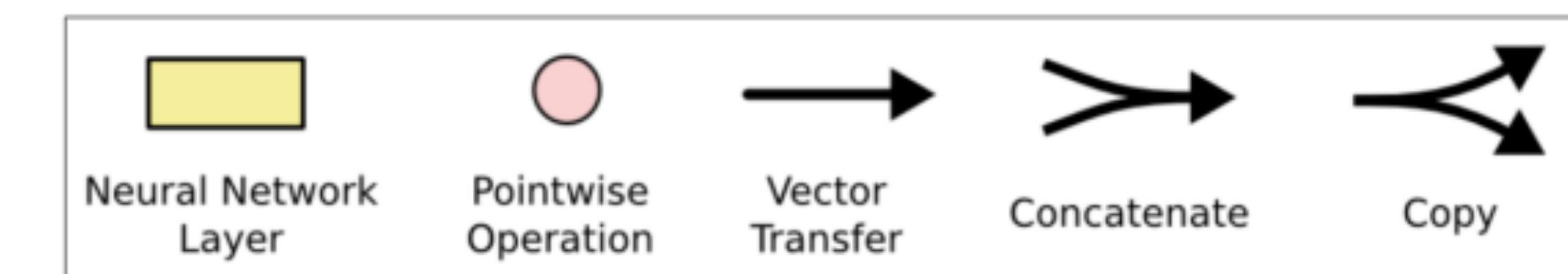


Diagram: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>



# LSTM Architecture

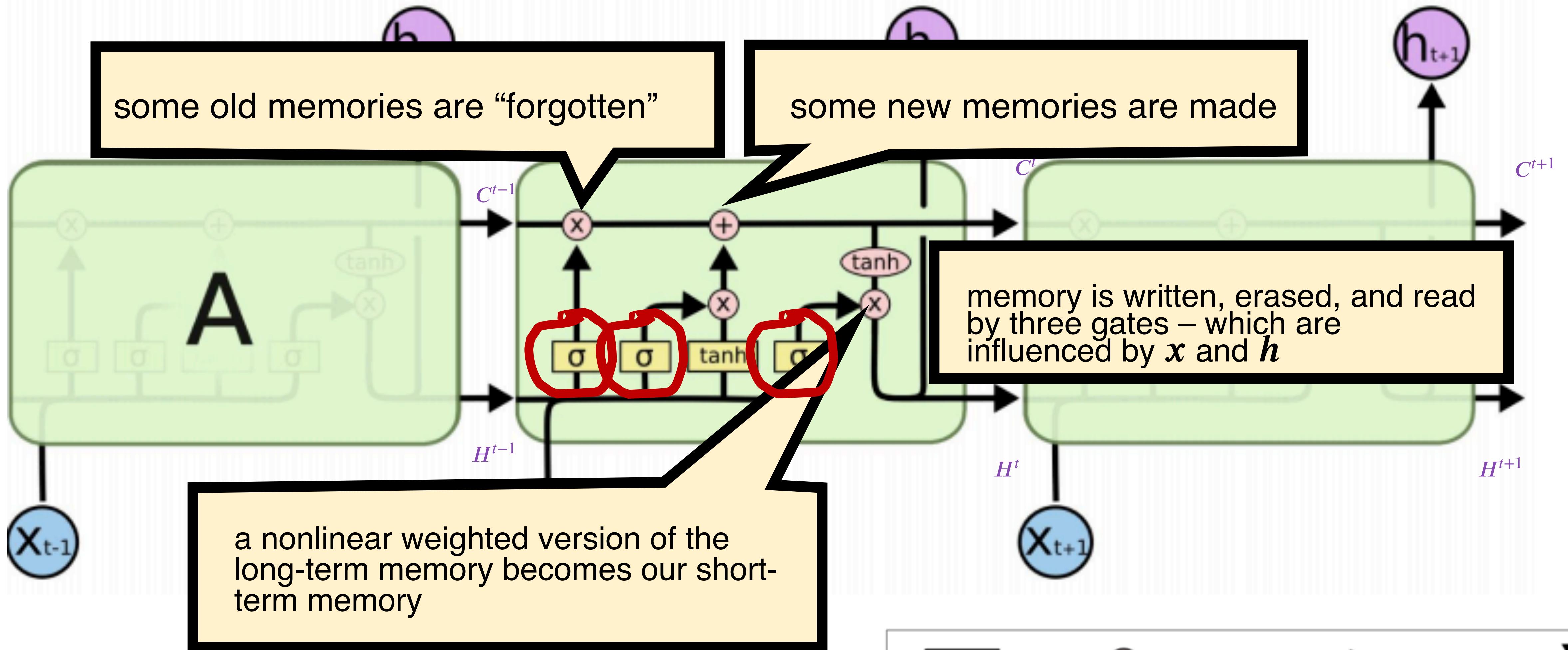
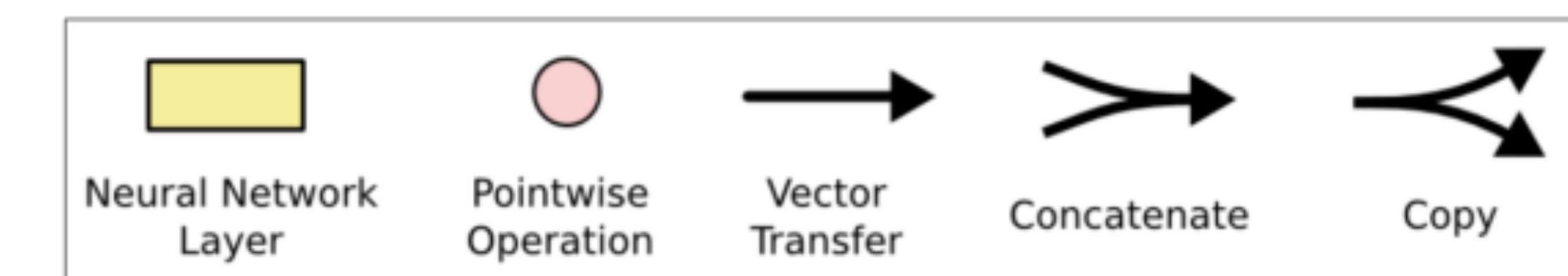
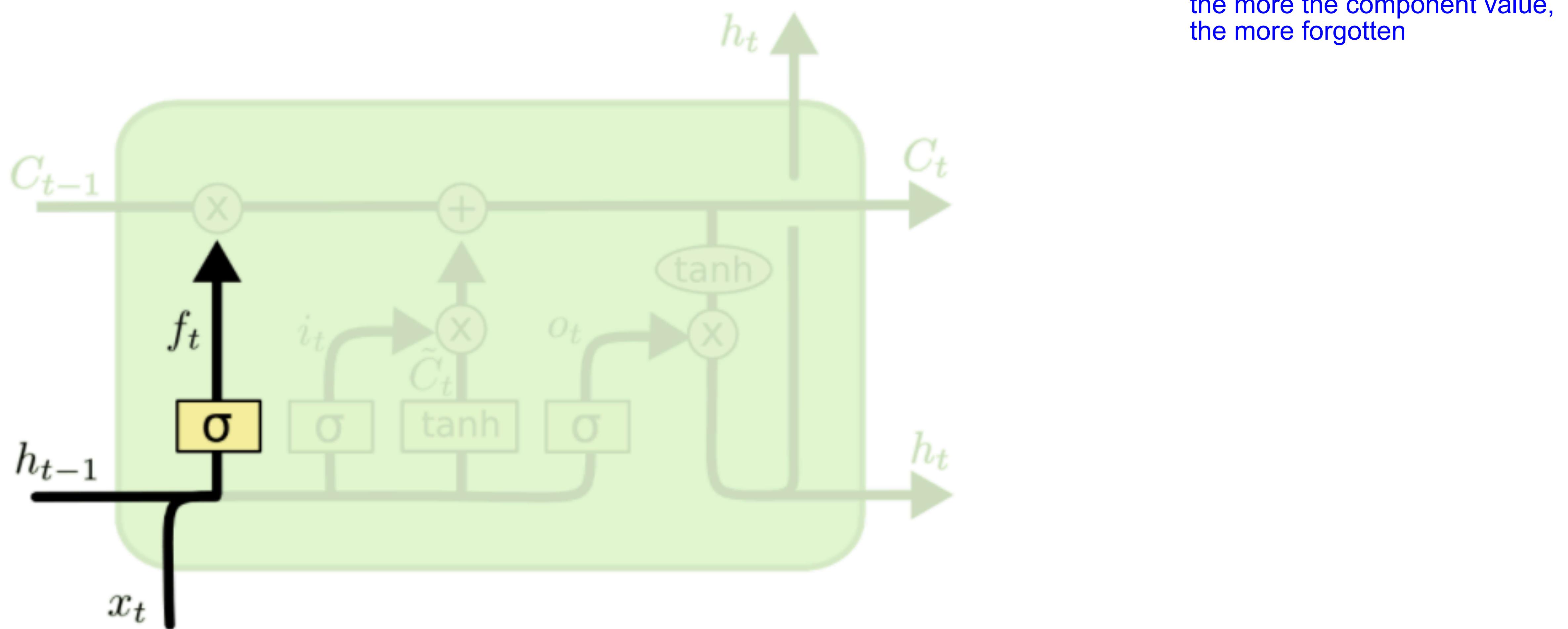


Diagram: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>



# LSTM Architecture (Forget Gate)

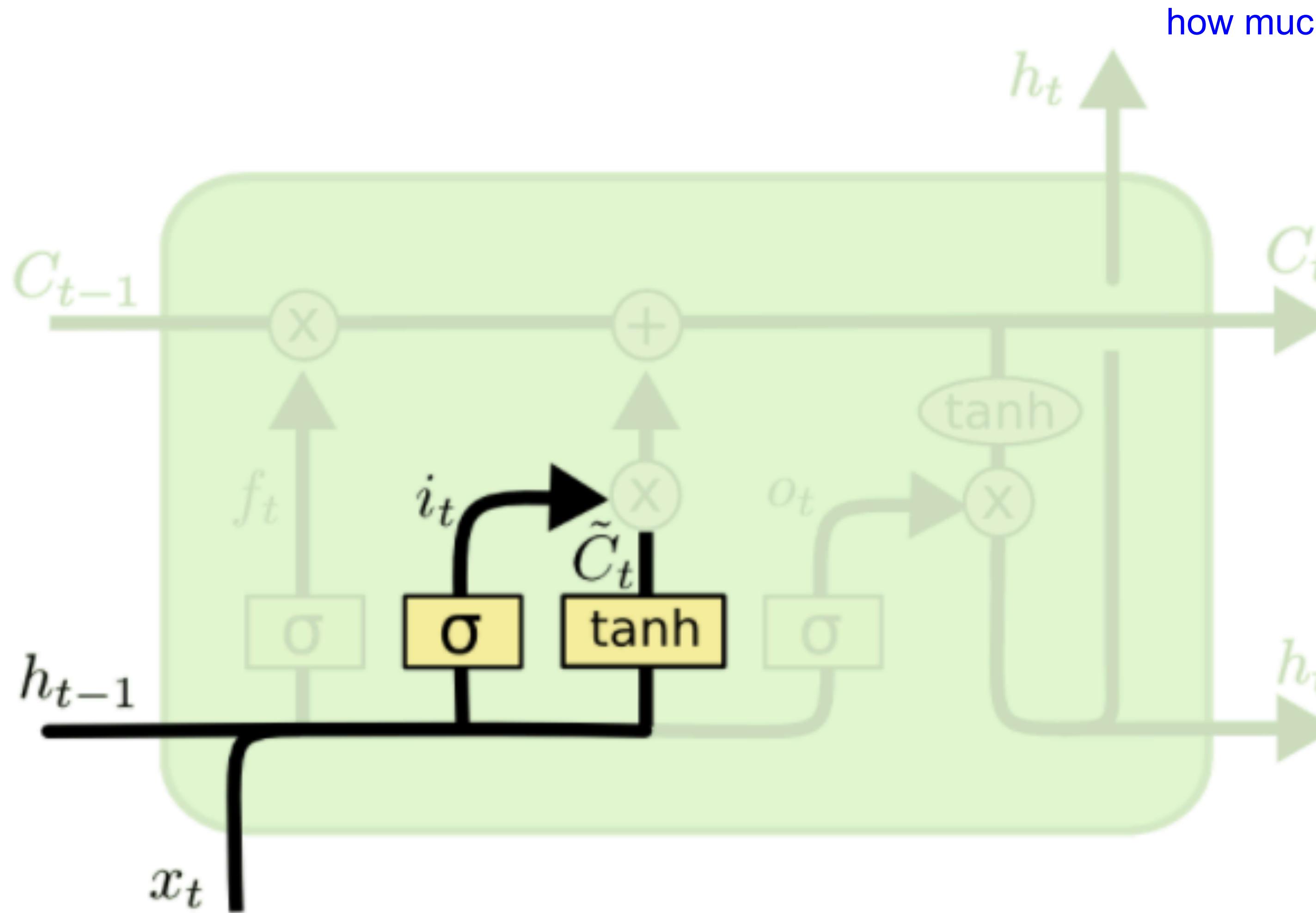
$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$



Imagine the cell state currently has values related to a previous topic.  
But now the conversation shifts. We need to forget things!

# LSTM Architecture (Input Gate)

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

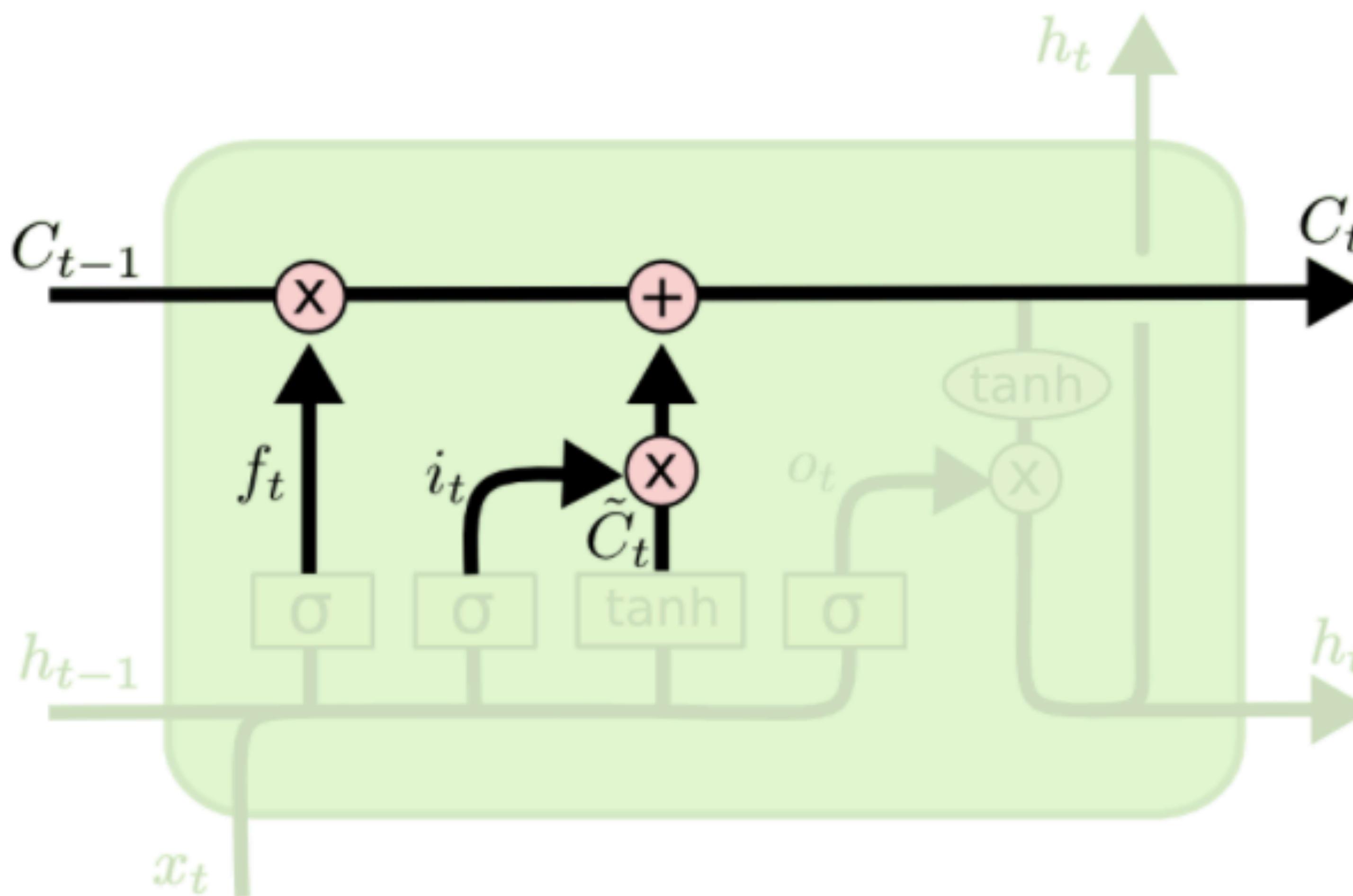


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Decides which values to update (by scaling each of the new, incoming info).

# LSTM Architecture



$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

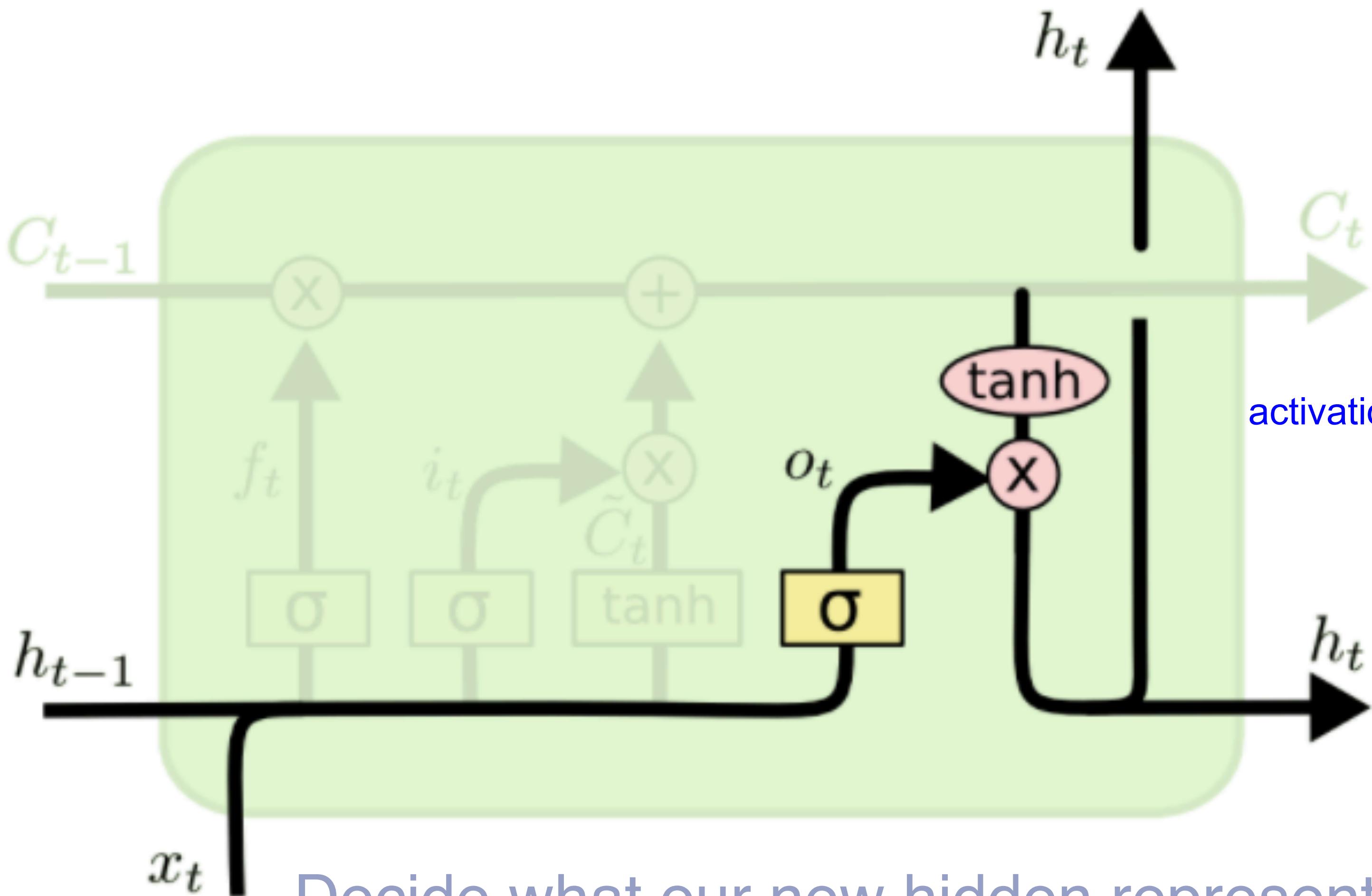
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

The cell state forgets some info, then it is simultaneously updated by us adding to it.

# LSTM Architecture (Output Gate)



$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

update

only term when long-term memories does anything

Decide what our new hidden representation will be. Based on:

- filtered version of the cell state, and
- weighted version of recurrent, hidden layer

## Total Recall?

It is still possible for LSTMs to suffer from vanishing/exploding gradients, but it's way less likely than with vanilla RNNs:

- If RNNs wish to preserve info over long contexts, it must delicately find a recurrent weight matrix  $\mathbf{W}_h$  that isn't too large or small
- LSTMs have 3 separate mechanism that adjust the flow of information (e.g., forget gate, if turned off, will preserve all info)

Cell sensitive to position in line:

```
The sole importance of the crossing of the Berezina lies in the fact  
that it plainly and indubitably proved the fallacy of all the plans for  
cutting off the enemy's retreat and the soundness of the only possible  
line of action--the one Kutuzov and the general mass of the army  
demanded--namely, simply to follow the enemy up. The French crowd fled  
at a continually increasing speed and all its energy was directed to  
reaching its goal. It fled like a wounded animal and it was impossible  
to block its path. This was shown not so much by the arrangements it  
made for crossing as by what took place at the bridges. When the bridges  
broke down, unarmed soldiers, people from Moscow and women with children  
who were with the French transport, all--carried on by vis inertiae--  
pressed forward into boats and into the ice-covered water and did not,  
surrender.
```

Cell that turns on inside quotes:

```
"You mean to imply that I have nothing to eat out of.... On the  
contrary, I can supply you with everything even if you want to give  
dinner parties," warmly replied Chichagov, who tried by every word he  
spoke to prove his own rectitude and therefore imagined Kutuzov to be  
animated by the same desire.
```

```
Kutuzov, shrugging his shoulders, replied with his subtle penetrating  
smile: "I meant merely to say what I said."
```

Cell that robustly activates inside if statements:

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,  
    siginfo_t *info)  
{  
    int sig = next_signal(pending, mask);  
    if (sig) {  
        if (current->notifier) {  
            if (sigismember(current->notifier_mask, sig)) {  
                if (! (current->notifier)(current->notifier_data)) {  
                    clear_thread_flag(TIF_SIGPENDING);  
                    return 0;  
                }  
            }  
        }  
        collect_signal(sig, pending, info);  
    }  
    return sig;  
}
```

A large portion of cells are not easily interpretable. Here is a typical example:

```
/* Unpack a filter field's string representation from user-space  
 * buffer. */  
char *audit_unpack_string(void **bufp, size_t *remain, size_t len)  
{  
    char *str;  
    if (!*bufp || (len == 0) || (len > *remain))  
        return ERR_PTR(-EINVAL);  
    /* of the currently implemented string fields, PATH_MAX  
     * defines the longest valid length.  
     */
```

The cell learns an  
operative time to  
“turn on”.

Cell that turns on inside comments and quotes:

```
/* Duplicate LSM field information. The lsm_rule is opaque, so
 * re-initialized. */
static inline int audit_dupe_lsm_field(struct audit_field *df,
    struct audit_field *sf)
{
    int ret = 0;
    char *lsm_str;
    /* our own copy of lsm_str */
    lsm_str = kstrdup(sf->lsm_str, GFP_KERNEL);
    if (unlikely(!lsm_str))
        return -ENOMEM;
    df->lsm_str = lsm_str;
    /* our own (refreshed) copy of lsm_rule */
    ret = security_audit_rule_init(df->type, df->op, df->lsm_str,
        (void **) &df->lsm_rule);
    /* Keep currently invalid fields around in case they
     * become valid after a policy reload. */
    if (ret == -EINVAL) {
        pr_warn("audit rule for LSM \\'%s\\' is invalid\n",
            df->lsm_str);
        ret = 0;
    }
    return ret;
}
```

The cell learns an operative time to “turn on”.

Cell that is sensitive to the depth of an expression:

```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
    int i;
    if (classes[class]) {
        for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
            if (mask[i] & classes[class][i])
                return 0;
    }
    return 1;
}
```

# LSTM Summary

## Strengths

- Almost always outperforms RNNs
- Captures long-range dependencies very well

## Issues?

- More weights to learn, thus:
- Higher training data demand
- Can still suffer from vanishing/exploding gradients

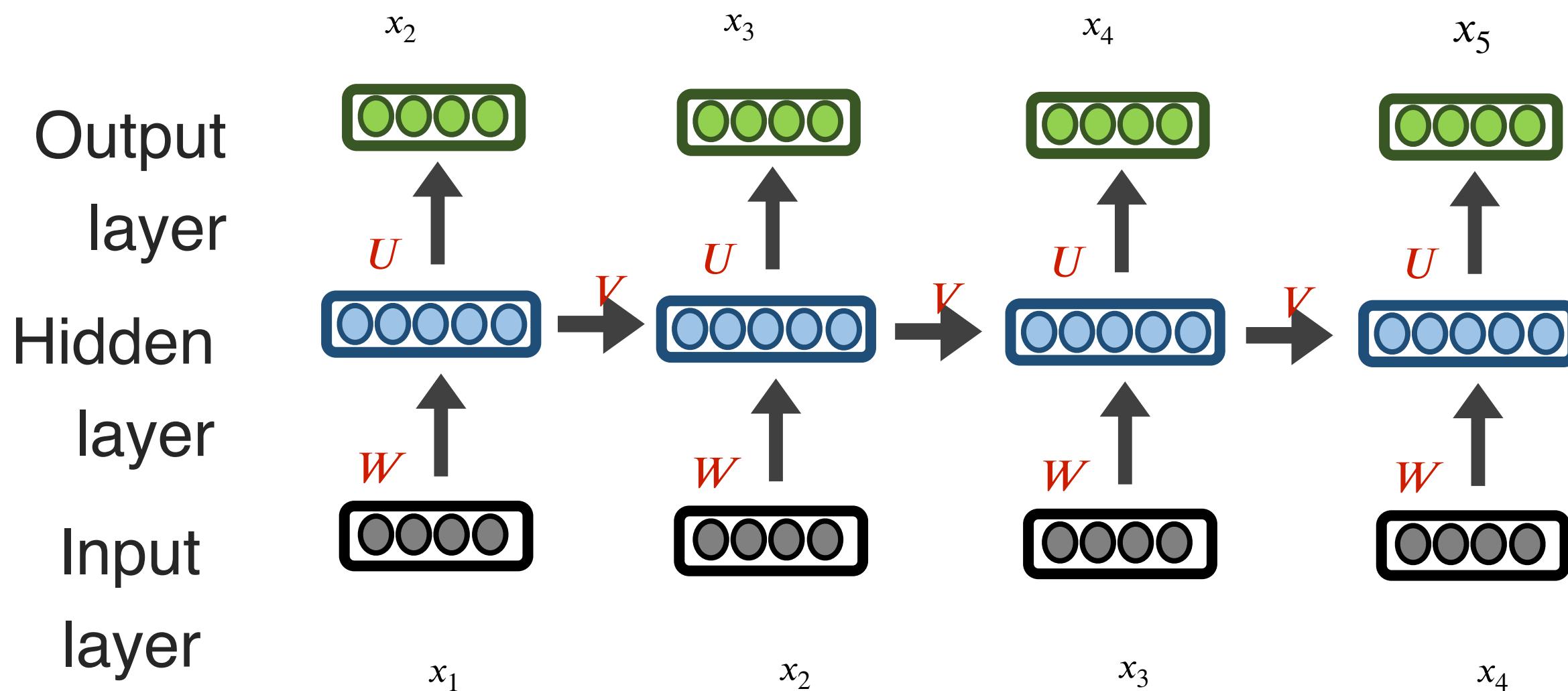
# LSTM Applications

If your goal isn't to predict the next item in a sequence, and you rather do some other [classification or regression task](#) using the sequence, then you can:

- Train an aforementioned model (e.g., LSTM) as a language model
- Use the hidden layers that correspond to each item in your sequence

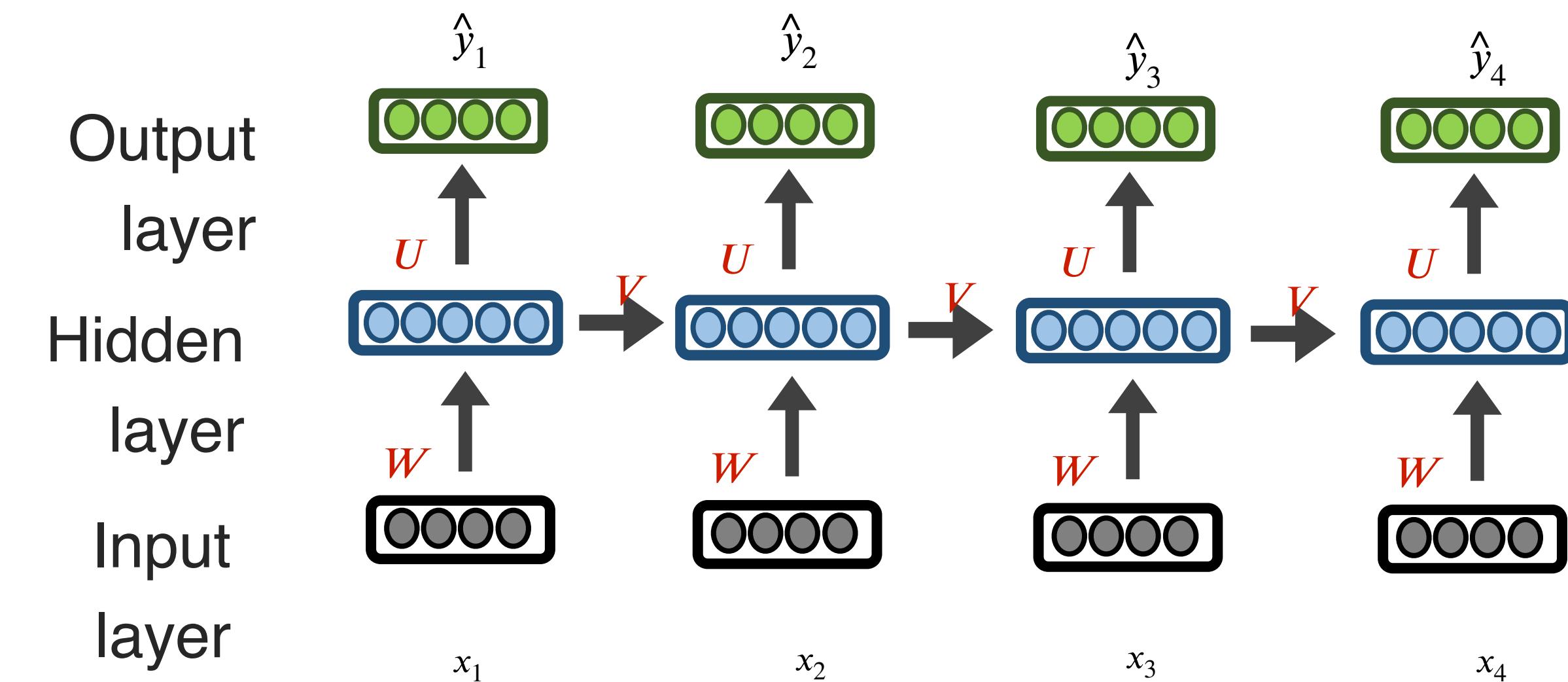
# LSTM Applications

## Language Modeling



Auto-regressive

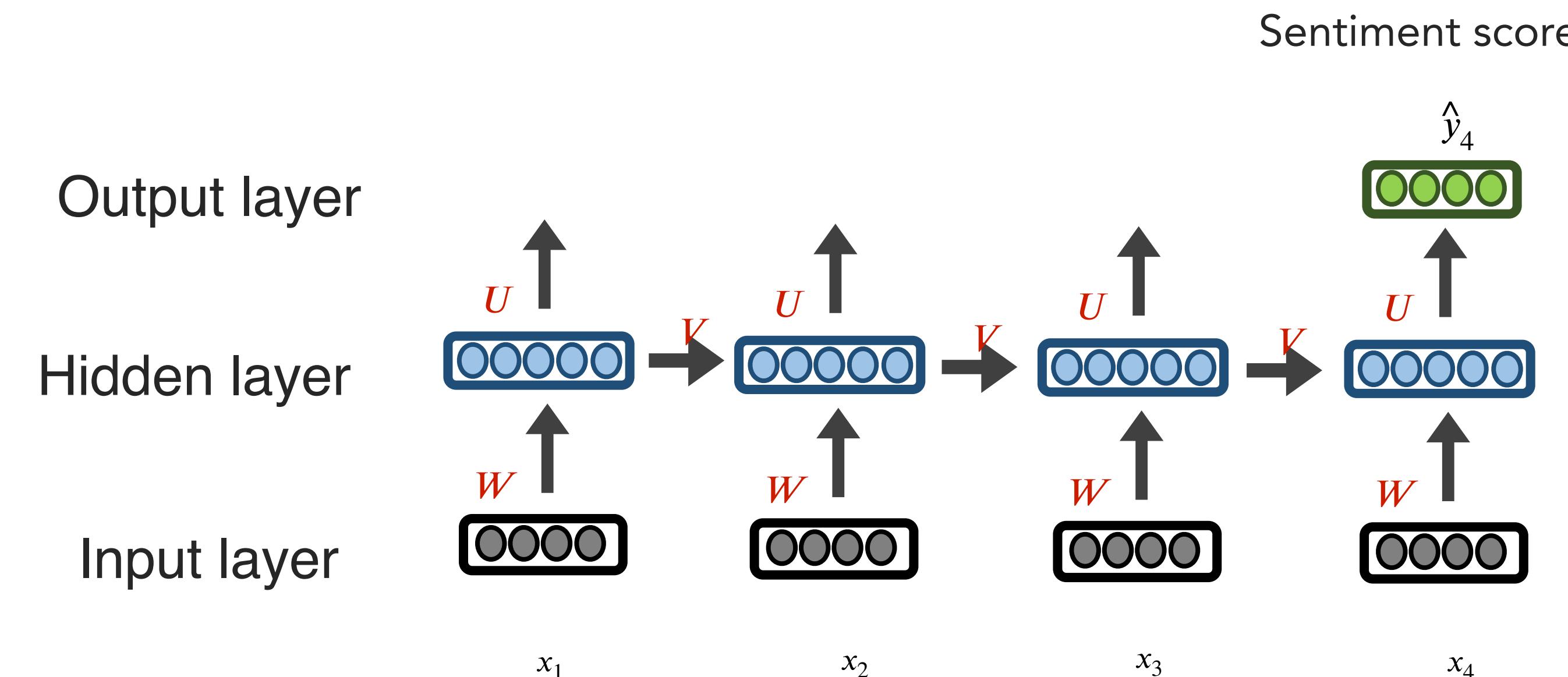
## 1-to-1 tagging/classification



Non-Auto-regressive

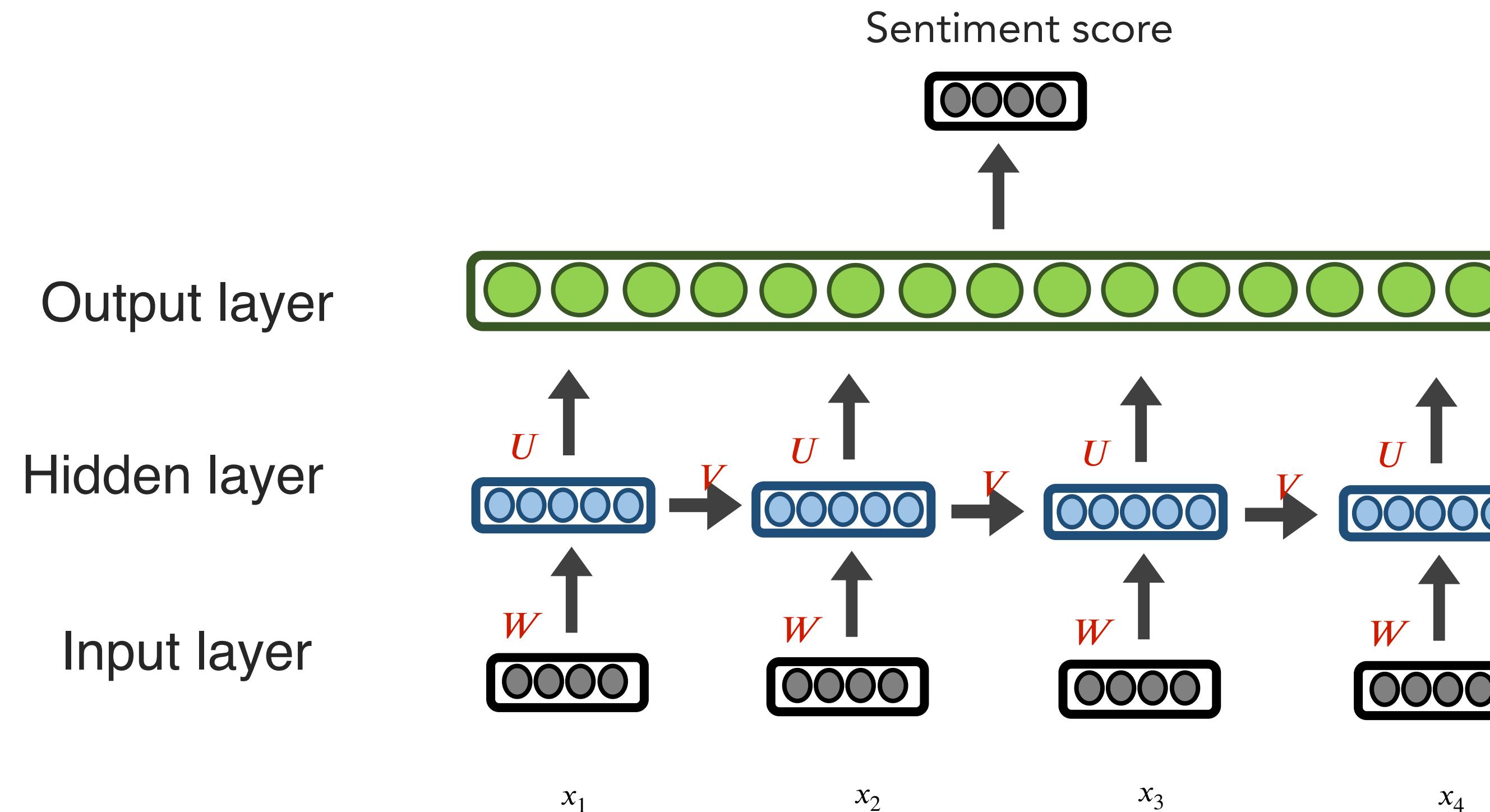
# LSTM Applications

## Many-to-1 classification



# LSTM Applications

## Many-to-1 classification





# **Bi-LSTM**

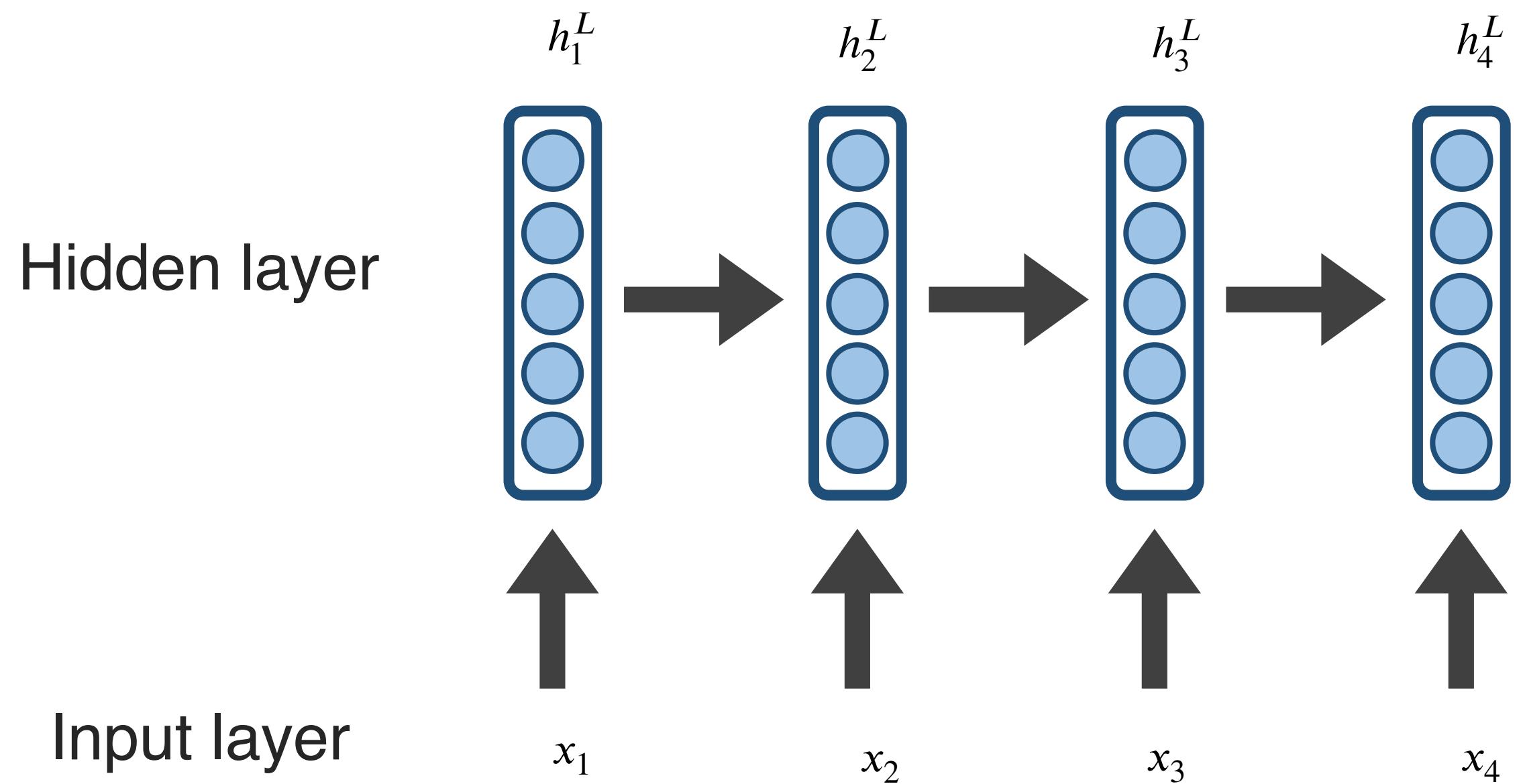
## Reading Bidirectionally

RNNs/LSTMs use the [left-to-right](#) context and sequentially process data.

If you have full access to the data at testing time, why not make use of the flow of information from [right-to-left](#), also?

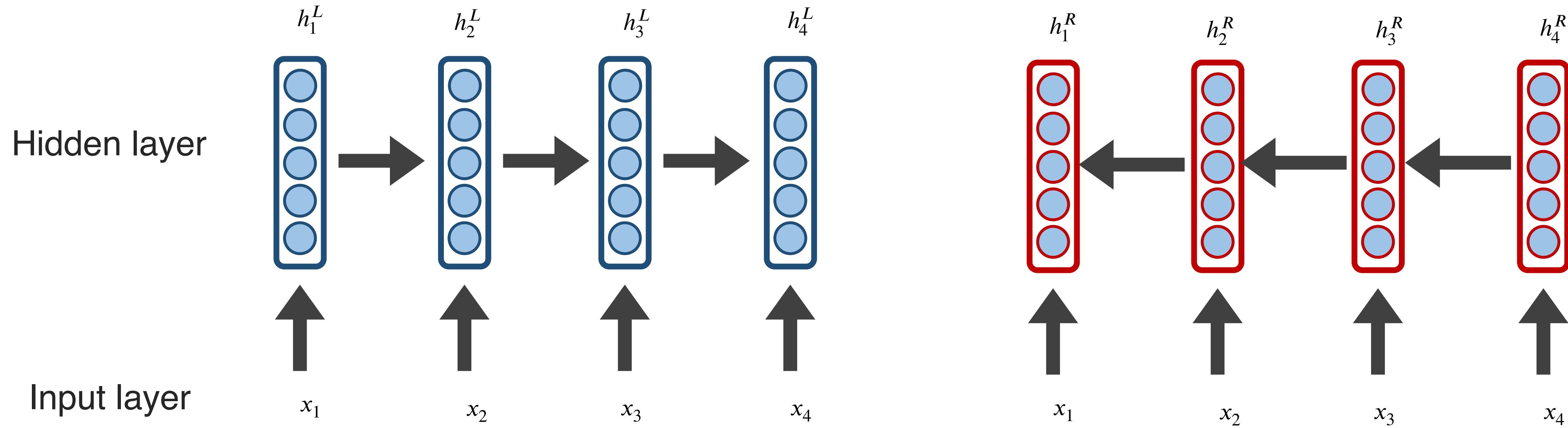
# Reading Bidirectionally

For brevity, let's use the follow schematic to represent an RNN



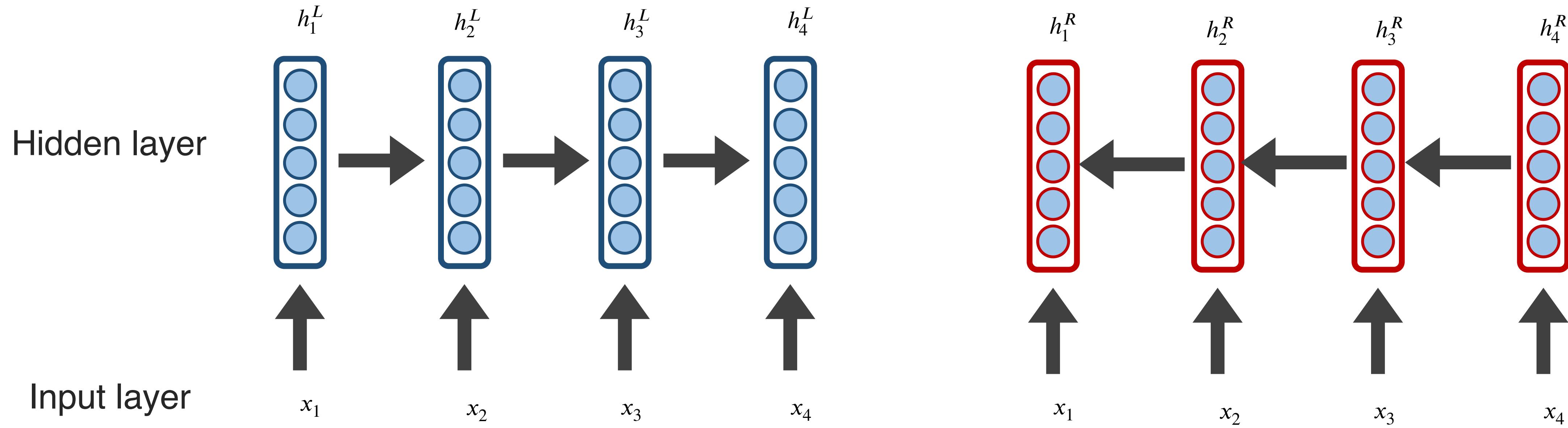
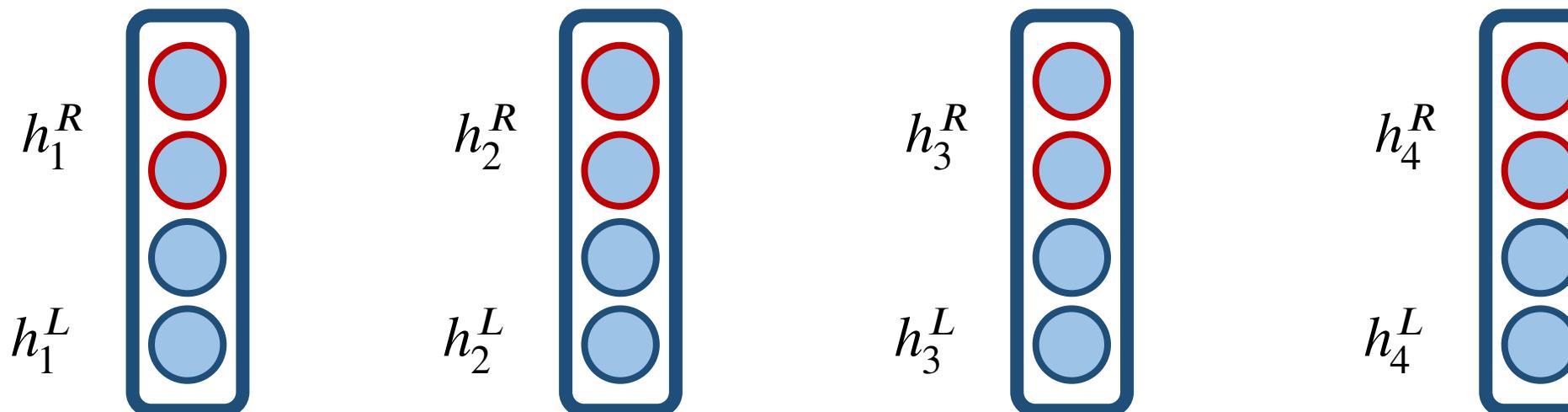
# Reading Bidirectionally

For brevity, let's use the follow schematic to represent an RNN

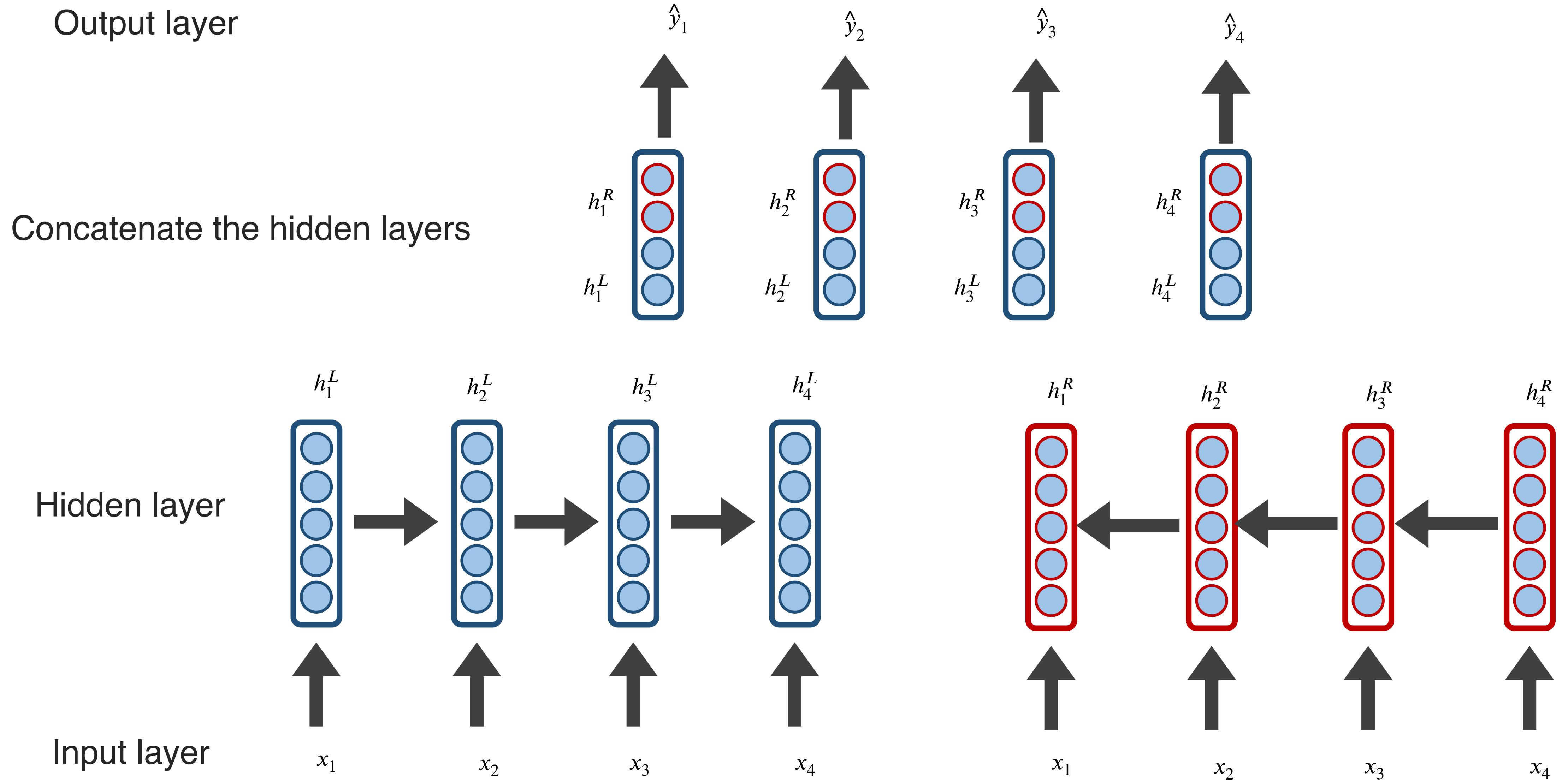


# Reading Bidirectionally

Concatenate the hidden layers



# Reading Bidirectionally



# Bi-LSTM Summary

## Strengths

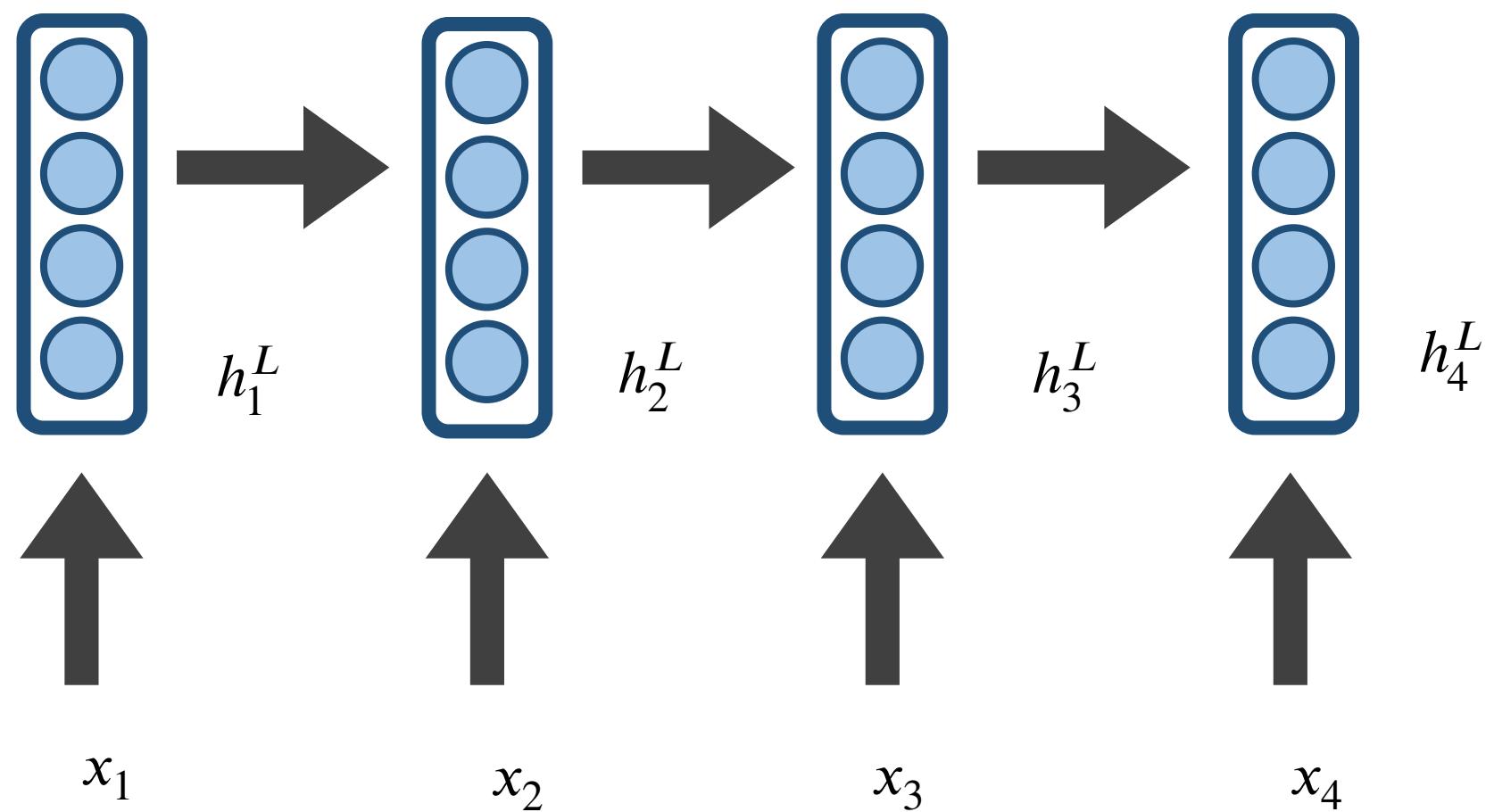
- Usually performs at least as well as uni-directional RNNs/LSTMs

## Issues?

- Slower to train
- Only possible if access to full data is allowed

# Stacked LSTMs

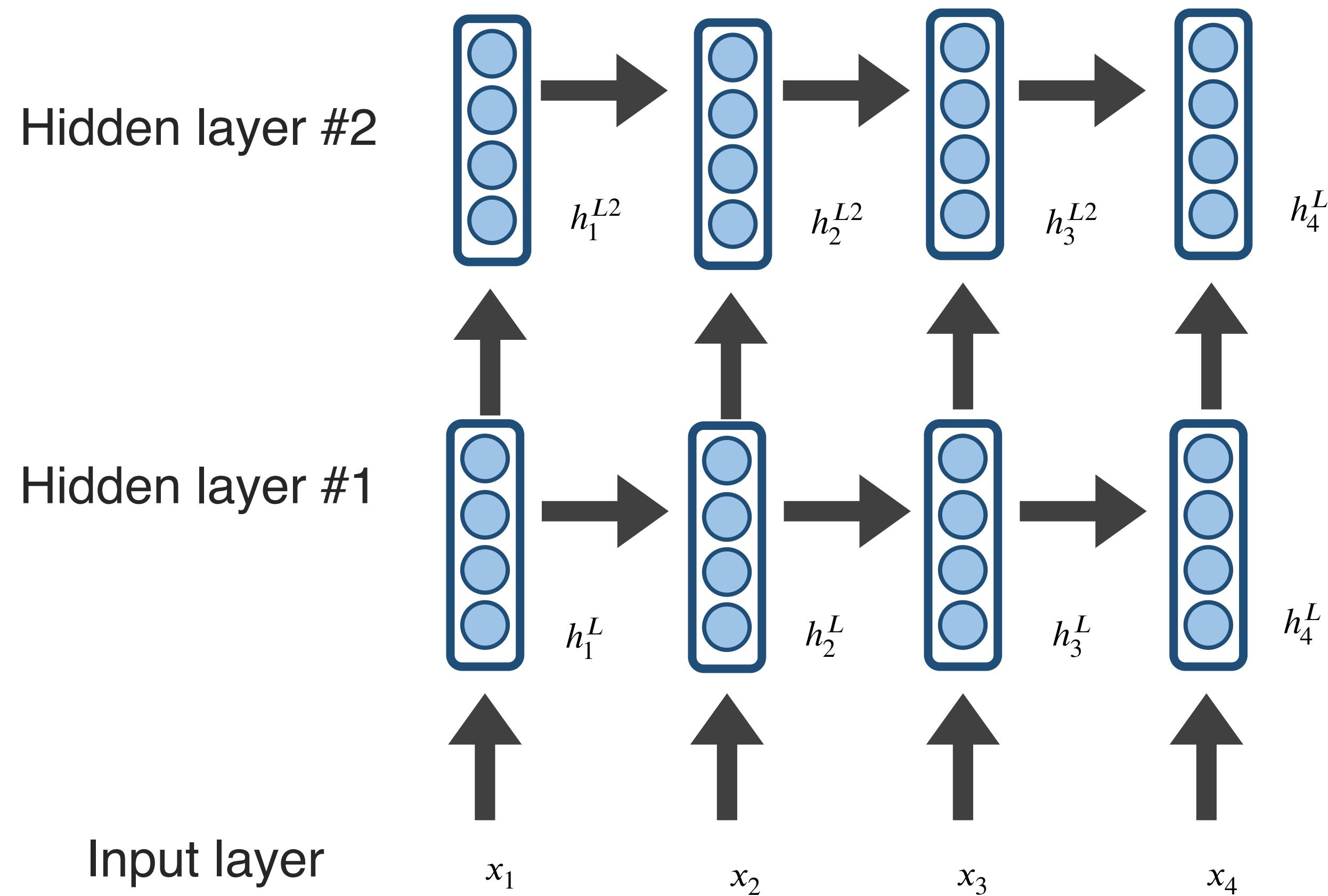
Hidden layer #1



Hidden layers provide an abstraction (hold “meaning”).

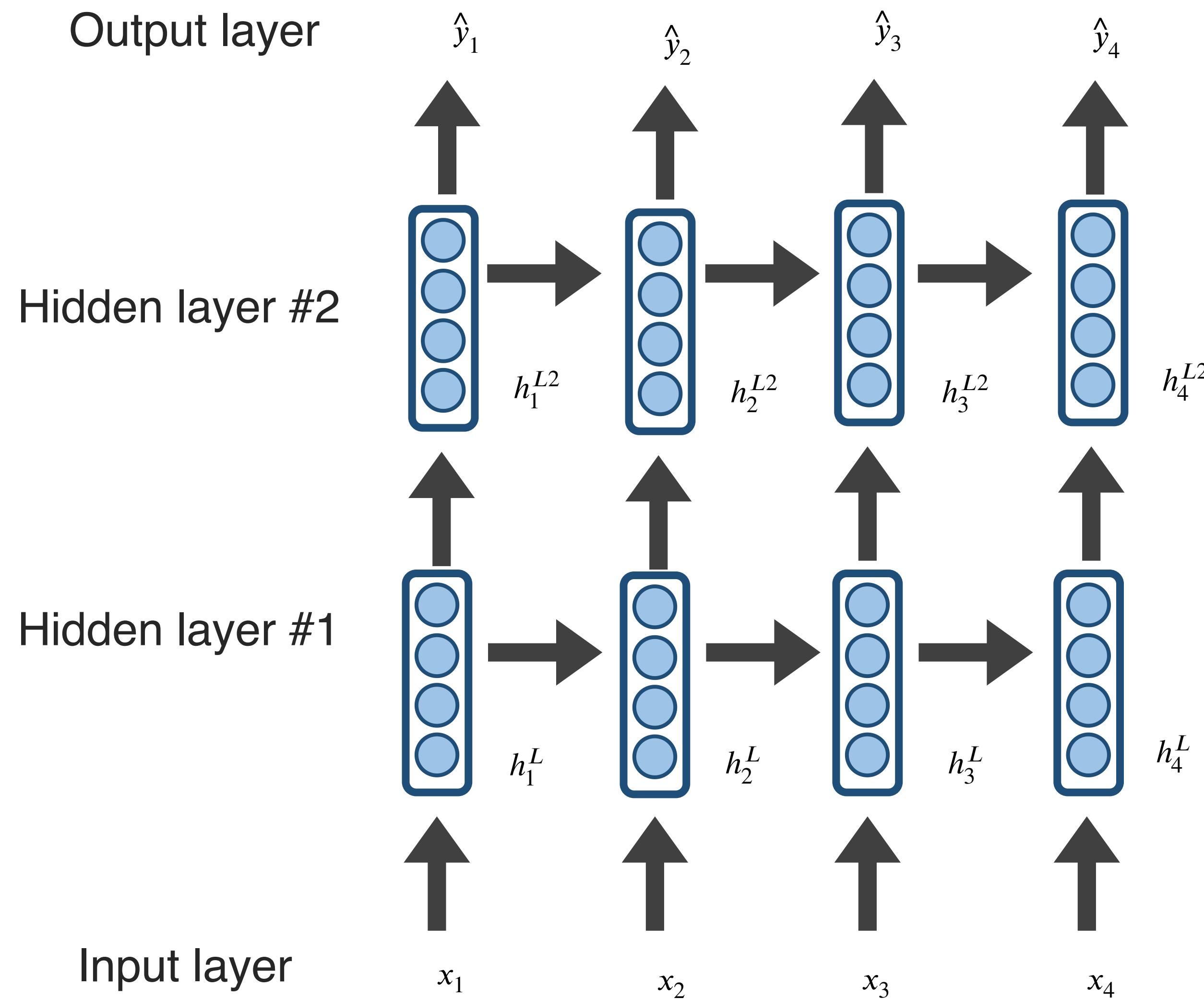
Stacking hidden layers provides increased abstractions.

# Stacked LSTMs



Hidden layers provide an abstraction (holds “meaning”).  
Stacking hidden layers provides increased abstractions.

# Stacked LSTMs



Hidden layers provide an abstraction (holds “meaning”).

Stacking hidden layers provides increased abstractions.

# ELMo

## General Idea:

- Goal is to obtain highly rich embeddings for each word (unique type)
- Use both directions of context (bi-directional), with increasing abstractions (stacked)
- Linearly combine all abstract representations (hidden layers) and optimize w.r.t. a particular task (e.g., sentiment classification)

# ELMo

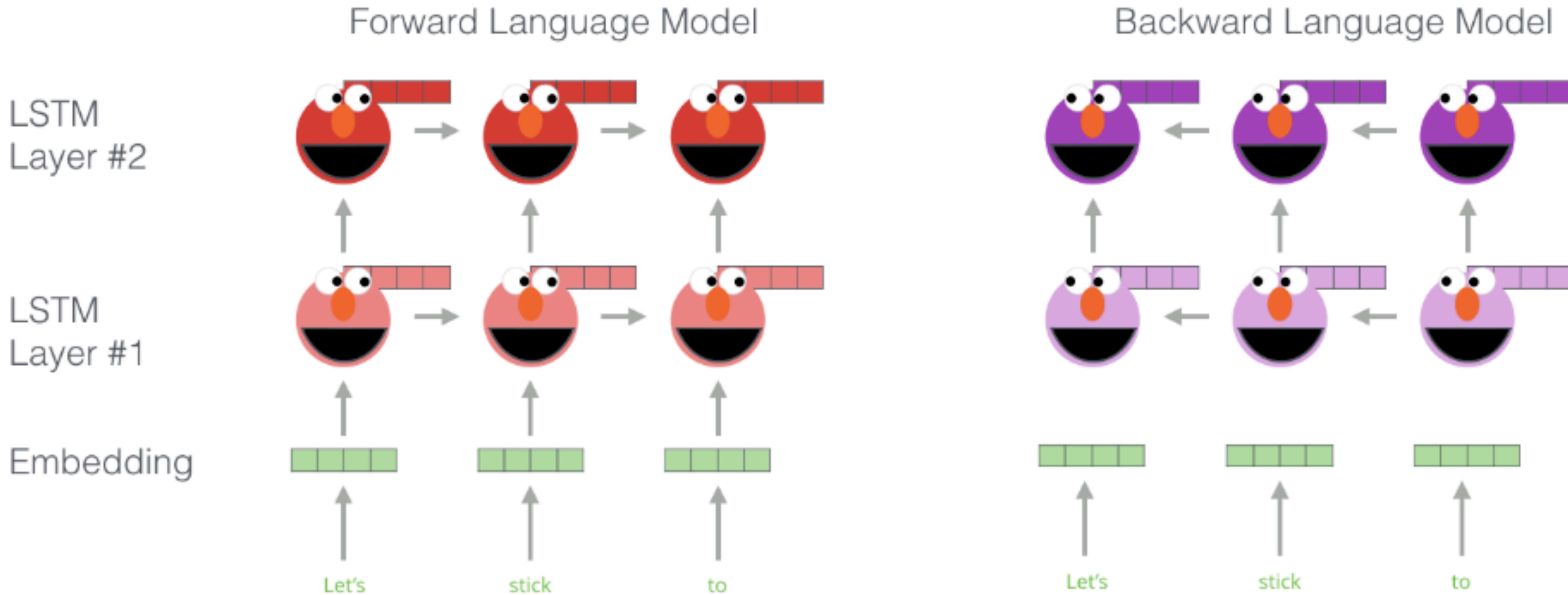


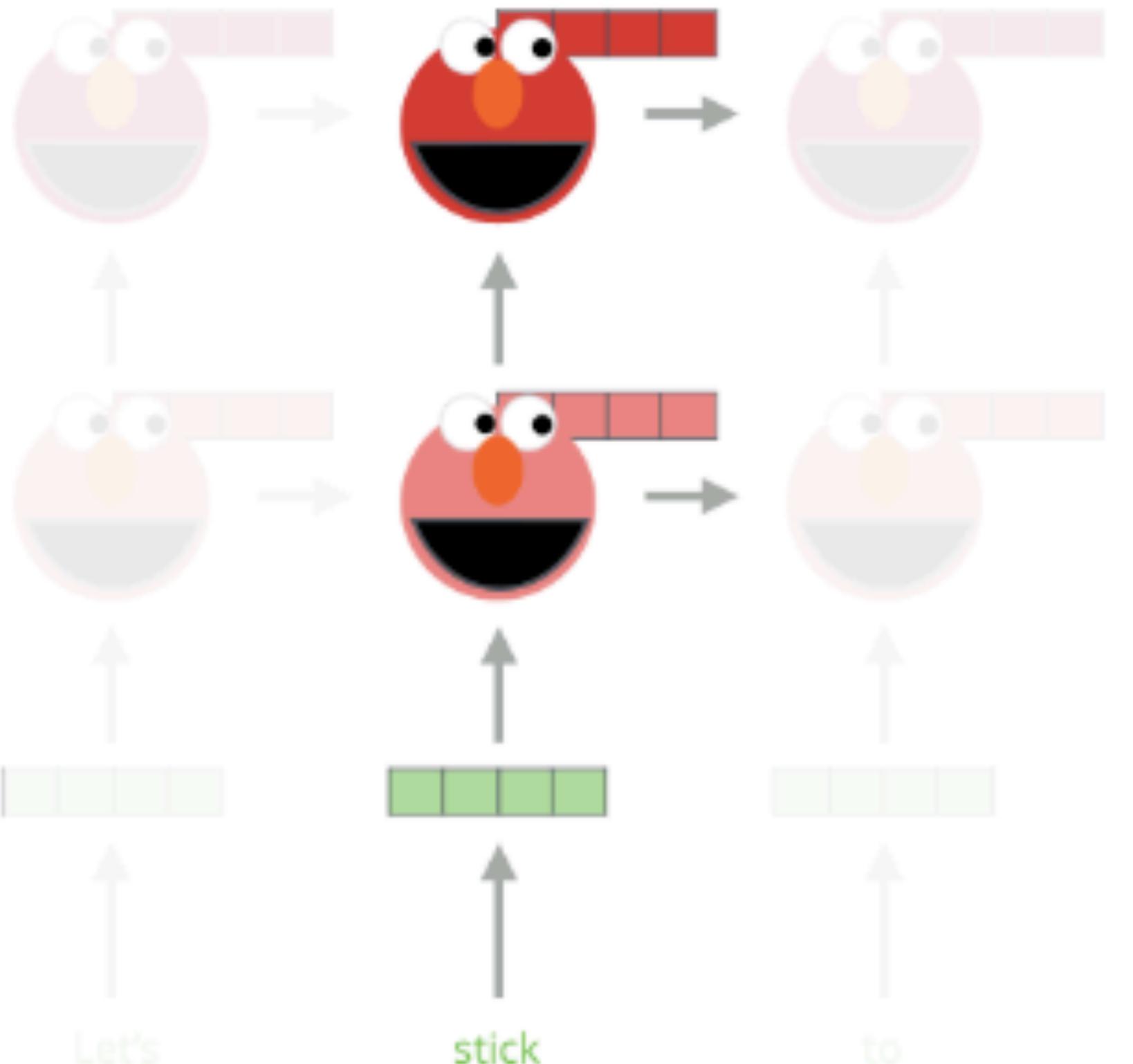
Illustration: <http://jalammar.github.io/illustrated-bert/>

## Embedding of “stick” in “Let’s stick to” - Step #2

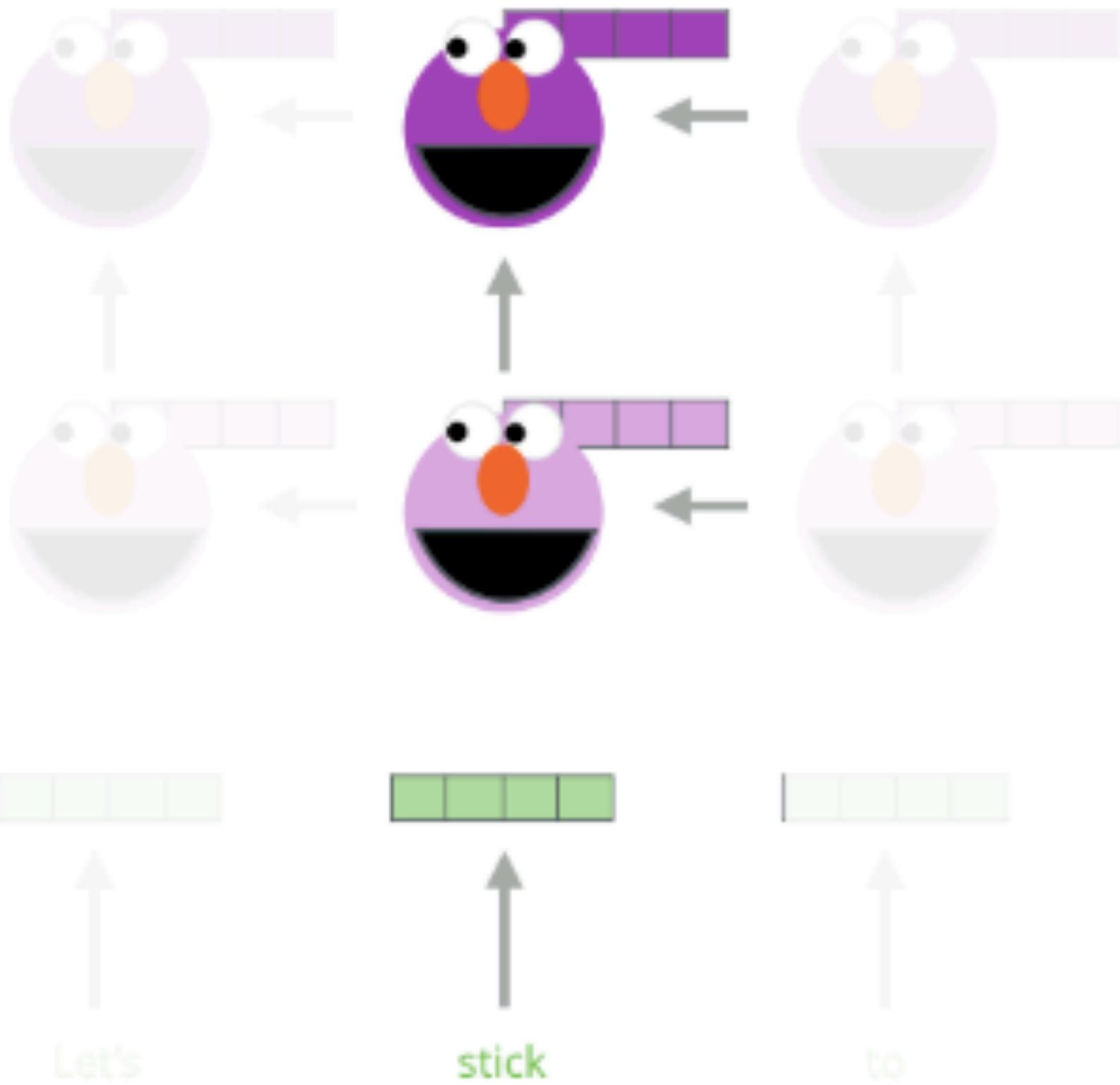
1- Concatenate hidden layers



Forward Language Model



Backward Language Model



2- Multiply each vector by a weight based on the task



3- Sum the (now weighted) vectors



ELMo embedding of “stick” for this task in this context

Illustration: <http://jalammar.github.io/illustrated-bert/>

# ELMo

TASK	PREVIOUS SOTA	OUR BASELINE	ELMo + BASELINE	INCREASE (ABSOLUTE/ RELATIVE)
SQuAD	Liu et al. (2017)	84.4	81.1	85.8
SNLI	Chen et al. (2017)	88.6	88.0	88.7 ± 0.17
SRL	He et al. (2017)	81.7	81.4	84.6
Coref	Lee et al. (2017)	67.2	67.2	70.4
NER	Peters et al. (2017)	91.93 ± 0.19	90.15	92.22 ± 0.10
SST-5	McCann et al. (2017)	53.7	51.4	54.7 ± 0.5

# ELMo

- ELMo yielded very good contextualized embeddings, which re-defined SOTA when applied to many NLP tasks.
- Main ELMo takeaway: given enough training data, having tons of explicit connections between your vectors is useful (the system can determine how to best use context)

# ELMo

- Distributed Representations can be:
  - Type-based (“word embeddings”)
  - Token-based (“contextualized representations/embeddings”)
- Type-based models include Bengio’s 2003 and word2vec 2013
- Token-based models include RNNs/LSTMs, which:
  - demonstrated profound results from 2015 onward.
  - can be used for essentially any NLP task.



# Attention

## Attention Motivation

What if the decoder (component that produces outputs), at each step, pays **attention** to a distribution of all of the encoder's (component that receives inputs) hidden states?

## Attention Motivation

What if the decoder (component that produces outputs), at each step, pays **attention** to a distribution of all of the encoder's (component that receives inputs) hidden states?

**Intuition:** when we (humans) translate a sentence, we don't just consume the original sentence, reflect on the meaning of the last word, then regurgitate in a new language; we **continuously think back at the original sentence** while focusing on **different parts**.

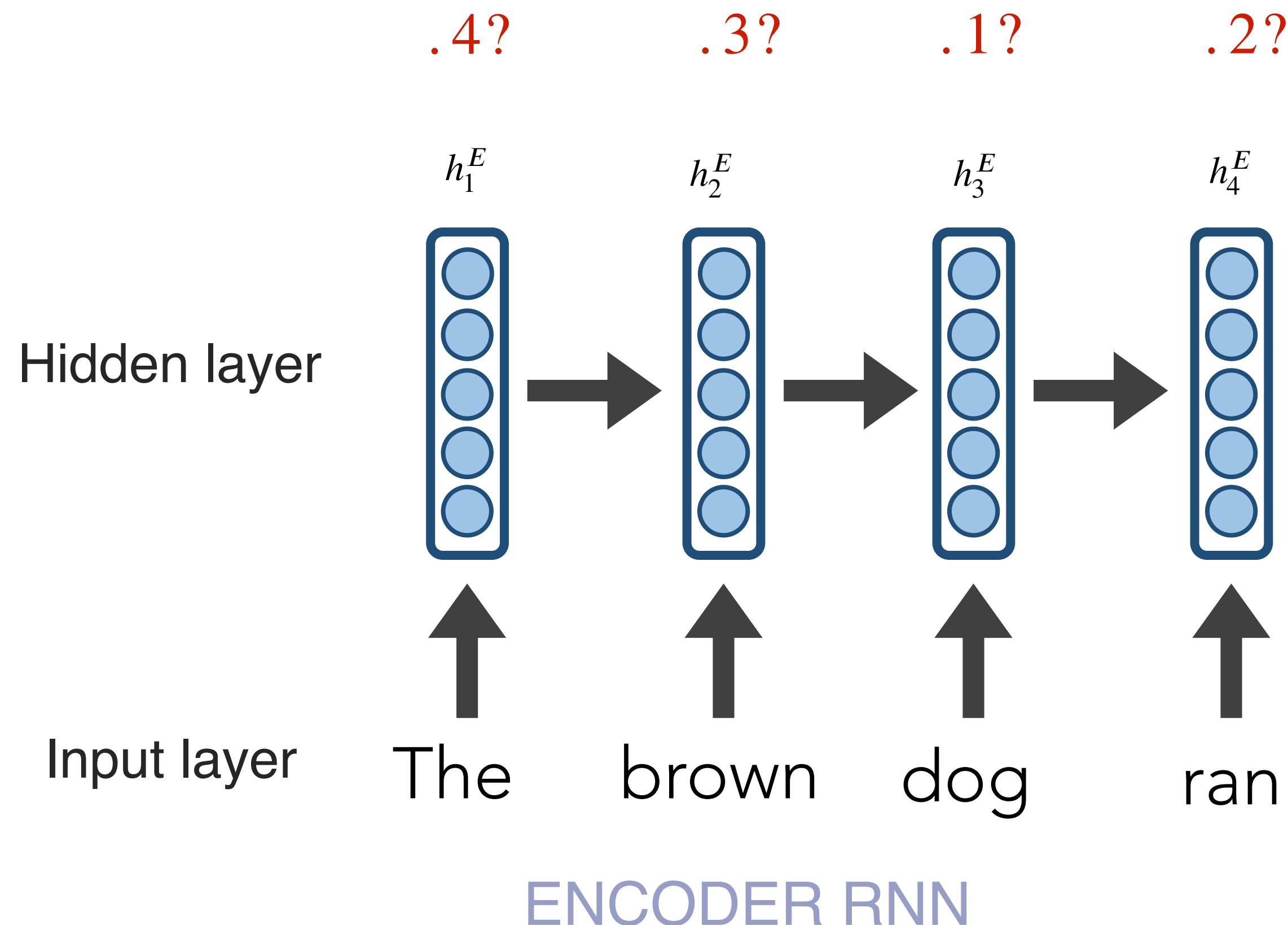
# Attention Motivation

The concept of attention within cognitive neuroscience and psychology dates back to the 1800s. [William James, 1890].

Nadaray-Watson kernel regression proposed in 1964. It locally weighted its predictions.

# Attention Architecture

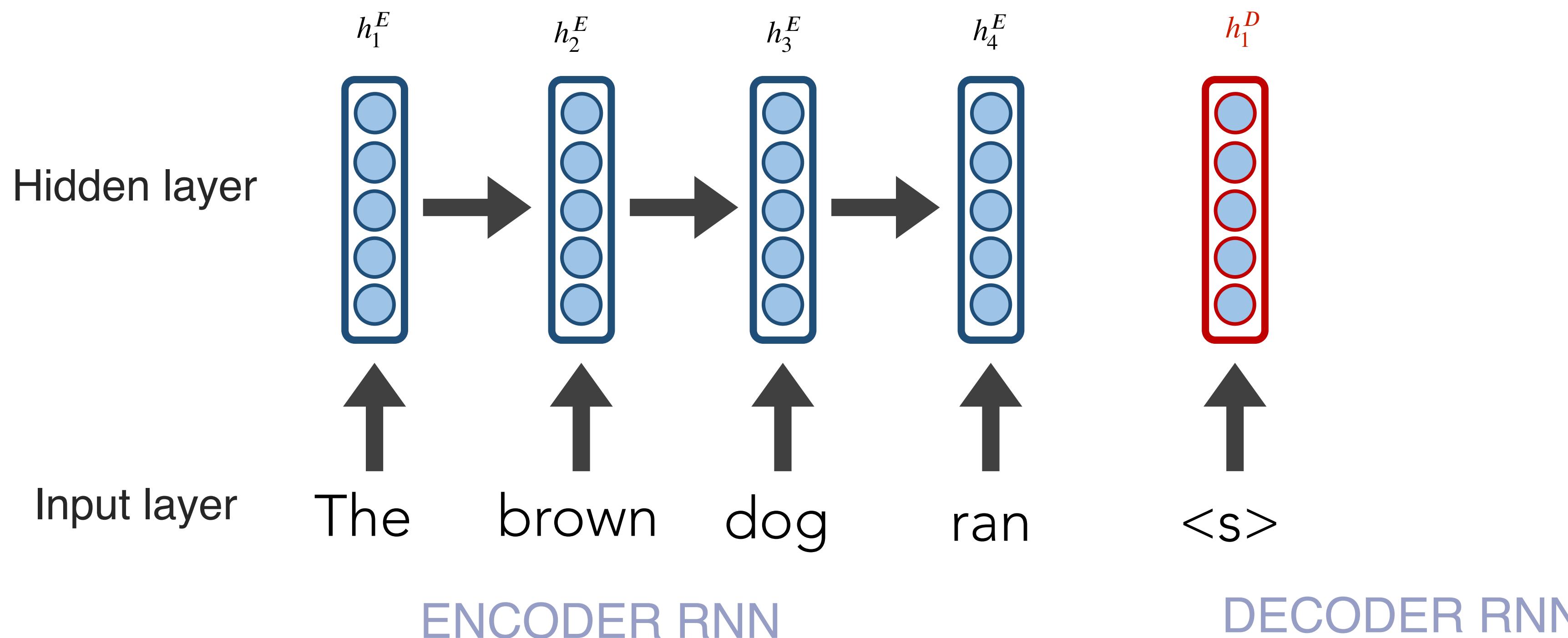
Q: How do we determine how much to pay attention to each of the encoder's hidden layers?



# Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

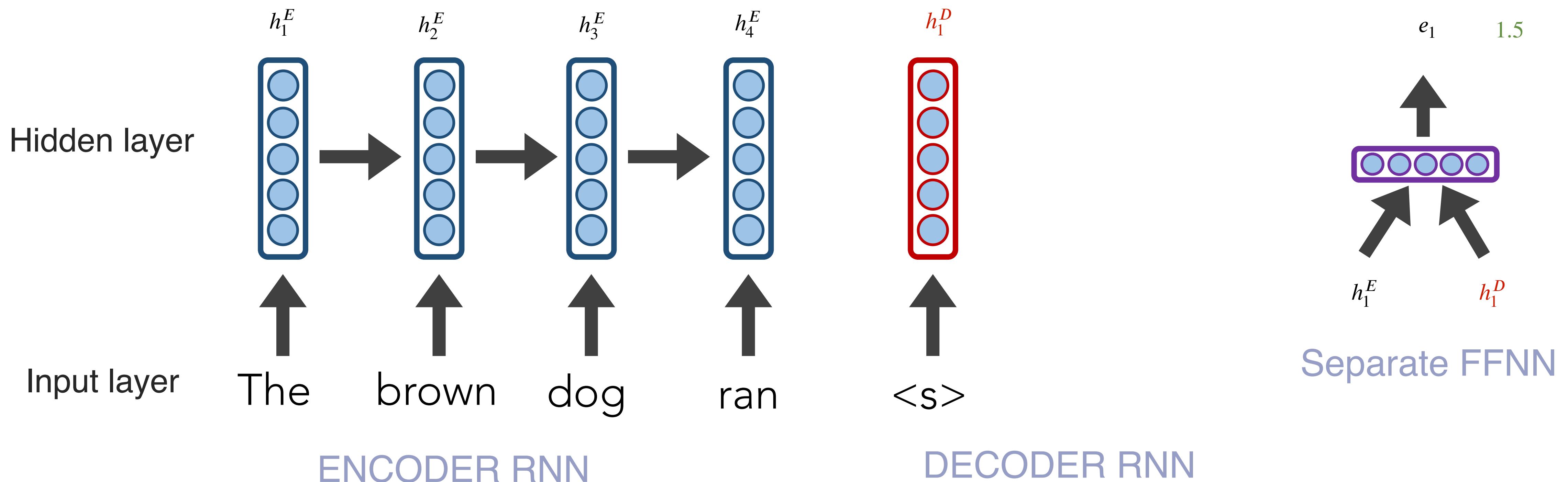
A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



# Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

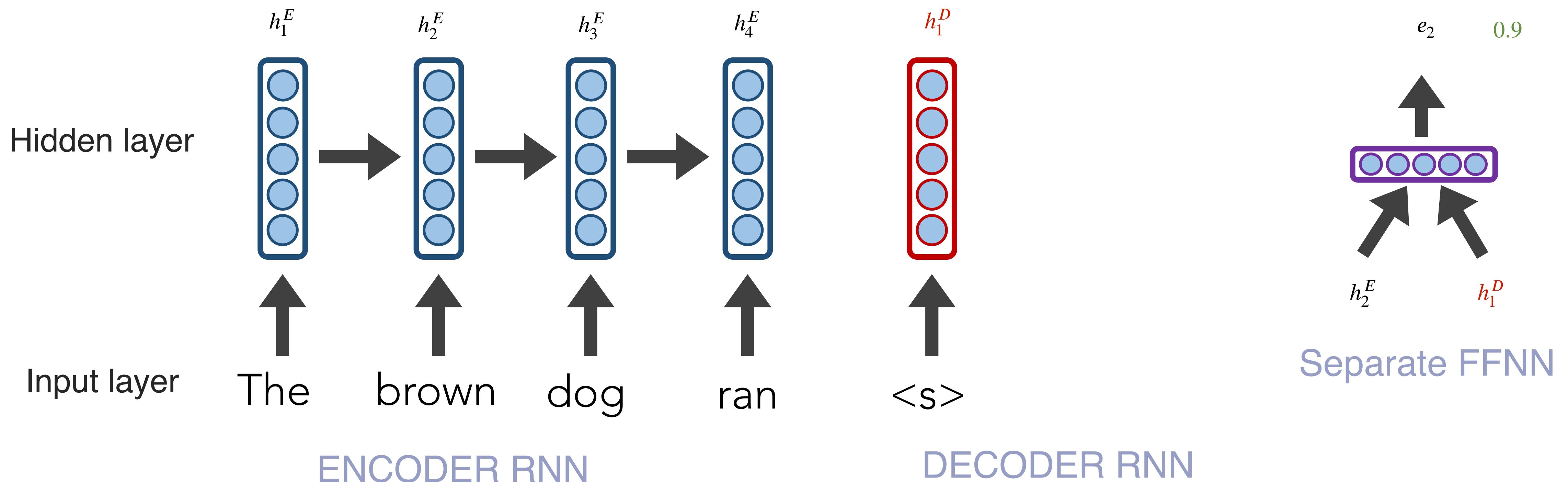
A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



# Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

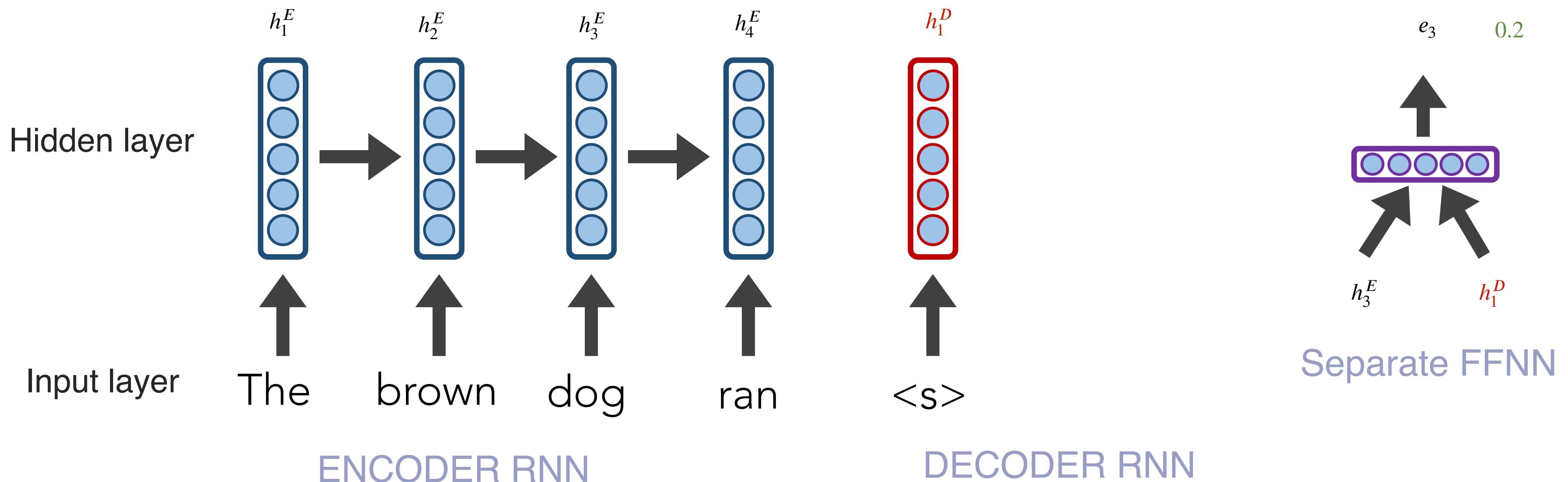
A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



# Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

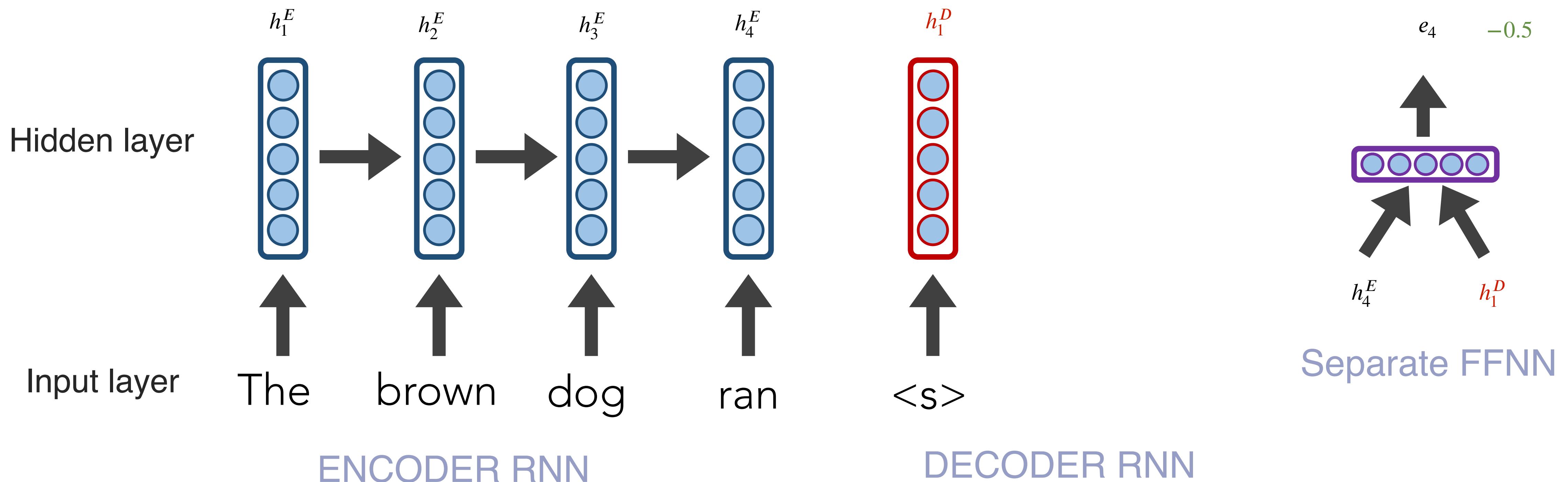
A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



# Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

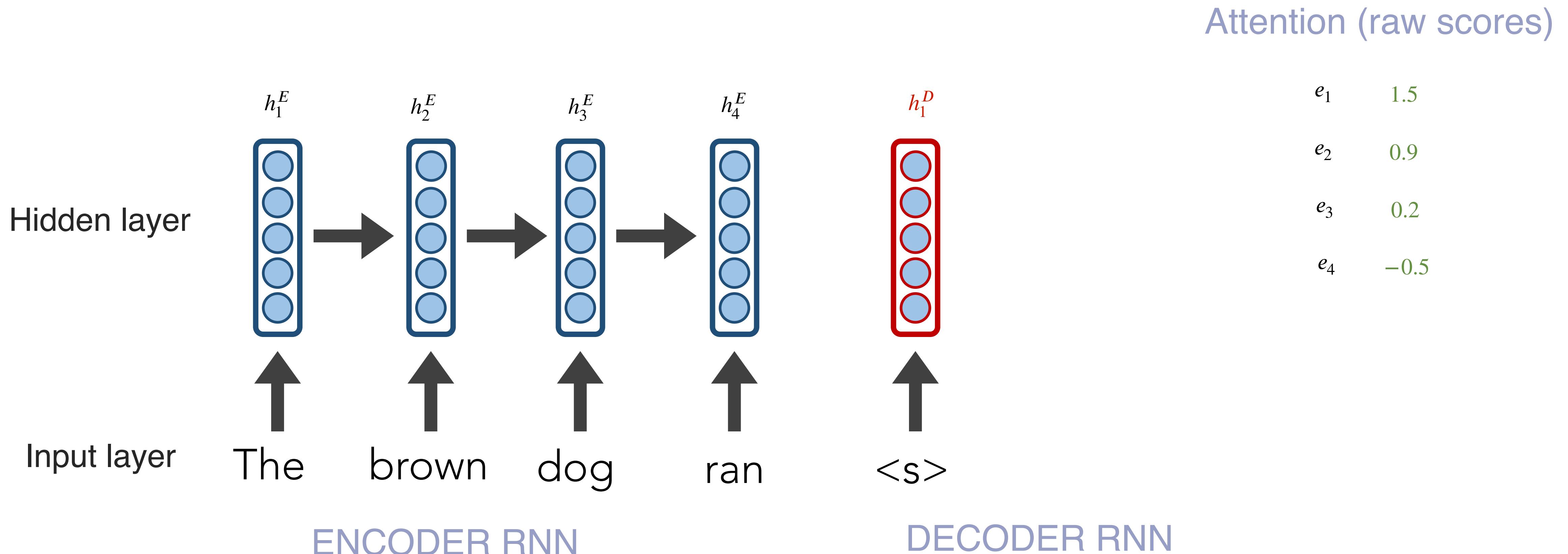
A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



# Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

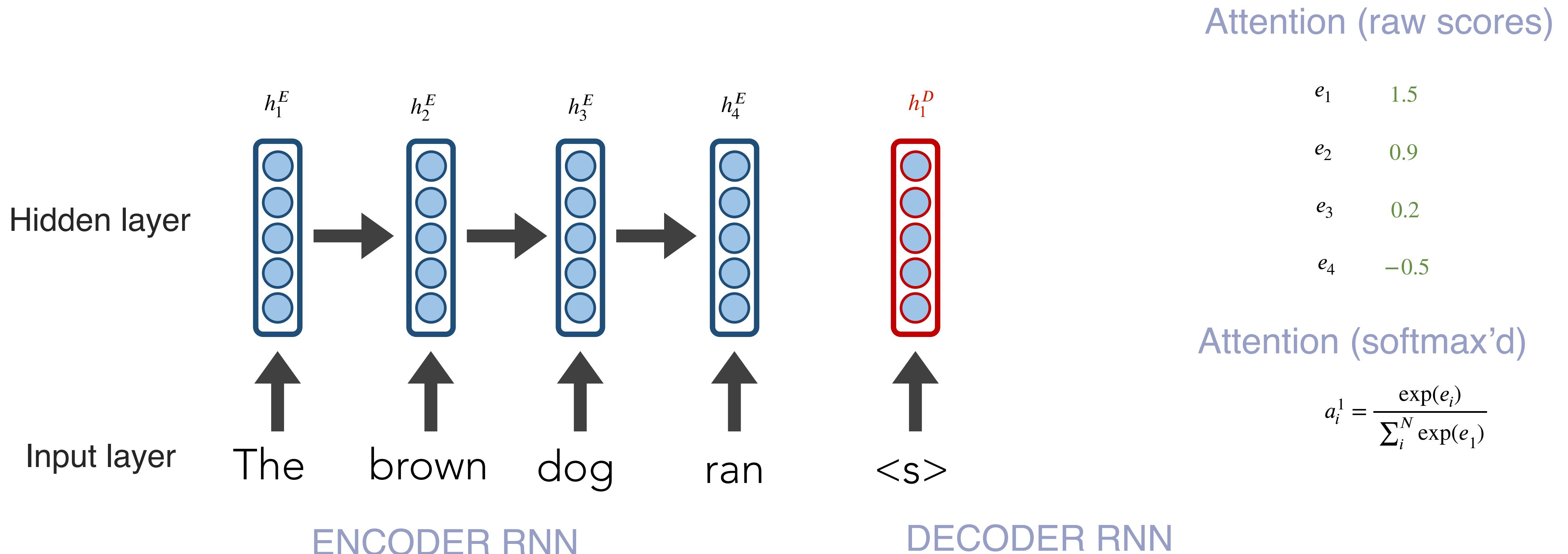
A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



# Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

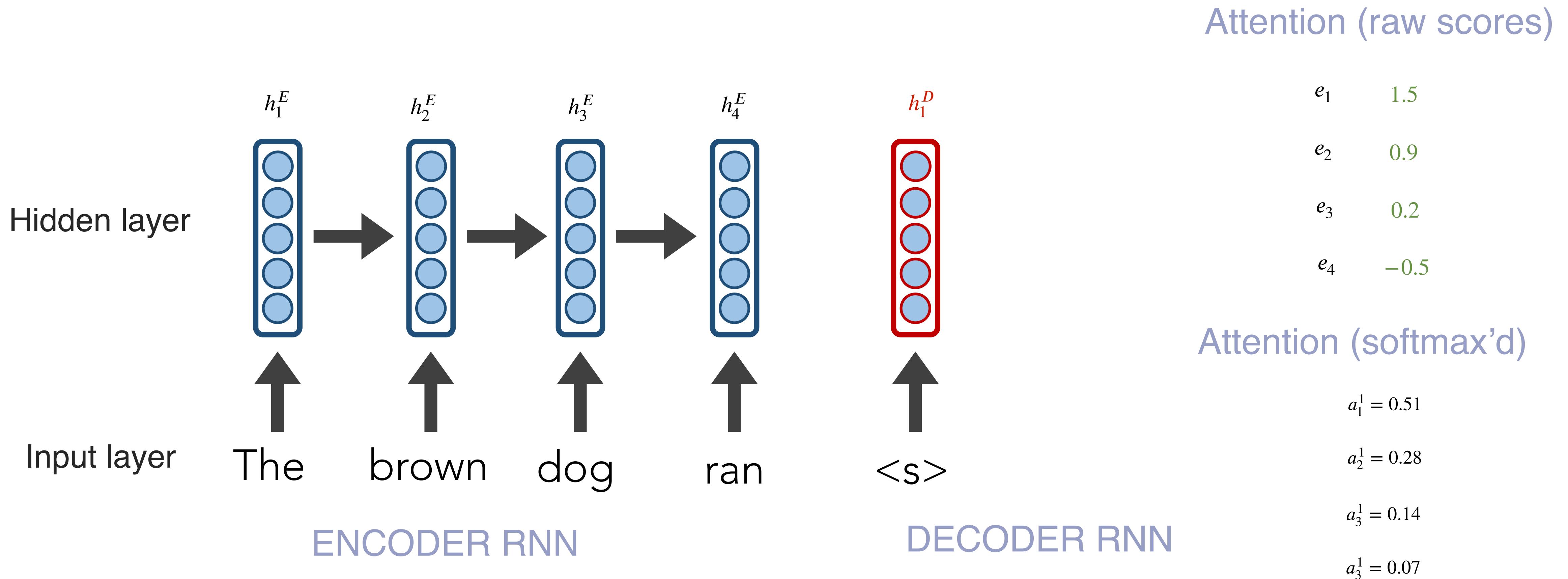
A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



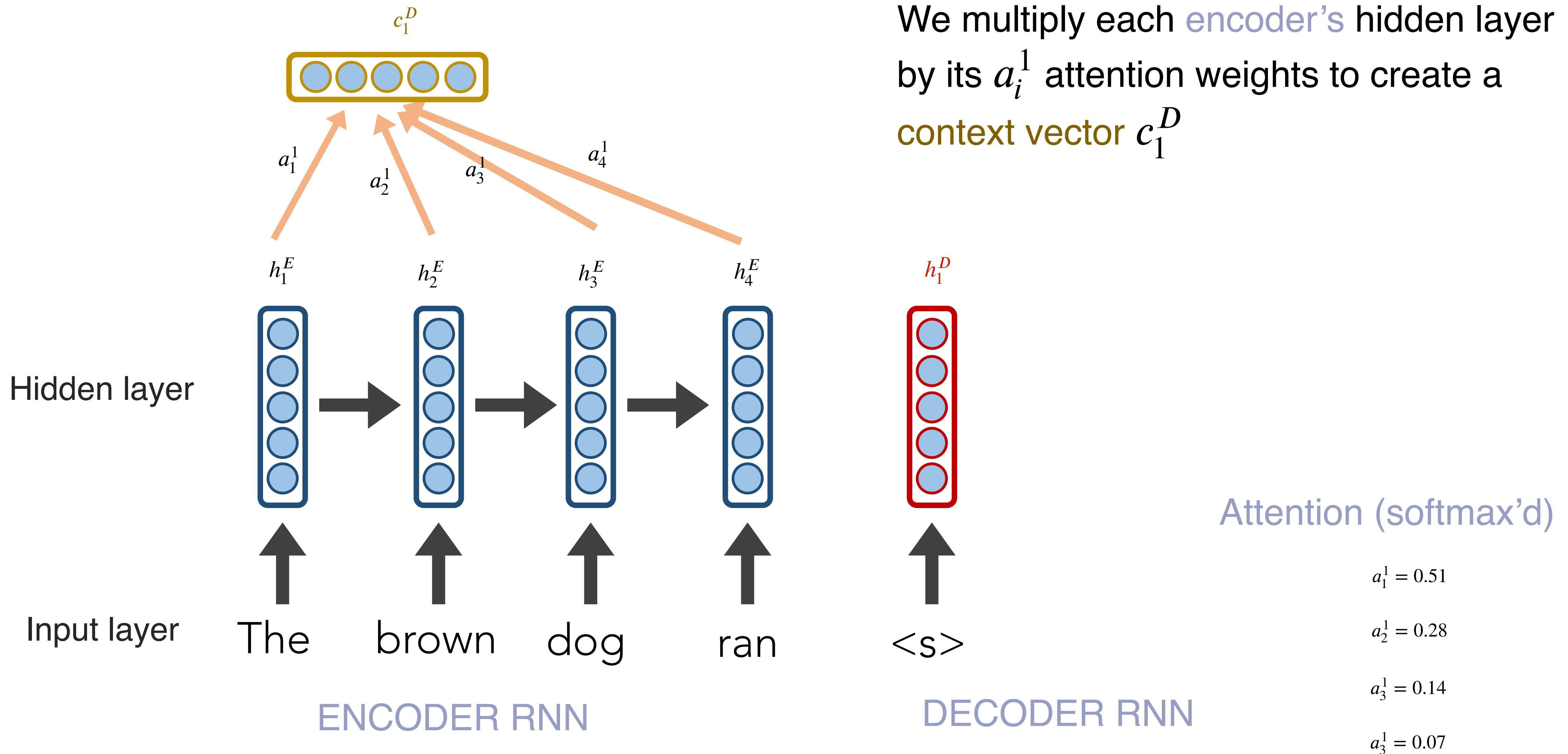
# Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!

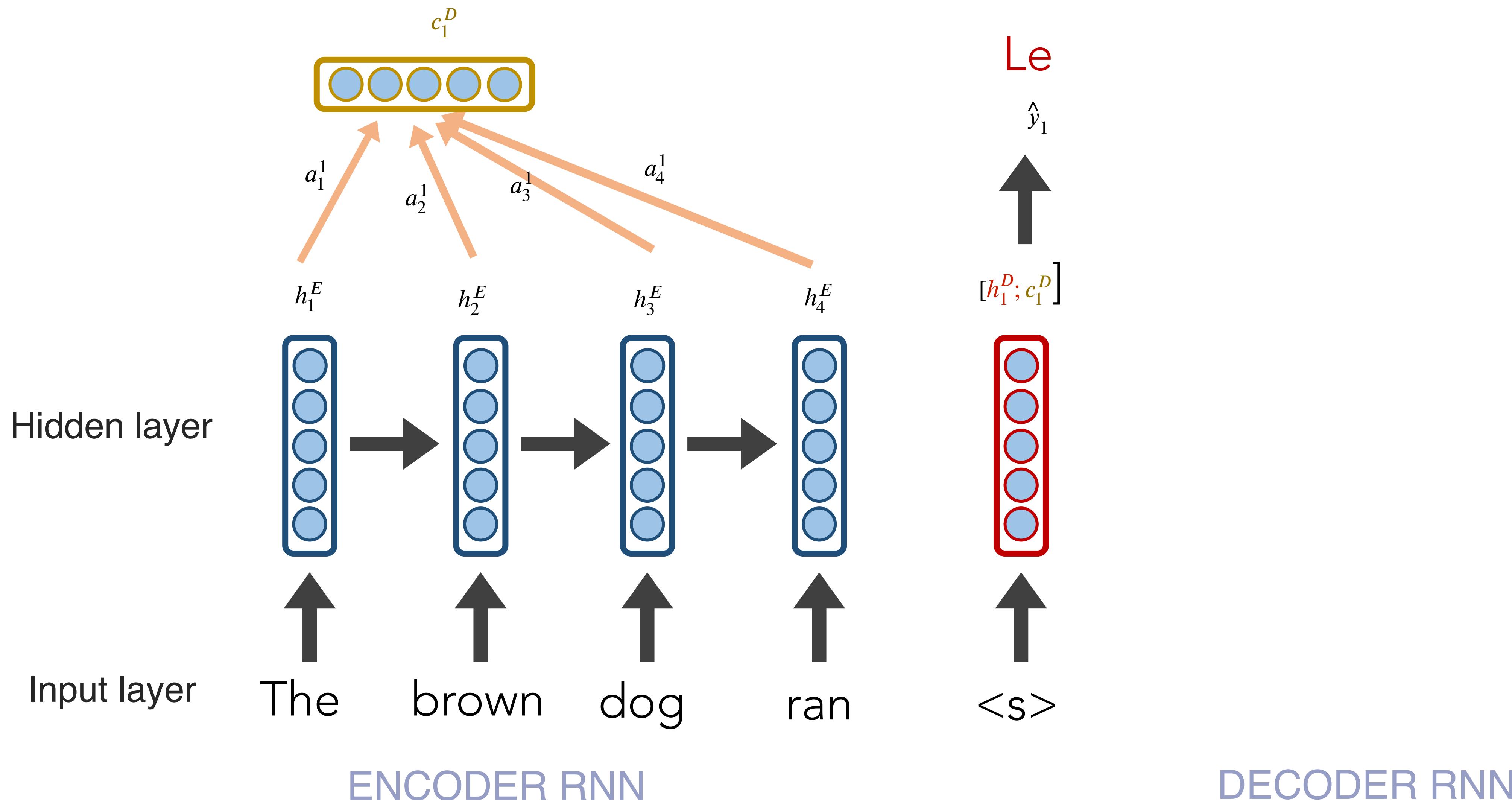


# Attention Architecture



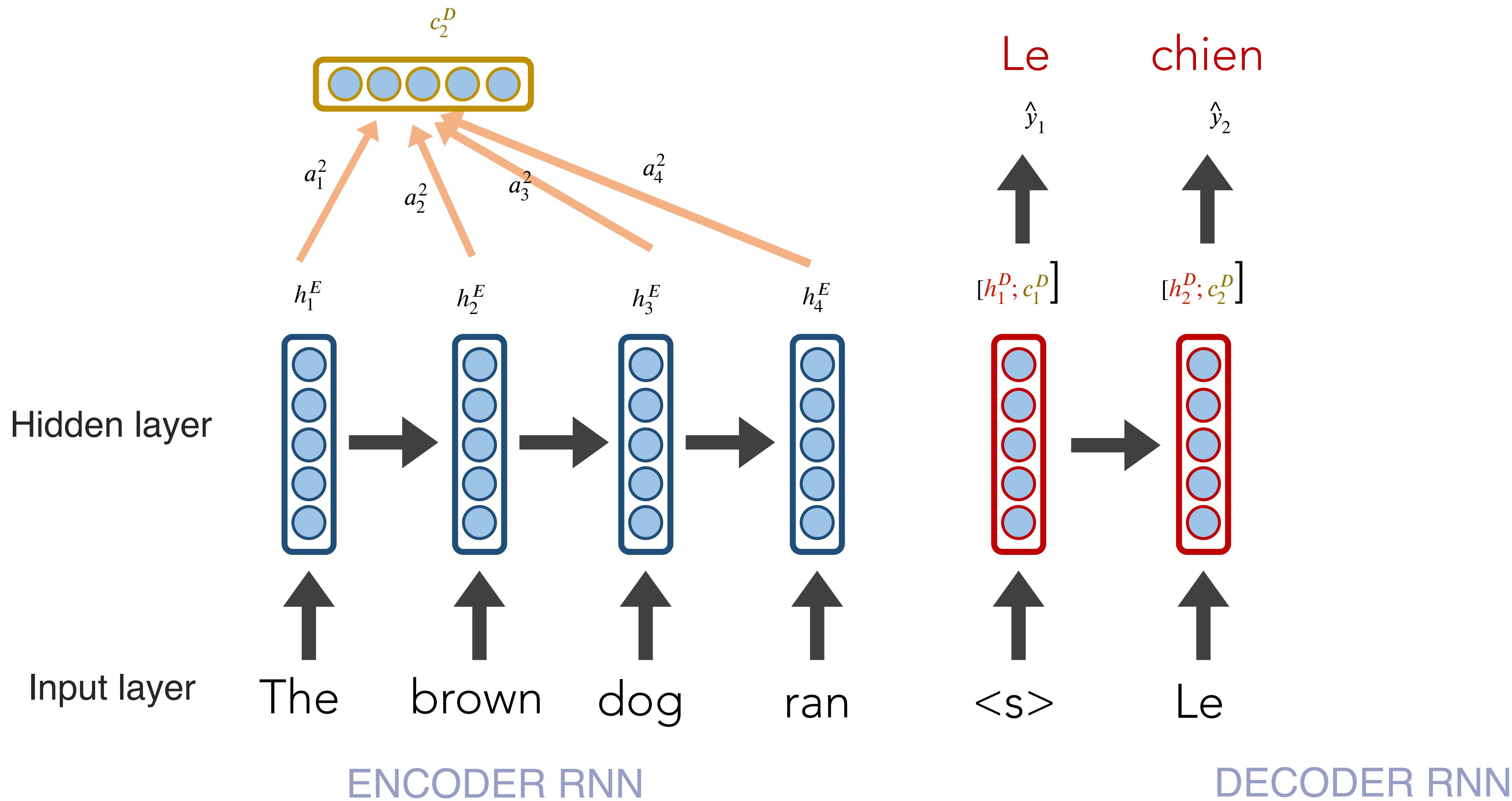
# Attention Architecture

REMEMBER: each attention weight  $a_i^j$  is based on the decoder's current hidden state, too.



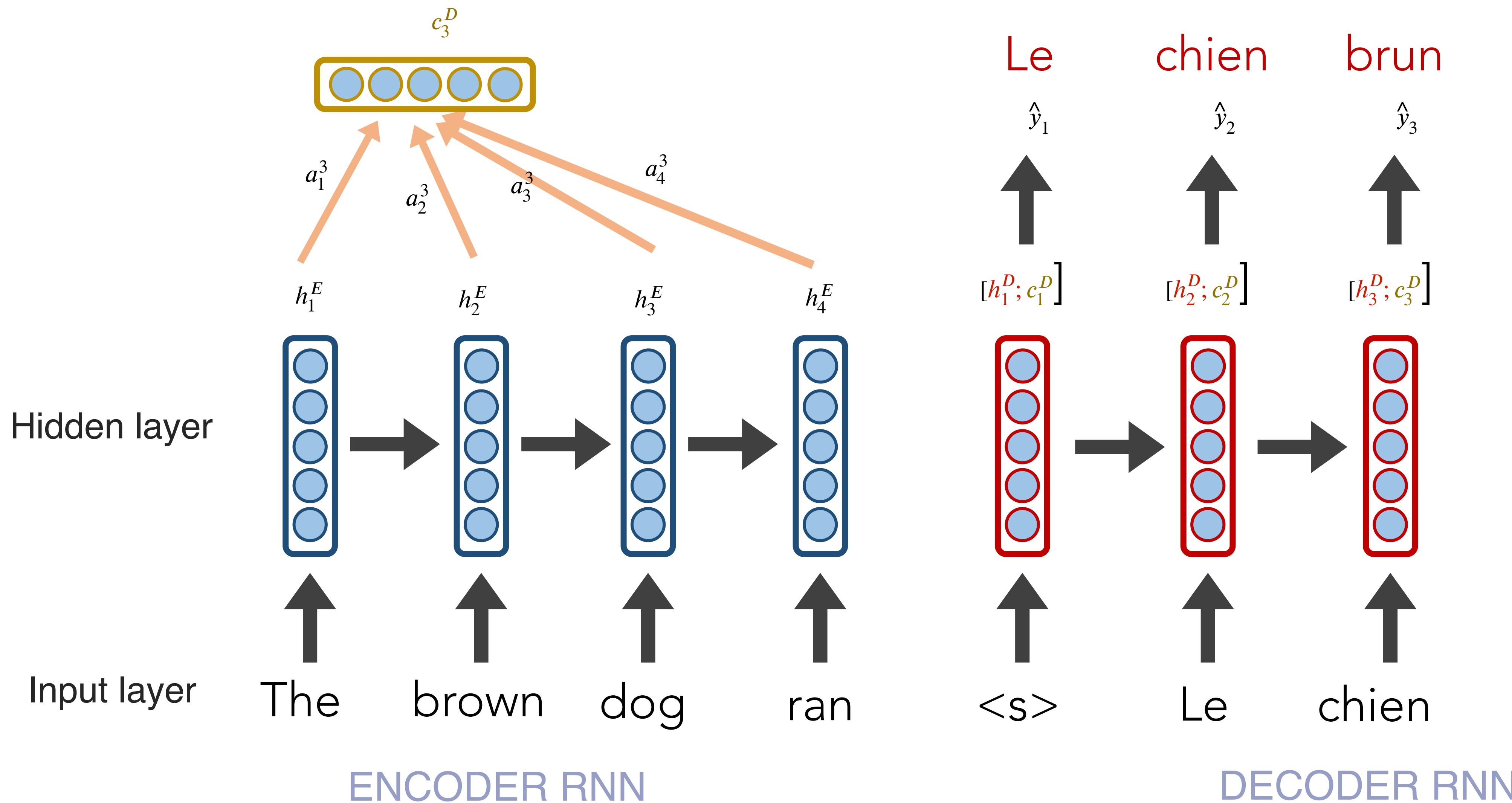
# Attention Architecture

REMEMBER: each attention weight  $a_i^j$  is based on the decoder's current hidden state, too.



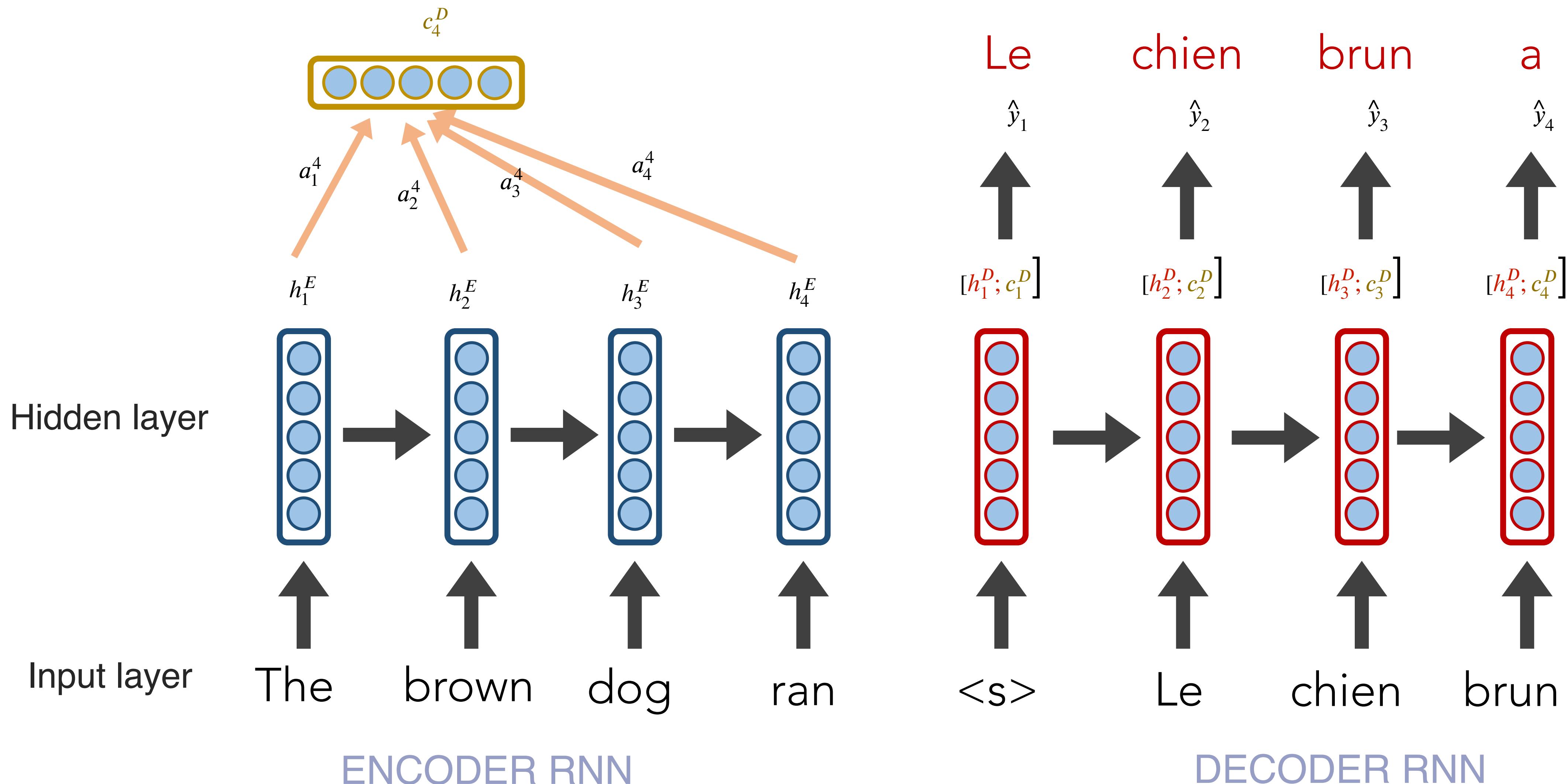
# Attention Architecture

REMEMBER: each attention weight  $a_i^j$  is based on the decoder's current hidden state, too.



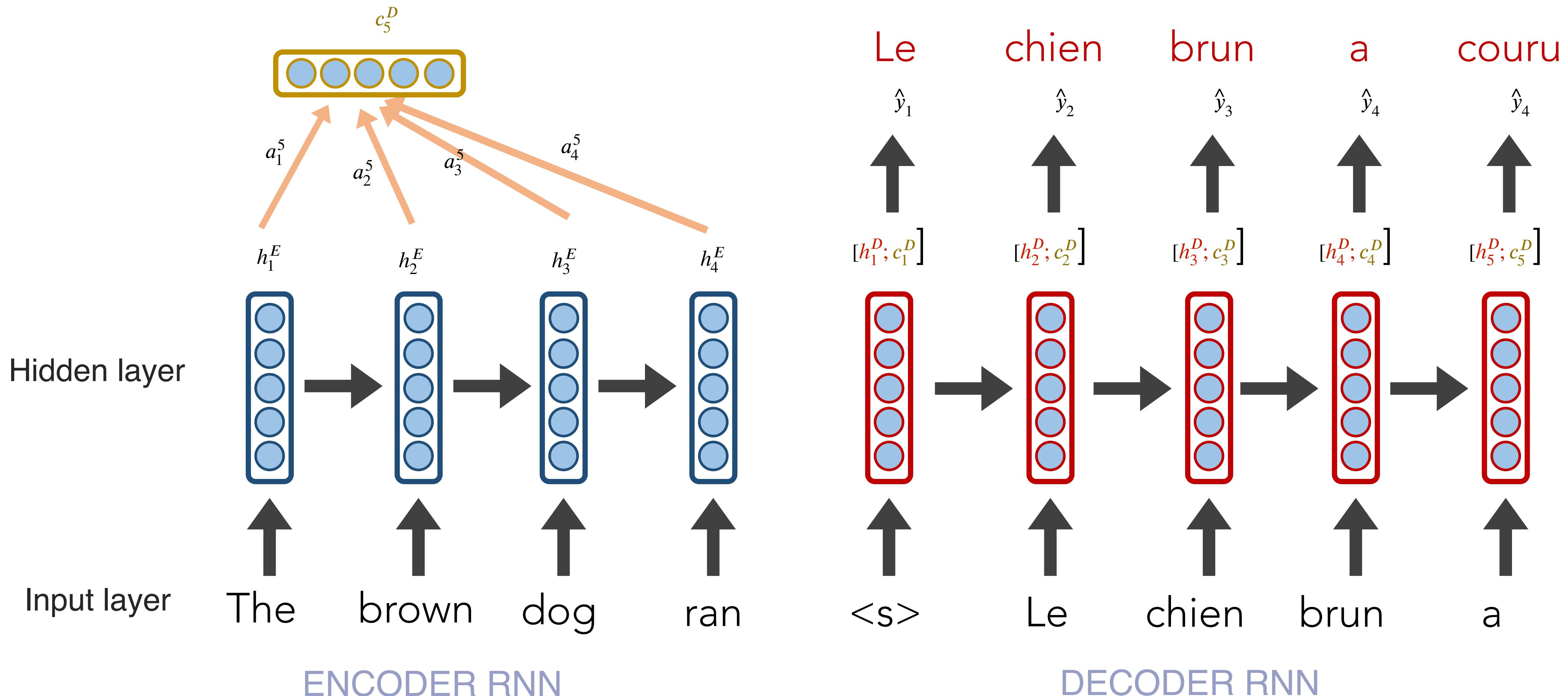
# Attention Architecture

REMEMBER: each attention weight  $a_i^j$  is based on the decoder's current hidden state, too.

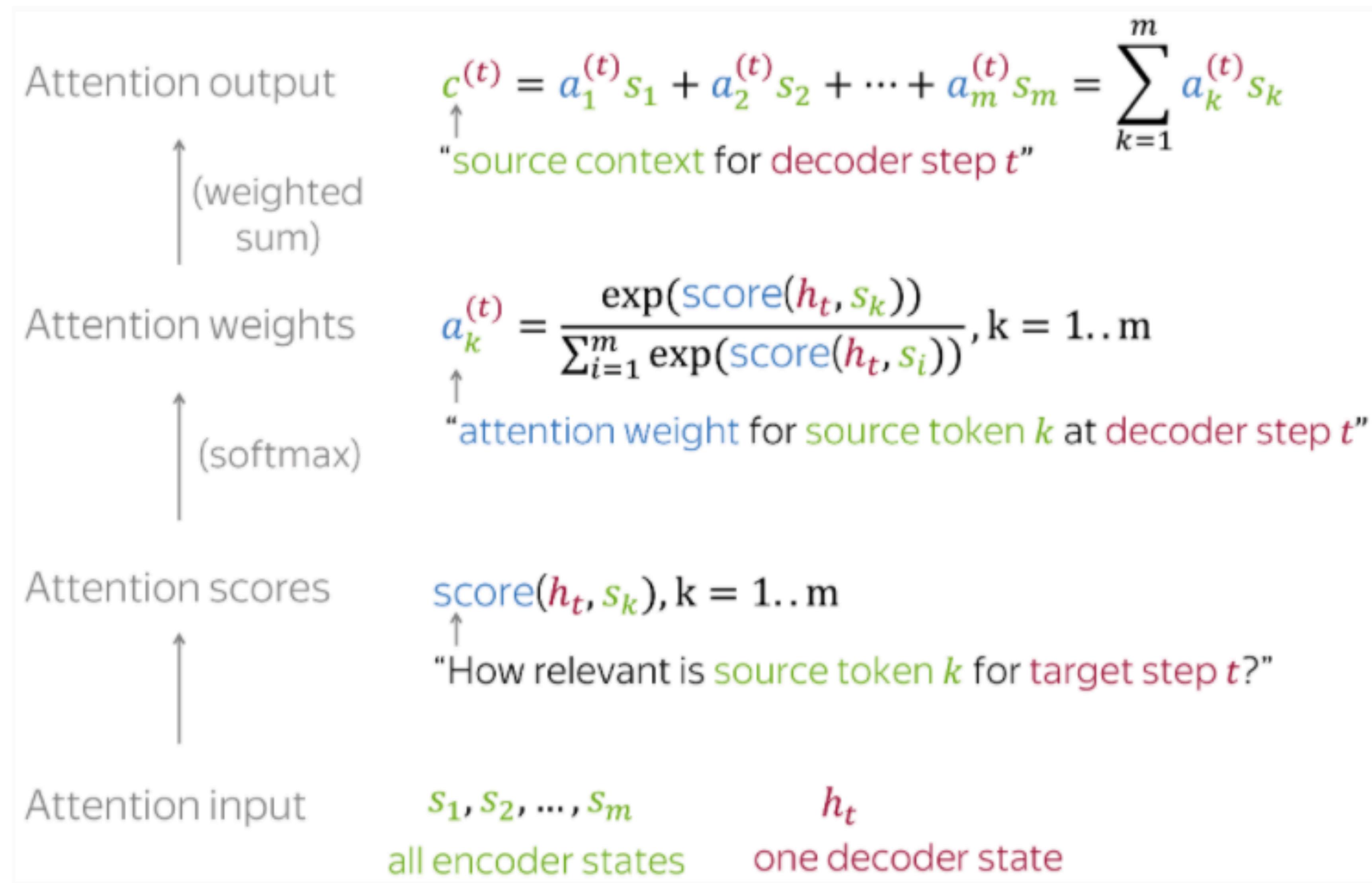


# Attention Architecture

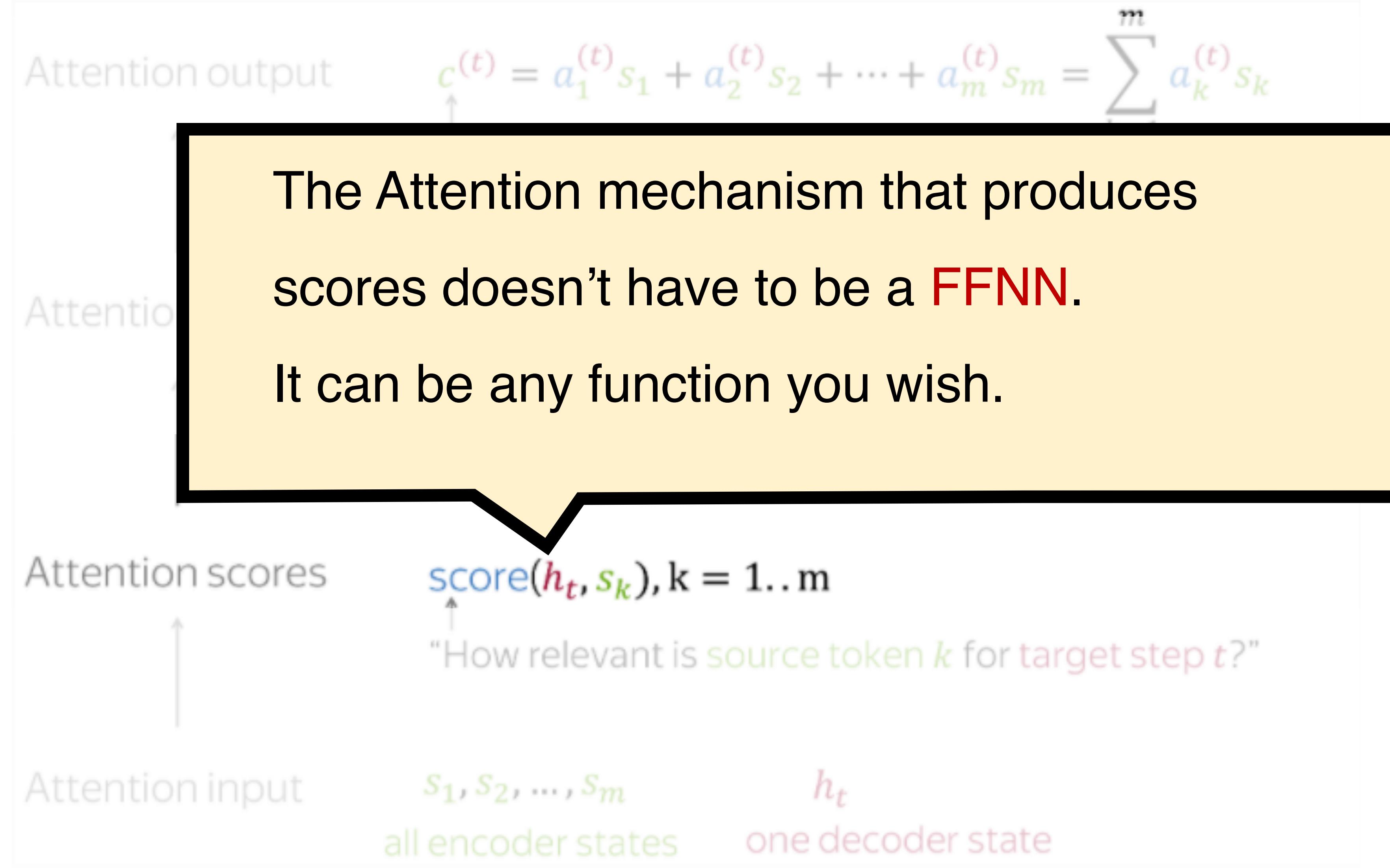
REMEMBER: each attention weight  $a_i^j$  is based on the decoder's current hidden state, too.



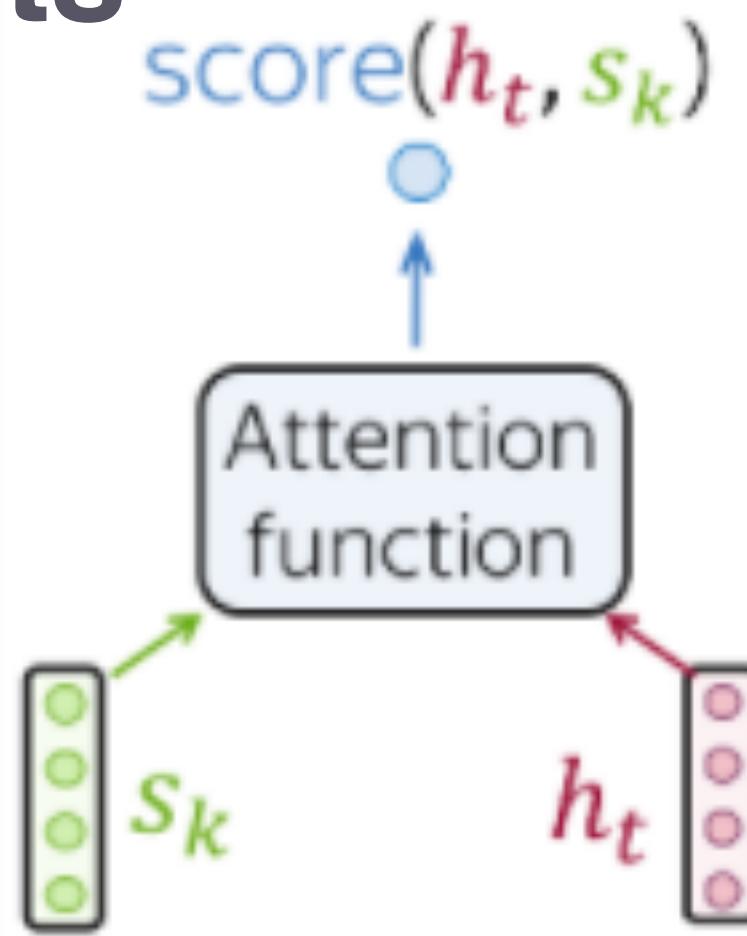
# Attention Summary



# Attention Summary



# Attention Variants



Dot-product

$$\begin{matrix} h_t^T \\ \text{---} \\ \text{---} \end{matrix} \times \begin{matrix} \text{---} \\ \text{---} \\ \text{---} \\ s_k \end{matrix}$$

$$\text{score}(h_t, s_k) = h_t^T s_k$$

Bilinear

$$\begin{matrix} h_t^T \\ \text{---} \\ \text{---} \end{matrix} \times \begin{matrix} W \\ \text{---} \end{matrix} \times \begin{matrix} \text{---} \\ \text{---} \\ \text{---} \\ s_k \end{matrix}$$

$$\text{score}(h_t, s_k) = h_t^T W s_k$$

Multi-Layer Perceptron

$$\begin{matrix} w_2^T \\ \text{---} \\ \text{---} \end{matrix} \times \tanh \left[ \begin{matrix} W_1 \\ \text{---} \end{matrix} \times \begin{matrix} \text{---} \\ \text{---} \\ \text{---} \\ h_t \end{matrix} \right] s_k$$

$$\text{score}(h_t, s_k) = w_2^T \cdot \tanh(W_1[h_t, s_k])$$

# Attention Summary

## Attention:

- greatly improves seq2seq results
- allows us to visualize the contribution each **encoding word** gave for each **decoder's word**

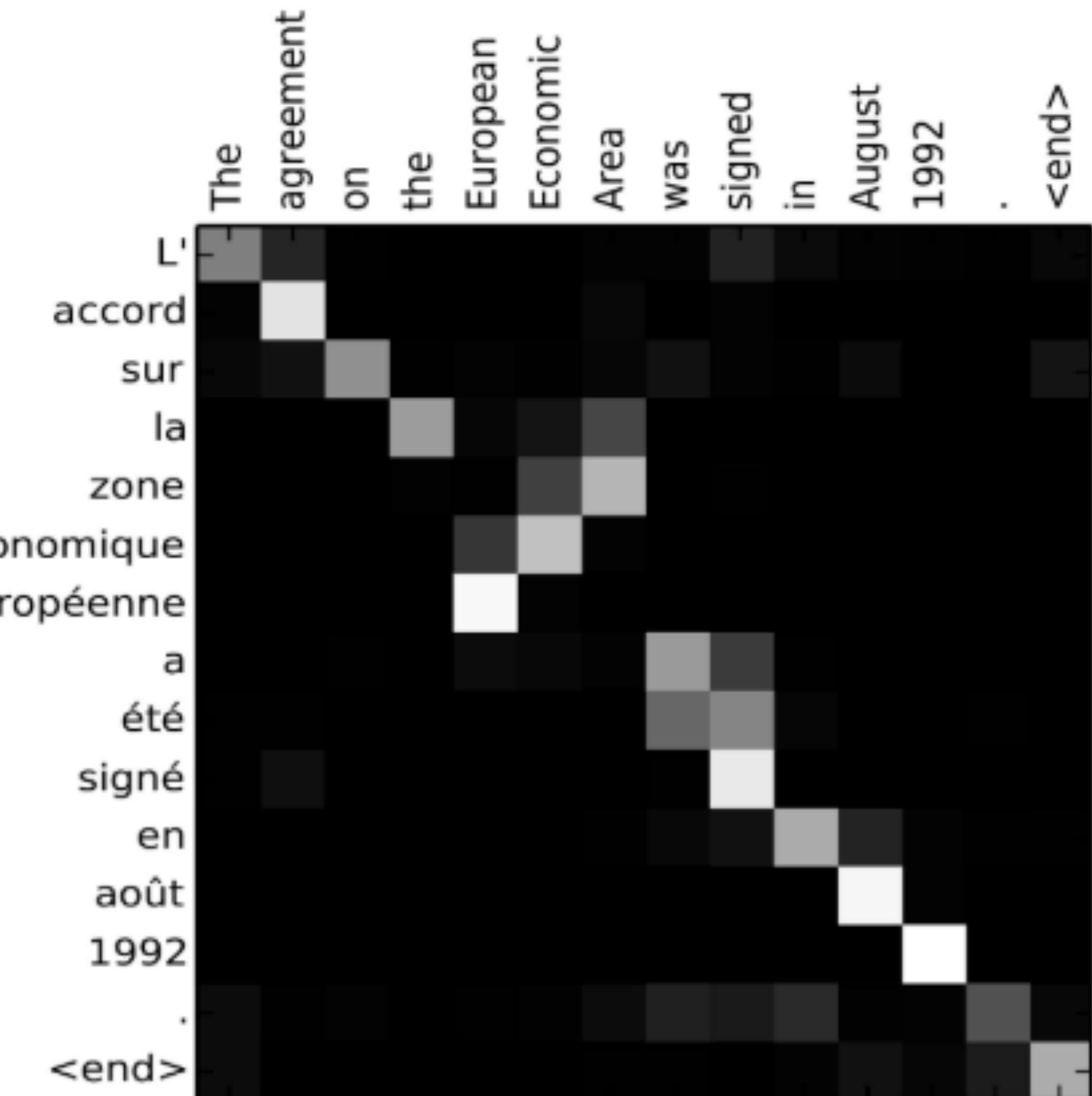
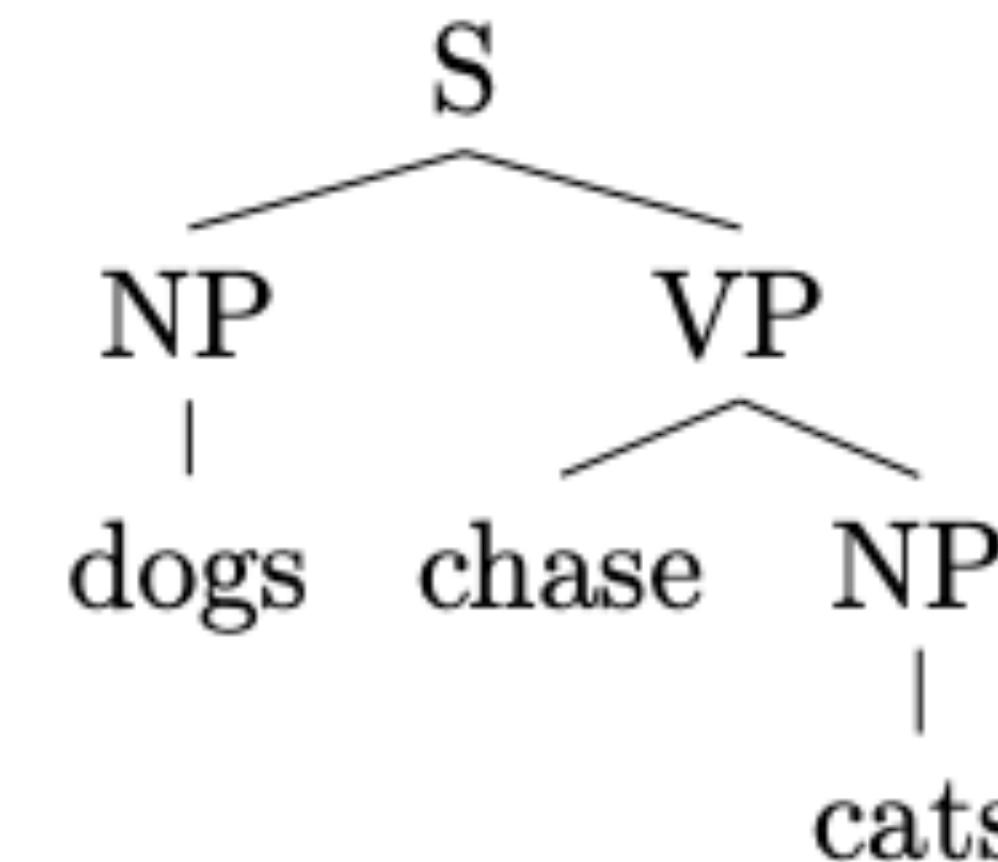


Image source: Fig 3 in [Bahdanau et al., 2015](#)

# Attention Applications

Input: dogs chase cats

Output:



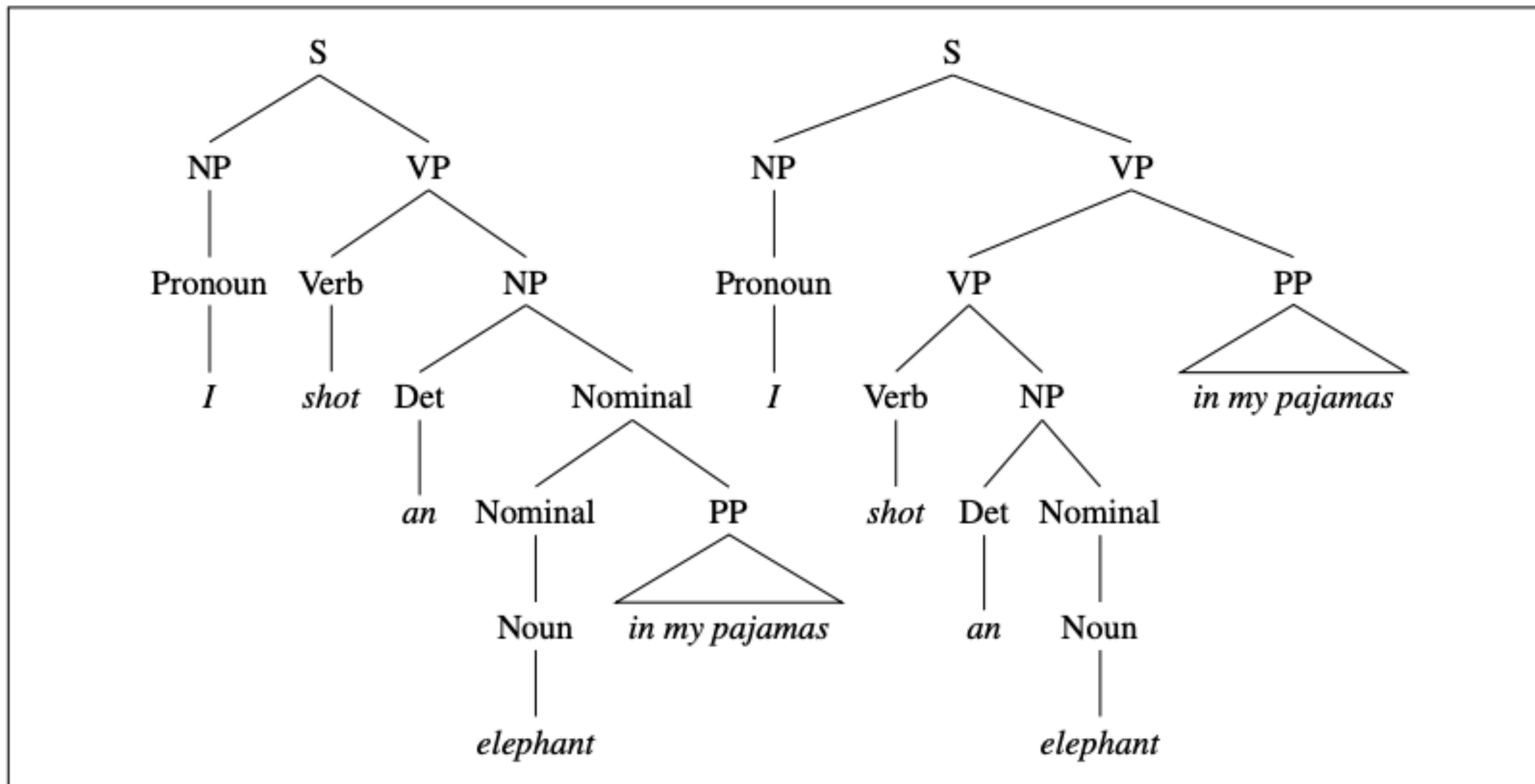
or a flattened representation

(S (NP dogs )<sub>NP</sub> (VP chase (NP cats )<sub>NP</sub> )<sub>VP</sub> )<sub>S</sub>

# Attention Applications

Input: I shot an elephant in my pajamas

Output:



**Figure 13.2** Two parse trees for an ambiguous sentence. The parse on the left corresponds to the humorous reading in which the elephant is in the pajamas, the parse on the right corresponds to the reading in which Captain Spaulding did the shooting in his pajamas.

# Attention Applications

Model	English			Chinese		
	LR	LP	F1	LR	LP	F1
Shen et al. (2018)	92.0	91.7	91.8	86.6	86.4	86.5
Fried and Klein (2018)	-	-	92.2	-	-	87.0
Teng and Zhang (2018)	92.2	92.5	92.4	86.6	88.0	87.3
Vaswani et al. (2017)	-	-	92.7	-	-	-
Dyer et al. (2016)	-	-	93.3	-	-	84.6
Kuncoro et al. (2017)	-	-	93.6	-	-	-
Charniak et al. (2016)	-	-	93.8	-	-	-
Liu and Zhang (2017b)	91.3	92.1	91.7	85.9	85.2	85.5
Liu and Zhang (2017a)	-	-	94.2	-	-	86.1
Suzuki et al. (2018)	-	-	94.32	-	-	-
Takase et al. (2018)	-	-	94.47	-	-	-
Fried et al. (2017)	-	-	94.66	-	-	-
Kitaev and Klein (2018)	94.85	95.40	95.13	-	-	-
Kitaev et al. (2018)	95.51	96.03	95.77	91.55	91.96	91.75
Zhou and Zhao (2019) (BERT)	95.70	95.98	95.84	<b>92.03</b>	92.33	92.18
Zhou and Zhao (2019) (XLNet)	96.21	96.46	96.33	-	-	-
Our work	<b>96.24</b>	<b>96.53</b>	<b>96.38</b>	91.85	<b>93.45</b>	<b>92.64</b>

Table 3: Constituency Parsing on PTB & CTB test sets.

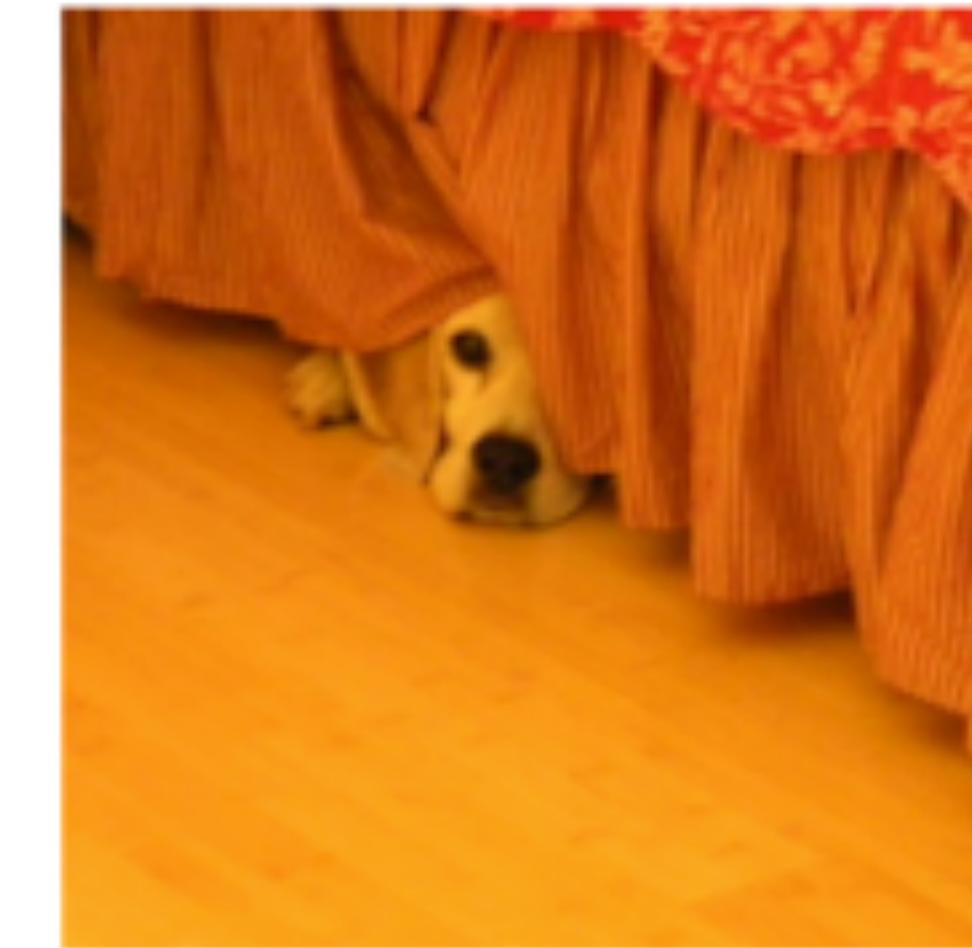
# Visualizing Attention

Input: image

Output: generated text



A woman is throwing a frisbee in a park.



A dog is standing on a hardwood floor.

Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)

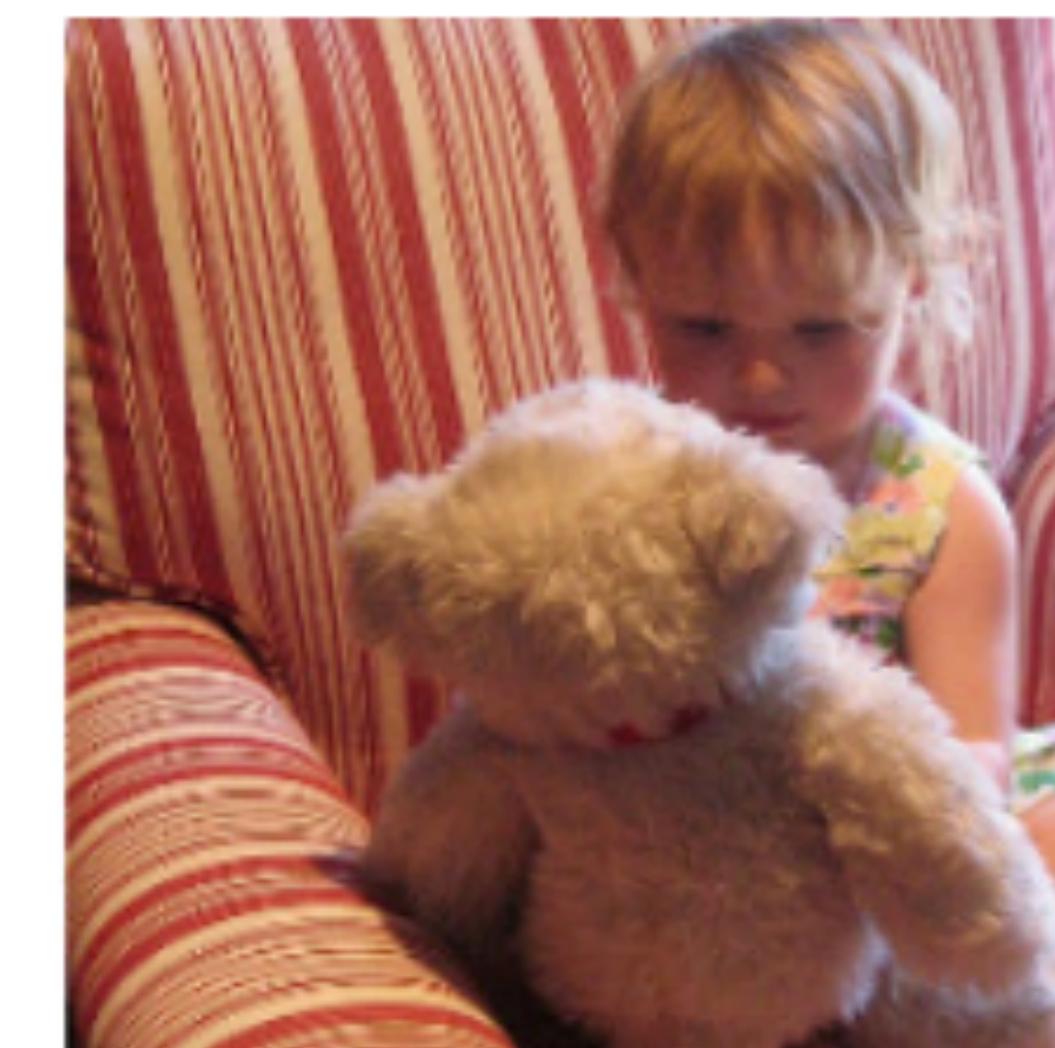
# Visualizing Attention

Input: image

Output: generated text



A stop sign is on a road with a mountain in the background.

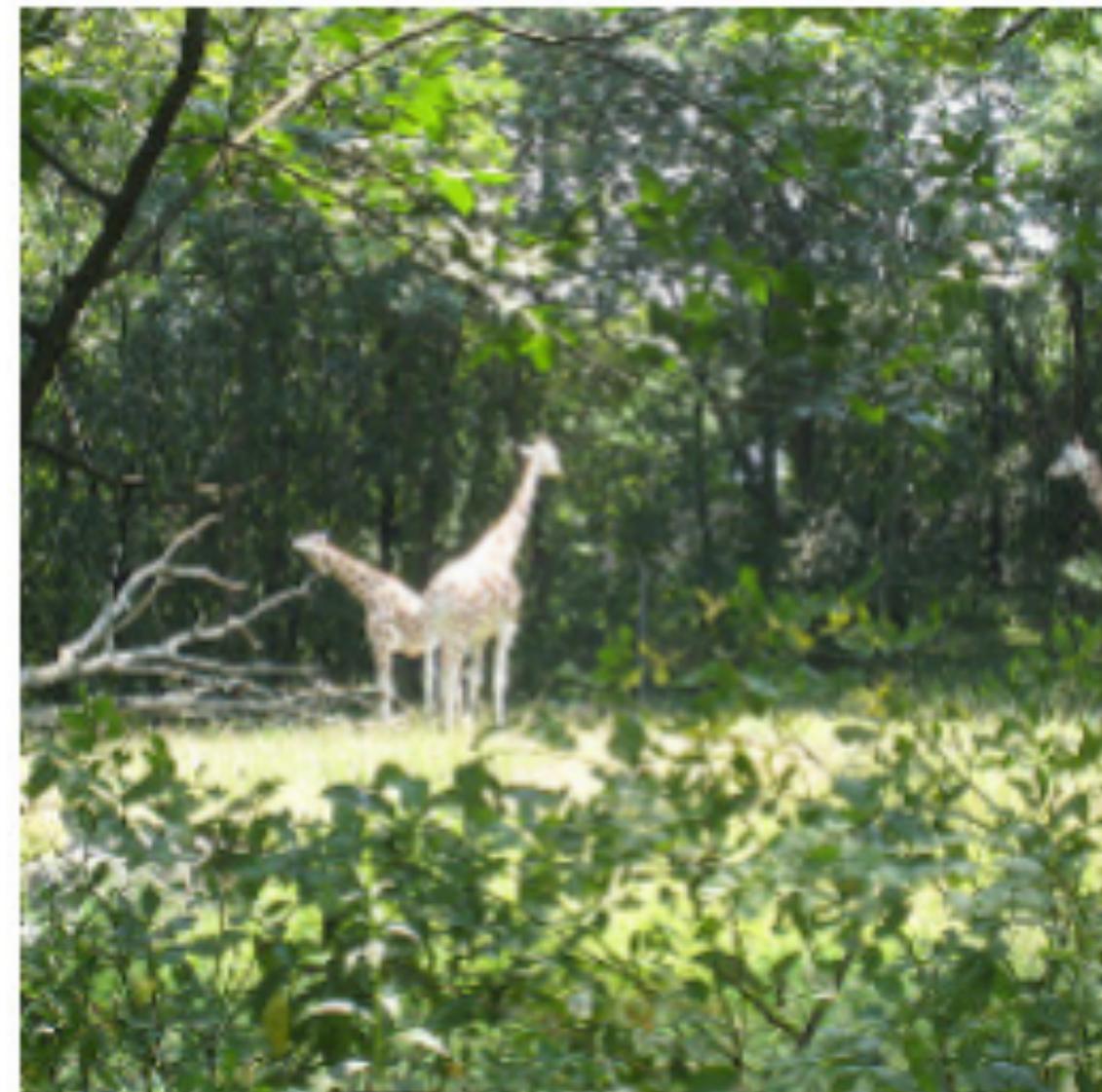


A little girl sitting on a bed with a teddy bear.

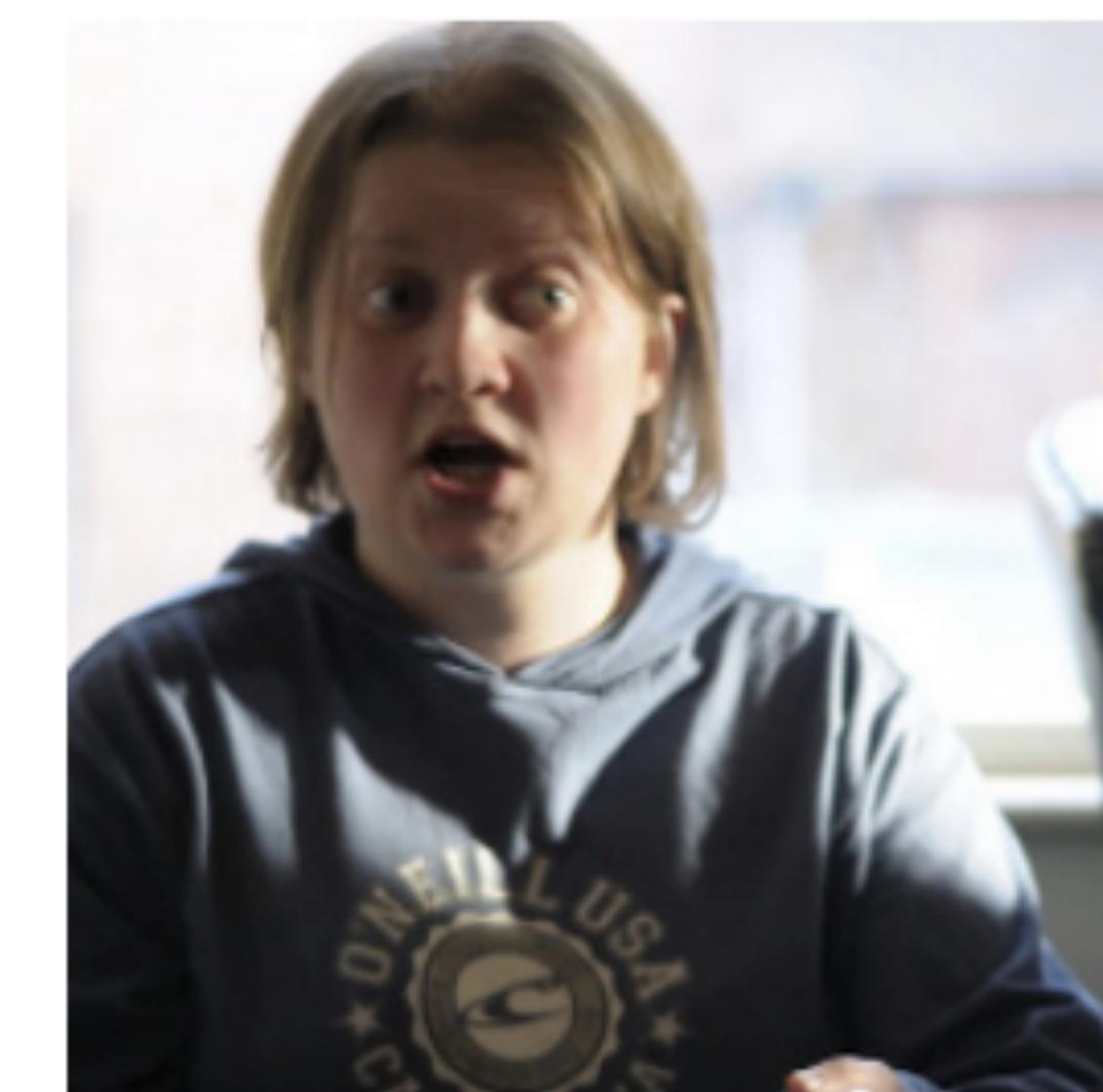


Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)

# Visualizing Attention



A large white bird standing in a forest.



A woman holding a clock in her hand.



*Figure 5. Examples of mistakes where we can use attention to gain intuition into what the model saw.*

# Visualizing Attention



A woman is sitting at a table  
with a large pizza.



A person is standing on a beach  
with a surfboard.



*Figure 5. Examples of mistakes where we can use attention to gain intuition into what the model saw.*

# Self Attention

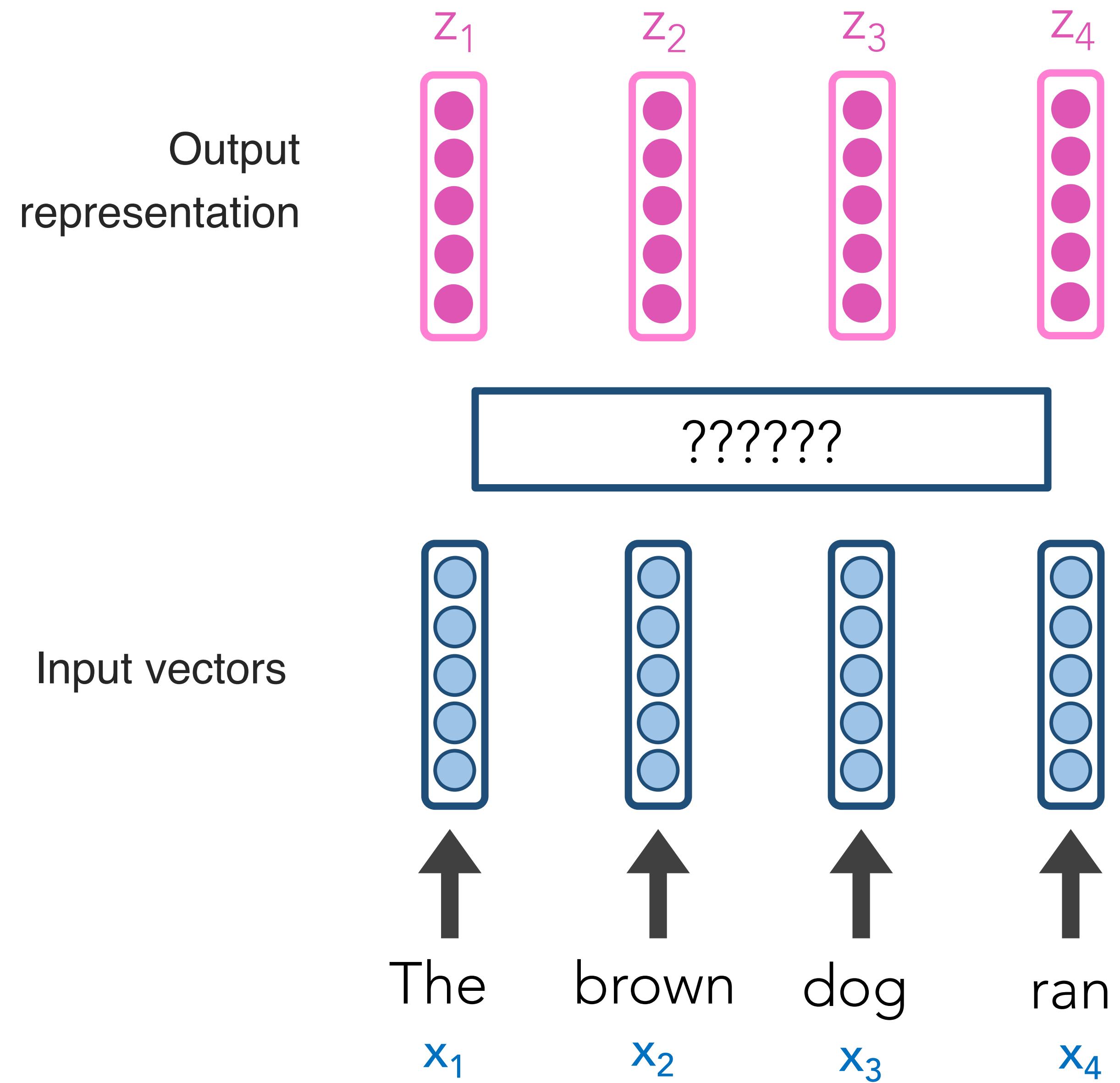
# Self Attention Motivation

- Word meaning is contextual!
  - *The shirt is green*
  - *The recruits are green*
  - *Green policy has wrecked the economy*
- But word (type) embeddings conflate all of these meanings
- Every occurrence (token) of “green” carries the same semantic mix
- Instead: embed individual term occurrences

# Self Attention Motivation

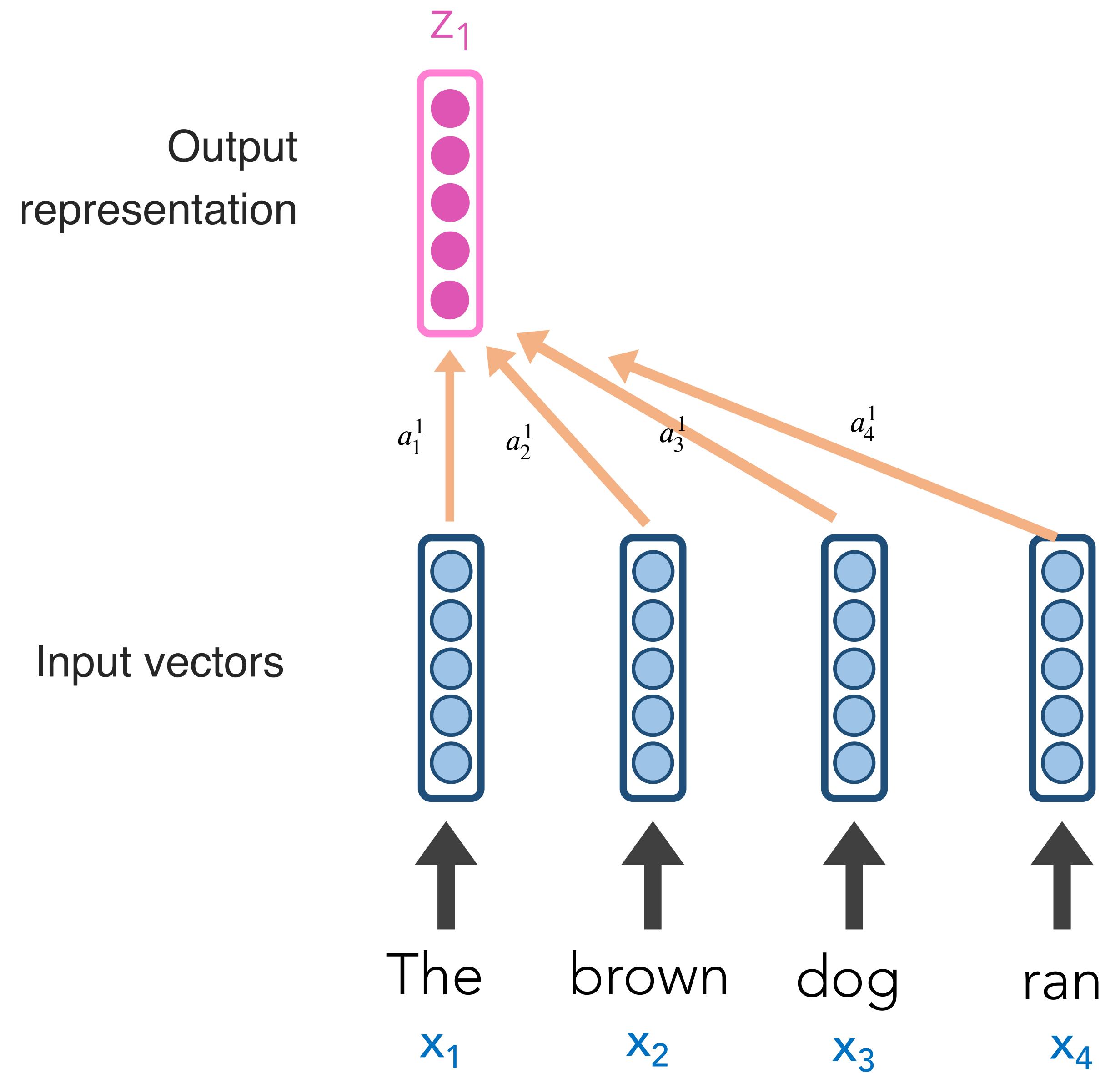
- Each word in a sequence is transformed into a rich, abstract representation ([context embedding](#)) based on the weighted sums of the other words in the same sequence (akin to deep CNN layers)
- Inspired by Attention, we want each word to determine, “how much should I be influenced by each of my neighbors”
- We want (explicit) positionality

# Self Attention Motivation



Self-Attention's goal is to create great representations,  $z_i$ , of the input

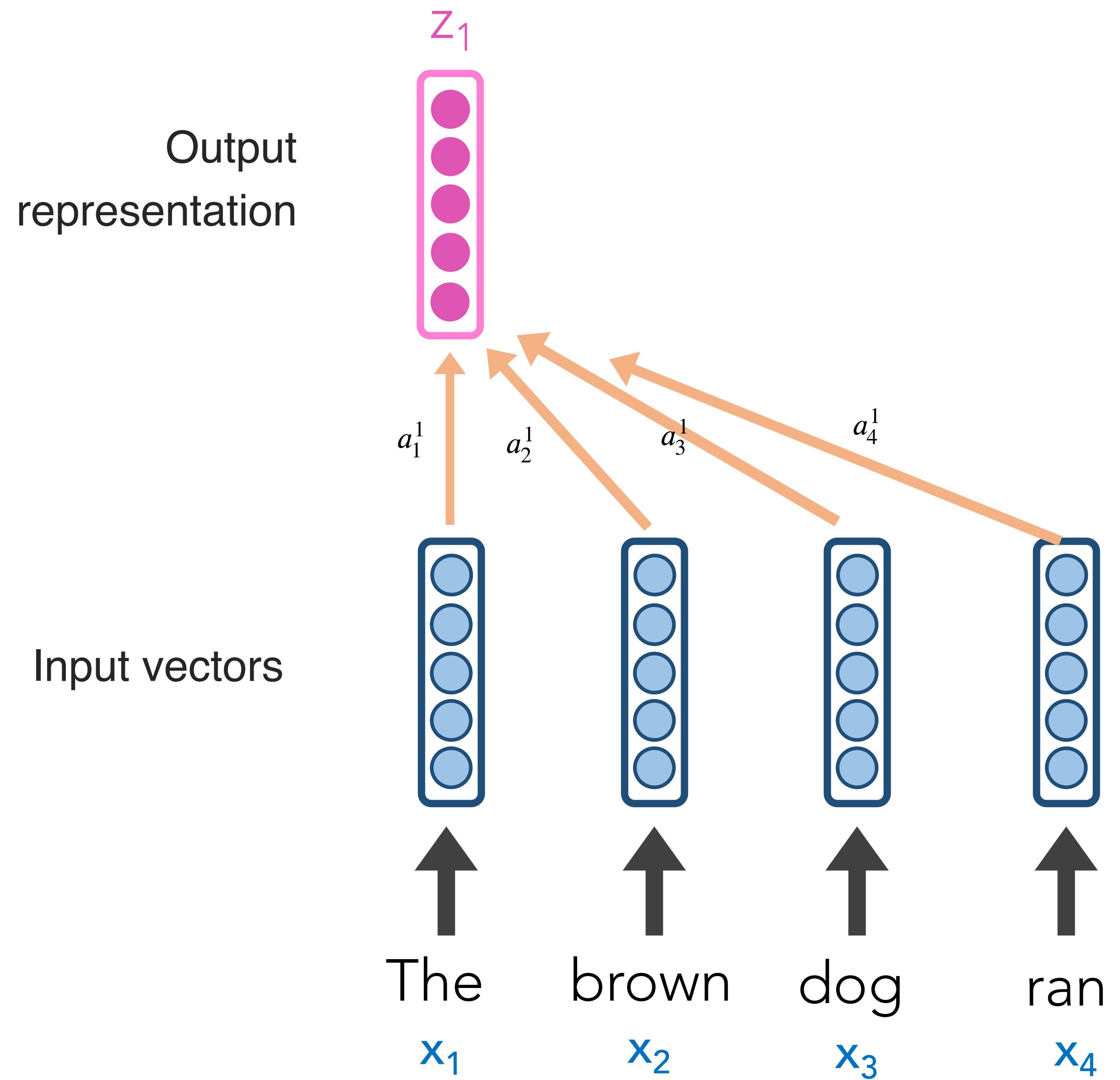
# Self Attention Motivation



Self-Attention's goal is to create great representations,  $z_i$ , of the input

$z_1$  will be based on a weighted contribution of  $x_1$ ,  $x_2$ ,  $x_3$ ,  $x_4$

# Self Attention Motivation

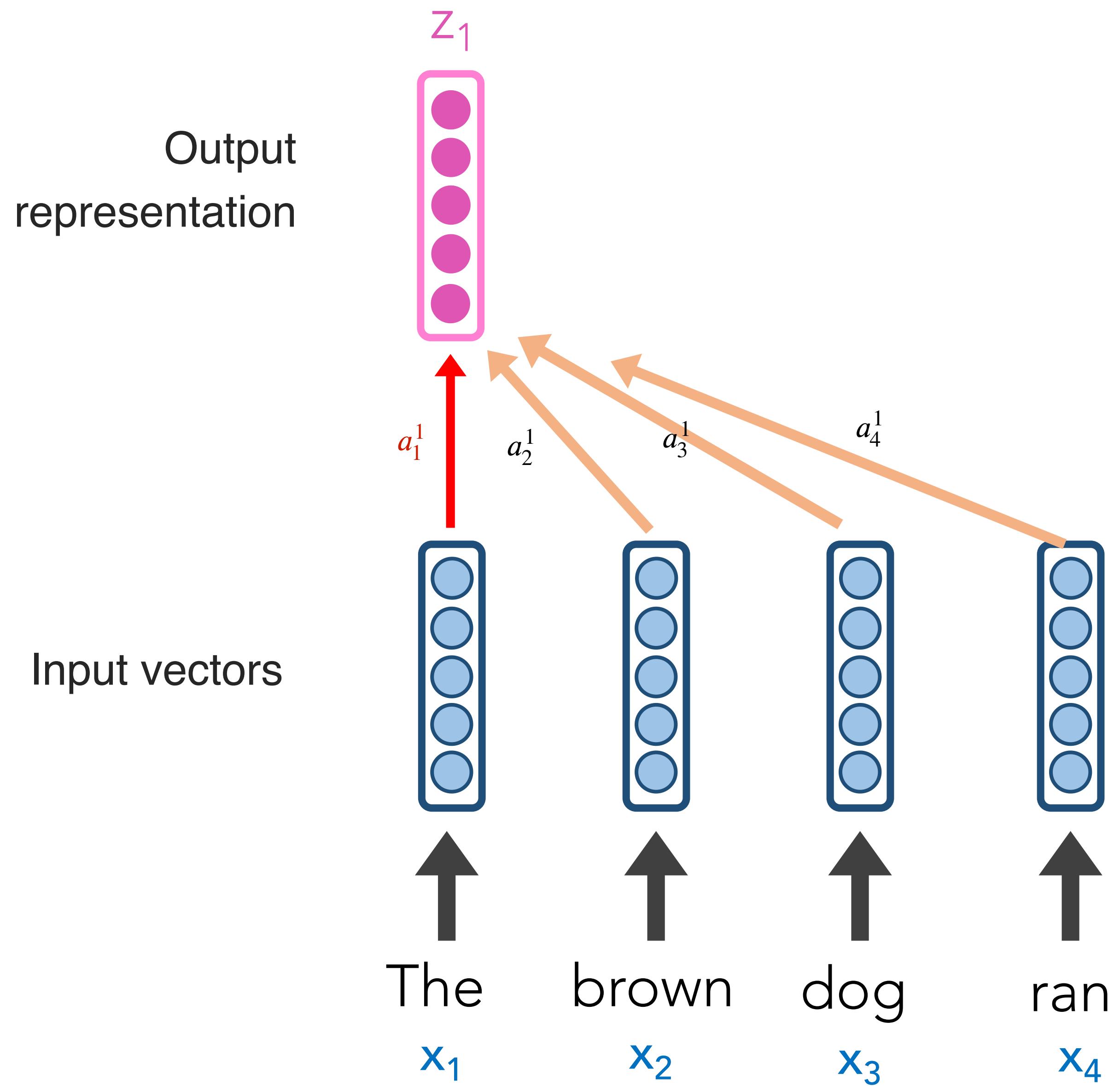


Self-Attention's goal is to create great representations,  $z_i$ , of the input

$z_1$  will be based on a weighted contribution of  $x_1, x_2, x_3, x_4$

$a_i^1$  is “just” a weight. More is happening under the hood, but it’s effectively weighting versions of  $x_1, x_2, x_3, x_4$

# Self Attention Motivation



Under the hood, each  $x_i$  has 3 small, associated vectors.

For example,  $x_1$  has:

- Query  $q_i$
- Key  $k_i$
- Value  $v_i$

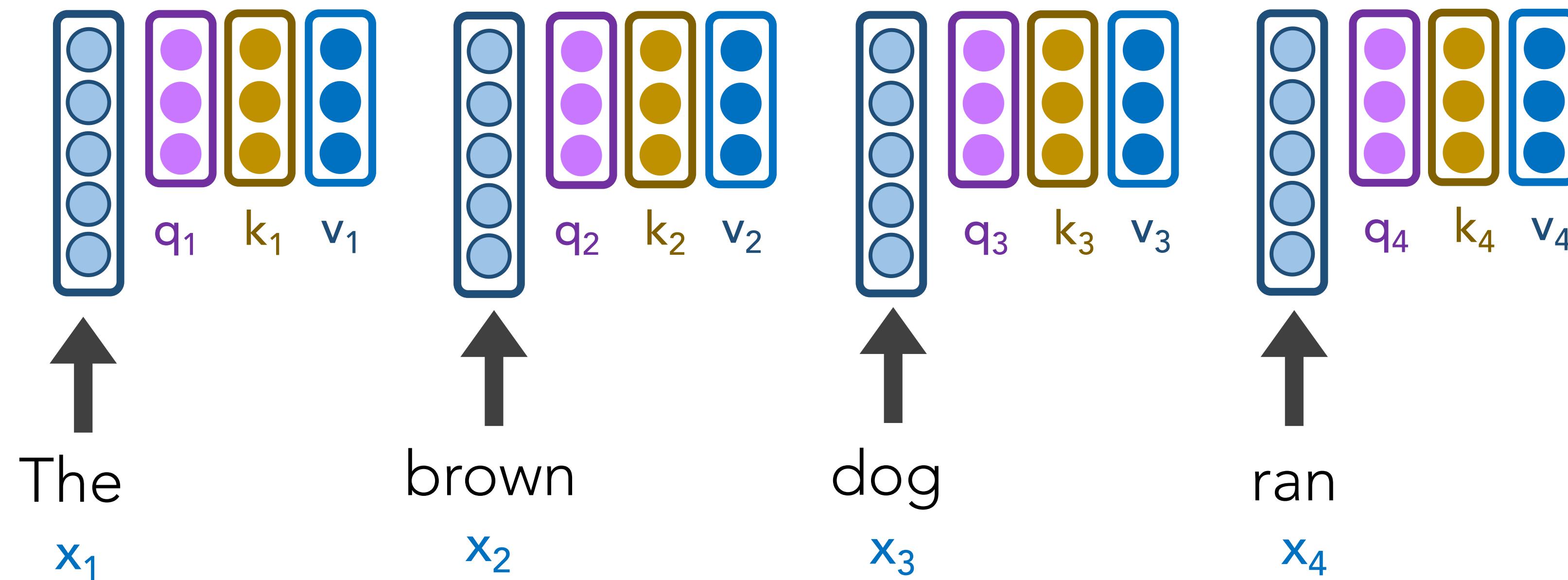
# Self-Attention Architecture

Step 1: Our Self-Attention Head has just 3 weight matrices  $W_q$ ,  $W_k$ ,  $W_v$  in total. These same 3 weight matrices are multiplied by each  $x_i$  to create all vectors:

$$q_i = W_q x_i$$

$$k_i = W_k x_i$$

$$v_i = W_v x_i$$



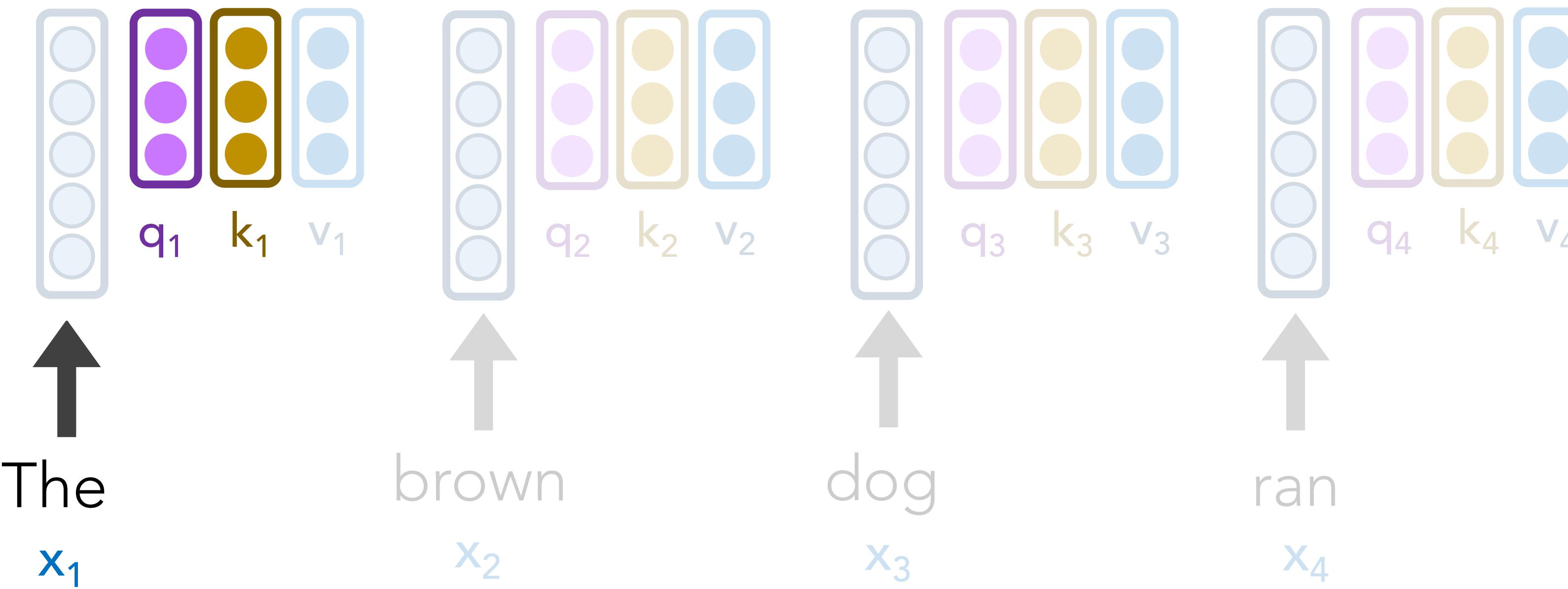
Under the hood, each  $x_i$  has 3 small, associated vectors. For example,  $x_1$  has:

- Query  $q_1$
- Key  $k_1$
- Value  $v_1$

# Self-Attention Architecture

Step 2: For word  $x_1$ , let's calculate the scores  $s_1, s_2, s_3, s_4$ , which represent how much attention to pay to each respective "word"  $v_i$

$$s_1 = q_1 \cdot k_1 = 112$$

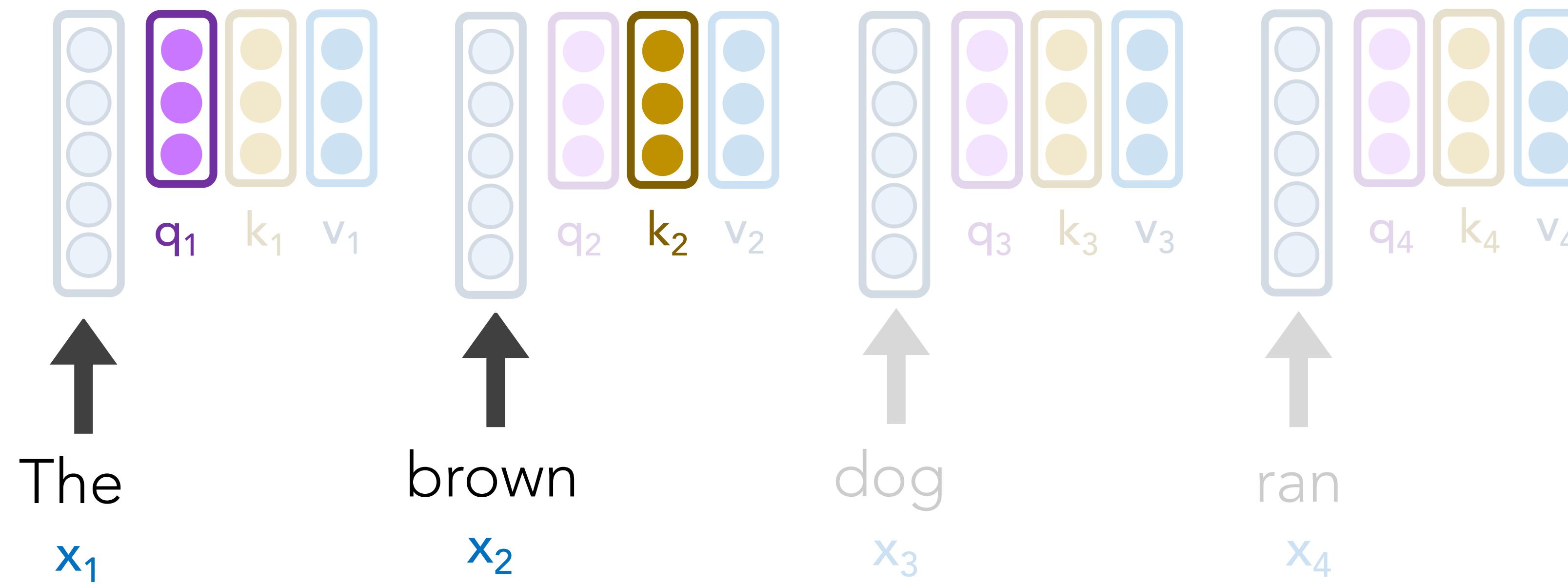


# Self-Attention Architecture

Step 2: For word  $x_1$ , let's calculate the scores  $s_1, s_2, s_3, s_4$ , which represent how much attention to pay to each respective "word"  $v_i$

$$s_2 = q_1 \cdot k_2 = 96$$

$$s_1 = q_1 \cdot k_1 = 112$$



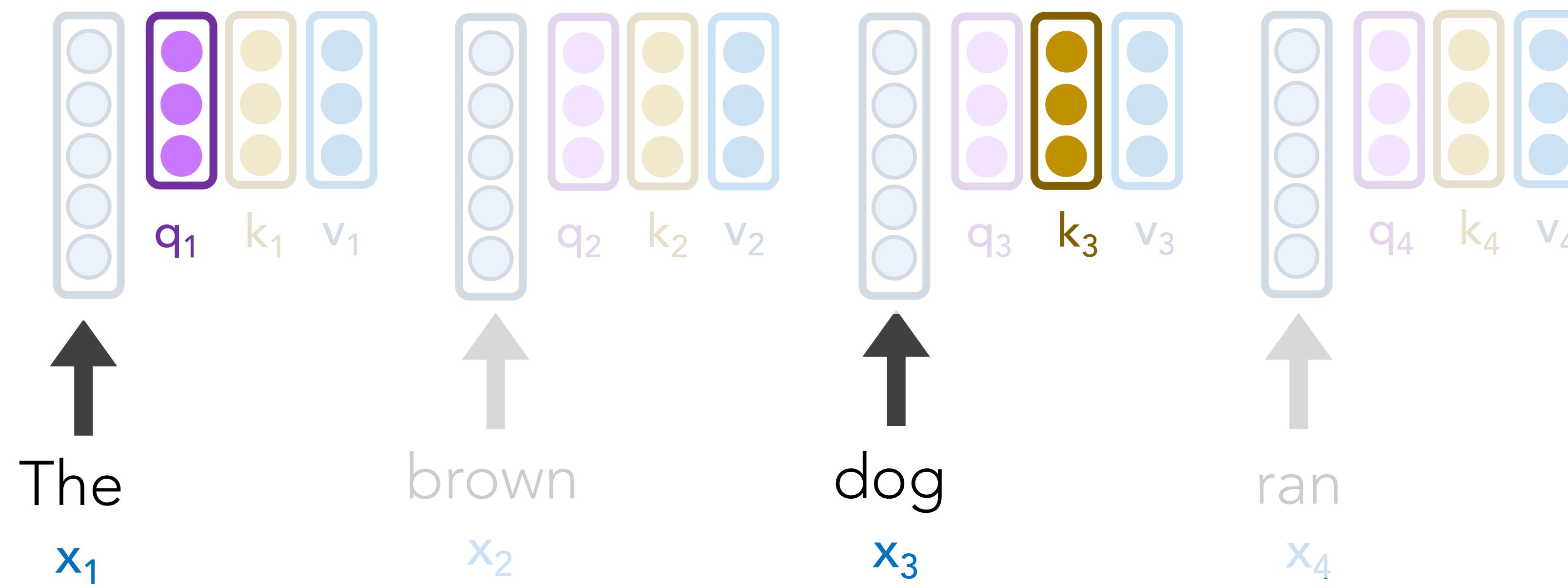
# Self-Attention Architecture

Step 2: For word  $x_1$ , let's calculate the scores  $s_1, s_2, s_3, s_4$ , which represent how much attention to pay to each respective "word"  $v_i$

$$s_3 = q_1 \cdot k_3 = 16$$

$$s_2 = q_1 \cdot k_2 = 96$$

$$s_1 = q_1 \cdot k_1 = 112$$



# Self-Attention Architecture

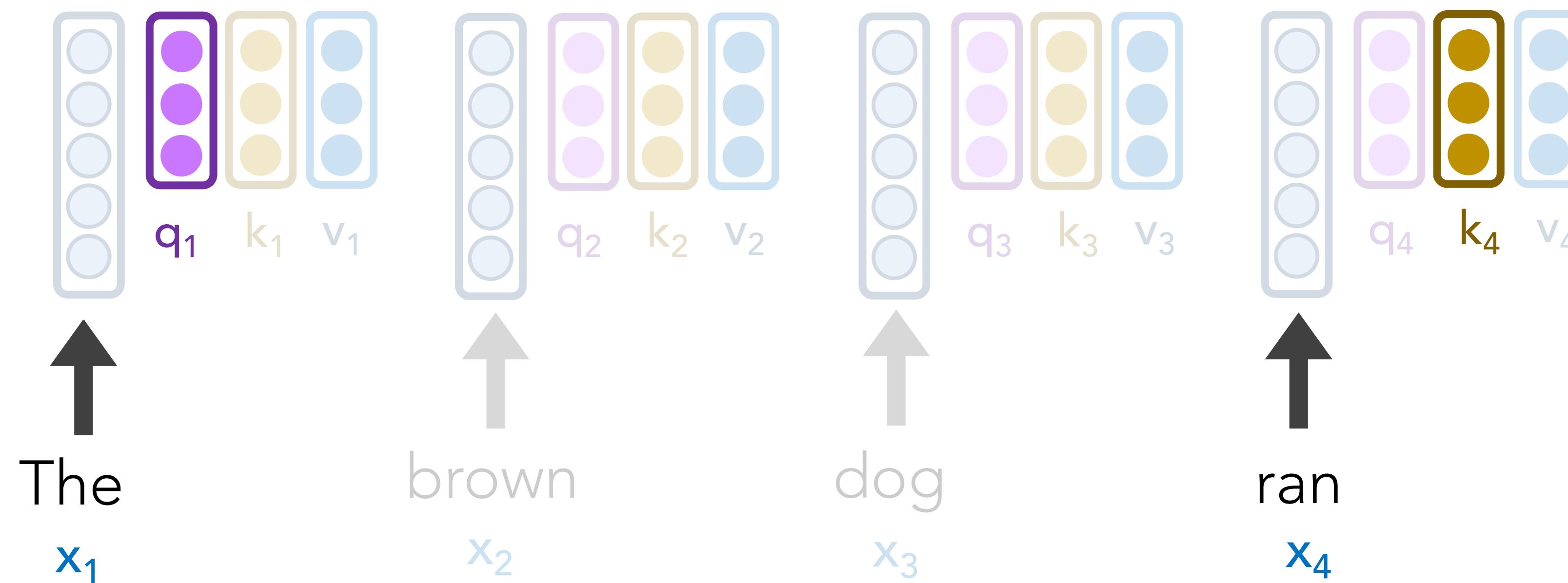
Step 2: For word  $x_1$ , let's calculate the scores  $s_1, s_2, s_3, s_4$ , which represent how much attention to pay to each respective "word"  $v_i$

$$s_4 = q_1 \cdot k_4 = 8$$

$$s_3 = q_1 \cdot k_3 = 16$$

$$s_2 = q_1 \cdot k_2 = 96$$

$$s_1 = q_1 \cdot k_1 = 112$$



# Self-Attention Architecture

Step 3: Our scores  $s_1, s_2, s_3, s_4$  don't sum to 1. Let's divide by  $\sqrt{\text{len}(k_i)}$  and **softmax** it

$$s_4 = q_1 \cdot k_4 = 8$$

$$a_4 = \sigma(s_4/8) = 0$$

$$s_3 = q_1 \cdot k_3 = 16$$

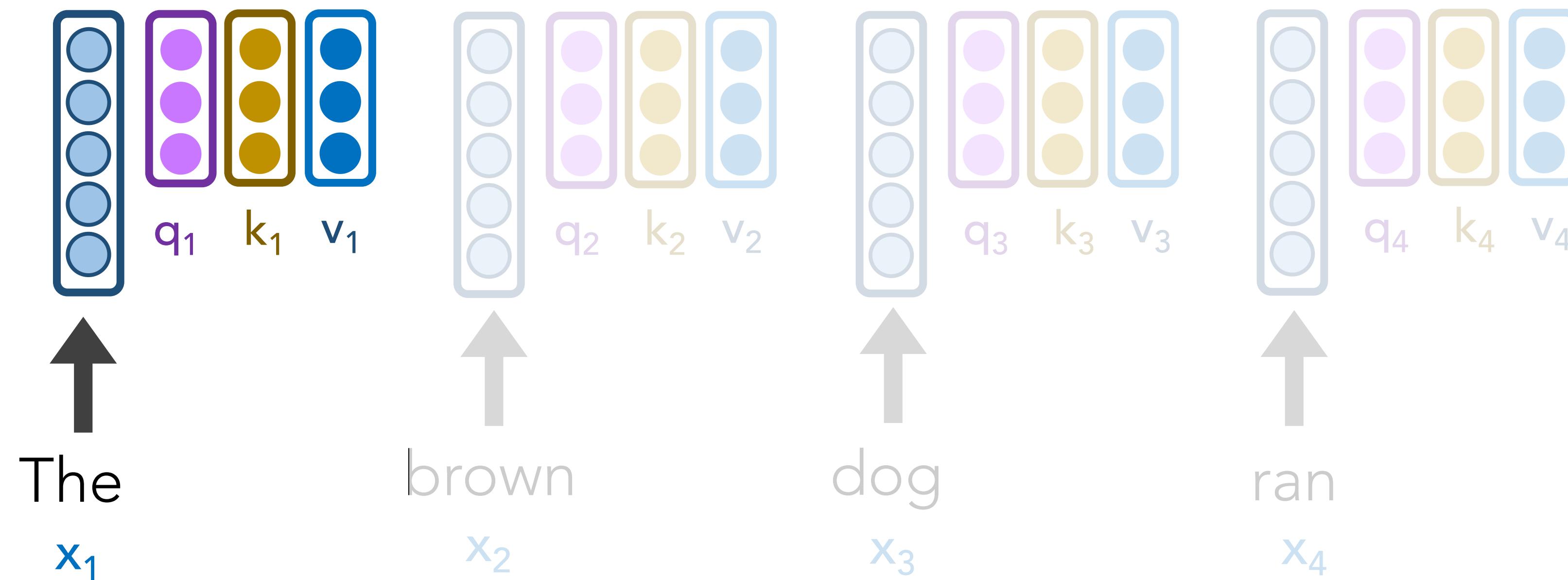
$$a_3 = \sigma(s_3/8) = .01$$

$$s_2 = q_1 \cdot k_2 = 96$$

$$a_2 = \sigma(s_2/8) = .12$$

$$s_1 = q_1 \cdot k_1 = 112$$

$$a_1 = \sigma(s_1/8) = .87$$



# Self-Attention Architecture

Step 3: Our scores  $s_1, s_2, s_3, s_4$  don't sum to 1. Let's divide by  $\sqrt{\text{len}(k_i)}$  and **softmax** it

$$s_4 = q_1 \cdot k_4 = 8$$

$$a_4 = \sigma(s_4/8) = 0$$

$$s_3 = q_1 \cdot k_3 = 16$$

$$a_3 = \sigma(s_3/8) = .01$$

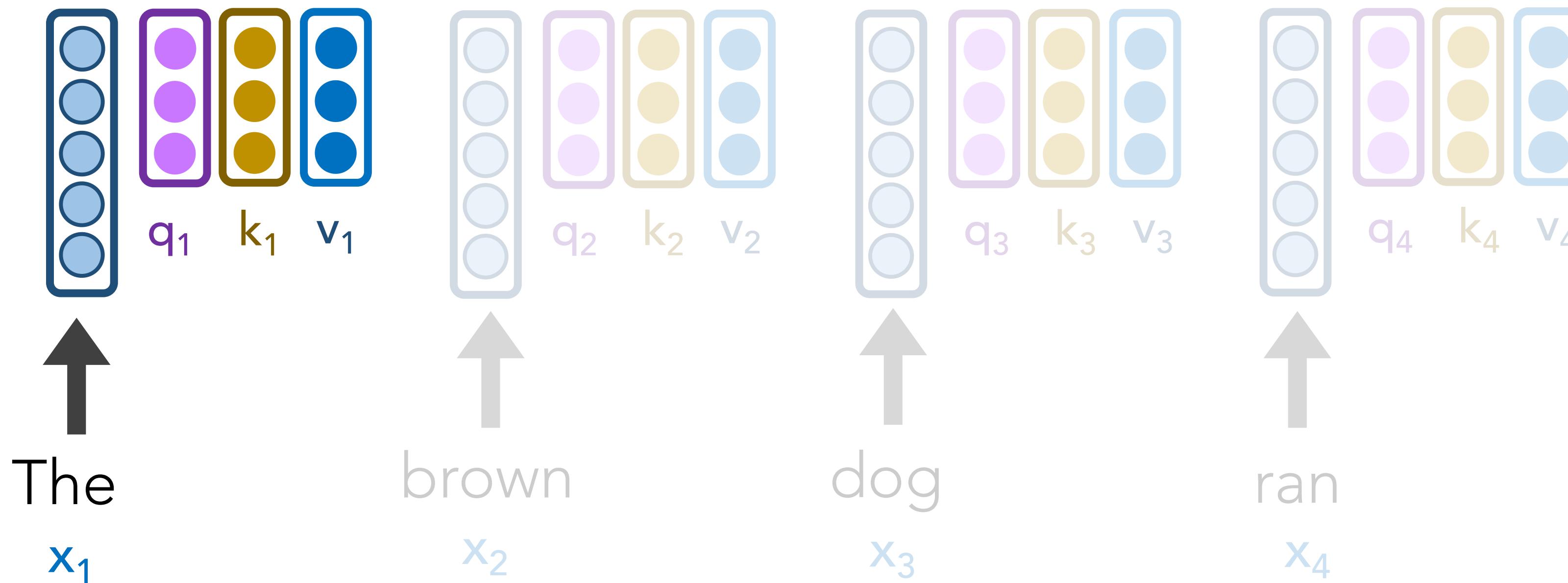
$$s_2 = q_1 \cdot k_2 = 96$$

$$a_2 = \sigma(s_2/8) = .12$$

$$s_1 = q_1 \cdot k_1 = 112$$

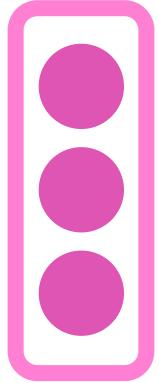
$$a_1 = \sigma(s_1/8) = .87$$

Instead of these  $a_i$  values directly weighting our original  $x_i$  word vectors, they weight our  $v_i$  vectors.

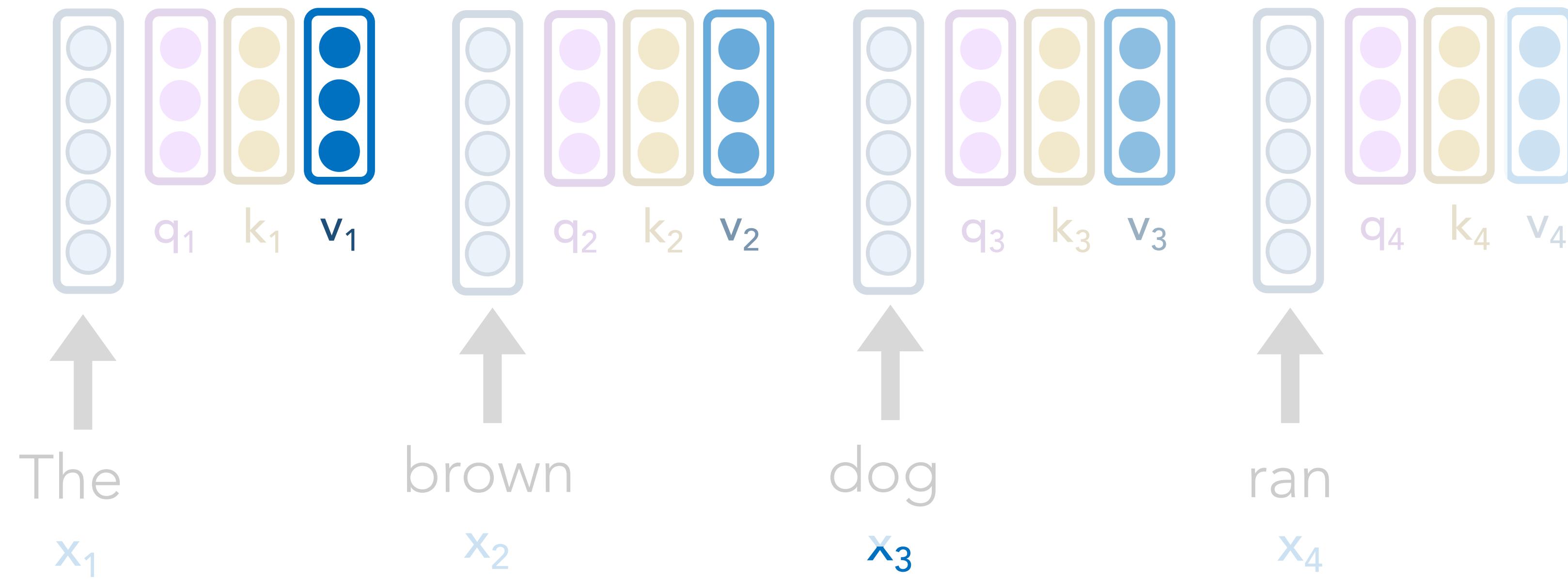


# Self-Attention Architecture

Step 4: Let's weight our  $v_i$  vectors and simply sum them up!

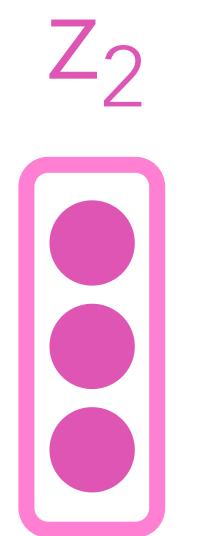
$$z_1$$


$$\begin{aligned} z_1 &= \mathbf{a}_1 \cdot v_1 + \mathbf{a}_2 \cdot v_2 + \mathbf{a}_3 \cdot v_3 + \mathbf{a}_4 \cdot v_4 \\ &= 0.87 \cdot v_1 + 0.12 \cdot v_2 + 0.01 \cdot v_3 + 0 \cdot v_4 \end{aligned}$$

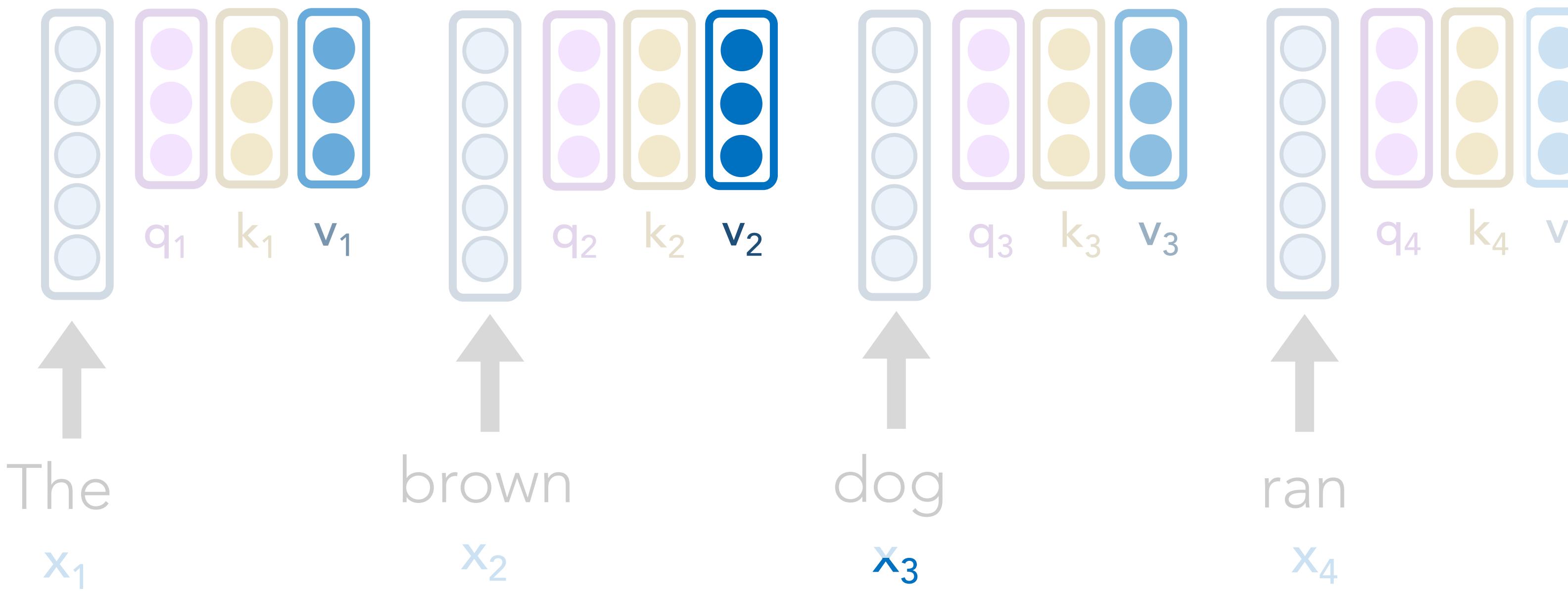


# Self-Attention Architecture

Step 5: We repeat this for all other words, yielding us with great, new  $z_i$  representations!

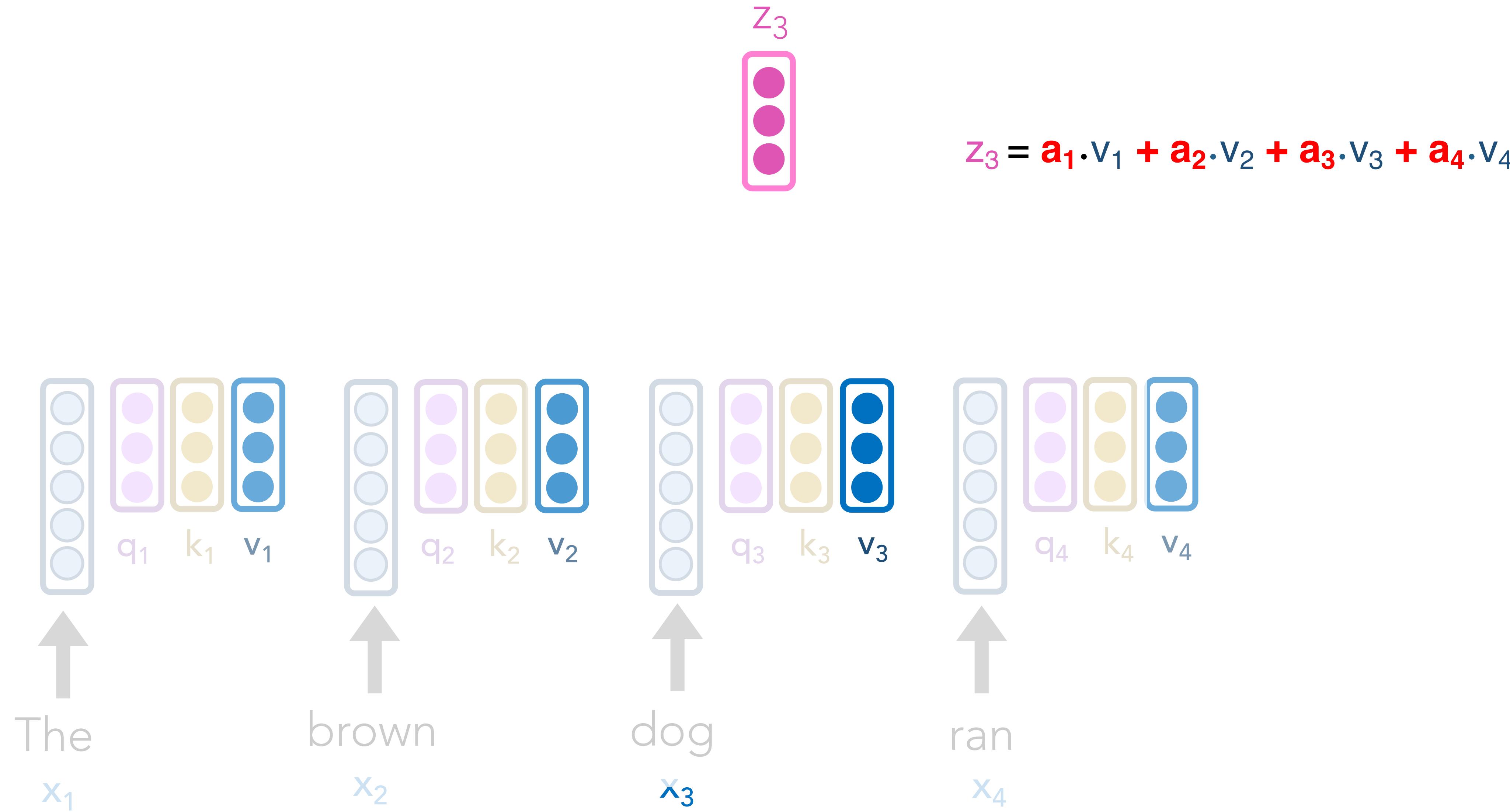


$$z_2 = \mathbf{a}_1 \cdot \mathbf{v}_1 + \mathbf{a}_2 \cdot \mathbf{v}_2 + \mathbf{a}_3 \cdot \mathbf{v}_3 + \mathbf{a}_4 \cdot \mathbf{v}_4$$



# Self-Attention Architecture

Step 5: We repeat this for all other words, yielding us with great, new  $z_i$  representations!

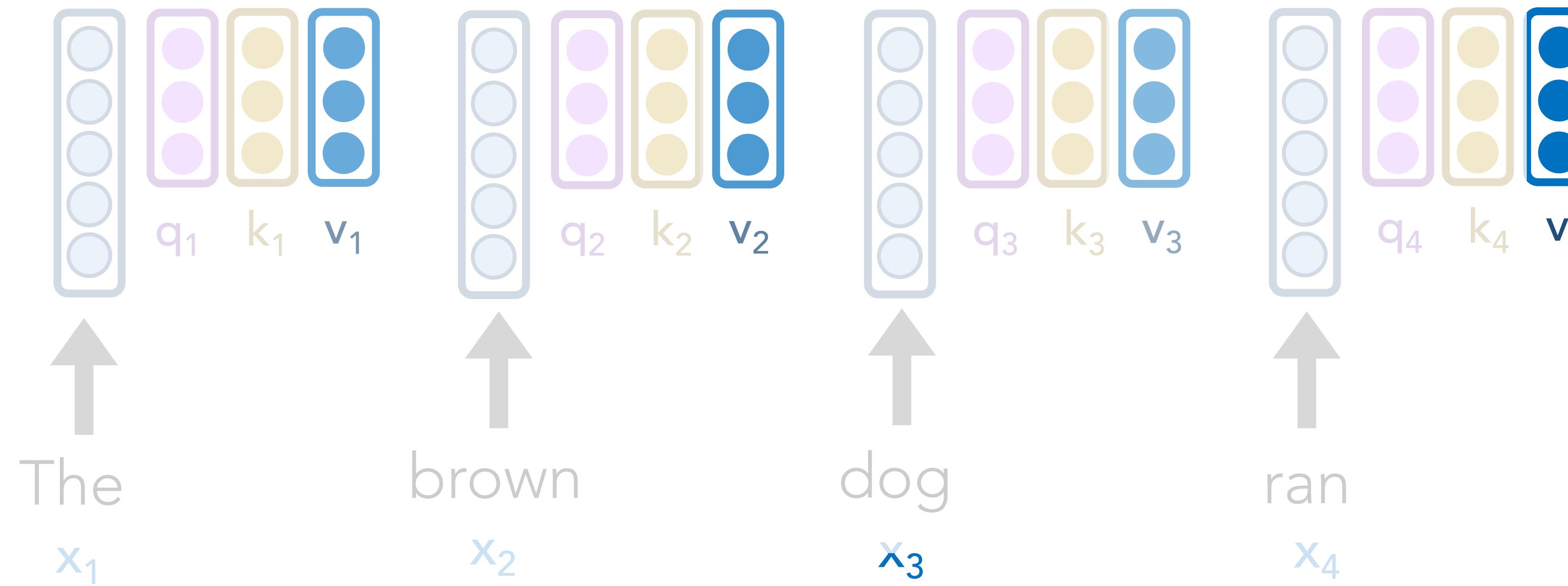


# Self-Attention Architecture

Step 5: We repeat this for all other words, yielding great, new  $z_i$  representations!

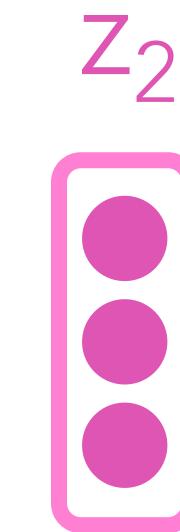


$$z_4 = \mathbf{a}_1 \cdot \mathbf{v}_1 + \mathbf{a}_2 \cdot \mathbf{v}_2 + \mathbf{a}_3 \cdot \mathbf{v}_3 + \mathbf{a}_4 \cdot \mathbf{v}_4$$



# Self-Attention Architecture

Let's illustrate another example:



$$z_2 = \mathbf{a}_1 \cdot \mathbf{v}_1 + \mathbf{a}_2 \cdot \mathbf{v}_2 + \mathbf{a}_3 \cdot \mathbf{v}_3 + \mathbf{a}_4 \cdot \mathbf{v}_4$$

Remember, we use the same 3 weight matrices

$\mathbf{W}_q$ ,  $\mathbf{W}_k$ ,  $\mathbf{W}_v$  as we did for computing  $z_1$ .

This gives us  $q_2$ ,  $k_2$ ,  $v_2$

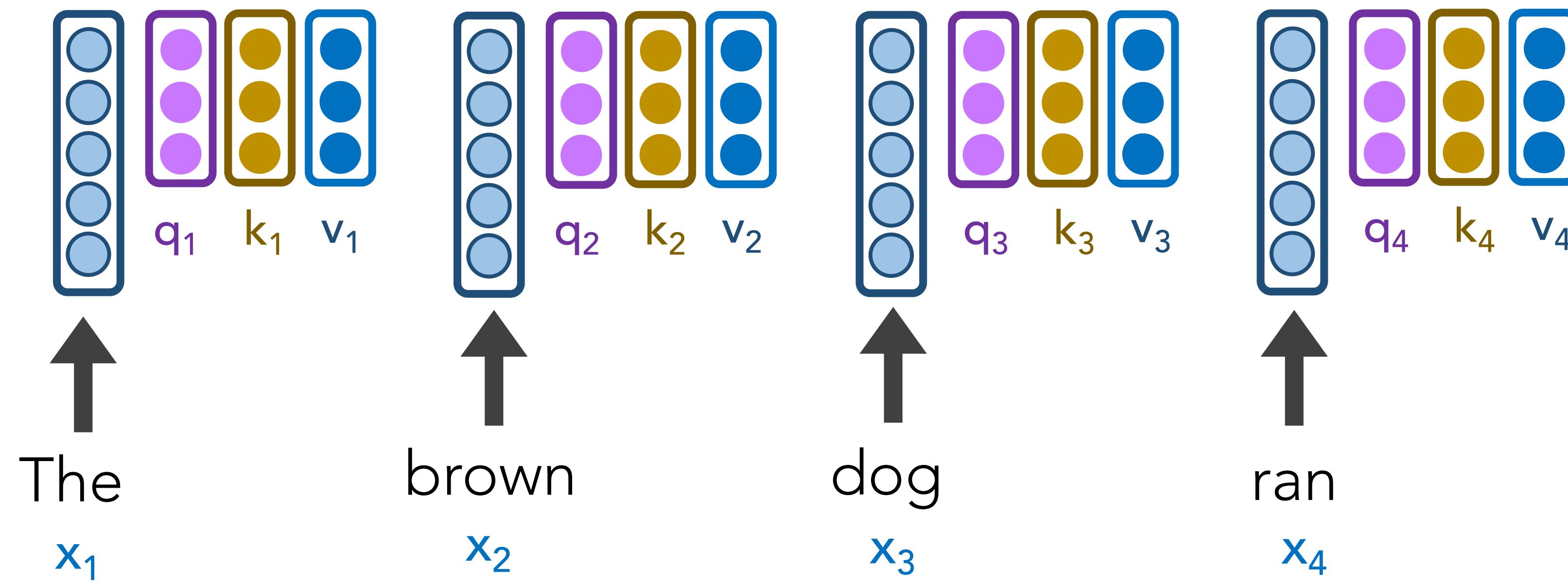
# Self-Attention Architecture

Step 1: Our Self-Attention Head I has just 3 weight matrices  $W_q$ ,  $W_k$ ,  $W_v$  in total. These same 3 weight matrices are multiplied by each  $x_i$  to create all vectors:

$$q_i = W_q x_i$$

$$k_i = W_k x_i$$

$$v_i = W_v x_i$$

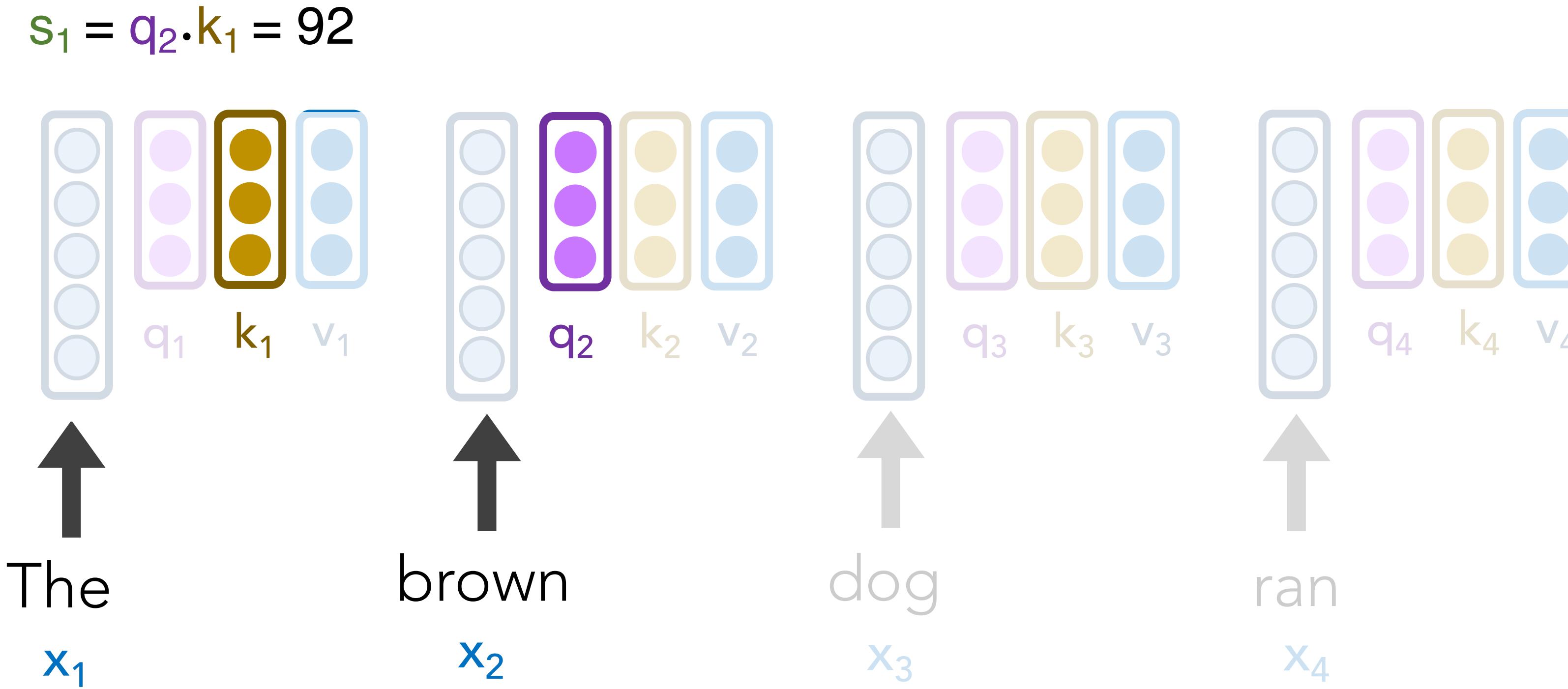


Under the hood, each  $x_i$  has 3 small, associated vectors. For example,  $x_1$  has:

- Query  $q_1$
- Key  $k_1$
- Value  $v_1$

# Self-Attention Architecture

Step 2: For word  $x_2$ , let's calculate the scores  $s_1, s_2, s_3, s_4$ , which represent how much attention to pay to each respective "word"  $v_i$

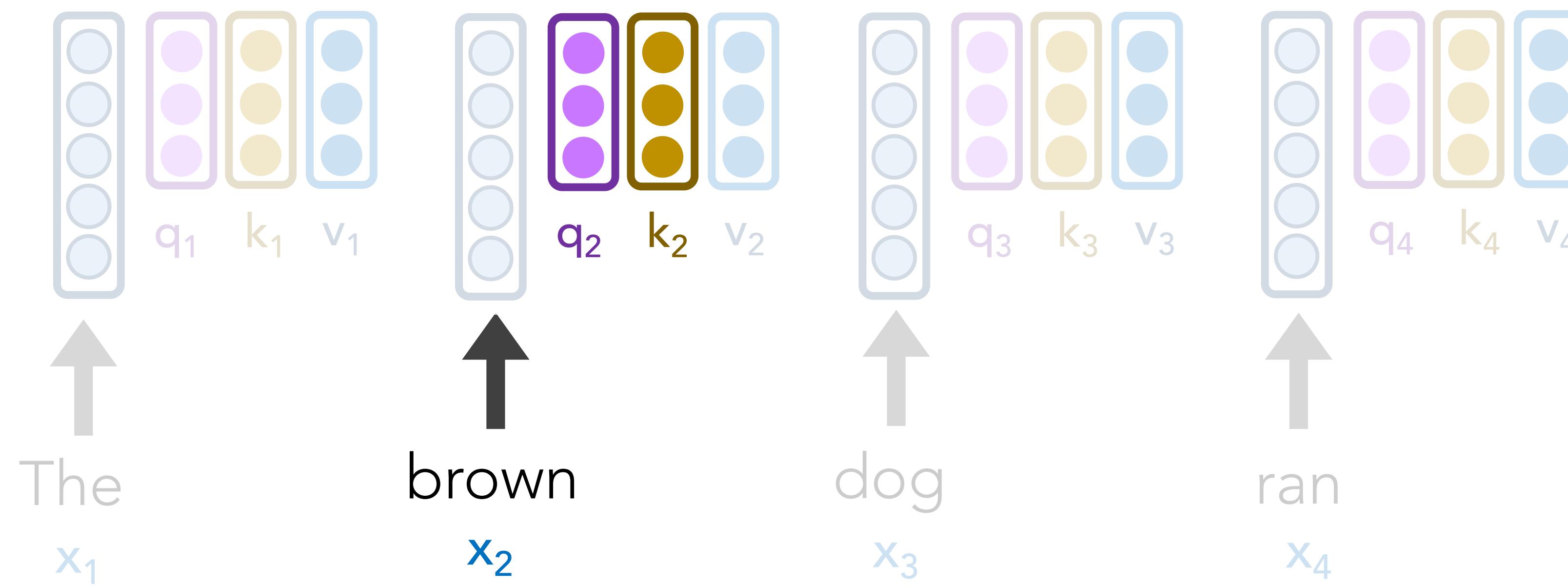


# Self-Attention Architecture

Step 2: For word  $x_2$ , let's calculate the scores  $s_1, s_2, s_3, s_4$ , which represent how much attention to pay to each respective "word"  $v_i$

$$s_2 = q_2 \cdot k_2 = 124$$

$$s_1 = q_2 \cdot k_1 = 92$$



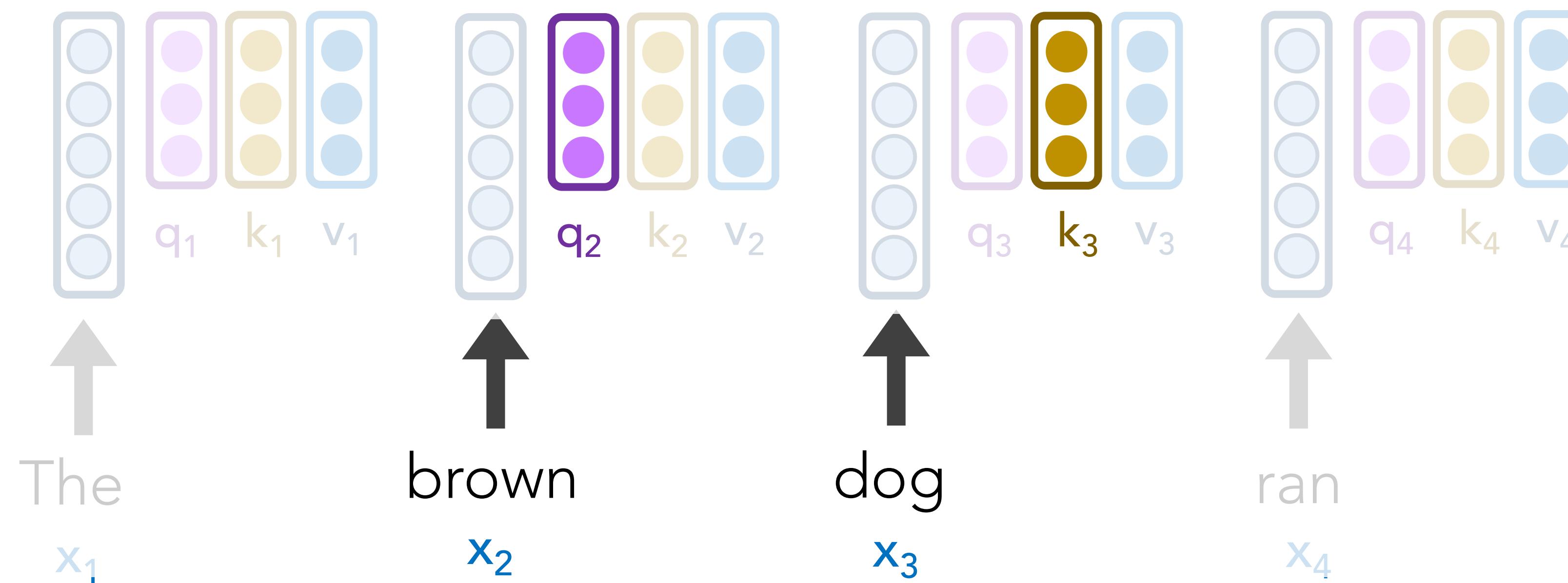
# Self-Attention Architecture

Step 2: For word  $x_2$ , let's calculate the scores  $s_1, s_2, s_3, s_4$ , which represent how much attention to pay to each respective "word"  $v_i$

$$s_3 = q_2 \cdot k_3 = 22$$

$$s_2 = q_2 \cdot k_2 = 124$$

$$s_1 = q_2 \cdot k_1 = 92$$



# Self-Attention Architecture

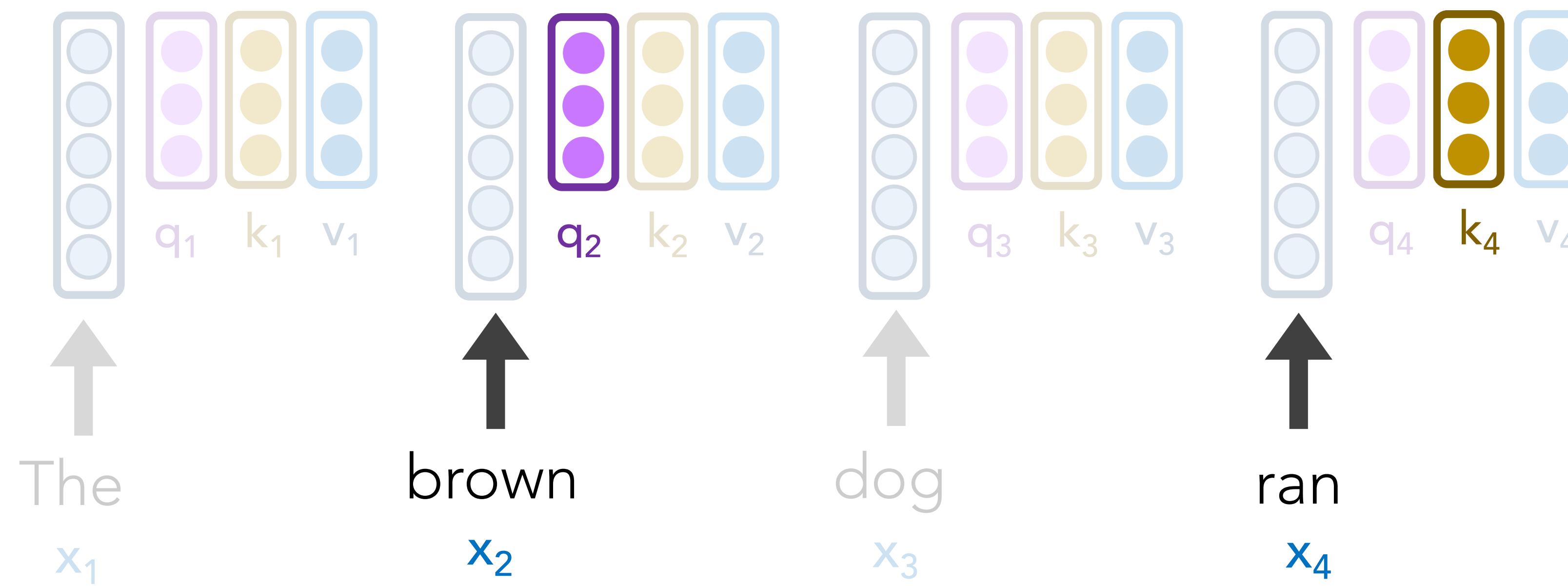
Step 2: For word  $x_2$ , let's calculate the scores  $s_1, s_2, s_3, s_4$ , which represent how much attention to pay to each respective "word"  $v_i$

$$s_4 = q_2 \cdot k_4 = 8$$

$$s_3 = q_2 \cdot k_3 = 22$$

$$s_2 = q_2 \cdot k_2 = 124$$

$$s_1 = q_2 \cdot k_1 = 92$$



# Self-Attention Architecture

Step 3: Our scores  $s_1, s_2, s_3, s_4$  don't sum to 1. Let's divide by  $\sqrt{\text{len}(k_i)}$  and **softmax** it

$$s_4 = q_2 \cdot k_4 = 8$$

$$a_4 = \sigma(s_4/8) = 0$$

$$s_3 = q_2 \cdot k_3 = 22$$

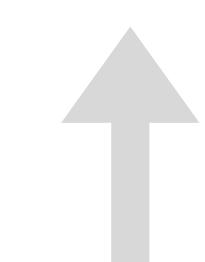
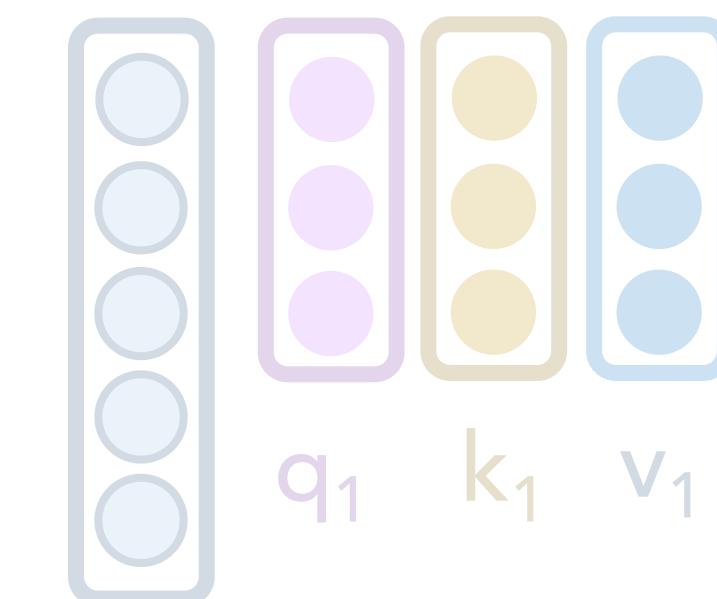
$$a_3 = \sigma(s_3/8) = .01$$

$$s_2 = q_2 \cdot k_2 = 124$$

$$a_2 = \sigma(s_2/8) = .91$$

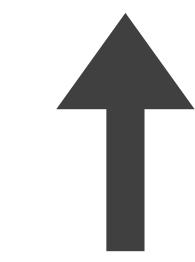
$$s_1 = q_2 \cdot k_1 = 92$$

$$a_1 = \sigma(s_1/8) = .08$$



The

$x_1$



brown

$x_2$



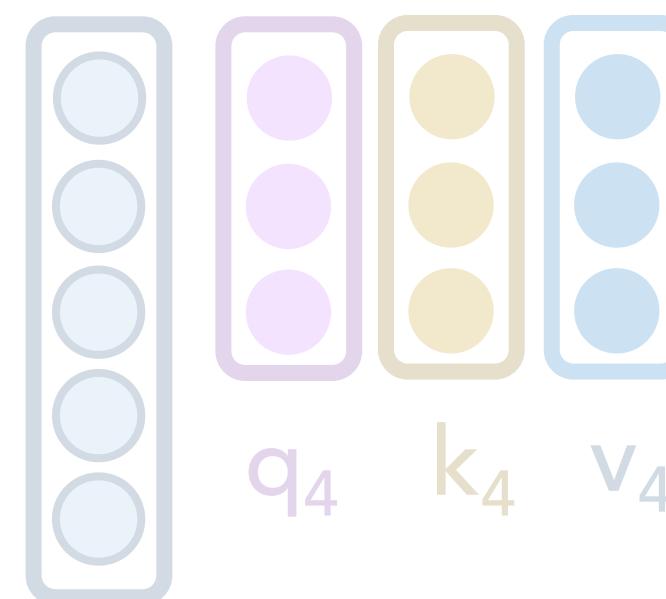
dog

$x_3$



ran

$x_4$



# Self-Attention Architecture

Step 3: Our scores  $s_1, s_2, s_3, s_4$  don't sum to 1. Let's divide by  $\sqrt{\text{len}(k_i)}$  and **softmax** it

$$s_4 = q_2 \cdot k_4 = 8$$

$$a_4 = \sigma(s_4/8) = 0$$

$$s_3 = q_2 \cdot k_3 = 22$$

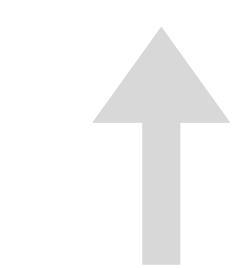
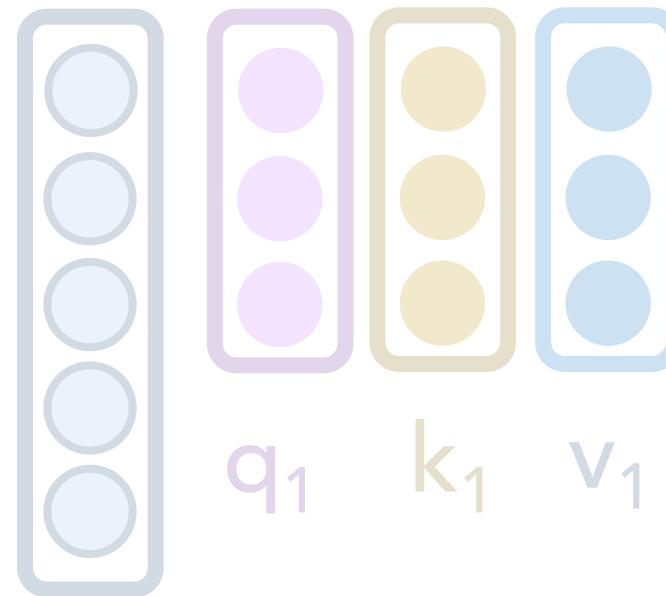
$$a_3 = \sigma(s_3/8) = .01$$

$$s_2 = q_2 \cdot k_2 = 124$$

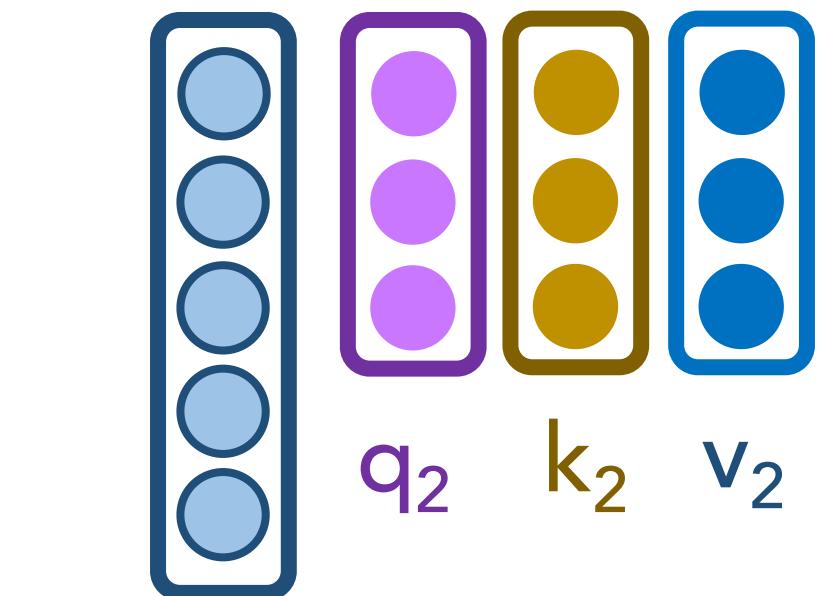
$$a_2 = \sigma(s_2/8) = .91$$

$$s_1 = q_2 \cdot k_1 = 92$$

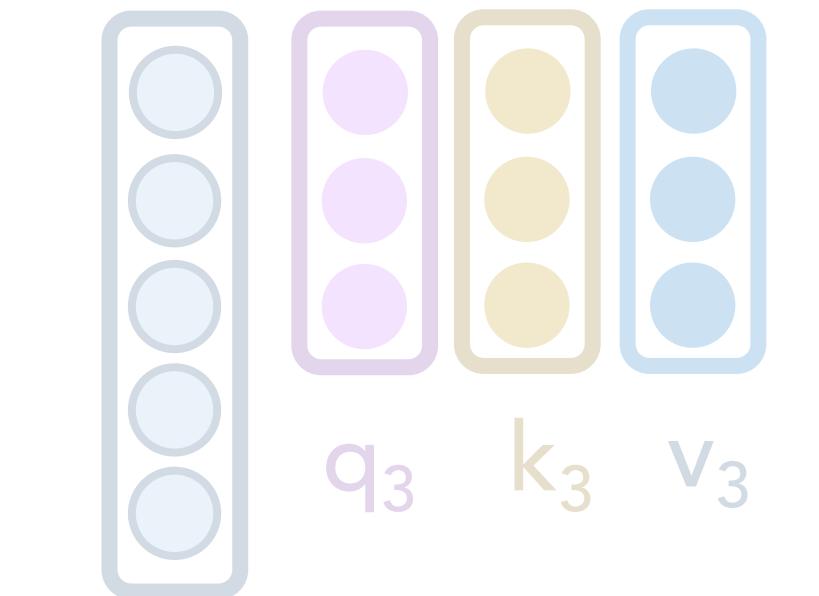
$$a_1 = \sigma(s_1/8) = .08$$



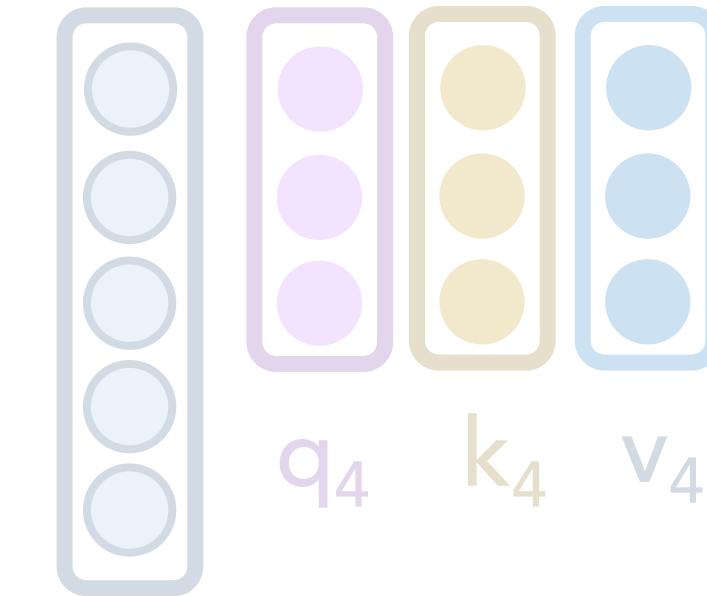
$x_1$   
The



brown  
 $x_2$



dog  
 $x_3$



ran  
 $x_4$

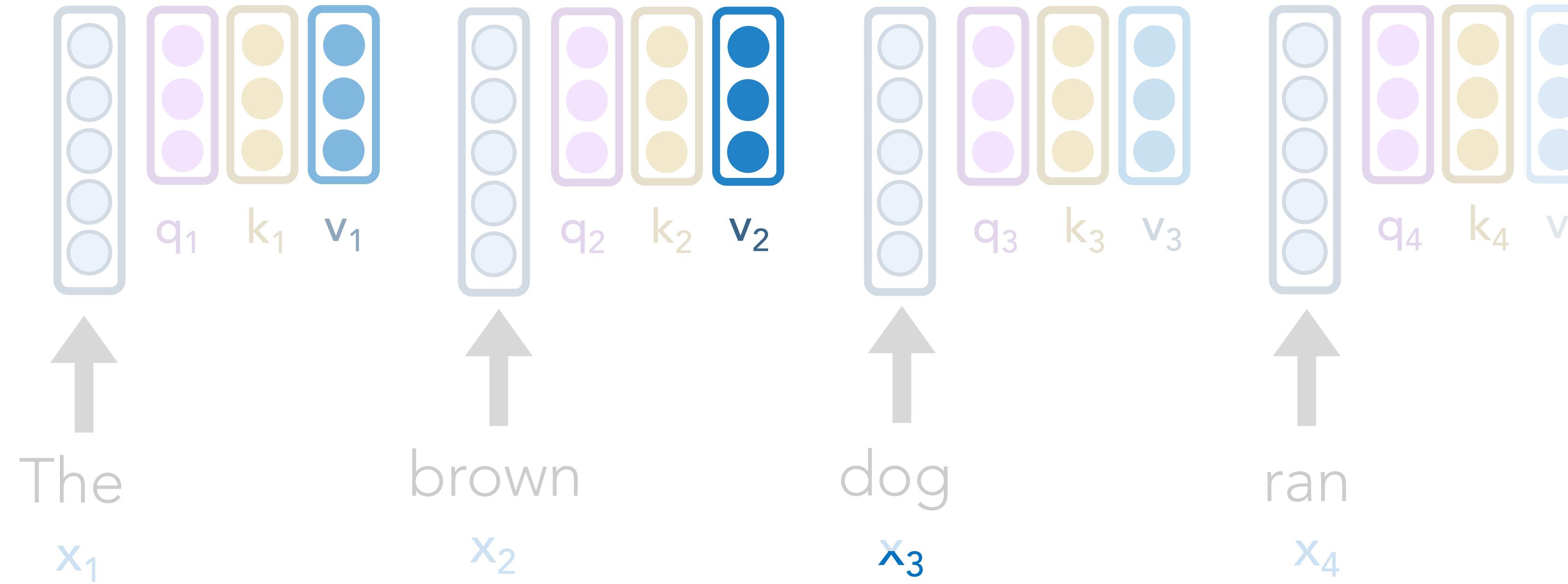
Instead of these  $a_i$  values directly weighting our original  $x_i$  word vectors, they directly weight our  $v_i$  vectors.

# Self-Attention Architecture

Step 4: Let's weight our  $v_i$  vectors and simply sum them up!

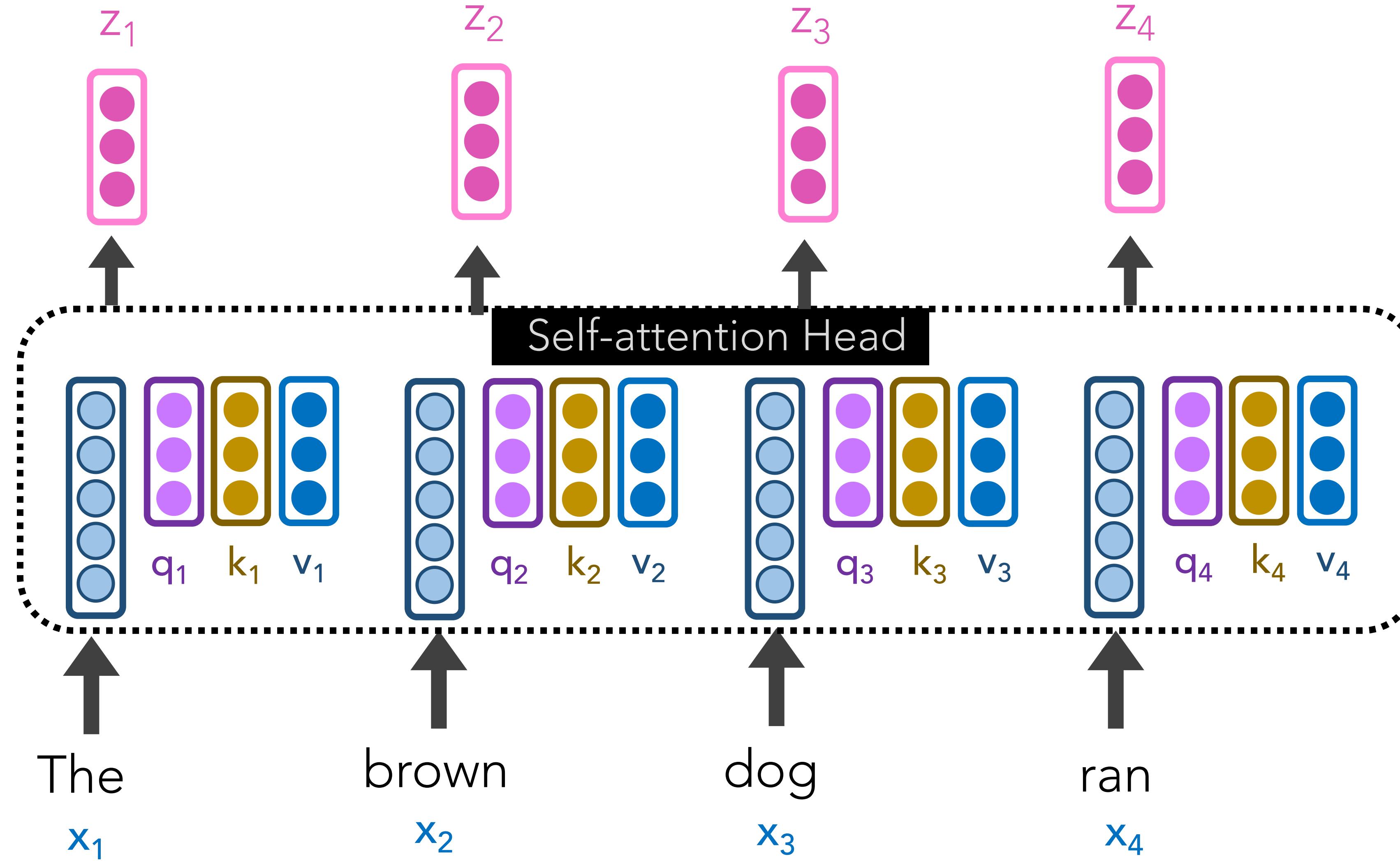
The diagram illustrates the calculation of  $z_2$  from four input vectors  $x_1, x_2, x_3, x_4$ . Each input vector is processed by three parallel linear projections: Query ( $q$ ), Key ( $k$ ), and Value ( $v$ ). The resulting query vectors  $q_1, q_2, q_3, q_4$  are shown as vertical stacks of circles in light blue, pink, yellow, and light blue respectively. The key vectors  $k_1, k_2, k_3, k_4$  are shown as vertical stacks of circles in light blue, pink, yellow, and light blue respectively. The value vectors  $v_1, v_2, v_3, v_4$  are shown as vertical stacks of circles in light blue, pink, yellow, and light blue respectively. An attention matrix  $A$  is applied to these vectors to produce the output  $z_2$ , which is represented as a vertical stack of three pink circles. The calculation is given by the equation:

$$z_2 = \mathbf{a}_1 \cdot \mathbf{v}_1 + \mathbf{a}_2 \cdot \mathbf{v}_2 + \mathbf{a}_3 \cdot \mathbf{v}_3 + \mathbf{a}_4 \cdot \mathbf{v}_4$$
$$= 0.08 \cdot \mathbf{v}_1 + 0.91 \cdot \mathbf{v}_2 + 0.01 \cdot \mathbf{v}_3 + 0 \cdot \mathbf{v}_4$$



# Self-Attention Architecture

Tada! Now we have great, new representations  $z_i$  via a self-attention head



# Self-Attention Summary

## Strengths

- Context-aware
- Much more powerful than static embeddings

## Issues?

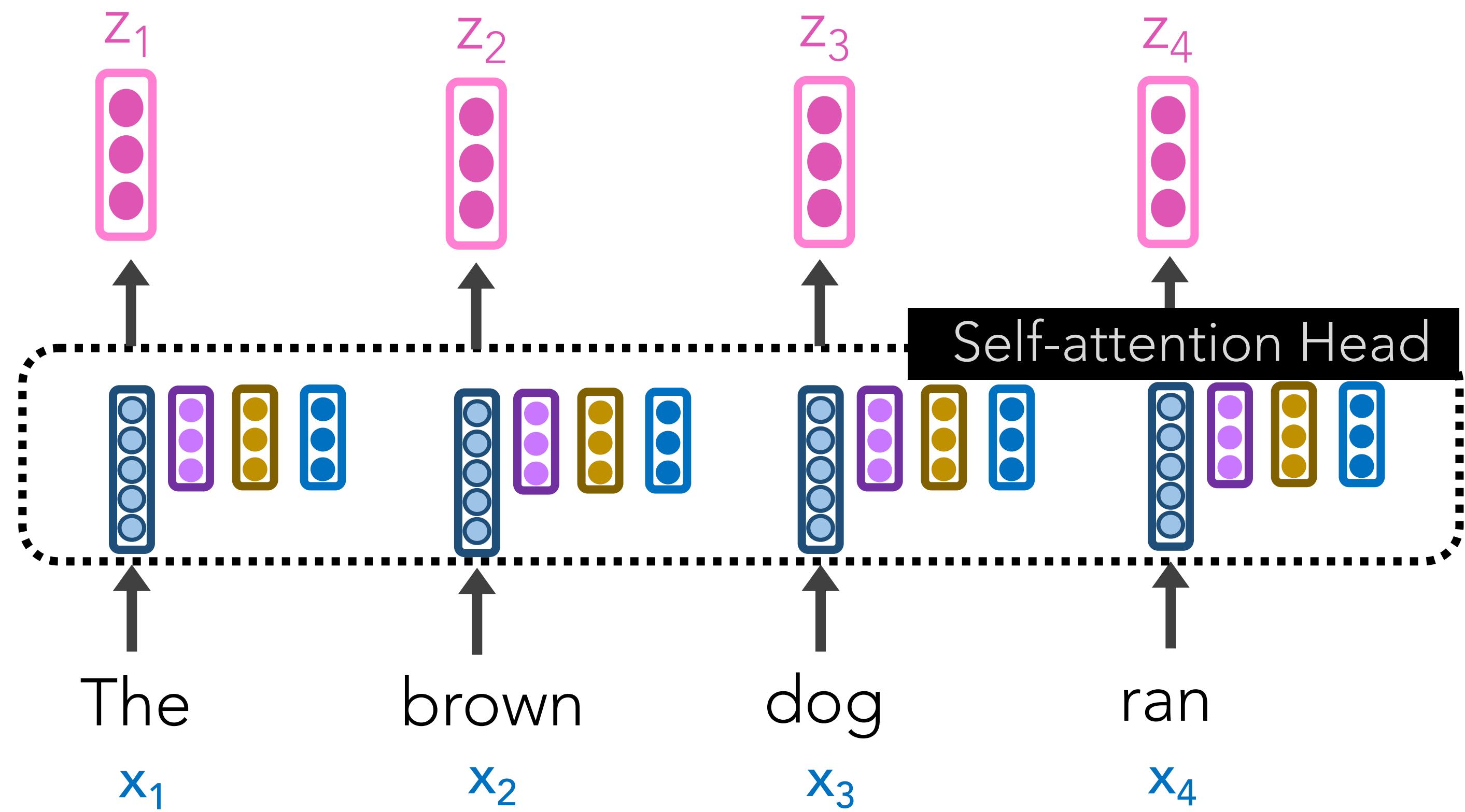
- Slower to train
- Cannot easily be pre-computed



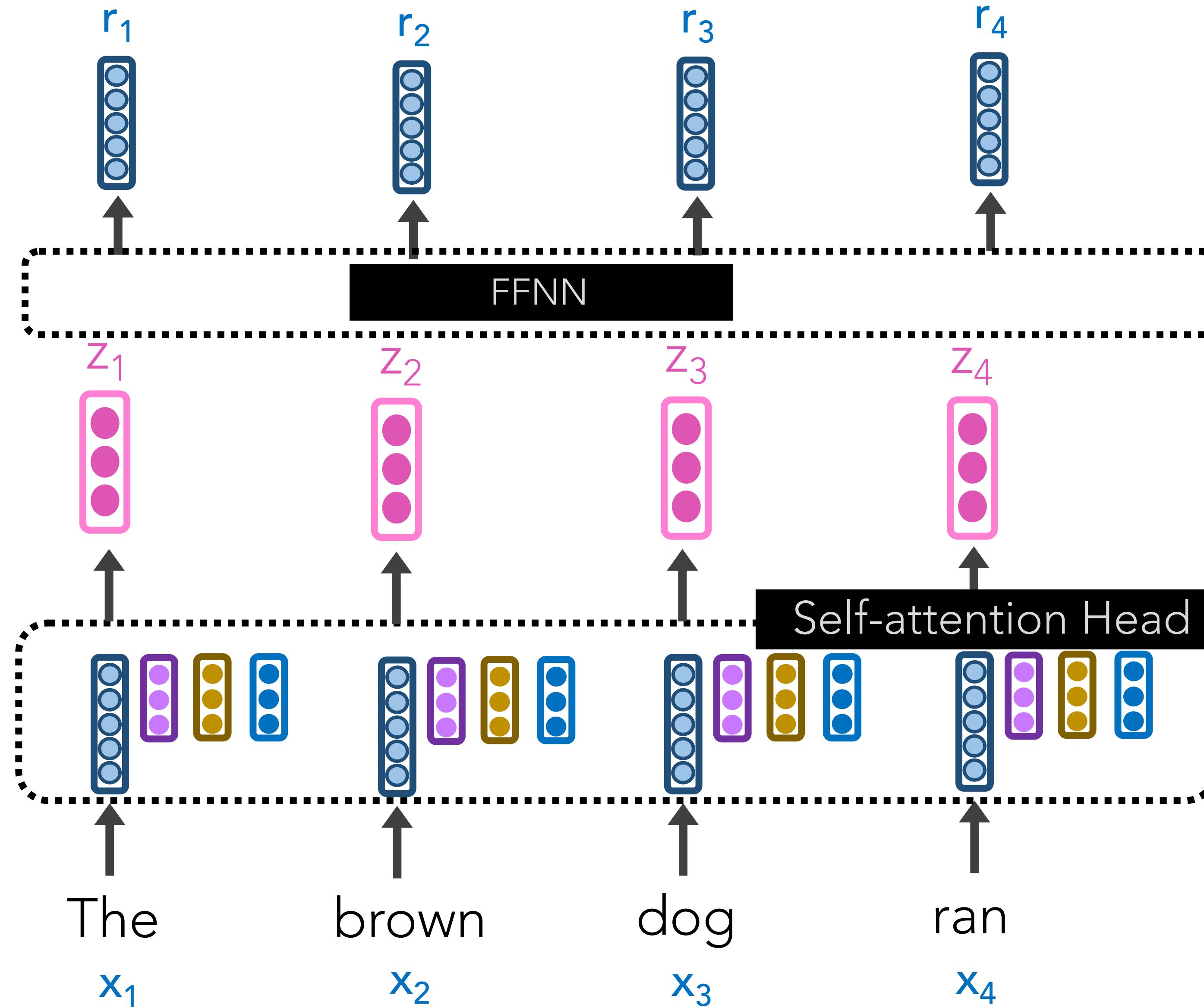
# Transformers

# Transformer Architecture

Let's further pass each  $z_i$  through a FFNN

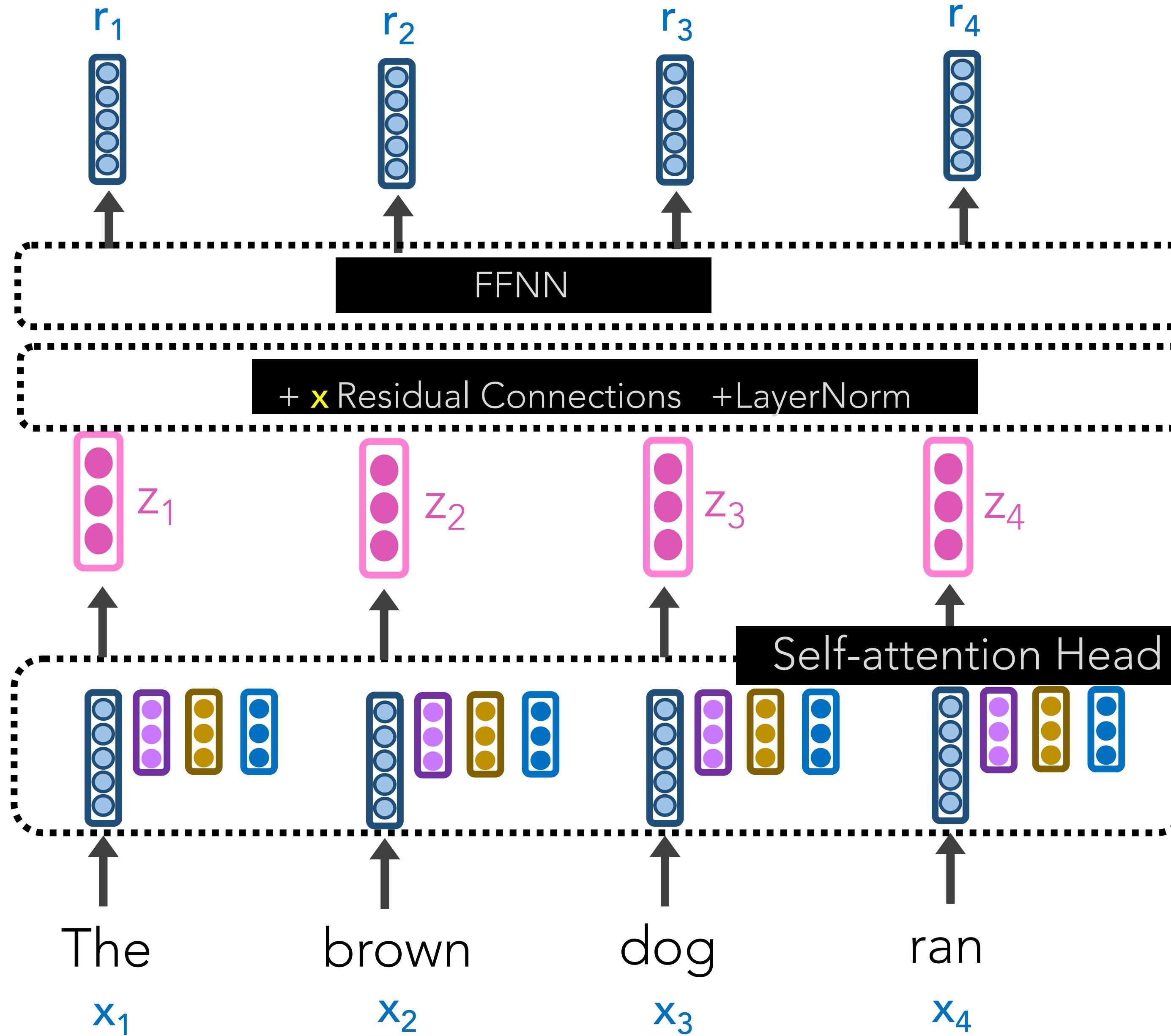


# Transformer Architecture



Let's further pass each  $z_i$  through a FFNN

# Transformer Architecture

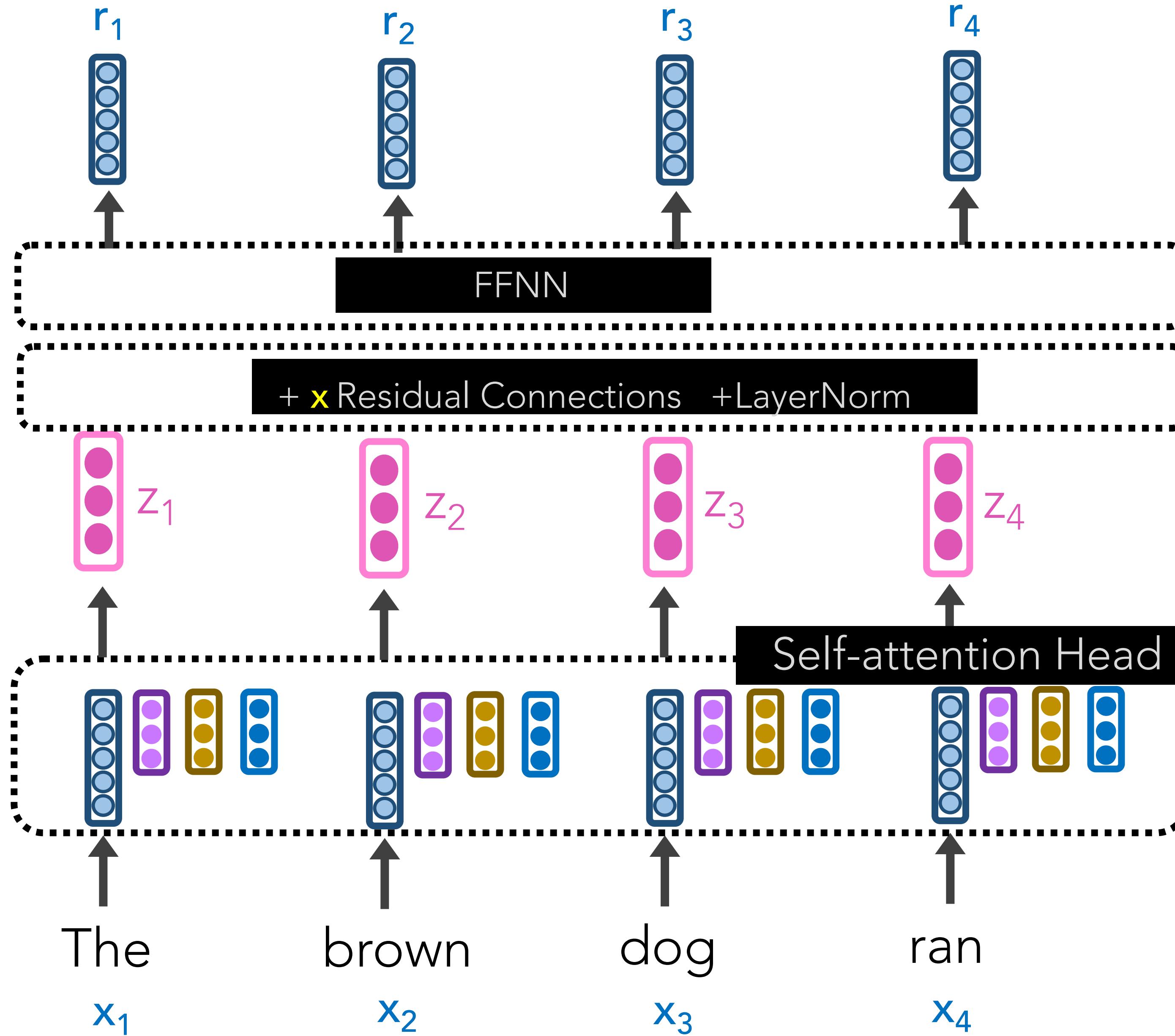


Let's further pass each  $z_i$  through a FFNN

We concat w/ a residual connection to help ensure relevant info is getting forward passed.

We perform LayerNorm to stabilize the network and allow for proper gradient flow. You should do this after the FFNN, too.

# Transformer Architecture



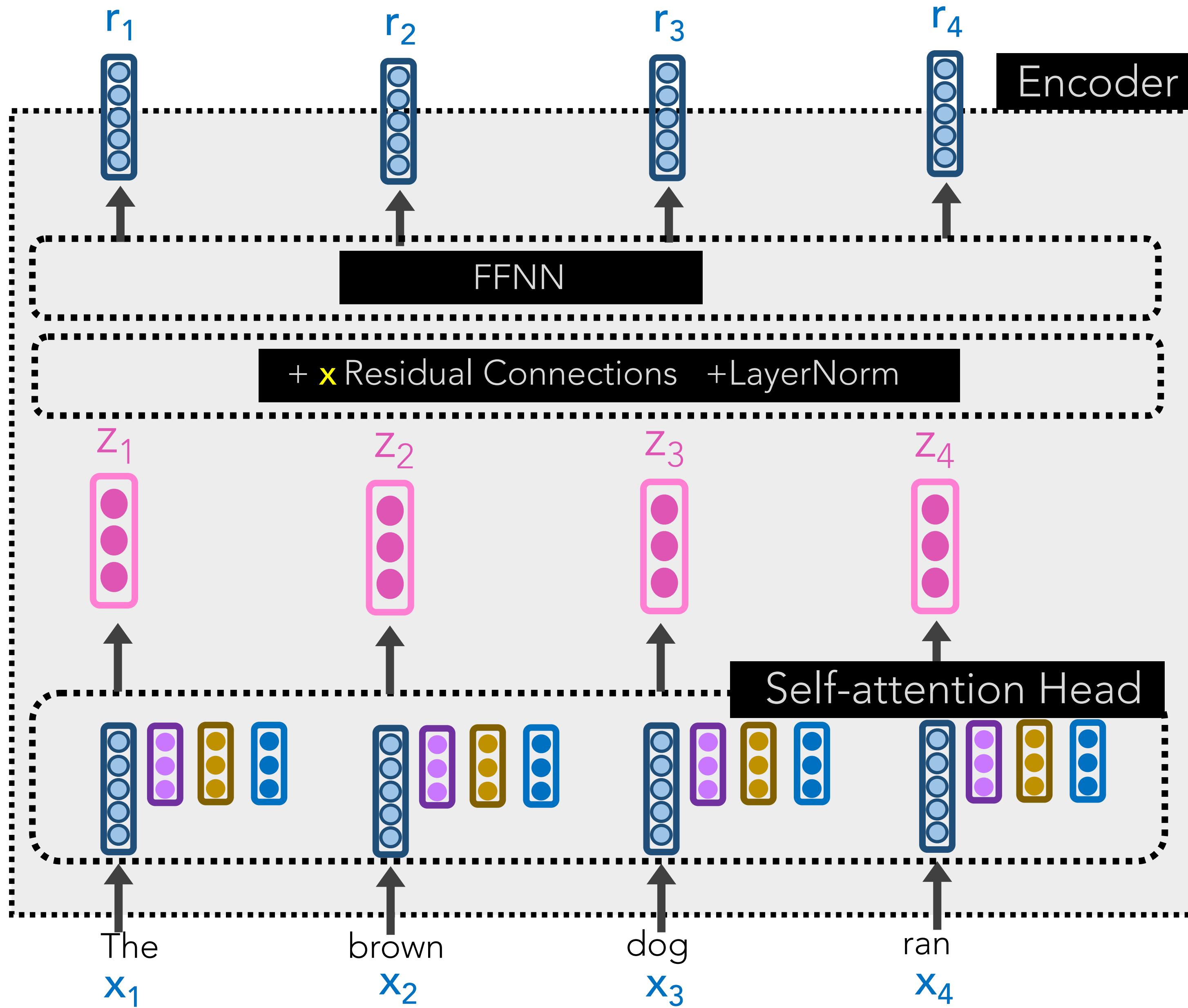
Let's further pass each  $z_i$  through a FFNN

We concat w/ a residual connection to help ensure relevant info is getting forward passed.

We perform LayerNorm to stabilize the network and allow for proper gradient flow. You should do this after the FFNN, too.

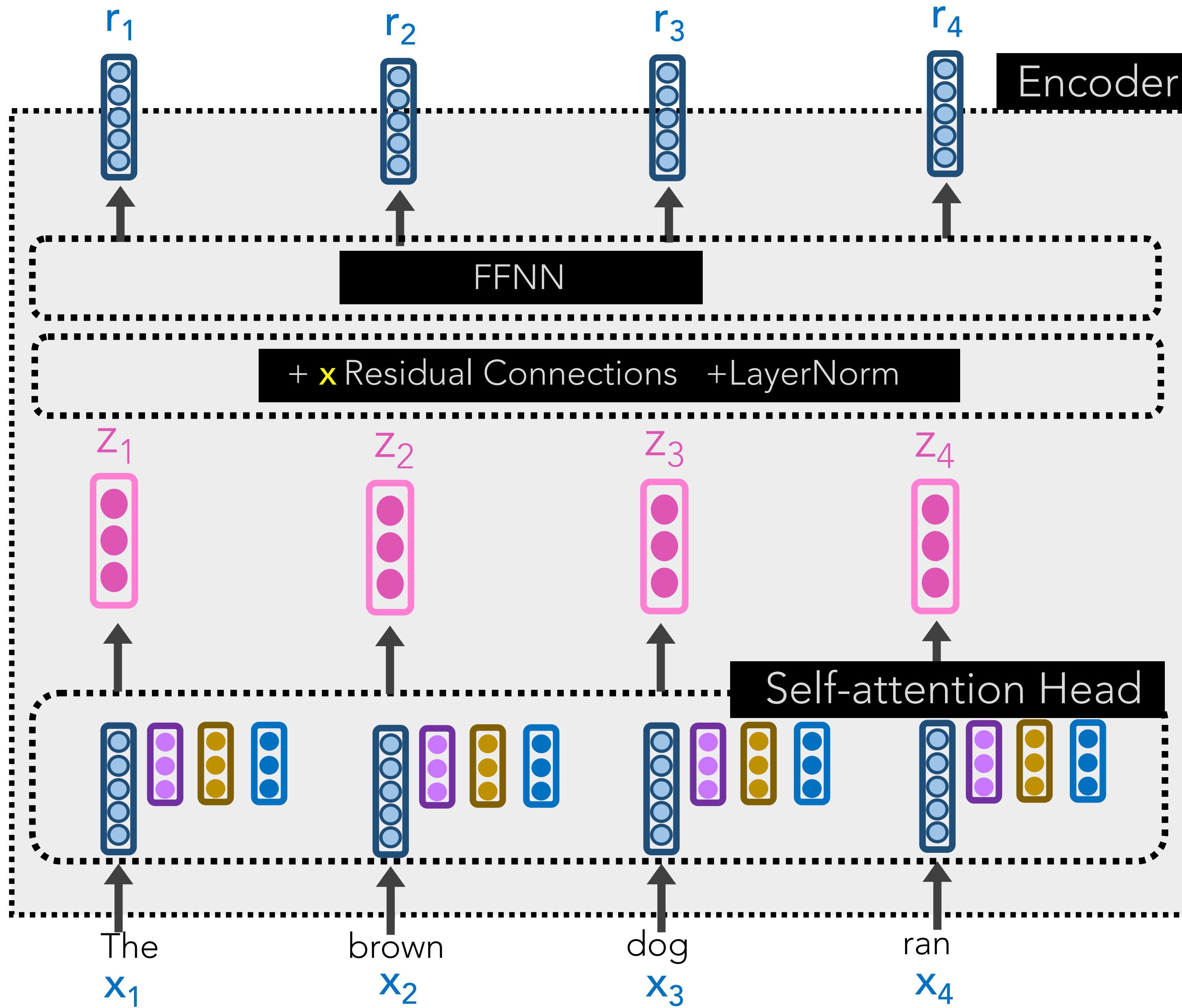
Each  $z_i$  can be computed in parallel, unlike LSTMs!

# Transformer Architecture



Yay! Our  $r_i$  vectors are our new representations, and this entire process is called a **Transformer Encoder**

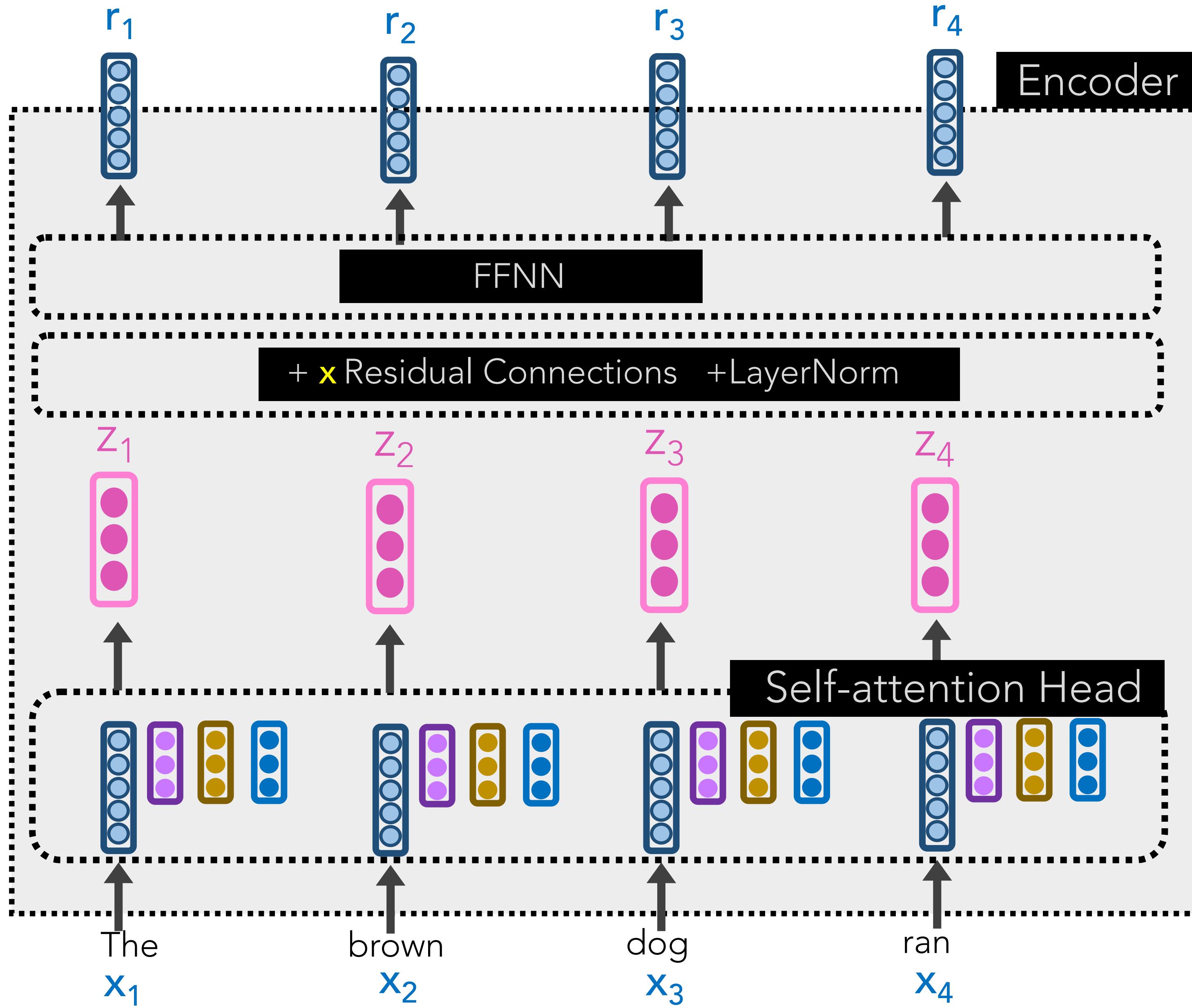
# Transformer Architecture



Yay! Our  $r_i$  vectors are our new representations, and this entire process is called a **Transformer Encoder**

Problem: there is no concept of positionality. Words are weighted as if a “bag of words”

# Transformer Architecture

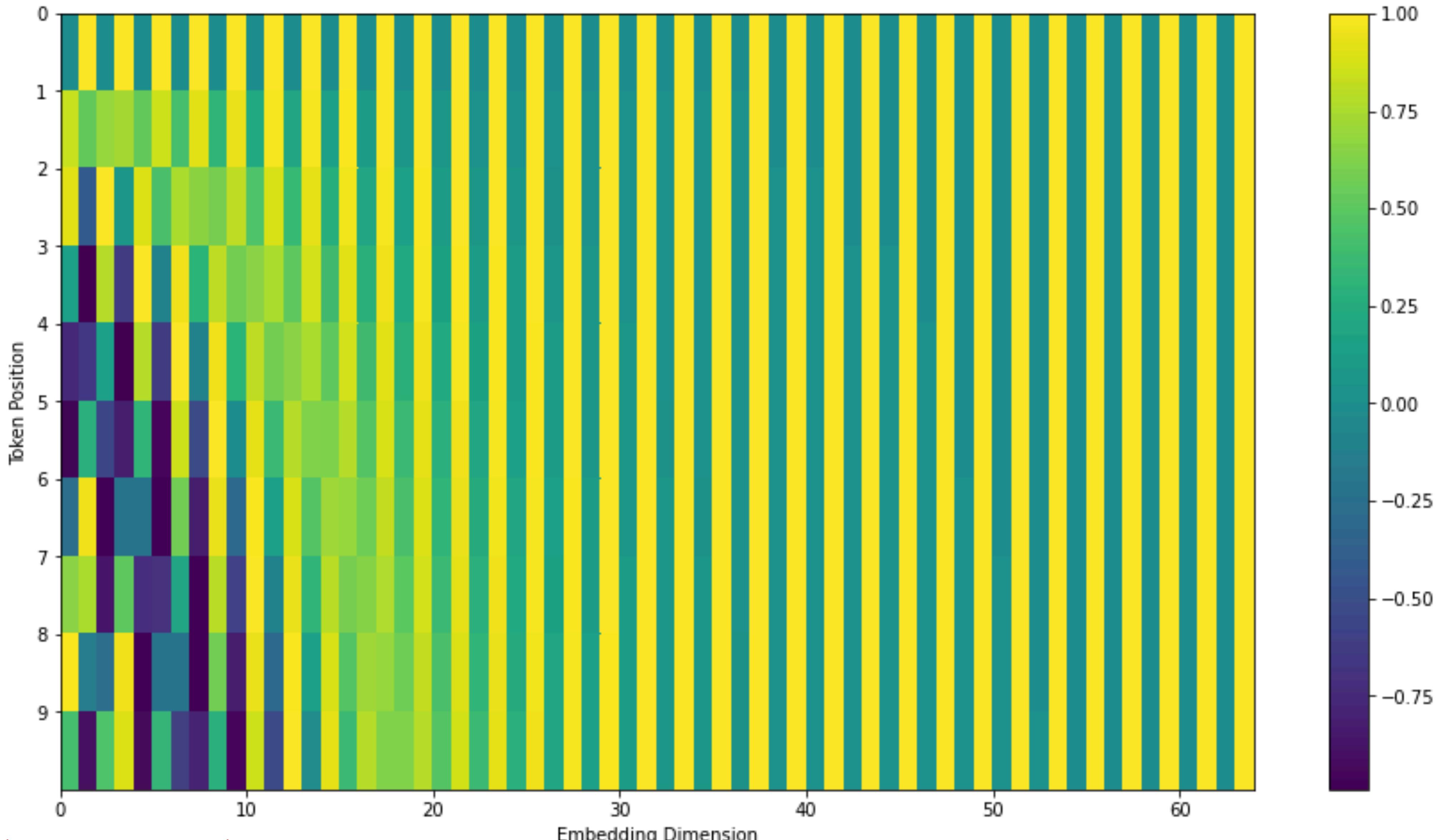


Yay! Our  $r_i$  vectors are our new representations, and this entire process is called a **Transformer Encoder**

**Problem:** there is no concept of positionality. Words are weighted as if a “bag of words”

**Solution:** add to each input word  $x_i$  a positional encoding  $\sim \sin(i)\cos(i)$

# Positional Encodings



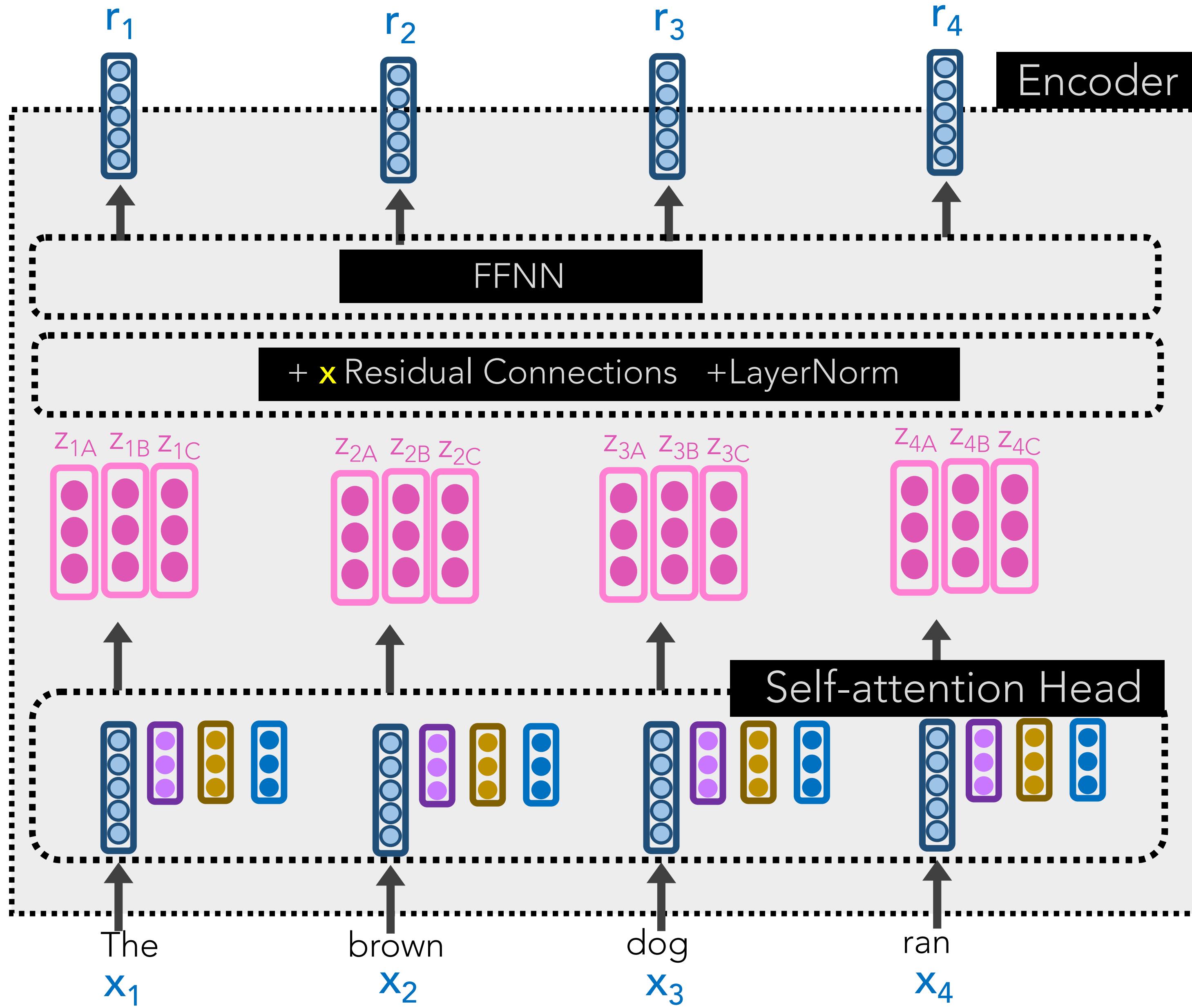
# Transformer Architecture

A Self-Attention Head has just one set of query/key/value weight matrices  $w_q, w_k, w_v$

Words can relate in many ways, so it's restrictive to rely on just one Self-Attention Head in the system.

Let's create Multi-headed Self-Attention

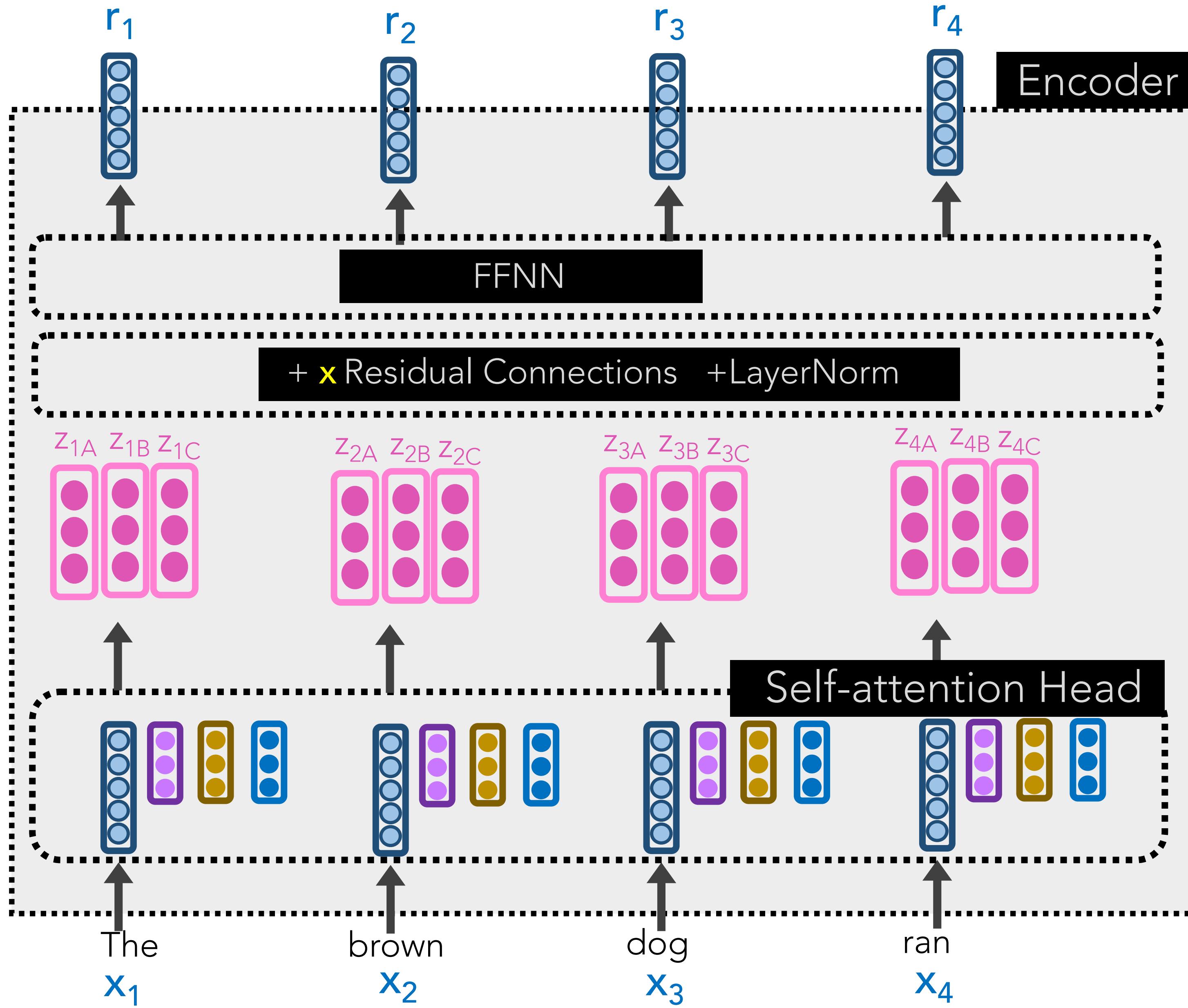
# Transformer Architecture



Each Self-Attention Head produces a  $z_i$  vector.

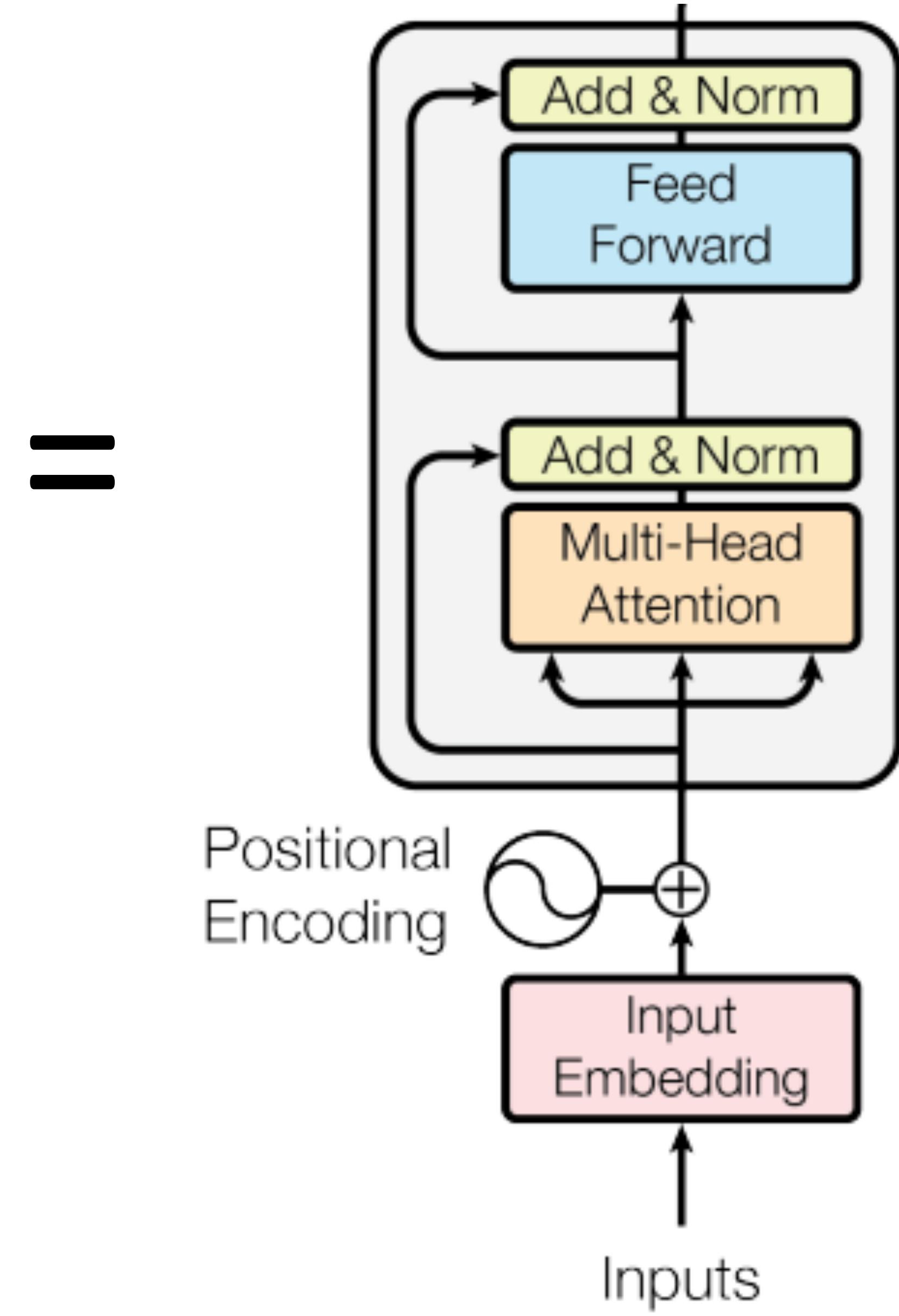
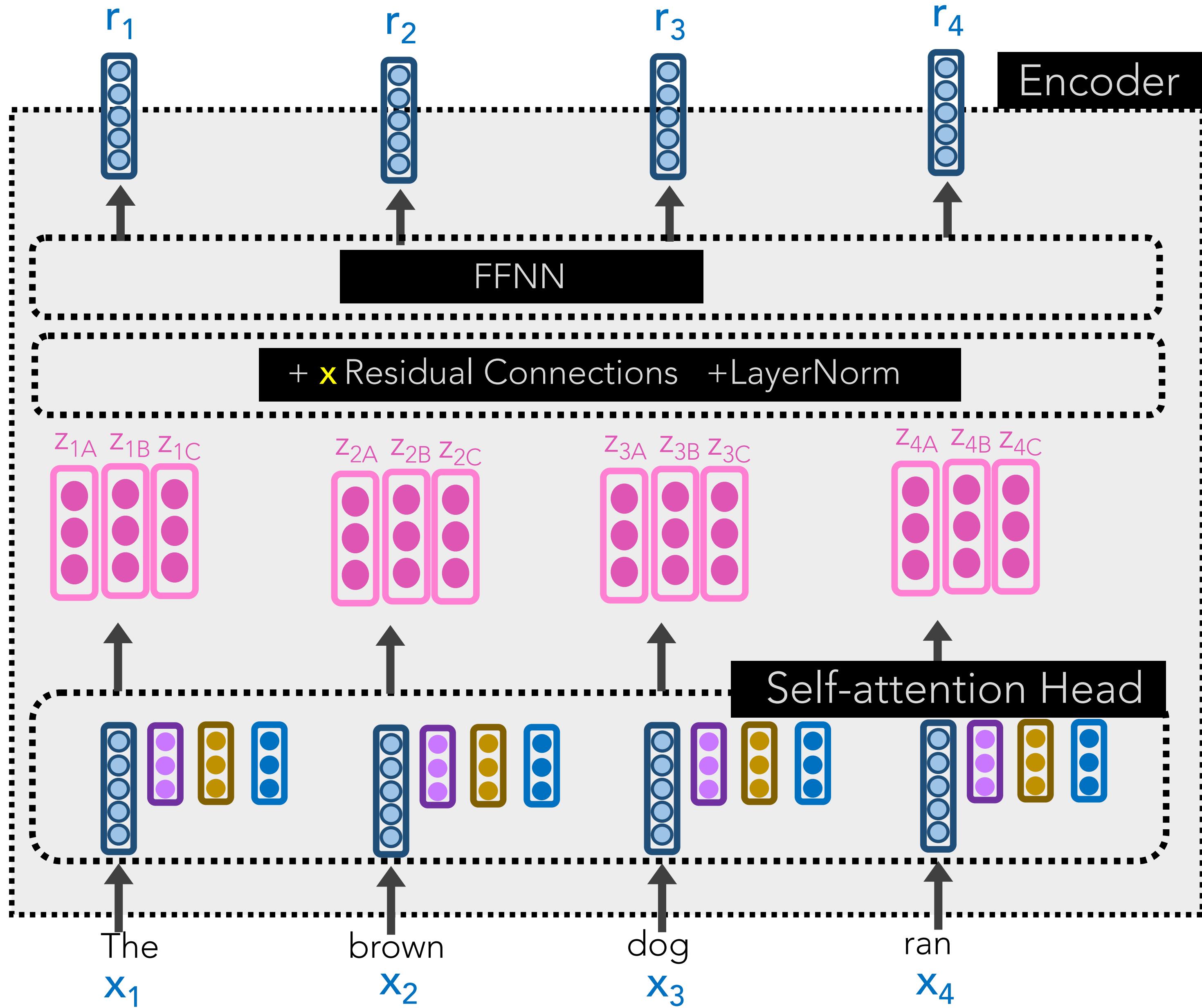
We can, in parallel, use multiple heads and concat the  $z_i$ 's.

# Transformer Architecture



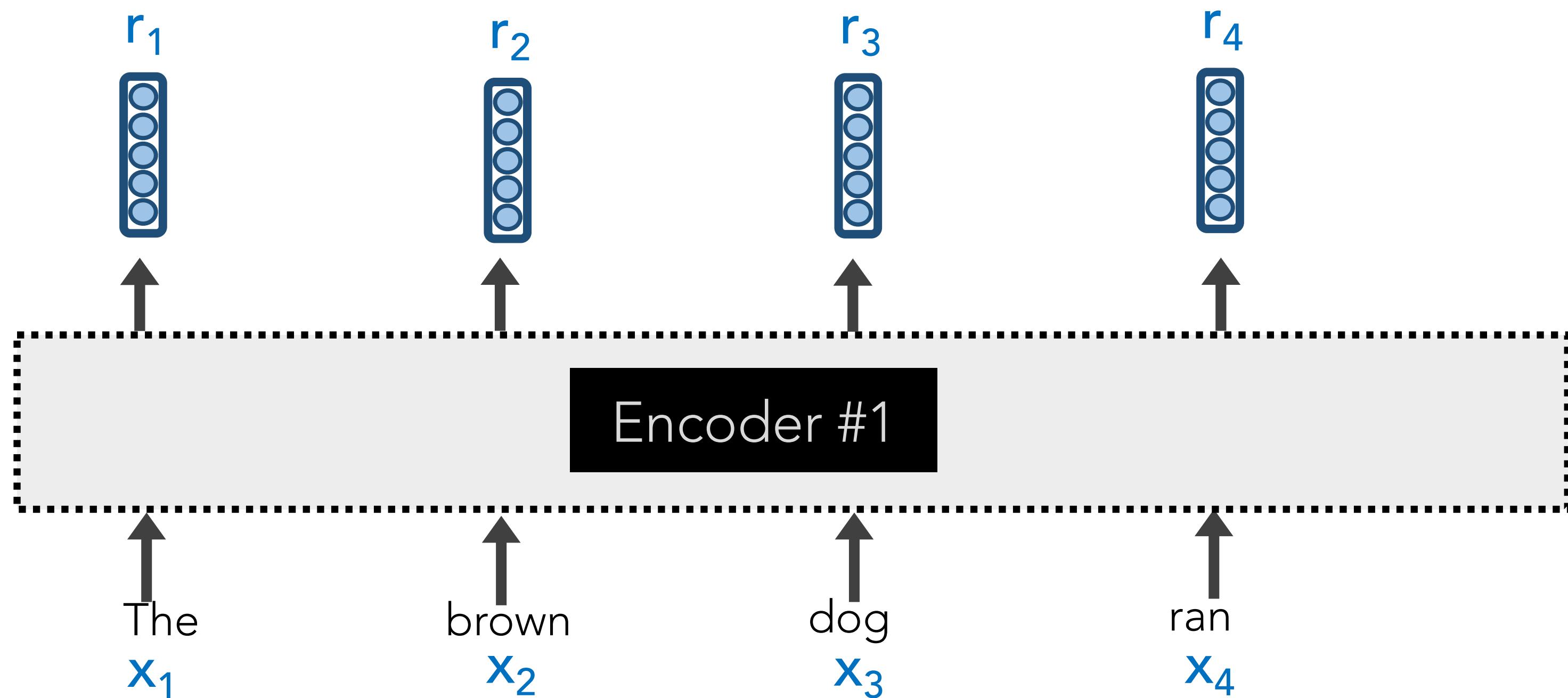
To recap: all of this looks fancy, but ultimately it's just producing a very good **contextualized embedding**  $r_i$  of each word  $x_i$

# Transformer Architecture



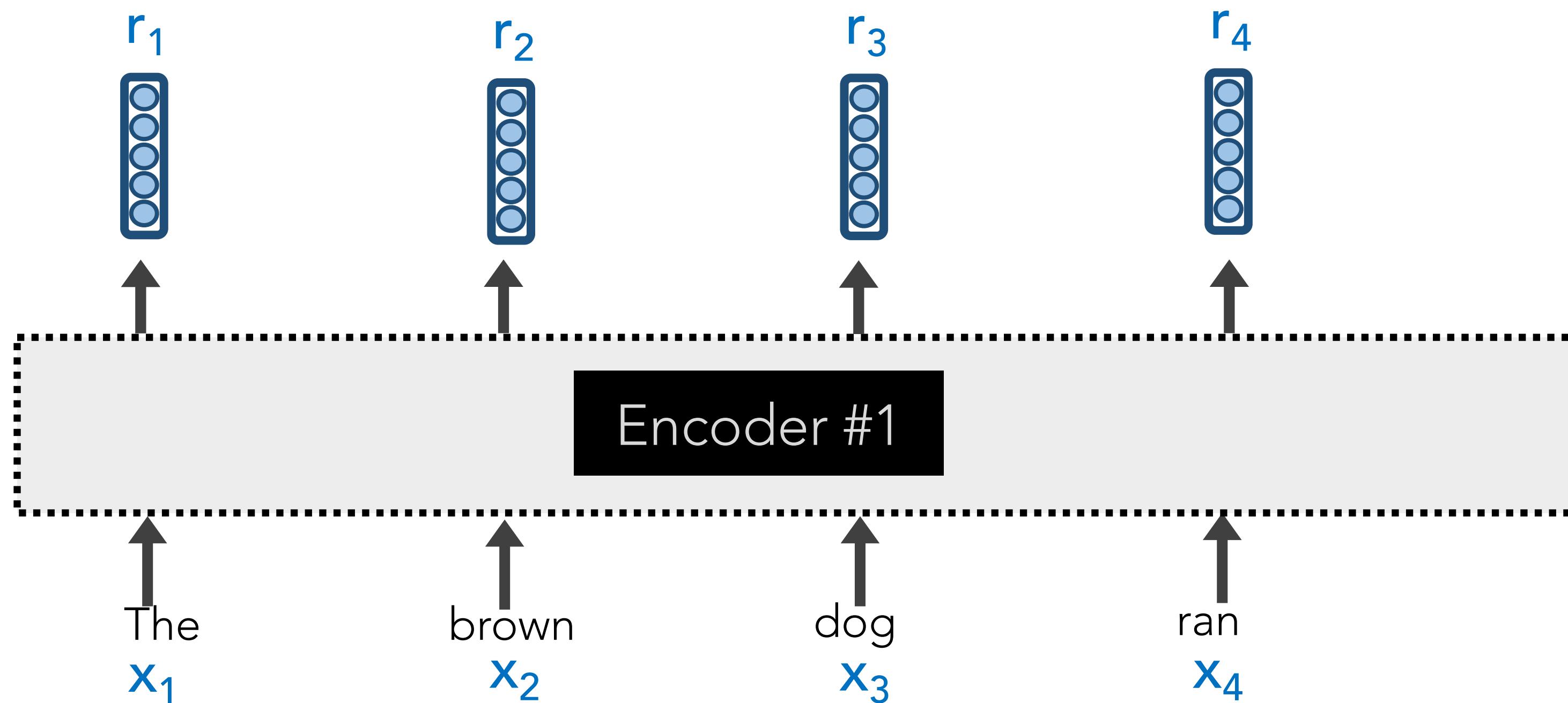
# Transformer Architecture

To recap: all of this looks fancy, but ultimately it's just producing a very good **contextualized embedding**  $r_i$  of each word  $x_i$



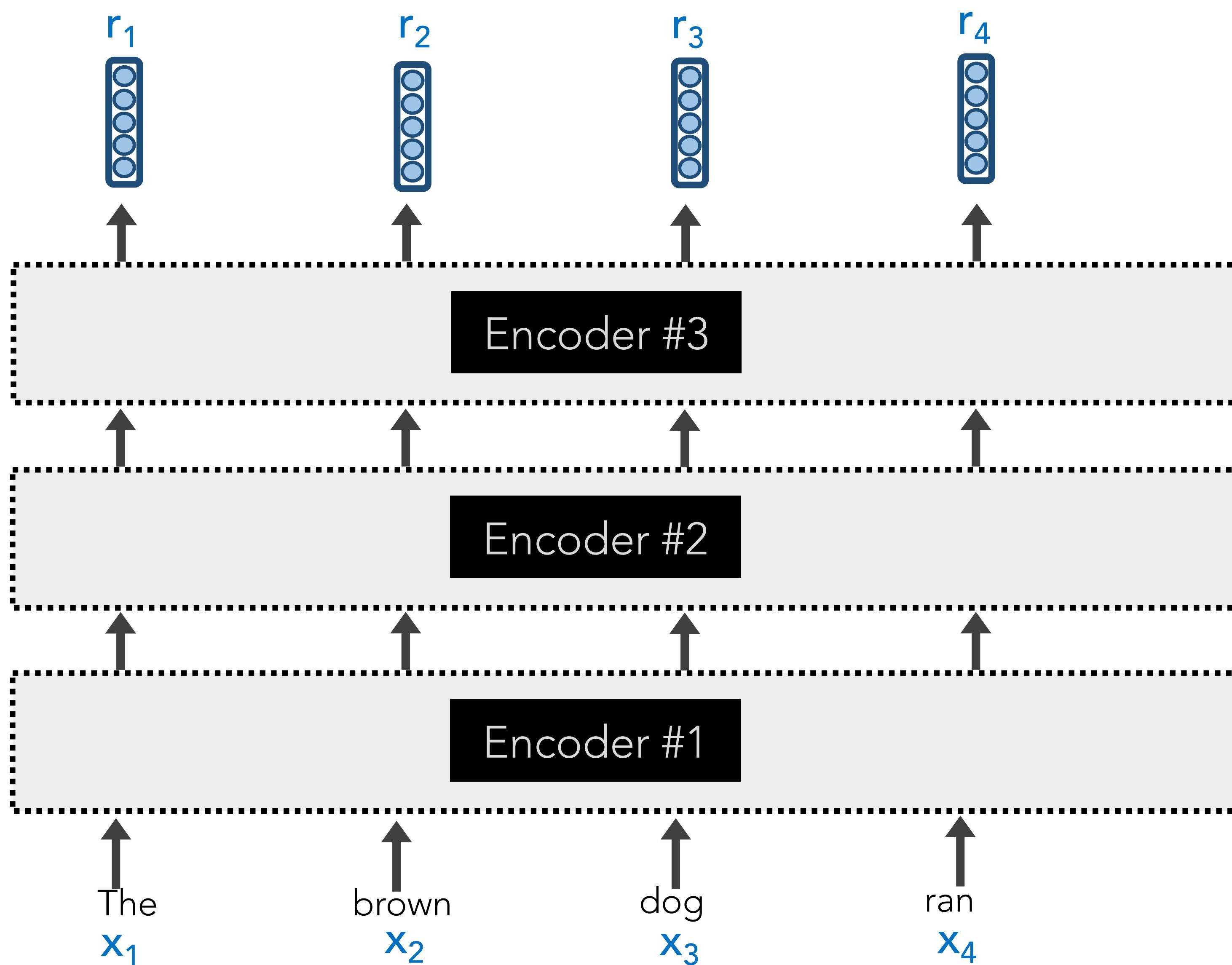
# Transformer Architecture

To recap: all of this looks fancy, but ultimately it's just producing a very good **contextualized embedding**  $r_i$  of each word  $x_i$



Why stop with just 1  
Transformer Encoder?  
We could stack several!

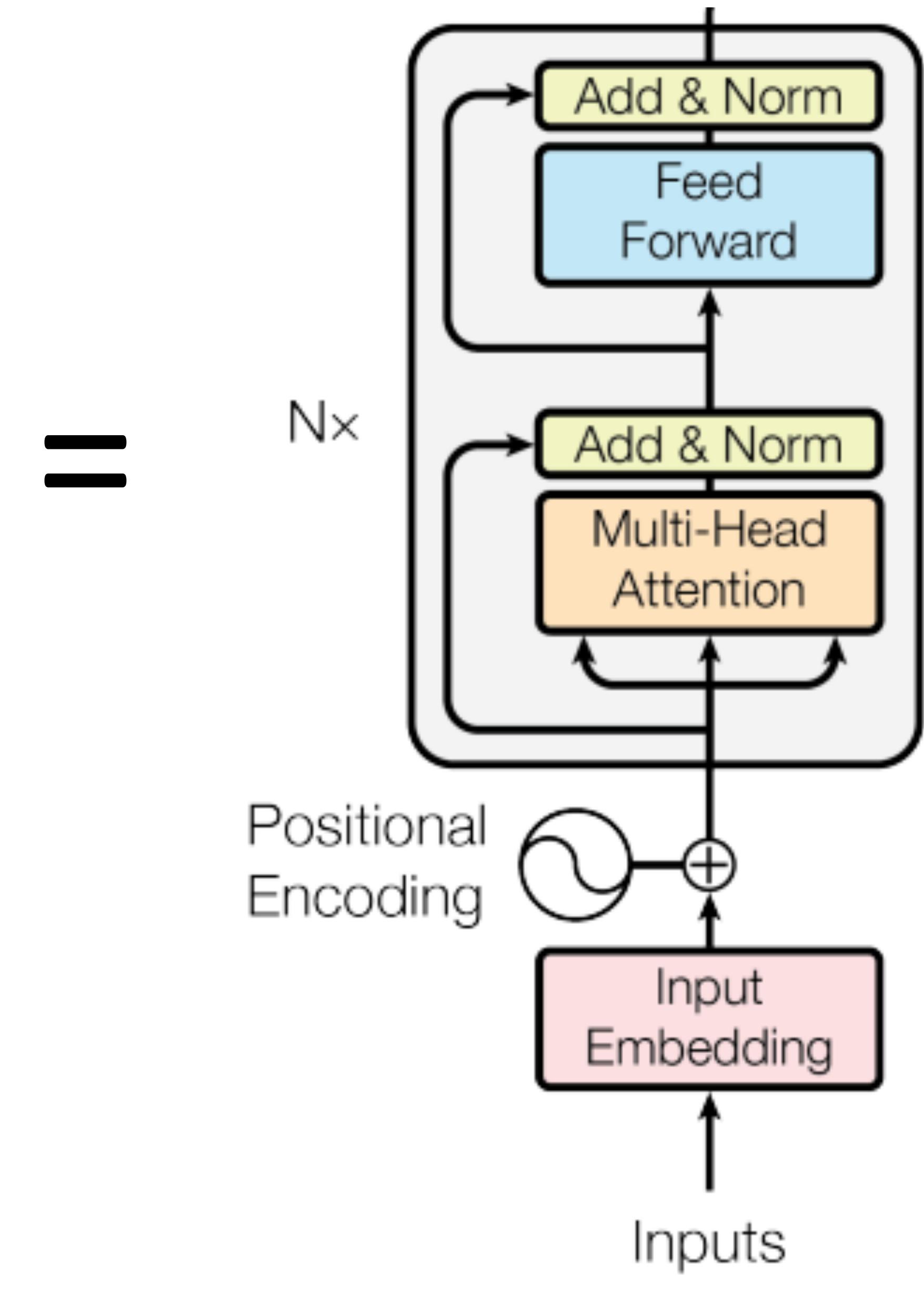
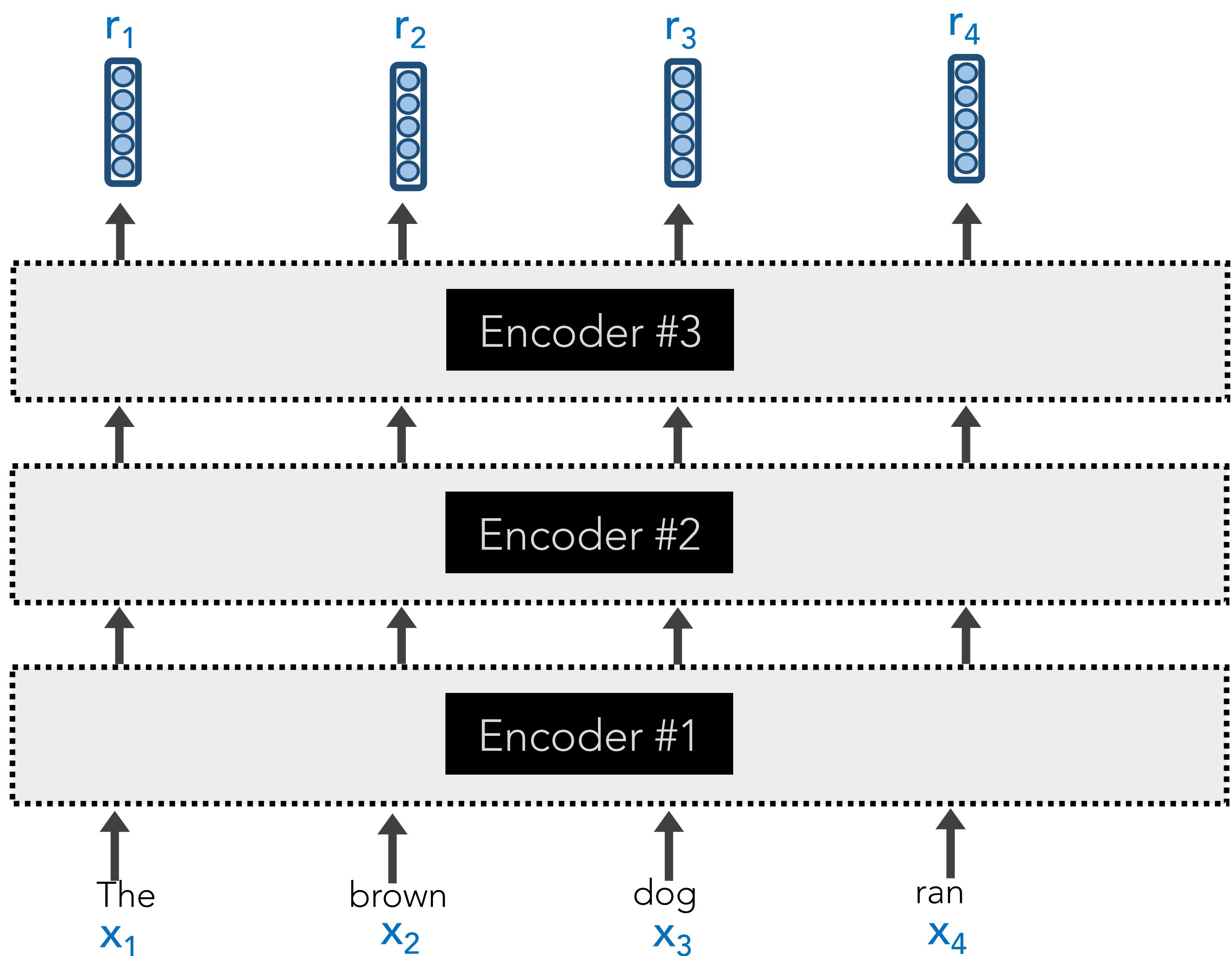
# Transformer Architecture



To recap: all of this looks fancy, but ultimately it's just producing a very good **contextualized embedding**  $r_i$  of each word  $x_i$

Why stop with just 1 Transformer Encoder?  
We could stack several!

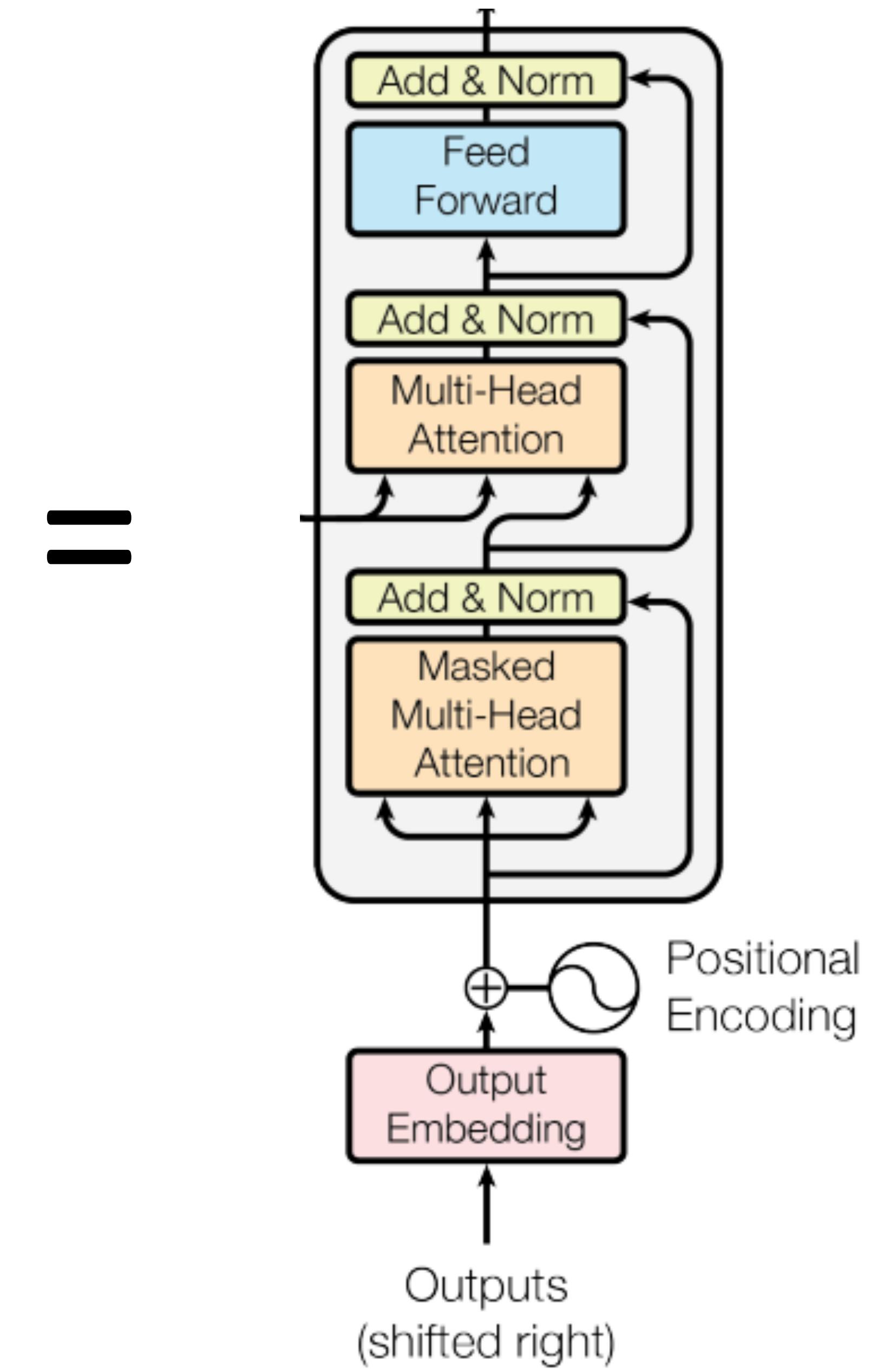
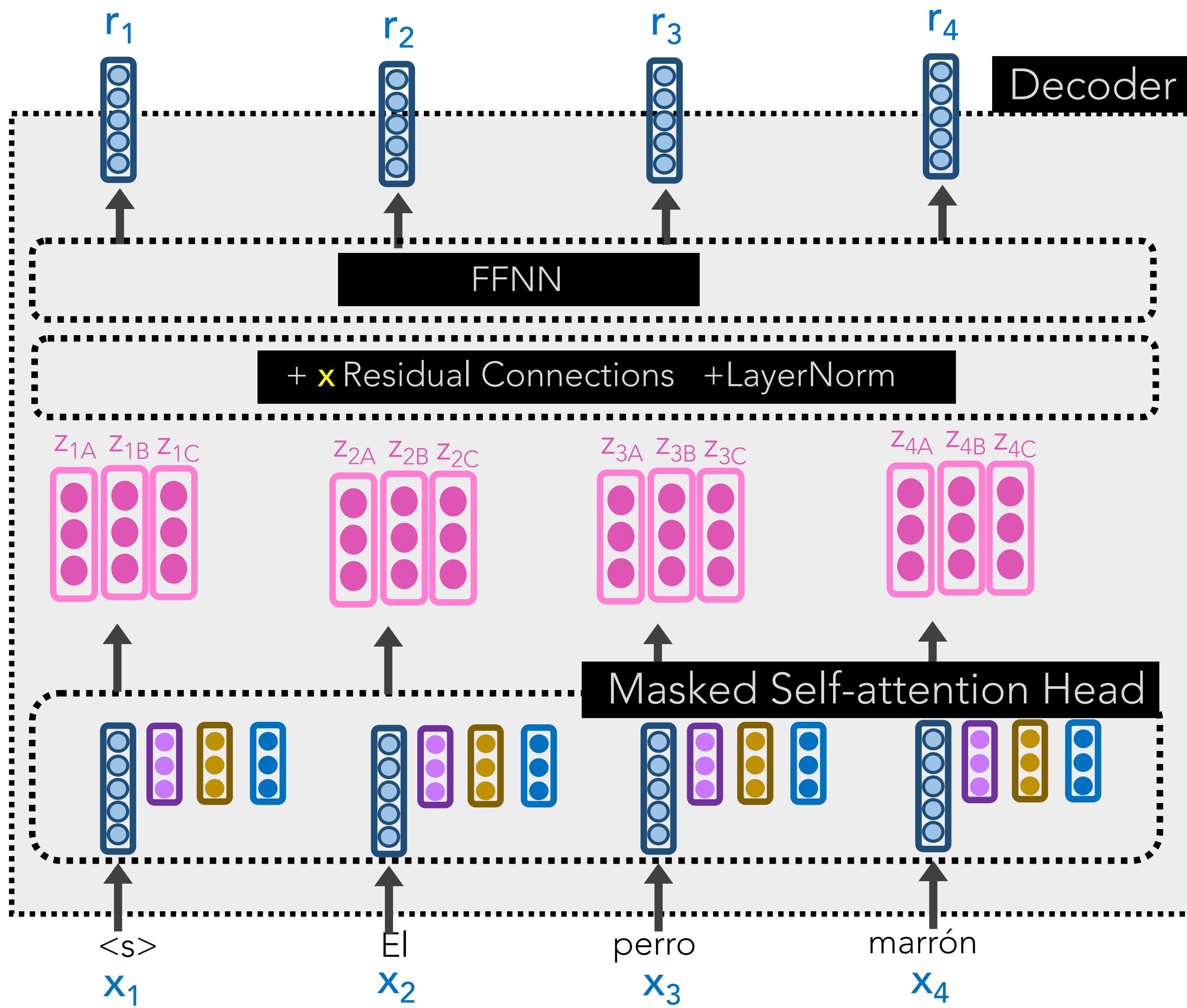
# Transformer Architecture



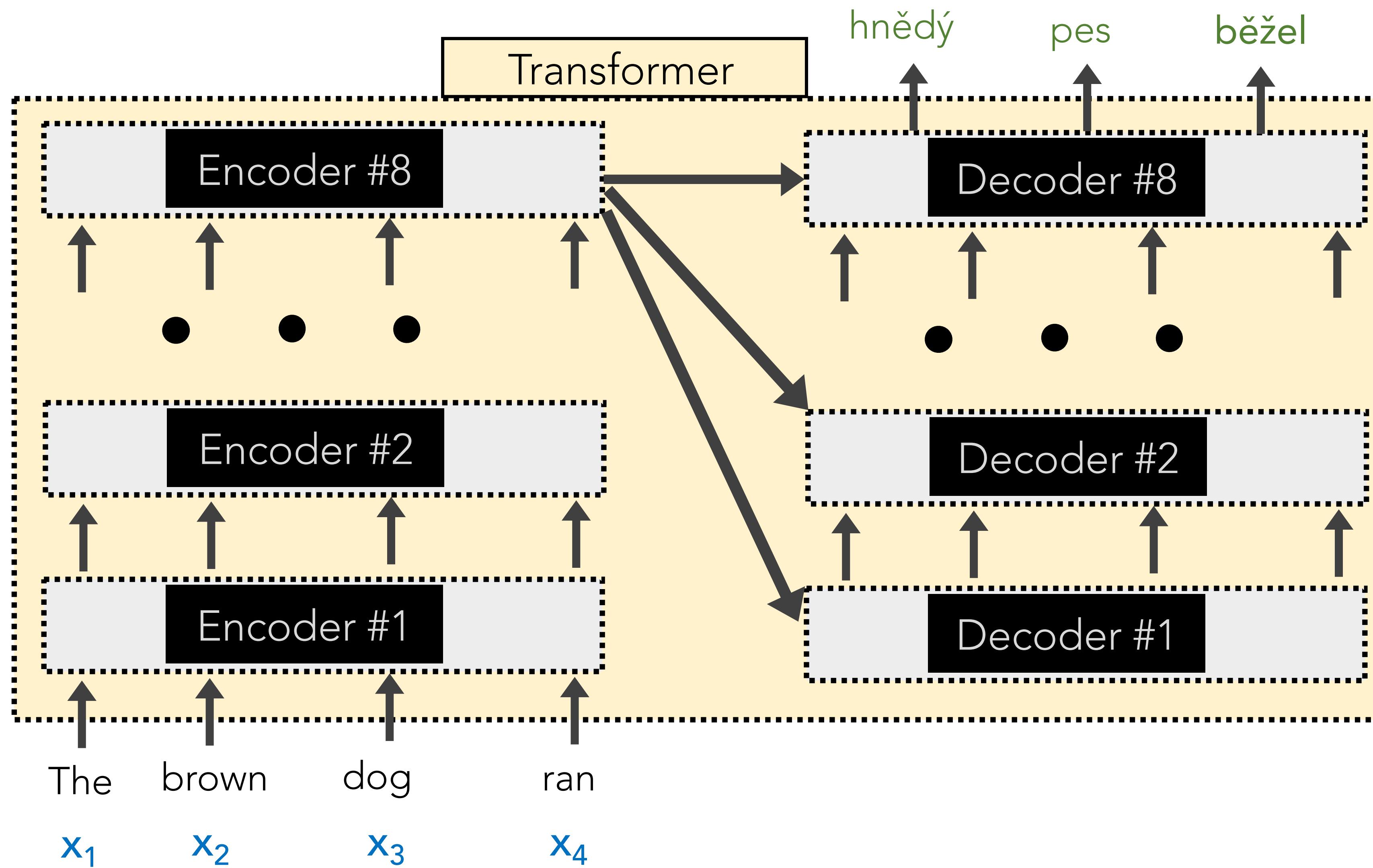
# Transformer Decoders

The original Transformer model was intended for Machine Translation, so it had Decoders, too

# Transformer Decoders



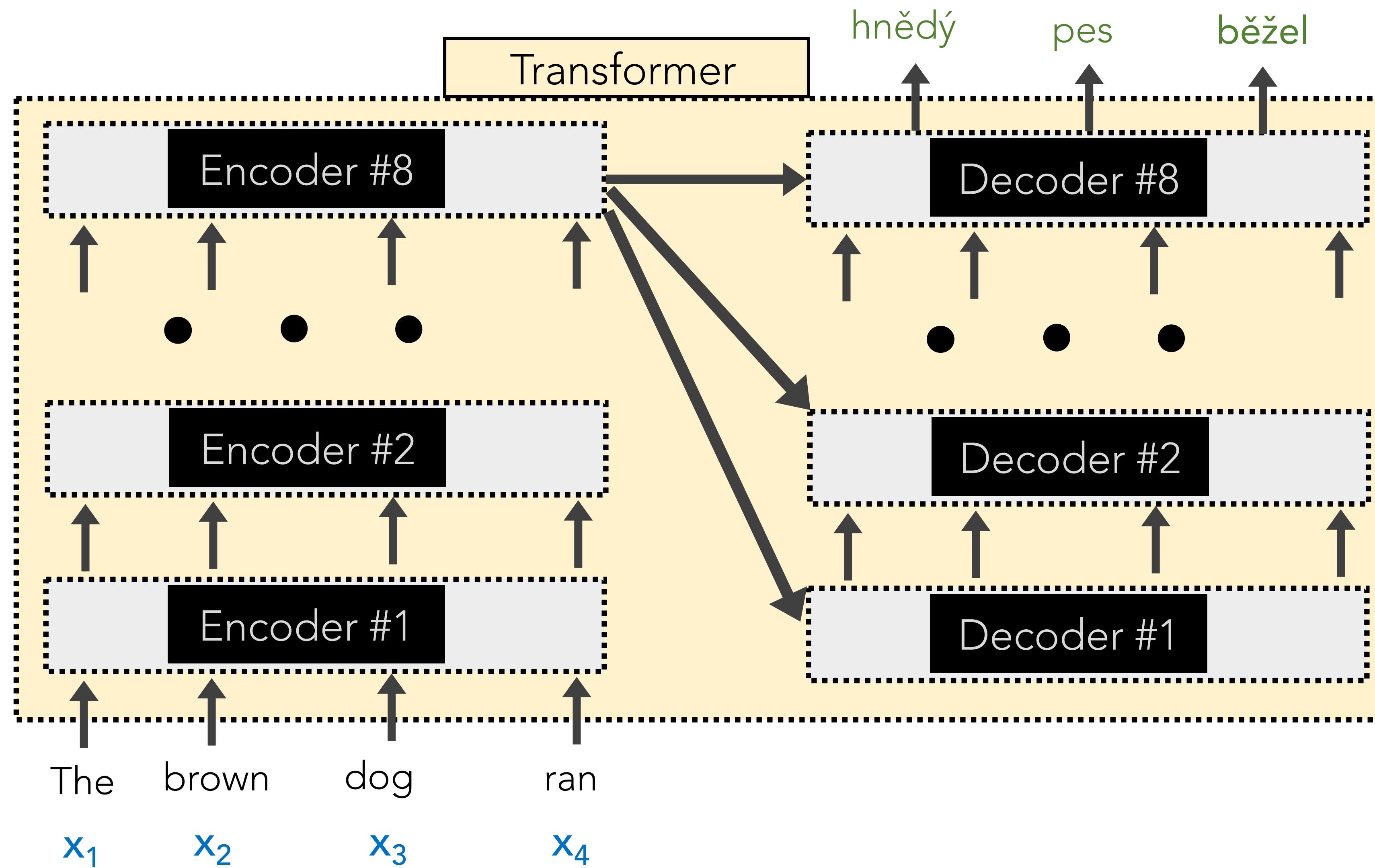
# Transformer Decoders



Transformer Encoders produce **contextualized embeddings** of each word

Transformer Decoders generate new sequences of text

# Transformer Decoders

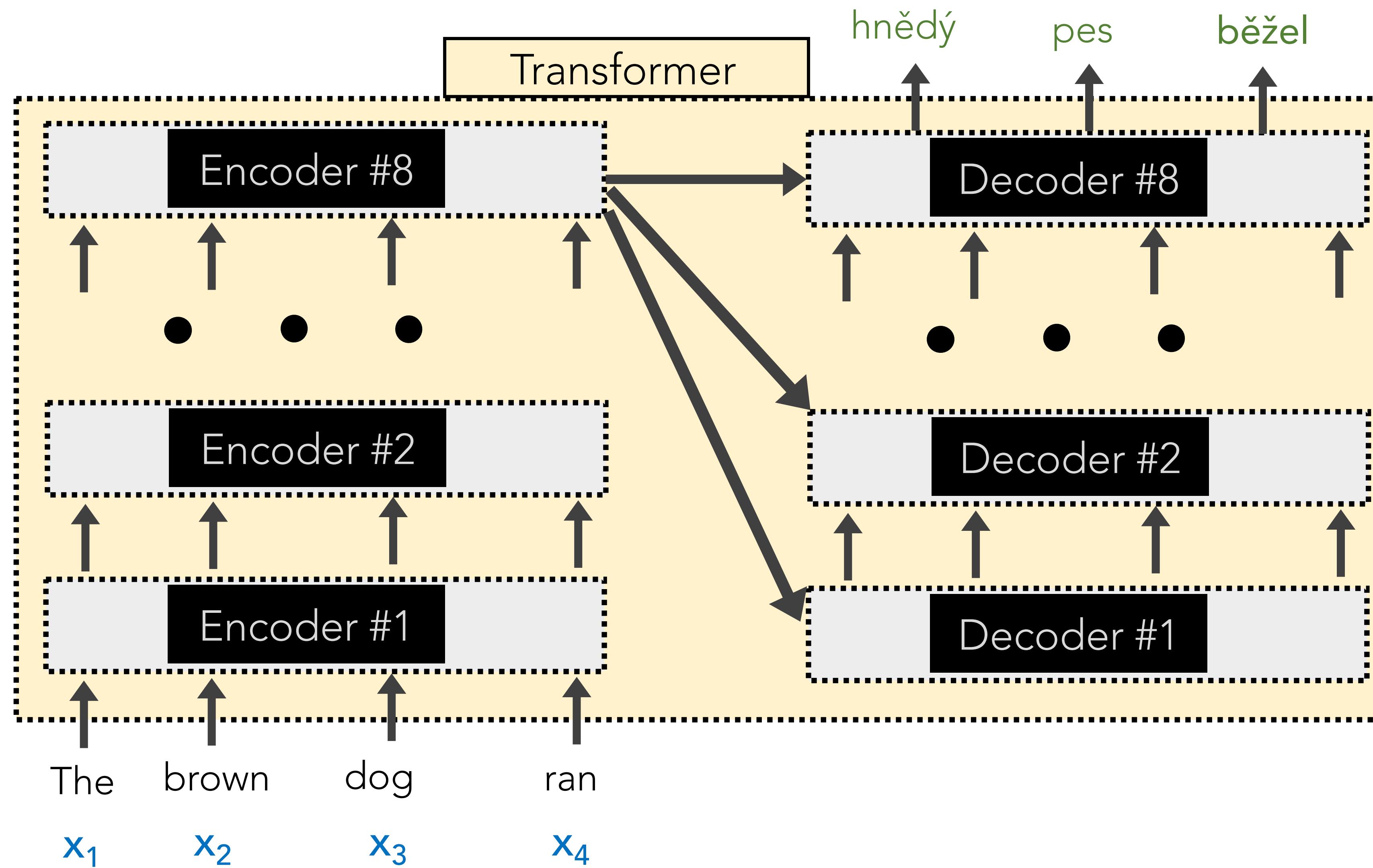


## NOTE

Transformer Decoders are identical to the Encoders, except they have an additional Attention Head in between the Self-Attention and FFNN layers.

This additional Attention Head **focuses on parts of the encoder's representations**.

# Transformer Decoders

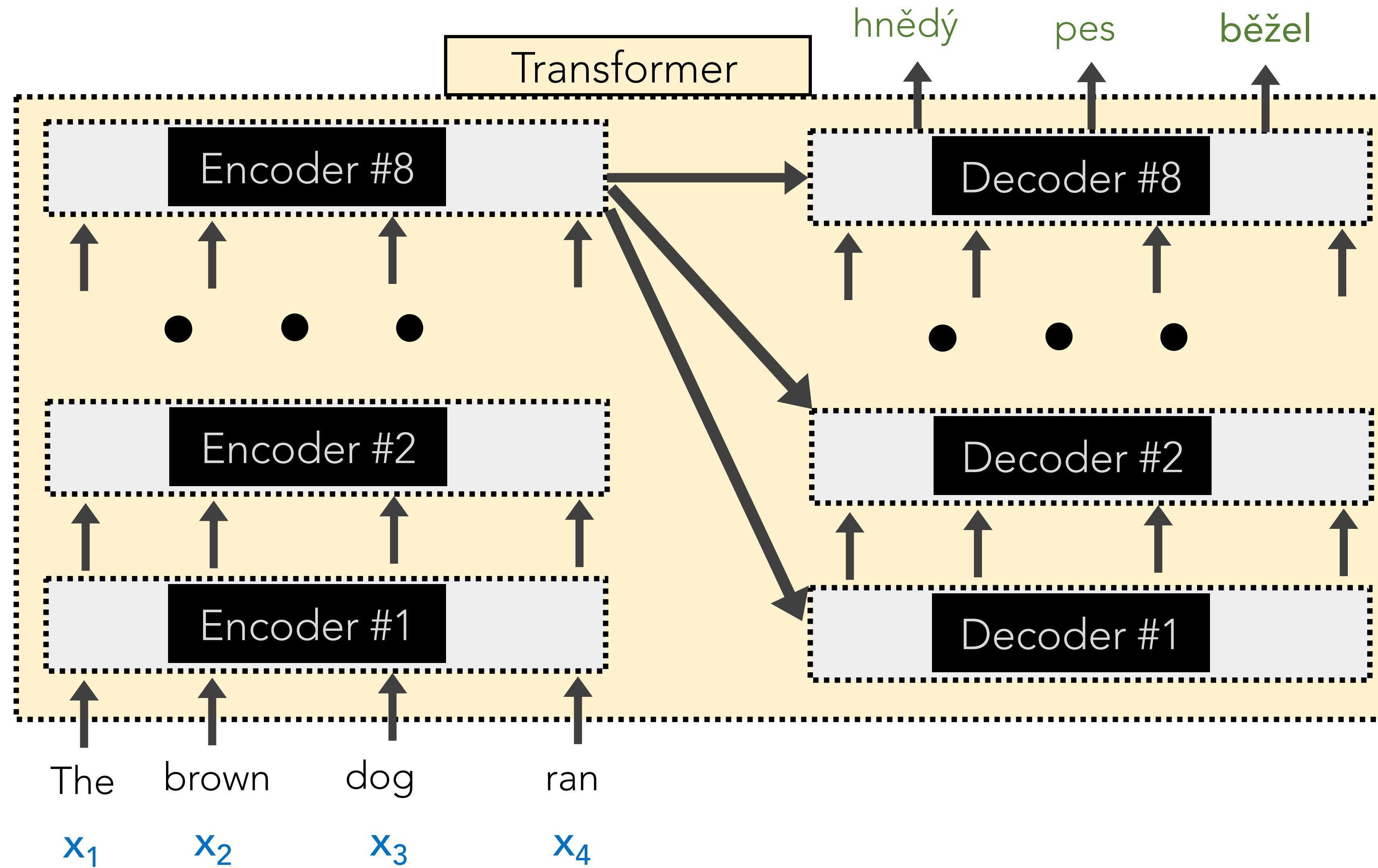


## NOTE

The **query** vector for a Transformer Decoder's **Attention Head** (not Self-Attention Head) is from the output of the previous decoder layer.

However, the **key** and **value** vectors are from the Transformer Encoders' outputs.

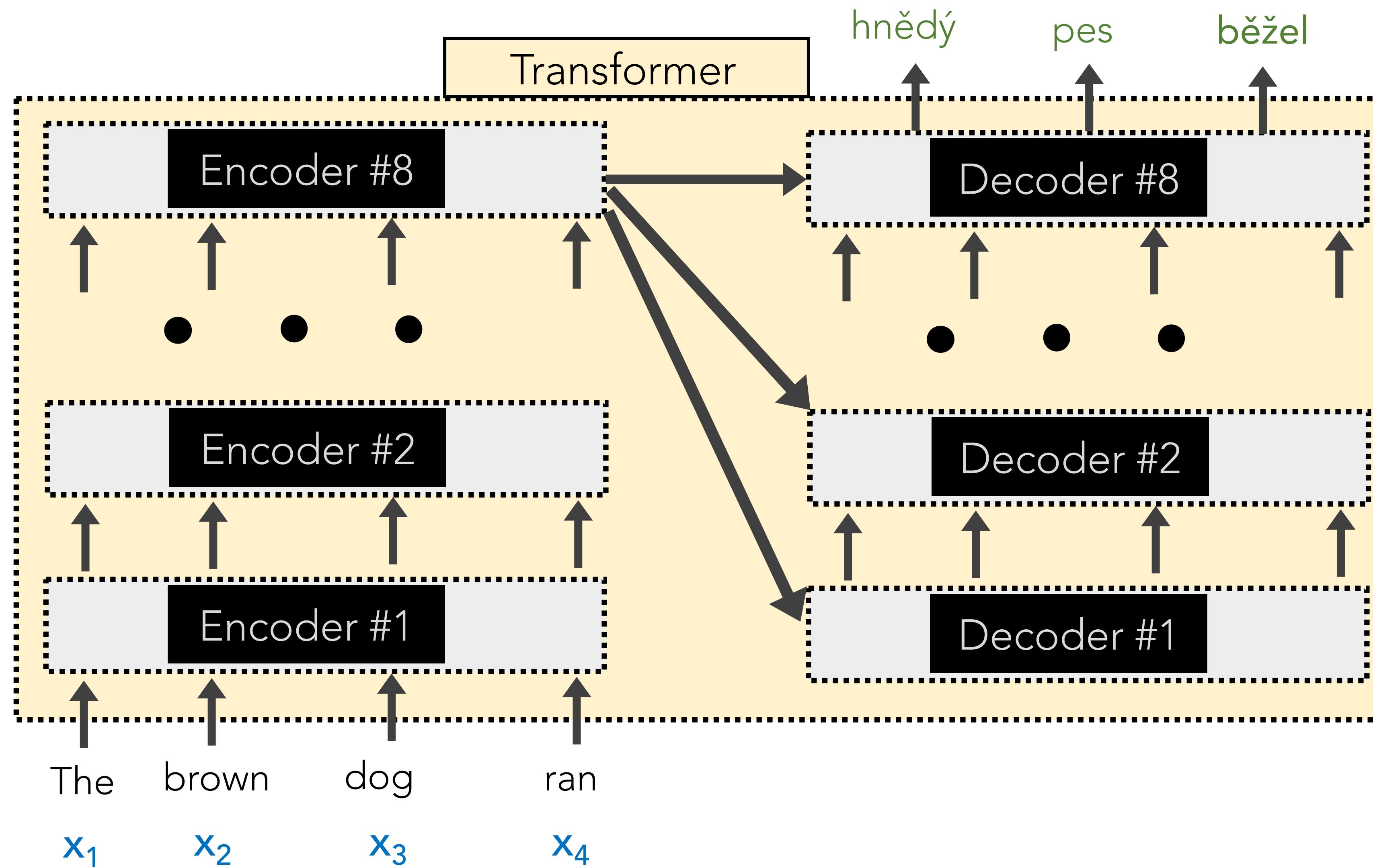
# Transformer Decoders



## NOTE

The **query**, **key**, and **value** vectors for a Transformer Decoder's **Self-Attention Head** (not Attention Head) are all from the output of the previous decoder layer.

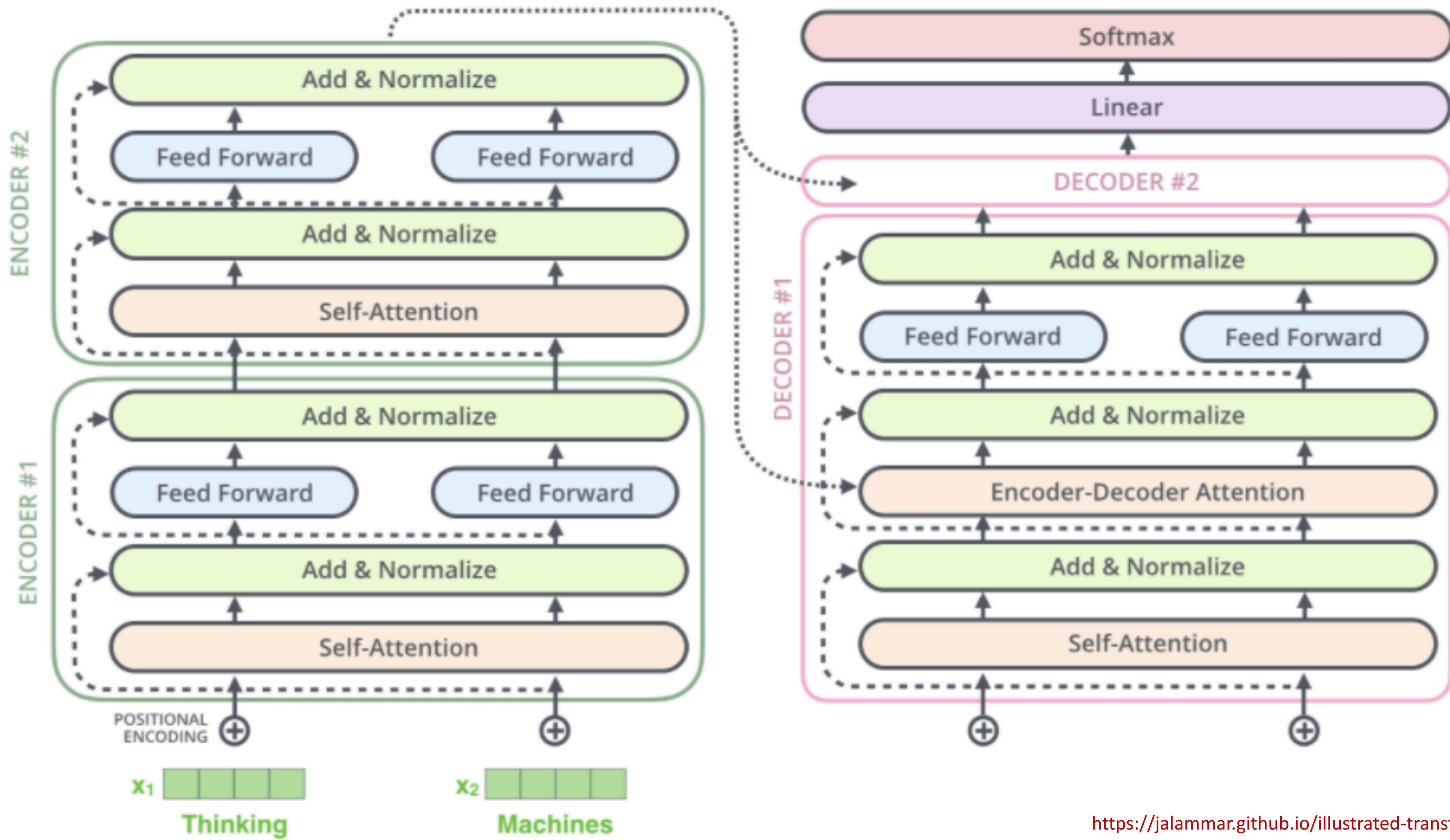
# Transformer Decoders



## IMPORTANT

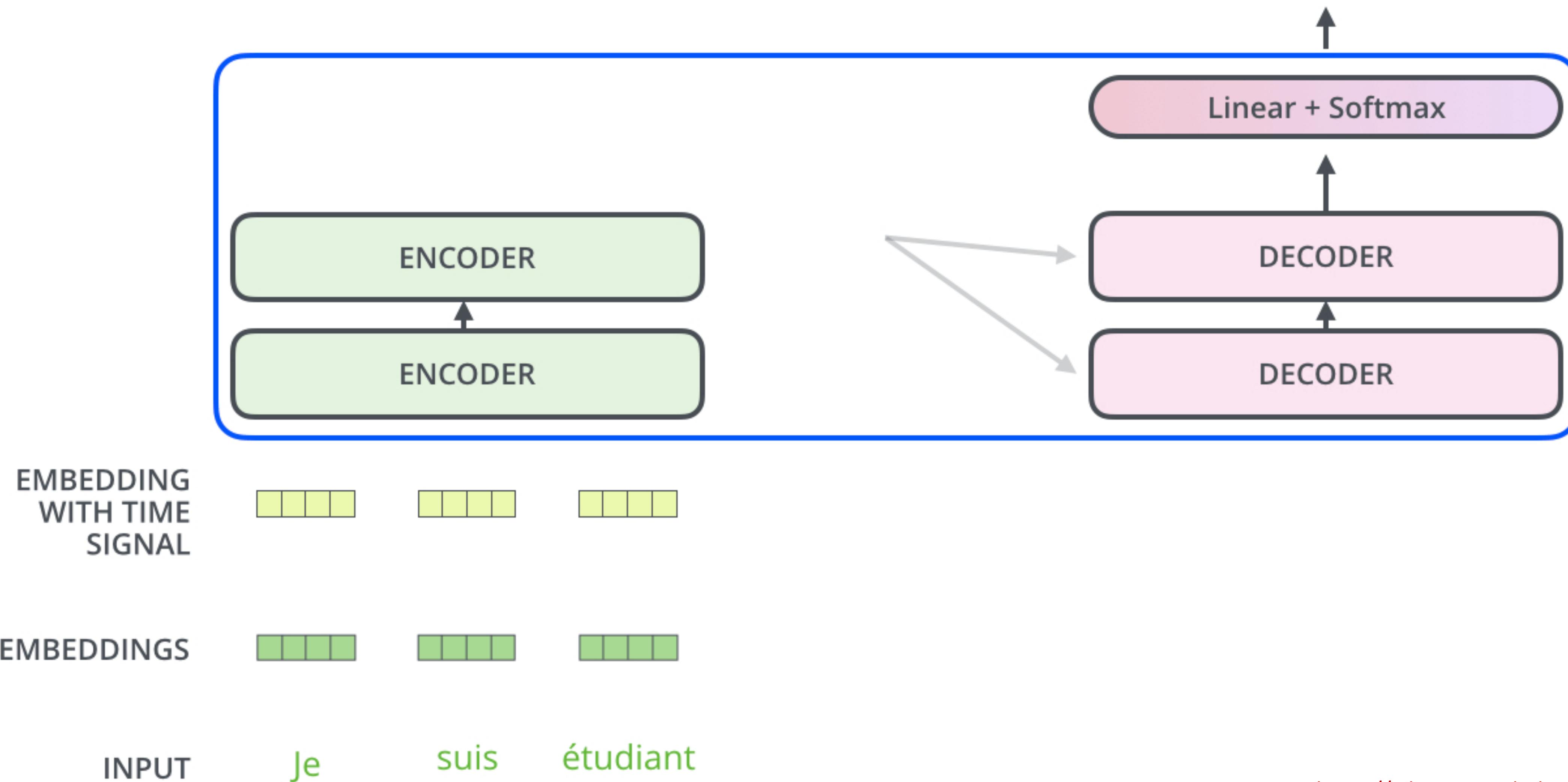
The Transformer Decoders have positional embeddings, too, just like the Encoders.

Critically, each position is only allowed to attend to the previous indices. This masked Attention preserves it as being an auto-regressive LM.



Decoding time step: 1 2 3 4 5 6

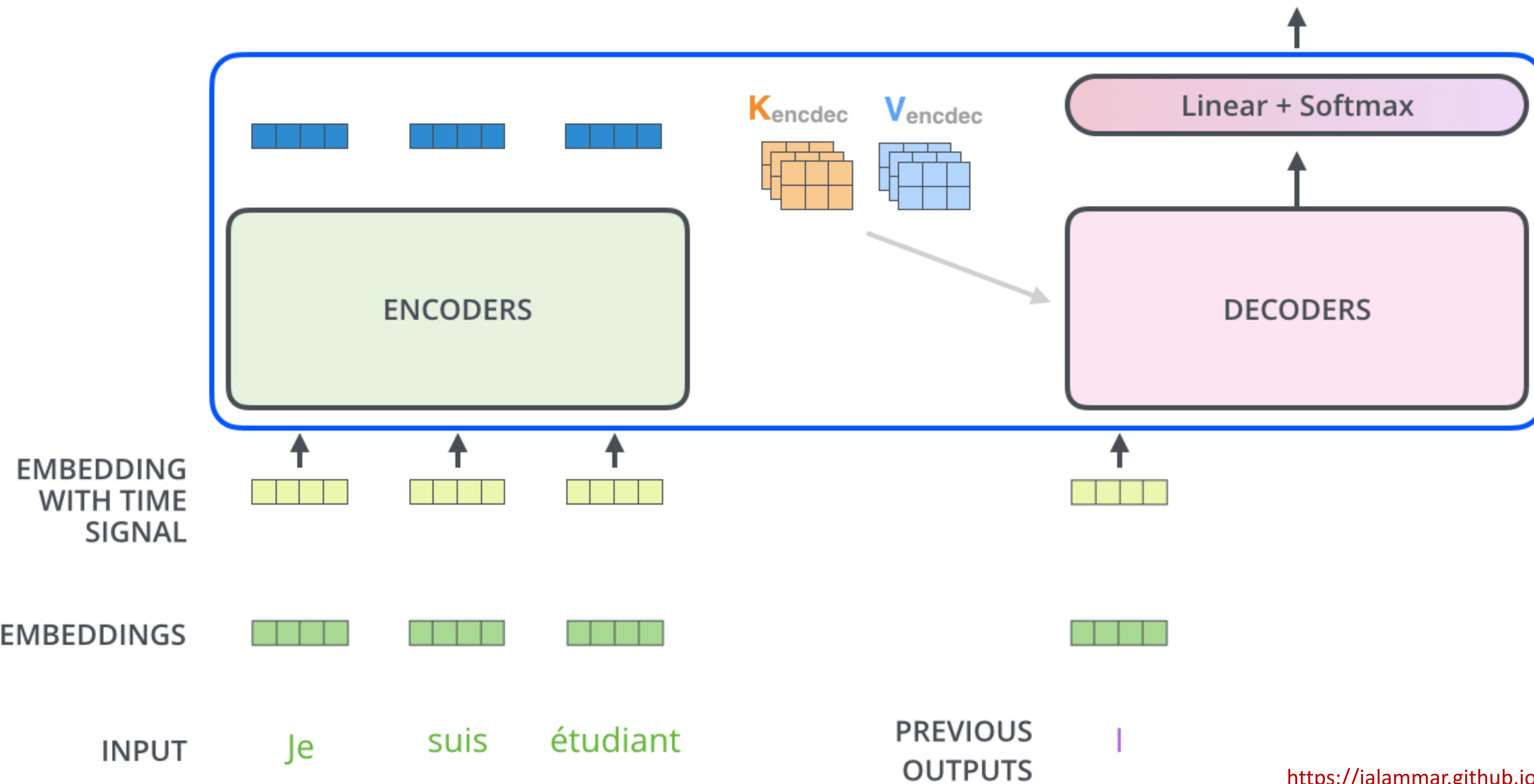
OUTPUT



Decoding time step: 1 2 3 4 5 6

OUTPUT

|



# Transformer Architecture

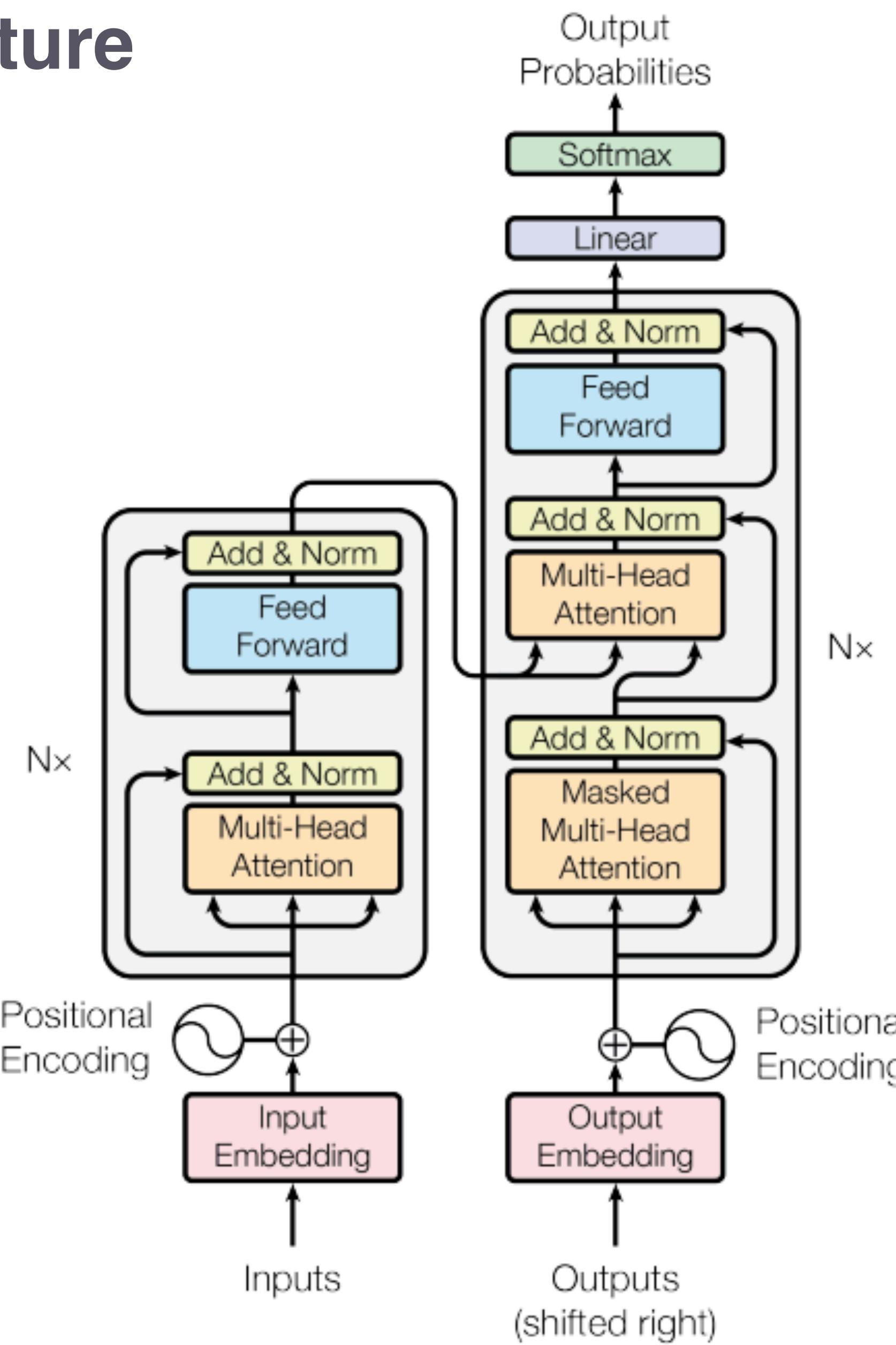


Figure 1: The Transformer - model architecture.

Attention is All you Need (2017) <https://arxiv.org/pdf/1706.03762.pdf>

# Complexity

Loss Function: cross-entropy (predicting translated word)

Training Time: ~4 days on (8) GPUs

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

# Complexity

$n$  = sequence length

$d$  = length of representation (vector)

Q: Is the complexity of self-attention good?

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

# Complexity

Important: when learning dependencies b/w words, you don't want long paths. Shorter is better.

Self-attention connects all positions with a constant # of sequentially executed operations, whereas RNNs require  $O(n)$ .

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

## Machine Translation results: state-of-the-art (at the time)

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	<b>41.29</b>	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1		<b><math>3.3 \cdot 10^{18}</math></b>
Transformer (big)	<b>28.4</b>	<b>41.8</b>		$2.3 \cdot 10^{19}$

# Performance

Machine Translation results: state-of-the-art (at the time)

You can train to translate from Language A to Language B.

Then train it to translate from Language B. to Language C.

Then, without training, it can translate from Language A to Language C

# BERT Motivation

- What if we don't want to decode/translate?
- Just want to perform a particular task (e.g., classification)
- Want even more robust, flexible, rich representation!
- Want positionality to play a more explicit role, while not being restricted to a particular form (e.g., CNNs)

# BERT Motivation

Bidirectional Encoder Representations from Transformers



# BERT Motivation

## Bidirectional Encoder Representations from Transformers

Like Bidirectional LSTMs, let's look in both directions



# BERT Motivation

## Bidirectional Encoder Representations from Transformers

Let's only use Transformer Encoders, no Decoders



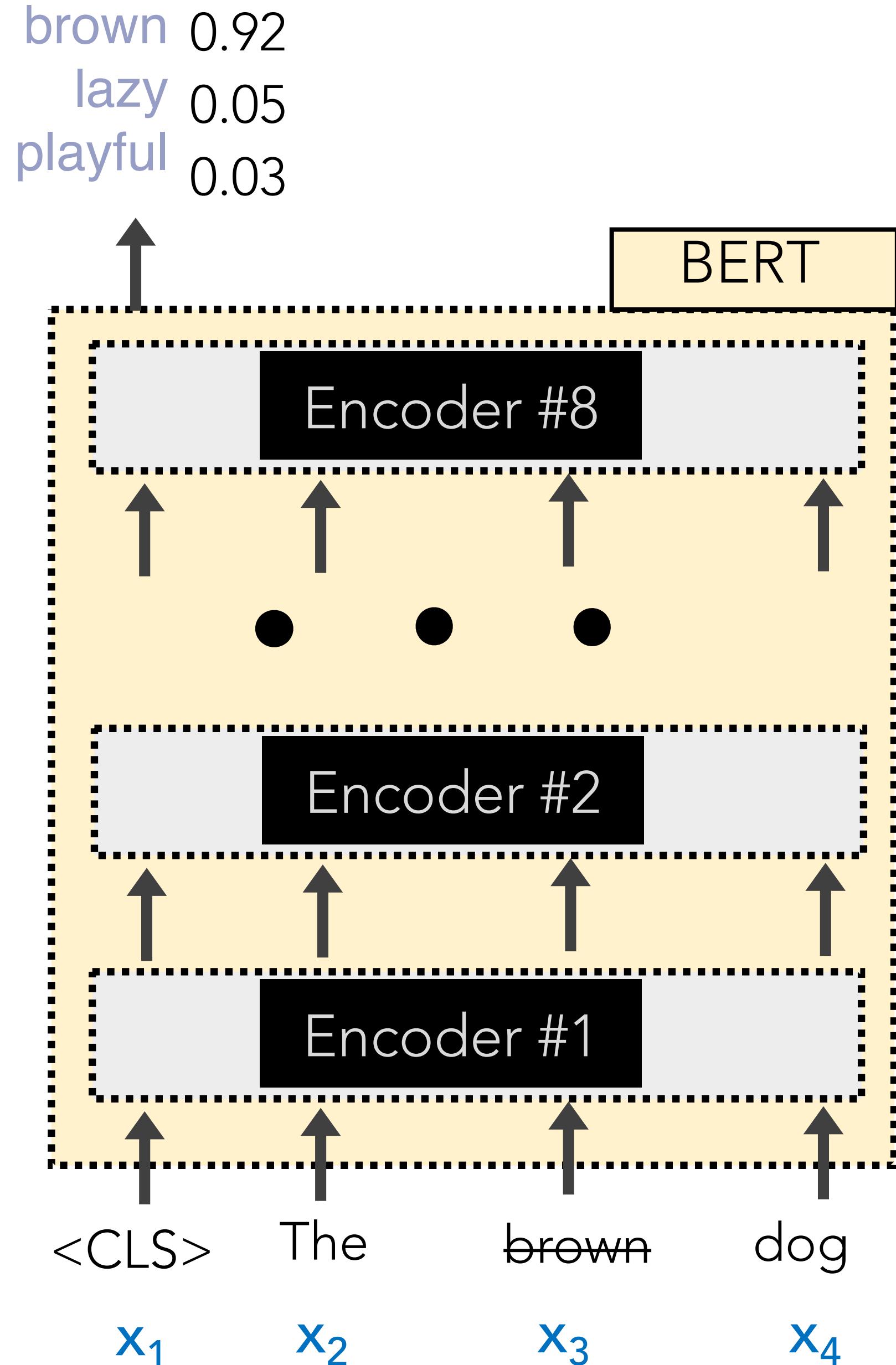
# BERT Motivation

## Bidirectional Encoder Representations from Transformers

It's a language model that builds rich representations



# BERT Architecture



BERT has 2 training objectives:

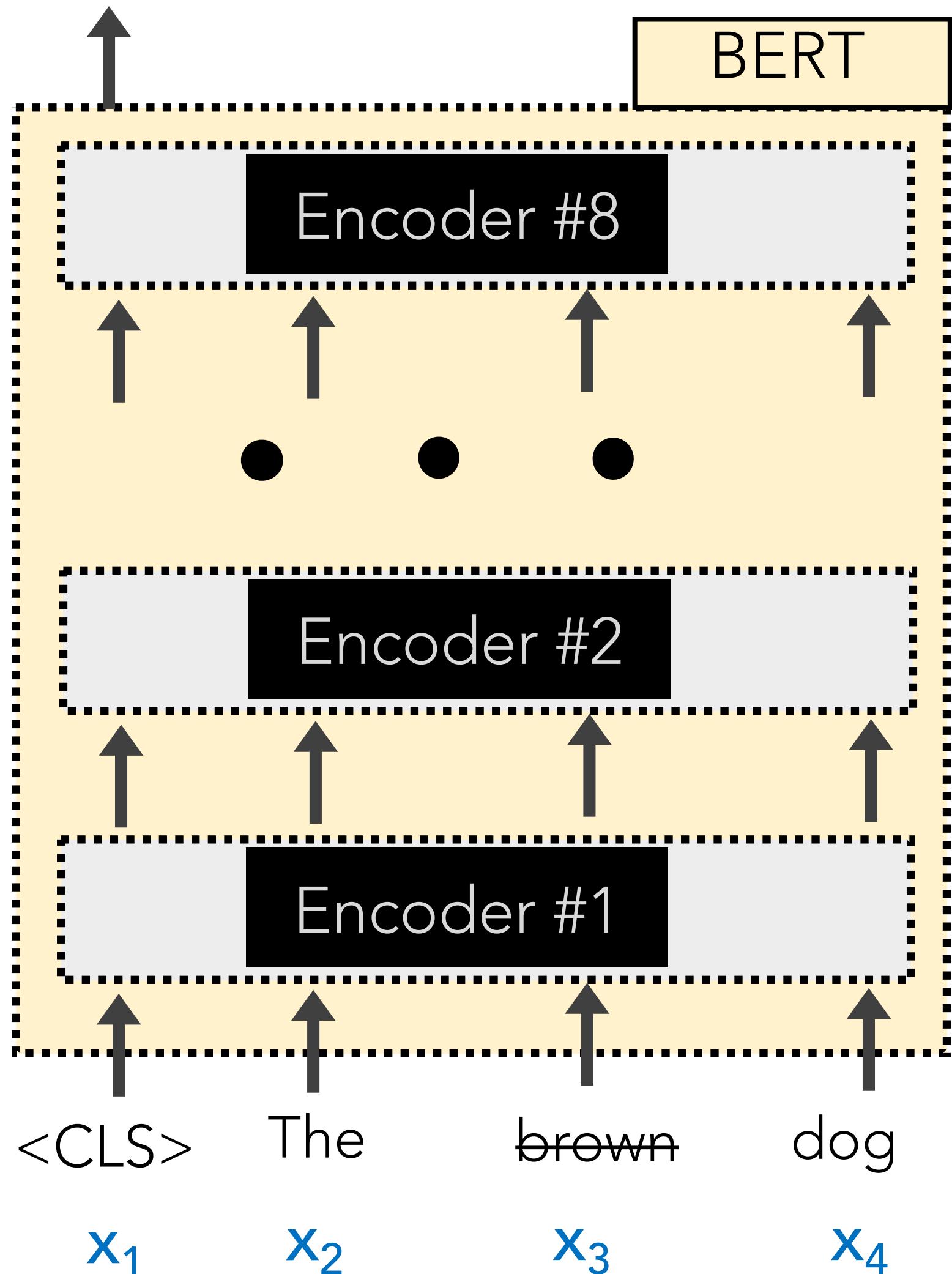
1. Predict the **Masked word** (a la CBOW)

15% of all input words are randomly masked.

- 80% become [MASK]
- 10% become unmasked again
- 10% are deliberately corrupted as wrong words

# BERT Architecture

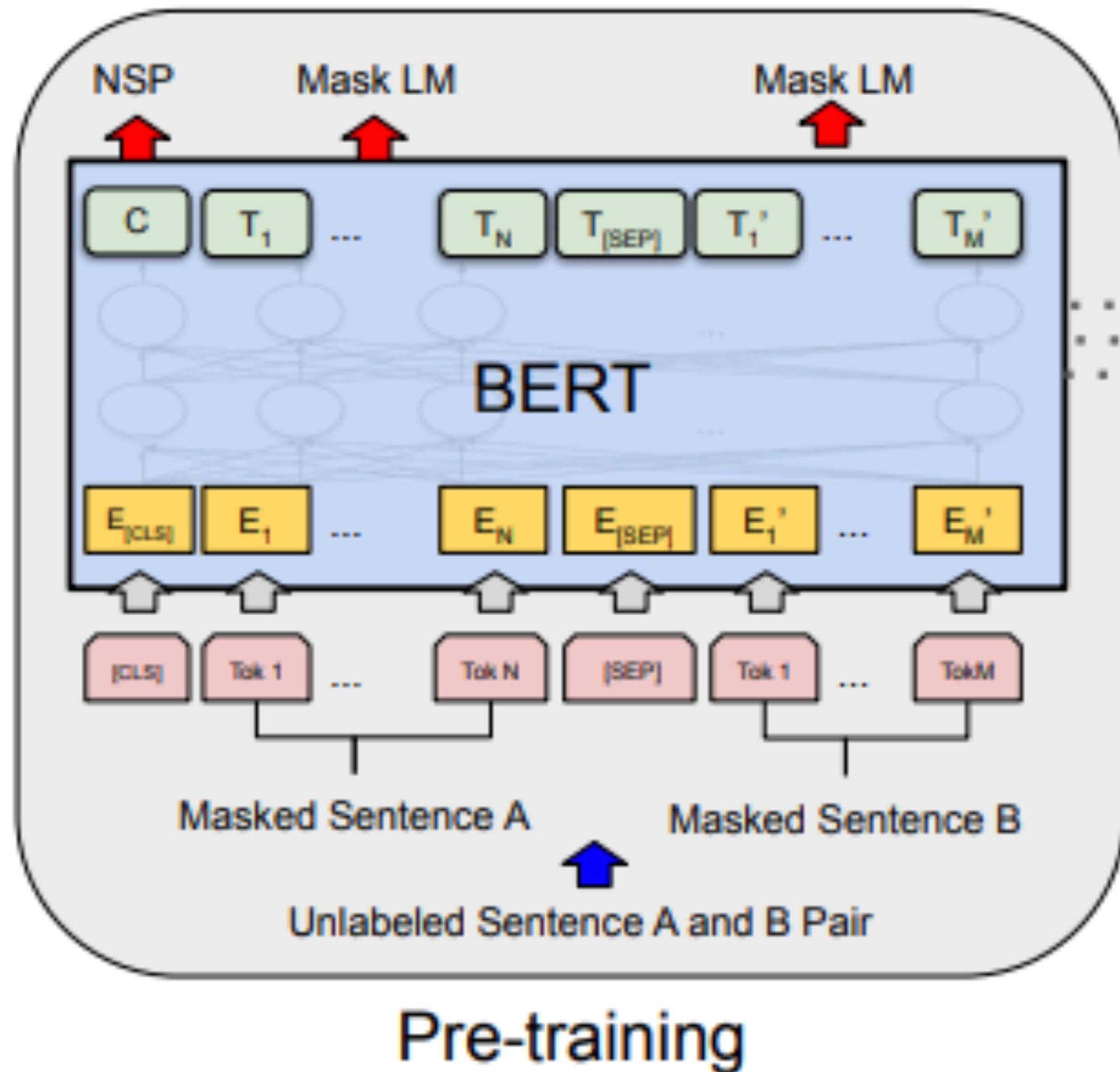
brown 0.92  
lazy 0.05  
playful 0.03



BERT has 2 training objectives:

2. Two sentences are fed in at a time. Predict if the second sentence of input truly follows the first one or not.

# BERT Architecture

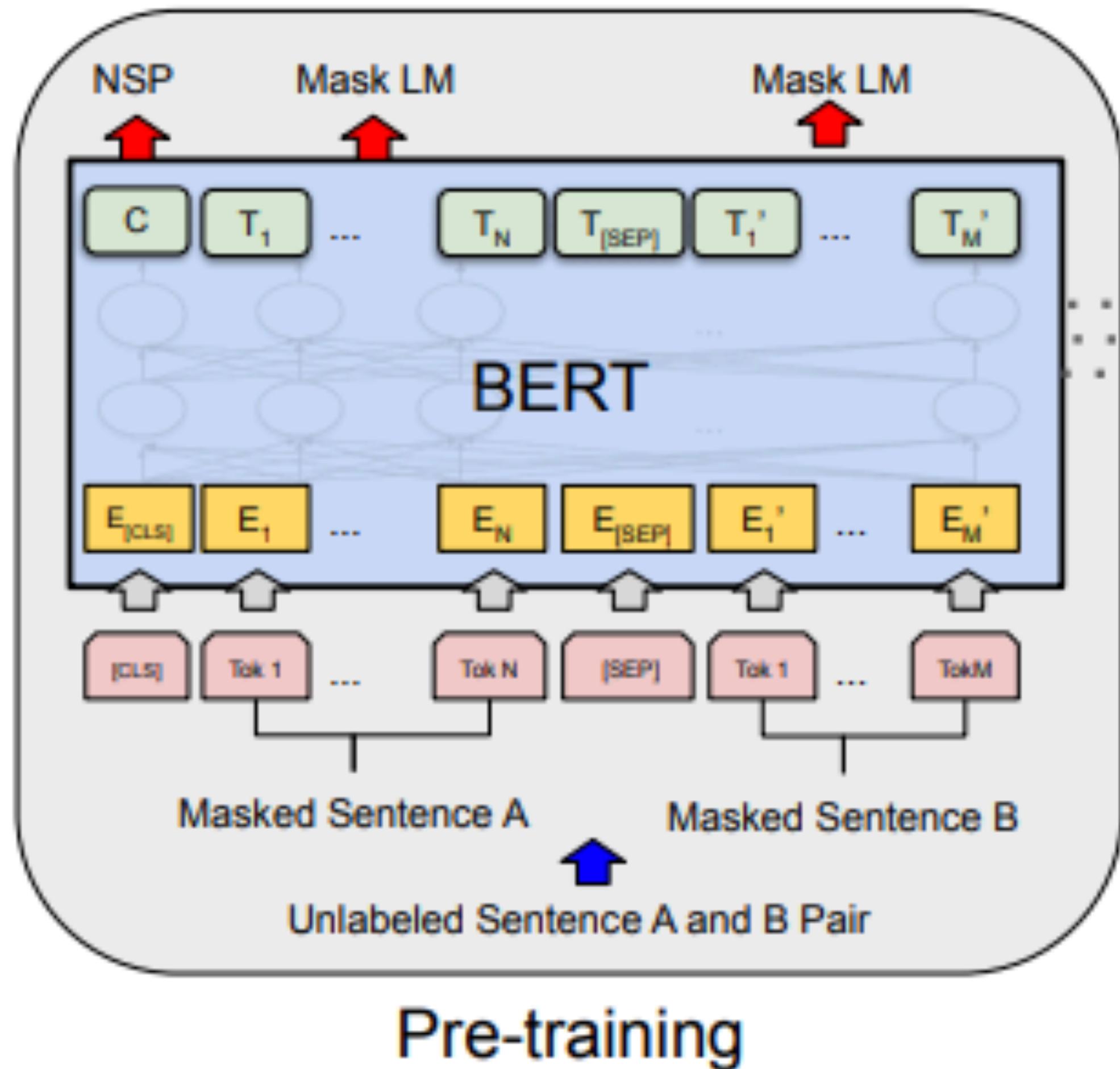


Every two sentences are separated by a **<SEP>** token.

50% of the time, the 2<sup>nd</sup> sentence is a randomly selected sentence from the corpus.

50% of the time, it truly follows the first sentence in the corpus.

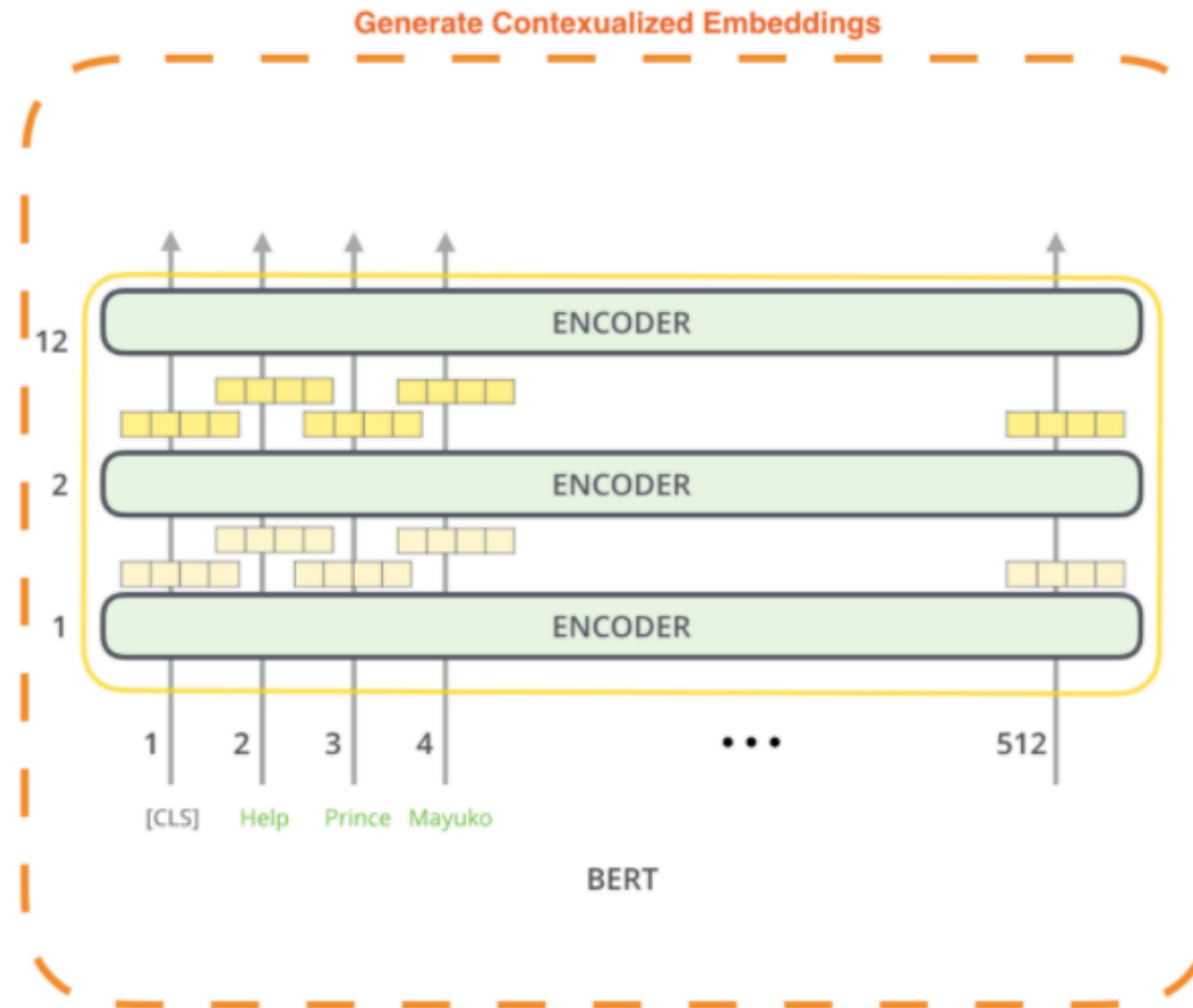
# BERT Architecture



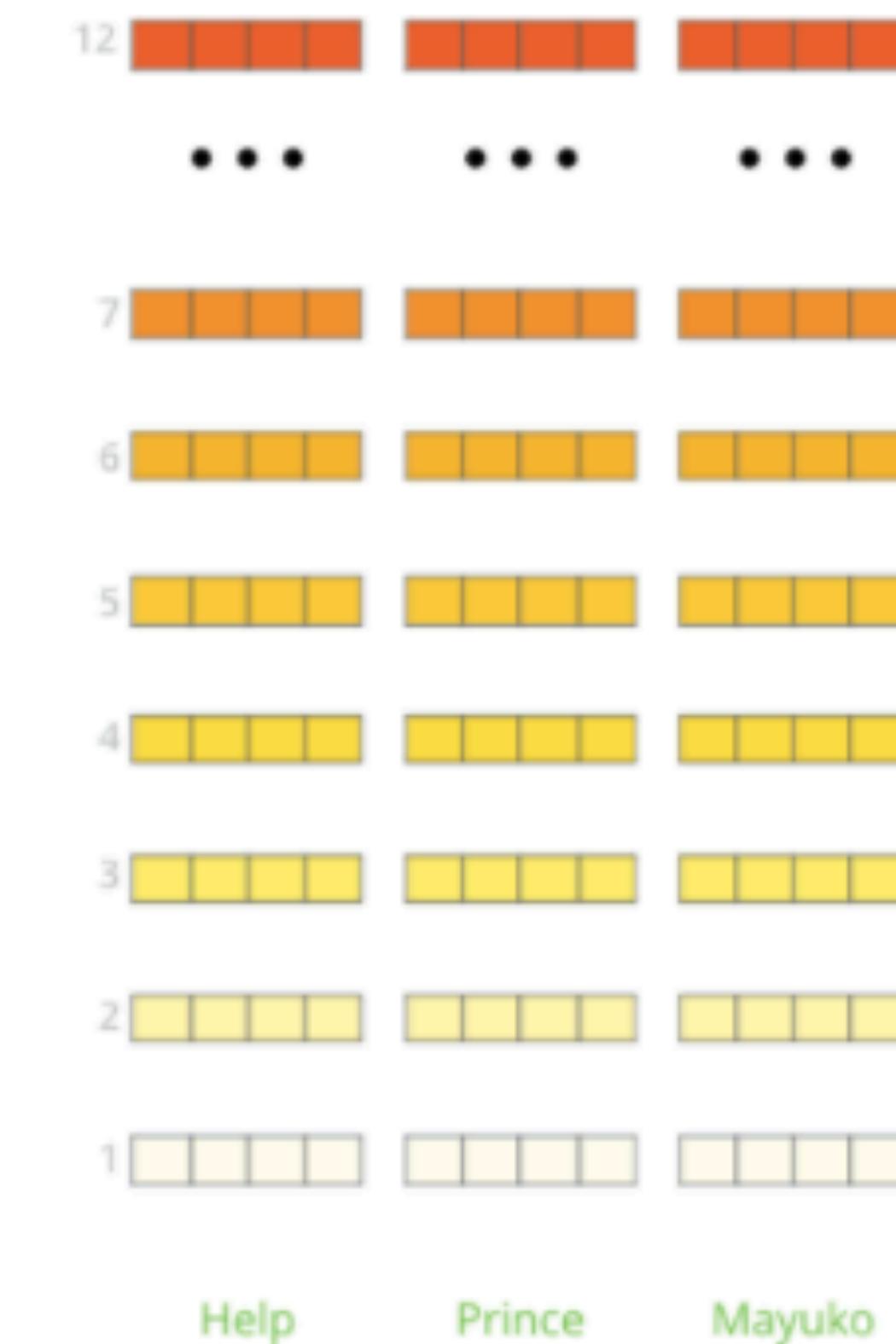
**NOTE:** BERT also embeds the inputs by their **WordPiece** embeddings.

**WordPiece** is a sub-word tokenization that learns to merge and use characters based on which pairs maximize the likelihood of the training data if added to the vocab.

# One could extract the contextualized embeddings

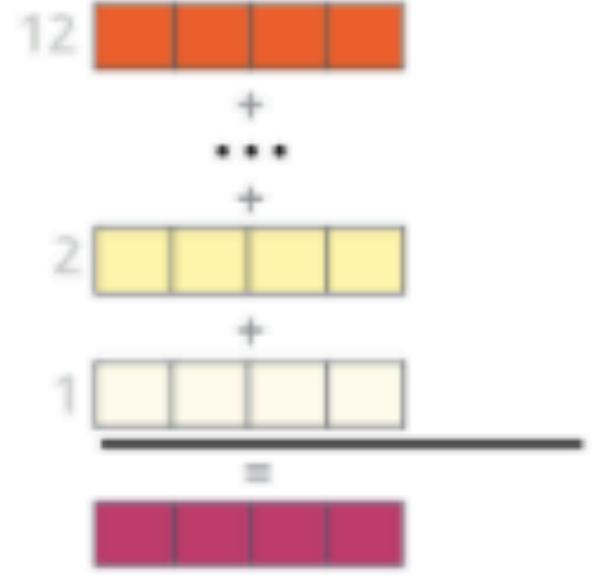
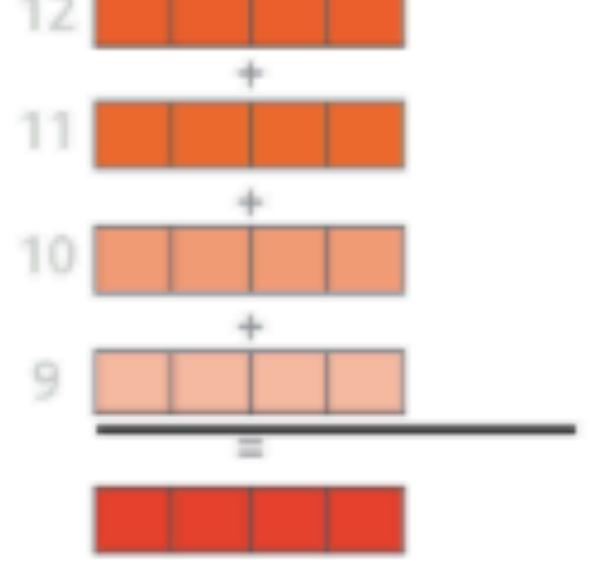


The output of each encoder layer along each token's path can be used as a feature representing that token.



But which one should we use?

# Later layers have the best contextualized embeddings

			Dev F1 Score
12		First Layer	Embedding  91.0
• • •		Last Hidden Layer	
7			94.9
6		Sum All 12 Layers	 95.5
5			
4		Second-to-Last Hidden Layer	 95.6
3			
2			
1		Sum Last Four Hidden	 95.9
			
Help		Concat Last Four Hidden	 96.1

# BERT yields state-of-the-art (SOTA) results on many tasks

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	Average -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
BERT <sub>BASE</sub>	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
BERT <sub>LARGE</sub>	<b>86.7/85.9</b>	<b>72.1</b>	<b>92.7</b>	<b>94.9</b>	<b>60.5</b>	<b>86.5</b>	<b>89.3</b>	<b>70.1</b>	<b>82.1</b>

Table 1: GLUE Test results, scored by the evaluation server (<https://gluebenchmark.com/leaderboard>).

# BERT Summary

## Strengths

- Great for embedding learning
- Powerful for transfer learning to other tasks

## Issues?

- Not easily generative (no decoders)

# What just happened!?

- RNN
- LSTM
- Attention
- Self-Attention
- Transformers

## What is to come...

- Tutorials **May 3rd + May 10th**
- Next lecture. In two weeks! **May 14th**
- LLMs
- In-context Learning
- Prompt Engineering
- HW1 due!