# CS388: Natural Language Processing
# Lecture 6: Language Modeling, Self Attention


Greg Durrett

The University of Texas at Austin

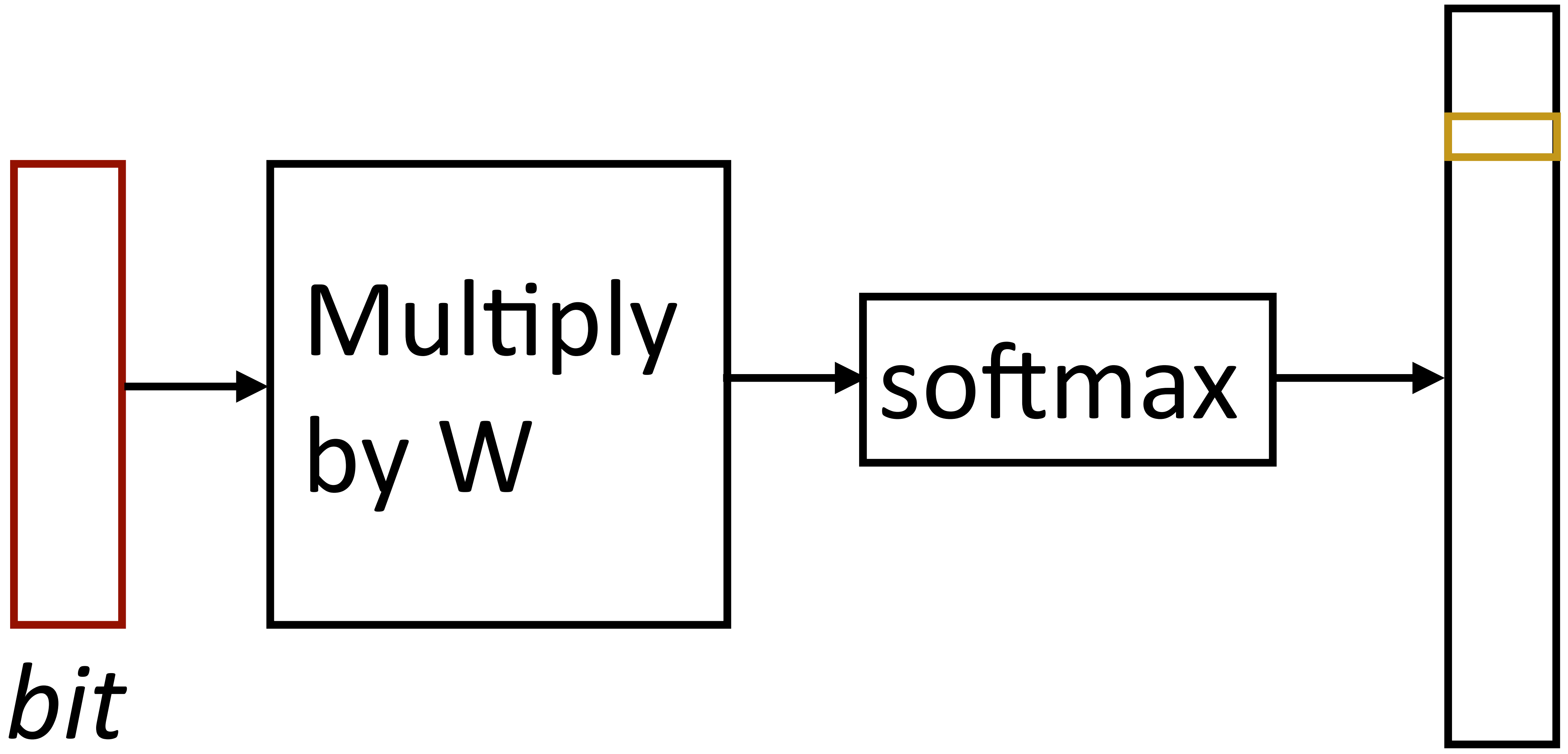
# Administrivia

- Project 1 due today
- Project 2 released tonight

# Recap: Skip-Gram

- Predict one word of context from word

*the dog bit the man*




gold = *dog*

$$P(w'|w) = \mathrm{softmax}(We(w))$$

- Parameters: *d* x |V| vectors, |V| x *d* output parameters (W) (also usable as vectors!)
- Predicting the next word from a word will be similar to language modeling (focus of this lecture!)

Mikolov et al. (2013)

# Recap: GloVe

- Objective = $\sum_{i,j} f(\text{count}(w_i, c_j)) \left(w_i^\top c_j + a_i + b_j - \log \text{count}(w_i, c_j)\right)\right)^2$

| | *the* | *dog* | *cat* | *ran* |
|---|---|---|---|---|
| *the* | 0 | 200 | 200 | 0 |
| *dog* | 200 | 0 | 0 | 15 |
| *cat* | 200 | 0 | 0 | 15 |
| *ran* | 0 | 15 | 15 | 0 |

Linear regression with 16 points: each element is plugged into the above equation

+ constant = log count of pair

(made up values — matrix will generally be symmetric, though)

Pennington et al. (2014)

# Recap: Using Embeddings

- Approach 1: learn embeddings as parameters from your data
- Approach 2: initialize using GloVe, keep fixed
- Approach 3: initialize using GloVe, fine-tune

- Nearly all modern transfer learning uses Approach 3 (e.g., fine-tuning BERT). And you don't just fine-tune embeddings, but instead use an entire language model

# Today

- Language modeling intro
- Neural language modeling
- Self-attention
- Multi-head self-attention
- Positional encodings (if time)

# Language Modeling

# Language Modeling

- Fundamental task in both linguistics and NLP: can we determine of a sentence is *acceptable* or not?
- Related problem: can we evaluate if a sentence is grammatical? Plausible? Likely to be uttered?
- Language models: place a distribution P(***w***) over strings ***w*** in a language. This is related to all of these tasks but doesn't exactly map onto them
- Today: autoregressive models $P(\mathbf{w}) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots$
- Turns out this is also useful as the backbone pre-training task! (But it originated with modeling of grammatical/plausible sentences)

# N-gram Language Models

$$P(\mathbf{w}) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots$$

- n-gram models: distribution of next word is a categorical conditioned on previous n-1 words $P(w_i|w_1, \ldots, w_{i-1}) = P(w_i|w_{i-n+1}, \ldots, w_{i-1})$
- Markov property: don't remember all the context but only consider a few previous words

I visited San ______

put a distribution over the next word

2-gram: P(w | San)

3-gram: P(w | visited San)

4-gram: P(w | I visited San)

# N-gram Language Models

$$P(\mathbf{w}) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots$$

- n-gram models: distribution of next word is a categorical conditioned on previous n-1 words $P(w_i|w_1, \ldots, w_{i-1}) = P(w_i|w_{i-n+1}, \ldots, w_{i-1})$

$$P(w|\text{visited San}) = \frac{\text{count}(\text{visited San}, w)}{\text{count}(\text{visited San})}$$

3-gram probability, maximum likelihood estimate from a corpus (remember: count and normalize for MLE)

- Just relies on counts, even in 2008 could scale up to 1.3M word types, 4B n-grams (all 5-grams occurring >40 times on the Web)

# Smoothing N-gram Language Models

- What happens when we scale to longer contexts?

$P(w|\text{to})$ *to* occurs 1M times in corpus

$P(w|\text{go to})$ *go to* occurs 50,000 times in corpus

$P(w|\text{to go to})$ *to go to* occurs 1500 times in corpus

$P(w|\text{want to go to})$ *want to go to*: only 100 occurrences

- Probability counts get very sparse, and we often want information from 5+ words away
- What can we do?

# Smoothing N-gram Language Models

I visited San ______   put a distribution over the next word

- Smoothing is very important, particularly when using 4+ gram models

$$P(w|\text{visited San}) = (1-\lambda)\frac{\text{count}(\text{visited San}, w)}{\text{count}(\text{visited San})} + \lambda\frac{\text{count}(\text{San}, w)}{\text{count}(\text{San})}$$

smooth this too!

- One technique is "absolute discounting:" subtract off constant *k* from numerator, set lambda to make this normalize (*k*=1 is like leave-one-out)

$$P(w|\text{visited San}) = \frac{\text{count}(\text{visited San}, w) - k}{\text{count}(\text{visited San})} + \lambda\frac{\text{count}(\text{San}, w)}{\text{count}(\text{San})}$$

- Smoothing schemes get very complex!

# The Power of Language Modeling

My name ______

- One good option (*is*)?

My name is ______

- Flat distribution over many alternatives. But hard to get a good distribution?

I visited San ______

- Requires some knowledge but not one right answer

The capital of Texas is ______

- Requires more knowledge (one answer…or is there?)

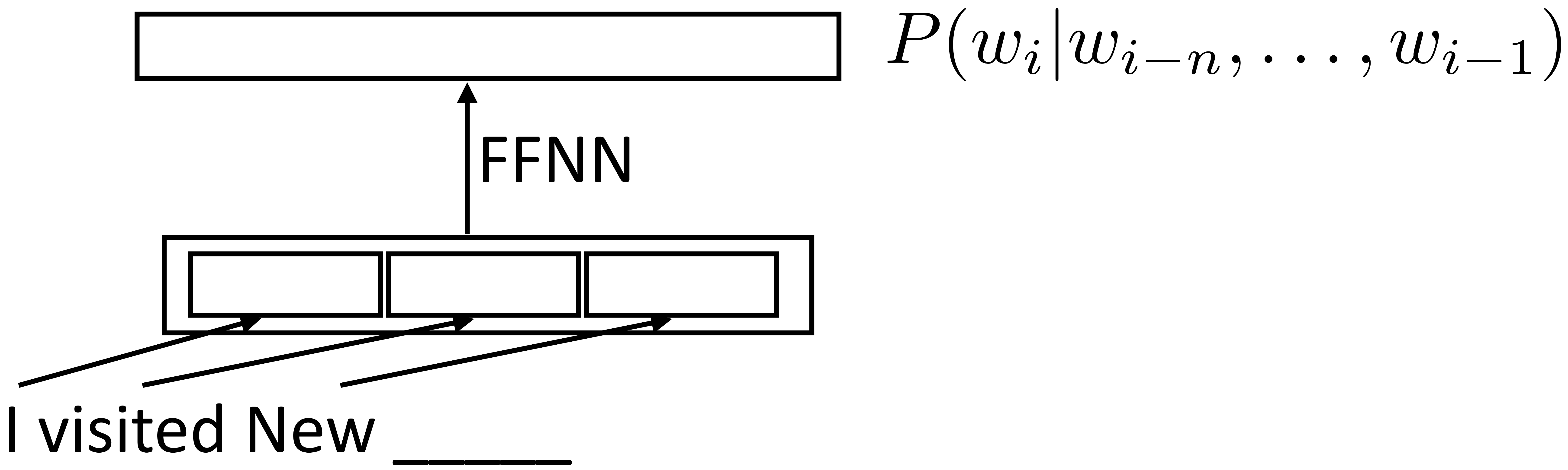
The casting and direction were top notch. Overall I thought the movie was ____

- Requires basically doing sentiment analysis!

# Neural Language Modeling

# Neural Language Models

- Early work: feedforward neural networks looking at context




- Slow to train over lots of data! But otherwise this seems okay?

Bengio et al. (2003)

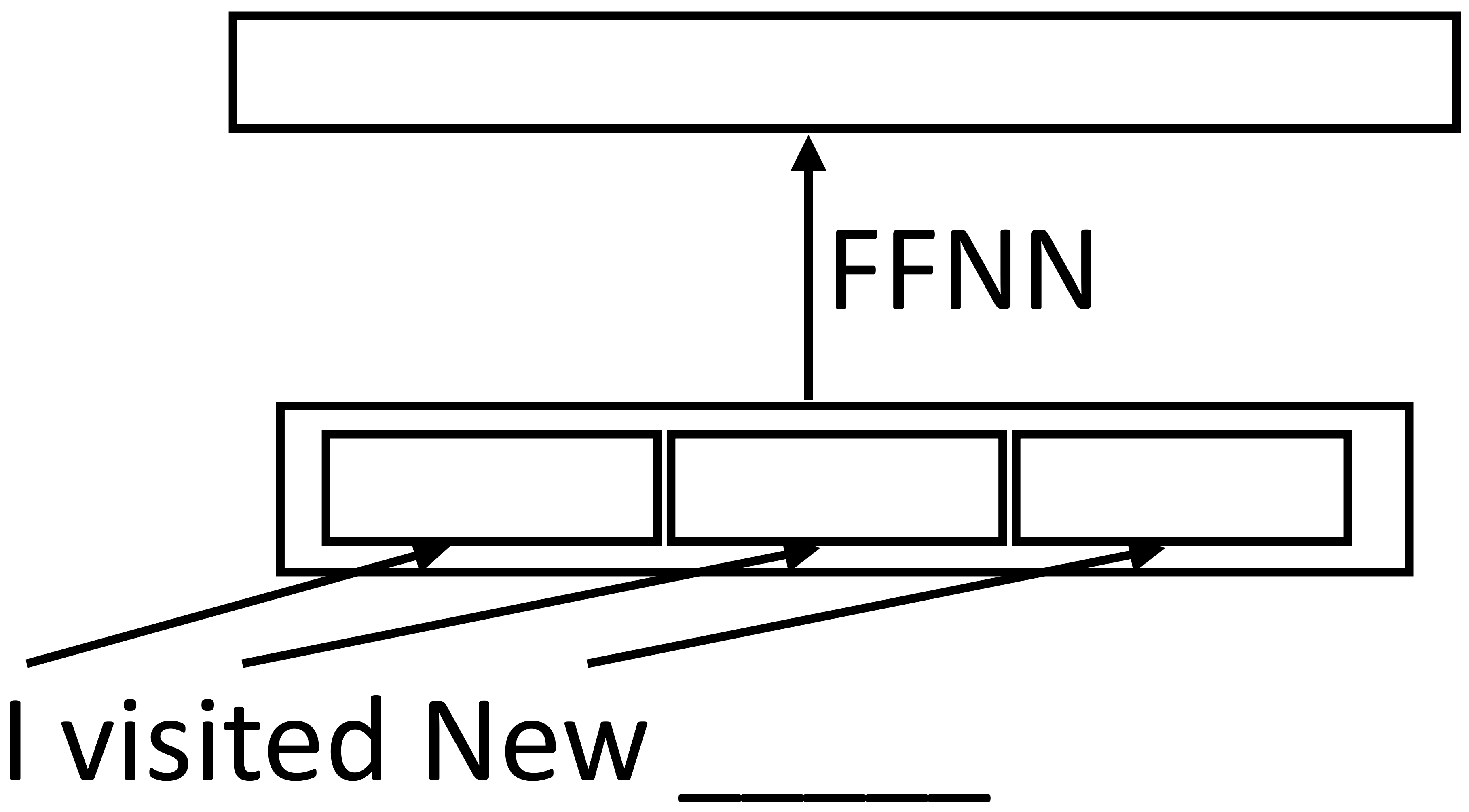
# Problems with FFNNs

***x*** = *I visited New York. I had a really fun time going up the* ___

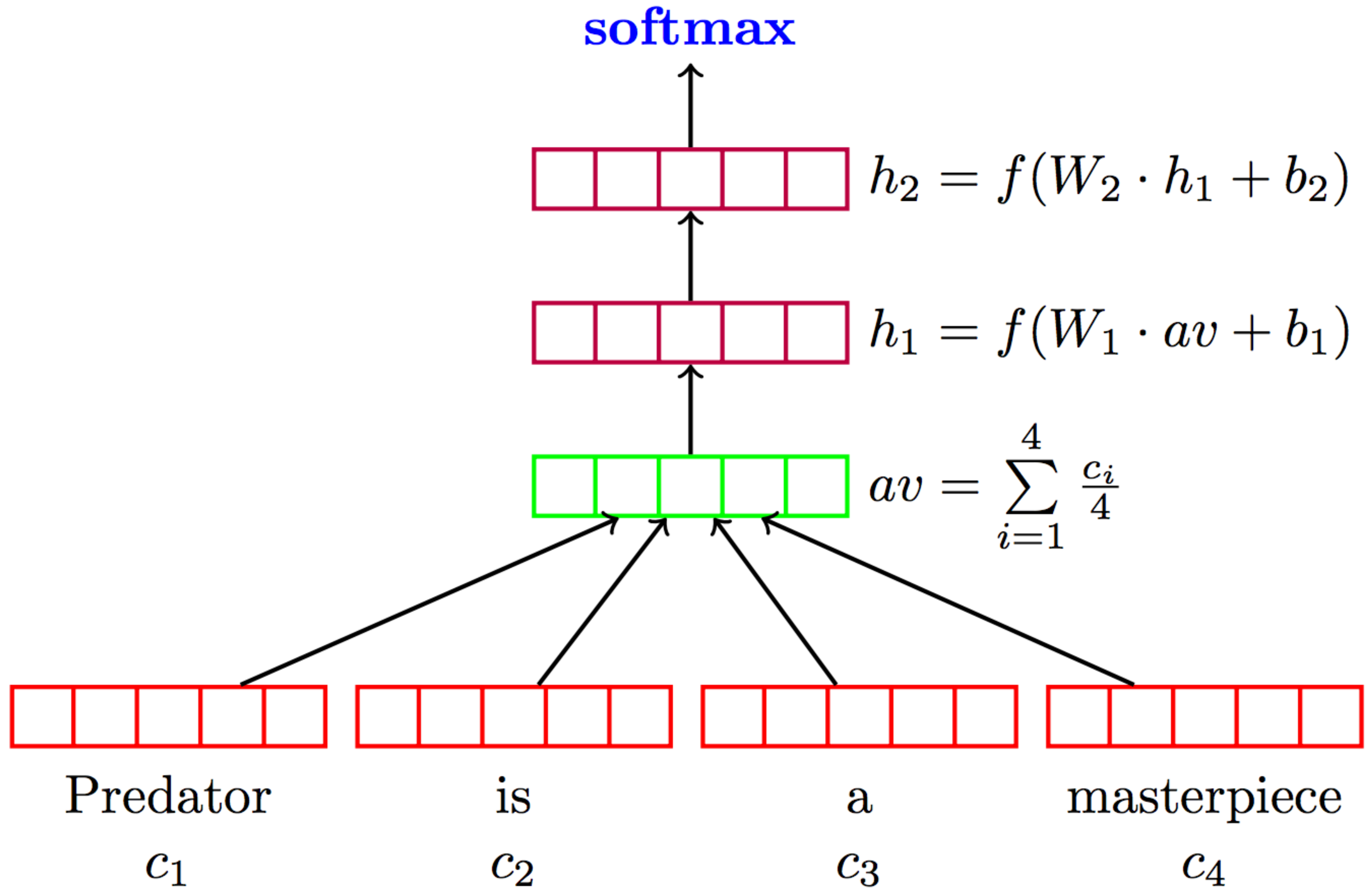
- What are some words that can show up here? How do we know?

- What do we learn from this example?

# Challenges of Neural Language Modeling




FFNN




DAN

- Advantages and disadvantages of these?

# Contextualized Embeddings

- Both RNNs and Transformers (and other models) can produce *contextualized embeddings*

$\boldsymbol{e} = (e_1, e_2, \ldots, e_n)$ $e_i = f(x_1, x_2, \ldots, x_i)$

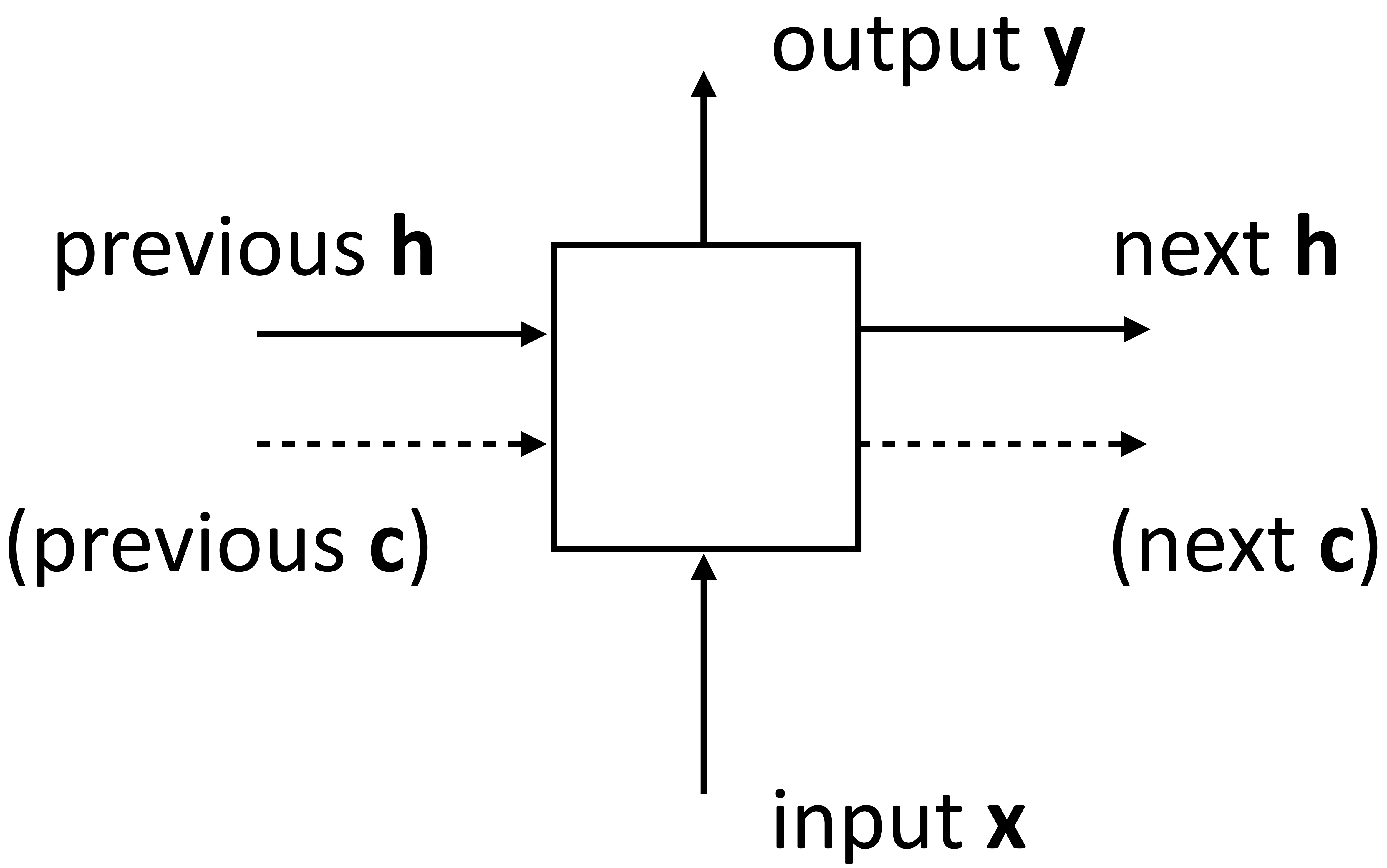
$\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$

- unidirectional representation (only looks at past words)

$\boldsymbol{x}$ = *I visited New York. I had a really fun time going up the ___*

- Can also have bidirectional embedding representations, but learning these needs *masked language models* (later in the course)
- One solution: $\boldsymbol{e}(\boldsymbol{x}) = \mathrm{f}(\boldsymbol{x}_{-1}, \textit{the})$

# RNNs: Why not?




- Slow. They do not parallelize and there are O(n) non-parallel operations to encode n items
- Even modifications like LSTMs still don't enable learning over very long sequences. Transformers can scale to thousands of words!

# (Self-)Attention

# Running Example

- Fixed-length sequence of 0s and 1s

0000000

0100001

0100101

0000101

100000000000000000000000000000000000000000001

- All zeroes = last digit is 0; any 1 = last digit is 1
- **Attention**: method to access arbitrarily* far back in context from this point

- RNNs generally struggle with this; remembering context for many positions is hard (though of course they can do this simplified example — you can even hand-write weights to do it!)

# Keys and Query

- Keys: embedded versions of the sentence; query: what we want to find

Assume 0 = [1, 0]; 1 = [0, 1] (one-hot encodings of the tokens); call these $e_i$

Step 1: Compute scores for each key

keys $k_i$

[1, 0] [1, 0] [0, 1] [1, 0]

0 0 1 0

query: $q$ = [0, 1] (we want to find 1s)

$s_i = k_i^T q$

0 0 1 0

# Attention

Step 1: Compute scores for each key

keys $k_i$

[1, 0] [1, 0] [0, 1] [1, 0]

0 0 1 0

query: $q$ = [0, 1] (we want to find 1s)

$s_i = k_i^T q$

0 0 1 0

Step 2: softmax the scores to get probabilities α

0 0 1 0 => (1/6, 1/6, 1/2, 1/6) if we assume e=3

Step 3: compute output values by multiplying embs. by alpha + summing

result = sum($\alpha_i e_i$) = 1/6 [1, 0] + 1/6 [1, 0] + 1/2 [0, 1] + 1/6 [1, 0] = [1/2, 1/2]

# Attention

keys $k_i$

[1, 0] [1, 0] [0, 1] [1, 0]

0 0 1 0

query: $q$ = [0, 1] (we want to find 1s)

(1/6, 1/6, 1/2, 1/6) if we assume e=3

result = sum($\alpha_i e_i$) = 1/6 [1, 0] + 1/6 [1, 0] + 1/2 [0, 1] + 1/6 [1, 0] = [1/2, 1/2]

How does this differ from just averaging the vectors (DAN)?

What if we have a very very long sequence?

# New Keys

keys $k_i$

[1, 0] [1, 0] [0, 1] [1, 0]

0 0 1 0

query: $q$ = [0, 1] (we want to find 1s)

We can make attention more peaked by not setting keys equal to embeddings.

$k_i = W^K e_i$

$$W^K = \begin{matrix} 10 & 0 \\ 0 & 10 \end{matrix}$$

[10, 0][10, 0][0, 10][10, 0]

0 0 1 0

What will new attention values be with these keys?

# Attention, Formally

- Original "dot product" attention: $s_i = k_i^T q$
- Scaled dot product attention: $s_i = k_i^T W q$
- Equivalent to having two weight matrices: $s_i = (W^K k_i)^T (W^Q q)$
- Other forms exist: Luong et al. (2015), Bahdanau et al. (2014) present some variants (originally for machine translation)

# Self-Attention

- Self-attention: **every word is both a key and a query simultaneously**

Q: seq len x d matrix  (d = embedding dimension = 2 for these slides)

K: seq len x d matrix

$$W^Q = \begin{matrix} 0 & 1 \\ 0 & 1 \end{matrix}$$

no matter what the value is, we're going to look for 1s

$$W^K = \begin{matrix} 10 & 0 \\ 0 & 10 \end{matrix}$$

"booster" as before

Note: there are many ways to set up these weights that will be equivalent to this

# Self-Attention

$$E = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$W^Q = \begin{matrix} 0 & 1 \\ 0 & 1 \end{matrix}$$

$$W^K = \begin{matrix} 10 & 0 \\ 0 & 10 \end{matrix}$$

$$Q = E\,(W^Q) = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}$$

$$K = E\,(W^K) = \begin{pmatrix} 10 & 0 \\ 10 & 0 \\ 0 & 10 \\ 10 & 0 \end{pmatrix}$$

Scores $S = QK^T$ $\quad S_{ij} = q_i \cdot k_j$

len x len = (len x d) x (d x len)

Let’s compute these now!

# Self-Attention

$$E = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$W^Q = \begin{matrix} 0 & 1 \\ 0 & 1 \end{matrix} \qquad W^K = \begin{matrix} 10 & 0 \\ 0 & 10 \end{matrix}$$

$$Q = E\,(W^Q) = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \qquad K = E\,(W^K) = \begin{pmatrix} 10 & 0 \\ 10 & 0 \\ 0 & 10 \\ 10 & 0 \end{pmatrix}$$

Scores $S = QK^T$ $\quad S_{ij} = q_i \cdot k_j$

len x len = (len x d) x (d x len)

Final step: softmax to get attentions A, then output is AE

*technically it's $A\,(EW^V)$, using a values matrix $V = EW^V$

# Self-Attention (Vaswani et al.)

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

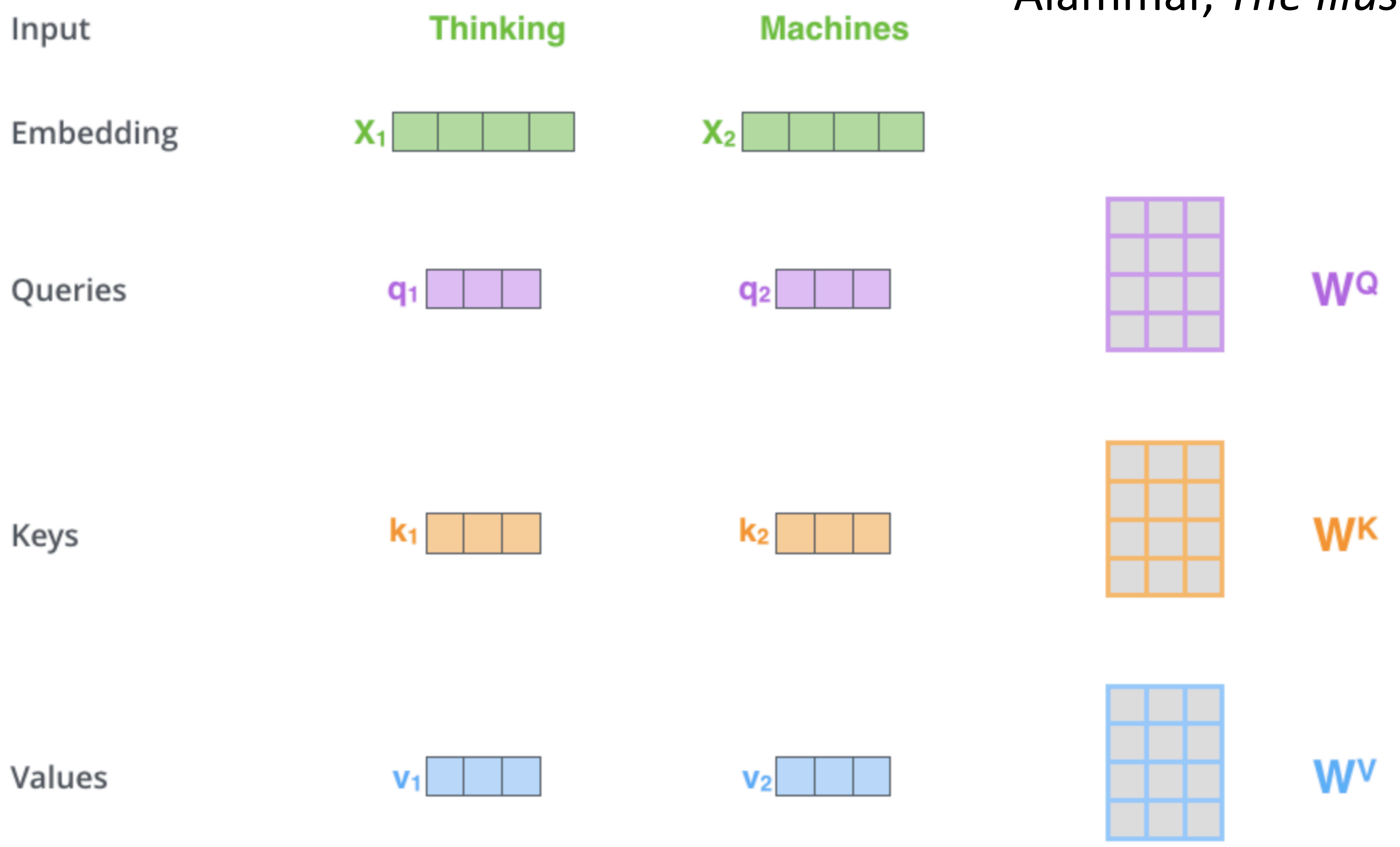
$Q = EW^Q,\ K = EW^K,\ V = EW^V$

- Normalizing by $\sqrt{d_k}$ helps control the scale of the softmax, makes it less peaked
- This is just one head of self-attention — produce multiple heads via randomly initialize parameter matrices (more in a bit)

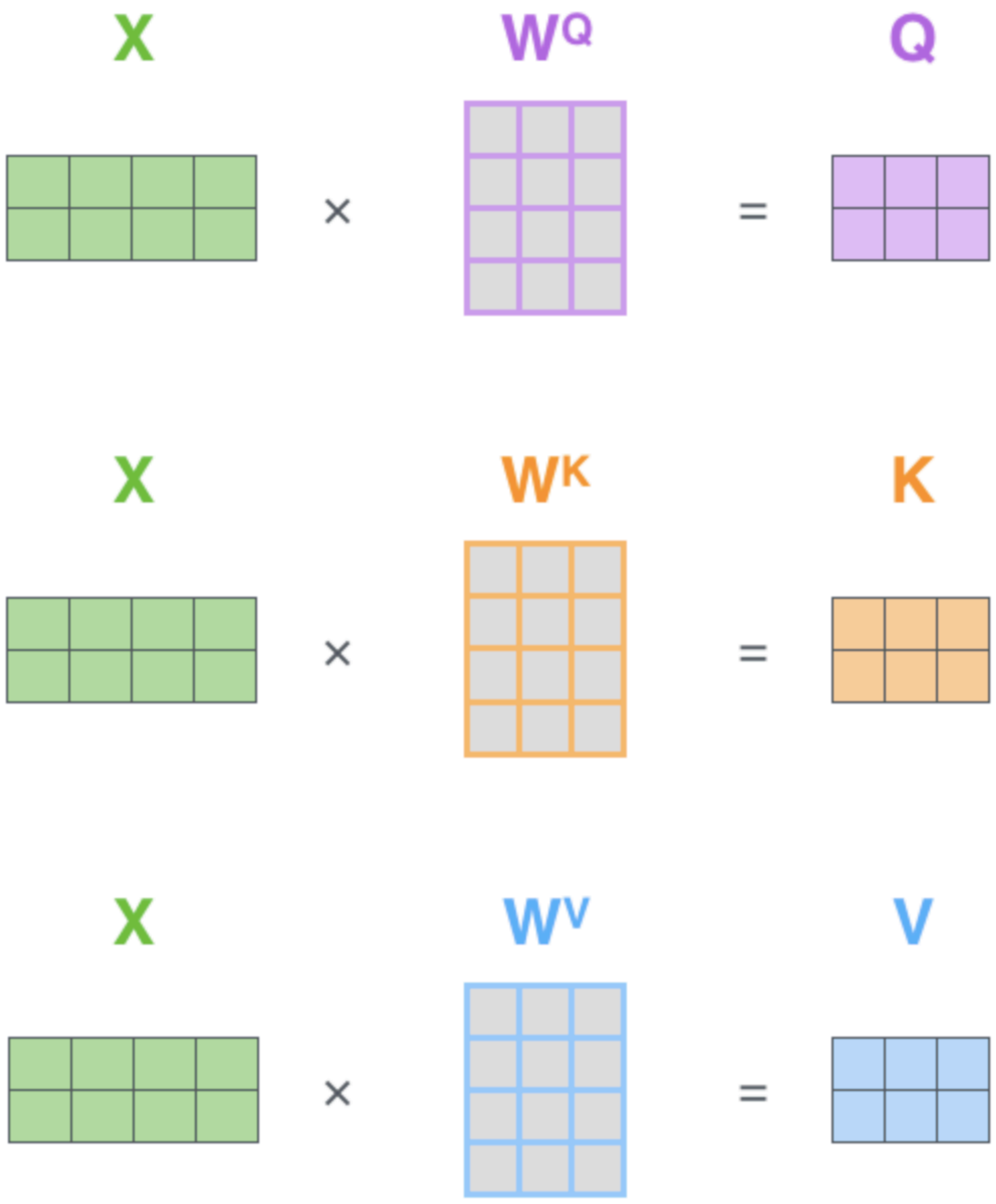
Vaswani et al. (2017)

# Self-Attention

Alammar, *The Illustrated Transformer*

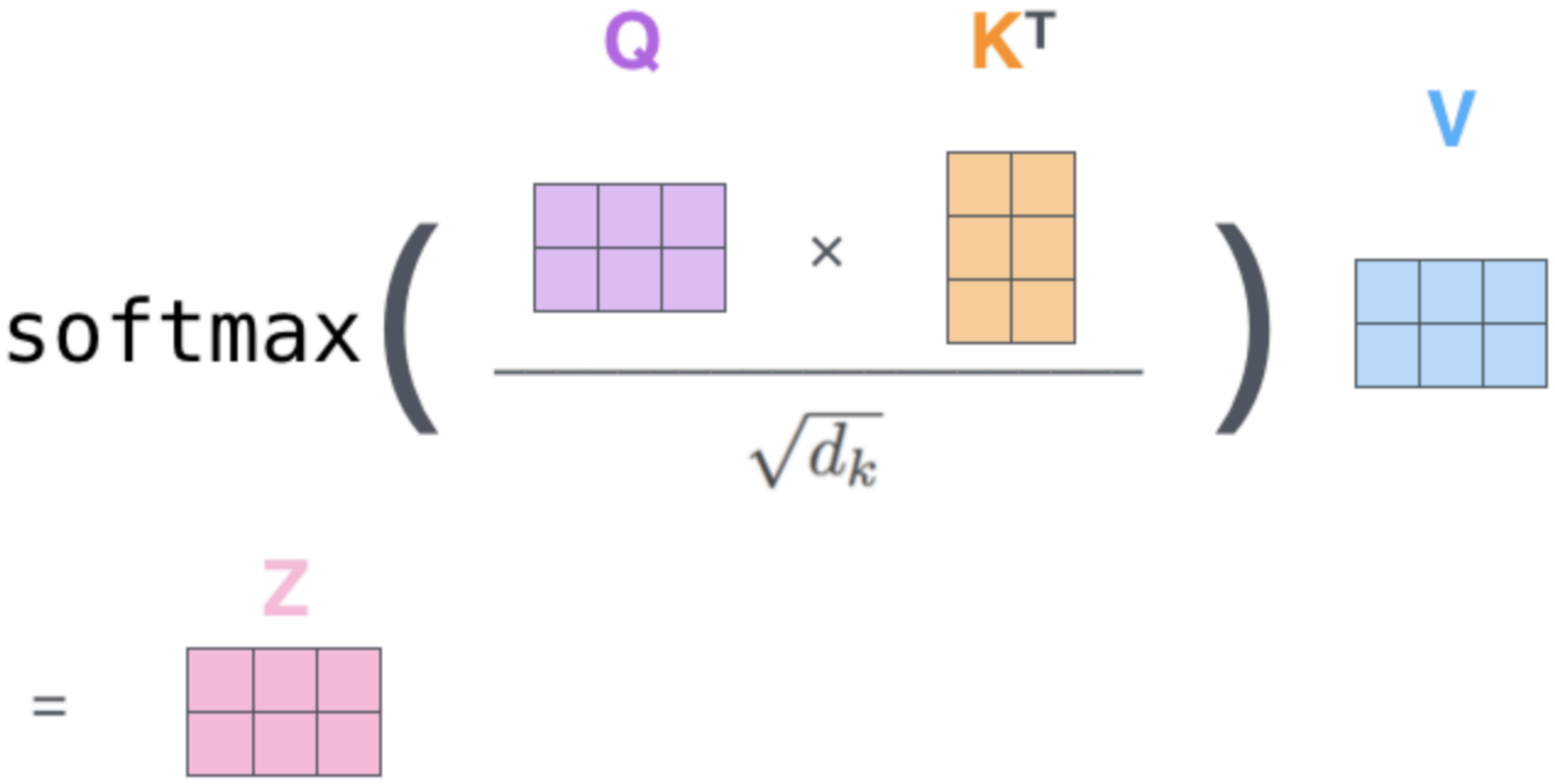


# Self-Attention




Alammar, *The Illustrated Transformer*

sent len x sent len (attn for each word to each other)




sent len x hidden dim

Z is a weighted combination of V rows

# Properties of Self-Attention

| 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) | $O(r \cdot n \cdot d)$ | $O(1)$ | $O(n/r)$ |

- *n* = sentence length, *d* = hidden dim, *k* = kernel size, *r* = restricted neighborhood size
- **Quadratic complexity**, but O(1) sequential operations (not linear like in RNNs) and O(1) "path" for words to inform each other

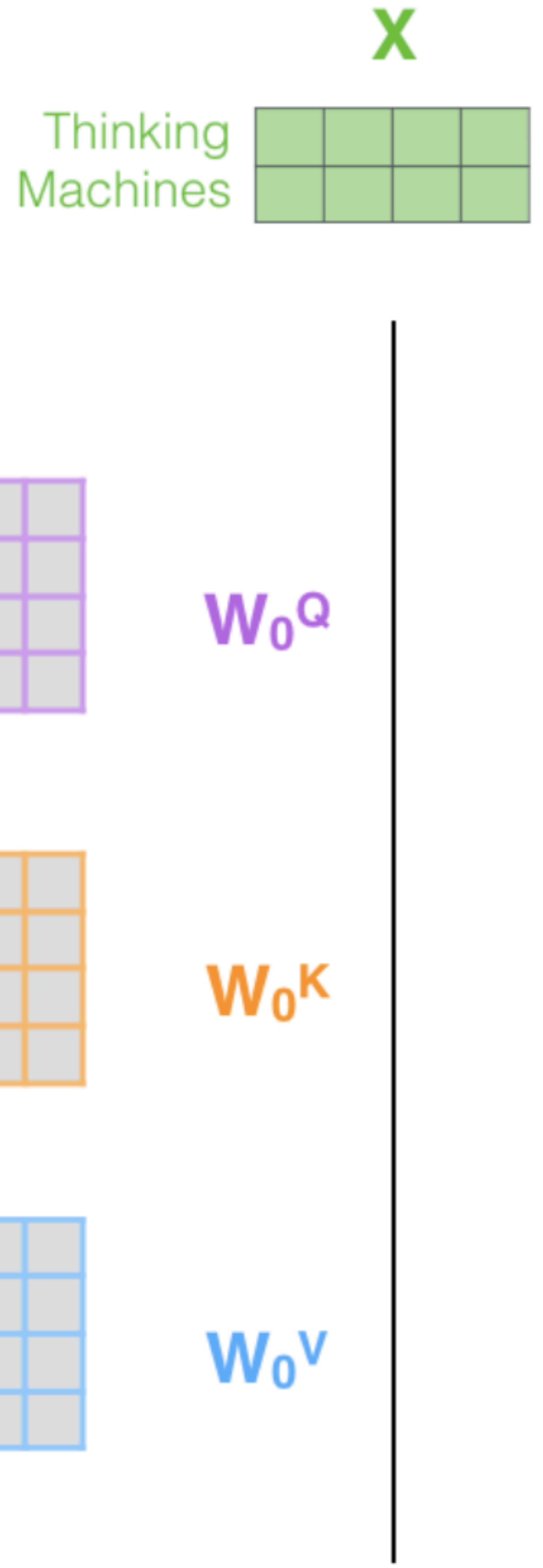
Vaswani et al. (2017)

# Multi-Head Self-Attention

# Multi-head Self-Attention

Just duplicate the whole computation with different weights:

Alammar, *The Illustrated Transformer*



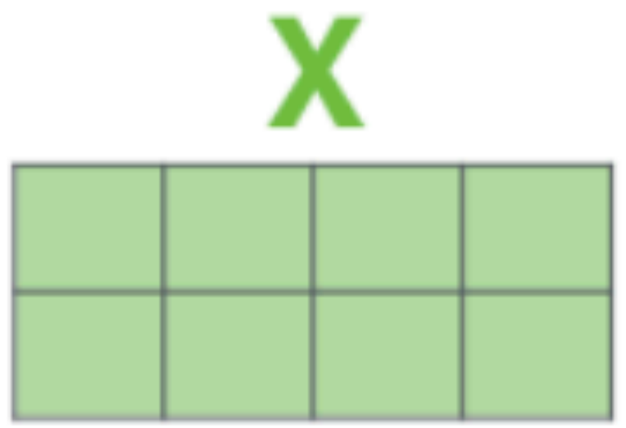
# Multi-head Self-Attention

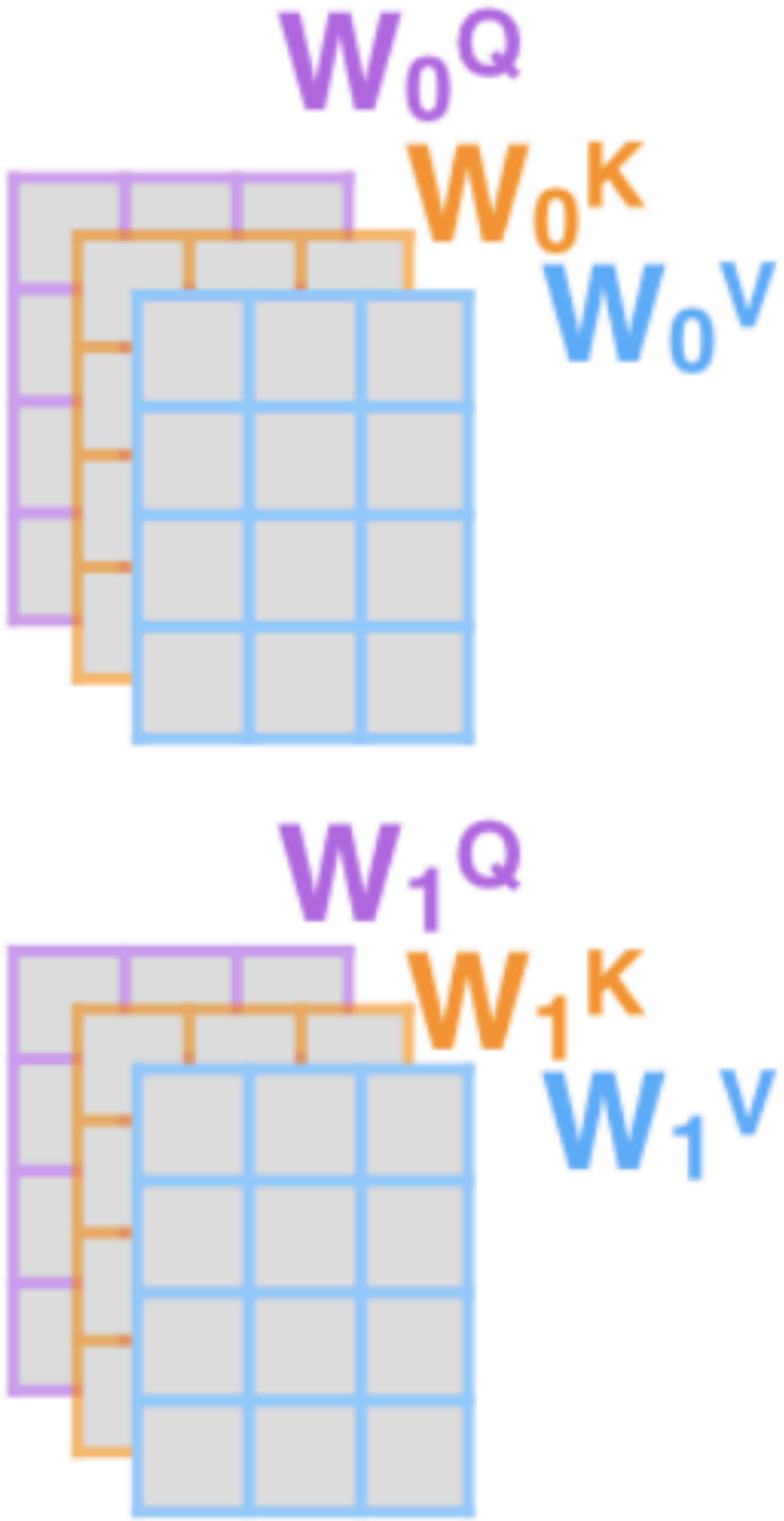
1) This is our input sentence*

2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

Thinking
Machines




* In all encoders other than #0, we don’t need embedding. We start directly with the output of the encoder right below this one

# Multi-head Self-Attention

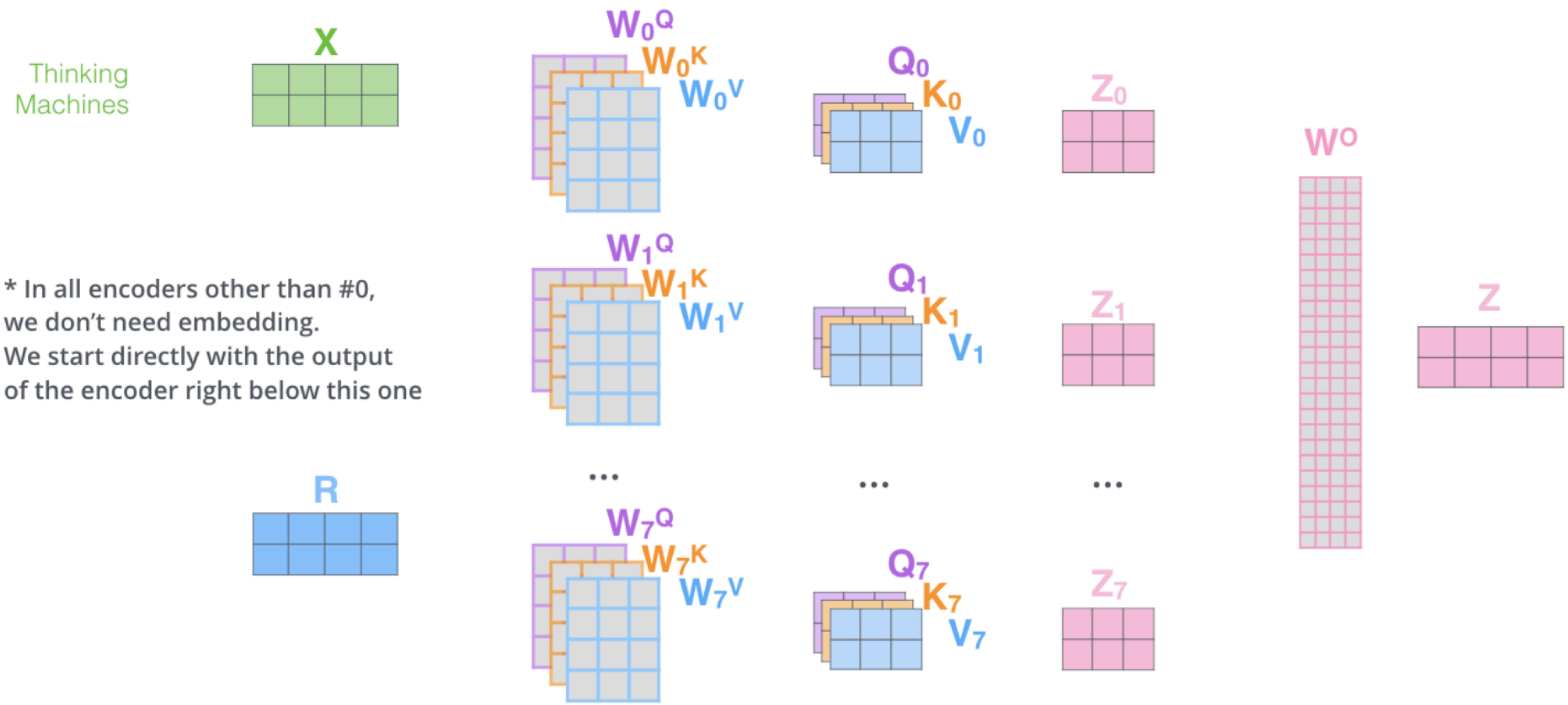
1) This is our input sentence*

2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

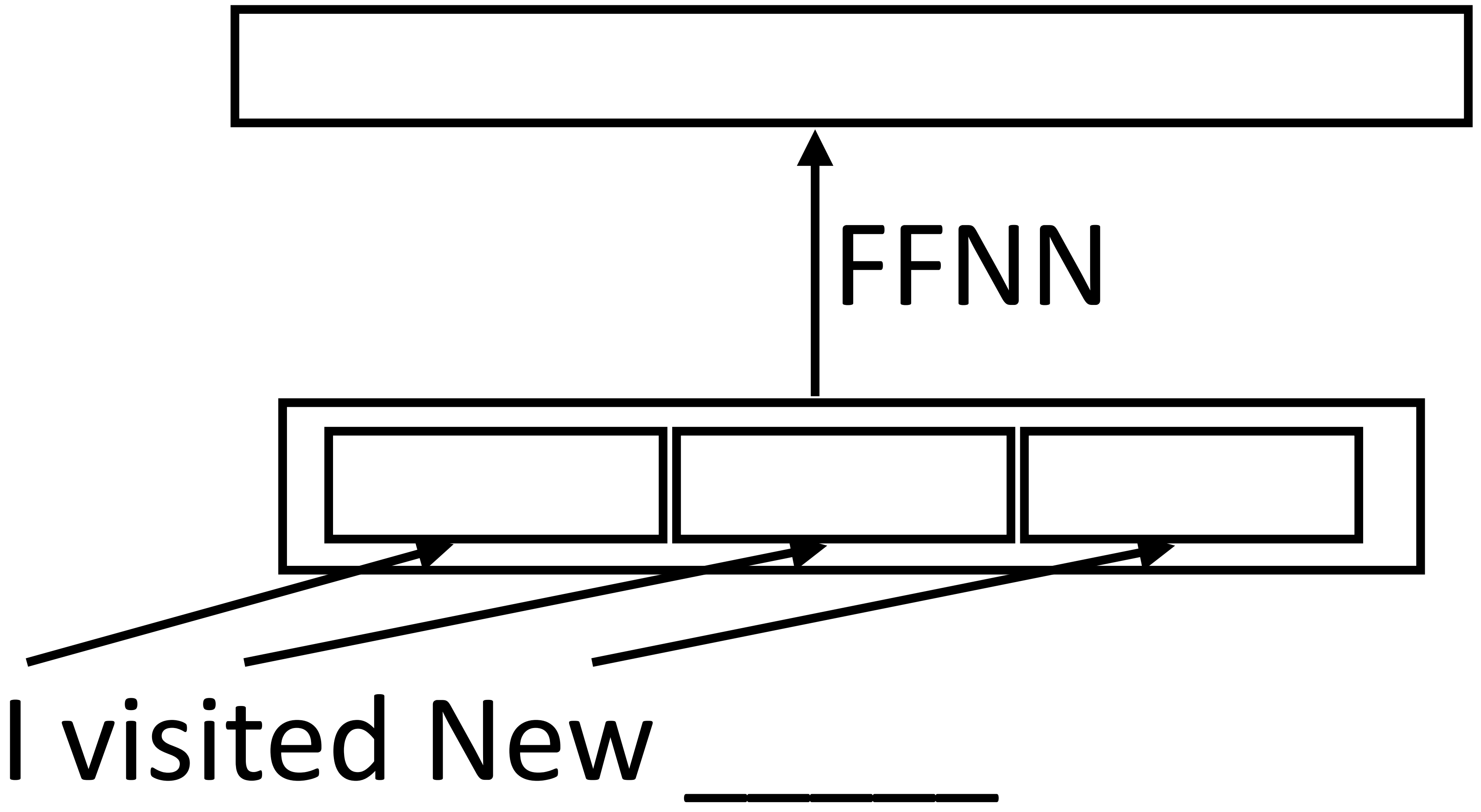
4) Calculate attention using the resulting Q/K/V matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix $W^O$ to produce the output of the layer

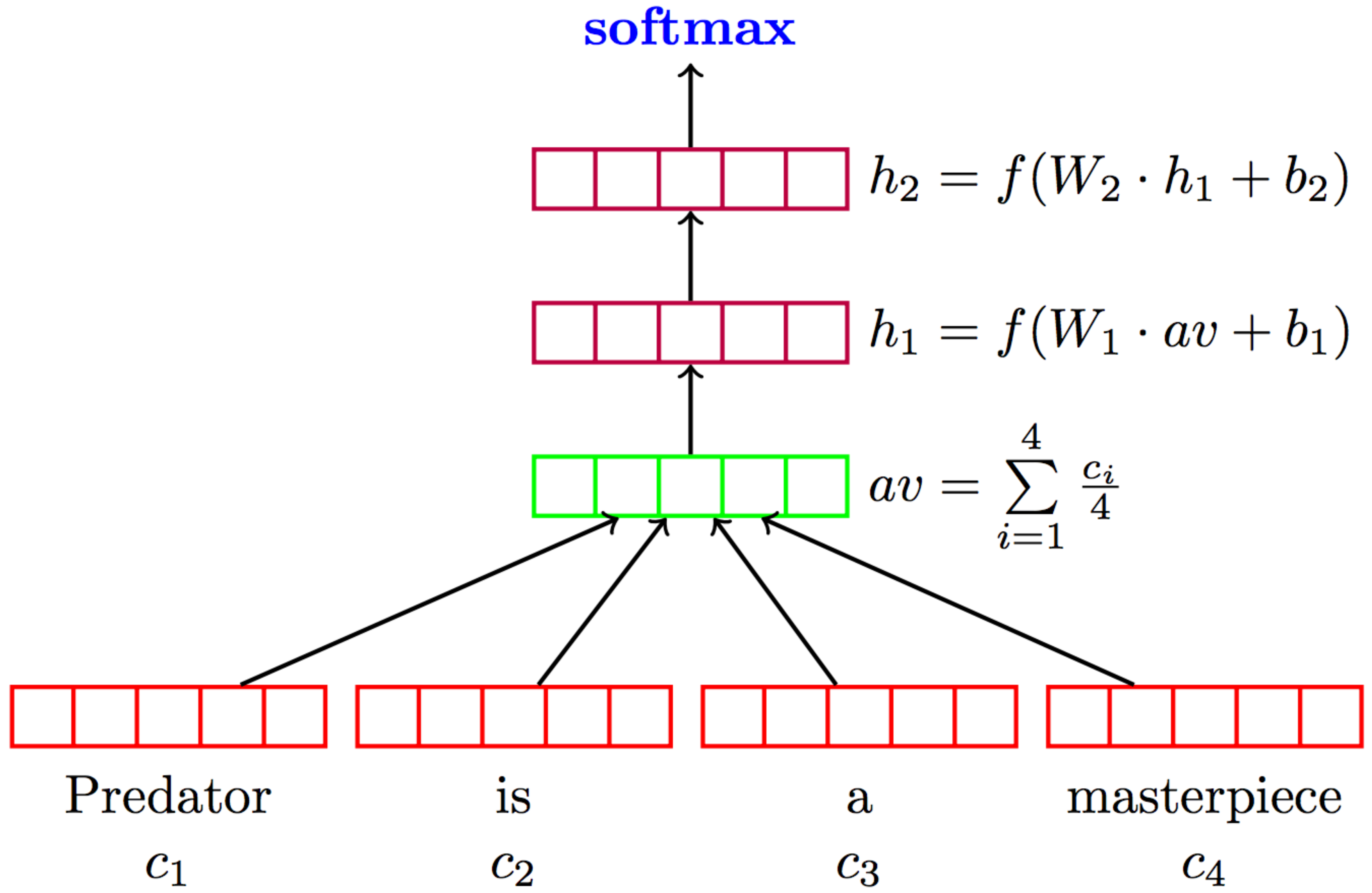



* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

# Challenges of Neural Language Modeling

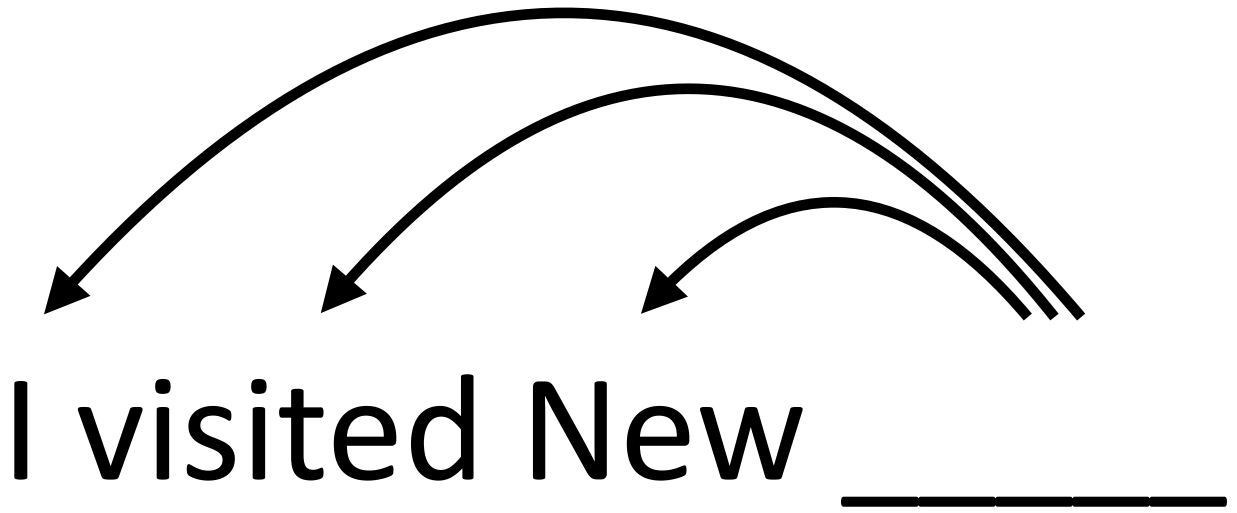



FFNN




DAN

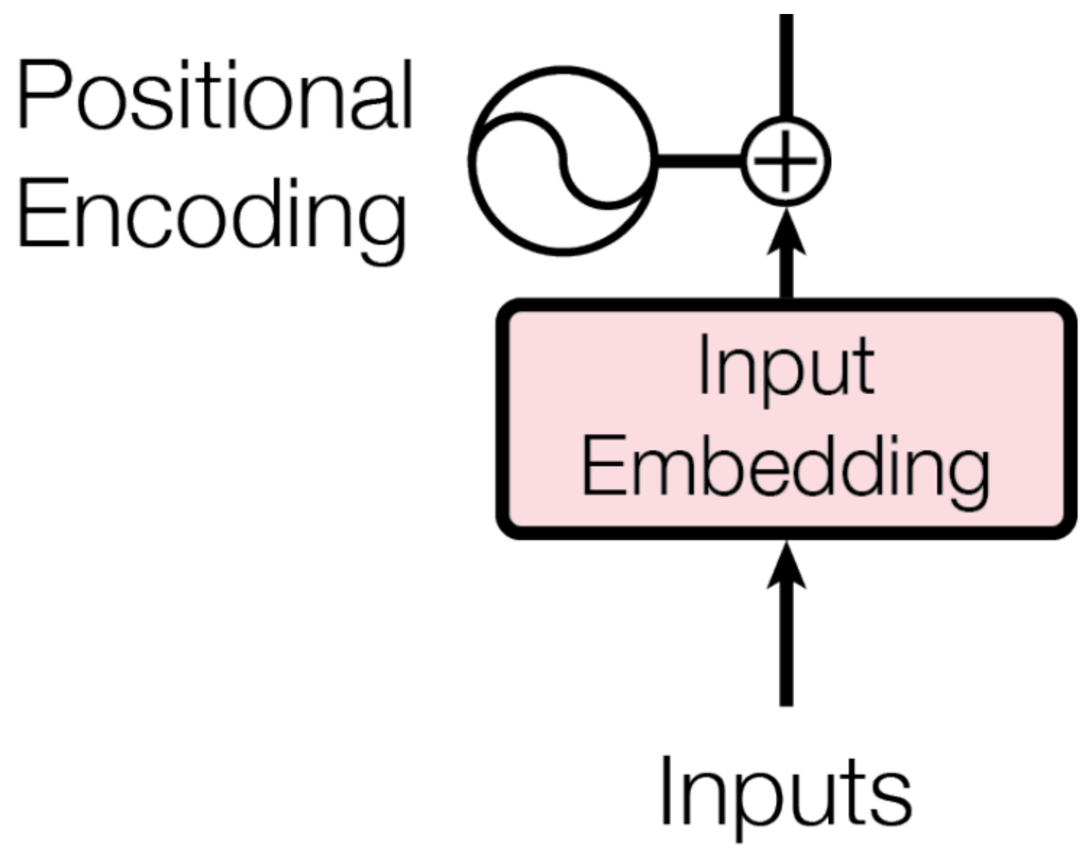
Self-attention:

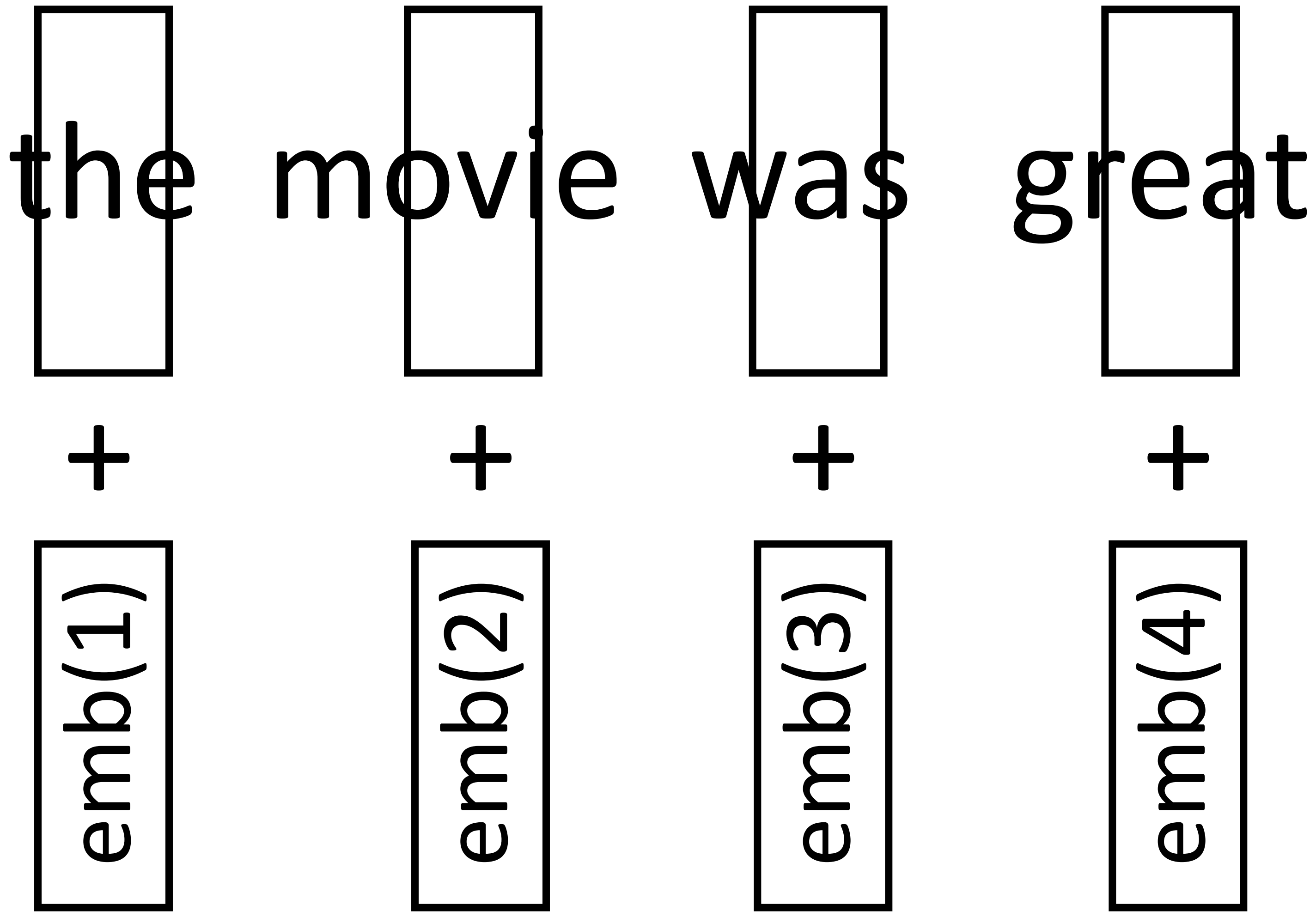



Still missing one component: position sensitivity

# Positional Encodings

# Transformers: Position Sensitivity



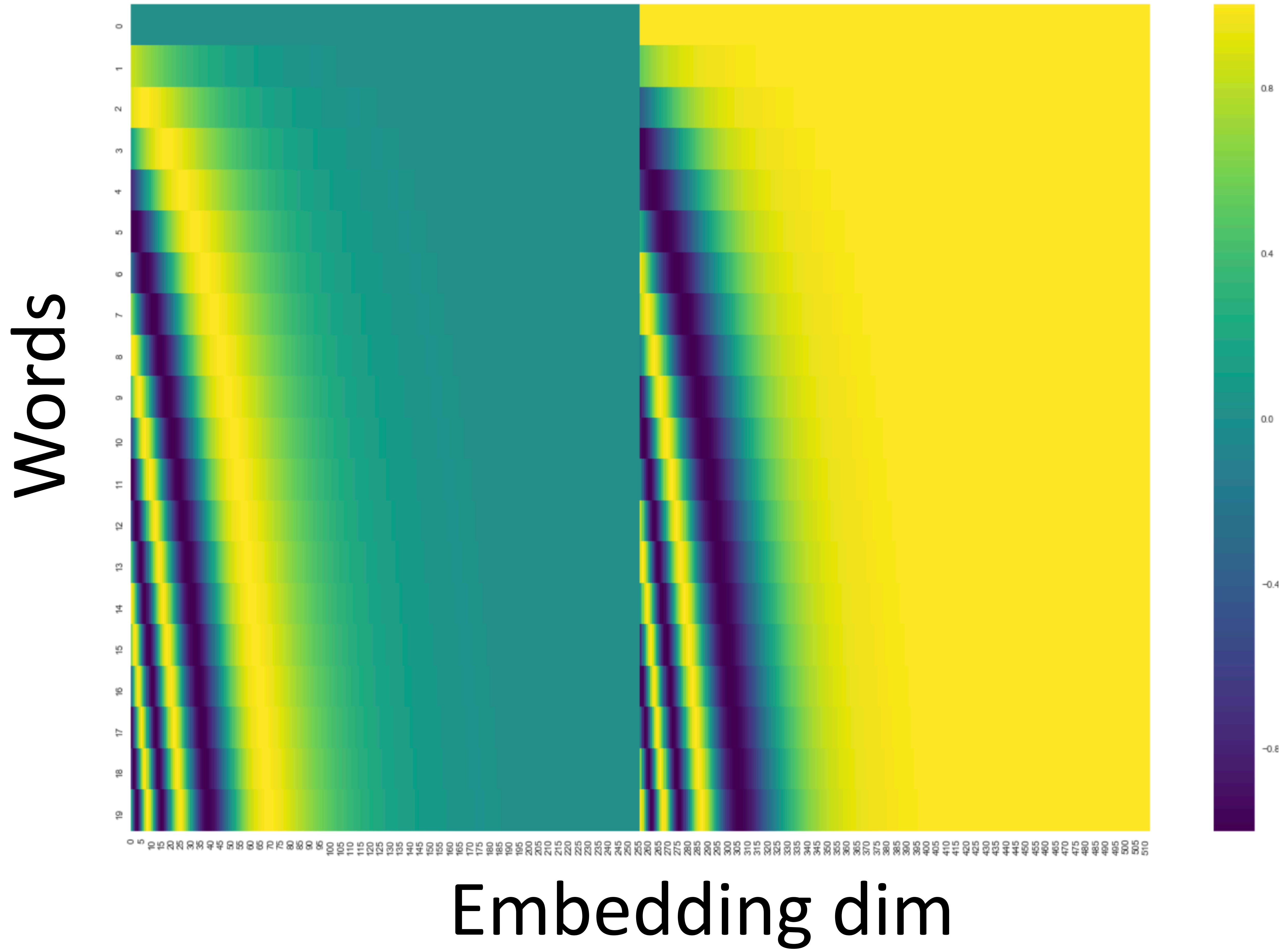




- Encode each sequence position as an integer, add it to the word embedding vector
- Why does this work?

# Transformers

Alammar, *The Illustrated Transformer*

- Alternative from Vaswani et al.: sines/cosines of different frequencies (closer words get higher dot products by default)



# Takeaways

- Language modeling is a fundamental task
- n-gram models are a basic, scalable solution but have limited context
- Self-attention is a solution to the question of: how do we look at a lot of context, efficiently, without blowing up parameter counts, and without forgetting far-back things?
- Next time: see the whole Transformer architecture and extensions of it