Transformers

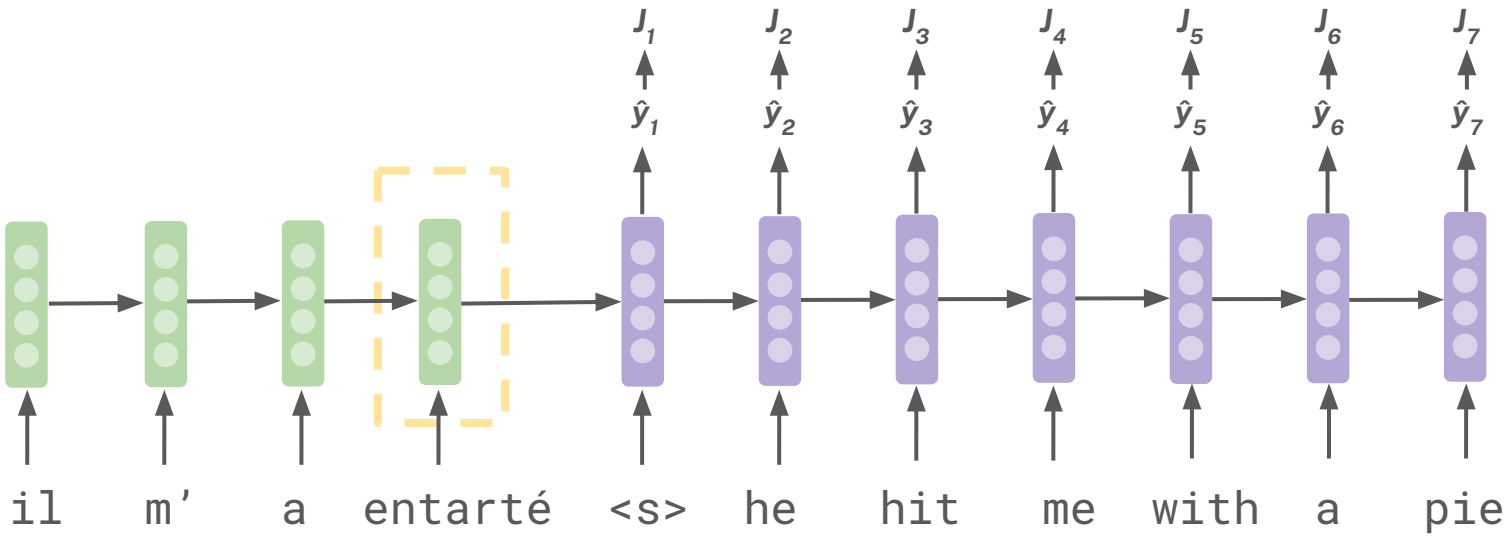
Today’s plan

1. Attention, or a seq2seq sequel to Monday’s lecture
2. ✨ the transformer ✨
3. Transformer variants

The bottleneck

One encoding vector needs to capture all of the info about the source sentence.

+ Longer sentences can lead to vanishing gradients!



Attention to the rescue

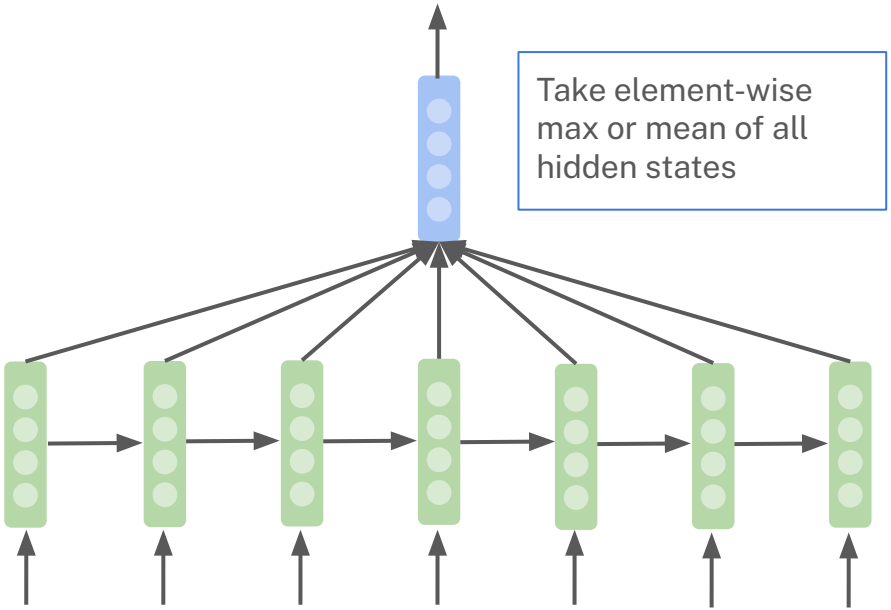
Attention provides a solution to the bottleneck problem.

Key idea: At each time step during decoding, focus on a particular part of the source sentence, using a direct connection to the encoder

We'll first show this via diagrams (no equations) and then we'll show equations.

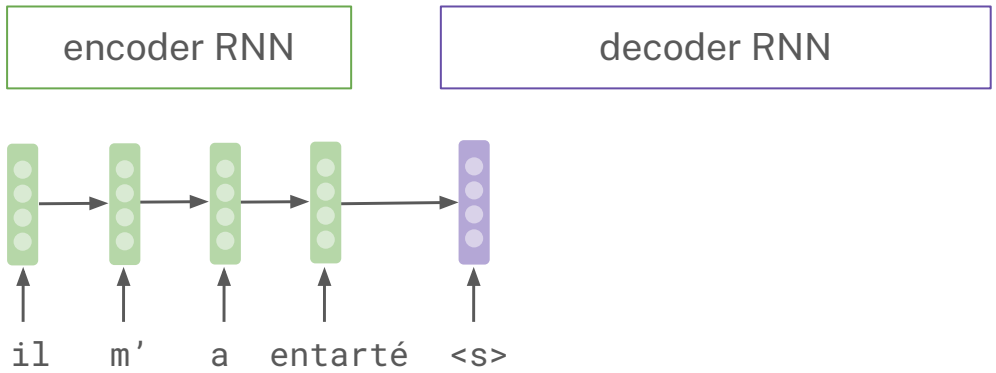
The starting point: mean-pooling for RNNs

One way of “passing information from the encoder” is to compute an **average**.



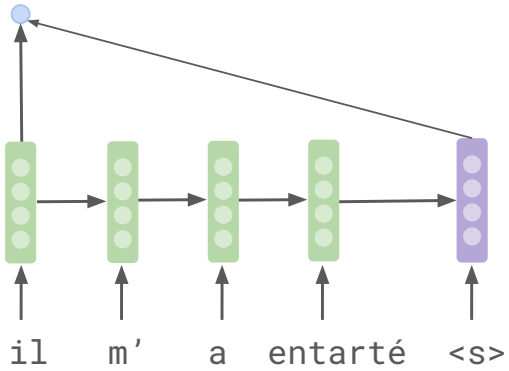
Attention is **weighted** averaging. This is powerful if the weights are learned!

Seq2seq with attention



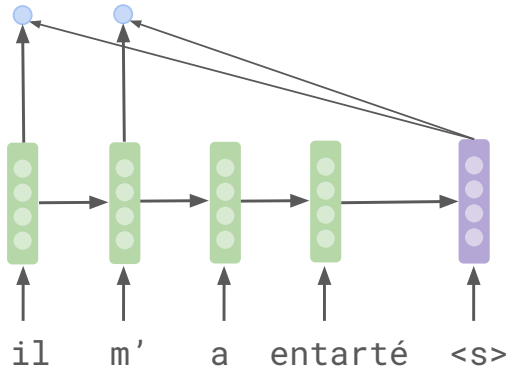
Seq2seq with attention

Attention scores, via dot product



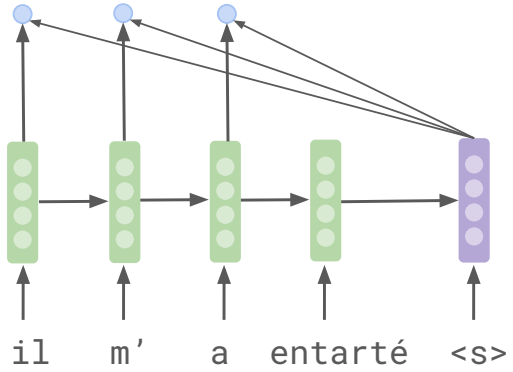
Seq2seq with attention

Attention scores, via dot product



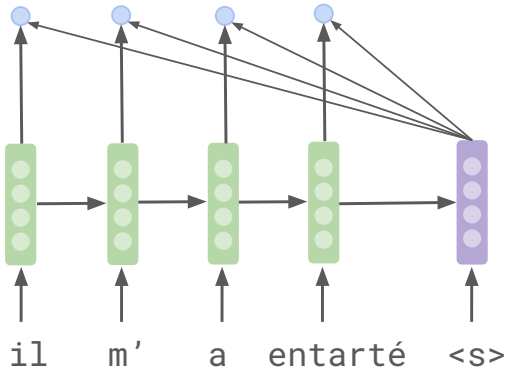
Seq2seq with attention

Attention scores, via dot product

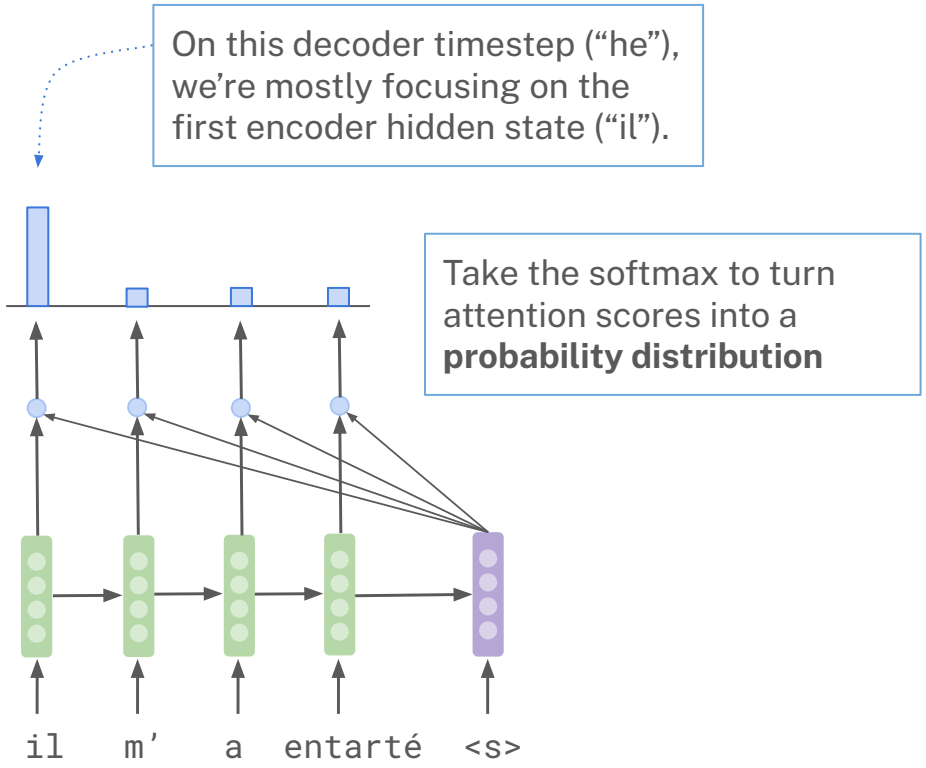


Seq2seq with attention

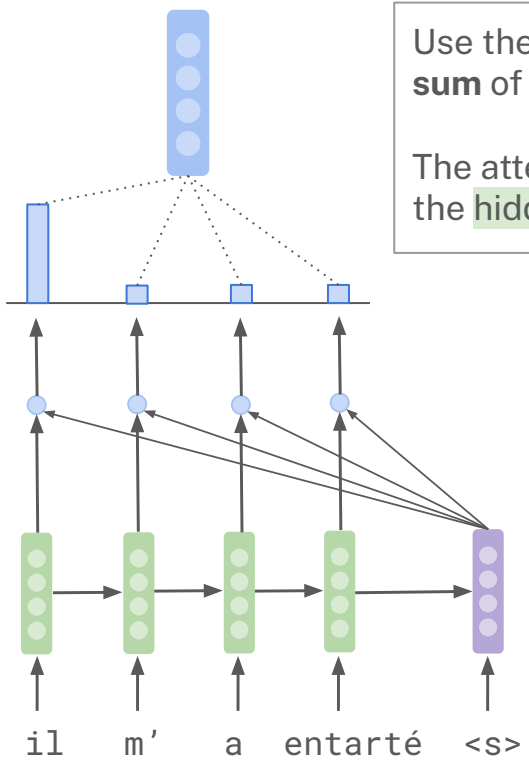
Attention scores, via dot product



Seq2seq with attention



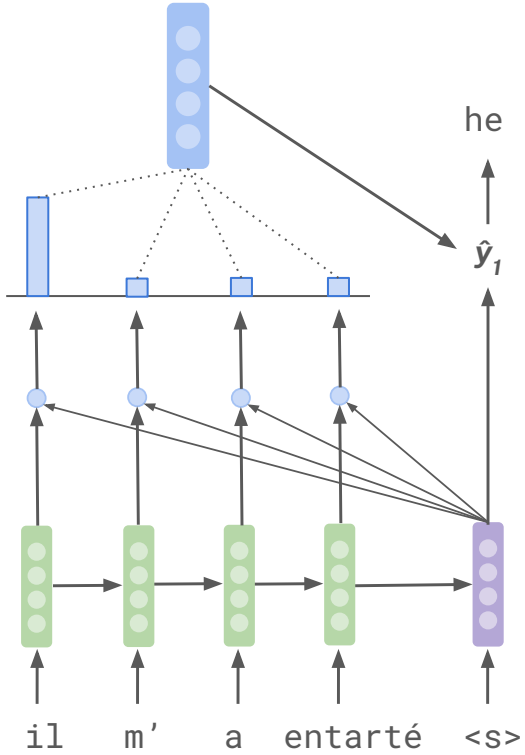
Seq2seq with attention



Use the attention distribution to take **a weighted sum** of the encoder hidden states.

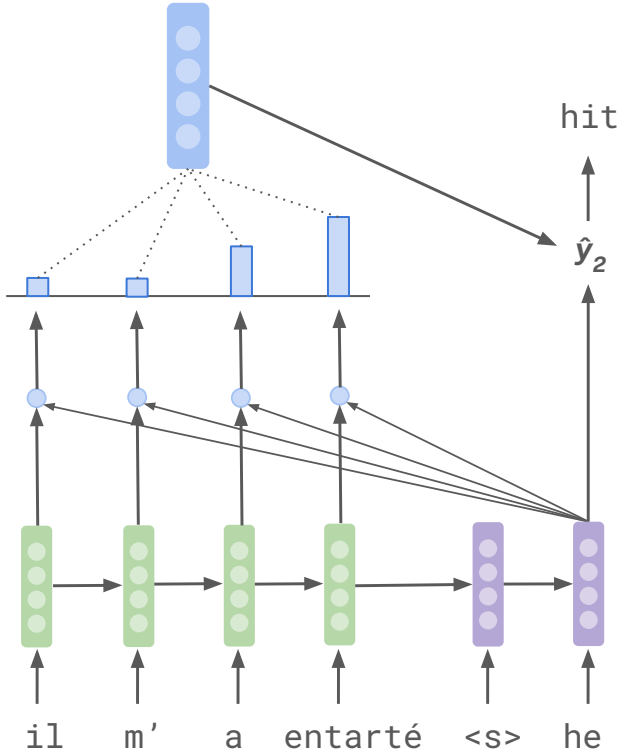
The attention output mostly contains info from the hidden states that receive high scores.

Seq2seq with attention

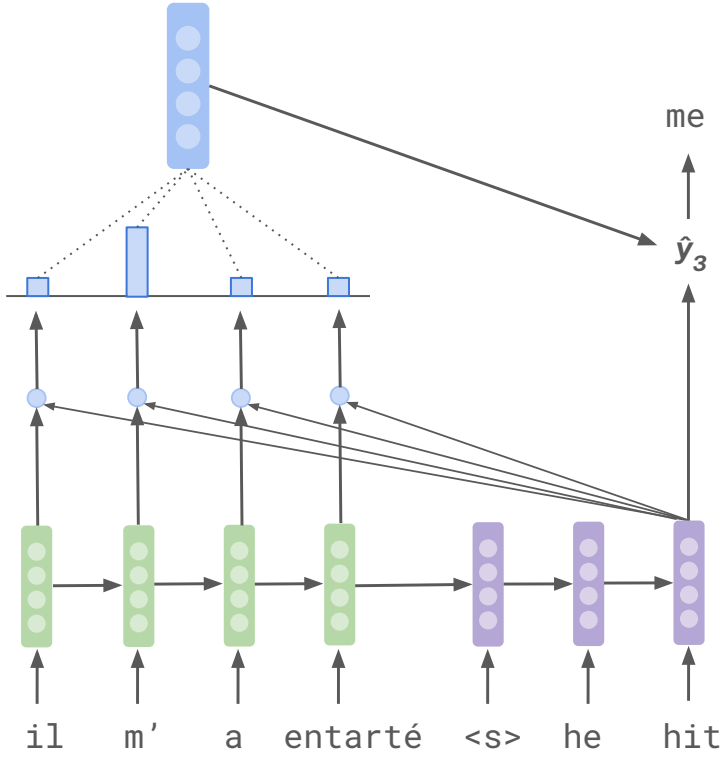


Concat attention output with decoder hidden state, and compute predicted word prob distribution.

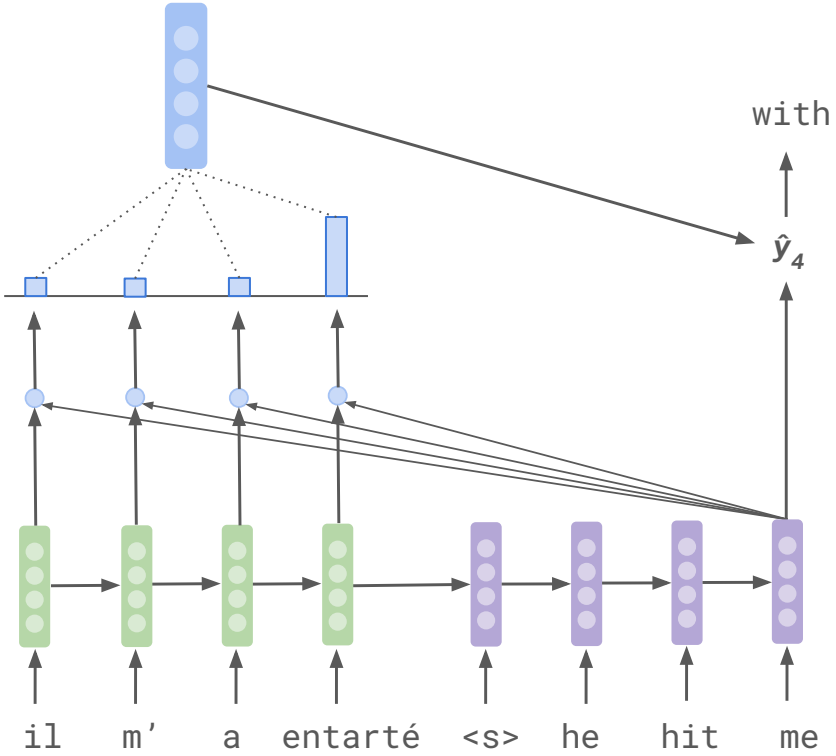
Seq2seq with attention



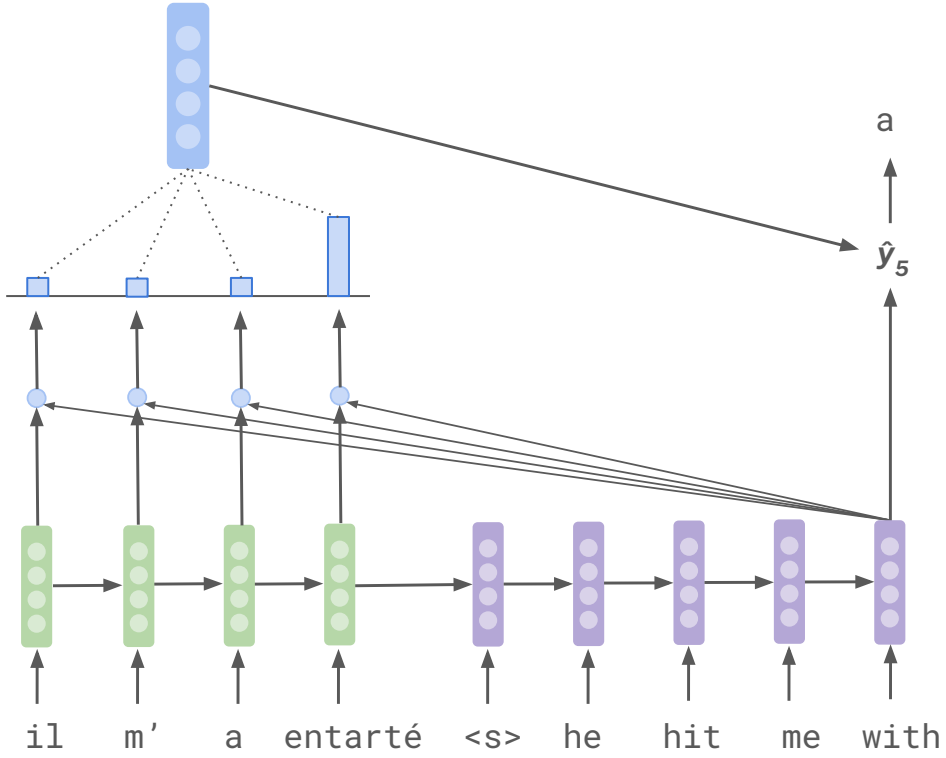
Seq2seq with attention



Seq2seq with attention

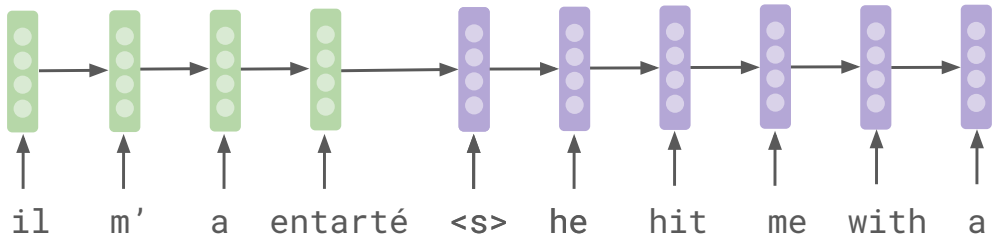


Seq2seq with attention

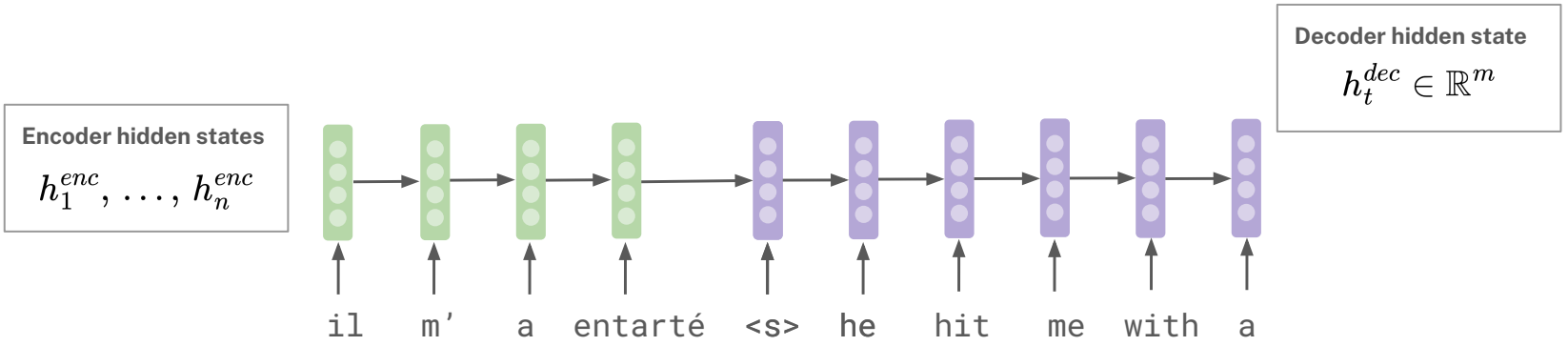


Seq2seq with attention

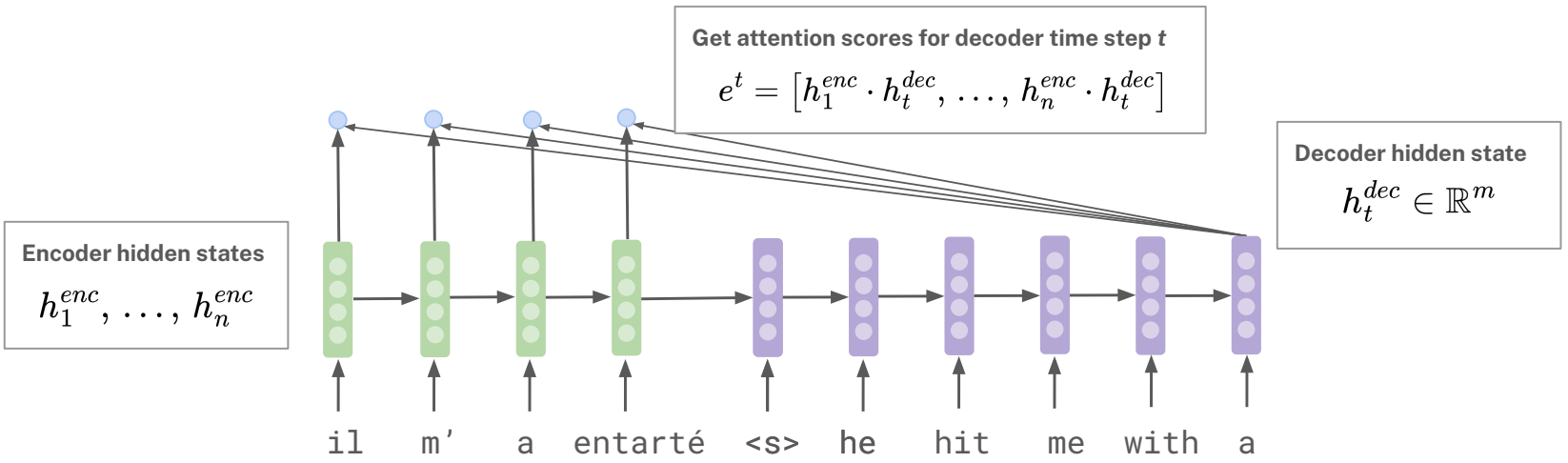
Equation time! 🌟



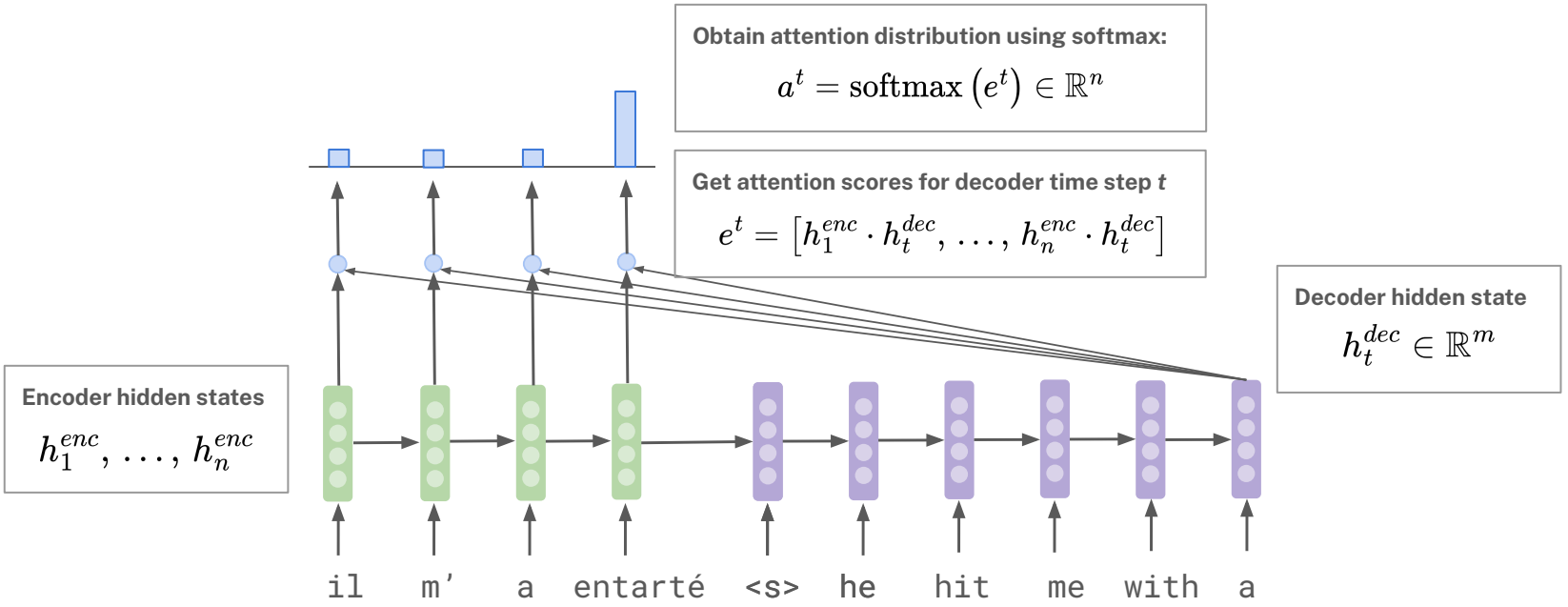
Seq2seq with attention



Seq2seq with attention



Seq2seq with attention



Seq2seq with attention

Compute weighted sum of encoder hidden states:

$$a_t = \sum_{i=1}^{n} a_i^t h_i^{enc} \in \mathbb{R}^m$$

Obtain attention distribution using softmax:

$$a^t = \text{softmax}\left(e^t\right) \in \mathbb{R}^n$$

Get attention scores for decoder time step *t*

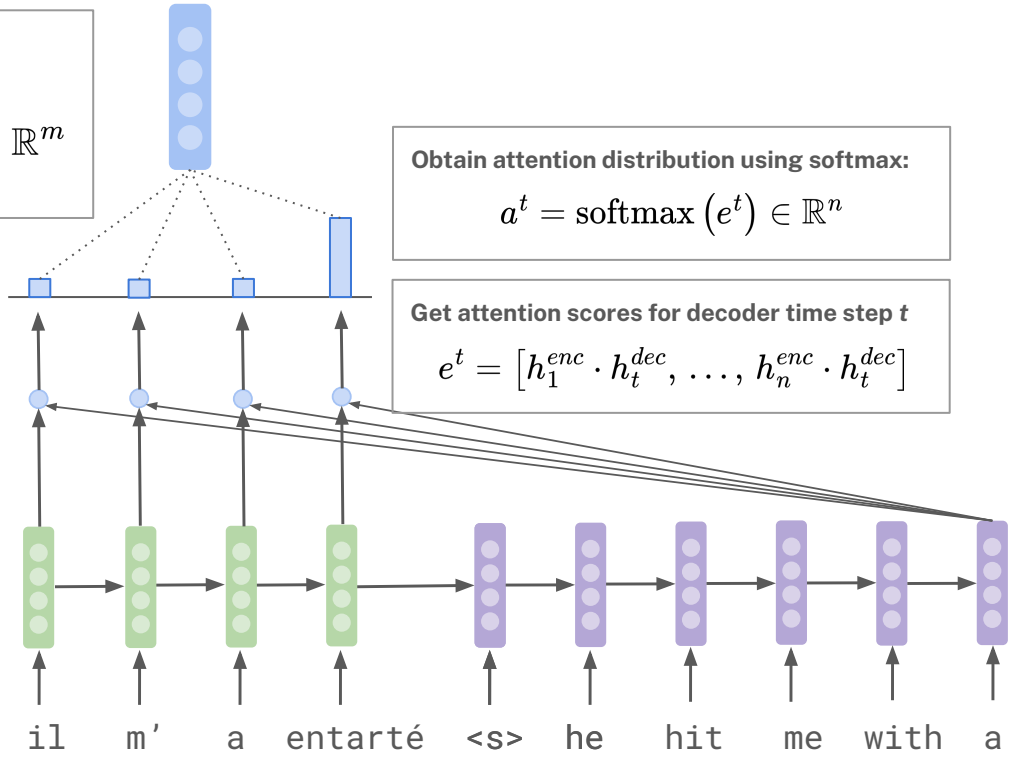
$$e^t = \left[h_1^{enc} \cdot h_t^{dec}, \ldots, h_n^{enc} \cdot h_t^{dec}\right]$$

Decoder hidden state

$$h_t^{dec} \in \mathbb{R}^m$$

Encoder hidden states

$$h_1^{enc}, \ldots, h_n^{enc}$$



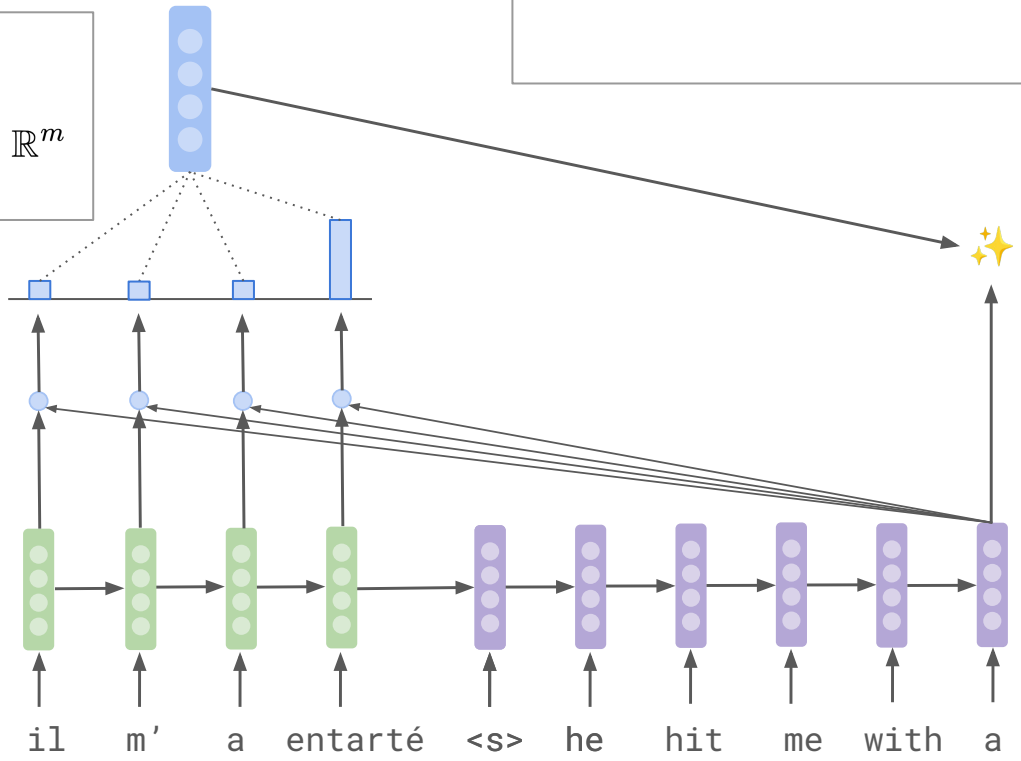
Seq2seq with attention

Compute weighted sum of encoder hidden states:

$$a_t = \sum_{i=1}^{n} a_i^t h_i^{enc} \in \mathbb{R}^m$$

Concat weighted sum w/ decoder state

$$[a_t; h_t^{dec}]$$



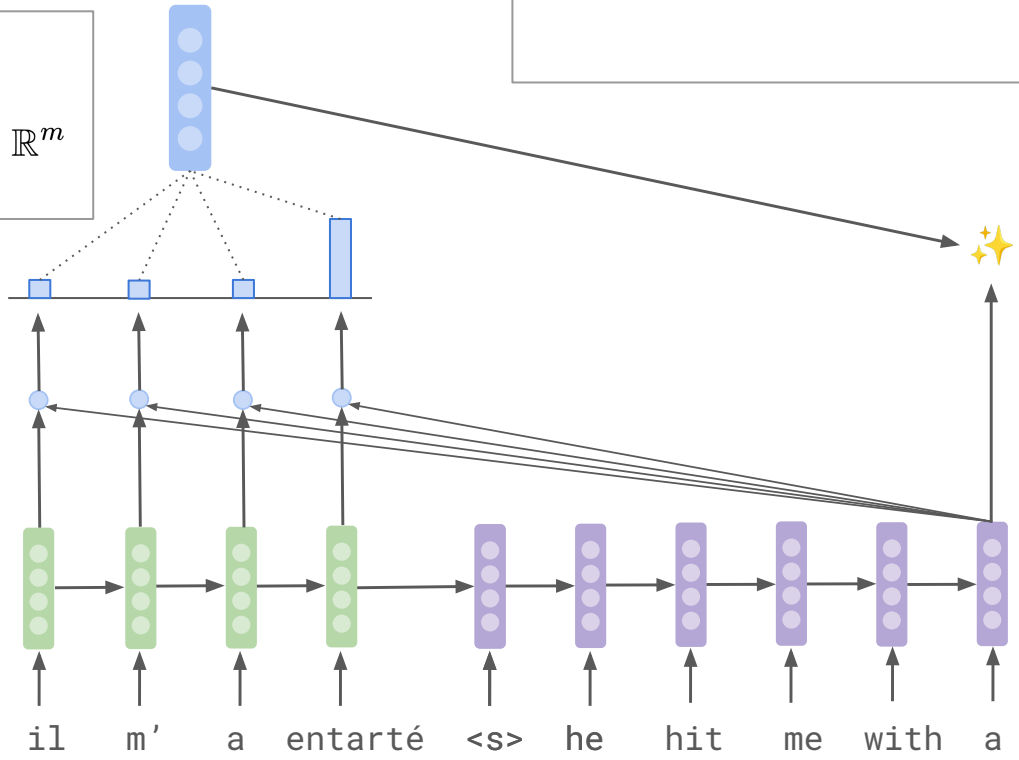
Seq2seq with attention

Compute weighted sum of encoder hidden states:

$$a_t = \sum_{i=1}^{n} a_i^t h_i^{enc} \in \mathbb{R}^m$$

Multiply by weights

$$W_c \left[a_t ; h_t^{dec} \right]$$



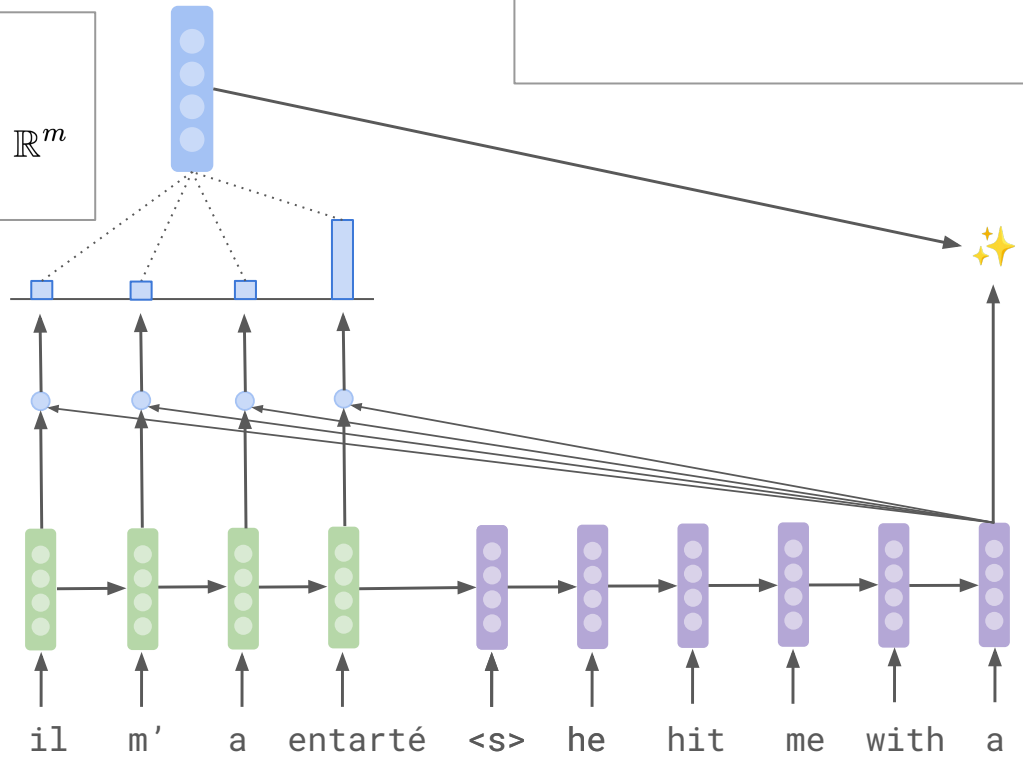
Seq2seq with attention

Compute weighted sum of encoder hidden states:

$$a_t = \sum_{i=1}^{n} a_i^t h_i^{enc} \in \mathbb{R}^m$$

Apply an activation function

$$\tilde{h}_t = \tanh\left(W_c \left[a_t; h_t^{dec}\right]\right) \in \mathbb{R}^m$$



Seq2seq with attention

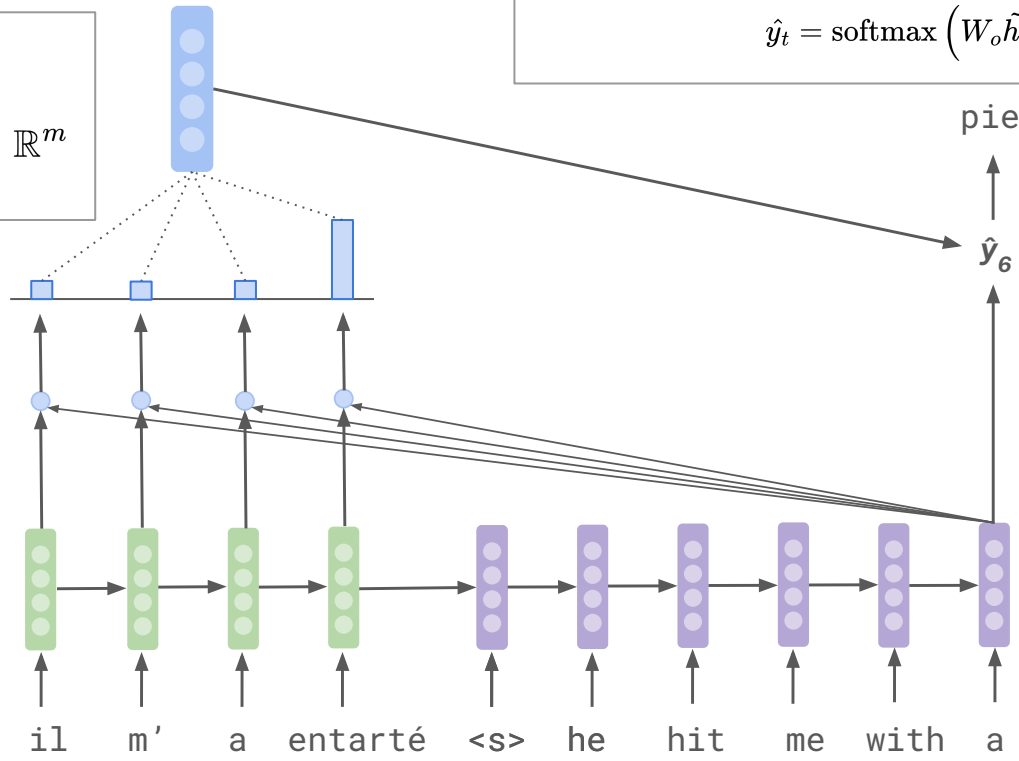
Compute weighted sum of encoder hidden states:

$$a_t = \sum_{i=1}^{n} a_i^t h_i^{enc} \in \mathbb{R}^m$$

Pass onto the output layer

$$\tilde{h}_t = \tanh\left(W_c\left[a_t; h_t^{dec}\right]\right) \in \mathbb{R}^m$$

$$\hat{y}_t = \text{softmax}\left(W_o \tilde{h}_t\right)$$



Seq2seq with attention

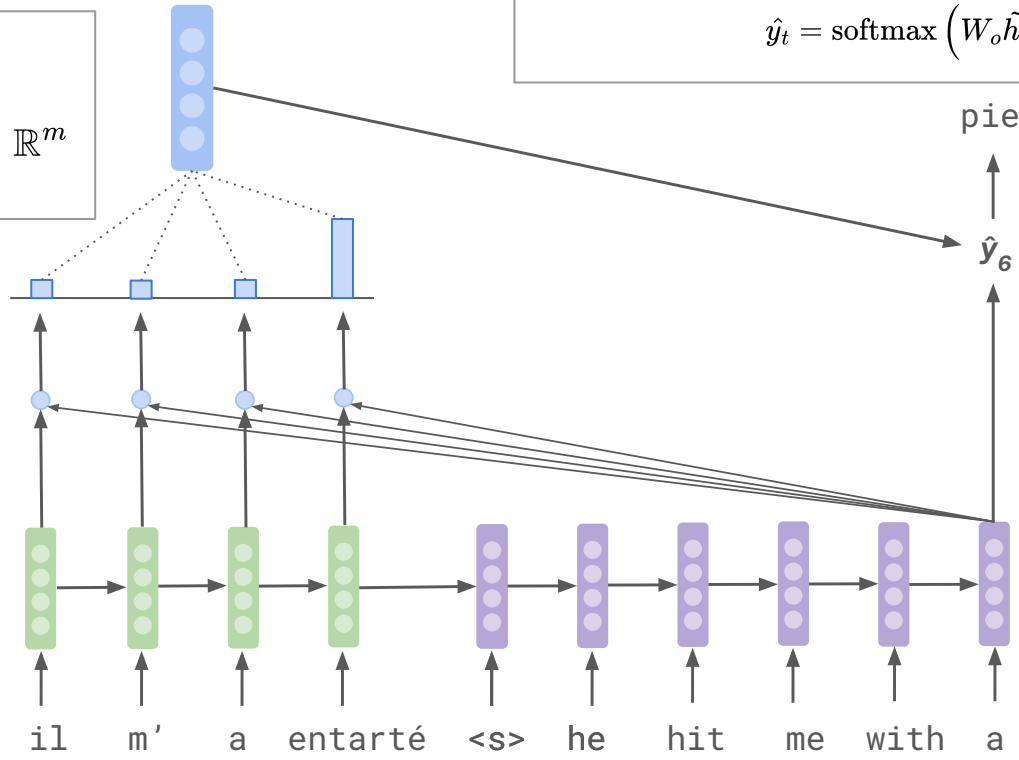
Compute weighted sum of encoder hidden states:

$$a_t = \sum_{i=1}^{n} a_i^t h_i^{enc} \in \mathbb{R}^m$$

Pass onto the output layer

$$\tilde{h}_t = \tanh\left(W_c \left[a_t; h_t^{dec}\right]\right) \in \mathbb{R}^m$$

$$\hat{y}_t = \text{softmax}\left(W_o \tilde{h}_t\right)$$



Question:

$$W_c \in \mathbb{R}^{??\times??}$$

Seq2seq with attention

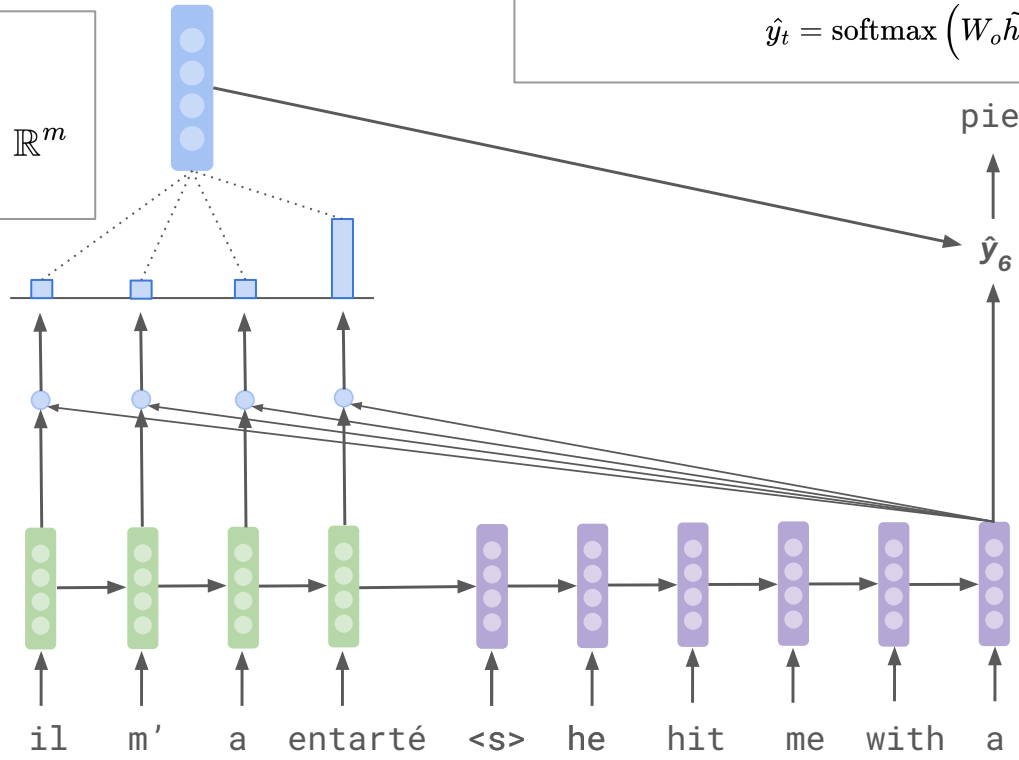
Compute weighted sum of encoder hidden states:

$$a_t = \sum_{i=1}^{n} a_i^t h_i^{enc} \in \mathbb{R}^m$$

Pass onto the output layer

$$\tilde{h}_t = \tanh\left(W_c \left[a_t; h_t^{dec}\right]\right) \in \mathbb{R}^m$$

$$\hat{y}_t = \text{softmax}\left(W_o \tilde{h}_t\right)$$



Question:

$$W_o \in \mathbb{R}^{??\times??}$$

A post-lecture note:

Expressions & equations from the previous slide

$$a_t = \sum_{i=1}^{n} a_i^t h_i^{enc} \in \mathbb{R}^m$$

$$\tilde{h}_t = \tanh\left(W_c \left[a_t; h_t^{dec}\right]\right) \in \mathbb{R}^m$$

$$\hat{y}_t = \text{softmax}\left(W_o \tilde{h}_t\right)$$

$$W_c \in \mathbb{R}^{??\times??}$$

The original answer for W_c's dimensions I had in my notes was ***2m x m***, from teaching materials provided by an NLP professor friend.

However, if we were to explain this "solution", it seemed like our process doesn't match what the solution says! The process:

$$h_t^{dec} \in \mathbb{R}^{m\times1} \qquad a_t \in \mathbb{R}^{m\times1} \qquad \left[a_t; h_t^{dec}\right] \in \mathbb{R}^{2m\times1}$$

$$W_c \left[a_t; h_t^{dec}\right] \in \mathbb{R}^{m\times1}$$

From the equation for $\tilde{h}$, since tanh doesn't affect dimensions

The above indicates that W_c is transforming our concatenated vector from *2m* x 1 to something that's *m* x 1.

A post-lecture note

Continuing from the prev slide:

$h_t^{dec} \in \mathbb{R}^{m \times 1}$ $a_t \in \mathbb{R}^{m \times 1}$ $[a_t; h_t^{dec}] \in \mathbb{R}^{2m \times 1}$ $W_c \left[a_t; h_t^{dec}\right] \in \mathbb{R}^{m \times 1}$

In order for $\boldsymbol{W_c}$ to turn our concatenated *2m*-dimensional vector back into an *m* dimensional vector, it should be *m* x *2m*. For instance, if we multiply **A** (an *m* x *n* dimensional matrix) by matrix **B** (an *n* x *p* dimensional matrix)**:**

Should be the same so that the rows of A can match up with the columns of B, to make it possible to compute the row-to-column dot products that make up the resultant matrix.

m x *n* *n* x *p*

These become the dimensions the result of **AB**: m x p

A post-lecture note

Two examples of how the inner dimension values match up during the multiplication, and the outer becomes the dimensions of the result:

$$W = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix} \in \mathbb{R}^{4\times 3} \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \in \mathbb{R}^{3\times 1} \qquad y = Wx = \begin{bmatrix} ax_1 + bx_2 + cx_3 \\ dx_1 + ex_2 + fx_3 \\ gx_1 + hx_2 + ix_3 \\ jx_1 + kx_2 + lx_3 \end{bmatrix} \in \mathbb{R}^{4\times 1}$$

$$W_c \in \mathbb{R}^{m\times 2m} \qquad [a_t; h_t^{dec}] \in \mathbb{R}^{2m\times 1} \qquad \tilde{h}_t = \tanh\left(W_c \left[a_t; h_t^{dec}\right]\right) \in \mathbb{R}^{m}$$

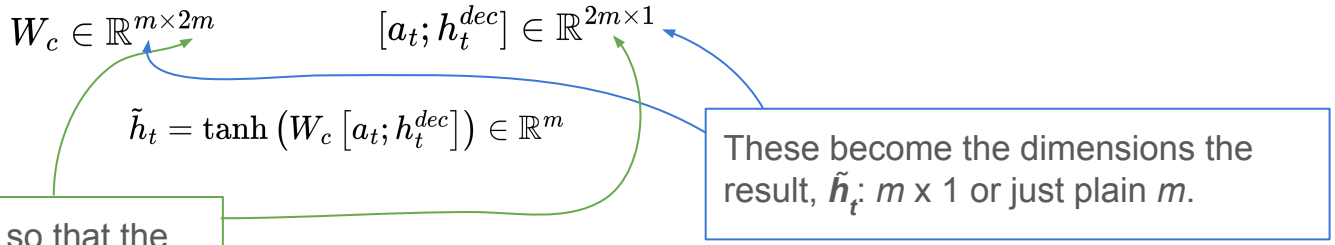
(*m* indicates *m* x 1 is implicit here)

Something is not adding up here... My friend said 2m x m. Am I a silly goose or is my friend a silly goose??

Lucy, during lecture

A post-lecture note

Continuing from the prev slide:



Should be the same so that the rows of A can match up with the columns of B, to make it possible to compute the row-to-column dot products that make up the resultant matrix.

These become the dimensions the result, $\tilde{h}_t$: *m* x 1 or just plain *m*.

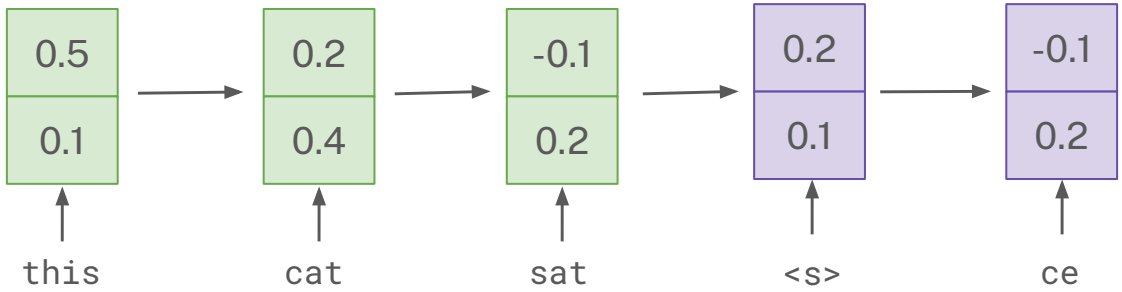
So why did the teaching notes say ***2m x m*** as the dimensions of W_c??? Well, NLP people sometimes think of vectors as row vectors (common in code, since often embeddings are stored as rows) instead of column vectors (like math equations have it). So their matrices could get transposed b/t equations and what they actually implement and write up. Altogether, ***I should have caught the discrepancy in this person's notes before reusing them***! 🕵 So the tl;dr is both me and my friend are silly gooses, together. Next slide continues our lecture...

Seq2seq with attention

Key idea: At each time step during decoding, focus on a particular part of source sentence.

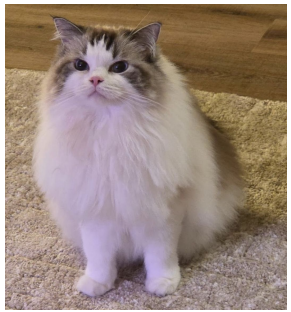
- Which part you focus on depends on the decoder's current hidden state
- Usually implemented as a probability distribution over the hidden states of the encoder

Question



Using dot product attention, which input word will have the highest attention value at the current time step (“ce”)?

A) the
B) cat
C) sat



Attention is great

Attention significantly improves NMT performance

Attention provides **a more “human-like” model of the MT process**

- Model can look back at the source sentence while translating, rather than needing to “remember” it all

Attention solves **the bottleneck problem**

- Attention allows decoder to look directly at source

Attention helps with **the vanishing gradient problem**

- Provides shortcut to faraway states

Attention is great

Attention provides **some interpretability**

➔ The network learns sequence alignment by itself!

One issue: attention has quadratic cost wrt sequence length

Attention, more generally

So far, we looked at attention for seq2seq NMT

However, you can use attention in many architectures and tasks

More general definition:

Given a set of vector **values** that map onto a vector of **keys**, attention computes a weighted sum of the values, dependent on a vector **query**.

Query *attends to* the **values**.

Attention, as we saw it earlier

In our seq2seq + attention LM:

- **Query** = decoder hidden state

$$h_t^{dec}$$

- **Keys** = encoder hidden states

$$h_1^{enc}, \ldots, h_n^{enc}$$

- **Values** = encoder hidden states

$$h_1^{enc}, \ldots, h_n^{enc}$$

Compute the attention scores w/ query & keys

$$e^t = \left[h_1^{enc} \cdot h_t^{dec}, \ldots, h_n^{enc} \cdot h_t^{dec}\right]$$

Get the attention distribution

$$a^t = \text{softmax}\left(e^t\right)$$

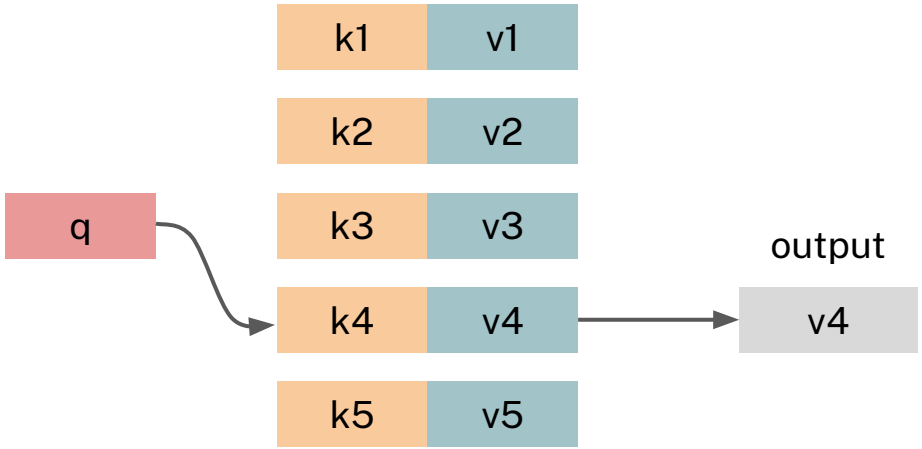
Take weighted sum of values

$$a_t = \sum_{i=1}^{n} a_i^t h_i^{enc}$$

Attention, more generally

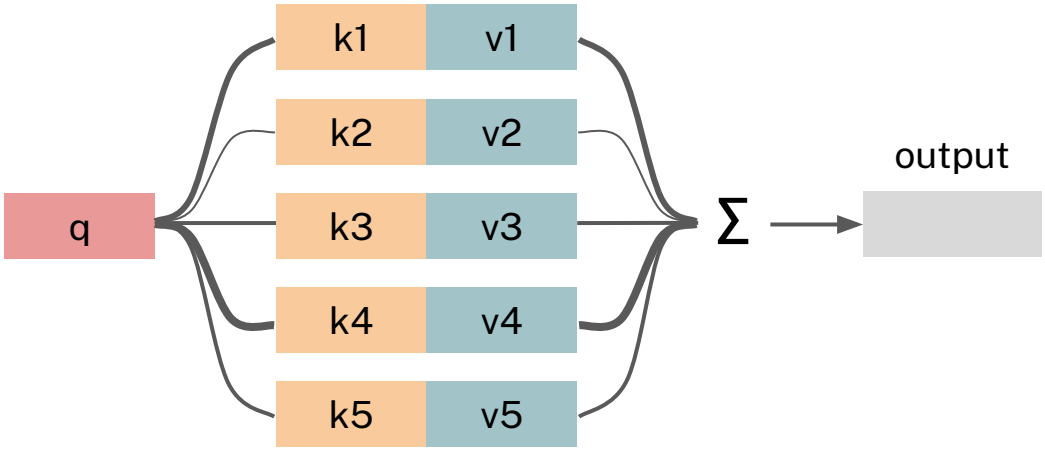
The terminology we're using comes from lookup tables

In a lookup table, we have a table of **keys** that map to **values**. The **query** matches one of the **keys**, returning its **value**.

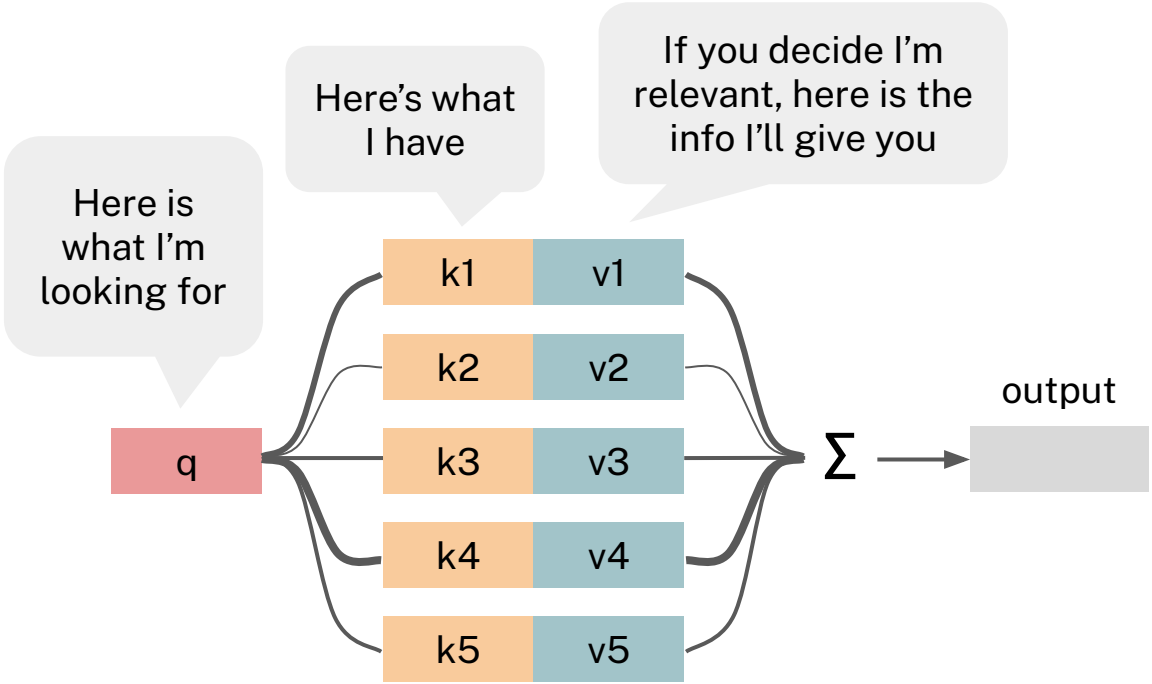


Attention, more generally

In attention, the **query** matches all **keys** *softly*, to a weight between 0 and 1. The **keys**' **values** are multiplied by the weights and summed.



Attention, more generally



Attention, as we saw it earlier

In our seq2seq + attention LM:

- **Query** = decoder hidden state

$$h_t^{dec}$$

- **Keys** = encoder hidden states

$$h_1^{enc}, \ldots, h_n^{enc}$$

- **Values** = encoder hidden states

$$h_1^{enc}, \ldots, h_n^{enc}$$

Compute the attention scores w/ query & keys

$$e^t = \left[h_1^{enc} \cdot h_t^{dec}, \ldots, h_n^{enc} \cdot h_t^{dec}\right]$$

Get the attention distribution

$$a^t = \text{softmax}\left(e^t\right)$$

Take weighted sum of values

$$a_t = \sum_{i=1}^{n} a_i^t h_i^{enc}$$

Generally, keys and values can be different vectors!

Attention, generally

A model general form

- **Query**

$$q$$

- **Keys**

$$k_1, \ldots, k_n$$

- **Values**

$$v_1, \ldots, v_n$$

Compute the attention scores w/ query & keys

$$e^t = [q \cdot k_1, \ldots, q \cdot k_n]$$

Get the attention distribution

$$a^t = \text{softmax}\left(e^t\right)$$

Take weighted sum of values

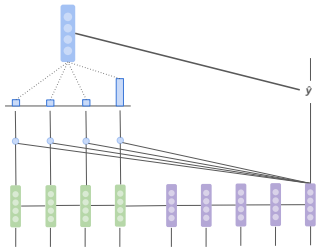
$$a_t = \sum_{i=1}^{n} a_i^t v_i$$

Do we even need recurrence at all?

Attention is a way to **pass information** in a neural net.

But this is also what RNNs are used for! But they lack parallelizability, inhibiting training on very large datasets.

Can we just get rid of the RNN?



2014-2017ish
Recurrence

Lots of trial

and error

2021+
(the rest of today's lecture)

RIP the RNN

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

The transformer

Transformer encoder-decoder

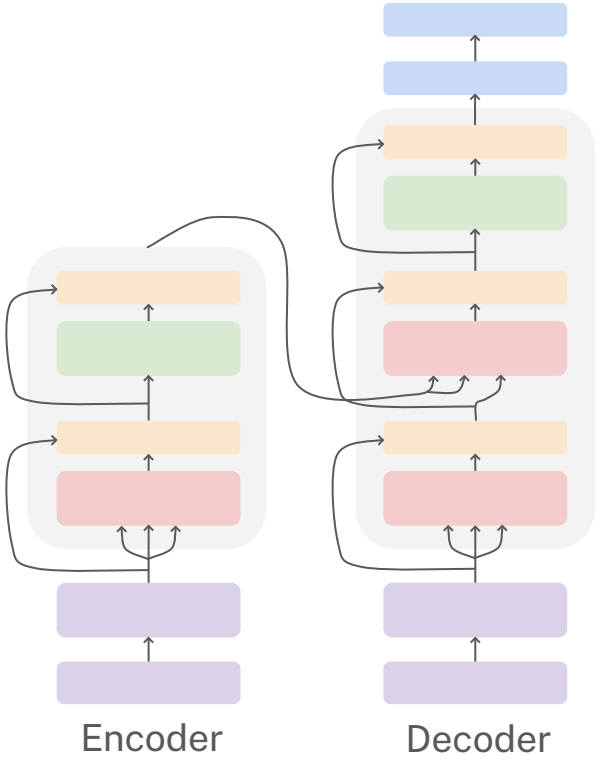
Designed and experimented on NMT

Transformer **encoder** = a stack of encoder layers

Transformer **decoder** = a stack of decoder layers

Key innovations: **multi-head**, **self-attention**

Replaces seq2seq + attention based on RNNs. No recurrence!



A key building block

Self Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

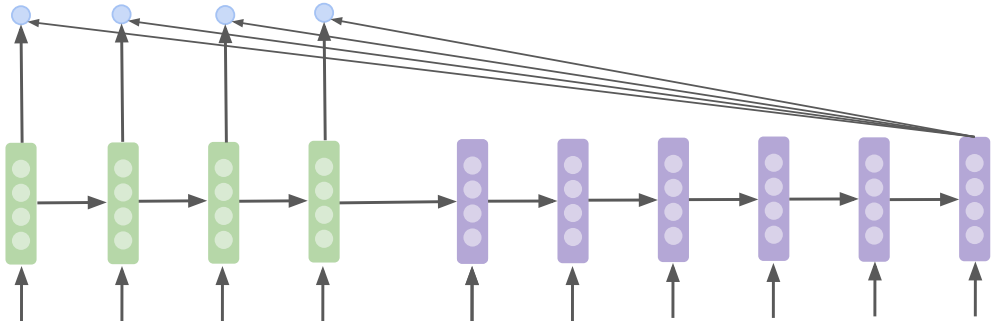
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

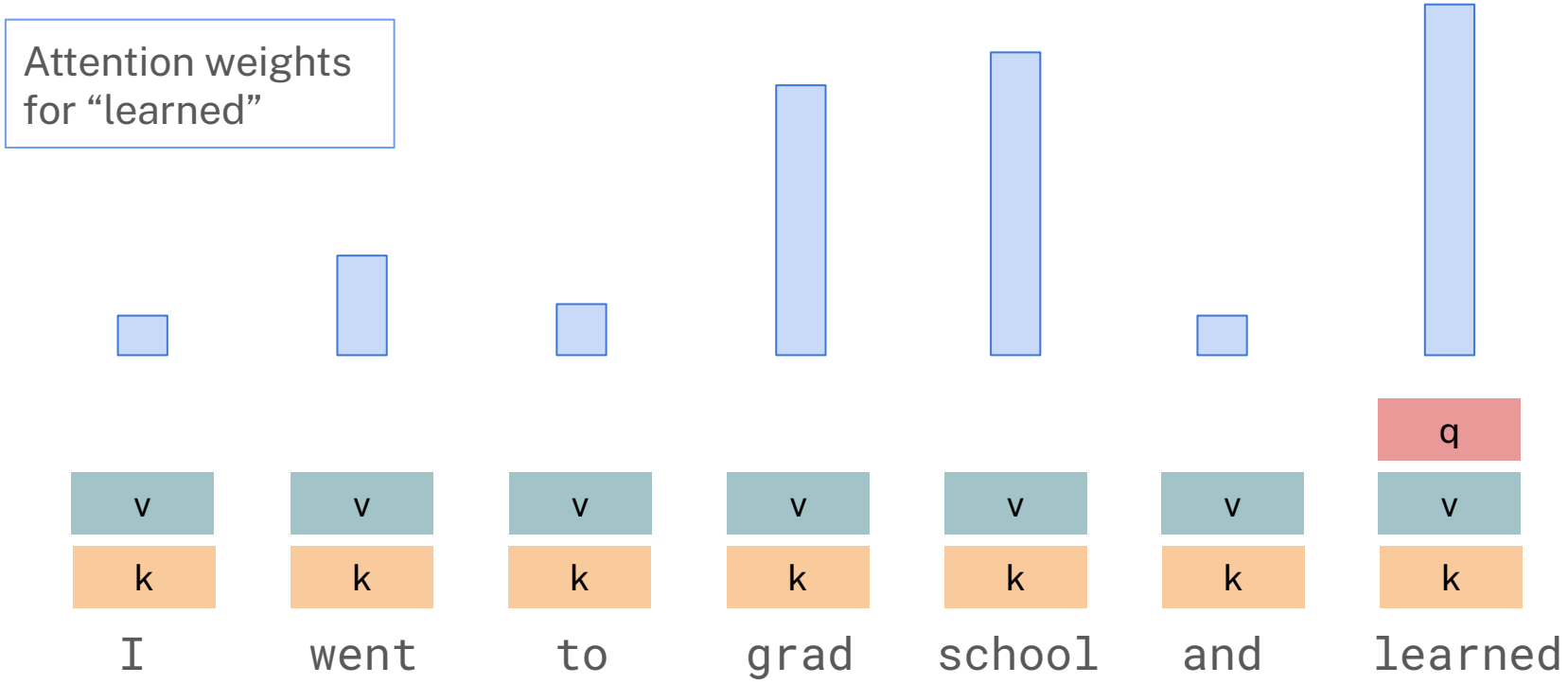
A key building block

What we talked about: **cross** attention, or paying attention to the input $\boldsymbol{x}$ to generate $\boldsymbol{y}_t$



What we need: **self** attention, or to generate $\boldsymbol{y}_t$, we need to pay attention to $\boldsymbol{y}_{\leq t}$

Self Attention Hypothetical Example



Self attention: k, q, v from the same sequence

Let $x_1, \ldots, x_n \in \mathbb{R}^d$ be input word embeddings for a sequence of words from vocabulary *V.*

Transform each word embedding with weight matrices ***Q***, ***K***, ***V***, each $\in \mathbb{R}^{d \times d}$

$q_i = Qx_i$ **(query)** $k_i = Kx_i$ **(keys)** $v_i = Vx_i$ **(values)**

Compute dot products b/t current query and all keys, normalize w/ softmax

$$e_{ij} = q_i \cdot k_j \qquad a_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})} = \text{softmax}\,(e_{ij})$$

Compute output for each word as a weighted sum of values: $o_i = \sum_j a_{ij} v_j$

Scaled dot product

Problem: the scale of the doc product increases as dimensions get larger!

$$a_{ij} = \text{softmax}\left(q_i \cdot k_j\right)$$

Fix: scale by the number of dimensions of the vector

$$a_{ij} = \text{softmax}\left(\frac{q_i \cdot k_j}{\sqrt{d}}\right)$$

Barriers and solutions for self attention

Barriers

No inherent notion of order! “The cat bit Lucy” isn’t distinguishable from “Lucy bit the cat”.

Solutions

Fixing the sequence order problem

Since self attention doesn't build in order information, we need to encode the order of a sequence in our keys, queries, and values.

Consider representing each sequence index as a vector

$$p_i \in \mathbb{R}^d, \text{ for } i \in \{1, 2, \ldots, n\}$$

We incorporate this vector into our self attention block by adding it to our input word embeddings, at the first layer:

$$\tilde{x}_i = x_i + p_i$$

Position representation vectors

Sinusoidal encoding, a.k.a. sine embeddings

- Concatenate sine and cosine functions of varying periods
  - Each dim corresponds to a sine or cosine. Position vectors are **a set of clock hands spinning at different speeds**.
- Now every position has a unique signature!
- Smooth and bounded (within 1 and -1) position values
- Relative distance b/t position vectors is preserved, e.g. the angle between position 1 and 2 is the same as the angle between position 5 and 6

Even-indexed dimensions

The 2*j*th dimension of p_i

$$\sin\left(\frac{i}{10000^{2j/d}}\right)$$

Odd-indexed dimensions

The (2j+1)th dimension of p_i

$$\cos\left(\frac{i}{10000^{2j/d}}\right)$$

Barriers and solutions for self attention

Barriers

No inherent notion of order! “The cat bit Lucy” isn’t distinguishable from “Lucy bit the cat”.

Solutions

Add position representations to the inputs.

Barriers and solutions for self attention

Barriers

No inherent notion of order! “The cat bit Lucy” isn’t distinguishable from “Lucy bit the cat”.

No nonlinearities for deep learning! It’s all just weighted averages.

Solutions

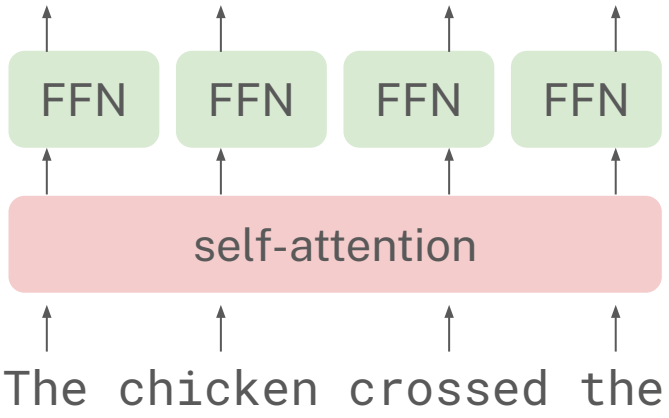
Add position representations to the inputs.

Adding nonlinearities to self attention

There's no elementwise nonlinearities to self attention.

This means adding more layers of self attention doesn't add more power or complexity.

Fix: add a feed-forward network (containing a ReLU) to post-process each self attention output



Barriers and solutions for self attention

Barriers

No inherent notion of order! “The cat bit Lucy” isn’t distinguishable from “Lucy bit the cat”.

No nonlinearities for deep learning! It’s all just weighted averages.

Solutions

Add position representations to the inputs.

Apply the same feedforward network to each self-attention output.

Barriers and solutions for self attention

Barriers

No inherent notion of order! “The cat bit Lucy” isn’t distinguishable from “Lucy bit the cat”.

No nonlinearities for deep learning! It’s all just weighted averages.

Need to ensure we don’t “look at the future” when predicting a sequence.

Solutions

Add position representations to the inputs.

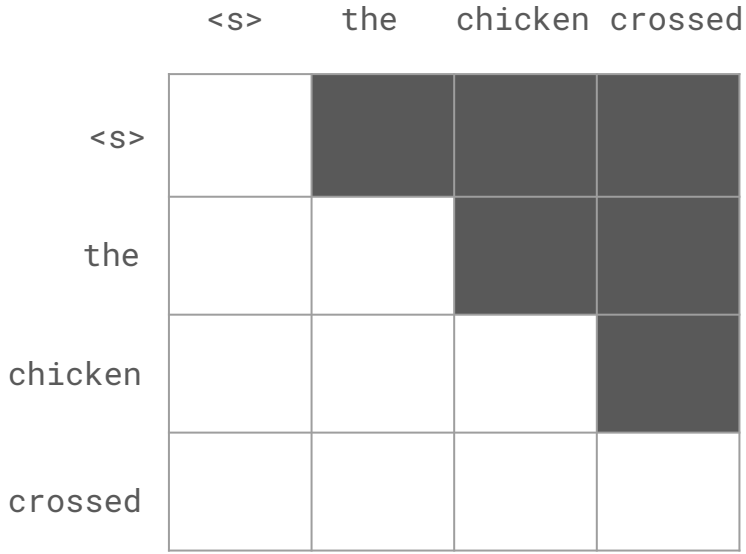
Apply the same feedforward network to each self-attention output.

Masking the future in self attention

When decoding, we can't peek at the future!

At inference, generation is **autoregressive**, so not knowing the future happens naturally.

During training, we have to enforce this **masking** ourselves



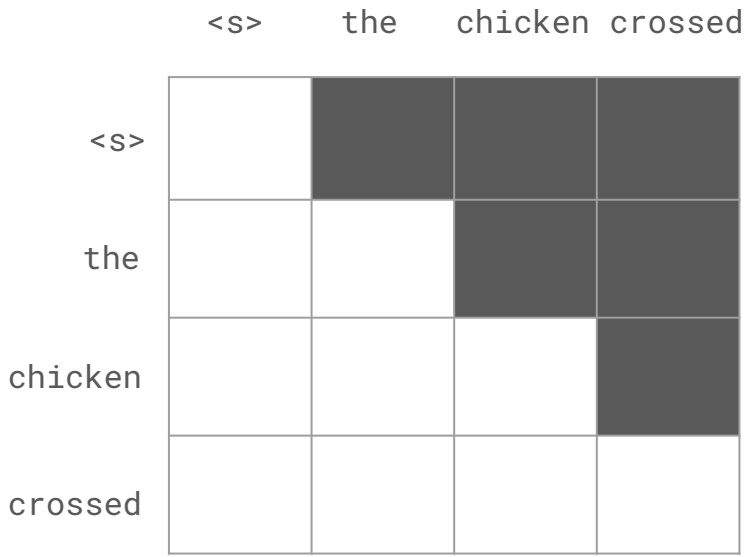
When computing attention for each word in each row, attend to words not greyed out in each column

Masking the future in self attention

During training:

At every timestep, we could change the set of keys and queries to include only past words. But this is inefficient.

To enable parallelization, we mask out attention to future words by setting attention scores to `-inf`.



Masking the future in self attention

During training:

At every timestep, we could change the set of keys and queries to include only past words. But this is inefficient.

To enable parallelization, we mask out attention to future words by setting attention scores to `-inf`.

	<s>	the	chicken	crossed
<s>	0.11	0.00	0.81	0.79
the	0.19	0.50	0.30	0.48
chicken	0.53	0.98	0.95	0.14
crossed	0.81	0.86	0.38	0.90

Let's say you have these attention scores

Masking the future in self attention

During training:

At every timestep, we could change the set of keys and queries to include only past words. But this is inefficient.

To enable parallelization, we mask out attention to future words by setting attention scores to `-inf`.

	<s>	the	chicken	crossed
<s>	0.11	-inf	-inf	-inf
the	0.19	0.50	-inf	-inf
chicken	0.53	0.98	0.95	-inf
crossed	0.81	0.86	0.38	0.90

Apply your attention mask.

Masking the future in self attention

During training:

At every timestep, we could change the set of keys and queries to include only past words. But this is inefficient.

To enable parallelization, we mask out attention to future words by setting attention scores to `-inf`.

	<s>	the	chicken	crossed
<s>	1.00	0.00	0.00	0.00
the	0.48	0.52	0.00	0.00
chicken	0.31	0.35	0.34	0.00
crossed	0.25	0.26	0.23	0.26

Compute the softmax along each row.

Barriers and solutions for self attention

Barriers

No inherent notion of order! “The cat bit Lucy” isn’t distinguishable from “Lucy bit the cat”.

No nonlinearities for deep learning! It’s all just weighted averages.

Need to ensure we don’t “look at the future” when predicting a sequence.

Solutions

Add position representations to the inputs.

Apply the same feedforward network to each self-attention output.

Mask out the future by artificially setting weights.

What we've covered so far

Self-attention is all you need

Actually you also need

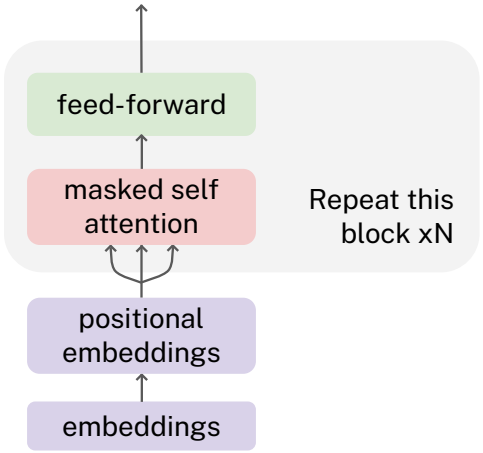
- **Position representations**
  - Sequence ordering
- **Nonlinearities**
  - With a feed-forward network
- **Masking**
  - Prevents peeking at future

What we've covered so far

Self-attention is all you need

Actually you also need

- **Position representations**
  - Sequence ordering
- **Nonlinearities**
  - With a feed-forward network
- **Masking**
  - Prevents peeking at future



our half-finished Transformer decoder

Up next: multi-head self attention

Self-attention is all you need

Actually you also need

Position representations

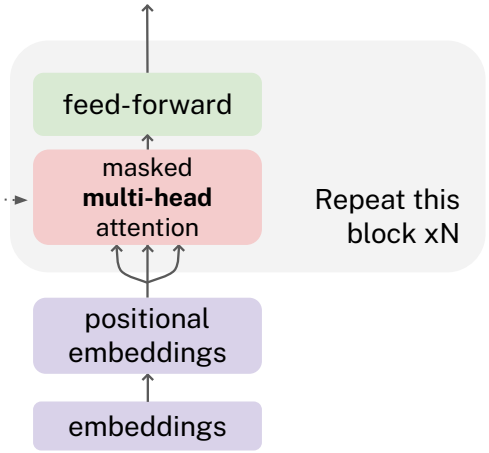
Sequence ordering

Nonlinearities

With a feed-forward network

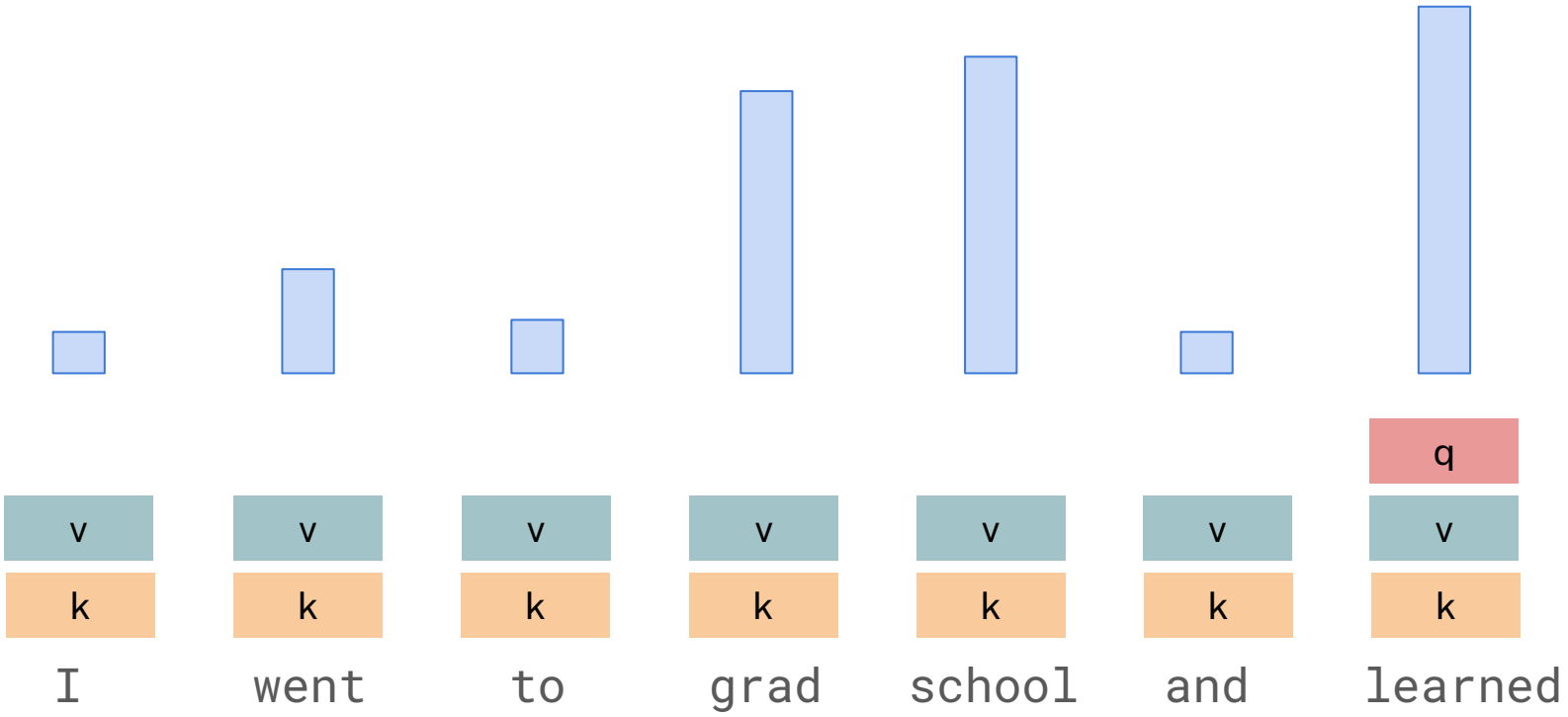
Masking

Prevents peeking at future



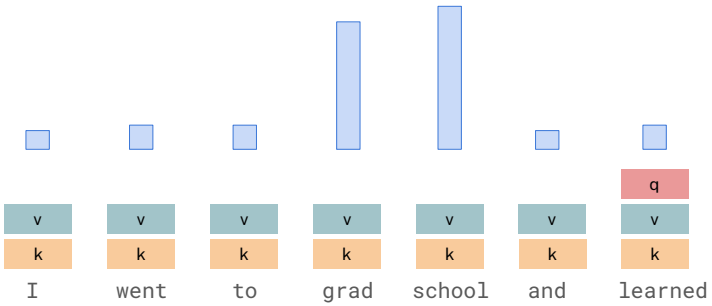
our half-finished
Transformer decoder

Recall this example

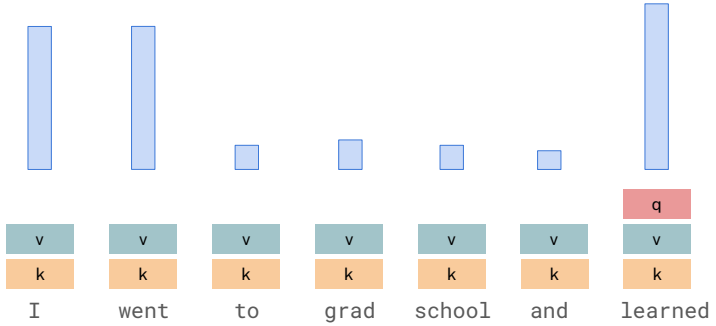


What if we have multiple attention heads?

Attention head 1 could attend to entities, e.g. locations



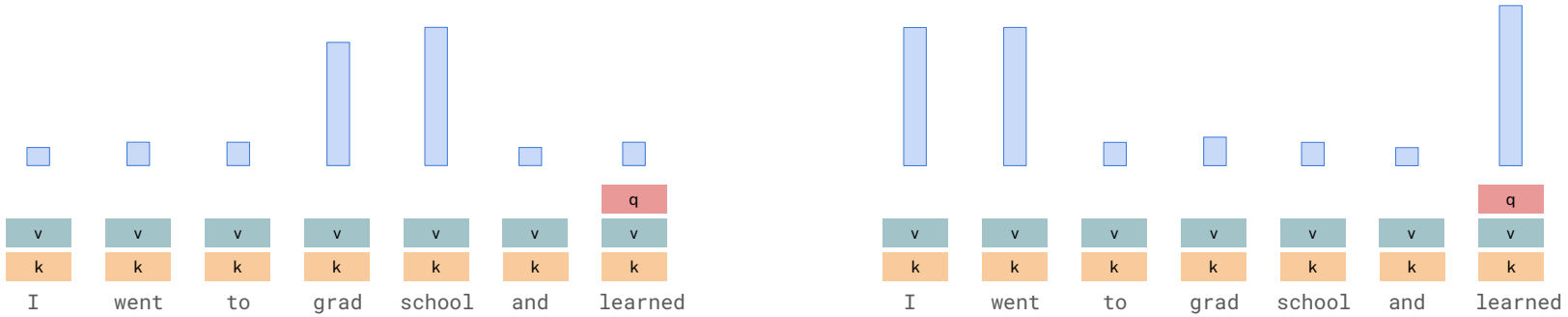
Attention head 2 could attend to syntactically relevant words



What if we have multiple attention heads?

Intuition: it's better to use multiple attention functions instead of one

Each attention function ("head") focuses on different key positions or content.



Up next: a walkthrough of multi-head attention w/ equations

Self attention: k, q, v from the same sequence

Let $x_1, \ldots, x_n \in \mathbb{R}^d$ be input word embeddings for a sequence of words from vocabulary *V.*

Transform each word embedding with weight matrices ***Q***, ***K***, ***V***, each $\in \mathbb{R}^{d \times d}$

$q_i = Qx_i$ **(query)** $k_i = Kx_i$ **(keys)** $v_i = Vx_i$ **(values)**

Compute dot products b/t current query and all keys, normalize w/ softmax

$$e_{ij} = q_i \cdot k_j \qquad a_{ij} = \text{softmax}\left(e_{ij}\right)$$

Compute output for each word as a weighted sum of values: $o_i = \sum_j a_{ij} v_j$

Let's use matrix notation!

Let $X = [x_1; \ldots; x_n] \in \mathbb{R}^{n \times d}$ be a stacking of words' input embeddings.

Transform each word matrix into:

$XQ \in \mathbb{R}^{n \times d}$ **(query)** $XK \in \mathbb{R}^{n \times d}$ **(keys)** $XV \in \mathbb{R}^{n \times d}$ **(values)**

Multiply keys and queries, normalize w/ softmax within rows

$$E = XQ(XK)^\top \qquad A = \text{softmax}\left(XQ(XK)^\top\right)$$

Compute output for each word as a weighted sum of values: $O = AXV \in \mathbb{R}^{n \times d}$

Multi-head attention

We can define multiple attention "heads" using multiple *Q*, *K*, *V* matrices

Let $Q_l, K_l, V_l \in \mathbb{R}^{d \times \frac{d}{h}}$ where *h* is the number of attention heads, and *l* ranges from 1 to *h*.

Multi-head attention

We can define multiple attention "heads" using multiple *Q*, *K*, *V* matrices

Let $Q_l, K_l, V_l \in \mathbb{R}^{d \times \frac{d}{h}}$ where *h* is the number of attention heads, and *l* ranges from 1 to *h*.

Each attention head performs attention independently:

$$O_l = \text{softmax}\left(XQ_lK_l^\top X^\top\right)XV_l \quad \in \mathbb{R}^{n \times \frac{d}{h}}$$

Multi-head attention

We can define multiple attention "heads" using multiple *Q*, *K*, *V* matrices

Let $Q_l, K_l, V_l \in \mathbb{R}^{d \times \frac{d}{h}}$ where *h* is the number of attention heads, and *l* ranges from 1 to *h*.

Each attention head performs attention independently:

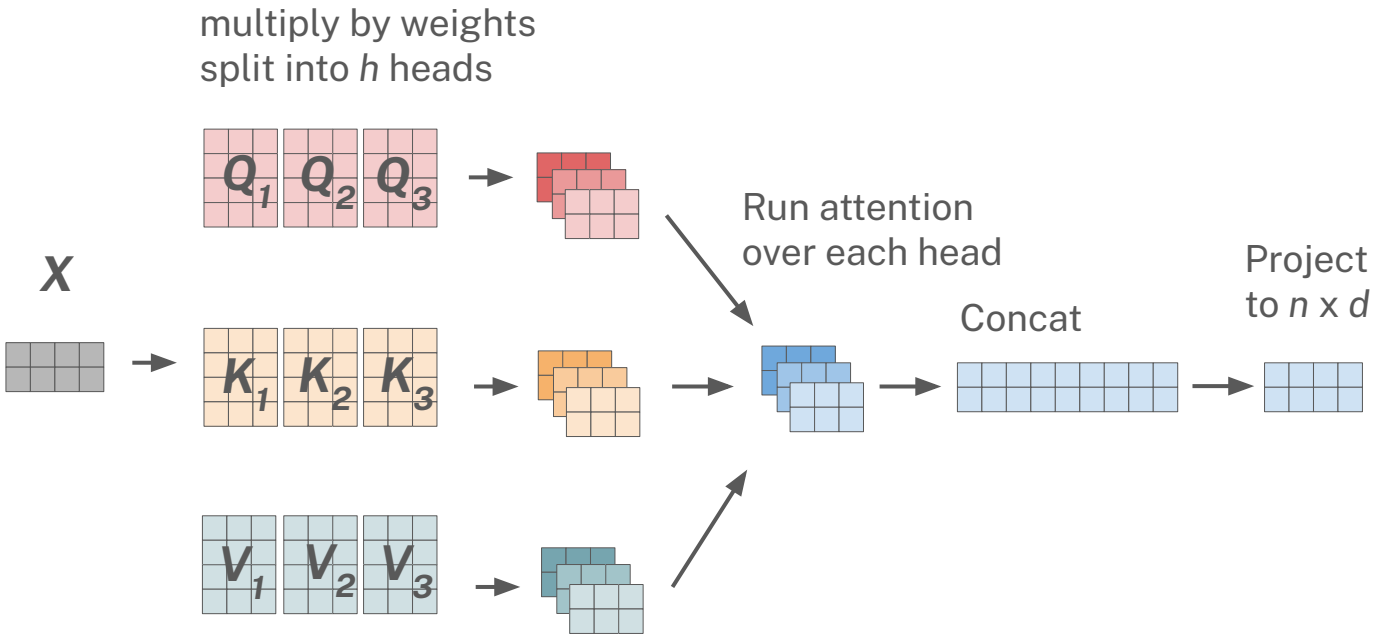
$$O_l = \text{softmax}\left(XQ_lK_l^\top X^\top\right)XV_l \quad \in \mathbb{R}^{n \times \frac{d}{h}}$$

Then the outputs of all the heads are concatenated, with $W_o \in \mathbb{R}^{d \times d}$.

$$O = [O_1, \ldots, O_h]W_o$$

Each head gets to "look" at different things.

Multi-head attention



Scaled dot product

Recall, with regular self attention, we actually scale our dot products by the number of dimensions of the vector.

$$a_{ij} = \text{softmax}\left(\frac{q_i \cdot k_j}{\sqrt{d}}\right)$$

For multi-head attention, we use the dimensionality, divided by the # of heads.

$$A_l = \text{softmax}\left(\frac{XQ_lK_l^\top X^\top}{\sqrt{d/h}}\right)$$

Multi-head attention

Multi-head self attention is **computationally efficient**.

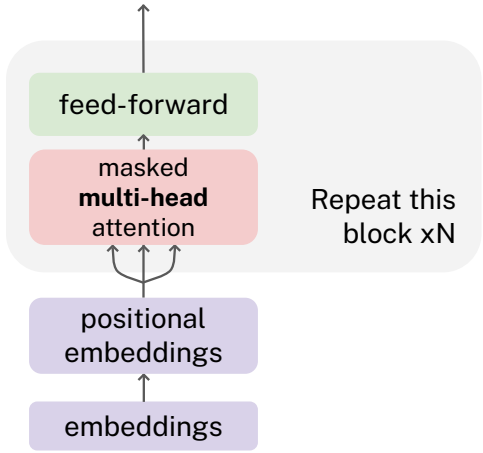
Even though we compute *h* attention heads, it's not more costly than regular self attention!

The matrices are the same sizes, just split up / rearranged!

By making use of matrix operations, computations are nearly identical.

Where we are now

Multi-head self attention allows us to focus on different parts of a sequence for different reasons.



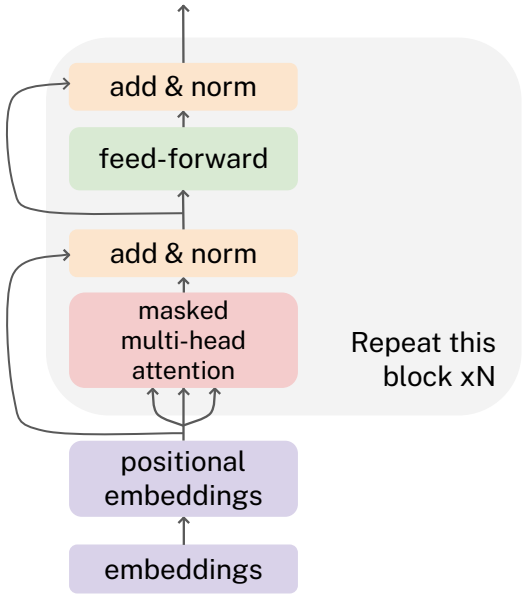
our half-finished
Transformer decoder

Up next: two optimization tricks

Residual connections

Layer normalization

In most diagrams, these are often written as “Add & Norm”



our kinda-finished Transformer decoder

Residual connections

Add an additive connection between the input and the output

$$\text{residual}\left(x,\ f\left(x\right)\right) = x + f\left(x\right)$$

Intuition:

- Prevents vanishing gradients
- Allows *f*, e.g. self-attention or FNN, to learn the diff between the inputs and outputs
- Widely used in convolutional neural networks (CNNs), and adopted in transformers

Layer normalization

Help models train faster!

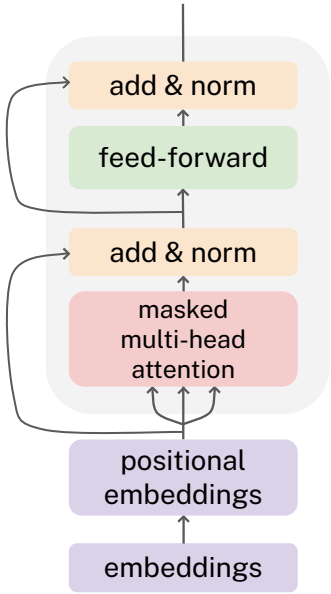
Reduces variance in the scale of hidden vector values.

$$\text{output} = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

Normalizes vectors to unit mean and standard deviation within each layer

Learned “gain” and “bias” parameters, that modulate the normalized value.

Something's missing at the top …



our kinda-finished
Transformer decoder

Linear & Softmax

Don’t forget our goal: predict the next word

The final output should be a probability distribution $\hat{y}$ over our vocabulary.

First, we convert the transformer output z into the shape we want (**linear layer**), and then apply **softmax** to make it a probability distribution.

$$\hat{y} = \text{softmax}\left(Uz\right) \in \mathbb{R}^{|V|} \qquad U \in \mathbb{R}^{|V| \times d}$$

During training: maximize the probability of the ground truth word

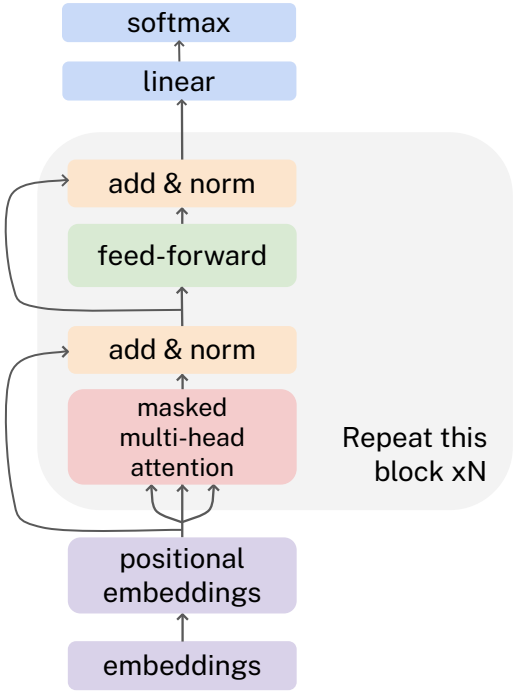
During inference: run decoding, e.g. take an argmax or sample

The transformer decoder

We finished our decoder!

Reminders:

- Add & normalize and FFN operate independently on each token.
- Self attention is the only component that models dependencies across tokens.

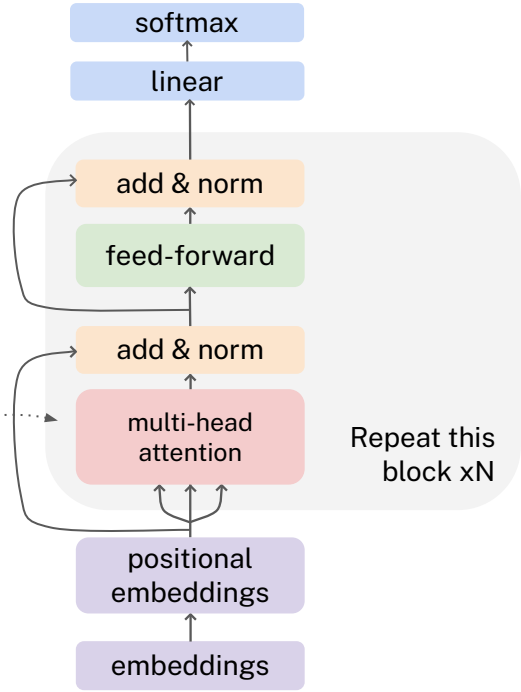


The transformer encoder

The decoder constrains the model to unidirectional context, e.g. language modeling.

If we want *bidirectional* context, can use the transformer encoder!

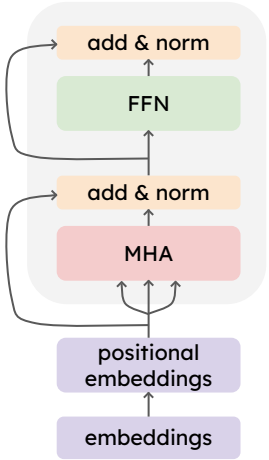
Only difference is we remove the masking in self-attention.



The transformer encoder-decoder

For a model to consider a source sentence with a bidirectional encoder and then generate the target with a unidirectional decoder, we can:

- Use a normal transformer encoder

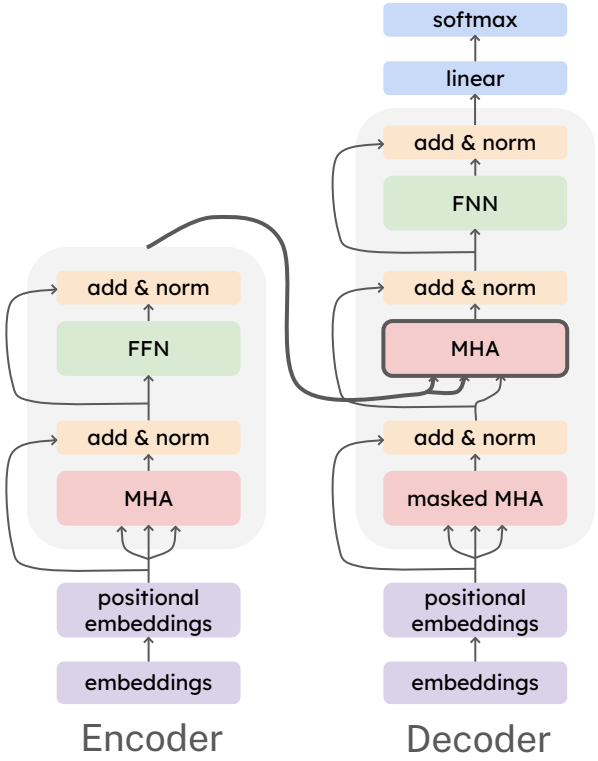


Encoder

The transformer encoder-decoder

For a model to consider a source sentence with a bidirectional encoder and then generate the target with a unidirectional decoder, we can:

- Use a normal transformer encoder
- Modify the transformer decoder to perform **cross-attention** to the output of the encoder



Transformers

Pros

- Easier to capture long-range dependencies, w/ attention b/t every pair of words
- Easier to parallelize

Cons

- Cannot generalize beyond a max sequence length defined before training, due to positional embeddings
- Another con on the next slide!

A key limitation

Quadratic computation in self-attention

- We compute all pairs of word interactions, so **our computation grows quadratically with sequence length**
- Can become slow and memory expensive

However, as transformers grow larger, a larger % of compute is *outside* the self-attention portion.

Very few large LMs use anything but quadratic cost attention, since cheaper methods tend not to work as well at scale.

Question for the future: is it worth trying to design cheaper alternatives?

Taking a step back…

We've come a long way!

RNNs

→ seq2seq

→ seq2seq + attention

→ transformers

In future weeks, when we say "language model", we'll most likely be referring to **transformer-based** language models.

practice questions

Let’s compute attention with code

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
```

Let’s compute attention with code

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
```

$XQ \in \mathbb{R}^{n \times d}$ (query) $XK \in \mathbb{R}^{n \times d}$ (keys) $XV \in \mathbb{R}^{n \times d}$ (values)

Let’s compute attention with code

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d = query.size(-1)
```

What comes next?

A) `scores = torch.matmul(key, query.transpose(-2, -1)) / math.sqrt(d)`

B) `scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d)`

C) `scores = torch.matmul(key, value.transpose(-2, -1)) / math.sqrt(d)`

D) `scores = torch.matmul(value, key.transpose(-2, -1)) / math.sqrt(d)`

Let’s compute attention with code

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d)
```

Let’s compute attention with code

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
```

Let’s compute attention with code

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
```

What comes next?

A) `torch.matmul(p_attn, value)`

B) `p_attn.masked_fill(mask == 0, -1e9)`

C) `torch.matmul(p_attn, key)`

D) `torch.matmul(p_attn, value.transpose(-2, -1))`

Let’s compute attention with code

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    res = torch.matmul(p_attn, value)
    return res, p_attn
```

$$O = \text{softmax}\left(XQ(XK)^\top\right)XV \quad \in \mathbb{R}^{n\times d}$$

Let’s compute attention with code

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    res = torch.matmul(p_attn, value)
    return res, p_attn
```

What’s missing?

Let’s compute attention with code

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    res = torch.matmul(p_attn, value)
    return res, p_attn
```

What part of the transformer is this?

```
class Mystery(nn.Module):

    def __init__(self, features, eps=1e-6):
        super(Mystery, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

What part of the transformer is this?

```
class Mystery(nn.Module):

    def __init__(self, d_model, d_ff, dropout=0.1):
        super(Mystery, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(self.w_1(x).relu()))
```

Question

What is the ratio of FLOPS (floating-point operations) look like for a 175B parameter model (e.g. GPT-3)?

MHA : FNN : others

a) 35% : 44% : 21%
b) 24% : 65% : 11%
c) 17% : 80% : 3%

transformer variants

Context

Since introduced in 2017 for NMT, we found that Transformers scale exceptionally well (GPU parallelization, no vanishing gradient issues, etc)

This ability to scale has made them very **powerful** at performing tasks

However, scaling also introduced new small and big challenges, requiring **many modifications** beyond the original Transformer design.

Transformer variants

Transformer variants

Today’s standard: Byte Pair Encoding (BPE)

We talked about this during our second lecture!

- Byte-level, frequency-based
- Start with individual bytes, then merge the most common co-occurring bytes
- Originally invented for data compression in 1994, GPT-2 picked it up, and it’s now standard in LLMs
- Relative to its precursor (on the next slide), BPE **packs the most data into the fewest tokens**. It can generate the same text with fewer tokens!

Original: WordPiece

Approach: start with individual characters, then merge co-occurring characters if merging increases the likelihood of the training data.

Character-level (not byte-level)

- If it encounters a character it wasn't trained on, it defaults to UNK
- Byte-level can represent any string of text w/o any UNK ever

Likelihood-based (not frequency-based)

- More expensive to calculate than BPE's "count and merge" algorithm

Transformer variants

RoPE: Rotary positional encodings

Goal: we want a *relative* position embedding, or some function *f* such that

$$f(x, i) \cdot f(y, j) = g(x, y, i - j)$$

That is, the output of dot product attention should only depend on the two words and their relative position, *i* -*j*.

Example: the relationship between “we” and “know” should be the same in both of these sentences:

- “we know that…”
- “Of course, we know that…”

Sine embeddings (covered earlier) don’t guarantee this (due to cross terms downstream)!

RoPE: Rotary positional encodings

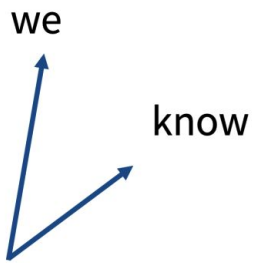
Another big difference from sine embeddings:

- Instead of incorporating position information at the *beginning* of the model
- We incorporate the position encoding in the **attention computation** in every layer

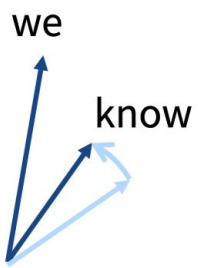
So, we apply RoPE on **queries** and **keys**.

RoPE: Rotary positional encodings

Instead of **adding** position to input embeddings, we **rotate** them based on position

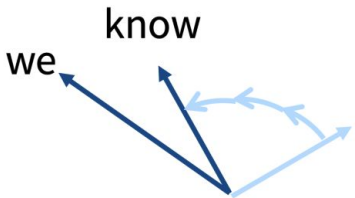


Position independent embedding



Embedding
“we know that”

Rotate we by ‘0 positions’
know by ‘1 positions’



Embedding
“of course we know”

Rotate we by ‘2 positions’
Rotate know by ‘3 positions’

From Tatsu’s slides

RoPE: Rotary positional encodings

Let's look at this in 2D first. This is a matrix that rotates vectors by θ:

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

If we compute the dot product of queries rotated by θ_1 and keys by θ_2:

$$(R_{\theta_1} q)^\top (R_{\theta_2} k) = q^\top R_{\theta_1}^\top R_{\theta_2} k = q^\top R_{\theta_2 - \theta_1} k$$

Transposing a rotation gives the *inverse* rotation ($-\theta_1$), and applying (multiplying) one rotation after another is equivalent to adding angles (θ_2-θ_1).

The only trace of position left here is the diff between the two angles!

If we shift tokens a few positions down a sequence, both angles grow by the same amount, and the score is identical. RoPE gives us **relative** position info.

RoPE: Rotary positional encodings

When we expand RoPE to more dimensions, we just pair up dimensions of our queries and keys and rotate them as if they're in 2D.

Here, t is the position of the word in the sequence, and R_t is our rotation matrix.

$$R_t = \begin{pmatrix} \cos t\theta_1 & -\sin t\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin t\theta_1 & \cos t\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos t\theta_2 & -\sin t\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin t\theta_2 & \cos t\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos t\theta_{d/2} & -\sin t\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin t\theta_{d/2} & \cos t\theta_{d/2} \end{pmatrix}$$

Transformer variants

Ordering of normalization

Post-norm - original transformers

MHA → Add → Norm → FFN → Add → Norm

Pre-norm - most modern LLMs

Norm → MHA → Add → Norm → FFN → Add

Pre-norm offers stability and larger learning rates for large models.

Choice of normalization

LayerNorm - original transformers

$$\text{output} = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

RMSNorm - most modern LMs

Doesn't subtract mean or add a bias term. Fewer operations and parameters, providing runtime and performance gains:

$$\text{output} = \frac{x}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 + \epsilon}} * \gamma$$

Transformer variants

Choice of activation

sigmoid	tanh	ReLU
$f(z) = \frac{1}{1+e^{-z}}$	$f(z) = \frac{e^{2z}-1}{e^{2z}+1}$	$f(z) = \max(0, z)$
$f'(z) = f(z)(1-f(z))$	$f'(z) = 1 - f(z)^2$	$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$

There's so many more:

ReLU, GeLU, SiLU, ELU, GLU, GeGLU, ReGLU, SeLU, SwiGLU, LiGLU, ...

Choice of activation

The current winner among modern LMs is **SwiGLU**.

Multiplies the two terms element by element

$$\text{SwiGLU}(x) = \text{SILU}(xW_g) \odot xW_u$$

Computes two projections of an input vector

Applies the SiLU activation to one projection

$$\text{SILU}(x) = x \cdot \sigma(x) = \frac{x}{1 - e^{-x}}$$

4 Conclusions

We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

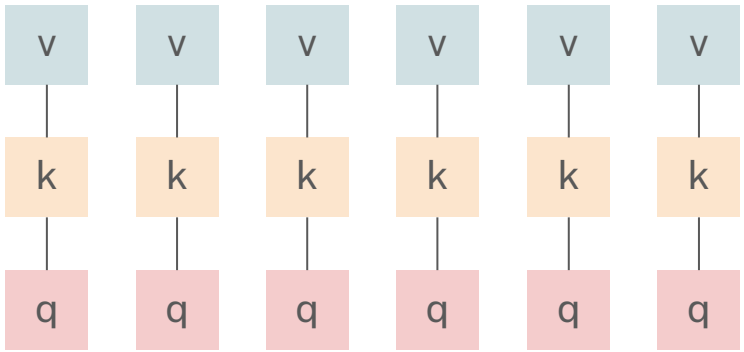
Transformer variants

Choice of self-attention

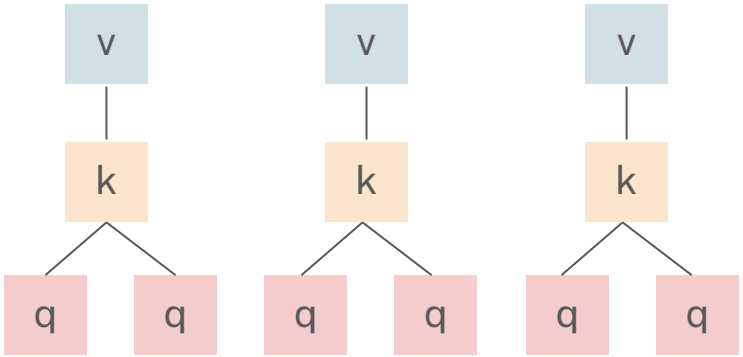
Grouped-query attention is the current winner

Shares the same key and value heads for a *group* of query heads

Saves memory, which leads to faster inference

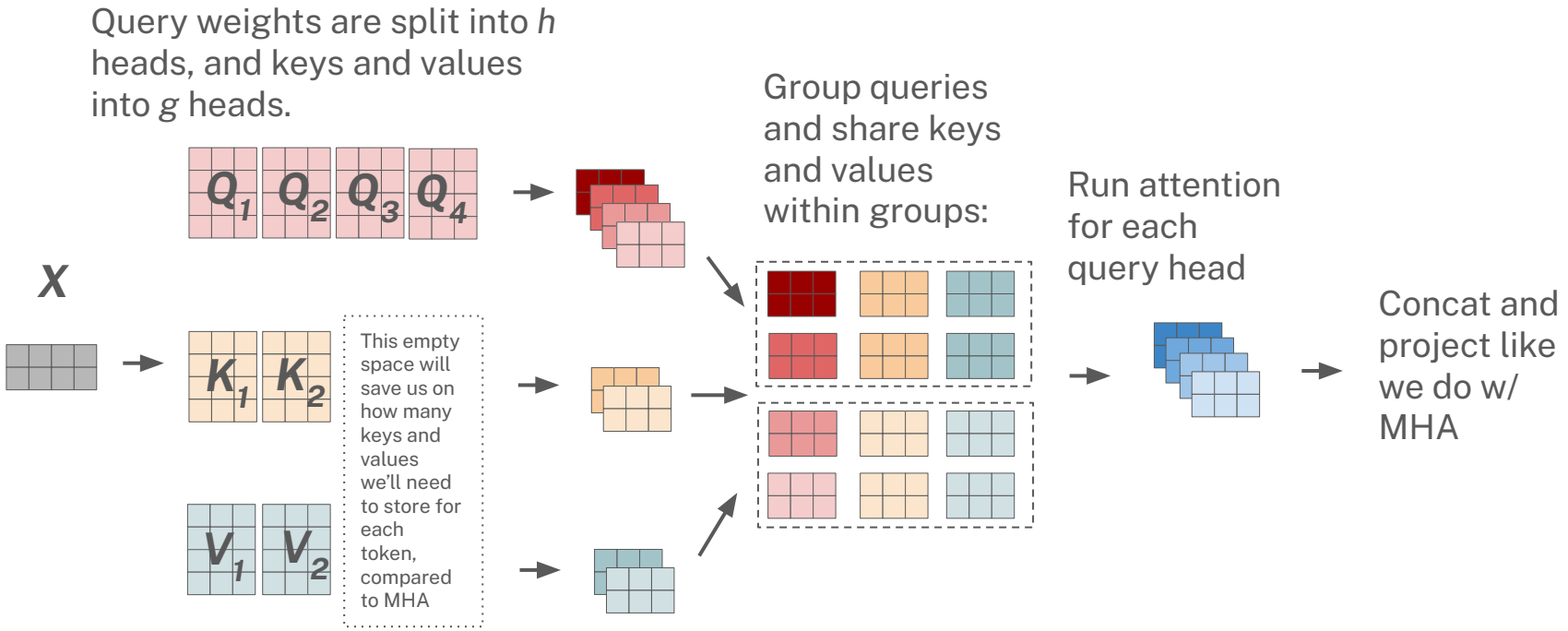


Multi-head attention



Grouped-query attention

Grouped-query attention



Transformer variants: summary

The modern LM typically involves...

Tokenization: BPE

Positional encoding: RoPE

Post-norm vs. pre-norm: pre-norm

LayerNorm vs. RMSNorm: RMSNorm

Activations: SwiGLU

Attention: Grouped-query attention

Open question

Current LLMs are predominantly based on transformer architectures.

How essential is it that LLMs use transformers?

One view: In 2018, many industry labs tried to train large LMs but with LSTMs. These attempts never worked.

Another view: With enough parameters (gazillions!), architecture matters less. Scale is all that matters! Just need a reasonable degree of nonlinearity and can be optimized stably.

Project brainstorming/proposal presentation

Sonia is helping me triaging team formation issues this week! If not resolved, please email both of us.

Your pitch next week should be **3 minutes**.

Submit one Google Slide.

We will compile all slides into one mega slide deck (taking the version of the document link closest to the deadline).

If you have presenter notes, have those with you, as we'll be running the mega presentation on Lucy's laptop on our end. Random order!!

Presentation grading rubric

- **Research question** (1 pt). This question should be scoped so that it is addressable within the weeks we have remaining in this class.
- **Core contribution** (2 pt). You should juxtapose your contributions against relevant prior papers, e.g. what do you add that past papers do not? Are you working on a new or spin-off problem that has not received the attention it deserves? Cite representative prior work.
- **Method** (1 pt). What models and tasks? Why these methods?
- **Data** (1 pt). You can find datasets by reading papers that contribute metadata-rich resources or benchmarks. You may also create a dataset from scratch, though it's possible that doing so may require most of the semester's time and become your main contribution (e.g. a resources and benchmarks paper).

Every member must present (-1 penalty if not)**.** You should assign each member one or two bullet points.