A brief introduction to (large) language models

Sofia Serrano

What are we going to talk about?

- The language modeling problem
- How do we learn a language model?
  - A quick primer on learning a model via gradient descent
  - The role of training data
- ✨✨ The Transformer ✨✨
  - The two things that make it such an improvement over our previous techniques for language modeling
  - More detail about both of those two things

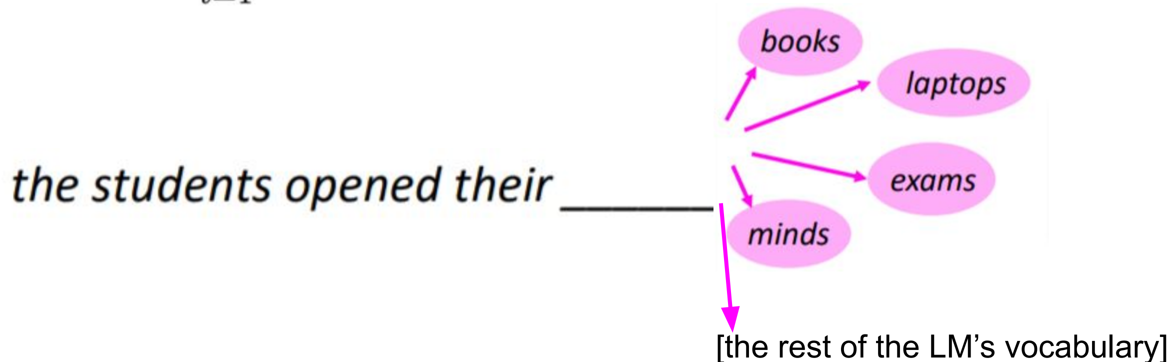
The language modeling problem

A **language model** answers the question: **What is p(text)?**

Tokenizing text using its predefined vocabulary (which includes a [STOP] token), a language model breaks that probability down as follows:

Just the chain rule of probability– no simplifying assumptions!

$$\text{p(text)} = \prod_{t=1}^{T} \text{p(token } t \text{ of text} \mid \text{tokens 1 through } t - 1 \text{ of text)}$$



Applying a language model

$\mathcal{V}$ = {permit, reject}   Our event space is $\mathcal{V}^*$ with <eos> at end   Our r.v. is X

What is p(X = reject permit <eos>)?

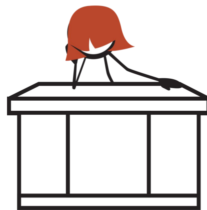
[start]

permit <eos> reject

permit <eos> reject permit <eos> reject

permit <eos> reject permit <eos> reject permit <eos> reject permit <eos> reject

Applying a language model



$\mathcal{V}$ = {permit, reject}   Our event space is $\mathcal{V}^*$ with <eos> at end   Our r.v. is X

What is p(X = reject permit <eos>)?

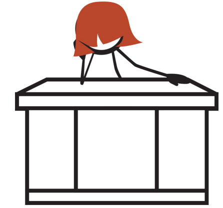
[start]

permit <eos> reject

permit <eos> reject permit <eos> reject

permit <eos> reject permit <eos> reject permit <eos> reject permit <eos> reject

Applying a language model



$\mathcal{V}$ = {permit, reject} Our event space is $\mathcal{V}^*$ with <eos> at end Our r.v. is X

What is p(X = reject permit <eos>)? Use chain rule of probability.

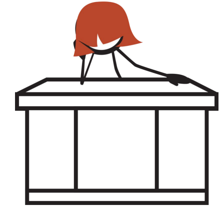
[start]

permit <eos> reject

permit <eos> reject permit <eos> reject

permit <eos> reject permit <eos> reject permit <eos> reject permit <eos> reject

Applying a language model



$\mathcal{V}$ = {permit, reject}   Our event space is $\mathcal{V}^*$ with <eos> at end   Our r.v. is X

p(X = reject permit <eos>) = p(reject | [start]) * p(permit | [start] reject) * p(<eos> | [start] reject permit)

[start]

permit <eos> reject

permit <eos> reject permit <eos> reject

permit <eos> reject permit <eos> reject permit <eos> reject permit <eos> reject

Applying a language model

$\mathcal{V}$ = {permit, reject}   Our event space is $\mathcal{V}^*$ with <eos> at end   Our r.v. is X

Note: as long as p(child of node) > 0 for each node and $\sum_{\text{child}}$ p(child of node) = 1 for each (non-eos) node, then $\sum_{\text{path in tree}}$ p(path) = 1

[start]

permit <eos> reject

permit <eos> reject permit <eos> reject

permit <eos> reject permit <eos> reject permit <eos> reject permit <eos> reject

Language models of this form can generate text

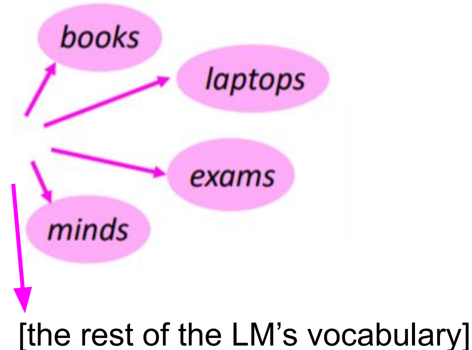
At each timestep, sample a token from the language model's new probability distribution over next tokens. (Might be naive sampling, top-k, nucleus sampling…)

The ____

The students ____

The students opened ____

The students opened their ____



[the rest of the LM's vocabulary]

How we learn a language model

How do we learn a language model?

Given a large corpus of text, split that text into all of its different language modeling subproblems and then:

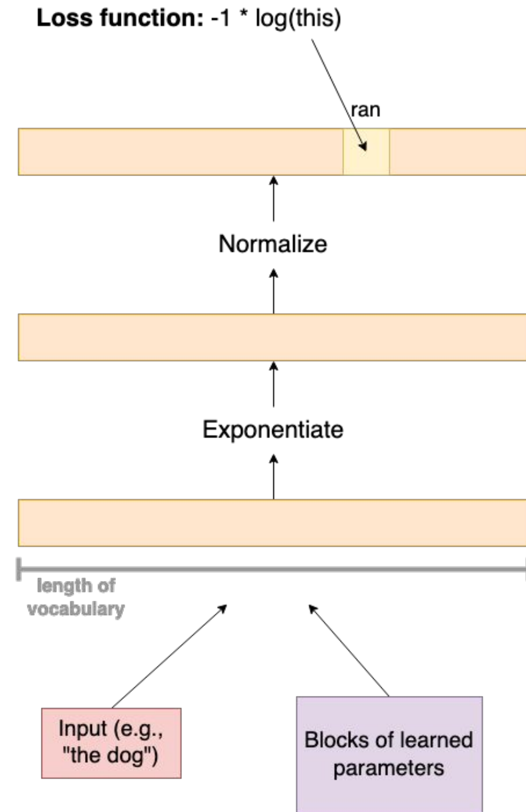
Maximize: p(observed token in text | all observed tokens before that token)

(summed over all tokens in text)

How do we maximize that quantity?

The dominant strategy from the past decade:

1. Compose a differentiable function of the input and some blocks of to-be-learned parameters
2. Have that function output a real-valued vector the length of the vocabulary
3. **Softmax** that vector to turn it into a probability distribution:
   a. Exponentiate it
   b. Normalize the exponentiated values
4. Treat the **negative log probability** of the correct token as your loss function
5. Differentiate with respect to the parameters, and perform gradient descent



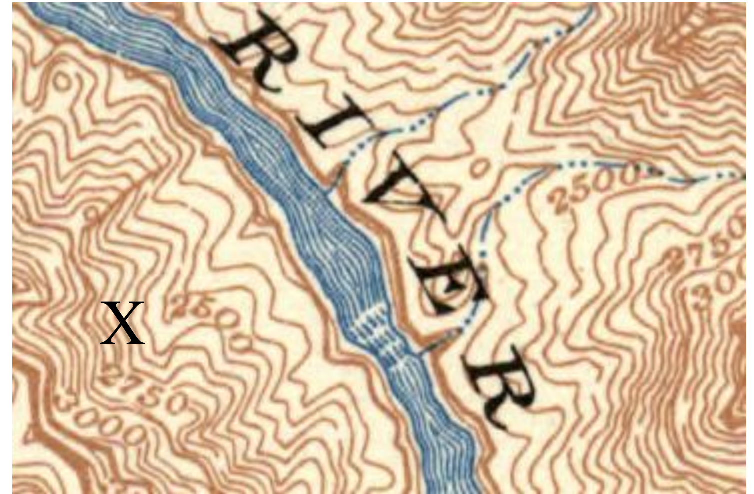
Intuition of gradient descent

How do I get to the bottom of this river canyon?

Look around me 360◦

Find the direction of steepest slope up

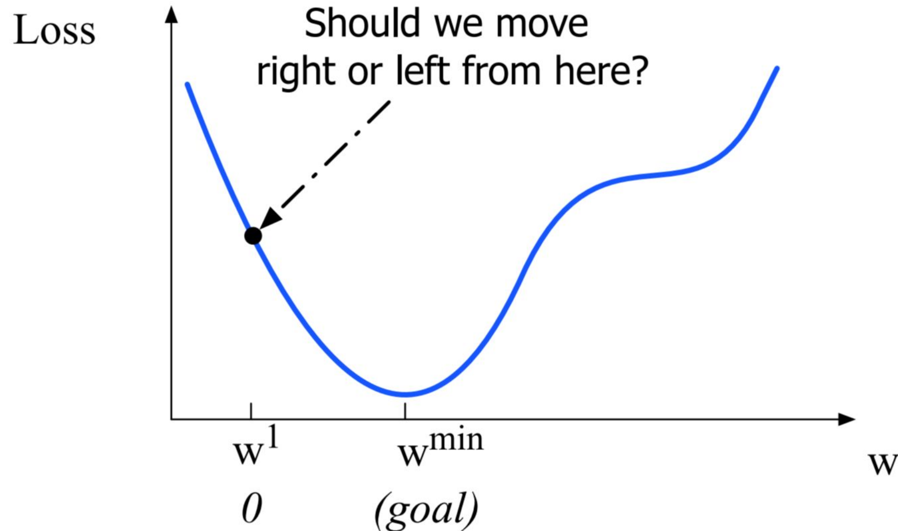
Go the opposite direction



Gradient descent: a throwback to calculus

Q: Given current parameter w, should we make w bigger or smaller to minimize our loss?

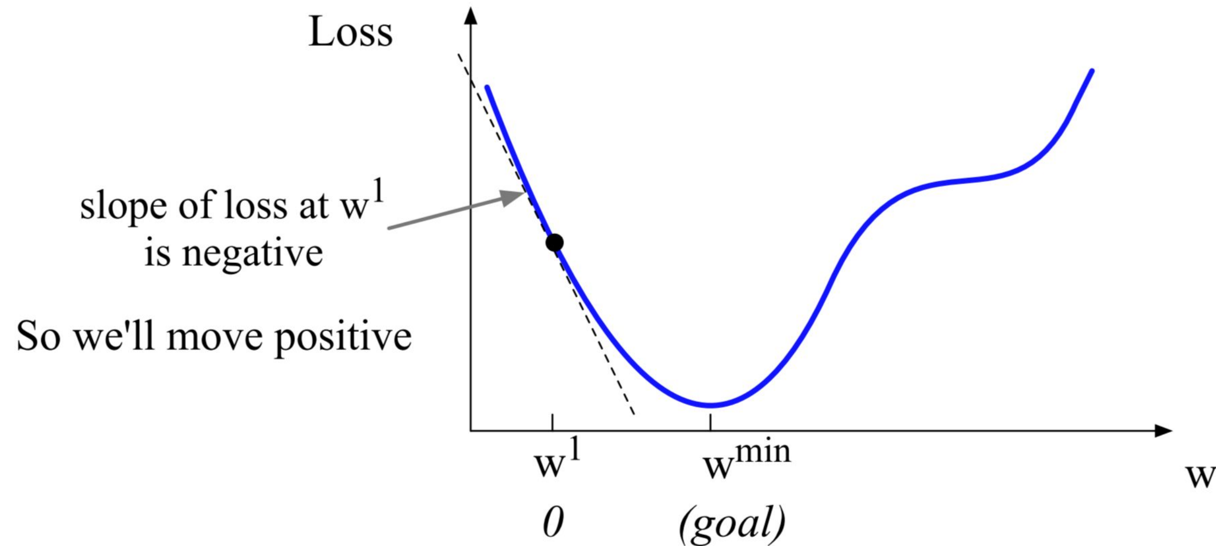
A: Move w in the reverse direction from the slope of the function



Let's first visualize for a single scalar w

Q: Given current w, should we make it bigger or smaller?

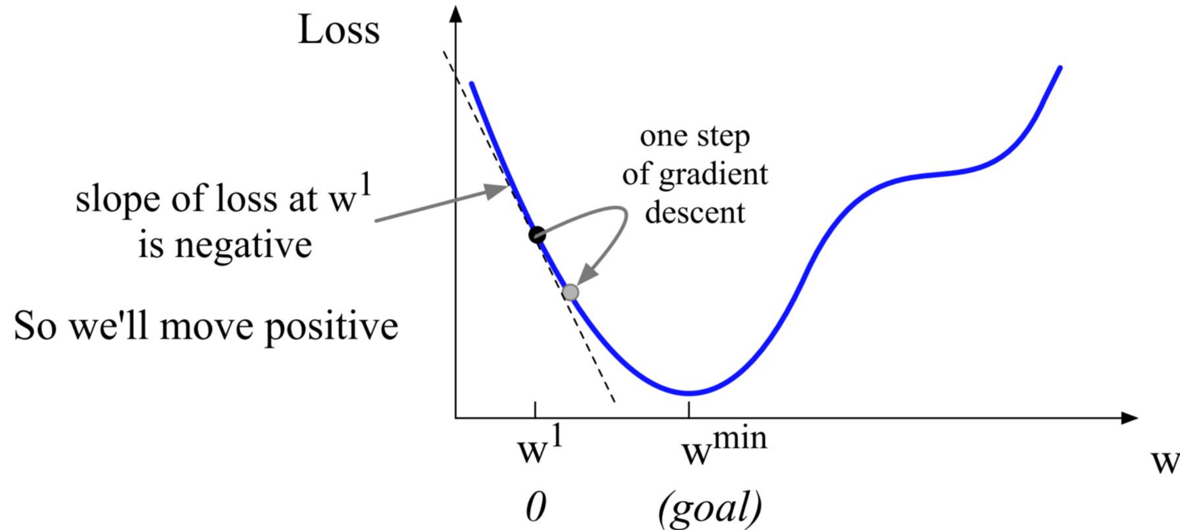
A: Move w in the reverse direction from the slope of the function



Let's first visualize for a single scalar w

Q: Given current w, should we make it bigger or smaller?

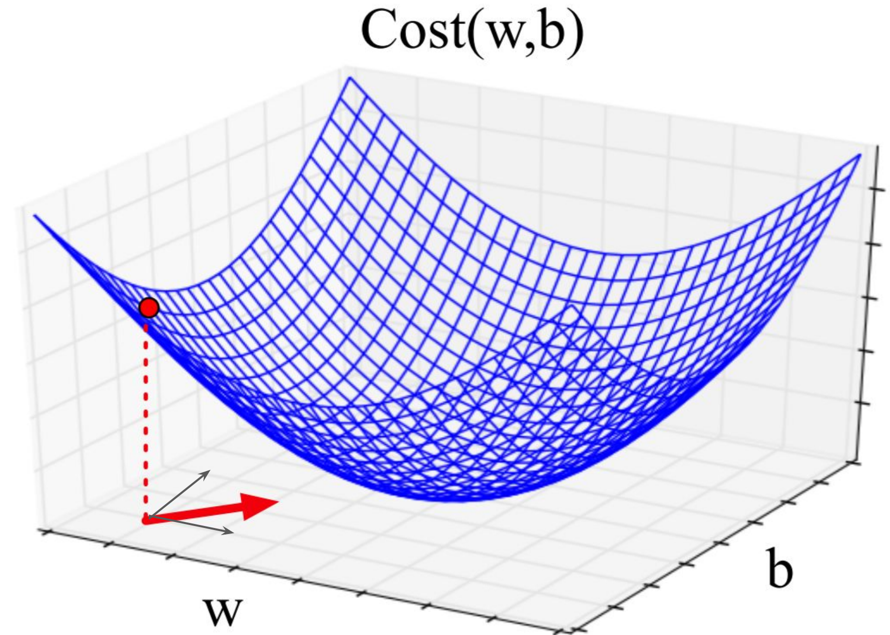
A: Move w in the reverse direction from the slope of the function



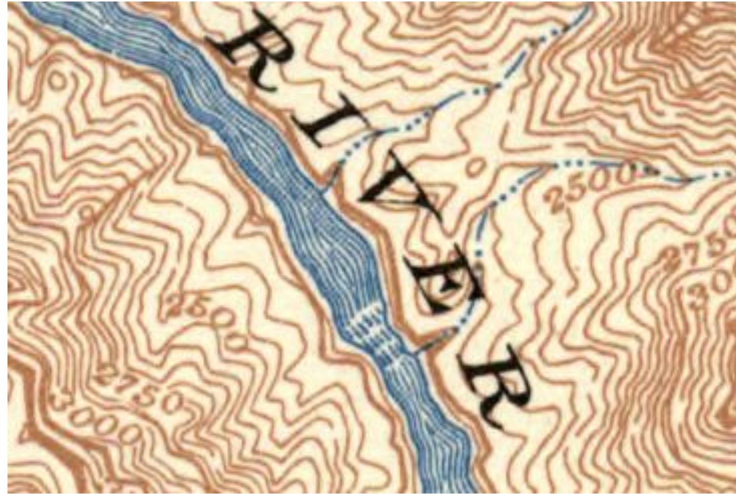
Now let’s imagine 2 dimensions, w and b

Visualizing the (negative) gradient vector at the red point

It has two dimensions shown in the x-y plane



Gradient Descent → Stochastic Gradient Descent



Key difference from our motivating scenario: in practice, calculating the exact gradient is really time-consuming.

So… we estimate the gradient using samples of data.

A brief aside: let’s talk about data

What does each instance of data is contribute?

Some of the nudges to a model’s parameters over the course of training.

Which data is used to train modern large language models?

Web text

… it’s kind of tough to give a more specific description than that.

See Dodge et al. EMNLP ‘21, “Documenting Large Webtext Corpora: A Case Study on the Colossal Clean Crawled Corpus”

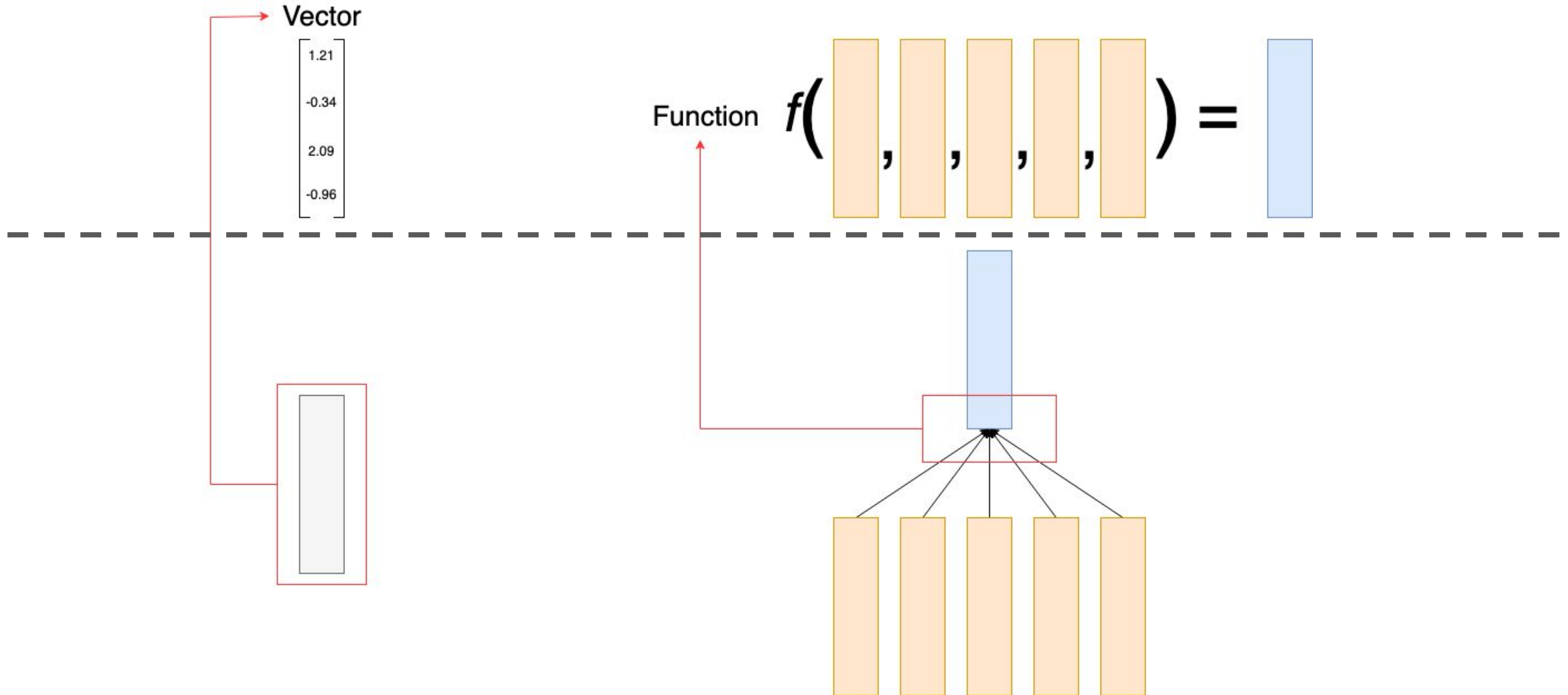
Also see Gururangan et al. EMNLP ‘22, “Whose Language Counts as High Quality? Measuring Language Ideologies in Text Data Selection”

✨✨ The Transformer ✨✨

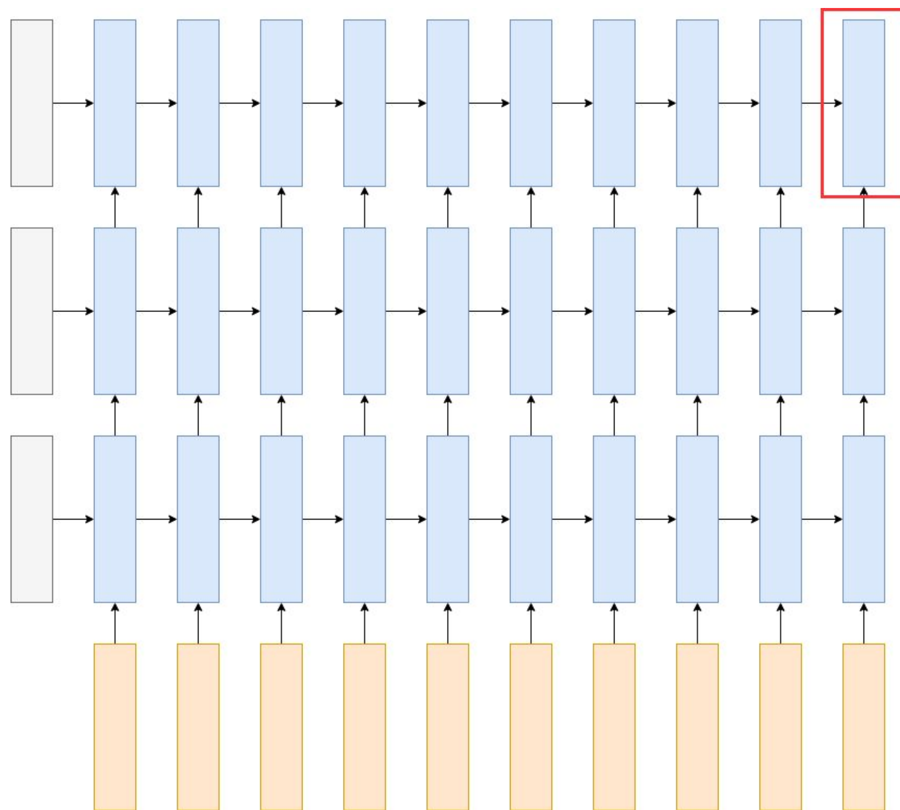
Why did the transformer make such a big difference for language modeling?

1. It allowed for faster learning of more model parameters on more data
2. It built in a method for contextualizing tokens with respect to other tokens in the sequence

A brief aside about some visual shorthand I'll be using

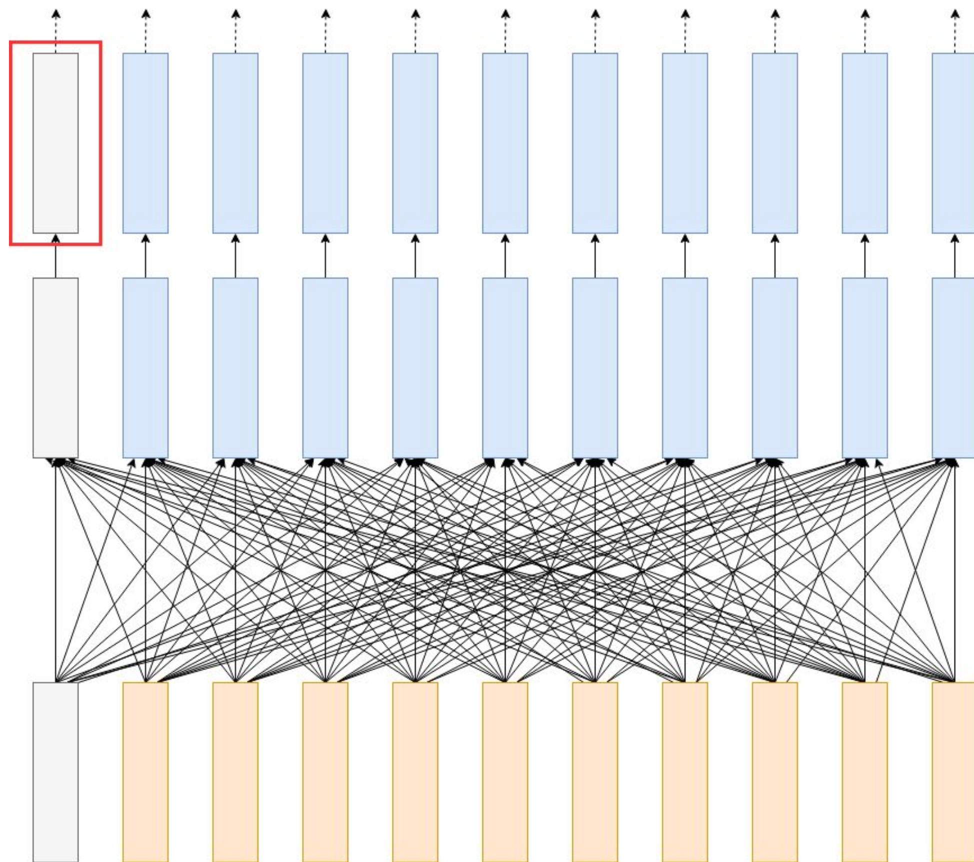


A 3-layer LSTM's calculations for an input of 10 tokens

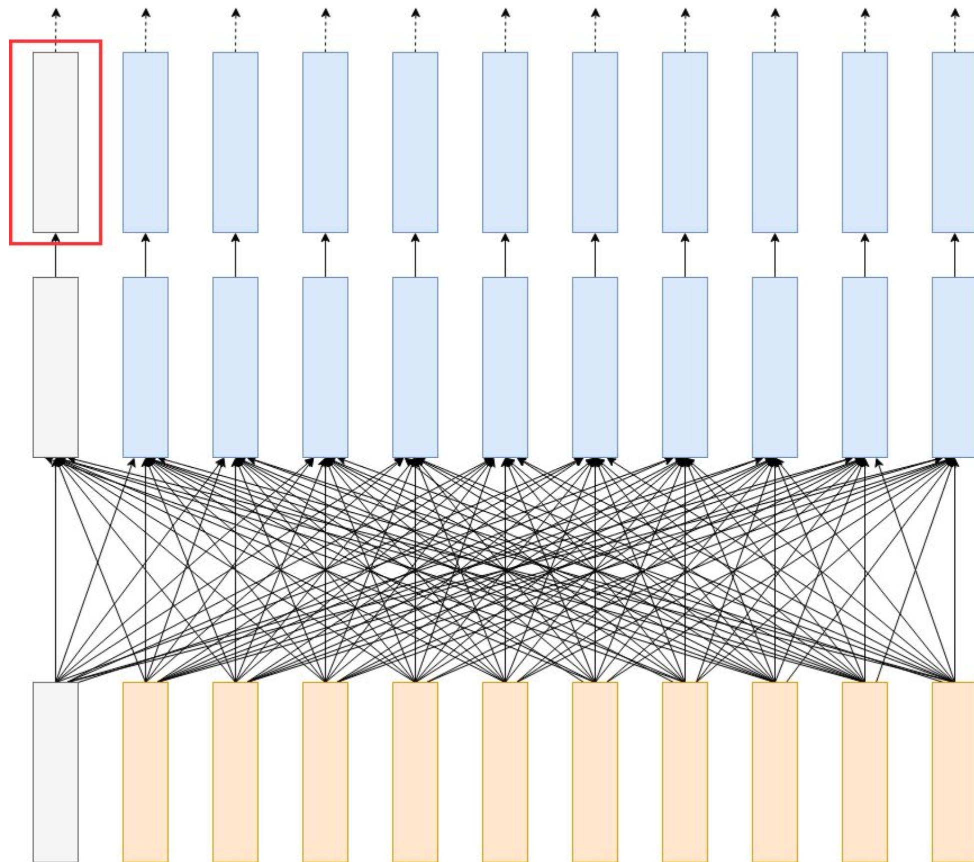


(For more on computing gradients via backpropagation, see colah's blog post on this topic)

One layer of the transformer architecture (Vaswani et al. 2017)

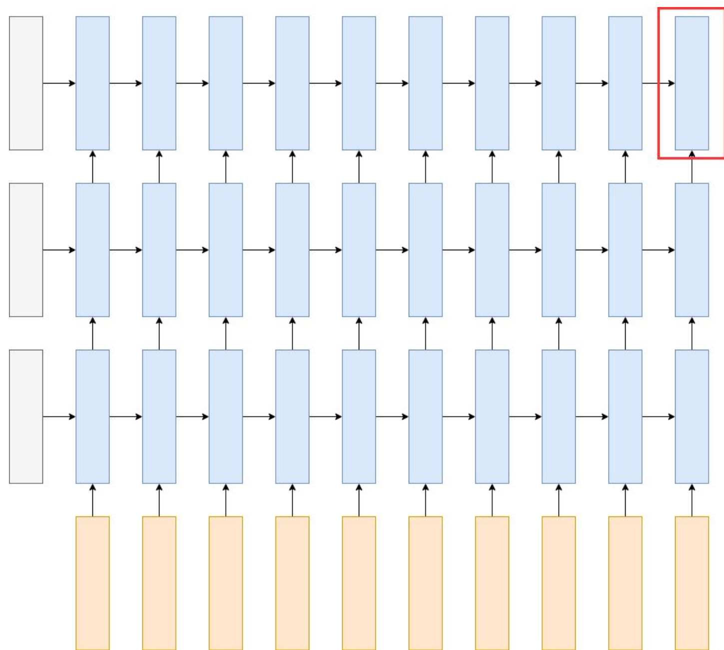


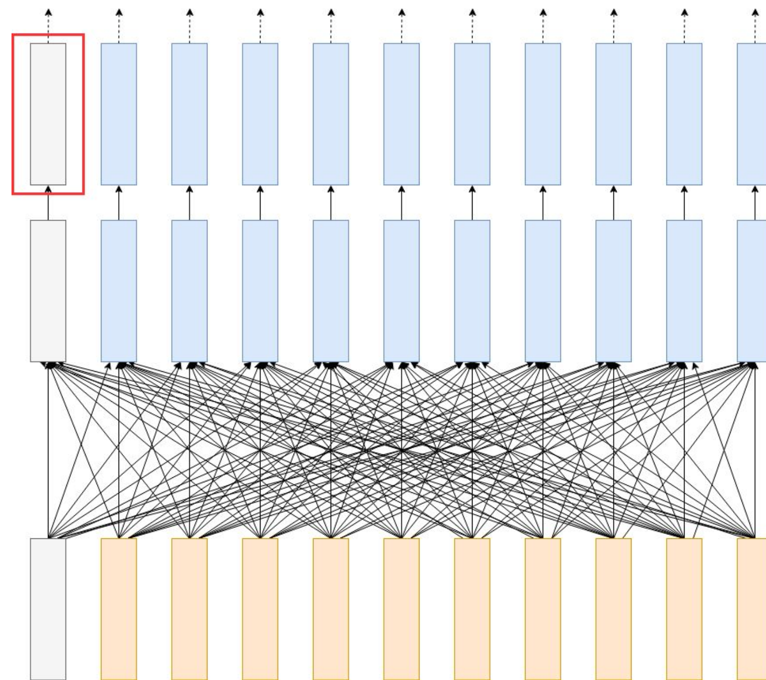
One layer of the transformer architecture (Vaswani et al. 2017)



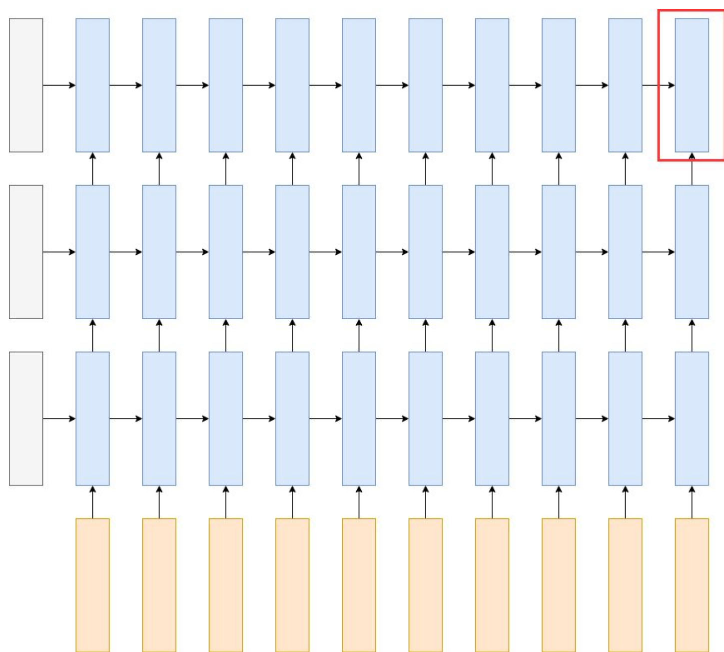
ok but how does this mess help anything sofia

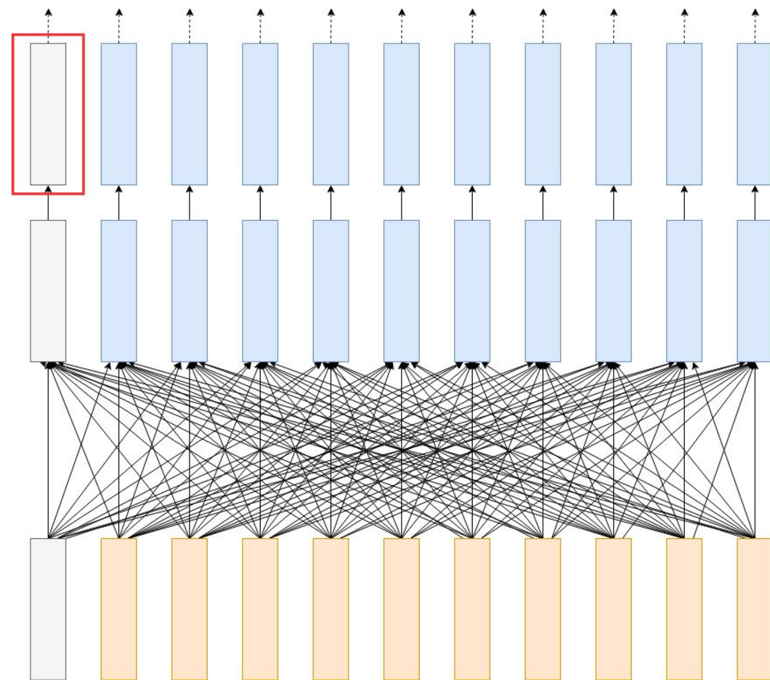
Comparing training times: how many functions do we need to backpropagate through?





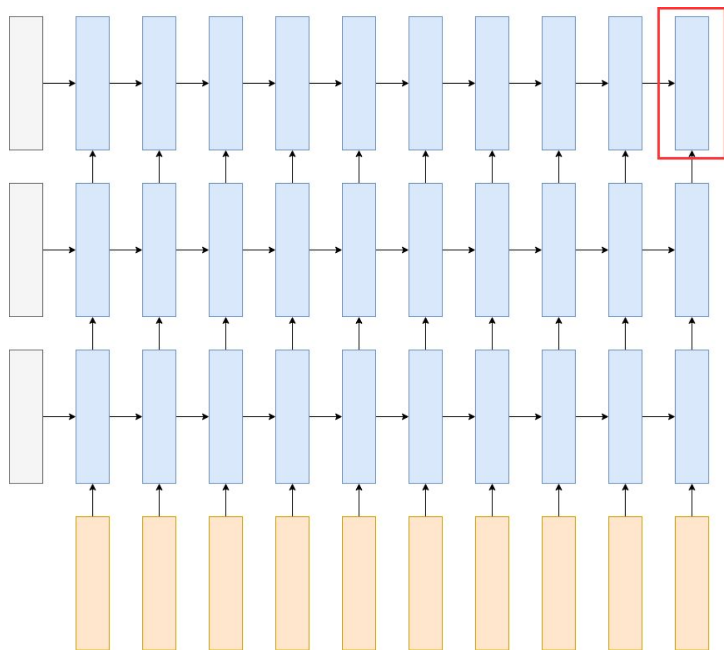
Comparing training times: how many functions do we need to backpropagate through?

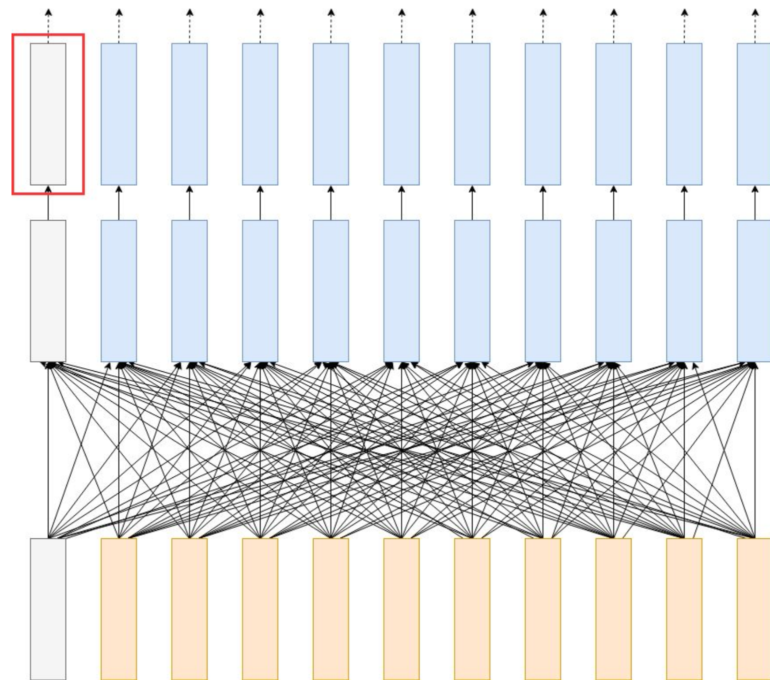




Transformers *parallelize* a lot of the computations that LSTMs make us do in sequence

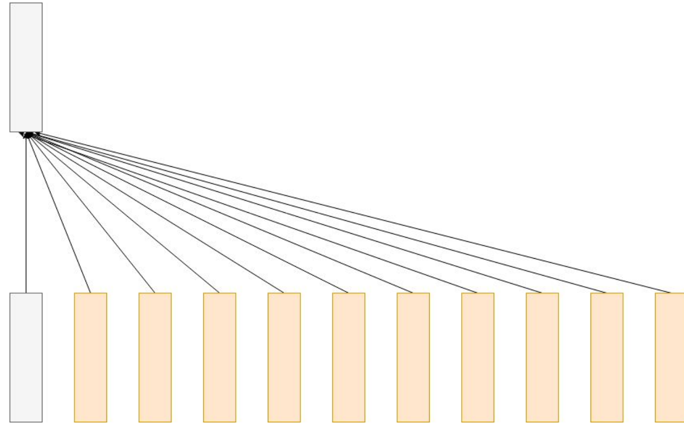
Comparing training times: how many functions do we need to backpropagate through?





Transformers *parallelize* a lot of the computations that LSTMs make us do in sequence
And (a very specific, but nonempty, subset of) you can therefore train a transformer on a ridiculously large amount of data in a way that you cannot for an LSTM.

What kind of function can take in a variable number of inputs like that without recursively applying an operation a bunch of times?



Attention mechanisms

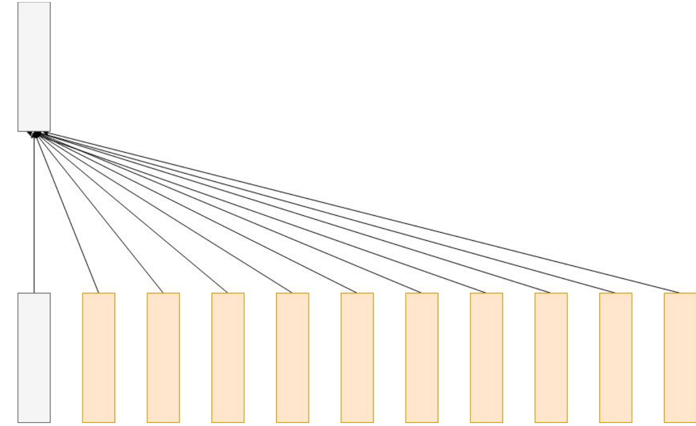
Building up to the attention mechanism

What about an average?

But we probably don't want to weight all input vectors equally…

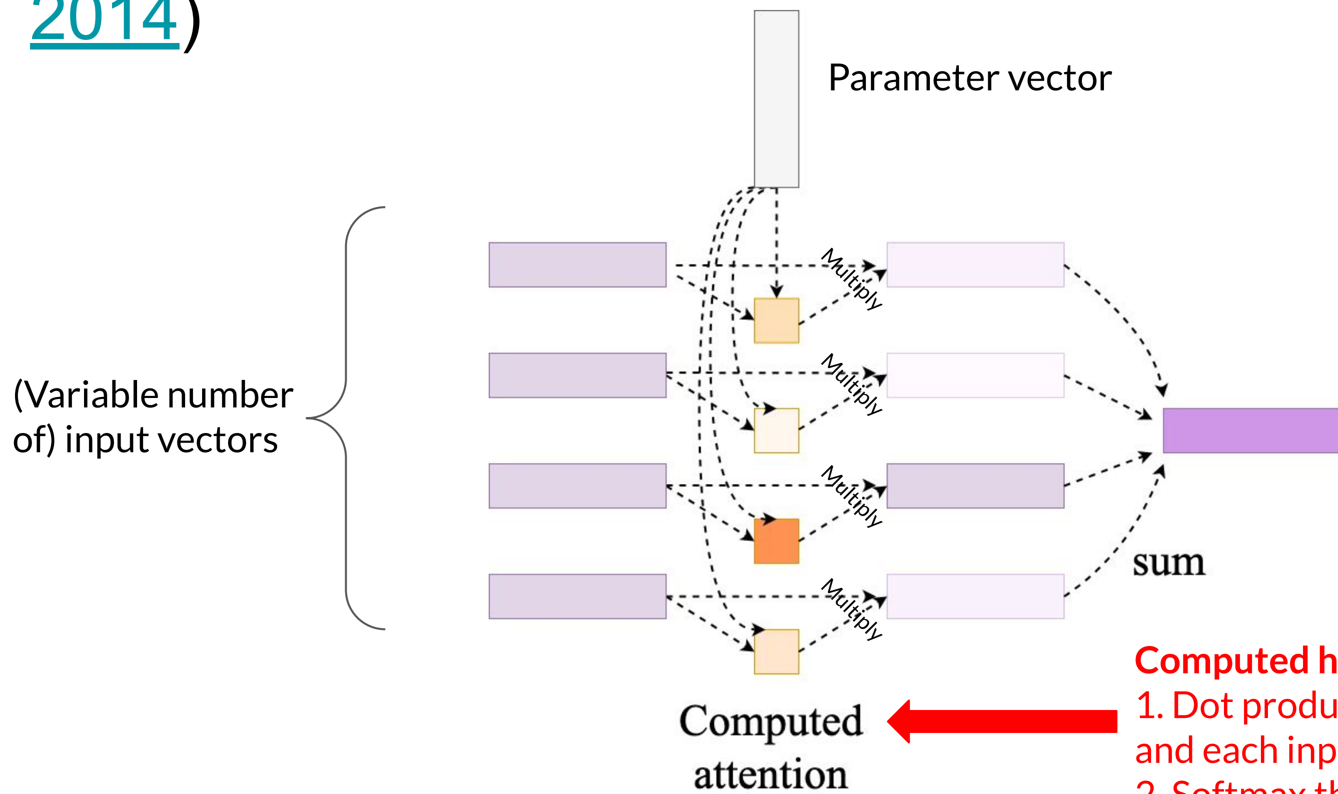
How about a weighted average?

Great idea! **How can we automatically decide the weights for a weighted average of the input vectors?**



What kind of function can take in a variable number of inputs like that without recursively applying an operation a bunch of times?

A simple form of attention (adapted from Bahdanau et al. 2014)

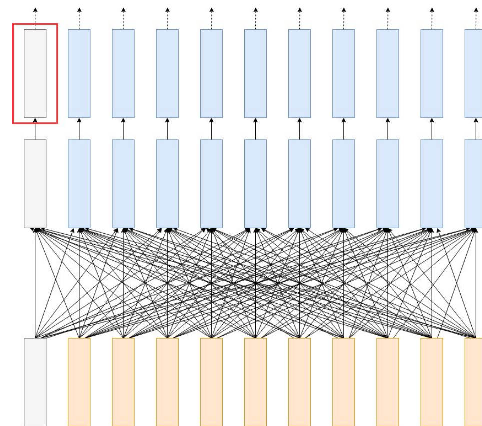


Computed how?
1. Dot product between param vector and each input vector
2. Softmax the set of resulting scalars.

Pros and cons

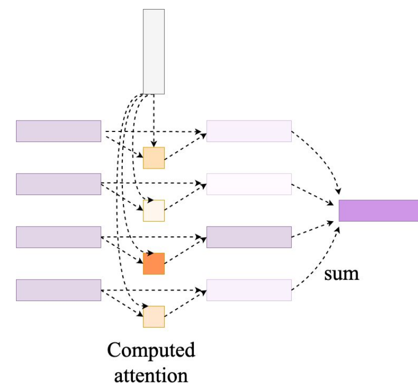
Pros:

- We have a function that can compute a weighted average (largely) in parallel of an arbitrary number of vectors!
- The parameters determining what makes it into our output representation are learned



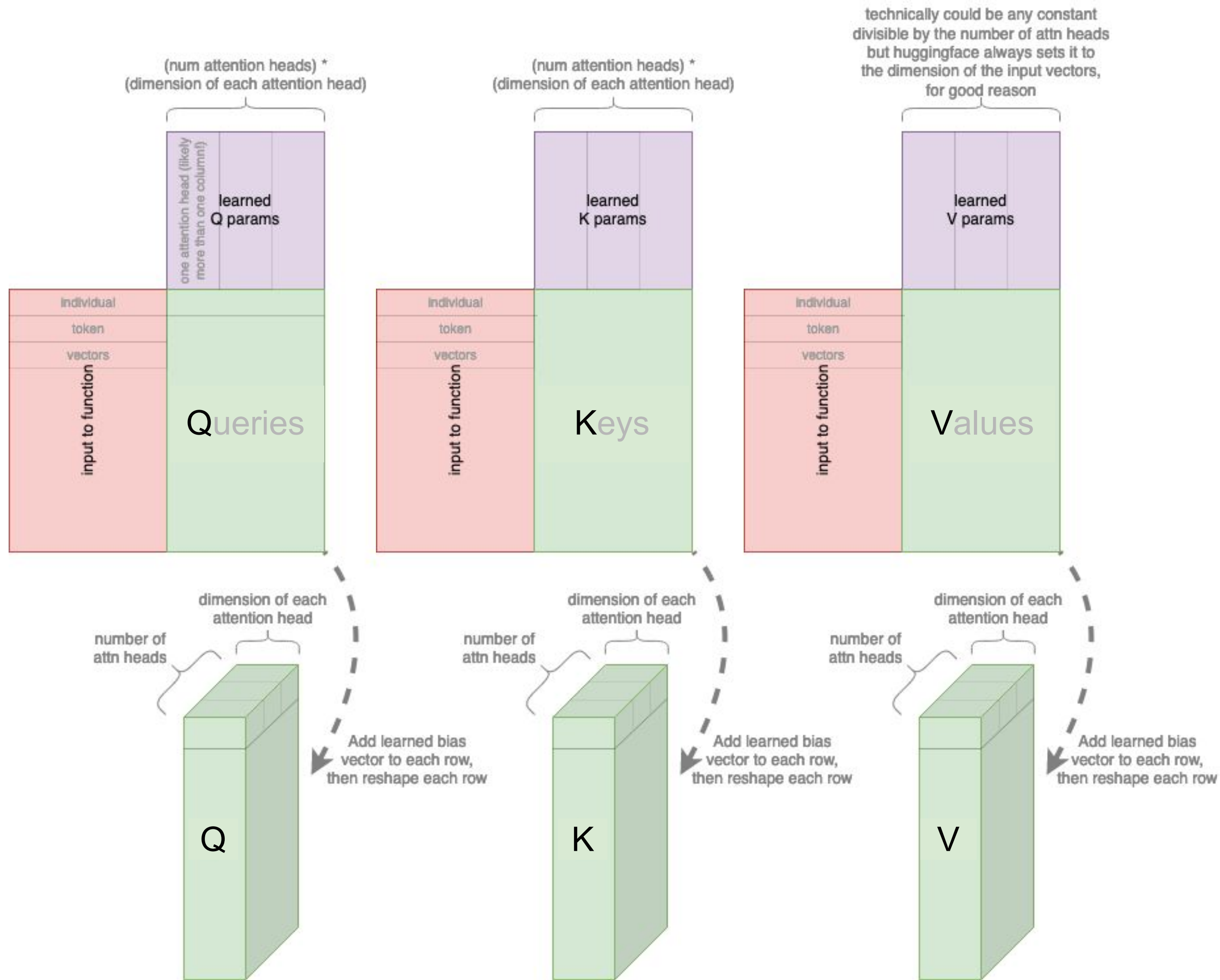
Cons:

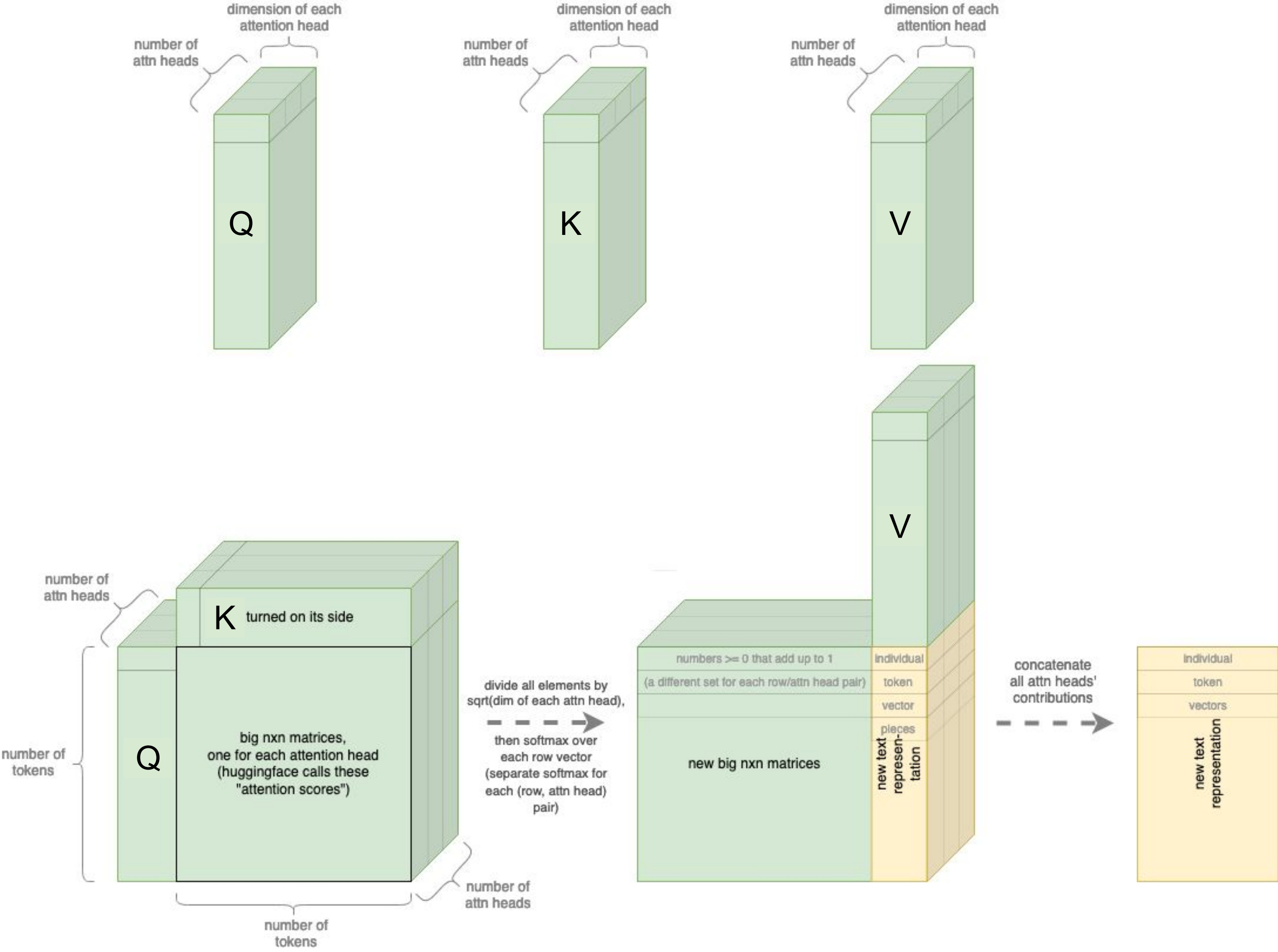
- We're also hoping to produce n different output token representations… and this just produces one…



Enter “self attention”

“What if instead of comparing each vector of the sequence to a single learned vector, we compared the sequence to *itself?”*


technically could be any constant divisible by the number of attn heads but huggingface always sets it to the dimension of the input vectors, for good reason
(num attention heads) * (dimension of each attention head)
(num attention heads) * (dimension of each attention head)
one attention head (likely more than one column!)
learned Q params
learned K params
learned V params
individual
token
vectors
input to function
Queries
Keys
Values
dimension of each attention head
number of attn heads
Add learned bias vector to each row, then reshape each row
Q
K
V

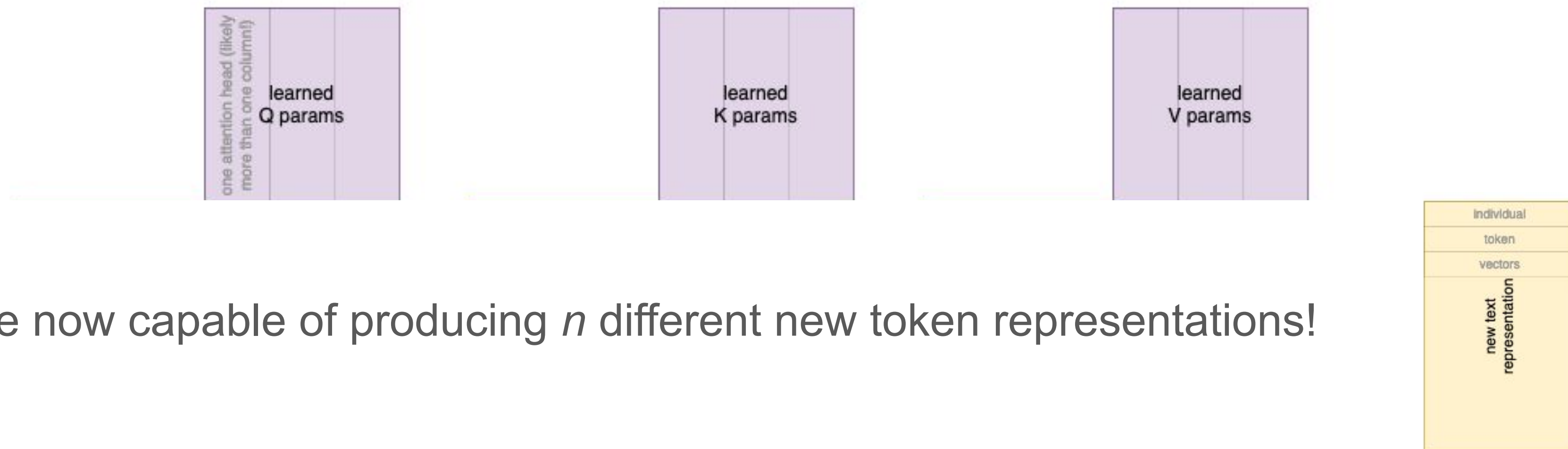


dimension of each attention head
number of attn heads
Q
dimension of each attention head
number of attn heads
K
dimension of each attention head
number of attn heads
V
V
number of attn heads
K
turned on its side
number of tokens
Q
big nxn matrices, one for each attention head (huggingface calls these "attention scores")
number of tokens
number of attn heads
divide all elements by sqrt(dim of each attn head),
then softmax over each row vector (separate softmax for each (row, attn head) pair)
numbers >= 0 that add up to 1
(a different set for each row/attn head pair)
new big nxn matrices
individual
token
vector
pieces
new text represen-tation
concatenate all attn heads' contributions
individual
token
vectors
new text representation

Hooray for self attention!

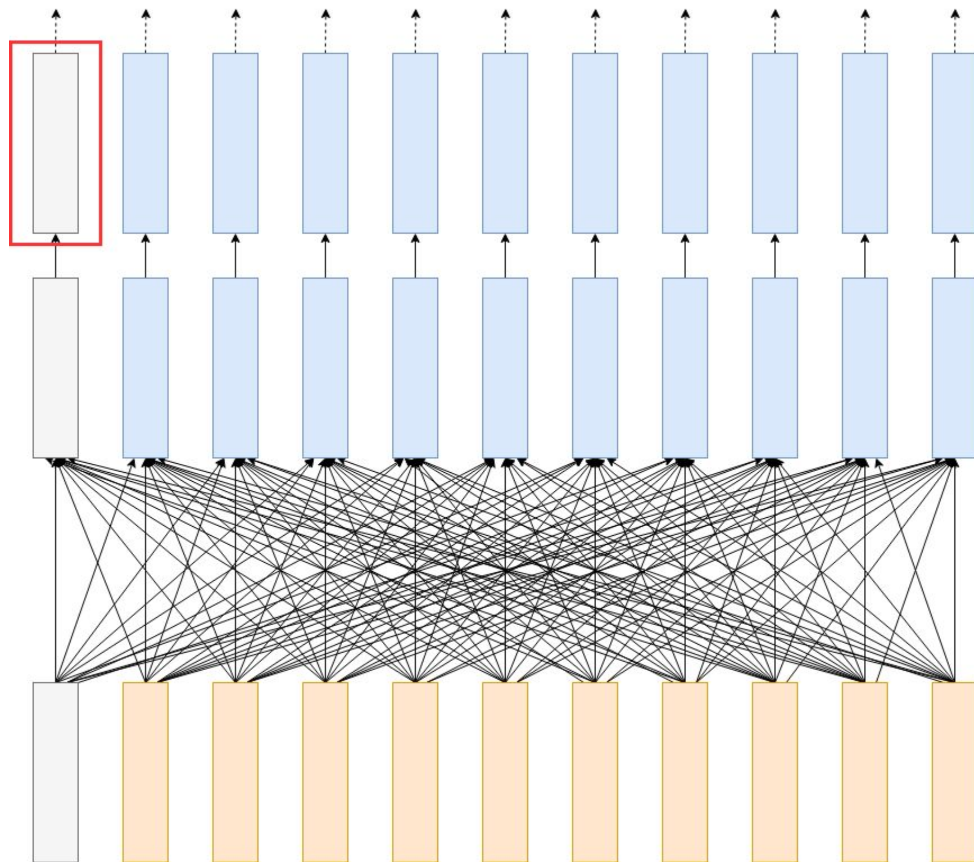
Our function is still made up almost entirely of matrix multiplications! *Which are very parallelizable* (→ efficient!)

We still learn fixed-size blocks of parameters that can be used for a sequence with an arbitrary length



We're now capable of producing *n* different new token representations!

Self attention is the key component of the transformer



That’s all I’ve got! Questions?