BZAN 583: Generative AI with Large Language Models for Business Applications

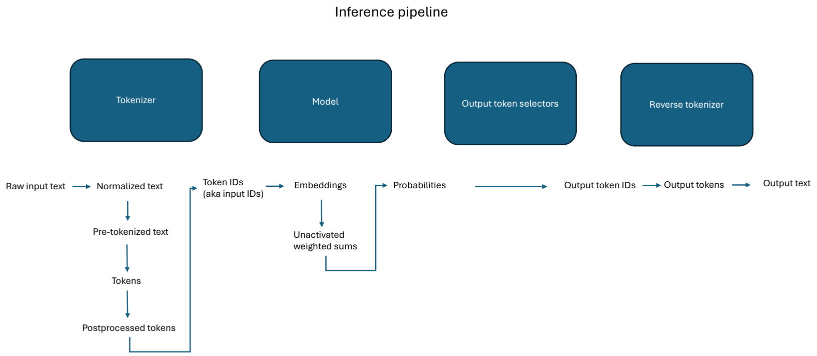
The Algorithm behind LLMs: Transformers

Michel Ballings

Outline I

Outline II

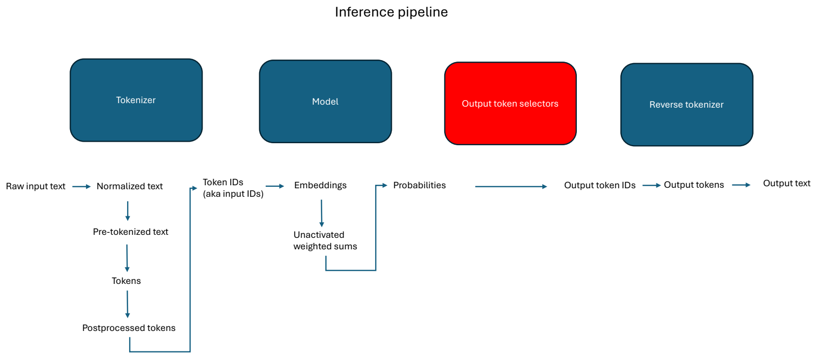
Inference Pipeline Overview



Outline I

Outline II

Output Token Selection



Outline I

Outline II

Background

- Also called sampling and decoding. Here we will use token selection to avoid confusion with other parts of the pipeline.
- In the introduction we talked about the text generation loop and how LLMs output one word at a time. We said that LLMs were *autoregressive*, because they generate the next word in a sequence based on the previous words it has already generated (and the prompt).
- We also talked about how LLMs output a probability distribution over the vocabulary (all the unique words it could choose from): every word gets a probability denoting how probable it is that this word is the next word that comes in the sequence (the conditional next word distribution).
- To calculate how probable a whole sequence of words is, we can simply multiply the conditional next word distributions. The probability of a sequence

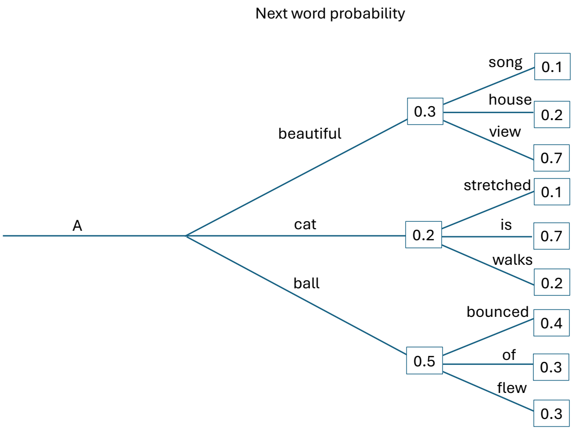
$$P(y_{1:T}|x) = \prod_{t=1}^{T} P(y_t|y_{1:t-1}, x), \tag{1}$$

where x denotes the prompt, y_t denotes a word at time step t, and T denotes the length of the generated sequence.

Outline I

Outline II

Next-word Probability Distribution



Outline I

Outline II

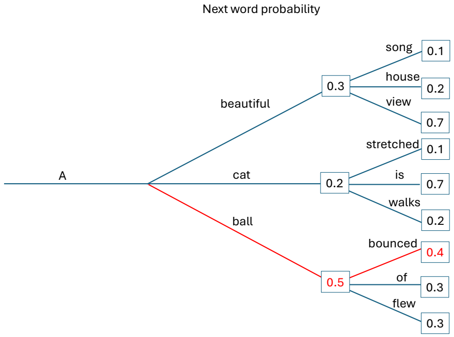
Decoding Methods

Greedy Search

At each timestep t, greedy search selects the word y with the highest probability as the next word. Formally, the next word

$$y_t = argmax_y P(y|y_{1:t-1}). \tag{2}$$

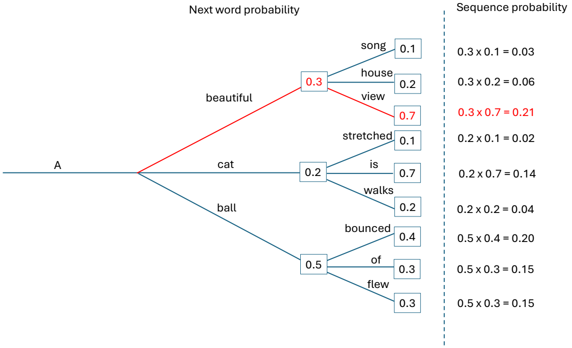
Below, starting from 'A' the algorithm chooses 'ball' because its probability of 0.5 is higher than the other two options (cat=0.2, beautiful=0.3). Rolling this out until length T=3, we get 'A ball bounced', with sequence probability 0.2 (=0.5 x 0.4)



Decoding Methods

Beam Search

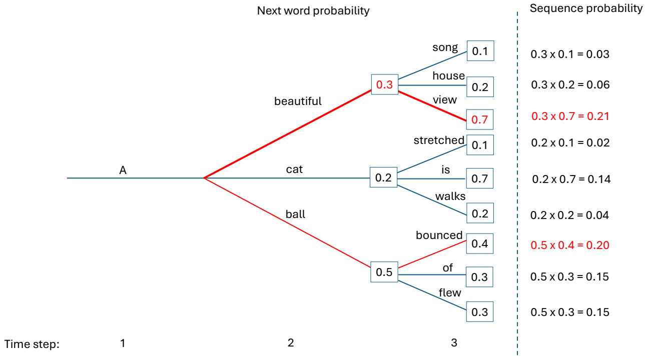
The largest drawback of greedy search is that it potentially misses out on sequences with a higher probability. For example, the word ‘view’ has a very high probability of 0.7 but when using greedy search the algorithm will never get to this word because the preceding word (‘beautiful’) will never be selected as next word after the word 'A'. In other words, a high-probability word is hidden behind a low-probability word. The greedy search method selected ‘A ball bounced’ with sequence probability equal to 0.2. In contrast, ‘A beautiful view’ has a sequence probability of 0.21.



Decoding Methods

Beam Search

- Beam search lowers the risk of missing out on high probability word sequences by retaining at any time step t the n most likely sequences up until t. For the example below, let the number of beams n=2.
- At t=2, the algorithm retains the most likely sequence ('A ball'), and the second most likely sequence ('A beautiful').
- At t=3 the algorithm finds that 'A beautiful view' has a larger probability (0.21) than 'A ball bounced' (0.20).



Decoding Methods

Beam Search

- Therefore beam search ends up returning 'A beautiful view' to the user, whereas greedy search returns 'A ball bounced'. Beam search will always find a sequence with greater than or equal probability than greedy search, but there is no guarantee that it will find the absolute most likely sequence.
- How many beams should we have? More beams results in better results, but it is also slower and the memory requirements are higher. Typical values: production settings: 10-100, research: 1000-3000
- Note that multiplying many probabilities will result in numerical underflow. Remember the sequence probability

$$P(y_{1:T}|x) = \prod_{t=1}^{T} P(y_t|y_{1:t-1}, x). \tag{3}$$

- Therefore the sum of the log probabilities is taken instead. The log is a monotonically increasing function and both options arrive at the same maximum. The sequence log probability

$$log\ P(y_{1:T}|x) = \sum_{t=1}^{T} log\ P(y_t|y_{1:t-1}, x). \tag{4}$$

Decoding Methods

Beam Search

- Because we will have to multiply more probabilities (sum more log probabilities) for longer sequences, shorter sequences will have a larger sequence probability (log probability)[1]. Therefore, beam search would favor shorter sequences. To mitigate this issue, the algorithm does length normalization (divide the sum of log probabilities by the number of words in the sequence), ensuring that the generated sequences are not unfairly penalized for being longer. The transformer library does this automatically. The log probability of the sequence

$$log\ P(y_{1:T}|x) = \frac{1}{T}\sum_{t=1}^{T} log\ P(y_t|y_{1:t-1}, x), \tag{5}$$

[1]Note that the logarithm of a value between 0 and 1 is a negative number. So summing a smaller number of negative values will result in a less negative total.

Decoding Methods

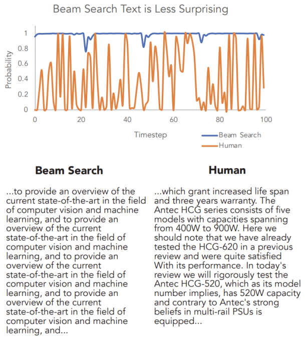
Beam Search

- Beam search can produce repeating n-grams because it prioritizes the most probable sequences at each step, which can lead to getting stuck in a loop where the model repeatedly selects the same high-probability n-gram combination.
- This can result in repetitive patterns in the generated text, especially if the training data exhibits such patterns or if the beam size is too narrow.
- This issue can be mitigated by penalizing repeated n-grams during the decoding process. Specifically we set the probability of next words that could create an already seen n-gram to 0.
- We need to use this with care. For example, an article about 'chocolate cookies' using a 2-gram penalty will only have the words 'chocolate cookies' once in the whole text. To mitigate this issue, we could, instead of setting the probability to 0, apply a smaller penalty so that the words can still be selected but with lower probability. The issue is then to choose the magnitude of the penalty.

Decoding Methods

Beam Search

- It turns out that, for decoding, maximization-based methods such as beam search lead to degenerate text (bland, incoherent, repetitive).
- This is a counter-intuitive finding, given that we are maximizing the likelihood during training. (`https://arxiv.org/pdf/1904.09751`)



Beam Search

...to provide an overview of the current state-of-the-art in the field of computer vision and machine learning, and to provide an overview of the current state-of-the-art in the field of computer vision and machine learning, and to provide an overview of the current state-of-the-art in the field of computer vision and machine learning, and to provide an overview of the current state-of-the-art in the field of computer vision and machine learning, and...

Human

...which grant increased life span and three years warranty. The Antec HCG series consists of five models with capacities spanning from 400W to 900W. Here we should note that we have already tested the HCG-620 in a previous review and were quite satisfied With its performance. In today's review we will rigorously test the Antec HCG-520, which as its model number implies, has 520W capacity and contrary to Antec's strong beliefs in multi-rail PSUs is equipped...

Figure 2: The probability assigned to tokens generated by Beam Search and humans, given the same context. Note the increased variance that characterizes human text, in contrast with the endless repetition of text decoded by Beam Search.

Decoding Methods

Sampling

On February 14, 2019, OpenAI surprised the scientific community with an impressive article written by GPT-2; the top generations relied on **randomness** in the decoding method (in particular top-k sampling performed best) (`https://arxiv.org/pdf/1904.09751`). We will discuss the following sampling methods:

- Plain vanilla (multinomial) sampling
- Top-k sampling
- Top-p (nucleus) sampling

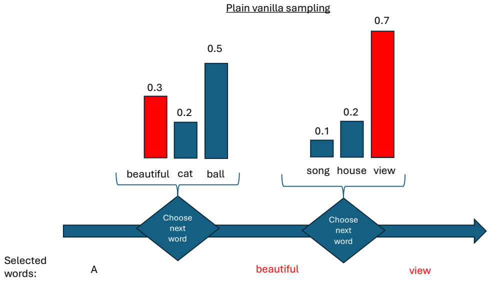
Sampling

Plain vanilla sampling

- Instead of Equation 2, we are going to sample the next word from the next word distribution:

$$y_t \sim P(y|y_{1:t-1}) \tag{6}$$

- This means that our text generation procedure is now stochastic.
- In the figure below, 'beautiful' is sampled from the conditional probability distribution $P(y|'A')$, and 'view' is sampled from $P(y|'A\ beautiful')$.



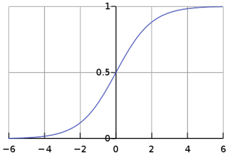
Sampling

Plain vanilla sampling

- While this sampling approach works well, in some cases the result may be incoherent (`https://arxiv.org/abs/1904.09751`).
- Can we influence the sampling? Yes, by tuning the *temperature* parameter.
- Before we can discuss temperature, we first need to discuss the softmax activation function:

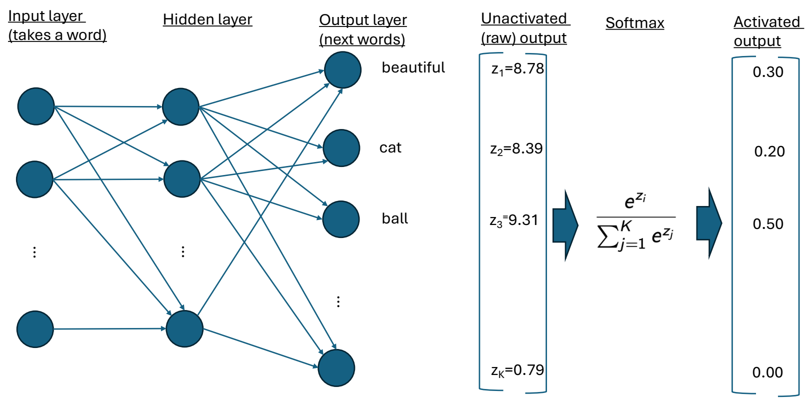
$$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{7}$$

- This function does two things:
  - map values z_i to [0,1], and
  - make sure that the sum of all the z_i equals 1.



Sampling

Plain vanilla sampling



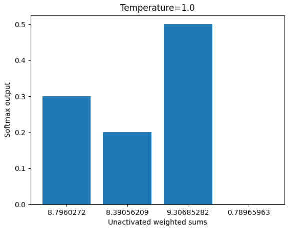
Sampling

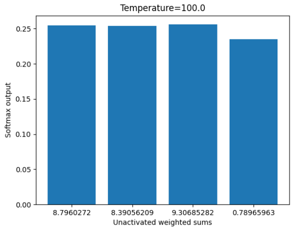
Plain vanilla sampling

- We can add a temperature parameter T to the softmax:

$$\frac{e^{\frac{z_i}{T}}}{\sum_{j=1}^{K} e^{\frac{z_j}{T}}} \tag{8}$$

- Comparing temperature 1 and 100 for unactivated weighted sums $= z = [8.7960272, 8.39056209, 9.30685282, 0.78965963]$





- Higher temperatures ($>$1) result in more diverse (sometimes incoherent) text
- Neutral temperature ($= 1$) has no impact on the softmax output.
- Lower temperatures ($>$0,$<$1) result in less diversity and creativity

Sampling

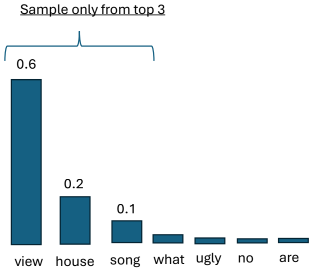
Top-k Sampling

- In plain-vanilla sampling it is unlikely that the algorithm will select very low probability words, but it is possible, especially across many steps. How do we mitigate this?
- Top-k sampling. Introduced in 2018 (`https://arxiv.org/pdf/1805.04833`). Adopted and made popular by GPT-2.
- In top-k sampling, the k most likely words are retained and the rest is ignored. This can help avoid a long tail of low probability words.
- Example; let's pretend that we need to sample our next word: $y_t \sim P(y|`A\ beautiful')$, and that we actually had six words (view, house, song, what, ugly, no, are) instead of the three (view, house, song) that we have been showing so far.

Sampling

Top-k Sampling

- Example continued. Let's pretend the distribution looks like this:

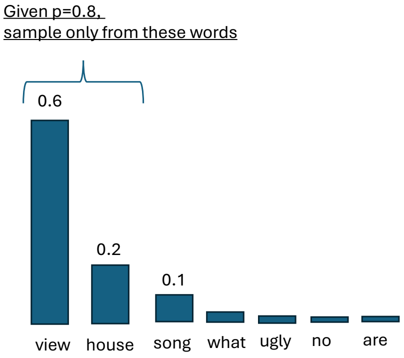


- Sampling from the last three words would clearly result in low quality sentences: 'A beautiful what', 'A beautiful ugly', 'A beautiful no', 'A beautiful are'.
- The solution is to only sample from the top-k, in this case k=3.

Sampling

Top-p (nucleus) Sampling

- In top-k we set the number of words the algorithm should consider.
- In top-p we set the cumulative probability of the words that the algorithm should consider.
- Top-p ranks the words from high to low probability and then selects the smallest number of words that cumulatively reach $p \in [0, 1]$ of the total probability mass.



Outline I

Outline II

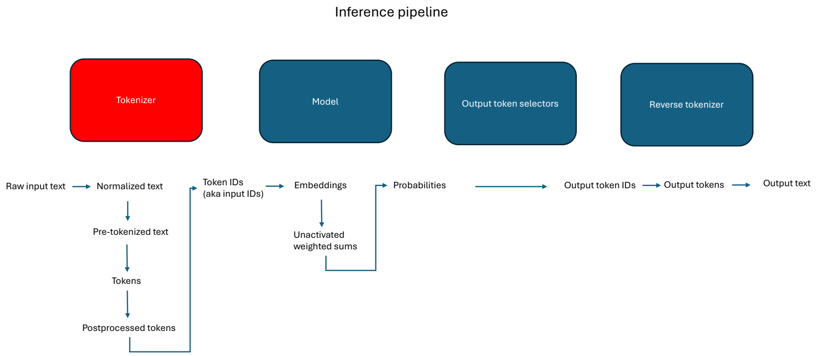
Conclusion

- Top-k and top-p sampling produce higher quality text than greedy and beam-search.
- It is presumed that this finding is driven by the likelihood maximizing nature of greedy and beam search together with the maximum likelihood training procedure
- Yet, top-k and top-p sampling have also been shown to generate repetitive text.
- Can we change the training procedure to learn models that work well with greedy and beam search?
- Yes, there is research (`https://arxiv.org/pdf/1908.04319`) on changing the loss function to include a penalty on high probabilities. The authors call it the unlikelihood loss, and coupled with greedy or beam search it is shown to outperform the standard loss with stochastic (top-k, top-p) decoders.
- It is therefore important to know how a model was trained so that we can select appropriate decoding methods.
- See *outputtokenselection.py* for greedy search, beam search, plain vanilla sampling, top-k sampling, and top-p sampling.

Outline I

Outline II

Tokenization



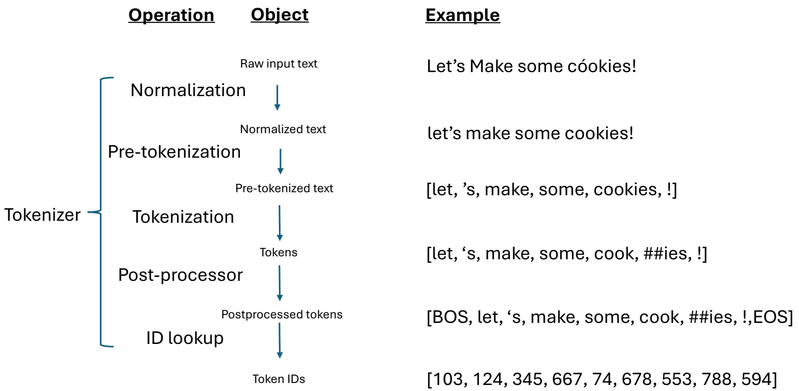
Outline I

Outline II

Tokenization steps

Problem	Happens where
Words and symbols $\rightarrow$ index (token IDs)	Tokenizer
Index $\rightarrow$ meaning (embeddings)	Embedding table

- Huggingface has developed the *tokenizers* library.



Let’s Make some cóokies!

let’s make some cookies!

[let, ’s, make, some, cookies, !]

[let, ‘s, make, some, cook, ##ies, !]

[BOS, let, ‘s, make, some, cook, ##ies, !,EOS]

[103, 124, 345, 667, 74, 678, 553, 788, 594]

Outline I

Outline II

Normalization

- Normalization includes
  - removing unnecessary whitespace,
  - converting to lowercase, and
  - removing accents.
- Normalization makes subsequent steps more efficient. For example, before normalization, there may be two words, ‘How’ and ‘how’. After normalization only one word remains: ‘how’.
- In traditional text processing pipelines (e.g., text mining), one will frequently see additional operations such as removal of stopwords, removal of punctuation, and lemmatization. The methods used in these pipelines do not consider the order of words, only the presence or absence of words (a bag of words). Since there is punctuation in all sentences, this data does not add value in these traditional methods and just makes the process go slower.
- In modern pipelines these additional operations would remove valuable data from the input, as the order of words is important.
- Example:
  - Input to normalization: Let’s Make some cóokies!
  - Output of normalization: let’s make some cookies!

Outline I

Outline II

Pre-tokenization

- During pre-tokenization the sequence of words is split up into words.
- Some pre-tokenizers split on whitespace. Others split on whitespace and punctuation. Others have different rules (e.g., extra whitespace is also encoded).
- Example of splitting on whitespace and punctuation:
  - Input to pre-tokenzation: let's make some cookies!
  - Output of pre-tokenization: [let, 's, make, some, cookies, !]

Outline I

Outline II

Tokenization Algorithms

- Word-based tokenization
- Character-based tokenization
- Subword tokenization
  - Byte-Pair Encoding tokenization (BPE) (as used in GPT-2)
  - WordPiece tokenization (as used in BERT)
  - SentencePiece or Unigram tokenization (as used in T5)

Word-based tokenization

- There are no additional steps beyond what we did in pre-tokenization.
- Example:
  - Input to word-based tokenization: [let, 's, make, some, cookies, !]
  - Output of word-based tokenization: [let, 's, make, some, cookies, !] (same as input)
- Advantage of word-based tokenization:
  - Short sequences. A lot of information is captured in a word (as opposed to a letter), so the sequence we have to feed to the LLM will be relatively shorter than if we used character level or subword level tokenization.
- Disadvantages of word-based tokenization:
  - Slow training and inference. Word-based tokenization results in large vocabularies. There are over 500,000 words in the English language. Each word receives an ID, starting from 0 and going up to the size of the vocabulary. The LLM will use these IDs to identify each word, and such a large lookup table results in slow lookups. Example of how inefficient this is: *cat* and *cats* are different words and the model will initially not know that they have similar meanings. Another example is *buy* and *buying*. It will require more training time to learning the model.
  - Continued on next slide

Word-based tokenization

- Disadvantages of word-based tokenization (continued from previous slide):
    - Low-accuracy models. We may need a custom token to represent words that are not present in our vocabulary, known as the UNK (unknown) token. The UNK token is a fallback strategy. Each time an unknown word is used as input to the tokenizer, the tokenizer will not be able to find a sensible representation (embedding) for the word and will fall back to selecting a catchall, shared, representation. This shared representation is the embedding that represents all unknown tokens. For example, if the words *cat* and *dog* are both not present in our vocabulary, and we would use these words in our prompt, the model would not be able to differentiate between them. The model would produce text such as 'John came home, finding his barking cat standing in the doorway wagging his tail with his leash on the floor in front of him.'

Character-based tokenization

- Example:
    - Input to character-based tokenization: [let, ’s, make, some, cookies, !]
    - Output of character-based tokenization: [l, e, t, ’, s, m, a, k, e, s, o, m, e, c, o, o, k, i, e, s, !]
- Splitting the text into characters has two main advantages:
    - A very small vocabulary, resulting in faster training and inference because lookup is very fast.
    - Far fewer unknown (out-of-vocabulary) tokens, because every word can be constructed from characters.
- Disadvantages:
    - Long sequences. Compared to words, characters capture less meaning: The LLM will have to process many more tokens per sentence.

Subword tokenization

- Frequently used words are treated as whole words; they are not split into subwords.
- Infrequently used words are split.
- Example of subword tokenization: promotion, expansion, constitution, abbreviation, promote, expanse, constitute, abbreviate
  - word-based tokenization: 8 tokens
  - subword-based tokenization: 6 tokens (promot, expans, constitut, abbreviat, ion, e)
- Example:
  - Input to subword tokenization: [let, 's, make, some, cookies, !]
  - Output of subword tokenization: [let, 's, make, some, cook, ies, !]
- Subword tokenization balances
  - the size of vocabularies,
  - number of out-of-vocabulary tokens, and
  - the length of sequences that need to be fed into the LLM.
- Subword tokenization is the dominant tokenization method

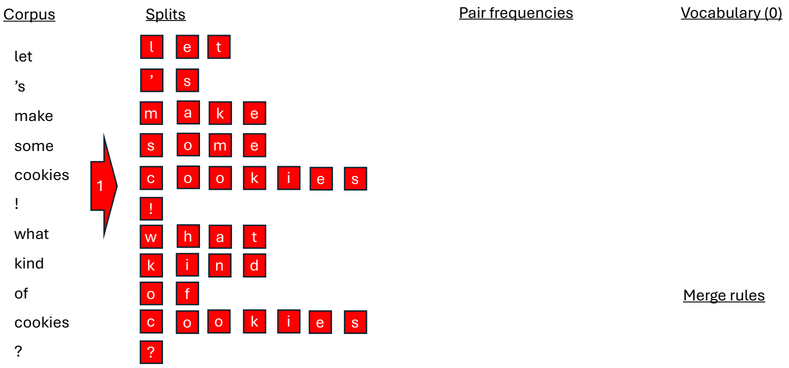
Byte-Pair Encoding tokenization

- Byte-Pair Encoding (BPE) originally was an algorithm to compress texts, and then OpenAI used it for tokenization of their GPT model.
- The original algorithm directly used bytes, but when it is used in natural language processing, it uses characters. There is also a byte-level BPE algorithm.
- BPE starts from a small vocabulary and learns merge rules and a vocabulary of a desired size. It merges tokens of the most common pairs.
- In the next slides we will explore this algorithm in depth.

Byte-Pair Encoding tokenization

Step 1: split up all the words in the pre-tokenized corpus

1



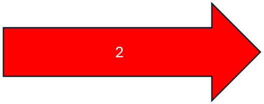
Byte-Pair Encoding tokenization

Step 2: gather unique splits and add to vocabulary

2

Corpus	Splits						
let	l	e	t				
’s	’	s					
make	m	a	k	e			
some	s	o	m	e			
cookies	c	o	o	k	i	e	s
!	!						
what	w	h	a	t			
kind	k	i	n	d			
of	o	f					
cookies	c	o	o	k	i	e	s
?	?						

Pair frequencies



Vocabulary (18)

a	i	o
’	h	!
d	e	k
s	m	f
?	l	w
c	t	n

Merge rules

Byte-Pair Encoding tokenization

Step 3: Calculate frequences of all possible pairs in the splits

3

Corpus	Splits						
let	l	e	t				
’s	’	s					
make	m	a	k	e			
some	s	o	m	e			
cookies	c	o	o	k	i	e	s
!	!						
what	w	h	a	t			
kind	k	i	n	d			
of	o	f					
cookies	c	o	o	k	i	e	s
?	?						

3

Pair frequencies

l + e : 1	k + i : 3
e + t : 1	i + e : 2
’ + s : 1	e + s : 2
m + a : 1	w + h : 1
a + k : 1	h + a : 1
k + e : 1	a + t : 1
s + o : 1	i + n : 1
o + m : 1	n + d : 1
m + e : 1	o + f : 1
c + o : 2	
o + o : 2	
o + k : 2	

Vocabulary (18)

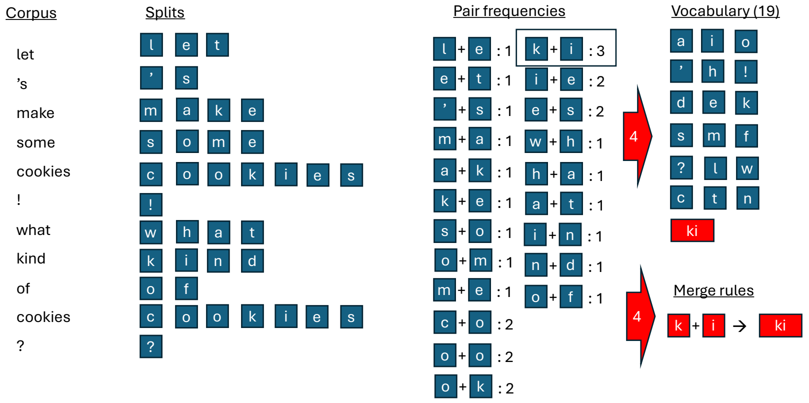
a	i	o
’	h	!
d	e	k
s	m	f
?	l	w
c	t	n

Merge rules

Byte-Pair Encoding tokenization

4

Step 4: Collect most frequent pair. If there is a tie, then take the first encountered one. Then store the combined version of the pair in the vocabulary and store the merge rule.



Corpus	Splits
let	l e t
’s	’ s
make	m a k e
some	s o m e
cookies	c o o k i e s
!	!
what	w h a t
kind	k i n d
of	o f
cookies	c o o k i e s
?	?

Pair frequencies

l + e	: 1	k + i	: 3
e + t	: 1	i + e	: 2
’ + s	: 1	e + s	: 2
m + a	: 1	w + h	: 1
a + k	: 1	h + a	: 1
k + e	: 1	a + t	: 1
s + o	: 1	i + n	: 1
o + m	: 1	n + d	: 1
m + e	: 1	o + f	: 1
c + o	: 2		
o + o	: 2		
o + k	: 2		

Vocabulary (19)

a	i	o
’	h	!
d	e	k
s	m	f
?	l	w
c	t	n
ki		

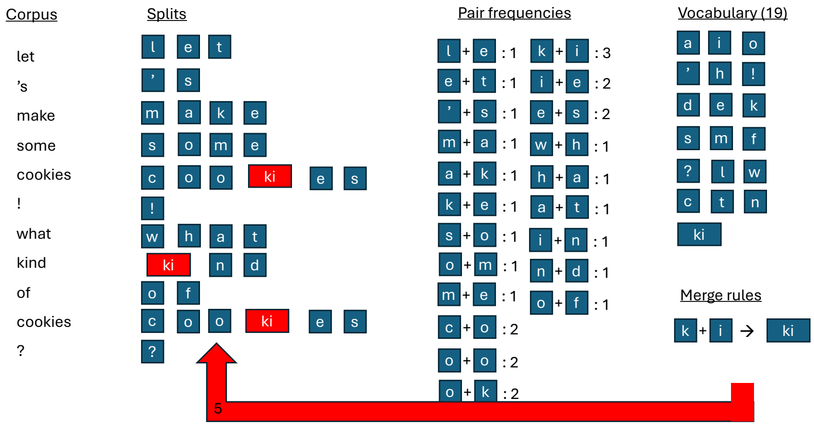
Merge rules

$k + i \rightarrow ki$

Byte-Pair Encoding tokenization

Step 5: Apply the merge rule to the splits.

5



Corpus	Splits
let	l e t
’s	’ s
make	m a k e
some	s o m e
cookies	c o o ki e s
!	!
what	w h a t
kind	ki n d
of	o f
cookies	c o o ki e s
?	?

Pair frequencies

l + e : 1	k + i : 3
e + t : 1	i + e : 2
’ + s : 1	e + s : 2
m + a : 1	w + h : 1
a + k : 1	h + a : 1
k + e : 1	a + t : 1
s + o : 1	i + n : 1
o + m : 1	n + d : 1
m + e : 1	o + f : 1
c + o : 2	
o + o : 2	
o + k : 2	

Vocabulary (19)

a i o ’ h ! d e k s m f ? l w c t n ki

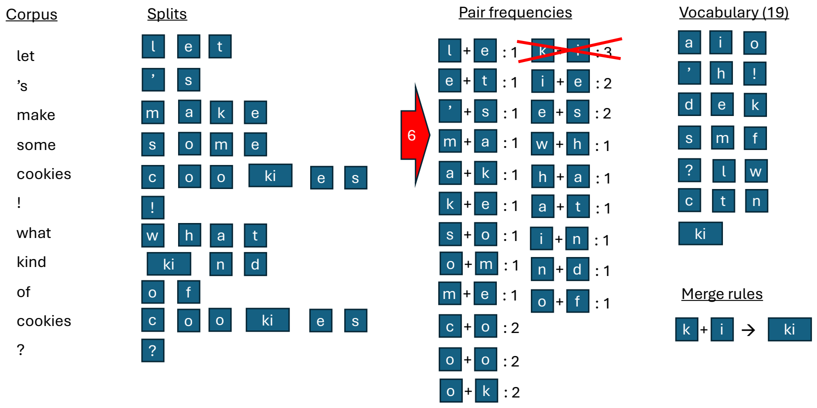
Merge rules

k + i → ki

Byte-Pair Encoding tokenization

We then repeat the process until a given vocabulary size has been reached.
Let's go on until we reach vocabulary size 21. Step 6: calculate the pair frequencies.

6



Corpus	Splits
let	l e t
's	' s
make	m a k e
some	s o m e
cookies	c o o ki e s
!	!
what	w h a t
kind	ki n d
of	o f
cookies	c o o ki e s
?	?

Pair frequencies

l + e : 1	~~k + i : 3~~
e + t : 1	i + e : 2
' + s : 1	e + s : 2
m + a : 1	w + h : 1
a + k : 1	h + a : 1
k + e : 1	a + t : 1
s + o : 1	i + n : 1
o + m : 1	n + d : 1
m + e : 1	o + f : 1
c + o : 2	
o + o : 2	
o + k : 2	

Vocabulary (19)

a	i	o
'	h	!
d	e	k
s	m	f
?	l	w
c	t	n
ki		

Merge rules

$k + i \rightarrow ki$

Step 6 is continued on the next slide.

Byte-Pair Encoding tokenization

7

We then repeat the process until a given vocabulary size has been reached.
Let’s go on until we reach vocabulary size 21. Step 6: calculate the pair frequencies.

Corpus	Splits
let	l e t
’s	’ s
make	m a k e
some	s o m e
cookies	c o o ki e s
!	!
what	w h a t
kind	ki n d
of	o f
cookies	c o o ki e s
?	?

6

Pair frequencies

l + e : 1	ki + e : 2
e + t : 1	ki + n : 1
’ + s : 1	i + e : 2
m + a : 1	e + s : 2
a + k : 1	w + h : 1
k + e : 1	h + a : 1
s + o : 1	a + t : 1
o + m : 1	i + n : 1
m + e : 1	n + d : 1
c + o : 2	o + f : 1
o + o : 2	
o + k : 2	

Vocabulary (19)

a	i	o
’	h	!
d	e	k
s	m	f
?	l	w
c	t	n
ki		

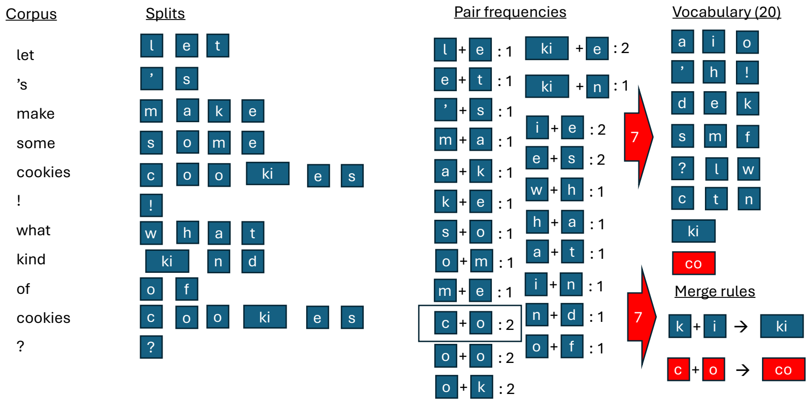
Merge rules

k + i → ki

Byte-Pair Encoding tokenization

8

Step 7: Collect most frequent pair. If there is a tie, then take the first encountered one. Then store the combined version of the pair in the vocabulary and store the merge rule.



Corpus	Splits
let	l e t
’s	’ s
make	m a k e
some	s o m e
cookies	c o o ki e s
!	!
what	w h a t
kind	ki n d
of	o f
cookies	c o o ki e s
?	?

Pair frequencies

l + e : 1	ki + e : 2
e + t : 1	ki + n : 1
’ + s : 1	i + e : 2
m + a : 1	e + s : 2
a + k : 1	w + h : 1
k + e : 1	h + a : 1
s + o : 1	a + t : 1
o + m : 1	i + n : 1
m + e : 1	n + d : 1
$\boxed{c + o : 2}$	o + f : 1
o + o : 2	
o + k : 2	

Vocabulary (20)

a i o
’ h !
d e k
s m f
? l w
c t n
ki
co

Merge rules

k + i → ki
c + o → co

Byte-Pair Encoding tokenization

Step 8: Apply the merge rule to the splits.

9

Corpus	Splits				
let	l	e	t		
’s	’	s			
make	m	a	k	e	
some	s	o	m	e	
cookies	co	o	ki	e	s
!	!				
what	w	h	a	t	
kind	ki	n	d		
of	o	f			
cookies	co	o	ki	e	s
?	?				

8

Pair frequencies

l + e : 1	ki + e : 2
e + t : 1	ki + n : 1
’ + s : 1	i + e : 2
m + a : 1	e + s : 2
a + k : 1	w + h : 1
k + e : 1	h + a : 1
s + o : 1	a + t : 1
o + m : 1	i + n : 1
m + e : 1	n + d : 1
c + o : 2	o + f : 1
o + o : 2	
o + k : 2	

Vocabulary (20)

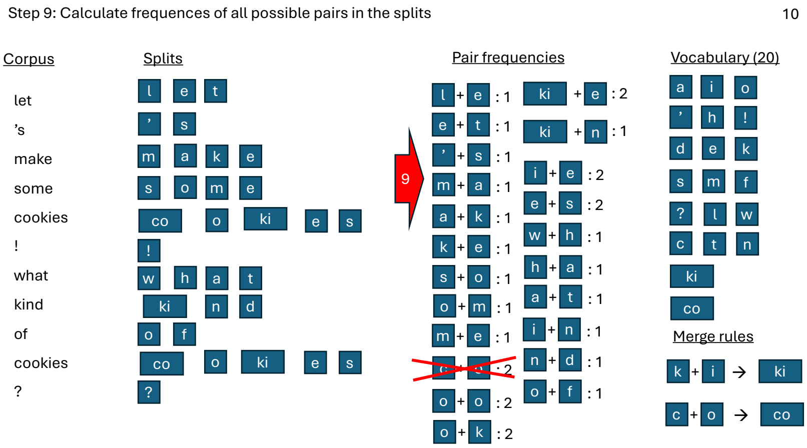
a	i	o
’	h	!
d	e	k
s	m	f
?	l	w
c	t	n
ki		
co		

Merge rules

$k + i \rightarrow ki$

$c + o \rightarrow co$

Byte-Pair Encoding tokenization



Step 9: Calculate frequences of all possible pairs in the splits

10

9

Corpus	Splits
let	l e t
’s	’ s
make	m a k e
some	s o m e
cookies	co o ki e s
!	!
what	w h a t
kind	ki n d
of	o f
cookies	co o ki e s
?	?

Pair frequencies

l + e : 1	ki + e : 2
e + t : 1	ki + n : 1
’ + s : 1	i + e : 2
m + a : 1	e + s : 2
a + k : 1	w + h : 1
k + e : 1	h + a : 1
s + o : 1	a + t : 1
o + m : 1	i + n : 1
m + e : 1	n + d : 1
~~c + o : 2~~	o + f : 1
o + o : 2	
o + k : 2	

Vocabulary (20)

a	i	o
’	h	!
d	e	k
s	m	f
?	l	w
c	t	n
ki		
co		

Merge rules

k + i → ki

c + o → co

Step 9 is continued on next slide.

Byte-Pair Encoding tokenization

Step 9: Calculate frequences of all possible pairs in the splits

11

Corpus	Splits
let	l e t
’s	’ s
make	m a k e
some	s o m e
cookies	co o ki e s
!	!
what	w h a t
kind	ki n d
of	o f
cookies	co o ki e s
?	?

9

Pair frequencies

l + e : 1	ki + e : 2
e + t : 1	ki + n : 1
’ + s : 1	i + e : 2
m + a : 1	e + s : 2
a + k : 1	w + h : 1
k + e : 1	h + a : 1
s + o : 1	a + t : 1
o + m : 1	i + n : 1
m + e : 1	n + d : 1
co + o : 2	o + f : 1
o + o : 2	
o + k : 2	

Vocabulary (20)

a i o
’ h !
d e k
s m f
? l w
c t n
ki
co

Merge rules

k + i → ki

c + o → co

Byte-Pair Encoding tokenization

12

Step 10: Collect most frequent pair. If there is a tie, then take the first encountered one. Then store the combined version of the pair in the vocabulary and store the merge rule.

Corpus	Splits
let	l e t
’s	’ s
make	m a k e
some	s o m e
cookies	co o ki e s
!	!
what	w h a t
kind	ki n d
of	o f
cookies	co o ki e s
?	?

Pair frequencies

l + e : 1	ki + e : 2
e + t : 1	ki + n : 1
’ + s : 1	i + e : 2
m + a : 1	e + s : 2
a + k : 1	w + h : 1
k + e : 1	h + a : 1
s + o : 1	a + t : 1
o + m : 1	i + n : 1
m + e : 1	n + d : 1
$\boxed{\text{co + o : 2}}$	o + f : 1
o + o : 2	
o + k : 2	

10

10

Vocabulary (21)

a	i	o
’	h	!
d	e	k
s	m	f
?	l	w
c	t	n
ki		
co	coo	

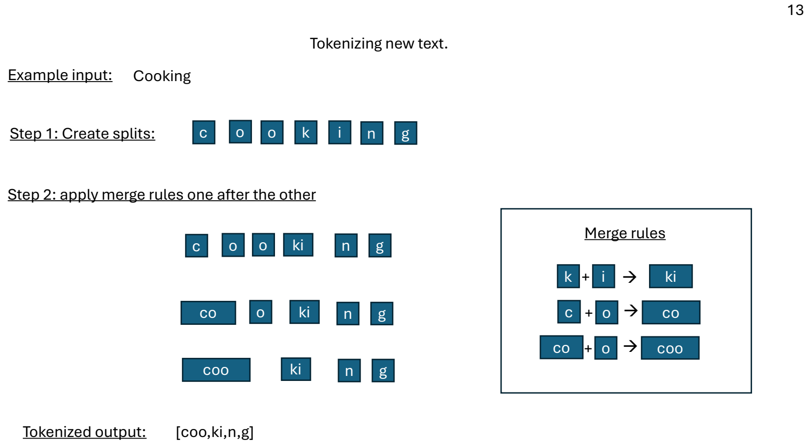
Merge rules

k + i → ki

c + o → co

co + o → coo

Byte-Pair Encoding tokenization


13

Tokenizing new text.

Example input: Cooking

Step 1: Create splits:

Step 2: apply merge rules one after the other

Tokenized output: [coo,ki,n,g]

WordPiece tokenization

- The WordPiece algorithm is similar to the BPE algorithm.
- A notable difference from the BPE algorithm is that instead of calculating pair frequencies, it calculates the following scores:

$$score = \frac{pair\ frequecy}{(frequency\ first\ element) \times (frequency\ second\ element)} \tag{9}$$

- By dividing the pair frequency by the product of the frequencies of the two elements, the algorithm prioritizes merging elements that are less frequent on their own than as a pair.
- For example, the elements *action* and *able* may be less likely to merge if *action* and *able* appear in many other words and have a high frequency.
- Further details of this algorithm are out of scope.

SentencePiece or Unigram tokenization

- Unigram tokenization works in the opposite direction of BPE and WordPiece: it starts from a large vocabulary and removes tokens until it reaches a given vocabulary size. We can obtain the initial vocabulary with any algorithm (e.g., BPE).
- At each training iteration the algorithm removes the token from the vocabulary that increases the loss over the corpus the least. The loss of the corpus is the sum of the losses of all the words. The loss function is the negative log likelihood.
- Example: the loss of the word *cookies* in the corpus

$$l(cookies) = freq(cookies) \times (-log(P(cookies))), \tag{10}$$

  - where $P(cookies) = P(cook) \times P(ies) = 0.5$ and $(-log(P(cookies))) = 0.69$. If we would remove *cook* from the vocabulary, and then
  - $P(cookies) = P(co) \times P(ok) \times P(ies) = 0.25$ and $(-log(P(cookies))) = 1.38$, we would thus be increasing the loss of cookies.
- Obviously this would also impact the loss of other words such as *cooking*. Every change to the vocabulary could potentially change the loss of several words, and therefore we have to recalculate the loss for the whole corpus. Since this is a costly operation, x% of the tokens are removed at the same time. Note that base characters are never removed.
- Further details of this algorithm are out of scope.

Outline I

Outline II

Post-processor

Post-processing involves

- adding special tokens that are required by the model (e.g., some models require a
  - BOS token (beginning of sentence),
  - EOS token (end of sentence),
  - and/or SEP token: separation token)),
- generating an attention mask (used during training on batches; all sequences in a batch need to be of the same length so padding is used; the attention mask allows the algorithm to ignore the padding), and
- generating token type IDs (some models require two different sequences to be encoded, and while these sequences are usually separated by a separation token, sometimes a binary mask is required by a particular model to identify the two sequences; e.g., the binary mask has all zeroes for the first sequence and all ones for the second sequence).

Outline I

Outline II

Token ID lookup

- The last step is to create a lookup table that has the tokens in the first column and the token IDs (also called input IDs) in the second column.
- The token IDs are then used by the embedding in the LLM.

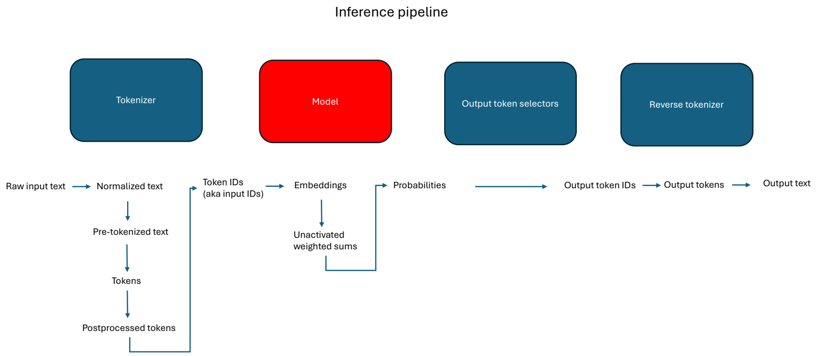
Summary

- Tokenization is foundational in helping LLMs learn.
- We have seen all the steps in tokenization, from normalization to token ID lookup.
- We have taken a deep dive in one tokenization algorithm: BPE
- Training a tokenizer: *tokenization.py*

Outline I

Outline II

Transformers



Outline I

Outline II

Attention Is All You Need

- 2017 paper by Google (`https://arxiv.org/pdf/1706.03762`) proposed the transformer architecture for a translation use case.
- Had an encoder and decoder. Since then, encoder only and decoder only (`https://arxiv.org/pdf/1801.10198`) models have been explored. Recently decoder only models are dominant.

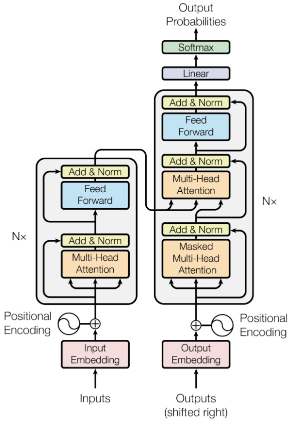
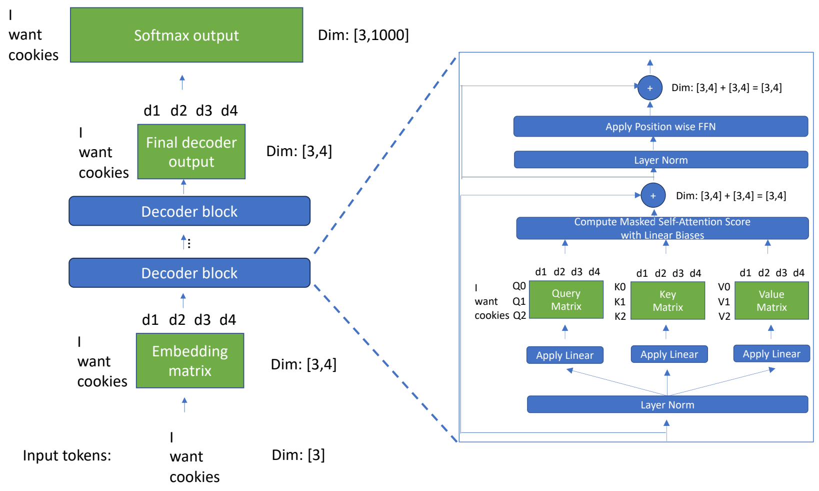


Figure 1: The Transformer - model architecture.

Outline I

Outline II

Decoder only transformer



Outline I

Outline II

Attention

- **Attention is a mechanism for pulling data from other rows and aggregating it into the current row. Since each token has its own row, attention allows each token to incorporate information from the other tokens.**
- Attention uses the concepts of **queries**, **keys** and **values** from database retrieval systems. When we type a query to search for some records in a database table, the search procedure will compare the query with the keys in a table and return the records that have a key equal to the query.

Attention

- Dot-product attention applies this look-up procedure mathematically. Before we study this, let's first note the following:
    - Attention computes how much each token should pay attention to the other tokens and then computes a linear combination of these records, resulting in a new column for each column of the token embedding. This is called self-attention because the tokens that are paying attention to each other are within one and the same sequence as opposed to another sequence.
    - We can do this multiple times. This only makes sense if we simply and slightly perturb the input matrix (e.g., through a linear dense layer) so that we get different results each time. We will receive one new column for each input column for each *lookup* and we can just concatenate these columns horizontally and use them as new features in the position-wise feed-forward neural network. This is called multi-head attention.

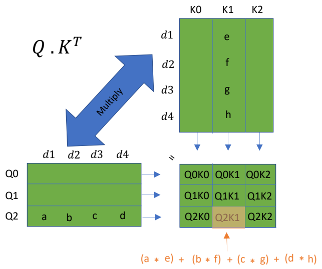
Attention

- The self-attention output

$$Z = softmax(QK^T)V, \tag{11}$$

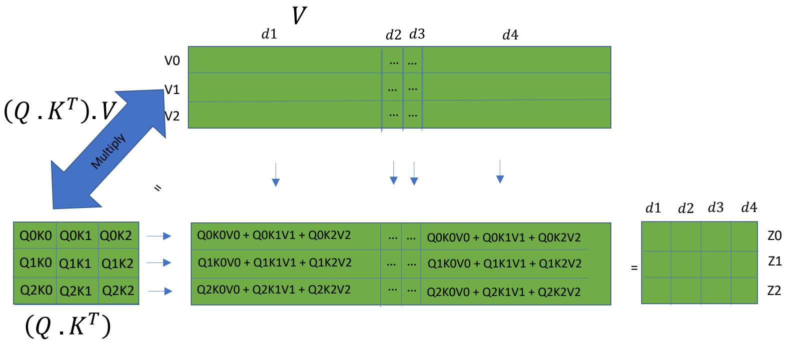
  where Q is the query matrix, K the key matrix and V the value matrix. Z is a matrix, with number of columns equal to the number of input columns.
- This particular form of attention is called dot-product attention, because we are applying the dot-product (matrix mutiplication between Q and the transpose of K).
- The softmax is applied to obtain weights that sum up to 1 to compute the weighted sum of the different records in the value matrix.
- In the following slides, let try to understand Equation 11. Without loss of generality we will be ignoring the softmax.

Attention



- Each row in the resulting matrix corresponds to a single query (i.e., record, time step).
- A column of a single row tells us how much a given query needs to pay attention to the key that corresponds to that column. We call it the attention score or attention weight.
- Remember that Q and K are actually the same matrix (slightly perturbed by the linear layers), so the matrix is paying attention to itself.
- The dot product tells us the similarity between Qi and Ki:
  - If two paired numbers that are multiplied (e.g., a and e) are both positive or negative, then the product will be positive $\rightarrow$ increases final sum
  - If the two paired numbers have opposite signs $\rightarrow$ decreases final sum
  - Thus, if the signs of the paired numbers are aligned we get a larger sum.

Attention



- The resulting matrix, Z, is the attention output for each query (i.e., record, token) returned by the attention module.
- A row in Z corresponds to a single query (i.e., token).
- The attention output for a given token (record) is the weighted sum of all the tokens (V), where the weights are the attention scores (QK^T)
- Attention rewrites each token as a mixture of the other tokens. Since each row corresponds to a token, attention pulls information from the other rows and aggregates it.

Attention

So what is the behavior we want from the model?

- We want the attention score to be high for two tokens that are relevant to each other. Therefore the vectors of each time step need to be (sign) aligned.
- We want the attention score to be low for two time steps that are unrelated to each other. Therefore the vectors of each token need to be (sign) misaligned.

How does the algorithm then produce the correct alignment?

- The algorithm adapts the weights of the linear dense layers, just like normal weights. It will learn the weights in such a way that the vectors of tokens that are relevant to each other are (sign) aligned and thus produce a high attention score.
- It will learn the weights in such a way that the vectors of tokens that are irrelevant to each other are (sign) misaligned and thus produce a low attention score.

Outline I

Outline II

Refinement: Scaling in Dot-Product Attention

- An attention score is a dot product:

$$q \cdot k = \sum_{i=1}^{d} q_i k_i$$

- Assume the components are roughly standardized:

$$\mathbb{E}[q_i k_i] = 0, \qquad \mathrm{Var}(q_i k_i) = 1$$

- Since variances add for independent terms:

$$\mathrm{Var}(q \cdot k) = d$$

- The corresponding standard deviation is:

$$\mathrm{sd}(q \cdot k) = \sqrt{d}$$

- We therefore scale dot products by their standard deviation:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V$$

Outline I

Outline II

Masking

- In some use cases, such as autoregressive generation, the model should only attend to tokens up to the current time step and not to future tokens.
- This is achieved by adding a mask matrix M to the attention scores:

$$Z = \text{softmax}(QK^T + M)V, \tag{12}$$

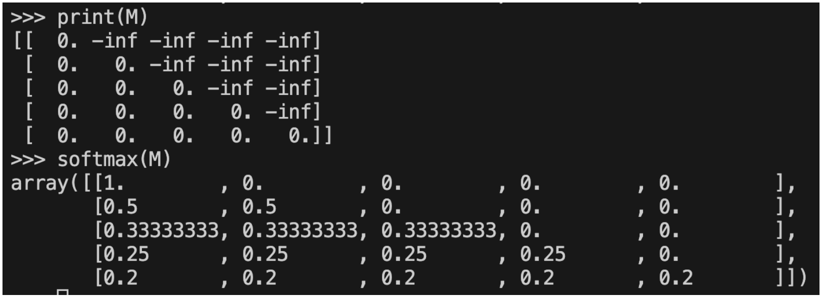
  where the softmax is applied row-wise.
- In our example, the causal mask M is:

```
#K   t0          t1        t2
M = np.array([
     [0.0,    -inf,    -inf],  # Qt0
     [0.0,     0.0,    -inf],  # Qt1
     [0.0,     0.0,     0.0]   # Qt2
], dtype='float32')
```

- Since $\text{softmax}(-\infty) = 0$, future tokens receive zero attention weight and are effectively ignored.

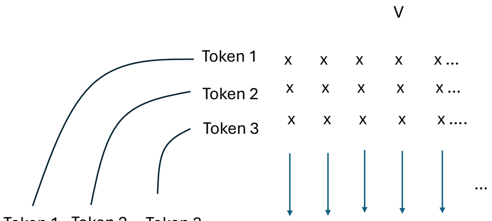
Masking

- Softmax(M) will have any particular token (from the perspective of the query, i.e., row) pay equal attention to all the other tokens that come before it.
- For example, token 4 (fifth row), pays 0.2 attention to itself, 0.2 attention to the token immediately preceding it, 0.2 to the token before that, ...



```
>>> print(M)
[[  0. -inf -inf -inf -inf]
 [  0.   0. -inf -inf -inf]
 [  0.   0.   0. -inf -inf]
 [  0.   0.   0.   0. -inf]
 [  0.   0.   0.   0.   0.]]
>>> softmax(M)
array([[1.        , 0.        , 0.        , 0.        , 0.        ],
       [0.5       , 0.5       , 0.        , 0.        , 0.        ],
       [0.33333333, 0.33333333, 0.33333333, 0.        , 0.        ],
       [0.25      , 0.25      , 0.25      , 0.25      , 0.        ],
       [0.2       , 0.2       , 0.2       , 0.2       , 0.2       ]])
```

Masking



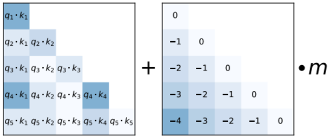
$$\text{softmax}(Q.K^T + M) \quad \begin{array}{ccccl} & \text{Token 1} & \text{Token 2} & \text{Token 3} & \\ \text{Token 1} & x & 0 & 0 & \rightarrow \text{Will only pay attention to token 1} \\ \text{Token 2} & x & x & 0 & \rightarrow \text{Will only pay attention to token 1 and 2} \\ \text{Token 3} & x & x & x & \rightarrow \text{Will only pay attention to token 1, 2 and 3} \end{array}$$

Outline I

Outline II

Attention with Linear Biases (ALiBi)

- Self-attention has no built-in notion of token order or distance.
- Previous versions of the transformer had positional embeddings, which would be added to embeddings to give the model information about token order and distance.
- A simpler, more efficient and better performing option is Attention with Linear Biases (ALiBi) (`https://openreview.net/pdf?id=R8sQPpGCv0`).

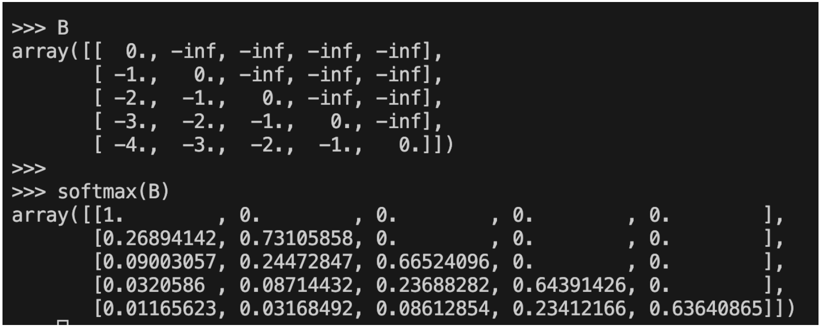


$q_1 \cdot k_1$
$q_2 \cdot k_1$ $q_2 \cdot k_2$
$q_3 \cdot k_1$ $q_3 \cdot k_2$ $q_3 \cdot k_3$
$q_4 \cdot k_1$ $q_4 \cdot k_2$ $q_4 \cdot k_3$ $q_4 \cdot k_4$
$q_5 \cdot k_1$ $q_5 \cdot k_2$ $q_5 \cdot k_3$ $q_5 \cdot k_4$ $q_5 \cdot k_5$
+
0
−1 0
−2 −1 0
−3 −2 −1 0
−4 −3 −2 −1 0
$\bullet m$

Figure 3: When computing attention scores for each head, our linearly biased attention method, ALiBi, adds a constant bias (right) to each attention score ($\mathbf{q}_i \cdot \mathbf{k}_j$, left). As in the unmodified attention sublayer, the softmax function is then applied to these scores, and the rest of the computation is unmodified. **m is a head-specific scalar** that is set and not learned throughout training. We show that our method for setting m values generalizes to multiple text domains, models and training compute budgets. When using ALiBi, we do *not* add positional embeddings at the bottom of the network.

Attention with Linear Biases (ALiBi).

- As you can see, softmax(B) is similar to the softmax(M), except that the token (query, in the row) will pay most attention to itself, less to the immediately preceding token, even less to the token before the immediately preceding token, . . .
- For example, token 4 (the fifth row) pays 0.63 attention to itself, 0.23 to the preceding token, 0.08 to the token before that, ...



```
>>> B
array([[  0., -inf, -inf, -inf, -inf],
       [ -1.,   0., -inf, -inf, -inf],
       [ -2.,  -1.,   0., -inf, -inf],
       [ -3.,  -2.,  -1.,   0., -inf],
       [ -4.,  -3.,  -2.,  -1.,   0.]])
>>>
>>> softmax(B)
array([[1.        , 0.        , 0.        , 0.        , 0.        ],
       [0.26894142, 0.73105858, 0.        , 0.        , 0.        ],
       [0.09003057, 0.24472847, 0.66524096, 0.        , 0.        ],
       [0.0320586 , 0.08714432, 0.23688282, 0.64391426, 0.        ],
       [0.01165623, 0.03168492, 0.08612854, 0.23412166, 0.63640865]])
```

Outline I

Outline II

Position-wise Feed-forward Neural Network

- The input to a Position-wise Feed-forward Neural Network is thus spread across several records. The word ‘position’ refers to the time step (token) in the sequence, and the word ‘position-wise’ refers to applying the feed-forward neural network to each time step separately and identically.
- How then can the neural network access data from previous time steps in any given time step? The answer is **attention**.
- **Attention** constructs a new variable per each input variable that contains data from the other time steps (i.e., records, tokens) that we used as input to the neural network as one batch of multiple records (i.e., time steps, tokens).

Outline I

Outline II

Training

- Next word prediction
- Loss function is cross entropy
- X is a matrix of vertically stacked tokens.
- Y is a matrix of vertically stacked one-hot encoded vectors (sparse variants are available).

Outline I

Outline II

Inference

- 1. Start from prompt: create vertically stacked tokens.
- 2. Get next word as output
- 3. Append the next word to the stacked embeddings
- 4. Repeat from step 2 until stopping criterion is reached (e.g., eos token or max length)

Outline I

Outline II

Layer Normalization

- Layer normalization normalizes activations *within a single token*, rather than across a batch, making it well suited for sequence models.
- For each token (row), we compute the mean μ and standard deviation σ across its hidden dimensions (columns), and normalize:

$$\hat{x} = \frac{x - \mu}{\sigma}$$

- This stabilizes training, but also reduces expressiveness if applied alone because pure normalization forces every token's hidden vector to have the same mean and variance, thereby destroying information carried in magnitude and offset that activation functions and linear layers rely on to behave differently.
- Therefore, layer normalization applies a *learned scale* γ and *learned offset* β:

$$\mathrm{LN}(x) = \gamma \hat{x} + \beta$$

- These parameters allow the model to choose the appropriate scale and center of the activations.
- If it is optimal to undo the normalization, the model can learn $\gamma = \sigma$ and $\beta = \mu$.

Outline I

Outline II

Embedding as the First Hidden Layer

- Consider a neural network with a vocabulary of size F.
- A token is represented as a one-hot *row vector*:

$$X \in \mathbb{R}^{1\times F}$$

- The first hidden layer is a linear (dense) layer with H hidden units:

$$W \in \mathbb{R}^{F\times H}$$

- The hidden activation is:

$$h = X.W \in \mathbb{R}^{1\times H}$$

- After training the full network, we can discard all subsequent layers.
- The vector h produced by this first hidden layer is the *embedding* of the token.

Embedding as a Lookup Operation

- If the one-hot vector X has a 1 in column i, then:

$$h = X.W = W[i,:]$$

- That is, multiplying by a one-hot vector simply selects the corresponding row of W.
- An embedding layer implements this operation directly as a *lookup*, rather than explicitly forming X and performing matrix multiplication.
- The matrix W contains the learned parameters; the embedding h is the output of the layer for a given token.