Quiz 1

15 minutes

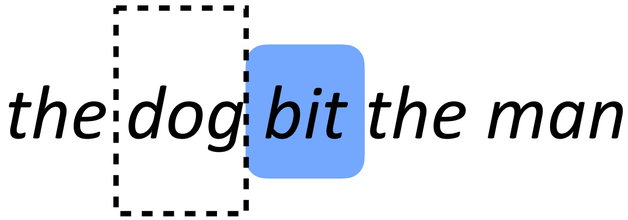
CS388: Natural Language Processing
Lecture 6: Self Attention

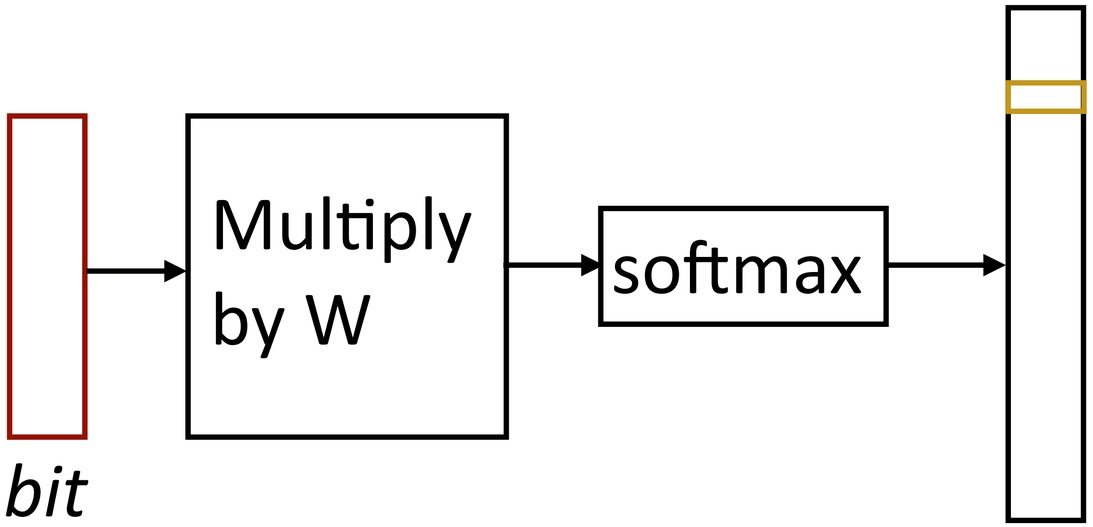
Elias Stengel-Eskin

Slide Credits: Greg Durrett

Recap: Skip-Gram

- Predict one word of context from word





gold = *dog*

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

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

	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 12 pairs: 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

- Many models we will see 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 review
- Neural language modeling review
- Self-attention
- Multi-head self-attention
- Positional encodings (if time)

Language Modeling

Recap: 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 a pre-training task like skip-gram!

Recap: N-gram LMs

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

- 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)

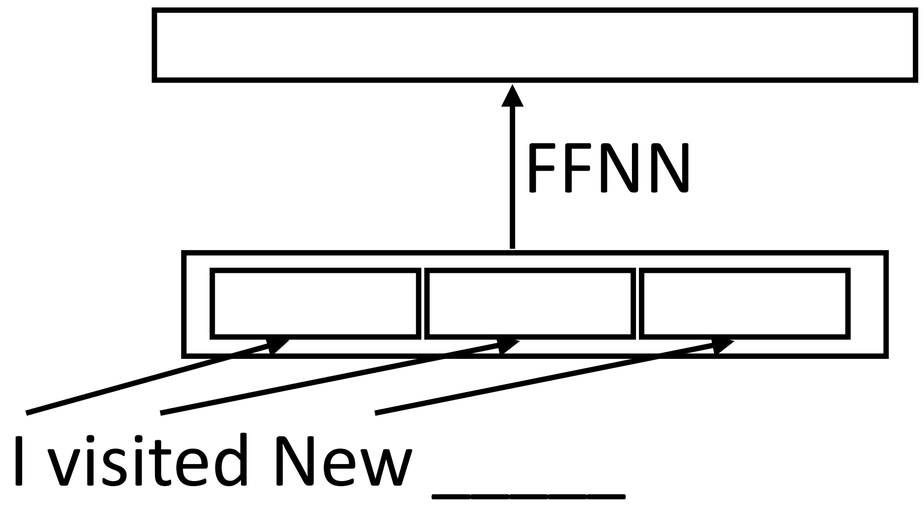
3-gram: P(w | visited San)

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

Neural Language Modeling

Neural Language Models

- Early work: feedforward neural networks looking at context



$$P(w_i|w_{i-n},\ldots,w_{i-1})$$

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

Bengio et al. (2000, 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?
- What was a challenge with the neural LM approach we've seen so far?

Contextualized Embeddings

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

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

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

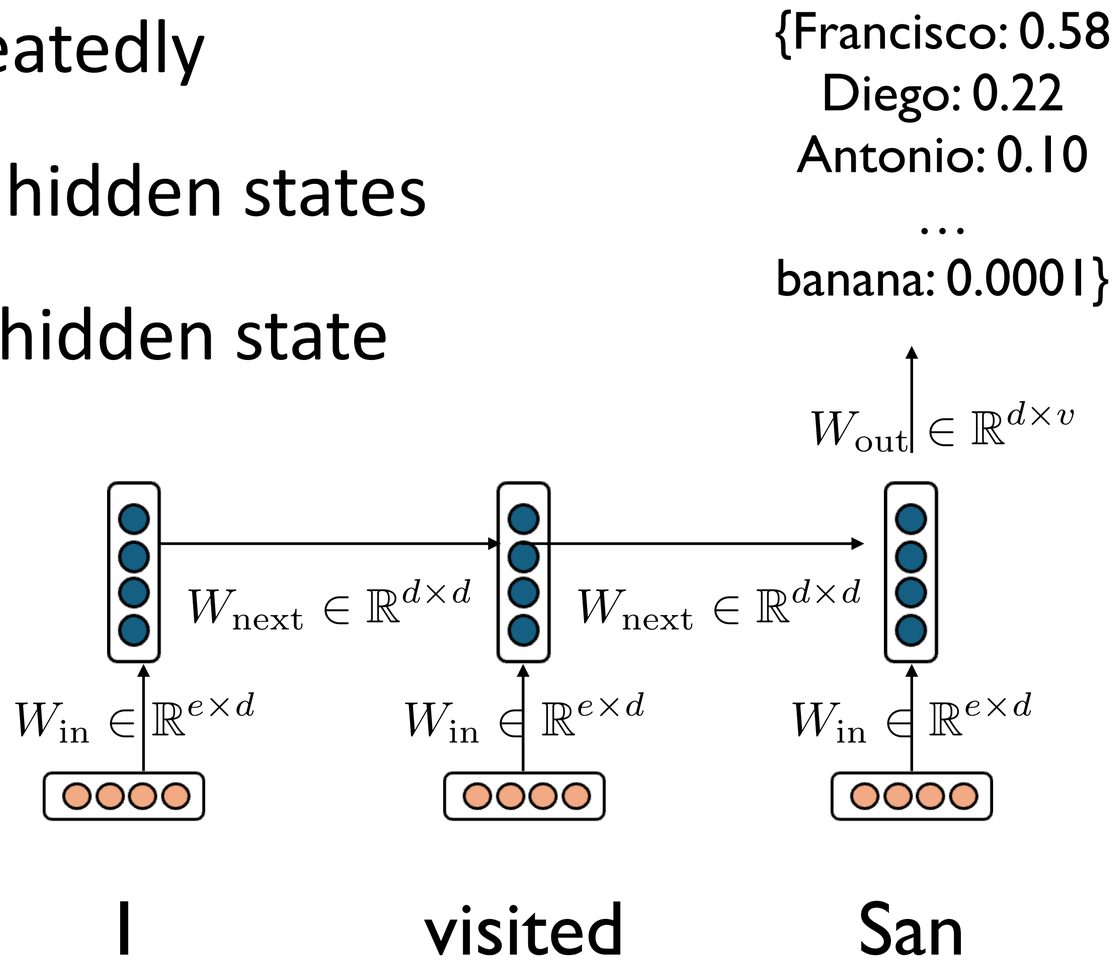
- unidirectional representation (only looks at past words)

$\boldsymbol{x}$ = *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: $\boldsymbol{e}(\boldsymbol{x}) = f(\boldsymbol{x}_{-1}, \textit{the})$

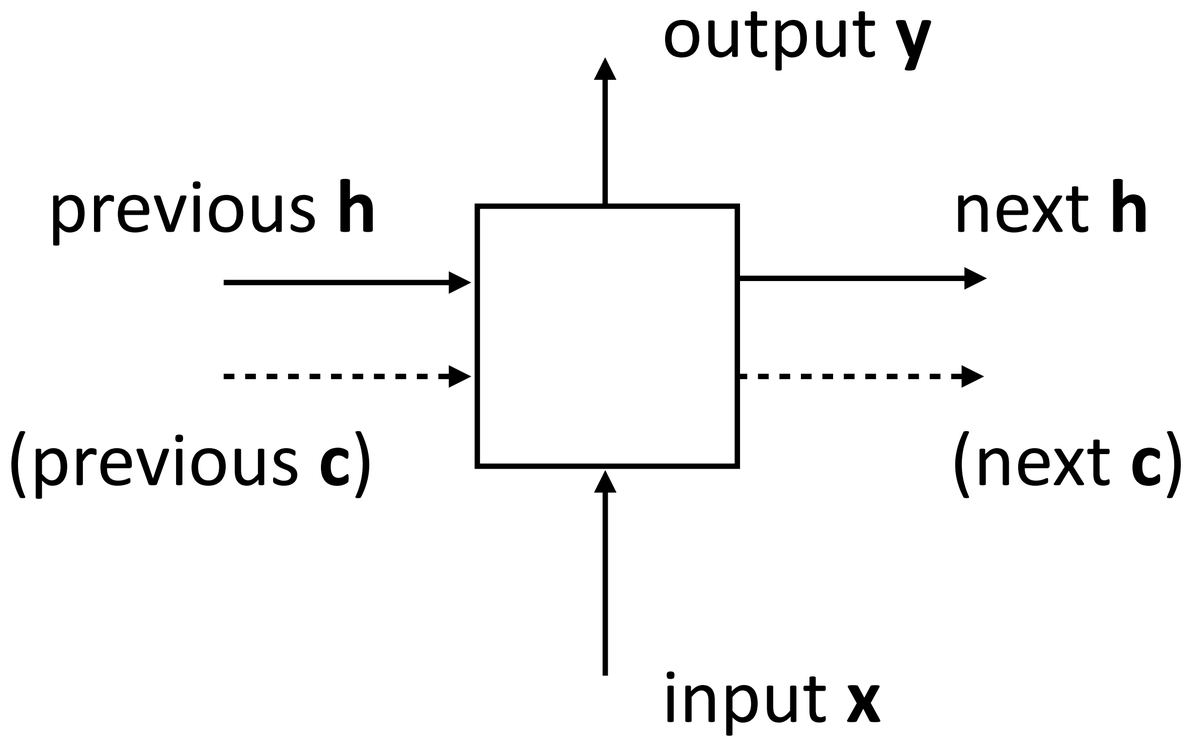
RNNs (briefly)

- Apply same weights repeatedly
- Weight to map inputs to hidden states
- Weight to bring forward hidden state



Elman, 1990
Bengio et al.,1994
Hochreiter and Schmidhuber, 1997

RNNs: Why not?

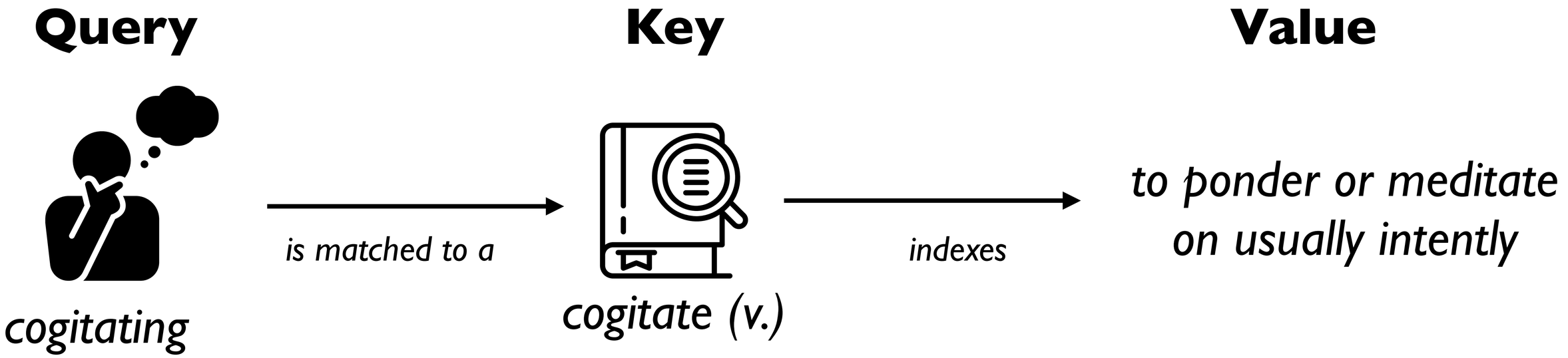


- Slow. They do not parallelize and there are O(n) non-parallel operations to encode n items
- Even modifications like LSTMs still don’t enable learning over very long sequences. Transformers can scale to millions of tokens!

(Self-)Attention

Attention Intro

- Think of attention as a dictionary lookup, in vector space



Running Example

- Fixed-length sequence of As and Bs

AAAAAAA

ABAAAAB

ABAABAB

AAAABAB

BAAAAAAAAAAAAAAAAAAAAAAAAAB

- All As = last letter is A; any B = last letter is B
- **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 A = [1, 0]; B = [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]

A A B A

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

$s_i = k_i^T q$

0 0 1 0

Attention

Step 1: Compute scores for each key

keys k_i

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

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

A A B A

$s_i = k_i^T q$

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]

A A B A

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

(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?

What if we have a very very long sequence?

New Keys

keys k_i

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

A A B A

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

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

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

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

0 0 1 0

What will new attention values be with these keys?

0, 0, 10, 0 —> softmax —> [4.5e-5, 4.5e-5, 0.99986, 4.5e-5]

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 Bs

$W^K = \begin{matrix} 10 & 0 \\ 0 & 10 \end{matrix}$ "booster" as before

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

Self-Attention

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

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

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

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

Let's compute these now!

Self-Attention

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

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

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

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.)

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

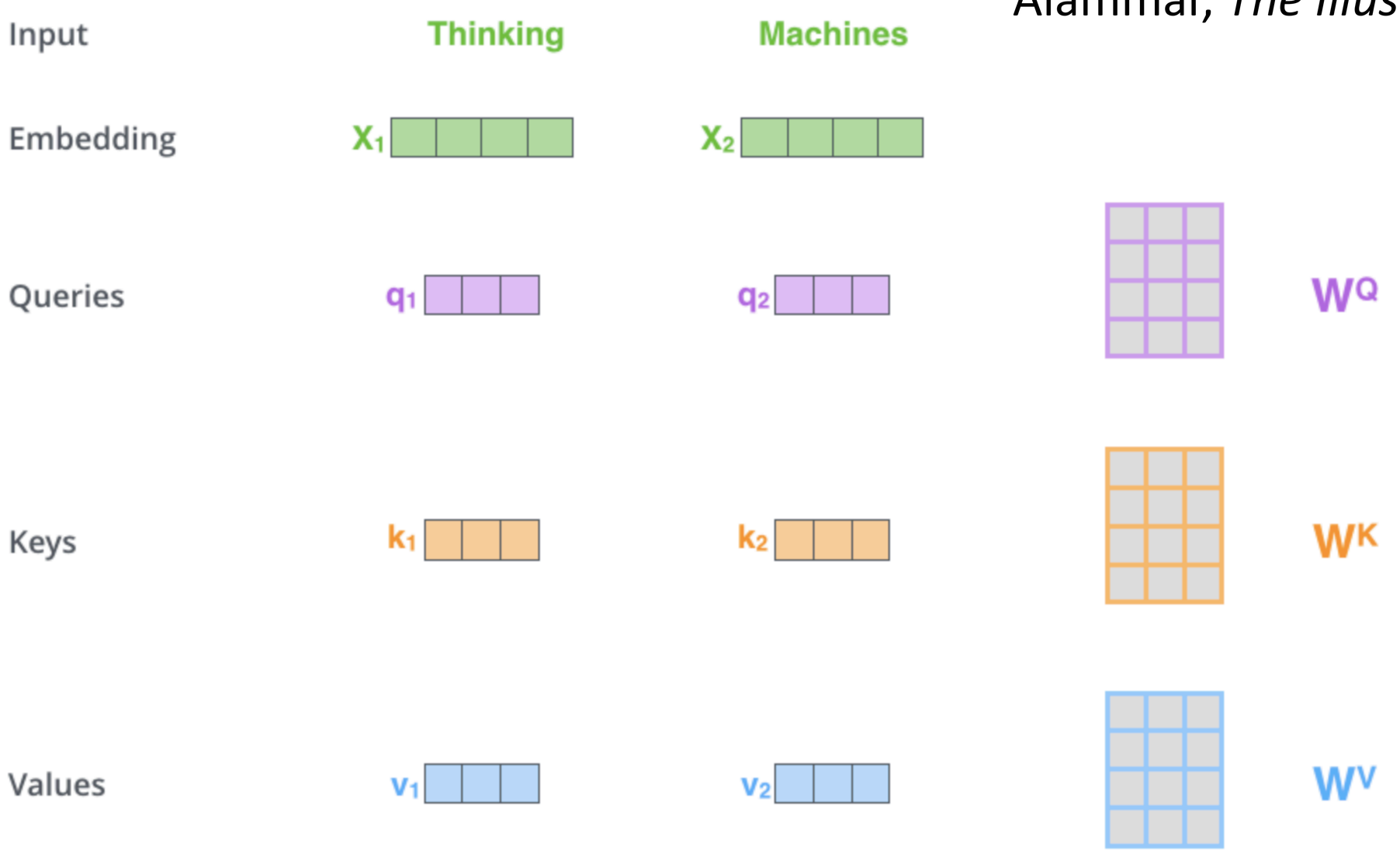
$Q = EW^Q$, $K = EW^K$, $V = EW^V$

- 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)

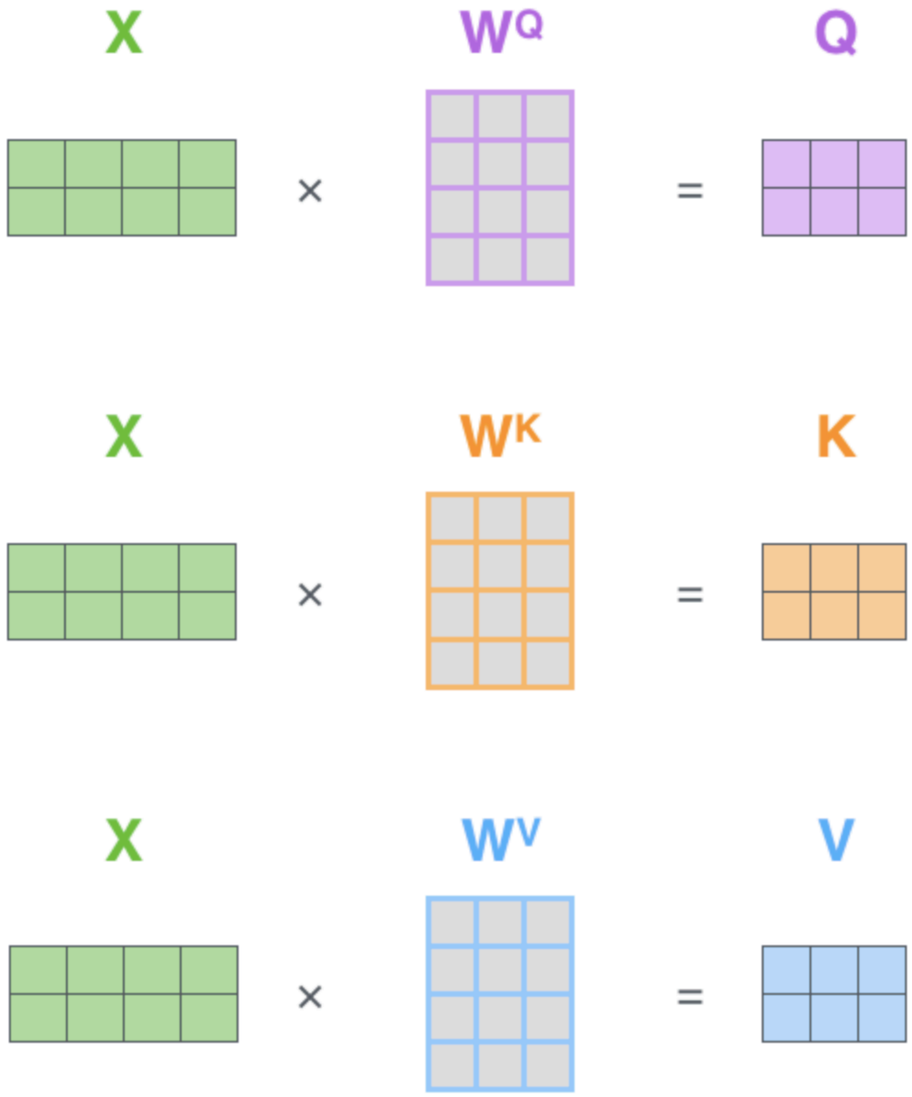
Vaswani et al. (2017)

Self-Attention

Alammar, *The Illustrated Transformer*



Self-Attention



Alammar, *The Illustrated Transformer*

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

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V = Z$$

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

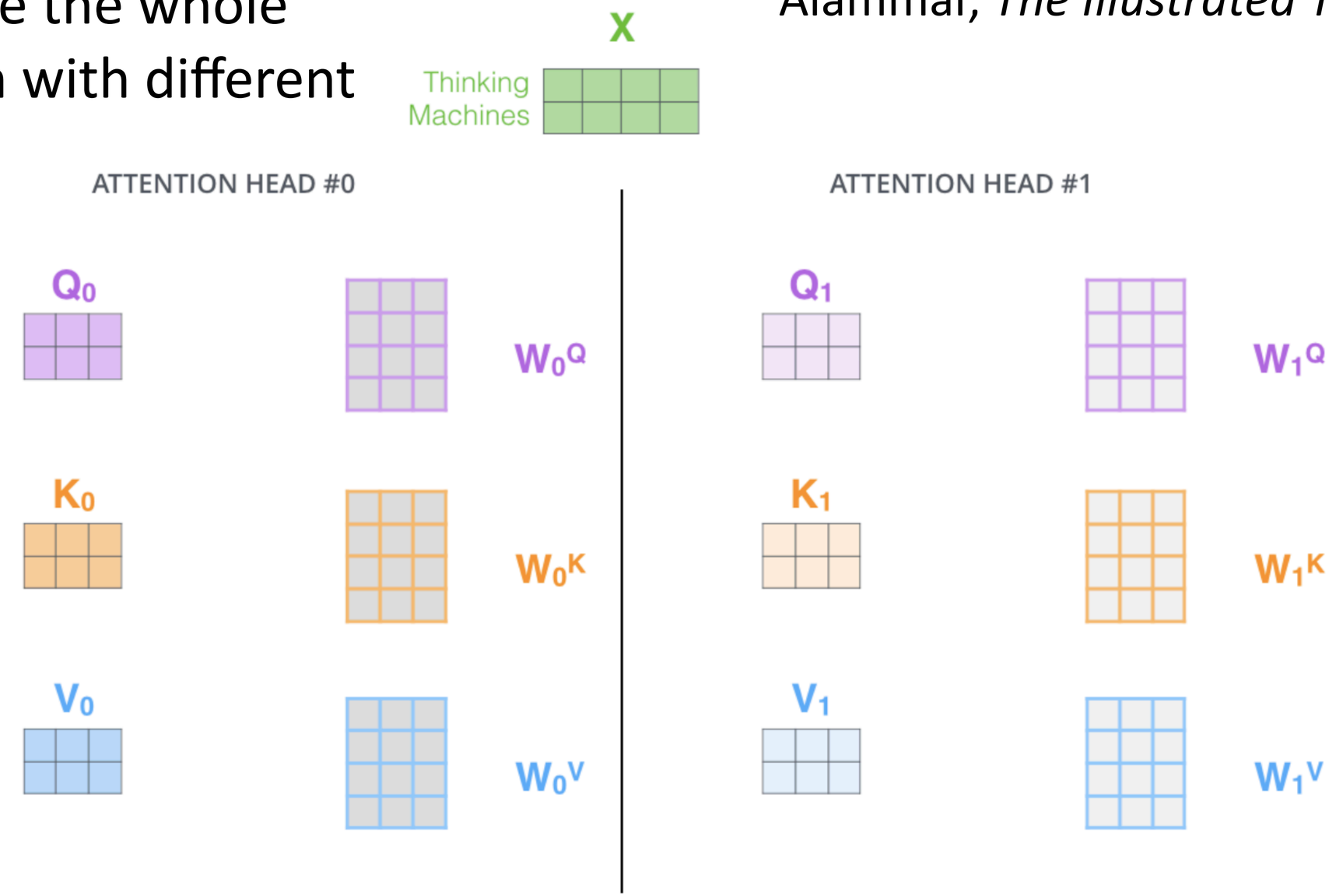
Vaswani et al. (2017)

Multi-Head Self-Attention

Multi-head Self-Attention

Just duplicate the whole computation with different weights:

Alammar, *The Illustrated Transformer*



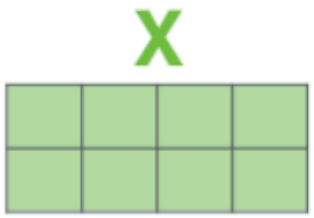
Multi-head Self-Attention

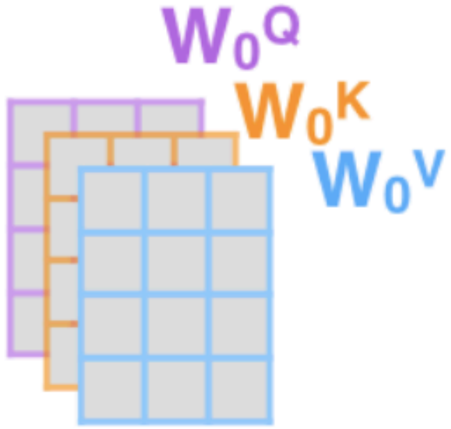
1) This is our input sentence*

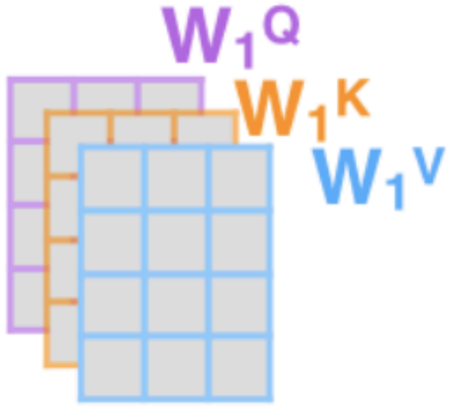
2) We embed each word*

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*

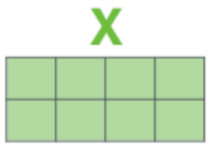
2) We embed each word*

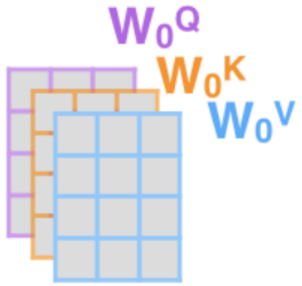
3) Split into 8 heads. We multiply X or R with weight matrices

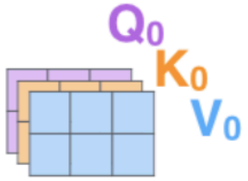
4) Calculate attention using the resulting Q/K/V matrices

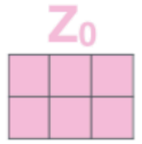
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

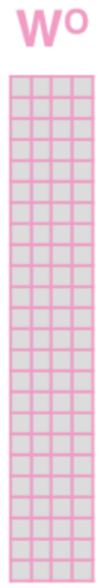
Thinking Machines

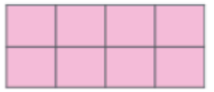




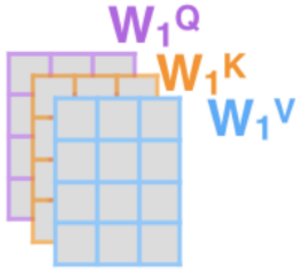


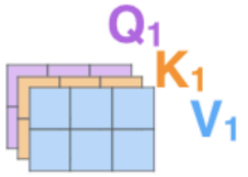


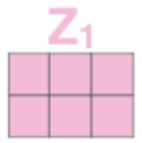




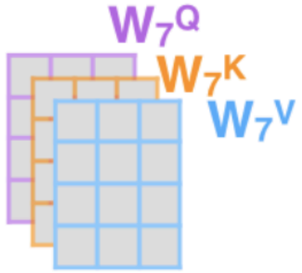
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



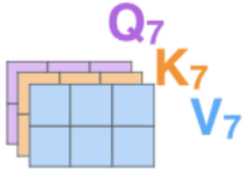




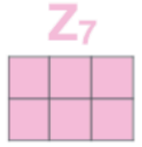
...



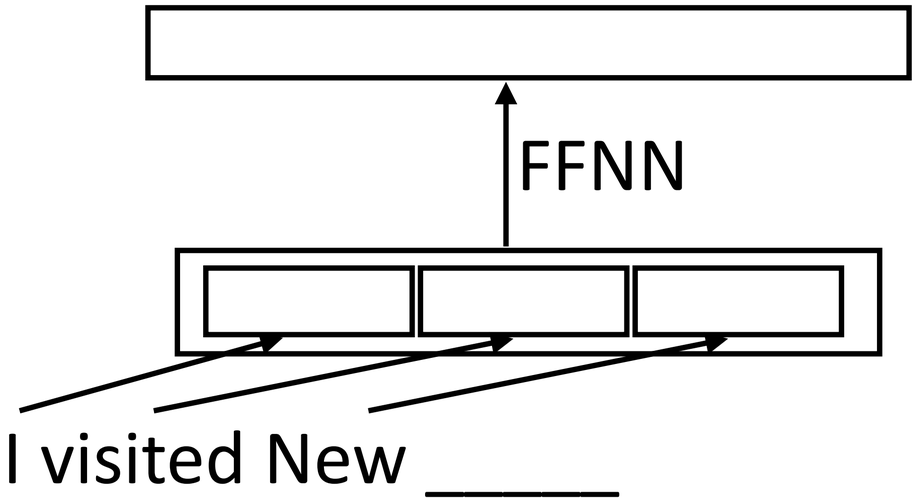
...



...

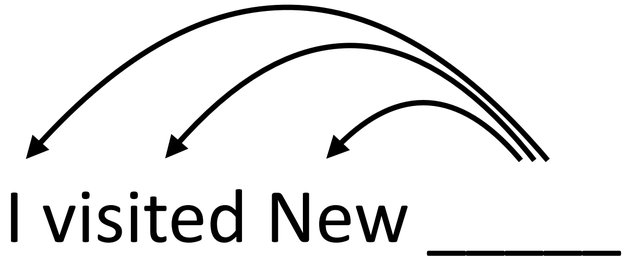


Challenges of Neural Language Modeling



FFNN

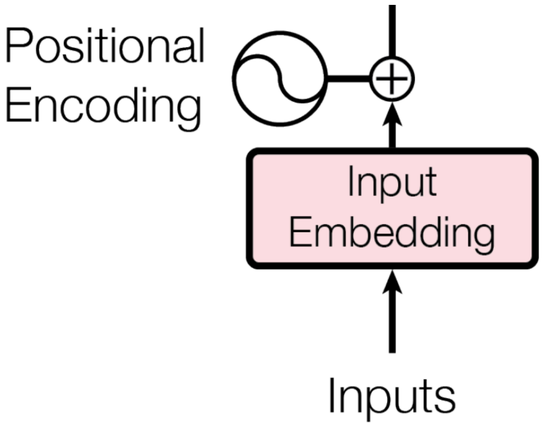
Self-attention:

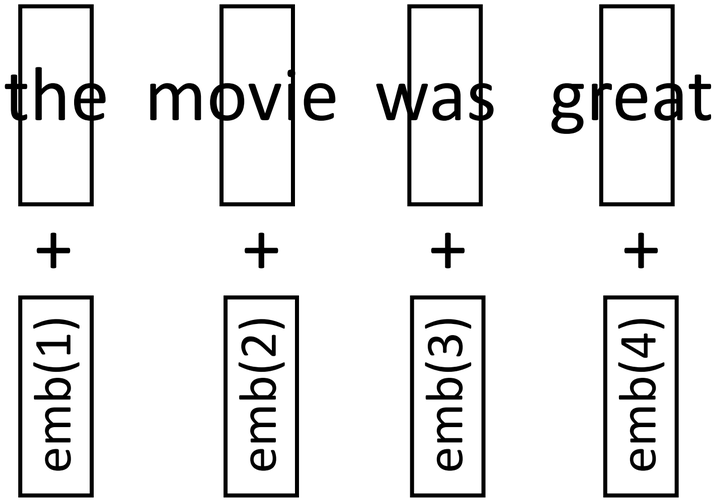


Still missing one component: position sensitivity

Positional Encodings

Transformers: Position Sensitivity



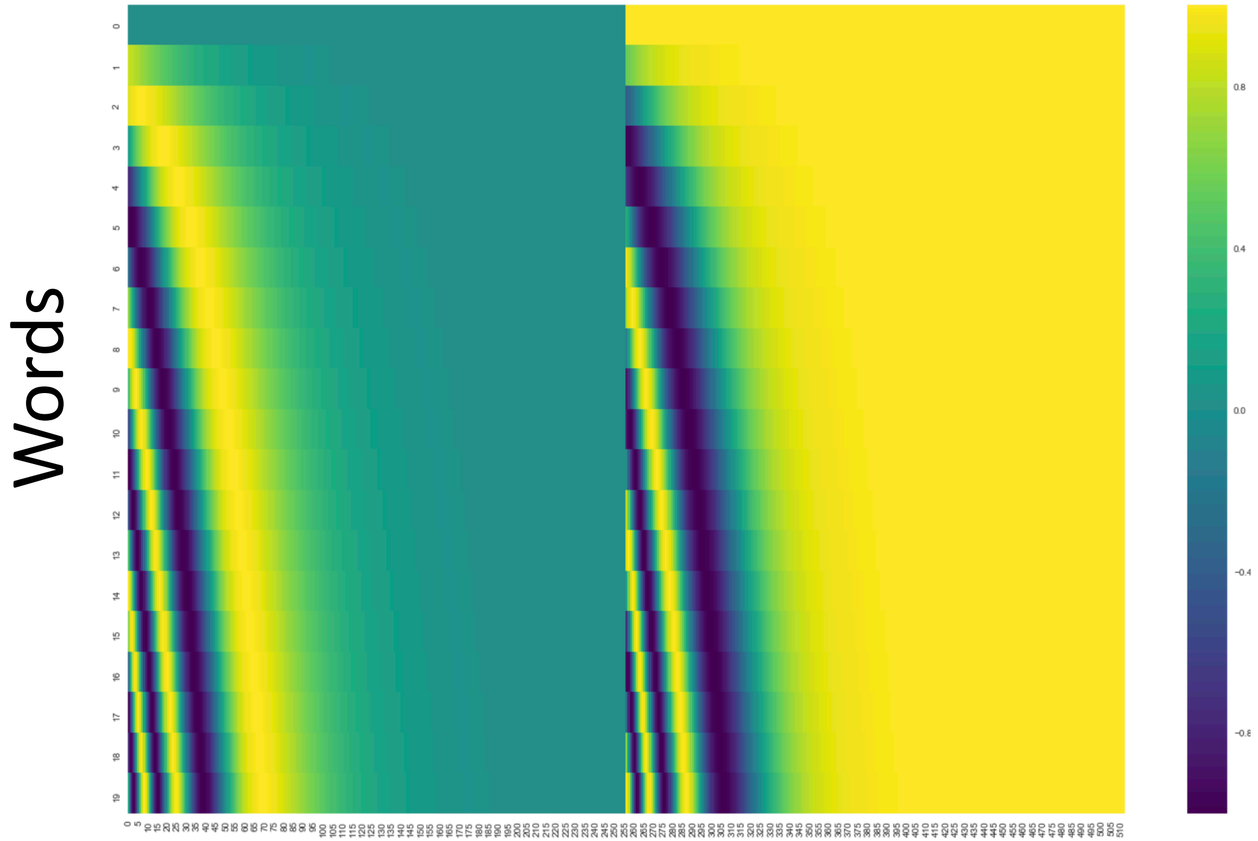


- Encode each sequence position as an integer, add it to the word embedding vector
- Why does this work?

Transformers

Alammar, *The Illustrated Transformer*

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



RoPE and variants

- Rotary Position Embeddings (RoPE)

RoFormer: Enhanced Transformer with Rotary Position Embedding
Su et al. (2024)

- Split query/key into 2D pairs and rotate by angle proportional to position $\theta_i * m$

$$\theta_i = 10000^{-2(i-1)/d}$$

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$\begin{aligned} q_m^T R(n \cdot \theta)^T R(m \cdot \theta) k_n \\ = q_m^T R((n-m)\theta) k_n \end{aligned}$$

function of n-m, relative

$$R^d_{\theta,n-m} = \begin{bmatrix} \cos(n-m)\theta_1 & -\sin(n-m)\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin(n-m)\theta_1 & \cos(n-m)\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos(n-m)\theta_2 & -\sin(n-m)\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin(n-m)\theta_2 & \cos(n-m)\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos(n-m)\theta_{d/2} & -\sin(n- \\ 0 & 0 & 0 & 0 & \cdots & \sin(n-m)\theta_{d/2} & \cos(n- \end{bmatrix}$$

apply to each 2D pair separately

- Intuition: Rotation preserves magnitude while allowing us to encode relative position.
- Rotation angle increases as distance increases. Later dimensions rotate more slowly.
  - For near tokens, late dimensions barely rotate. For far tokens, rotate.

NoPE?

- Do we even need positional embeddings?
- Positional embeddings help encode linguistic prior (order matters, relative position matters)
- But they can also hurt length generalization (generalizing beyond lengths seen during training)
- It turns out that even without explicit positional embeddings, Transformers can learn positional information
  - Why?

The Impact of Positional Encoding on Length Generalization in Transformers, Kazemnaejad et al. (2023)
https://sebastianraschka.com/llm-architecture-gallery/nope/

Takeaways

- 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