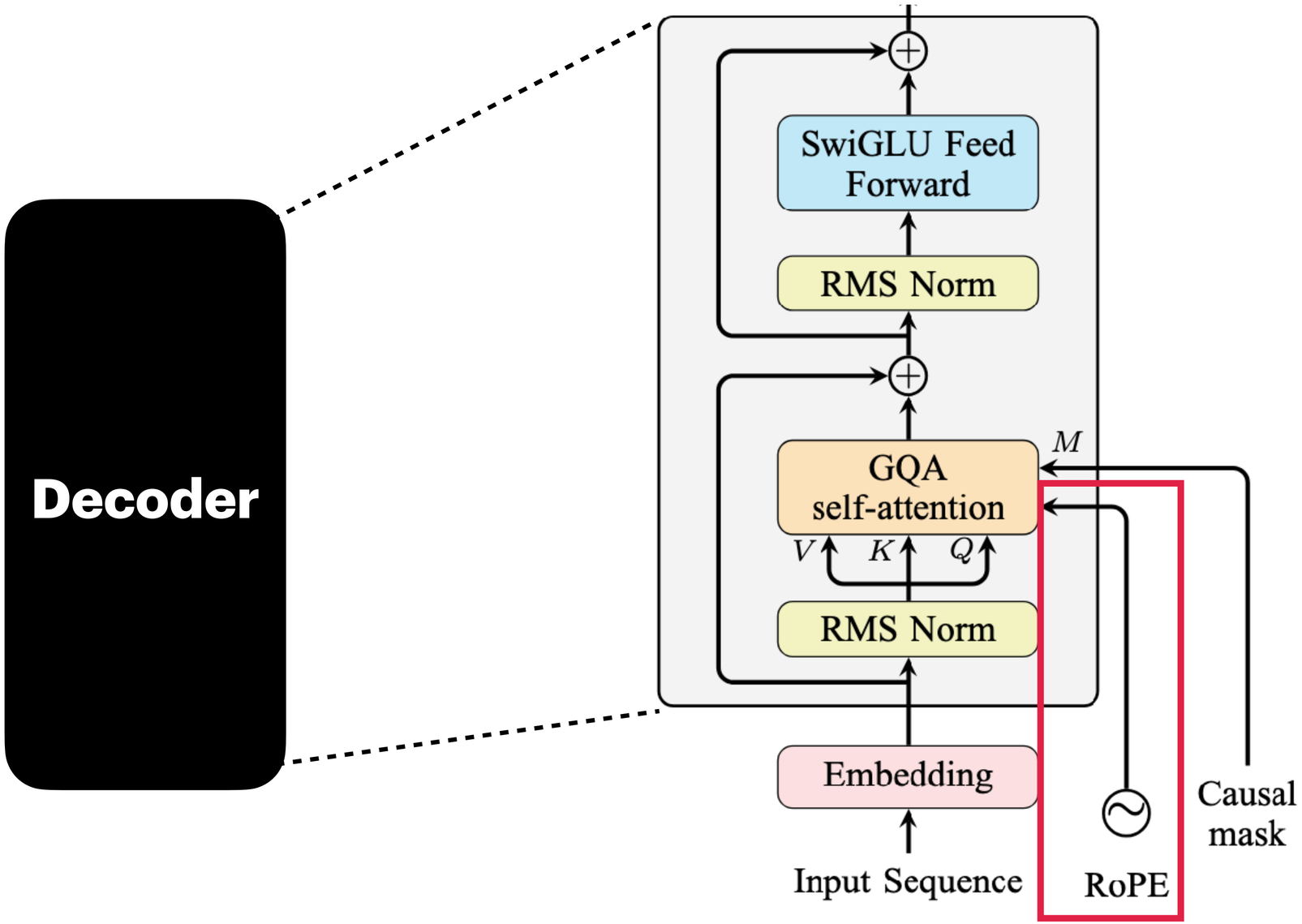
Tokenization and Positional Embeddings

Aaron Mueller
CS599 B1: Advanced Natural Language Processing
Sep. 10, 2026

Reminder: The Decoder Block

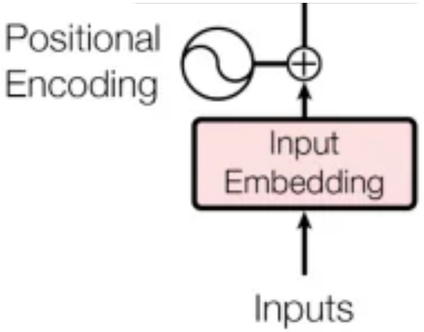


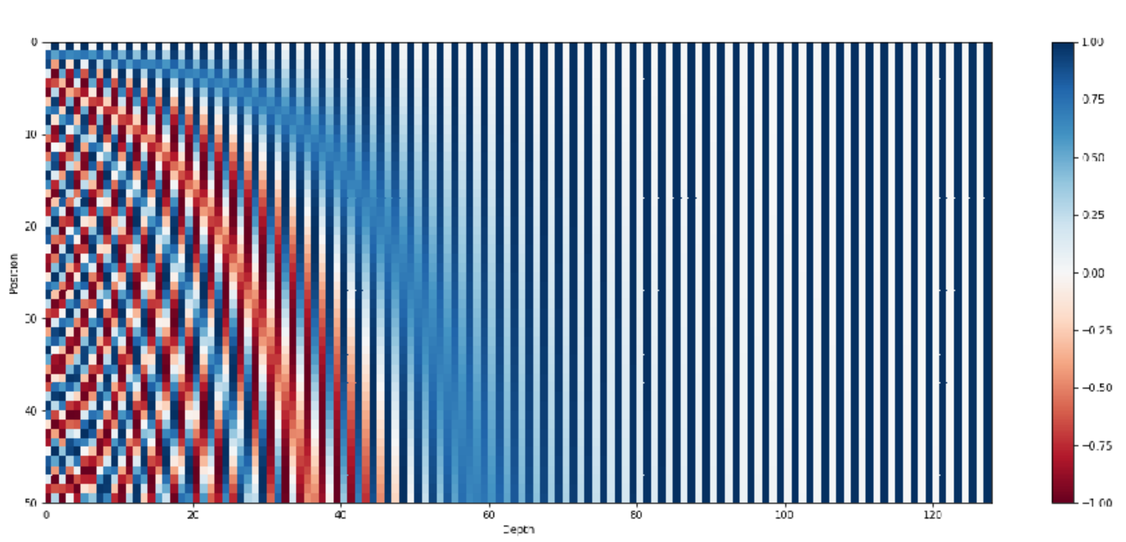
Positional Embeddings

Back in the Day

- Self-attention on its own does not convey position
  - Position is important: “dogs chase cats” is very different than “cats chase dogs”

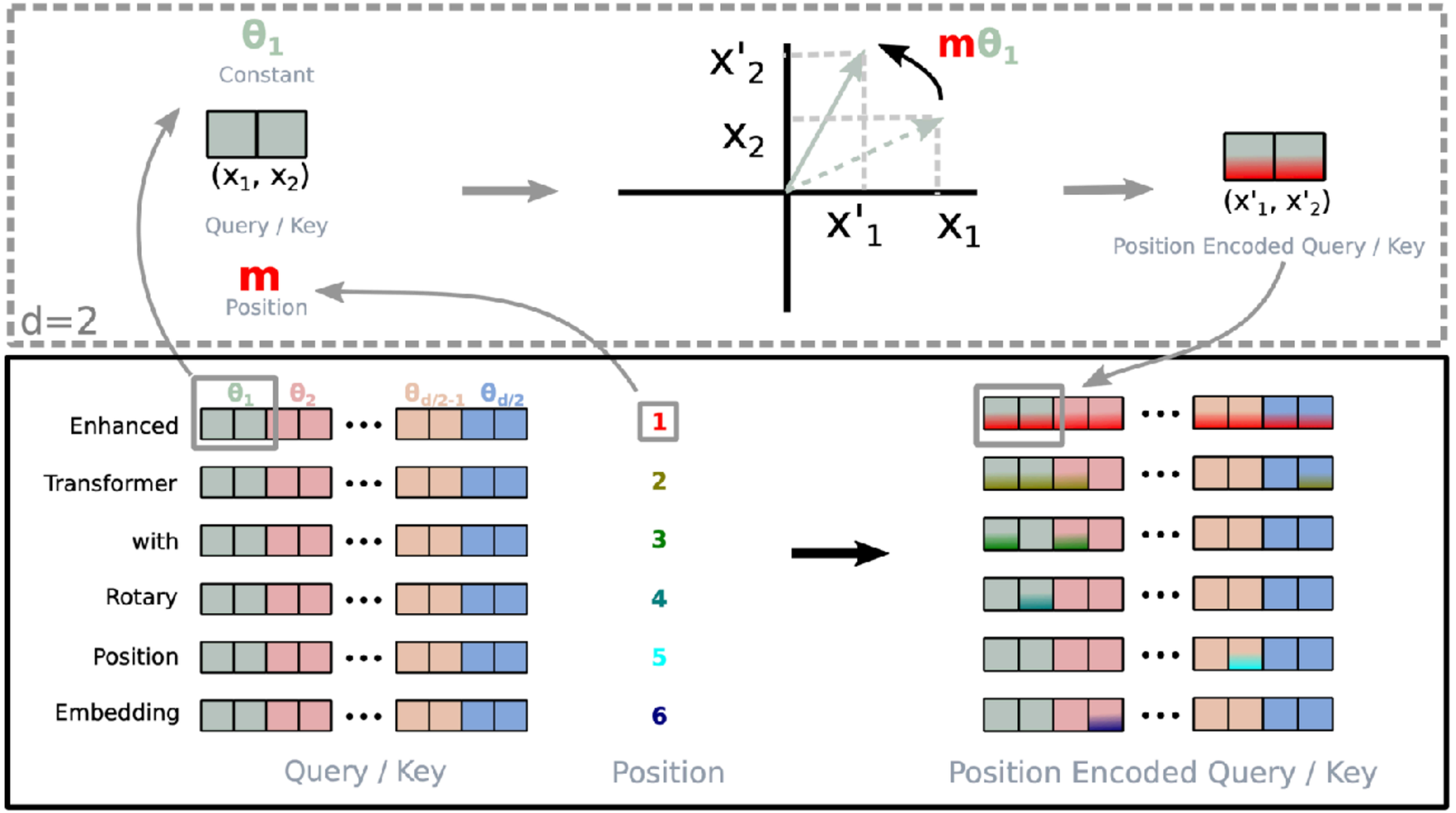
- Vaswani et al. (2017) added position information to the inputs:
  - These encode **absolute position**: each index has a unique representation
  - Uses alternating sine and cosine frequencies ->





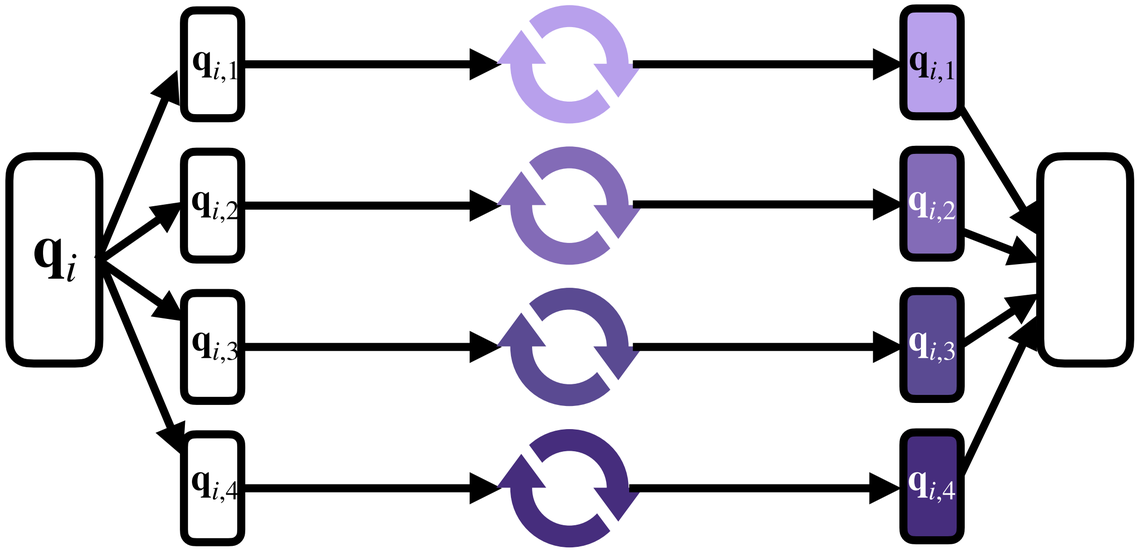
Rotary Position Embedding (RoPE)

We don't usually care about *absolute* token position. We care more about the *relative distance* between tokens. This is the main idea behind RoPE **(Su et al., 2021)**

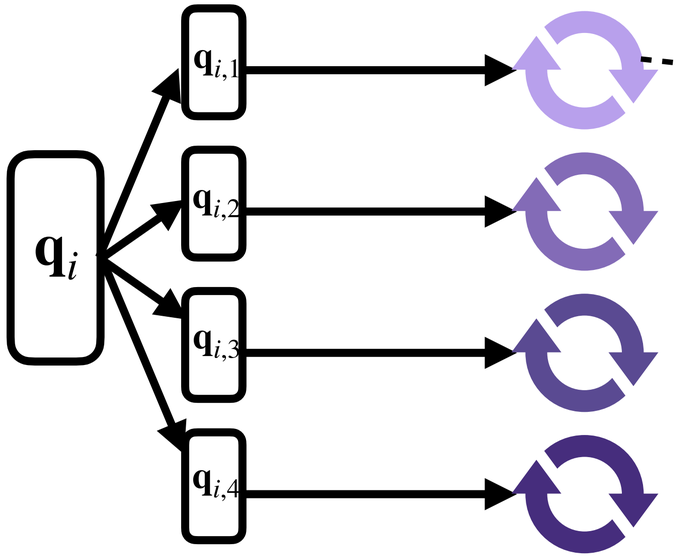


Rotary Position Embedding (RoPE)

1. Break representation $\mathbf{q}_i$ at position i into $\frac{d}{2}$ vectors of length 2
2. Rotate each vector $\mathbf{q}_{i,k}$ by some amount depending on position i and vector index k



Rotary Position Embedding (RoPE)



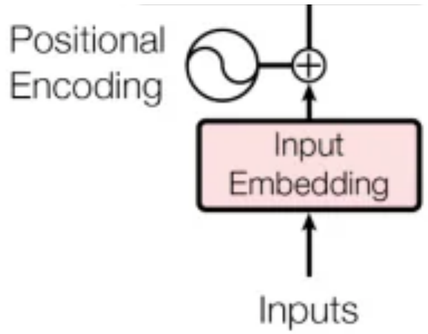
$$\theta_{i,k} = \frac{i}{\Theta^{(2k-2)/d}}$$

$$R_{i,k} = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}$$

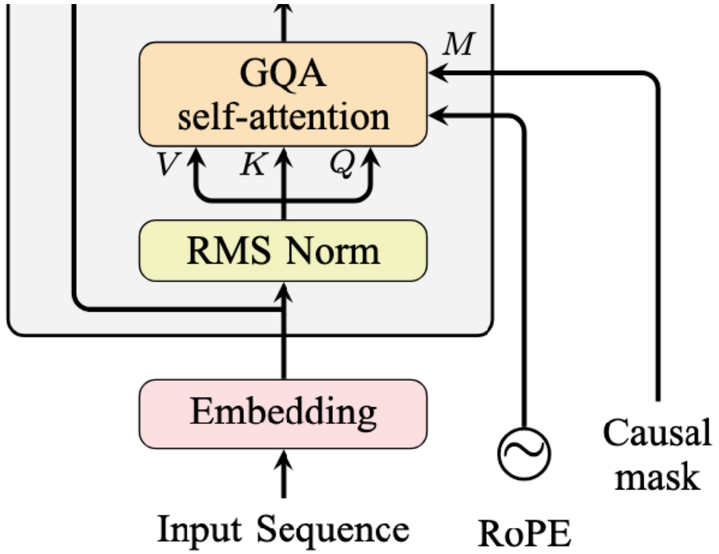
Rotate each $\mathbf{q}_{i,k}$ by $\theta_{i,k}$, where $\theta_{i,k}$ depends on the vector index and the dimensionality of the embeddings d

Θ is a constant. Su et al. used 10000, but recent models use 500K - 1M

Rotary Position Embedding (RoPE)



In Vaswani et al. (2017), the positional embeddings were added to the word embeddings at the input.



With RoPE, we apply positional embeddings to the **keys** and **queries** just before self-attention is applied

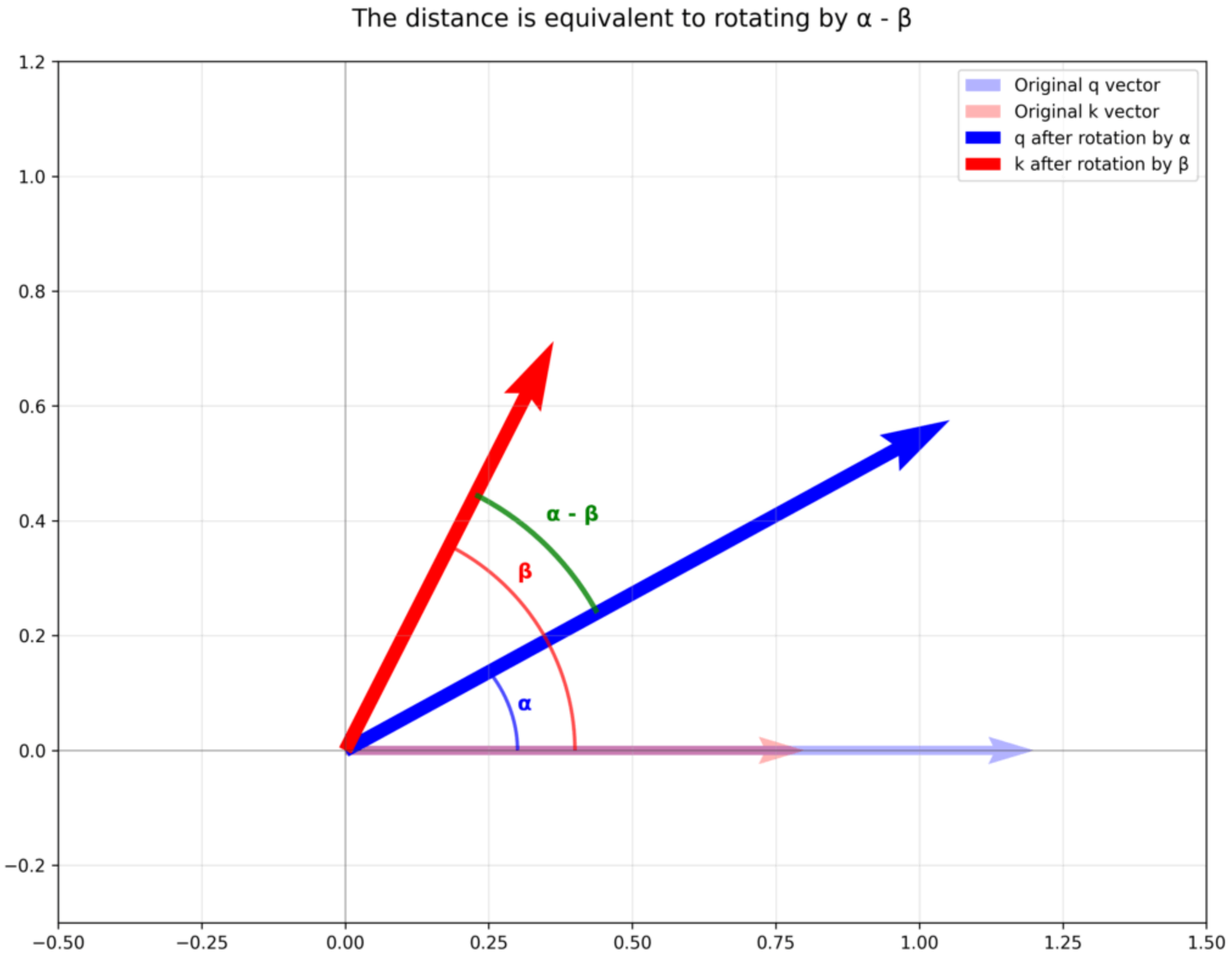
Rotary Position Embedding (RoPE)

- For a given query token $\mathbf{q}_i = W_Q \mathbf{x}_i$, apply our rotation matrix R_i to get $\mathbf{q}'_i = R_i \mathbf{q}_i = R_i W_Q \mathbf{x}_i$
- As we saw, $\mathbf{q}_i$ is actually split into a bunch of 2D vectors rotated by different angles

$$R_{i,k} = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}$$

$$\theta_{i,k} = \frac{i}{\Theta^{(2k-2)/d}}$$

- Do the exact same thing for key tokens $\mathbf{k}_j = W_K \mathbf{x}_j$:
  - $\mathbf{k}'_j = R_j \mathbf{k}_j$, using the same rotation angles $\theta_{j,k}$ as for queries



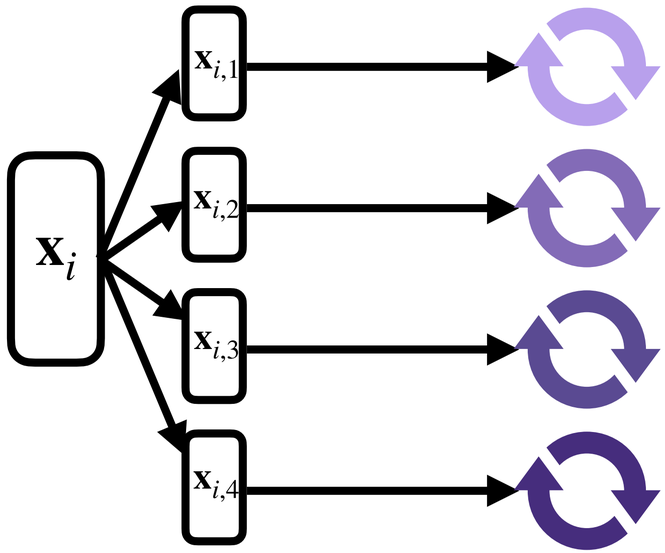
Imagine rotating $\mathbf{q}_i$ by angle α and $\mathbf{k}_j$ by angle β

The absolute values of α and β are irrelevant; only $\alpha - \beta$ matters

For two tokens at positions i and j, the rotation modifies them proportionally to $|i - j|$

[Cesconetto, 2026]

Why rotate different dimensions by different amounts?



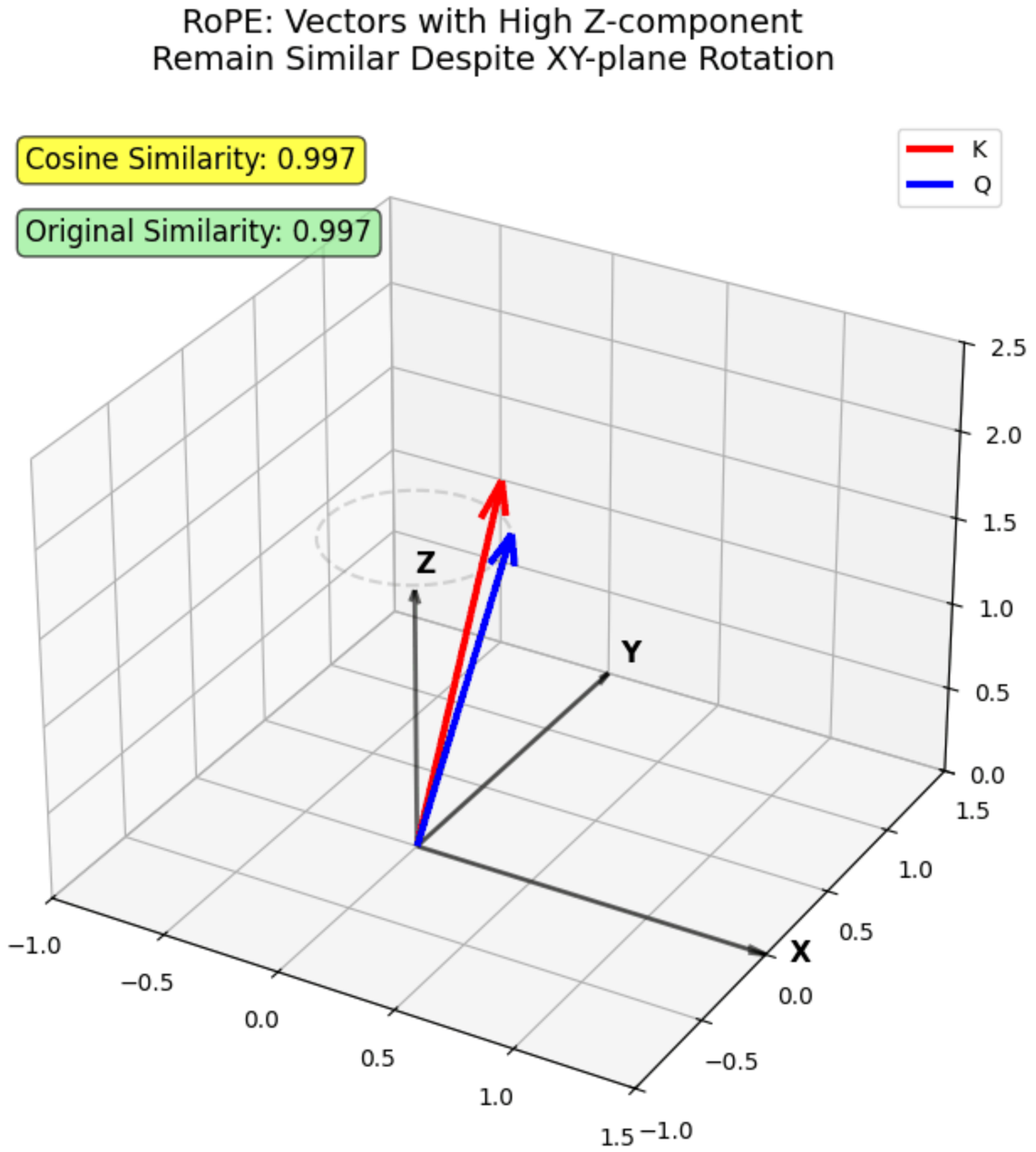
This seems odd at first. Why not just rotate the entire query/key vector by the same angle depending only on token position?

Earlier dimensions rotate more quickly; later dimensions rotate more slowly.

This lets the model choose what features should have shorter- or longer-range influence!

Early dimensions must attend to important short-range phenomena; they rotate fast.

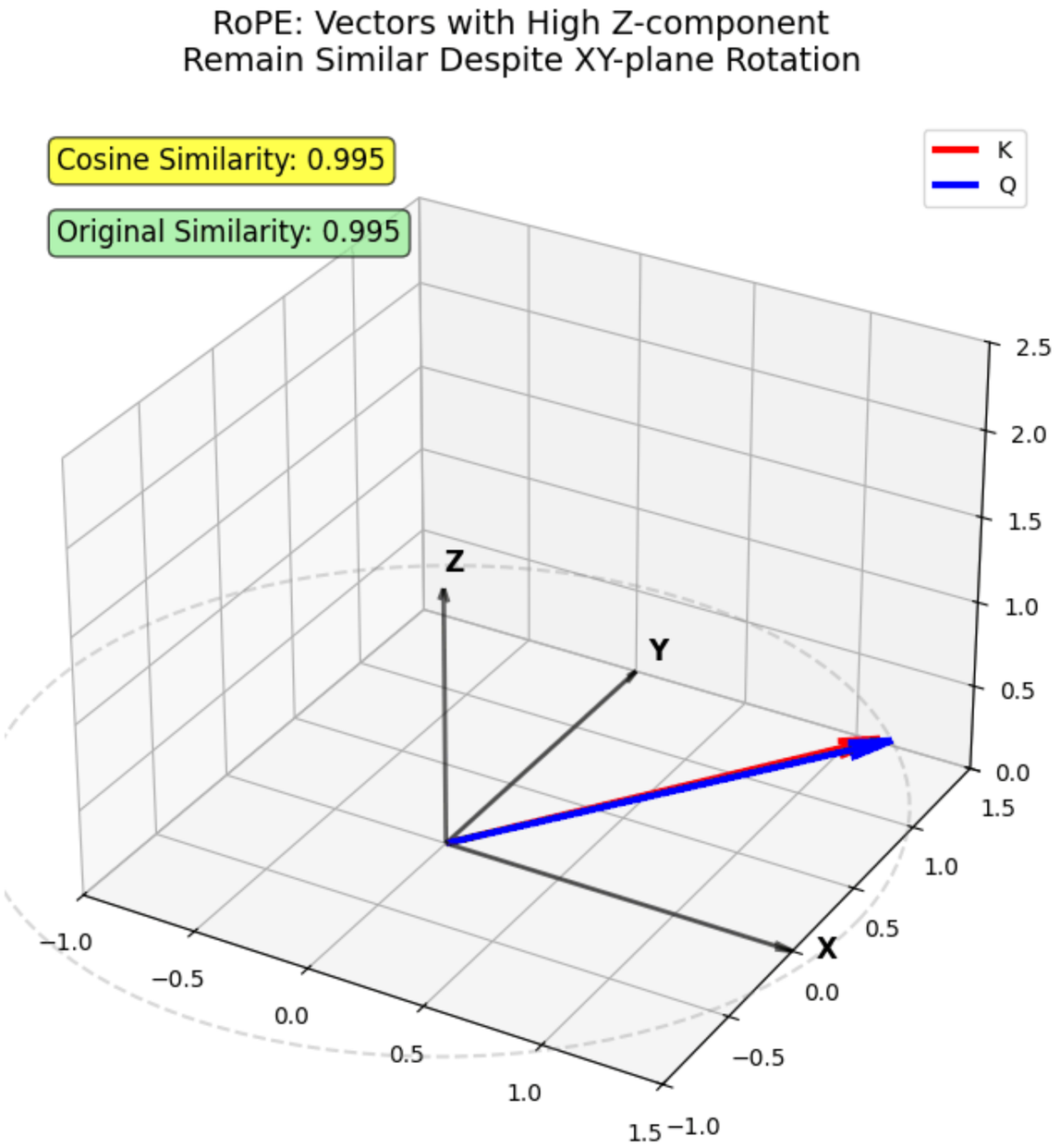
Later dimensions can contain information that should be matched across long distances.



$$\theta_{i,k} = \frac{i}{\Theta^{(2k-2)/d}}$$

The higher k is (the higher the dimension index), the less rotation you'll have.

This allows the queries and keys to align at more distantly separated positions.



$$\theta_{i,k} = \frac{i}{\Theta^{(2k-2)/d}}$$

The lower k is (the lower the dimension index), the more rotation you'll have.

This means that fine-grained/local positional differences will have greater influence in these dimensions

Yet another RoPE extensioN method (YaRN)

- What if we'd like to extend the size of the context window after training?
- If RoPE is trained with encodings up to context window length L, you can expand to L' via:

$$\theta_{i,k} = \frac{iL/L'}{\Theta^{(2k-2)/d}}$$

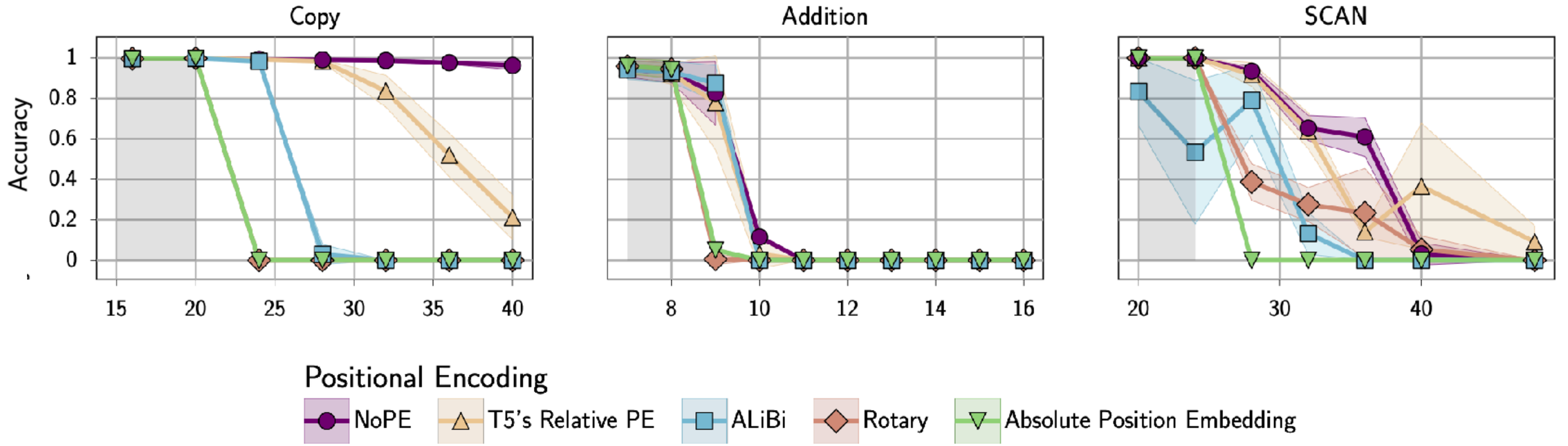
- At index k, we can define a notion of "wavelength", which is how many tokens are needed to do a full rotation:

$$\lambda_k = \frac{2\pi}{\theta_k} = 2\pi\Theta^{\frac{2k}{|D|}}$$

- When λ_k is small, don't use position interpolation; if large, use it
- We can also introduce temperature t on attention to further rescale it

No Positional Embeddings (NoPE)

- Is any of this really necessary? Some like **Kazemnejad et al. (2023)** find that models can generalize surprisingly well without position embeddings
- As part of Homework 1, you'll try this out for yourself!



Tokenization

Tokens

- The base unit of input to a language model is a **token**.

- So far, we've assumed that tokens are words for clarity. These days, word-level tokenization is almost never used.

- Why not?

Words as Tokens

- Many assume that a **word** is the basic unit of language, so let's start there.
- How might we want to preprocess text before mapping to word representations?

Mr. Johnson thinks the boys' stories about San Francisco aren't amusing.

Mr. Johnson thinks the boys | ' stories about San | Francisco are | n't amusing | .

- Split contractions, separate punctuation from words, handle compounds and multi-word concepts
- *Maybe* filter stop words, lowercase

Words as Tokens

First idea: take the top-*k* most common words in a dataset as our vocabulary.

Text:	The	man	saw	the	cat	.
Tokens:	**11**	**387**	**720**	**5**	**407**	**3**

Tokens are represented as *indices* in a vocabulary

The Finite Vocabulary Problem

- There are an *infinite* number of words. Thus, any finite vocabulary based in words will not fully cover natural language.

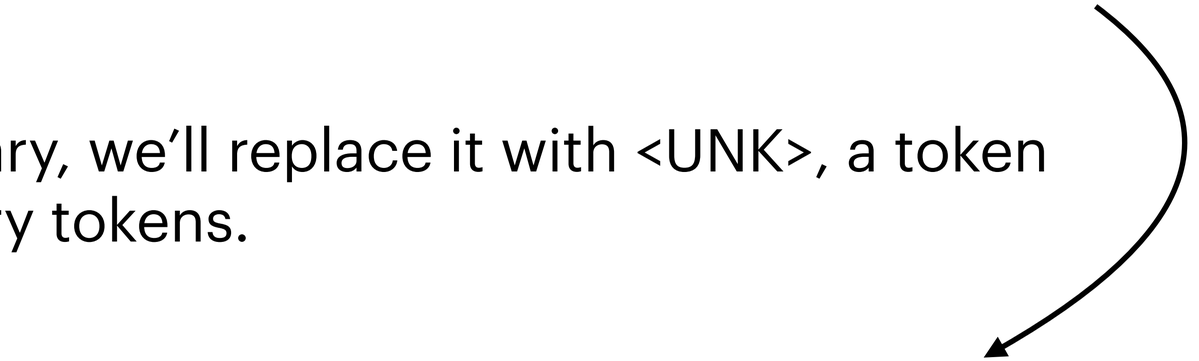
- What if we see tokens in the test set that weren't in the training set? What if the vocabulary is too small for how big the dataset is?

- If we encounter a word not in our vocabulary, we replace it with a special <UNK> token.
  - <UNK> has its own representation and probability.
  - This token will kill our language model's quality fast. We want to minimize how often this token appears as much as possible.

The Finite Vocabulary Problem

The subject was Argus-eyed; he perceived the glint of a feline's eyes.

If a word is outside our vocabulary, we'll replace it with <UNK>, a token for unknown or out-of-vocabulary tokens.



The subject was <UNK>; he <UNK> the <UNK> of a <UNK> eyes.

Other Limitations of Word Tokenizers

- Word-level tokenizers will consider different forms of the same word as different tokens:

run, runs, ran, running

apple, apples

- This means these forms will all have separate representations
- Also an issue in languages that have very complex **morphology.**

Characters

The man saw the cat.

↓ *Character-level tokenizer*

T,h,e,_,m,a,n,_,s,a,w,_,t,h,e,_,c,a,t,.

Pros:

- Solves the finite-vocabulary problem—to a degree.
  (But may not work as well for Chinese, which has >100,000 characters.)
- Easy to implement.

Cons:

- Can be hard to train a good language model. *Long* contexts,
  and the same character can appear in many different contexts.

Bytes

A byte is a sequence of 8 bits. There are only 256 possible bytes.

Strings can be mapped to/from bytes via an encoding called UTF-8

```
>>> s = "BU is in Boston"
>>> b = s.encode("utf-8")
>>> print(list(b))
[66, 85, 32, 105, 115, 32, 105, 110, 32, 66, 111, 115, 116, 111, 110]
```

There are far more than 256 characters in the world. How does UTF-8 handle this?

```
>>> s = "你好"
>>> b = s.encode("utf-8")
>>> print(list(b))
[228, 189, 160, 229, 165, 189]
```

Some characters consist of multiple bytes

2016: Subword Tokenization

- Developed for machine translation by **Sennrich et al. [2016]**

> "The main motivation behind this paper is that the translation of some words is transparent in that they are translatable by a competent translator even if they are novel to him or her, based on a translation of known subword units such as morphemes or phonemes."

- Later used in BERT, RoBERTa, GPT-2, among other models
- Relies on a simple algorithm called *byte-pair encoding*

Byte-pair Encoding

1. *Split corpus into characters.*

the man saw the cat.

t,h,e,_,m,a,n,_,s,a,w,_,t,h,e,_,c,a,t,.

2. *Count each pair of characters:*

(t,h): 2

(h,e): 2

(m,a): 1

(a,n): 1

...

3. Merge the highest-frequency pair into one token:

(t,h) -> th

th,e,_,m,a,n,_,s,a,w,_,th,e,_,c,a,t.

4. Repeat *m* times, where *m* is the number of merges (a hyperparameter).

Byte-pair Encoding

the man saw the cat.

th,e,_,m,a,n,_,s,a,w,_,th,e,_,c,a,t.

2. *Count each pair of* ***tokens****:*

(th,e): 2
(m,a): 1
(a,n): 1
...

3. Merge the highest-frequency pair into one token:

(th,e) -> the

the,_,m,a,n,_,s,a,w,_,the,_,c,a,t.

4. Repeat *m* times, where *m* is the number of merges (a hyperparameter).

Byte-pair Encoding

```
function BYTE-PAIR ENCODING(strings C, number of merges k) returns vocab V

  V ← all unique characters in C          # initial set of tokens is characters
  for i = 1 to k do                       # merge tokens k times
    t_L, t_R ← Most frequent pair of adjacent tokens in C
    t_NEW ← t_L + t_R                     # make new token by concatenating
    V ← V + t_NEW                         # update the vocabulary
    Replace each occurrence of t_L, t_R in C with t_NEW   # and update the corpus
  return V
```

1. *Split inputs into characters.*
2. *Count each pair of tokens.*
3. *Merge the highest-frequency pair into a new token. Do not merge across word boundaries.*
4. *Repeat k times, where k is the number of merges (a hyperparameter).*

Byte-pair Encoding

- To avoid <UNK>, all possible characters or symbols need to be in the base vocab. This can be a *lot*!
    - Unicode has hundreds of thousands, and growing!

- GPT-2 uses *bytes* rather than *characters* as the base vocabulary (only 256 of them), and applies BPE on top of byte sequences (with some special rules to prevent certain kinds of merges).

- Depending on our dataset, our vocabulary size is typically in [32K, 256K]

Byte-pair Encoding over Bytes

- We can apply this same algorithm to sequences of bytes
- More frequent pairs get merged; we eventually get characters/words

- **Pro:** Never need to worry about out-of-vocabulary items
- **Con:** It's possible in theory to generate invalid characters, or characters in the wrong script

Hyperparameters

- In practice, common tokenizers tend to use subword vocabularies with tens of thousands to hundreds of thousands of entries.
  - BERT (2018): 30,522
  - GPT-2 (2019): 50,257
  - Llama 3.1 (2024): 128,256
  - GPT-4o (2024): ≈200,000
  - Qwen 3.5 (2026): 248,320

Unigram Tokenization

- Does it always make sense to pre-tokenize based on whitespace?
- Instead of splitting by frequency, why don't we define a loss function that defines what good tokenization looks like? Given segmentation T:

$$p(x_{1:L} \mid T) = \prod_{(i,j) \in T} p(x_{i:j})$$

Probability of full sequence

Probability of a specific token

- This is just a unigram language model.

Unigram Tokenization

- Given a large starting vocabulary V (all characters or bytes plus frequent substrings), we want to maximize the likelihood $p(x_{1:L})$ of the full corpus.

1. Use expectation maximization to estimate $p(x)$ for each token $x \in V$
2. For each $x \in V$, compute how much the corpus likelihood would go down if x were removed from the vocabulary
3. Keep the top 80% of tokens in V by the score in 3. (Do not prune base characters.)
4. Repeat until $|V|$ is the size we want

SentencePiece

- SentencePiece is a *library*, not an algorithm
  - Includes support for BPE and Unigram tokenizers

- SentencePiece's contribution is to treat input as a raw Unicode stream with no language-specific pretokenizer
  - Whitespaces are replaced with _ (not an underscore!) and treated like any other character - although we usually disallow merges across word boundaries
  - Supports most but not all characters; for unsupported characters, fall back to bytes

- **Used by:** Llama 1 and 2, Gemma, many multilingual models

Tokenization in Practice

- Step 0: Chunk your text into n non-overlapping blocks for parallel processing
- Step 1: Pretokenization
- Step 2: Compute merges/token vocabulary with your favorite algorithm, like BPE
- Step 3: Encode and decode

Tokenization in Practice

Chunking

Spot. Spot saw the shiny car and said, "Wow, Kitty, your car is so bright and clean!" Kitty smiled and replied, "Thank you, Spot. I polish it every day."

After playing with the car, Kitty and Spot felt thirsty. They found a small pond with clear water. They drank the water and felt very happy. They played together all day and became best friends.

<|endoftext|>

Once upon a time, in a small yard, there was a small daisy. The daisy had a name. Her name was Daisy. Daisy was very small, but she was also very happy.

One day, Daisy saw a dog. The dog was big and had a name too. His name was Max. Max liked to play in the yard. Daisy liked to watch Max play. Max and Daisy became friends.

Every day, Max would come to the yard to play. Daisy would watch and smile. They were very happy together. And even though Daisy was small, she knew that she had a big friend in Max.

<|endoftext|>

Split into n chunks to process in parallel

Do not include special tokens like <|endoftext|>

The provided `pretokenization_example.py` script in the homework will give you good chunk boundaries

Tokenization in Practice

Pretokenization

- We then break everything into a tokens using this regular expression:

```
r"""'(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+"""
```

- This nifty expression does a lot:
  - ?\p{L}+ finds one or more letters with leading spaces
  - ?\p{N}+ finds numbers
  - (?:[sdmt]|ll|ve|re) splits off contractions
  - And more

Tokenization in Practice

Encoding and Decoding

Encoding: map string to sequence of token IDs

- Naive idea: match the longest token that applies at the current step
  - *Do not do this!*
- You should first pre-tokenize the text in the same way you did before training your BPE tokenizer, and then apply the merges to the pre-tokens in the same order of creation. See HW1 instructions for more detail.
- Don’t forget special tokens!

Tokenization in Practice

Encoding and Decoding

Decoding: map sequence of token IDs to string

- A bit easier: look up each ID's corresponding vocabulary entry (a byte sequence), concatenate them, then decode into a Unicode string
- If a token ID does not produce a valid Unicode string, replace malformed bytes with the official Unicode replacement character `U+FFFD`

Open Problems in Tokenization

A tokenizer trained well for one corpus may not generalize well because of:

- **Language imbalance**: A great English tokenizer would not necessarily be a good Turkish tokenizer
- **Domain shift**: A tokenizer that works well for scientific articles would not necessarily work well for social media
- **Temporal shift**: A tokenizer trained on internet text from before the year 2000 may not effectively handle text from the 2025 internet.

Open Problems in Tokenization

Handling numbers is particularly tricky. Let's say you want to represent this sequence:

```
85,219 x 20 =
```

A BPE-based tokenizer might spit out something like:

```
[8, 5, \,, 21, 9, x, 20, =]
```

Clearly this isn't great. Some models (like Gemma 2) just split all digits into their own tokens; others (like Llama 3) preserve common multi-digit sequences. There are trade-offs to both approaches.

Homework Tips

- Even with a good implementation, encoding the training dataset into tokens may take a *long* time (a few hours on my machine)
    - Be sure to use multiprocessing

- Remember that you're allowed to ask LLMs for help, but also that you should aim to understand what's in your implementation

Homework Tips

You should now have what you need to approach the following problems in HW1:

- Tokenization: `unicode1`, `unicode2`, `train_bpe`, `train_bpe_tinystories`, `tokenizer`, `tokenizer_experiments`
- Modeling: `linear`, `embedding`, `rmsnorm`, `positionwise_feedforward`, `rope`, `softmax`, `scaled_dot_product_attention`, `multihead_self_attention`, `transformer_block`, `transformer_lm`, `cross_entropy`

Next Week

- *Tuesday:* Optimizers and training
  - Adam
  - AdamW
  - Training loops

- *Thursday:* Scaling laws, emergence, and in-context learning