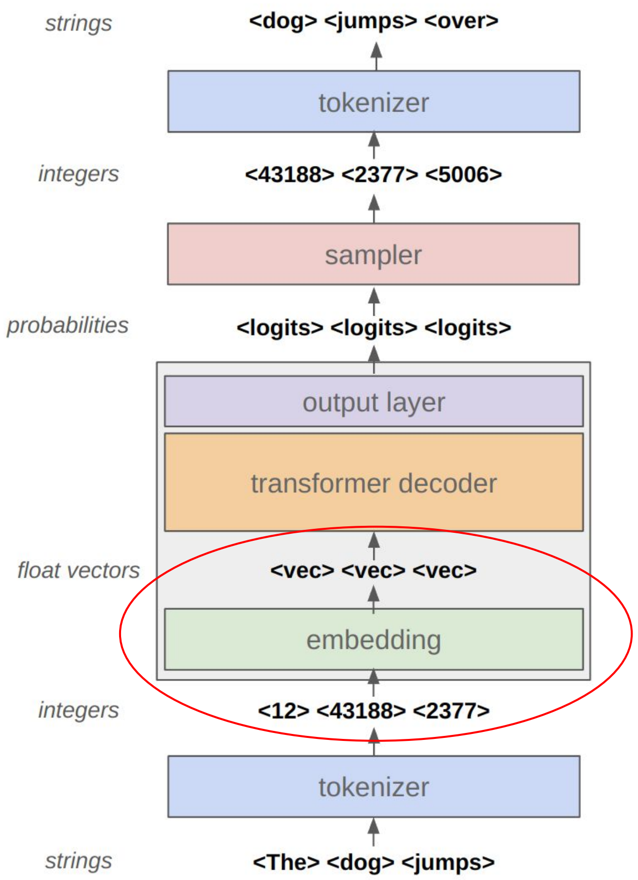
Embeddings

EN.705.743: ChatGPT from Scratch

Embeddings Intuition



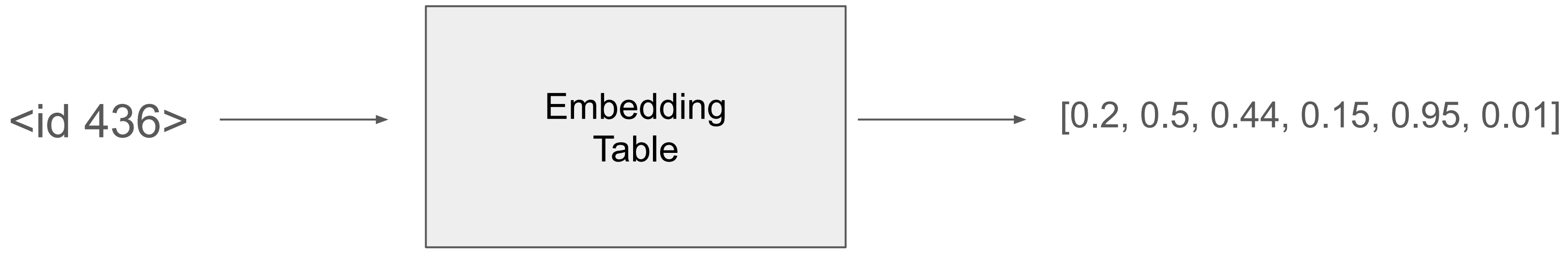
Embeddings

Last lecture, we saw how to convert text into integers so that we finally have numeric inputs for a model.

The first part of the model is an embedding, which does a second conversion: integers to floating point vectors.

Embeddings

In the general sense, an embedding is a vector representation of a specific label or id. An embedding associates a label with floating point features.



Thing with an integer id.

Representation of that thing, numerically.

Lookup-Table View

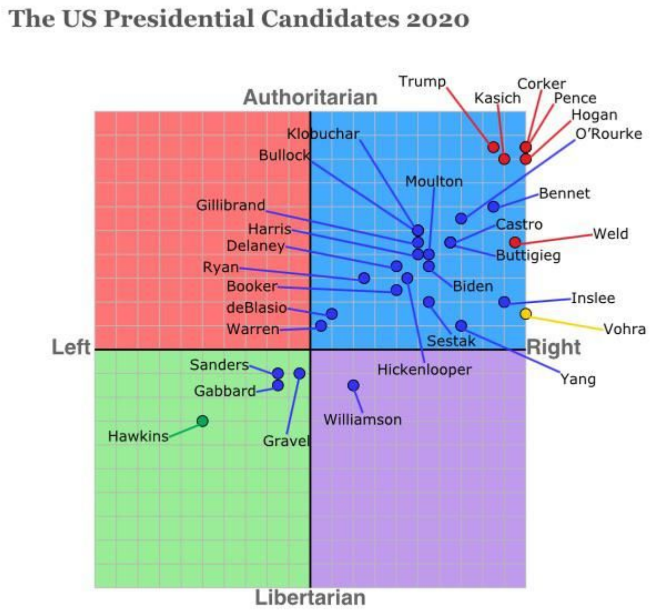
One way to think about an embedding is that it is a simple lookup table for vectors that represent various things:

ID	Vector
Bennet	[0.85, 0.6]
Biden	[0.54, 0.36]
Booker	[0.4, 0.25]
Buttigieg	[0.63, 0.46]
…	…

N-Dimensional Space View

Another way to think about this is that an embedding associates each id with a point in N-dimensional space.

ID	Vector
Bennet	[0.85, 0.6]
Biden	[0.54, 0.36]
Booker	[0.4, 0.25]
Buttigieg	[0.63, 0.46]
…	…



Word (Token) Embeddings

Embeddings are useful because they replace an id with a representation that we can do computation with.

In our case, we want to replace each <id> in our vocabulary with some floating point vector that represents that word.

$$\begin{tikzcd}
\text{The dog barks.} \arrow[d, bend left, "\text{tokenizer}"] \\
\text{<54> <259> <11013>} \arrow[d, bend left, "\text{embedding}"] \\
\text{[vector embedding 54] [vector embedding 259] [vector embedding 11013]}
\end{tikzcd}$$

What are the features?

Our embedding should somehow capture what the word means.

The embedding is N dimensional so it is hard to interpret, but some common properties are things like:

- Similar meanings group (“*dog”* is close to “*canine”* in our N dimensional space)
- Similar concepts group (“*dog”* is also close to “*cat”*)
- Co-occurrences may group (“*boat*” and “*ocean*”)
- Similar usages group (Common proper names might group)

Somehow the N dimensional vector encodes the meaning of the word and how it may be used.

Embedding Size

Our goal is to find, for each <id> in our vocabulary, a vector of length D that represents the given word (or subword).

Typically, D is chosen to be the same input/output dimension as our transformer model (next week’s subject). Typically values range from several hundred (512 or 768) to several thousand. GPT3 uses D=12,288.

These embeddings are very high dimensional -> hard to construct and hard to interpret. We need to learn them!

Embeddings can be a confusing term:

Size

Token Embeddings

Word Embeddings

These are embeddings of *items in a sequence.*
These are learned *upstream* of a model, to provide the model with features describing each sequence entry.

i.e. Inside OpenAI Language Models

Sentence Embeddings

…

Document Embeddings

These are embeddings of an *entire sequence*.
These are often the *output* of a model, used to concisely describe an entire block of text with one vector.

i.e. Output of OpenAI Embedding Models

Word Embeddings, ~10 Years Ago

Word2Vec (Google, 2013) (Also in this week’s reading)

One option is to learn the embeddings with a separate model. Word2Vec is an approach that is based on the idea: **A word should be related to its neighboring words in context.**

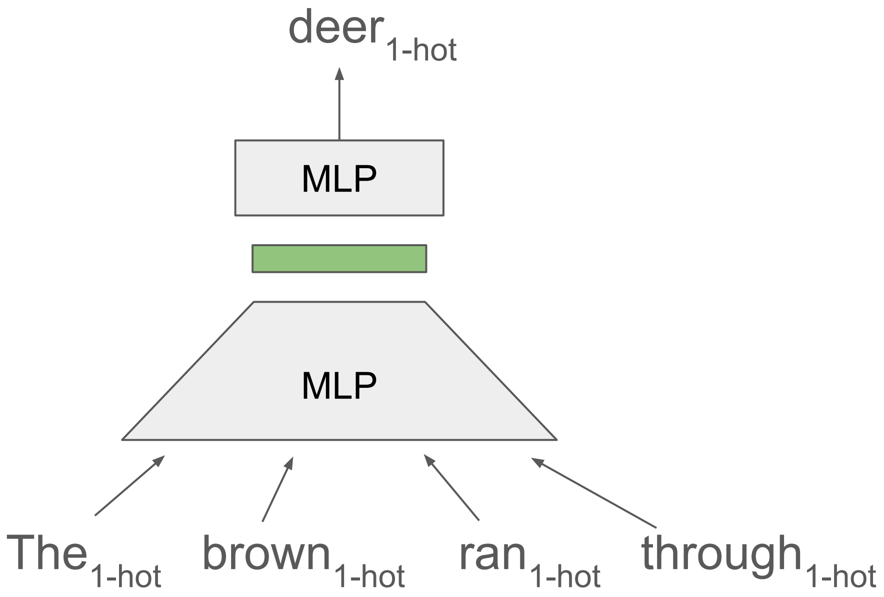
For example, if we have some snippet of text:

The brown deer ran through the meadow.

We have some loose associations between (deer, brown), (deer, meadow), (deer, ran) etc. Over many many examples, these associations should average out into some understanding of the word (“deer”).

Word2Vec Architecture

We can build a model which, given surrounding context, predicts a missing word:



We simply input/output words with a one-hot encoding of their class id:

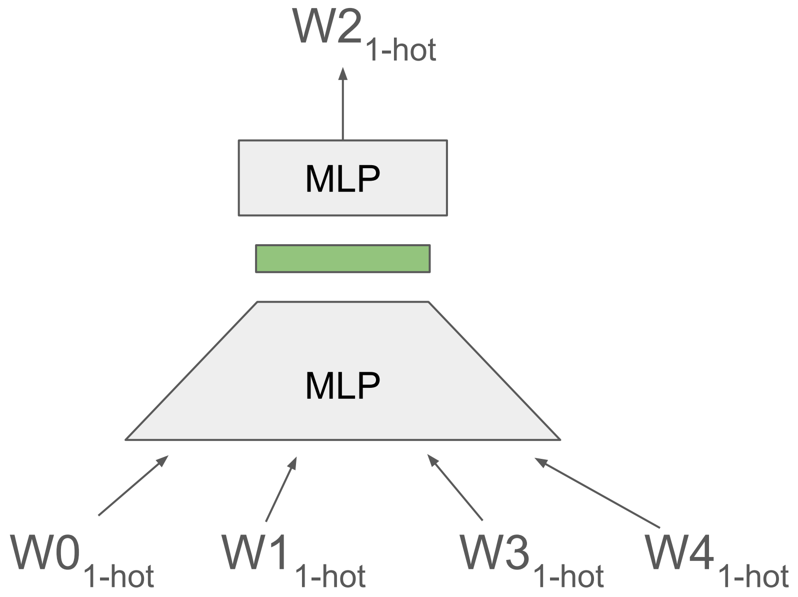
<id=4> becomes [0 0 0 0 **1** 0 0 …]

The inner representation is taken as the encoding of the missing word.

This is very similar to an autoencoder.

Word2Vec Architecture, CBOW Method

We can run this on every sequence of N words in our dataset (here N=5).



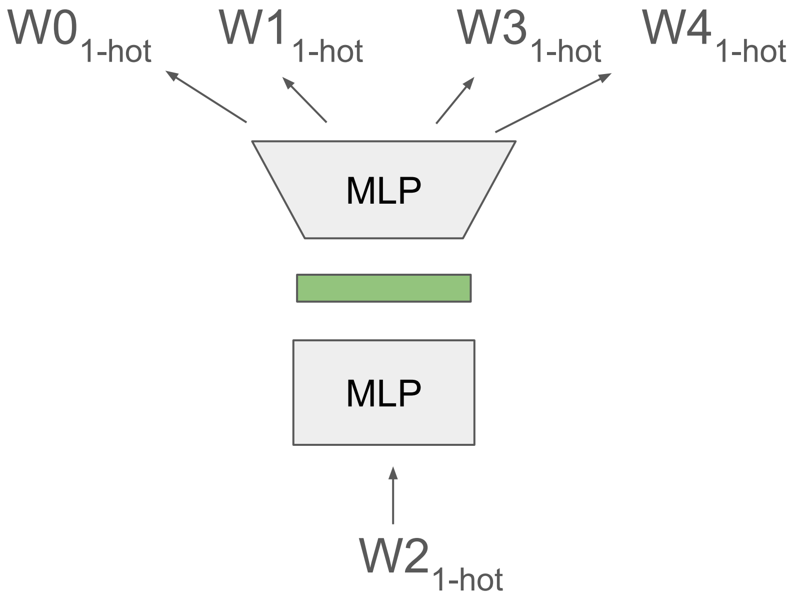
This is called the **“continuous bag-of-words”** method, since we can slide a window of size N over our data, continuously capturing N words.

Question: How do we use this after training?

It’s not that easy…

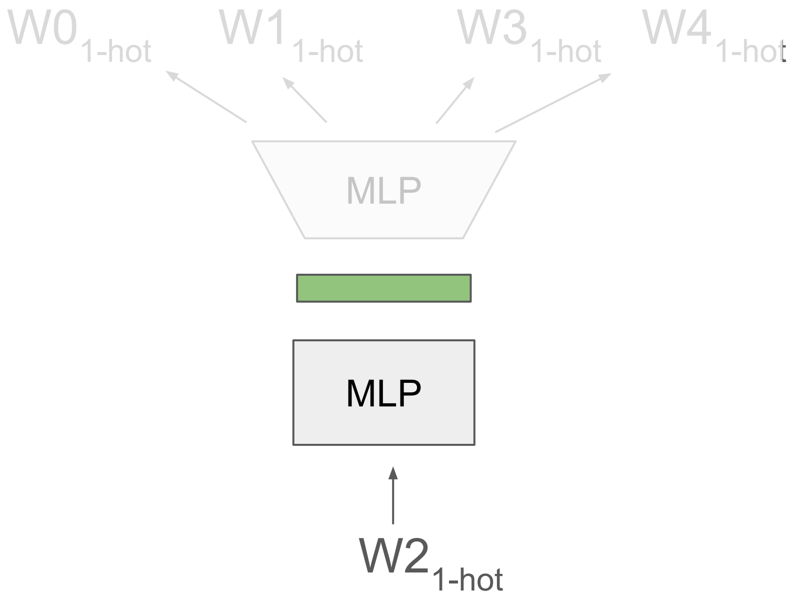
Word2Vec Architecture, Skip-Ngram Method

To make this easier to use, we can actually flip it around:



Word2Vec Architecture, Skip-Ngram Method

To make this easier to use, we can actually flip it around:



This formulation is easier to use after training.

We can input a word and get the associated embedding by using only the first half of the model.

GloVe (**Glo**bal **Ve**ctors) (Stanford, 2014)

GloVe is another method you may hear about. Word2Vec computes associations based on local context windows (i.e. group of 5 words).

GloVe instead considers global co-occurrences. i.e., For some word W in the vocabulary, if should be related to the N words that it typically co-occurs with **across the entire dataset.**

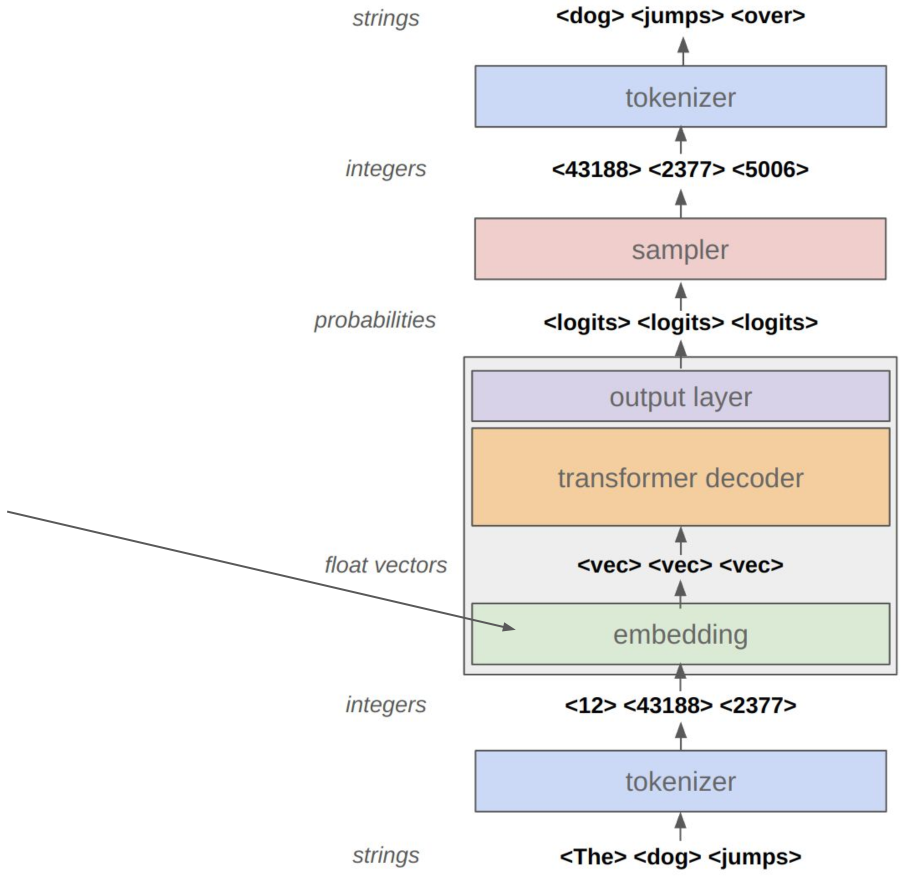
GloVe and Word2Vec are popular methods for learning a **separate embedding model.**

Word Embeddings (Now)

Modern Approach

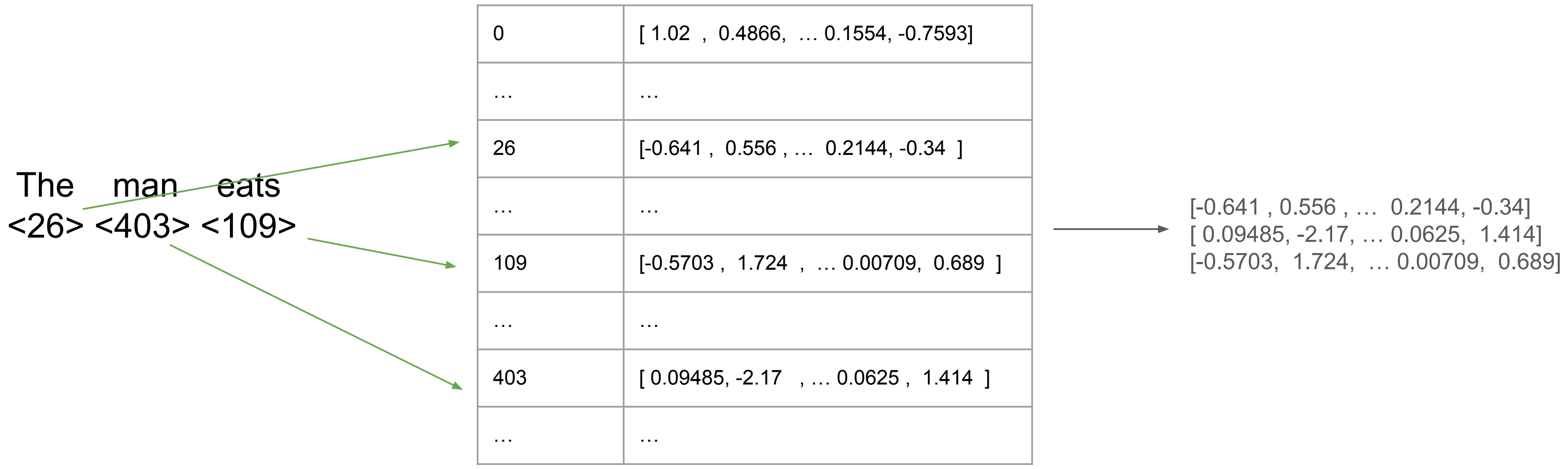
We can also have embeddings learned as a part of the overall optimization of our model.

In the age of transformers, we have an "embedding layer" that we can directly backpropagate through.



Embedding Layer

Main Idea: We initialize our embedding table randomly. When we train our model, we replace <ids> with the associated vectors, even if they are initially meaningless:

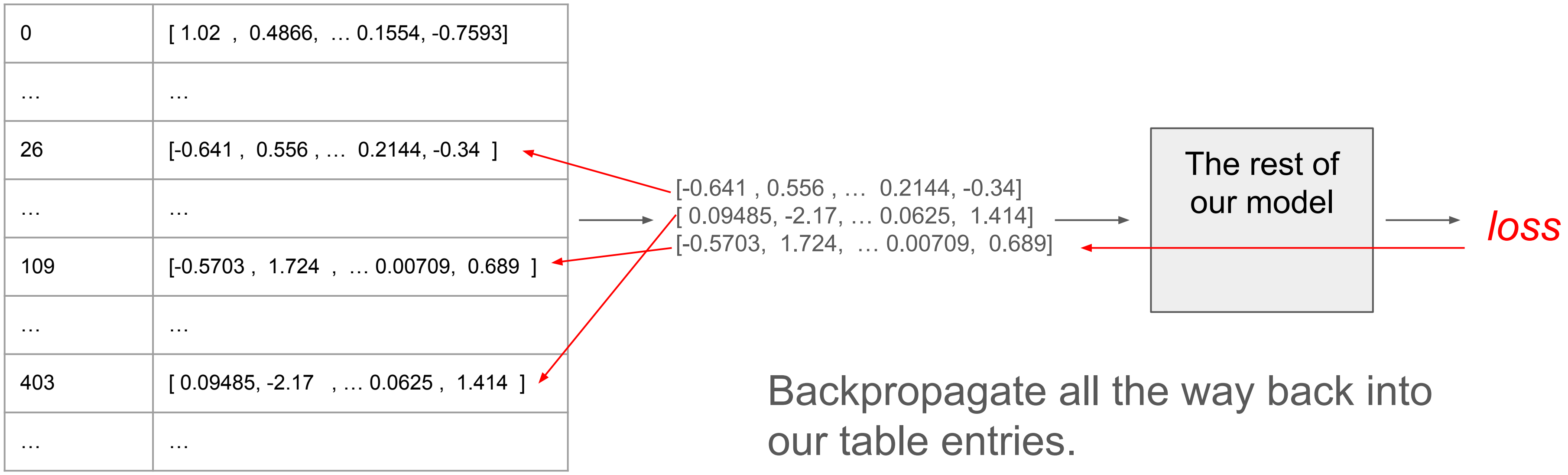


0	[1.02 , 0.4866, … 0.1554, -0.7593]
…	…
26	[-0.641 , 0.556 , … 0.2144, -0.34]
…	…
109	[-0.5703 , 1.724 , … 0.00709, 0.689]
…	…
403	[0.09485, -2.17 , … 0.0625 , 1.414]
…	…

[-0.641 , 0.556 , … 0.2144, -0.34]
[0.09485, -2.17, … 0.0625, 1.414]
[-0.5703, 1.724, … 0.00709, 0.689]

Embedding Layer

We feed these vectors into the rest of the model as “input”, and we backpropagate back into the vectors during training.



0	[1.02 , 0.4866, … 0.1554, -0.7593]
…	…
26	[-0.641 , 0.556 , … 0.2144, -0.34]
…	…
109	[-0.5703 , 1.724 , … 0.00709, 0.689]
…	…
403	[0.09485, -2.17 , … 0.0625 , 1.414]
…	…

Backpropagate all the way back into our table entries.

Embedding Layer Backprop

Parallel to MLP: Recall that for a fully connected layer, we have:

$$y = Wx + b$$

And part of updating W is to compute how y changes w.r.t W:

$$dy/dW = x$$

However, we could also compute an update for our input x:

$$dy/dx = W$$

This is exactly what we are doing for our embeddings. We can feed them into the model and compute an update for them. This is also used to pass a gradient to a prior layer from the current layer.

Embedding Layer Notes

A few notes:

1) The entries in our embedding table are also the parameters being updated, so sometimes these are called the “parameters” or “weights” of the embedding layer.

2) Unlike other NN layers, the embedding layer is <u>outputting</u> these parameters. The outputs are subsets of the weights.

3) The good news is, if you tell pytorch that the weight tensor requires a gradient (instantiate it as a Parameter), it should pretty much figure all this out for you.

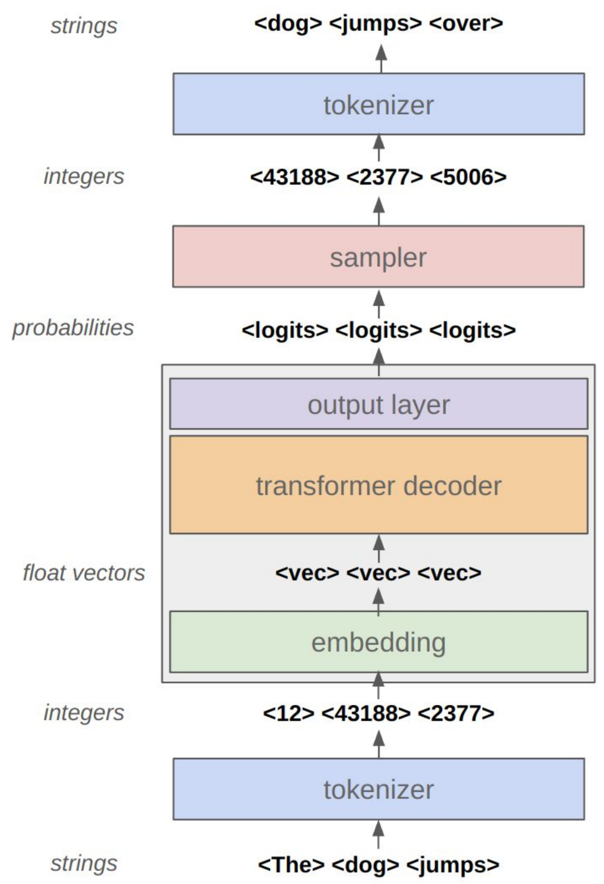
Applying this to our model

So, when building our GPT model, we initialize an embedding layer of size N by D, where N is the size of our vocabulary, and D is the length of the vector representation of each token.

Tokenizer has a pre-built vocabulary of size N.

The rest of our model expects vectors of length D.

So the embedding learns NxD values. (One vector of length D for each of N tokens).



Side note on size

Good to think about the size of the embedding.

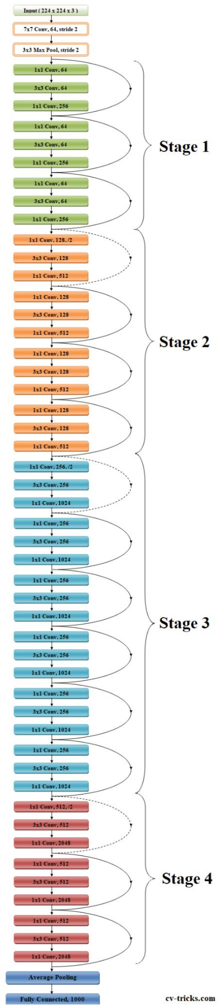
Let’s say our vocabulary is size N=~50,000

And we want to represent each word with a vector of length ~1000.

This is (50,000)*(1000) = 50 million parameters!

For a model like GPT3, which has D=12,288, the embedding is ~600 million parameters.

For comparison, ResNet-50 has 25 million parameters (architecture diagram on right).



Recap for Homework

An embedding layer has a parameter tensor that is size N x D.

The inputs to the layer are integers (usually a 2D tensor of integers with shape (batch, sequence_length).

We use these to index entries in the parameter tensor, and we use these retrievals directly as our layer’s output.

Since our outputs are also Parameters (require a gradient), pytorch will backprop into them and update them.

Position Embeddings

Two Embeddings for LLMs

Transformers (next week) technically perform “set” operations, not “sequence” operations. That is, they do not understand ordering. This is critical in language!

The lion ate the hyena and the gazelle. $\overset{?}{=}$ *The lion and the hyena ate the gazelle.* $\overset{?}{=}$ *The gazelle ate the hyena and the lion.*

Without intervention, a transformer treats all of these identically.

Two Embeddings for LLMs

To fix this, we embed each token **and its position in the sequence.**

Text:	The	lion	ate	the	gazelle …
Token:	<32>	<5364>	<27721>	<32>	<2309>...
Position:	<0>	<1>	<2>	<3>	<4>...

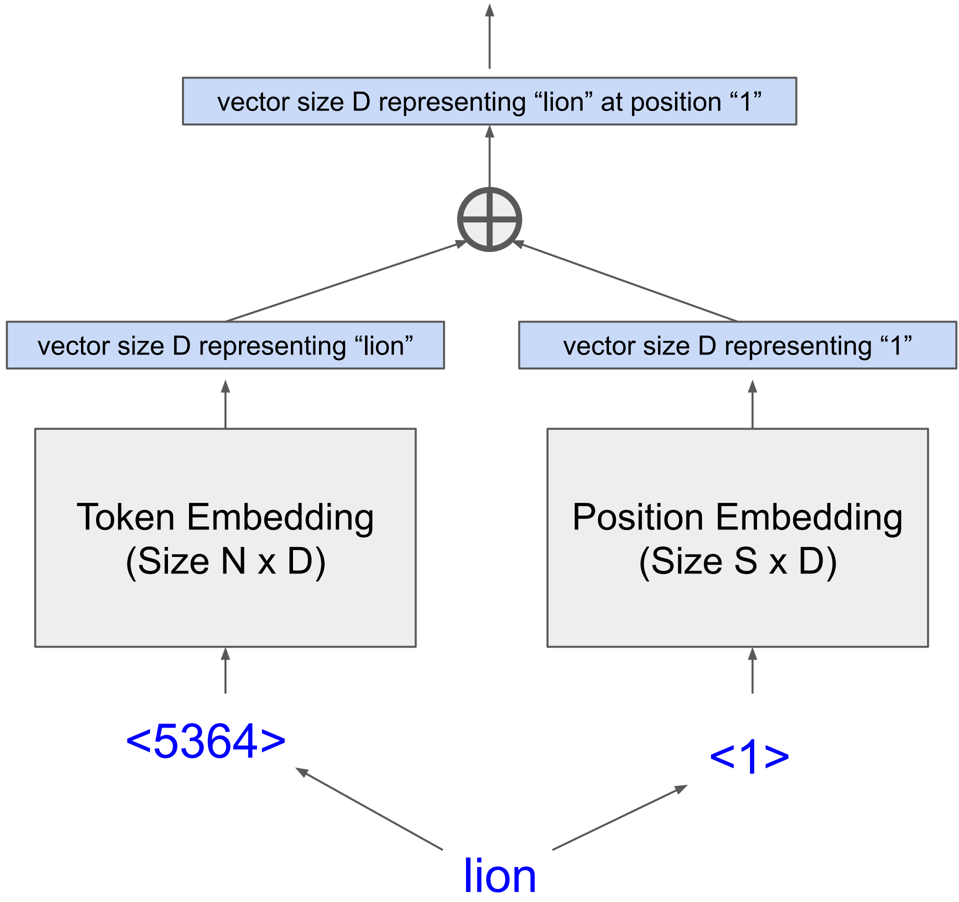
Our embedding module will have two embeddings, one to convert token ids to vectors, and one to convert position ids to vectors.

Two Embeddings for LLMs

Text:	The	**lion**	ate	the	gazelle …
Token:	<32>	**<5364>**	<27721>	<32>	<2309>...
Position:	<0>	**<1>**	<2>	<3>	<4>...

Our position embedding has size S x D, where S is the maximum sequence length that we will encounter (the highest possible position).

We simply add the embeddings together to get our final embedding, which describes a token at a specific position.


vector size D representing "lion" at position "1"
vector size D representing "lion"
vector size D representing "1"
Token Embedding (Size N x D)
Position Embedding (Size S x D)
<5364>
<1>
lion

Position Embeddings

The position embedding can seem to be a strange mechanism at first. Note that we are using D floating point values to represent a single integer.

In the word embedding, an integer input has lots of meaning (it corresponds to a specific word which has meaning, nuance, connotations, context).

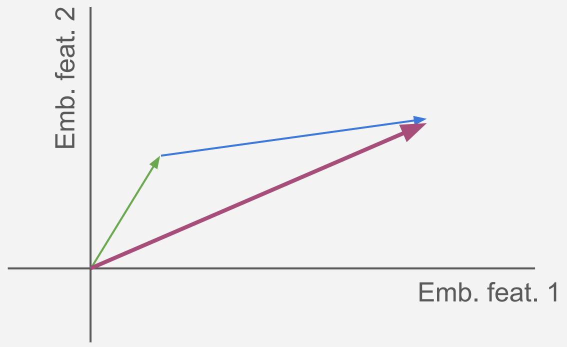
In our position embedding, the integer input really is just an integer. The embedding of “23” just means “position 23”.

However, the position embedding needs to modify an entire token embedding, hence the larger size (next slide).

Note: Position Embeddings as an “Offset”

If the word embeddings learn the meaning of each token, the positions embeddings learn an offset which will shift the representation of the token such that it now encodes position as well. Imagine just 2 features (D=2):

Lion <id 254>:	[0.1, 0.3]	(from word embedding)
+ Position 4:	[0.4, 0.1]	(from position embedding)
=	**[0.5, 0.4]**	(final embedding of word)



Summary / Homework

By far, the easiest thing to do (in terms of the complexity of our model), is to just use the exact same mechanism as word embeddings for positions embeddings.

One learns N vectors of size D, to represent tokens.

The other learns S vectors of size D, to represent position.

Functionally they work exactly the same way. This is a very common approach and it works pretty well. For our homeworks, this is what we will do (for this week, you will implement a single Embedding class, and later we will use it twice).

History and Future

There are a few more complex variants which are worth knowing about for breadth:

Sinusoidal Embeddings (Historical-ish)

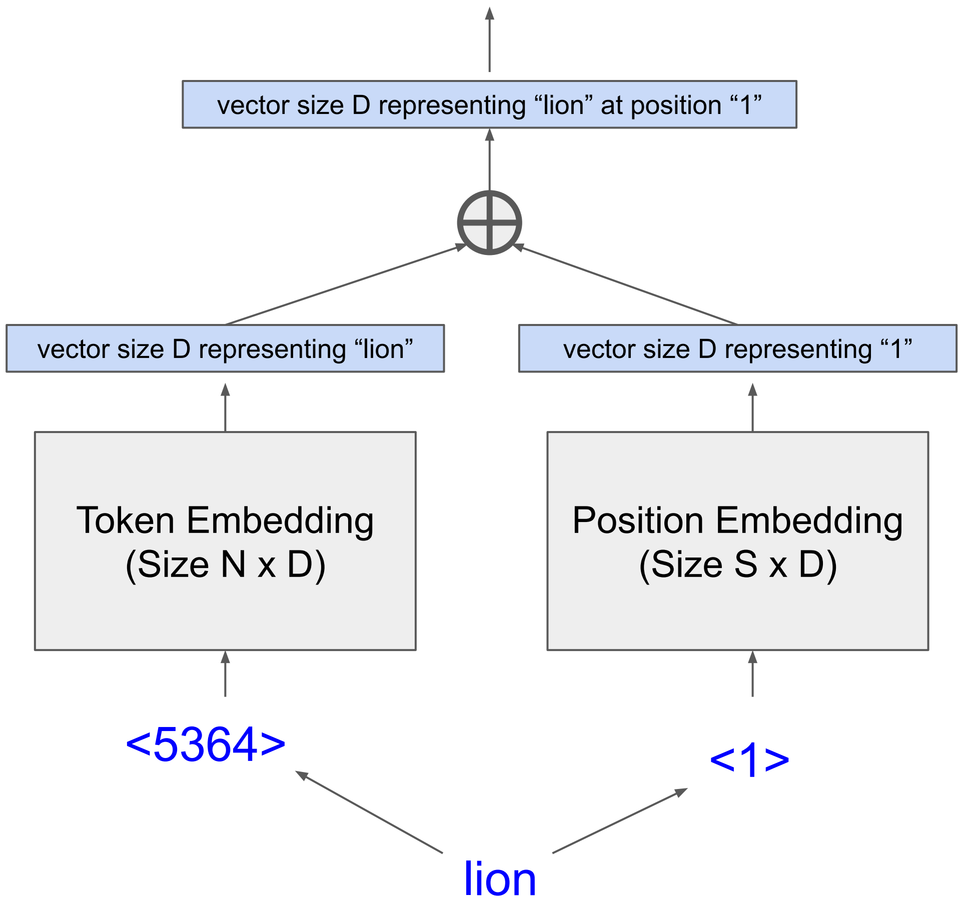
Rotary Position Embeddings (Modern)

Sinusoidal Position Embeddings

Sinusoidal Embeddings

It seems wasteful to learn S*D values just to represent integers (this may be millions of parameters).

The original transformers paper thought so too, so they had a different approach: Sinusoidal Embeddings

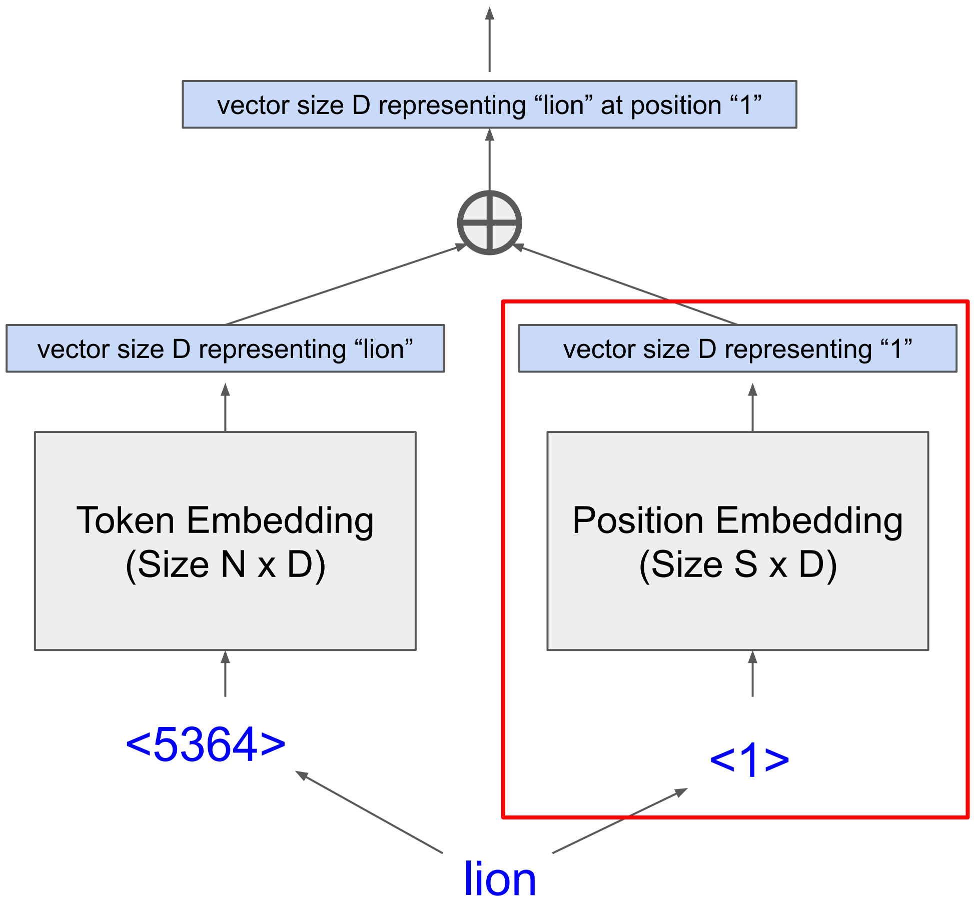

vector size D representing “lion” at position “1”
vector size D representing “lion”
vector size D representing “1”
Token Embedding (Size N x D)
Position Embedding (Size S x D)
<5364>
<1>
lion

Sinusoidal Embeddings

Revisit the objective of the position embedding module.

We want to express a single integer as D floating-point values.

Could we just construct some set of S vectors of length D, such that each has a unique pattern?



vector size D representing "lion" at position "1"
vector size D representing "lion"
vector size D representing "1"
Token Embedding (Size N x D)
Position Embedding (Size S x D)
<5364>
<1>
lion

Sinusoidal Embeddings

Rather than learning vectors, construct S vectors of length D which each having a unique pattern. The model will learn to use these on its own.

Specifically:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where *pos* is the position of the token, and *i* is a floating point value from 0 to $D=d_{\text{model}}$.

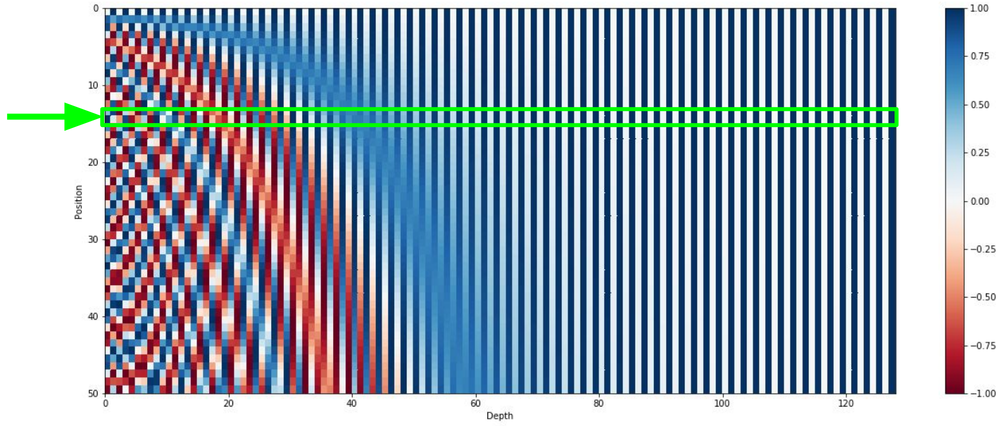
Sinusoidal Embeddings

Rather than learning vectors, construct S vectors of length D which each having a unique pattern. The model will learn to use these on its own.

Specifically:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where *pos* is the position of the token, and *i* is a floating point value from 0 to D=d_{model}.



For each word, grab row at position of word.

Figure: Amirhossein Kazemnejad

Comparison

Sinusoidal embeddings are pre-computed, so the model has less to learn. On the other hand, you have to compute them!

In practice, both methods work equally well. It’s just a matter of preference.

Most practitioners seem to prefer the simplicity and flexibility of two learned embeddings.

Rotary Position Embeddings (“RoPE”)

Current Practice

Very modern LLMs (last 1-2 years) use an improvement on these techniques, called Rotary Position Embeddings (RoPE).

A key insight is that a token’s meaning is not derived from its absolute position, but from the **relative position of other tokens.** This is significant in documents hundreds or thousands of tokens long.

The	**lion**	ate	the	gazelle
0	**1**	2	3	4

$\longrightarrow$

The	**lion**	ate	the	gazelle
-1	**0**	+1	+2	+3

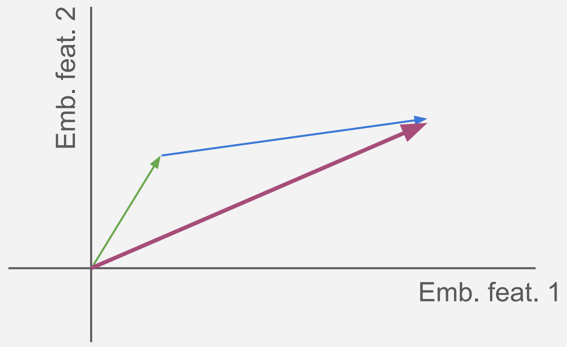
Note: A good explanation of RoPE: https://www.youtube.com/watch?v=o29P0Kpobz0

Current Practice

Instead of offsetting embeddings with an addition (absolute embedding), RoPE offsets embeddings by an angle θ. Consider D=2, each word embedded by 2 features:

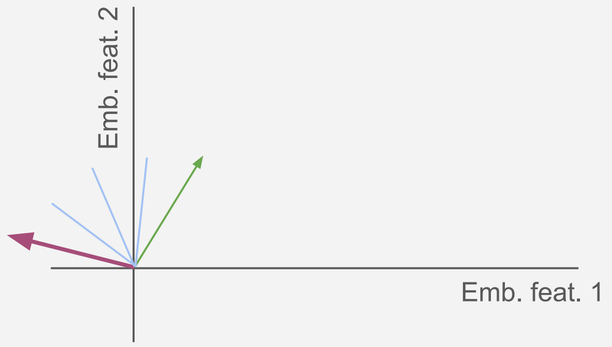
Absolute

Lion <id 254>:	[0.1, 0.3]	(from word embedding)
+ Position 4:	[0.4, 0.1]	(from position embedding)
=	**[0.5, 0.4]**	(final embedding of word)



Rotary

Lion <id 254>:	[0.1, 0.3]	(from word embedding)
Rotated by 4*θ		
=	**[-0.31, 0.05]**	(final embedding of word)

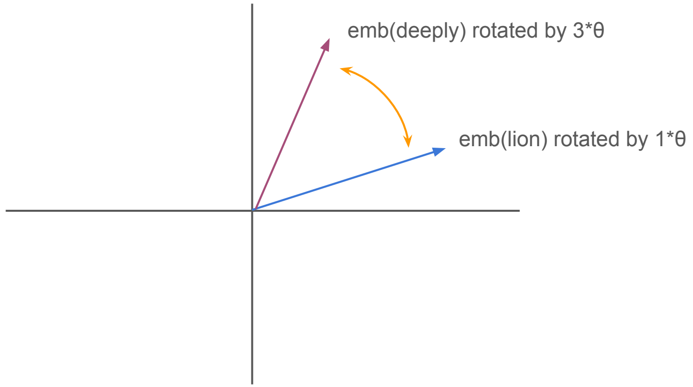


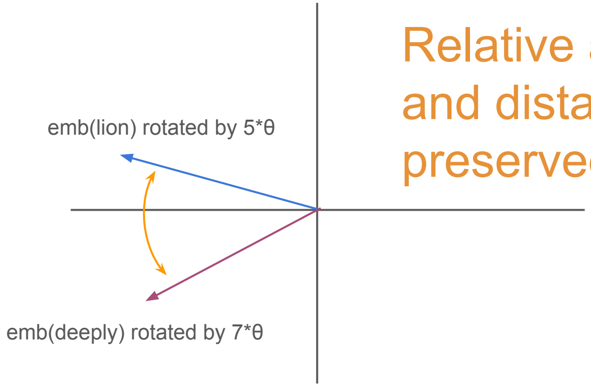
Why is this a good idea?

Rotating by [position]*[theta] preserves the relative angle and distance between two words:

0 1 2 3
The lion slept deeply.

0 1 2 3 4 5 6 7
After a long day, the lion slept deeply.





Relative angle and distance is preserved.

Applying to D>2 features.

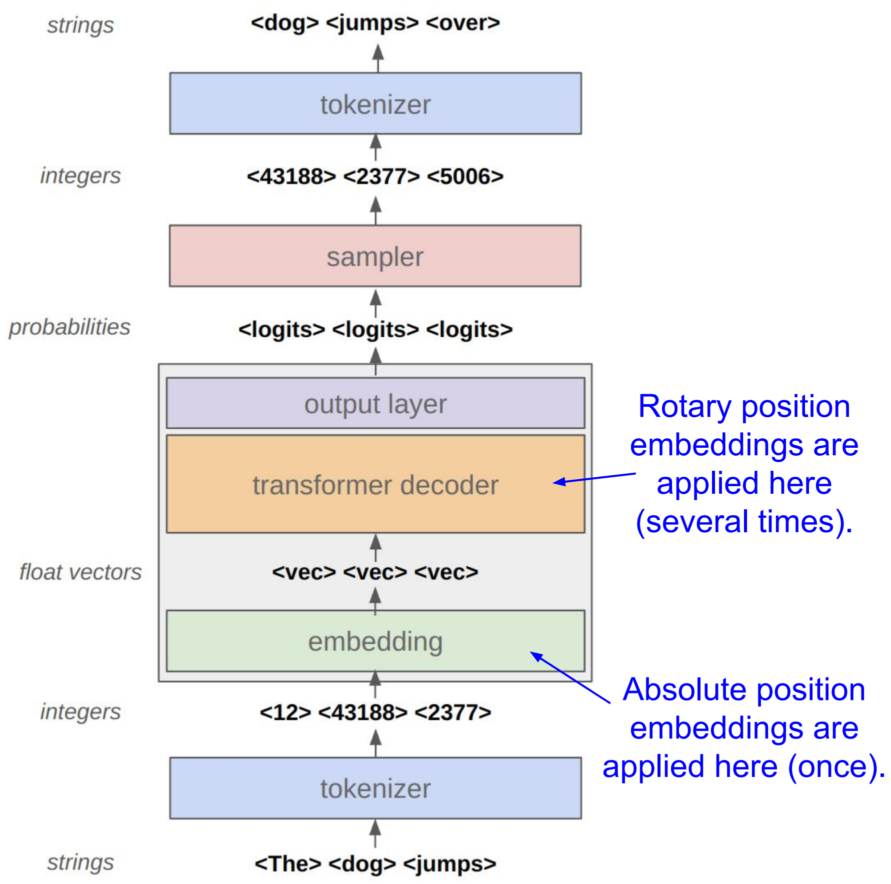
Simply rotate each pair of features. Each pair uses a different value for θ:

$$\text{Emb}(\text{<token id>}) = [\underbrace{0.1, 0.5,}_{\substack{\text{Rotate}\\ \text{by } \theta_0}} \underbrace{0.3, 0.8,}_{\substack{\text{Rotate}\\ \text{by } \theta_1}} \underbrace{0.8, 0.1,}_{\substack{\text{Rotate}\\ \text{by } \theta_2}} \underbrace{0.2, 0.1,}_{\substack{\text{Rotate}\\ \text{by } \theta_3}} \underbrace{0.8, 0.9}_{\substack{\text{Rotate}\\ \text{by } \theta_4}} \ldots.]$$

Leftover Notes

RoPE is a little complex to implement in practice, but does result in improved models (RoPE models consistently outperform previous models by a few % better loss).

RoPE is applied many times throughout the model, rather than just at the beginning. As computations are performed on the vectors in the transformer layers, the rotations are applied periodically to keep the position information “fresh”.



End of Lecture