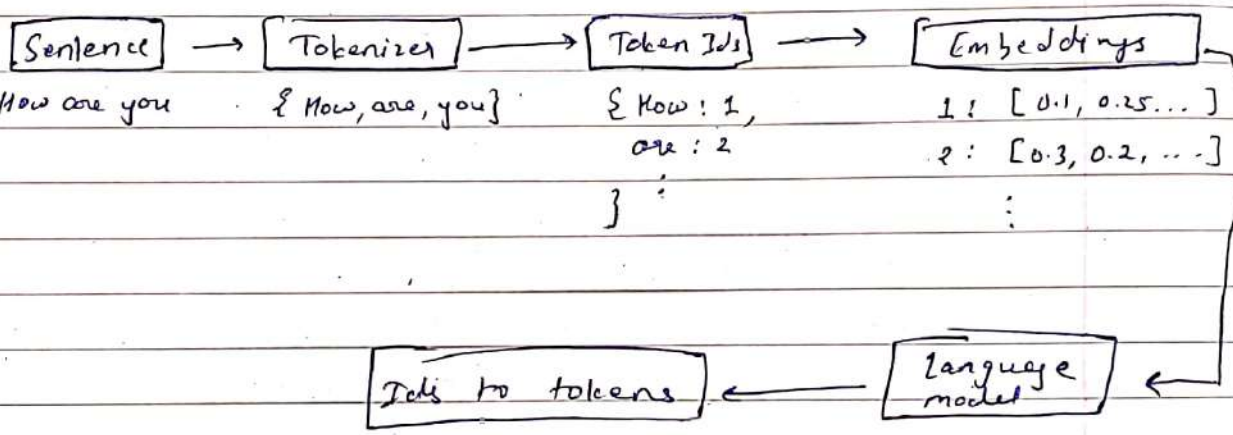
# * Tokenization

Given a sentence, breaking it down into words → sub-words and then constructing an embedding matrix that represents all of those words / sub-words (tokens) is called tokenization.

For LLMs following is the flow




→ Challenges with tokenization (we will try to deal with these in the approaches that we discuss)

- Picking optimal vocabulary size
- Out of Vocabulary
- Handling misspelled words
- Relationship between similar words (enjoy, enjoyed, enjoying)

→ Types of Tokenization

- Word level
- Sub-Word level
  - Byte Pair Encoding (BPE)
  - Word Piece
  - Sentence Piece
- Character level

Date: / /



- Word Level Tokenization

Given a sentence we pick all the space seperated unique words as our vocabulary.

Lets say insted of a sentence we have a huge text dataset that gives us 500k words

The size of vocab/embeddding matrix will be 500k x $d_{model}$ so with every word $d_{model}$ no. of parameters get added which is huge

Other problem is the output matrix is sequence length x 500k and then we do softmax for each row, so softmax is calculated over 500k size which is computationaly very expensive

- Character level Tokenization

If we do this we would have < 100 total tokens in our vocabulary as no. of characters are limited.

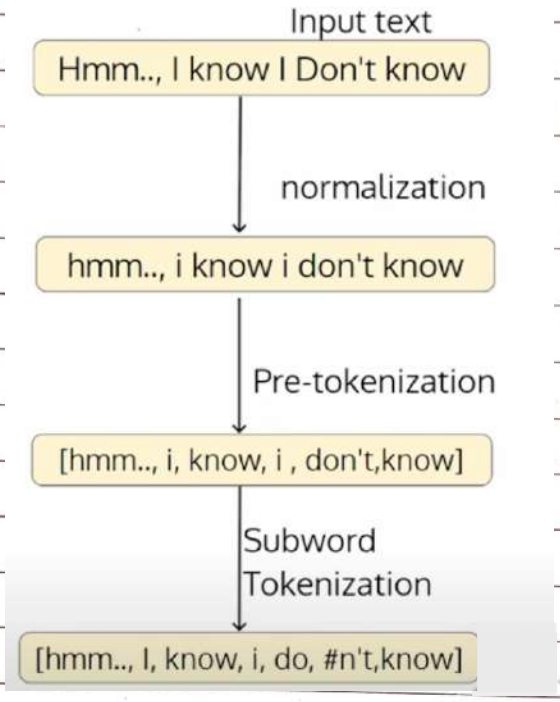
Here the problem would be sequence length, for more 5 words you would need to have approx sequence length of 25 A.dive to this all the weights & matrices in the encoder-decoder block also increase & computations also increase.

All the challenges remain as it is, we now need something in middle like sub-word tokenization. Niether word nor character.

Date: / /



- Sub-Word Tokenization




What sub-word tokenization will give us is the following

text data → I enjoyed playing cricket

vocab = [ I, enjoy, #ed, play, #ing, cricket ]

# means it is a split

this will be useful at the end, when model predicts words with # you merge it to previous word to get original sentence

How do you split data into tokens ?

- use ⇒ BPE, Word Piece, Sentence Piece

DATE:

# • Byte Pair Encoding (BPE)

Let's say we have the following dummy ~~vocabulary~~ textual data.

| low lower newest widest |
|---|

In real ~~world~~ world we might have a lot of words with meaningful sentences

**Step 1**

Break the sentence into words by splitting on space, remove special characters & make everything lowercase etc.

- low
- lower
- newest
- widest

} Following are all the unique words in the data

**Step 2**

Count the character frequencies (We start of with character & move towards word & stop somewhere in the middle)

| | | | | |
|---|---|---|---|---|
| l | - 2 | n | - 1 | This is also the initial vocabulary that we have. We keep adding to this |
| o | - 2 | s | - 2 | |
| w | - 4 | t | - 2 | |
| e | - 5 | i | - 1 | |
| r | - 1 | d | - 1 | |

**Step 3**

Now, starting from the first ~~word~~ entry/character in the above table start building pairs

DATE:

Merge 1

| | |
|---|---|
| [(l, o) - 2] | (e, w) - 1 |
| [(o, w) - 2] | [(e, s) - 2] |
| [(w, e) - 2] | [(s, t) - 2] |
| (e, r) - 1 | (w, i) - 1 |
| (n, e) - 1 | (i, d) - 1 |

Highlighted pairs have the maximum frequency, so we pick the first pair that occurred in the corpus ie (l, o)

Now, we do the merge operation on the selected pair & add 'lo' to the vocabulary. Since we are adding 'lo' we need to subtract the frequency from individual 'l' & 'o'?

Thus, Vocab looks like following

| | | |
|---|---|---|
| l - (2-2) = 0 | n - 1 | lo = 2 |
| o - (2-2) = 0 | s - 2 | |
| w - 4 | t - 2 | |
| e - 5 | i - 1 | |
| r - 1 | d - 1 | |

This is the completion of merge (1)

Merge 2

Pairs,

| | |
|---|---|
| [(lo, w) - 2] | (e, w) - 1 |
| [(w, e) - 2] | [(e, s) - 2] |
| (e, r) - 1 | [(s, t) - 2] |
| (n, e) - 1 | (w, i) - 1 |
| | (i, d) - 1 |

Ishan Modi

PAGE NO.:
DATE:

Note- we removed the pair (l, o) & (o, w) because we have a larger string that represents both. Also the frequency of 'l' & 'o' are zero, it means they do not exist independently if 'lo' is present

Merged,

| | | |
|---|---|---|
| l - 0 | n - 1 | lo - (2-2) = 0 |
| o - 0 | s - 2 | low - 2 |
| w - (4-2) = 2 | t - 2 | |
| e - 5 | i - 1 | |
| r - 1 | d - 1 | |

Merge 3 Pairs,

| | |
|---|---|
| (low, e) - 1 | (w, e) - 1 |
| (e, r) - 1 | [(e, s) - 2] |
| (n, e) - 1 | [(s, t) - 2] |
| (e, w) - 1 | (w, i) - 1 |
| | (i, d) - 1 |

Note - The order has changed because we started 'building pairs using 'low lower newest widest' using the latest vocabulary.

Merged,

| | | |
|---|---|---|
| l - 0 | n - 1 | lo - 0 |
| o - 0 | s - (2-2) = 0 | low - 2 |
| w - 2 | t - 2 | es - 2 |
| e - (5-2) = 3 | i - 1 | |
| r - 1 | d - 1 | |

DATE:

Merge 4

Pairs:

| | |
|---|---|
| (low, e) - 1 | (w, e) - 1 |
| (e, r) - 1 | [(es, t) - 2] |
| (n, e) - 1 | (w, i) - 1 |
| (e, w) - 1 | (i, d) - 1 |

Merged,

| | | |
|---|---|---|
| l - 0 | n - 1 | lo - 10 |
| o - 0 | s - 0 | low - 2 |
| w - 2 | t - (2-2) = 0 | es - (2-2) = 0 |
| e - 3 | i - 1 | est - 2 |
| r - 1 | d - 1 | |

Similarly you can continue with as many merges as you would like / till there are no more pairs to construct

In practice we have </w> characters following each word.

'low </w> lower </w> newest </w> widest </w>'

This helps us differentiate between

newest</w> & estate

Generally we do 30k - 50k merges this means vocab size = (30k to 50k) + initial 100 characters. The number of merges = vocab size because each merge adds an element to vocab

PAGE NO.:
DATE:

Step 4

Get the splits in the data using the vocabulary
'low' 'lower' 'newest' 'widest' (</w> is used in practice)

Lets just pick one word 'newest' for simplicity. We iterate through the vocab in the exact order

n | e | w | e | s | t — char based split

n | e | w | es | t — encountered 'es' so merged

n | e | w | est — encountered 'est' so merged

So the word 'newest' is broken / tokenized to 4 tokens 'n', 'e', 'w', 'est'

**BPE Algorithm →**

```python
import re
from collections import defaultdict

def get_stats(vocab):
    pairs = defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i], symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
    for word in v_in:
        w_out = p.sub(''.join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out

def get_vocab(data):
    vocab = defaultdict(int)
    for line in data:
        for word in line.split():
            vocab[' '.join(list(word)) + ' </w>'] += 1
    return vocab

def character_to_word_bpe(data, n):
    vocab = get_vocab(data)
    for _ in range(n):
        pairs = get_stats(vocab)
        best = max(pairs, key=pairs.get)
        vocab = merge_vocab(best, vocab)
    return vocab

# Example usage
corpus = "low lower newest widest"
data = corpus.split('.')
n = 4  # Number of merge operations
bpe_vocab = character_to_word_bpe(data, n)
print(bpe_vocab)
```

Ishan Modi

- Word Piece Tokenizer

Almost everything is similar to BPE, there is only one difference

If you recall in the 1st merge in BPE we had following pairs

(l, o) - 2
(o, w) - 2
⋮
(i, d) - 1

Here we there was a tie in the pre frequency of many pairs but we picked the 1st occurrence.

For word Piece, instead of 1st occurrence we use the following formula

$$\text{score} = \frac{\text{count}(\alpha, \beta)}{\text{count}(\alpha)\ \text{count}(\beta)}$$

we calculate this score for all the tied frequency (2) in this case I pick a value with maximum score

eg-

$$\text{score}(l, o) = \frac{\text{count}(l, o)}{\text{count}(l)\ \text{count}(o)}$$

$$= \frac{2}{2 \times 2} = \frac{1}{2}$$

score (o, w) = $\frac{2}{2 \times 4} = \frac{1}{4}$

Similar for all other same frequency

so, (l, o) & e.g. (s, t) have same score $\frac{1}{2}$

and same highest frequency 2. So if score is also same again we pick 1st ie - (l, o)

- **Sentence Piece Tokenizer**

Following is a small problem with BPE & wordpiece, that we want to address using sentence piece.

- lets say V = { h, e, l, l, o, he, el, lo, ll, hell }

here word 'hello', can be split as follows

h | e | l | l | o — 'he' occurred first so we merge

he | l | l | o — 'el' is not there so we ignore & 'll' occurred so we merge

he | l | ll | o — 'lo' occurred so we merge

he | l | lo } this is the tokenized value

- lets say V = { h, e, l, l, o, el, he, lo, ll, hell }

here word 'hello'

h | el | lo } This is the tokenized value

It greedily chooses the segment, and doesn't think about the best segmentation

Till now, we were starting off with characters & building words.

The idea of Sentences piece is to start of with a larger vocabulary & reduce it probabilistically.

Step 1

Construct a reasonably large seed vocabulary using BPE or extended suffix array algorithm

Step 2

E-step

Estimate the probability for every token in the given vocabulary using frequency counts in training corpus

Lets take a dummy example

Corpus - I love playing volleyball and football. I also love meeting people sharing ideas and learning from their experiences

Vocab from BPE

| | | | |
|---|---|---|---|
| I | volley | also | their |
| love | ball | meet | experiences |
| play | volleyball | people | |
| ing | and | shar | |
| vol | foot | ideas | |
| le | football | learn | |
| y | al | from | |

PAGE NO.:
DATE:

## Frequency in corpus

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| I - 2 | volley - 1 | also - 1 | their - 1 |
| love - 2 | ball - 2 | meet - 1 | experiences - 1 |
| play - 1 | volleyball - 1 | people - 1 | |
| ing - 4 | and - 2 | shar - 1 | |
| vol - 1 | foot - 1 | ideas - 1 | |
| le - 3 | football - 1 | learn - 1 | |
| y - 2 | al - 3 | from - 1 | |

Total Frequencies --> 35

## Probability

$$P(\text{token}) = \frac{\text{frequency of the token in the corpus}}{\Sigma \text{ frequencies of all the tokens in the corpus}}$$

Calculate the above for all tokens in the vocab.
For the sake of sanity we will only do for one

$$P(\text{ing}) = \frac{4}{35}$$

## Step 3 - M-Step

Use Viterbi Algorithm to segment the corpus & return optimal segment that maximizes the log likelihood

Pick all words from corpus one by one & start segmenting we will do it for one word.

Segment 1 → vol le y ball

$P(\text{volleyball}) = P(\text{vol})\ P(\text{le})\ P(\text{y})\ P(\text{ball})$

$\log(P(\text{volleyball})) = \log(P(\text{vol})) + \log(P(\text{le})) + \log(P(\text{y})) + \log(P(\text{ball}))$

$= \log\left(\frac{1}{35}\right) + \log\left(\frac{3}{35}\right) + \log\left(\frac{2}{35}\right) + \log\left(\frac{2}{35}\right)$

$= -5.09$

Segment 2 → Volley ball

$P(\text{volleyball}) = P(\text{volley})\ P(\text{ball})$

$\log(P(\text{volley})) = \log(P(\text{volley})) + \log(P(\text{ball}))$

$= \log\left(\frac{1}{35}\right) + \log\left(\frac{2}{35}\right)$

$= -2.79$

Segment 3 → Volleyball

$P(\text{Volleyball}) = P(\text{volleyball})$

$\log(P(\text{volleyball})) = \log(P(\text{volleyball}))$

$= \log\left(\frac{1}{35}\right)$

$= -1.54$

We pick and store the third segment that has only one token `volleyball` because it has the highest log likelihood

Ishan Modi

Similarly we pick up & store the segment for all words in the corpus.

At the end of this steps we would have optimal segments for all words

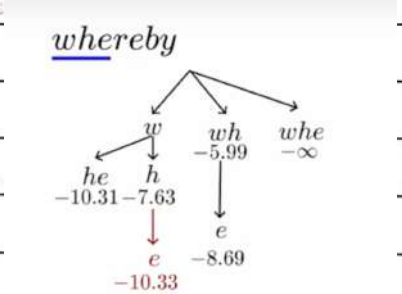
Now these segments become our vocab and we repeat steps 2 & 3. The only change in subsequent iterations here is that after step 2, we will remove x% of token that have smallest likelihoods

We do this until the vocab is reduced to the required size

There is still a question, how do we get segments for a given word in the corpus

→ Brute force - calculating for all possible segment would be computationally very expensive

→ Use trie & prune as you traverse




| Token | log(p(x)) |
|---|---|
| b | -4.7 |
| e | -2.7 |
| h | -3.34 |
| r | -3.36 |
| w | -4.29 |
| wh | -5.99 |
| er | -5.34 |
| where | -8.21 |
| by | -7.34 |
| he | -6.02 |
| ere | -6.83 |
| here | -7.84 |
| her | -7.38 |

Note- When it comes to tokenization there is something known as the fertility rate.

Fertility rate → Measure of at an average how many tokens each word is broken down into during tokenization

∴ Fertility rate → $\frac{\text{no. of Total tokens after tokenization of all words in the corpus}}{\text{Total no. of words in the corpus}}$

# * Positional Encoding

→ Till now, we have only looked at what is called absolute positional encoding (PE). Sinusoidal PE & learnable PE are two examples we have looked so far. What absolute PE does is that it looks at the entire sequence length and gives # tokens positions starting from 0 to end

→ There are two problems with this

- Lets say we have a five token sentence that starts at position 0 & ends at position 4.

Now, if we shift this sentence to position 10 to 14 the results were observed to change. The accuracy went down.

So there is a dependence of absolute position. This can be removed if we pick a word and give positions relative to that word eg - 0, 1, 2, -1, -2 with respect to a word.

- Second problem is the sequence length, if we have sequence length 512 & thus we learn 512 Positional embeddings.

we would not be able to deal with 1024 embeddings sequence length at inference. Predicting larger sequence won't be possible

Also if during the training all samples provided were less than 300 sequence length, model won't even generalize to sequence length above 300 at inference.

These are the reasons for us to use Relative Positional Encodings

→ Relative Positional Encodings

As we discussed earlier we take each word as a center point give it position zero & give & next words positions from 1 ... and previous words positions from -1 .....

| | somewhere | something | incredible | is | waiting | to | be | known |
|---|---|---|---|---|---|---|---|---|
| somewhere | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| something | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| incredible | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
| is | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 |
| waiting | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 |
| to | -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 |
| be | -6 | -5 | -4 | -3 | -2 | -1 | 0 | 1 |
| known | -7 | -6 | -5 | -4 | -3 | -2 | -1 | 0 |

PE

Now, earlier we just had 1 ~~embedding~~ for one token in absolute PE, which we added to token embedding

Here, you have T (sequence length) PE for one token that you add to token embedding.

- Speed

For Absolute PE we add 1 PE to 1 token embedding & we do it for T (sequence length) tokens. So speed is O(T)

For Relative PE uptill now we add T PE to 1 token embeddings & we do it for T tokens. So speed is $O(T^2)$

- Memory

For Absolute PE we have T PE

For Relative PE as seen in above figure -7 to 7 for sequence length 8, ie we have (2T - 1) PE

- Formula

For T = 8

$$h_3 = x_3 + (p_{-3} + p_{-2} + \dots + p_4)$$

$$h_i = x_i + \sum_{j=0}^{T-1} p_{j-i}$$

Now, in the next section we try to simplify the above

Lets look at how these PE look inside the transformer block computation:

attention matrix before softmax looks like following

$$e_{ij} = q_i k_j^T$$

recall that,

$$q_i = h_i W_Q = (x_i + p_i) W_Q$$

$$k_j = h_j W_K = (x_j + p_j) W_K$$

Thus,

$$e_{ij} = (x_i + p_i) W_Q W_K^T (x_j + p_j)^T$$

$$= (x_i W_Q + p_i W_Q)(W_K^T x_j^T + W_K^T p_j^T)$$

$$= \boxed{x_i W_Q W_K^T x_j^T} + x_i W_Q W_K^T p_j^T + p_i W_Q W_K^T x_j^T + \boxed{p_i W_Q W_K^T p_j^T}$$

- $x_i W_Q W_K^T x_j^T$: interaction between key word embedding and query word embedding. Attention without PE
- $x_i W_Q W_K^T p_j^T$ + $p_i W_Q W_K^T x_j^T$: correlation between word & positional embedding
- $p_i W_Q W_K^T p_j^T$: add constant that comes from PE

Research shows that the middle two terms don't contribute a lot to the overall attention score

What if we use the below formula during attention computation rather than adding PE at the begining

$$e_{ij} = x_i W_q \left( x_j W_K + P_{ij}^K \right)^T$$

($x_i$: $d$, $W_q$: $d \times d$, $x_j$: $d$, $W_K$: $d \times d$, $P_{ij}^K$: $d$)

$P_{ij}^K$ is also known as positional bias.

Note - The $d$ dimension that you see here is the dimension of head n such heads concatenate to form d model

Here in the above we have removed the addition part

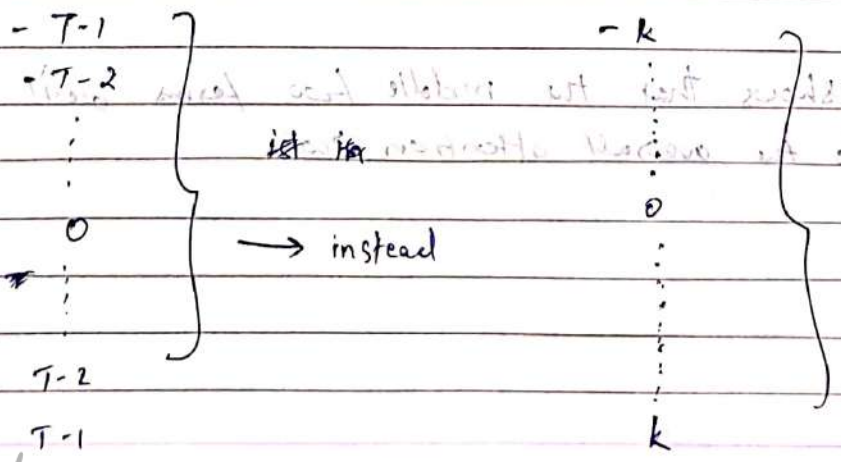
$$P_{ij}^K = P_{j-i}$$

($P_{j-i}$: $j-i$ → query word, $K$ → key word)

we are just picking one element from the previous summation formula

But, this still doesn't generalize to arbitrary lengths, what if we have larger sequence length at test time.

We do a simple hack, currently we have $2T-1$. PE



Thus even if we have more sequence length we clip it to k, so we will only have $2k+1$ PE

following is formula for clipping

$$P_{ij}^{K} = w^{K}_{clip(j-i,k)}$$

$$clip(j-i,k) = \max(-k, \min(j-i,k))$$

So the formula for attention becomes

$$e_{ij} = x_i W_Q (x_j W_K + P_{ij}^{K})^T$$

$$= x_i W_Q W_K^T x_j^T + x_i W_Q (P_{ij}^{K})^T$$

Then,

$$\alpha_{ij} = softmax(e_{ij})$$

Now, add relative PE for value matrix

$$z_i = \sum_j \alpha_{ij} (x_j W_V + P_{ij}^{V})$$

Note - $P_{ij}^{K}$, $P_{ij}^{V}$ are shared accross heads but not layers

at every layer we add seperate $P_{ij}^{K}$ & $P_{ij}^{V}$

## RPE Variations

recall the decomposition

$$e_{ij} = x_i W_Q W_K^T x_j^T + x_i W_Q W_K^T p_j^T + p_i W_Q W_K^T x_j^T + p_i W_Q W_K^T p_j^T$$

**Transformer-XL** (recursivevly extend context)

$$e_{ij} = x_i W_Q W_K^T x_j^T + x_i W_Q R_{j-i}^T + u W_K^T x_j^T + v W_K^T R_{j-i}^T$$

Where,

$R_{i-j}^T$ is the relative distance between the position $i$ and $j$ computed from sinusoidal function (non-learnable)

$u, v$ are learnable vectors.

**T5-bias**

All that is required is adding a constant, why not just do that?

$$e_{ij} = x_i W_Q W_K^T x_j^T + r_{j-i}$$

Where,

$r_{i-j} \in \mathbb{R}$ are learnable scalars, shared across layers. Maximum relative distance is clipped to

It also greatly reduces number of learnable parameters when the model size is scaled

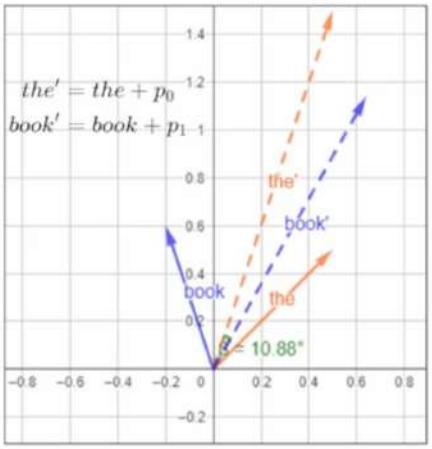
→ Rotary Positional Embeddings (ROPE)

Following are the relative PE we have seen so far

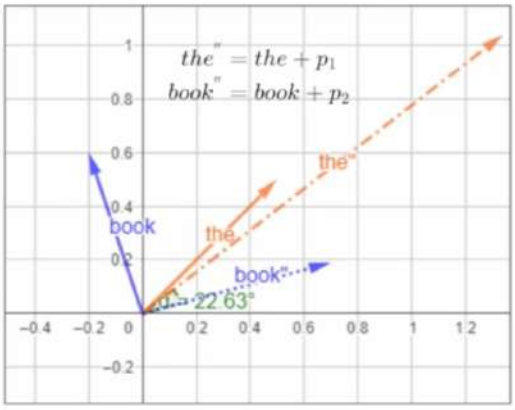
| | |
|---|---|
| Shaw | $e_{ij} = x_i W_Q W_K^T x_j^T + x_i W_Q (p_{ij}^K)^T$ |
| T-XL | $e_{ij} = x_i W_Q W_K^T x_j^T + x_i W_Q R_{j-i}^T + u W_K^T x_j^T + v W_K^T R_{j-i}^T$ |
| T5 | $e_{ij} = x_i W_Q W_K^T x_j^T + r_{j-i}$ |

There is still a small problem with relative PE, lets see it in detail

the book . . .




read the book . . .




As seen in the above images the relative distance between the words "the" & "book" is same in both sentences ie - they are next to one another.

But if we look at the vectors they are completely changed originally both images had same vectors

Now, $qk^T \Rightarrow q \cdot k$ is the operation happening here

$q \cdot k = |q||k| \cos\theta$, can we somehow keep $\theta$ same for these words is the idea behind rotary PE

Earlier, we had the following eqⁿ for relative PE

$$e_{ij} = x_i W_Q \cdot (x_j W_K + P^k_{ij})^T$$

$$= \underbrace{x_i W_Q W_K^T x_j^T}_{\downarrow} + \underbrace{x_i W_Q (P^k_{ij})^T}_{\downarrow}$$

we ignore this term

$$x_i W_Q \cdot (x_j W_K)^T$$

$$x_m W_Q e^{im\theta} \cdot (x_n W_K e^{in\theta})^T$$

(here we have replaced i, j with m, n because j also represents imaginary number)

$$\therefore e_{mn} = \underbrace{x_m W_Q e^{jm\theta}} \; \underbrace{(x_n W_K e^{jn\theta})^T}$$

now instead of transpose we use Hermitian transpose

Hermitian Transpose

$$A = \begin{bmatrix} 1 & -2-i & 5 \\ 1+i & i & 4-2i \end{bmatrix}$$

We first transpose the matrix:

$$A^T = \begin{bmatrix} 1 & 1+i \\ -2-i & i \\ 5 & 4-2i \end{bmatrix}$$

Then we conjugate every entry of the matrix:

$$A^H = \begin{bmatrix} 1 & 1-i \\ -2+i & -i \\ 5 & 4+2i \end{bmatrix}$$

$$\therefore e_{mn} = x_m W_Q e^{jm\theta} (x_n W_K e^{jn\theta})^H$$

$$= x_m W_Q (x_n W_K)^T e^{j(m-n)\theta}$$

This is all above about formulation, how to solve in real life

$$e^{jx\theta} = \cos(x\theta) + j\sin(x\theta)$$

$$e^{jx\theta} = \begin{bmatrix} \cos(x\theta) & -\sin(x\theta) \\ \sin(x\theta) & \cos(x\theta) \end{bmatrix}$$

replacing $x$ by $(m-n)$

$$e^{j(m-n)\theta} = \begin{bmatrix} \cos((m-n)\theta) & -\sin((m-n)\theta) \\ \sin((m-n)\theta) & \cos((m-n)\theta) \end{bmatrix}$$

Thus,

$$e_{mn} = \underbrace{x_m W_\theta \begin{bmatrix} \cos(m\theta) & -\sin(m\theta) \\ \sin(m\theta) & \cos(m\theta) \end{bmatrix}}_{} \underbrace{\begin{bmatrix} \cos(n\theta) & \sin(n\theta) \\ \sin(n\theta) & \cos(n\theta) \end{bmatrix}^T W_k^T x_n^T}$$

$$= x_m W_\theta \begin{bmatrix} \cos(m-n)\theta & -\sin(m-n)\theta \\ \sin(m-n)\theta & \cos(m-n)\theta \end{bmatrix} W_k^T x_n^T$$

$1\times 2 \quad 2\times 2 \quad 2\times 2 \quad 2\times 2 \quad 2\times 1$

Now if dimension increases beyon 2 it looks like following

$$e_{mn} = x_m W_\theta R_m \cdot (x_n W_k R_n)^T$$

$1\times d \quad d\times d \quad d\times d \quad 1\times d \quad d\times d \quad d\times d$

$$R_m = \begin{bmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \cdots & 0 & 0 \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \cdots & 0 & 0 \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos(m\theta_{d/2}) & -\sin(m\theta_{d/2}) \\ 0 & 0 & 0 & 0 & \cdots & \sin(m\theta_{d/2}) & \cos(m\theta_{d/2}) \end{bmatrix}$$

$$\left\{ \theta_i = 10000^{\frac{-2(i-1)}{d}}, \; i \in (1, 2, \ldots d/2) \right\}$$

Note - RoPE is applied in all layers individually but unlike relative RE it doesn't calculate or add PE to the value matrix, it is just calculated for key/Query transformation.

## → Attention With Linear Bias (ALiBi)

Following is the PE we saw with TS bias

$$q_i = x_i W_Q W_K^T x_j^T + r_{j-i}$$

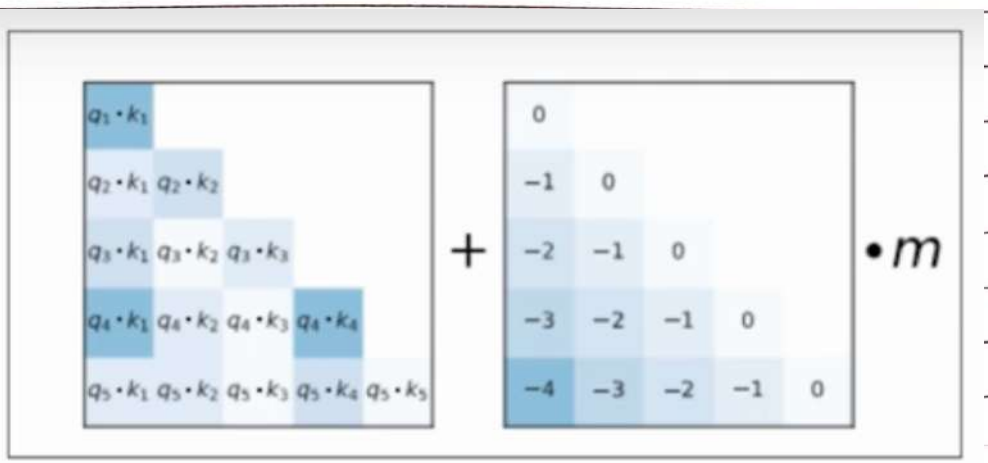
where $r_{j-i} \in R$ are learnable scalars

ALiBi makes these scalar fixed instead of learnable

- Formula

$$\text{softmax}(q_i K^T + m \cdot [-(i-1), \ldots -2, -1, 0])$$

- Understanding



Here,

$m$ is scalar that follows geometric progression

for $n=8$ heads, $m_i = \frac{1}{2^i}$, $i = 1, \ldots, n$

and since this is relative PE, this is applied at all layers individually.

# * Types of Attention Mechanisms

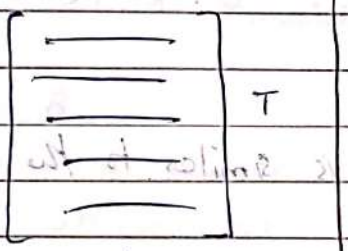
## → Time & Space complexity of Attention Mechanism

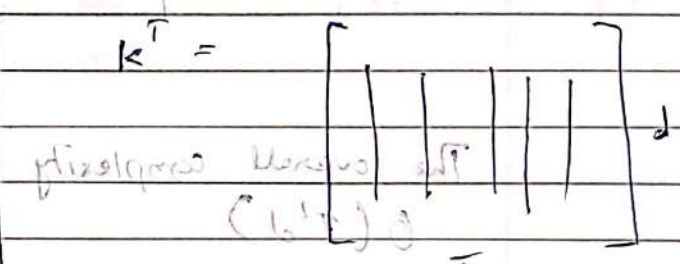
As seen in earlier parts, following is the formulation for attention

$$Attn = softmax\left(\frac{Q K^T}{\sqrt{d_k}}\right) . V$$

Time Complexity

- $Q K^T$

$Q$ = 


$K^T$ = 


Here T is number of tokens & d is the $d_{model}$ (size of each token embedding)

Lets pick a row of Q matrix & multiply by all columns of $K^T$

Here total $\boxed{(T.d)}$ multiplications happen. & $\boxed{T.(d-1)}$ additions happen

∴ Total complexity for one row = $\boxed{O(Td)}$

So for T rows of Q matrix

Overall Complexity = $\boxed{O(T^2 d)}$

- divide by $\sqrt{d_k}$ & then softmax

The entire operation of normalization has a complexity of $O(T^2)$

- Now, Attn. V $\longrightarrow$ softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)$

$A$ = $T \times T$ matrix, $V$ = $T \times d$ matrix

The overall complexity is similar to the first case
$O(T^2 d)$

$\therefore$ Total = $O(T^2 d) + O(T^2) + O(T^2 d)$

$\Rightarrow O(T^2 d)$.

Space Complexity

Let's say,

- B = Batch Size
- V = Number of tokens in Vocab
- T = Sequence length

- $d_{ff}$ = dimension of feed forward layer
- $d_{model}$ = dimension of input embedding
- $n_h$ = no. of heads
- N = No of model parameters.

Now,

Multi head Attention

- $(mha) = (Q_{T\times d} + K^T_{T\times d} + V_{T\times d})\, n_h$

$= (3(T\times d))\, n_h$

$= 3T \times n_h \times d$

$= 3T \times d_{model}$ $\quad \because n_h \times d = d_{model}$

For batch size B

$= 3BTd_{model}$

- $(Attn) = softmax(QK^T)$

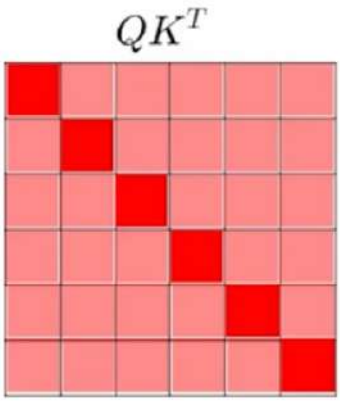
$= Bn_hT^2$

Total $= 3BTd_{model} + Bn_hT^2$

$\Rightarrow$ Thus it can be said that memory scales up linearly with batch size & quadratically with sequence length

→ Attention Mechanisms

(1) Full Attention

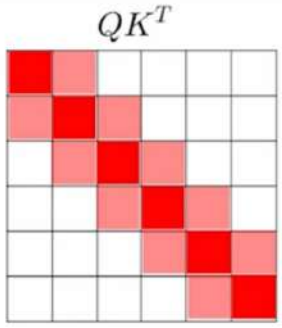



Here every word attends to itself and all the other words in the context

This time complexity is $O(T^2 d)$

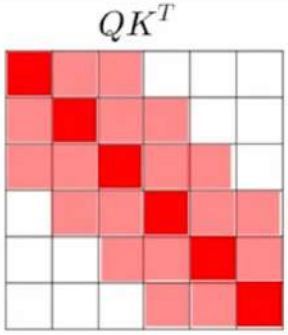
- $T$ → no. of tokens in the sequence
- $d$ → hidden dimension of the embedding

(2) Strided Local Attention




Here each word attends to itself & 2 adjacent neighbours, one before & one after, thus $c = 3$.

The complexity becomes linear $O(3Td)$

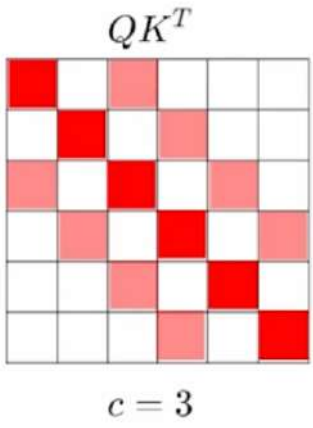



Similar to above but $c = 5$ in this case

The complexity is still linear $O(5Td)$

Note - These attentions are called local attentions because we are looking at / paying attention to only few words in the vicinity

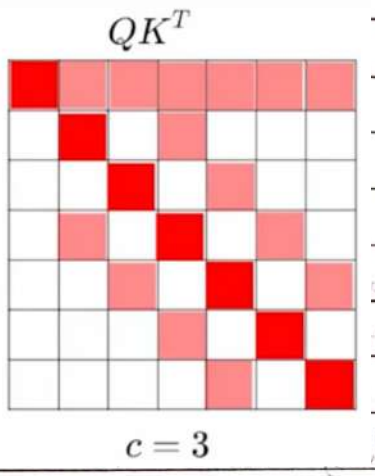
(3) Dialated Attention (hybrid local attention with dilations to capture broader context)




In this case we still look at adjacent neighbours, but pay attention to alternate words. Here $c=3$

The complexity is linear
$O(3Td)$

(4) Global + Local Attention




Here we can use any of the above local attention / dialated attention idea

But for some tokens we would also pay attention to all the other words

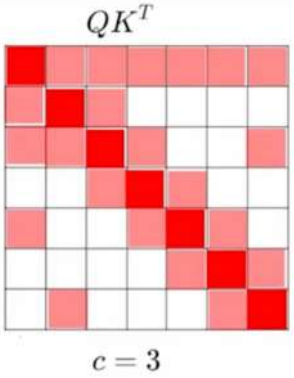
The complexity in this case will be linear
$O((3T + T)d + 3d)$

This can also be written as

$$O(3(T-1)d + Td)$$

local attention component — global attention component

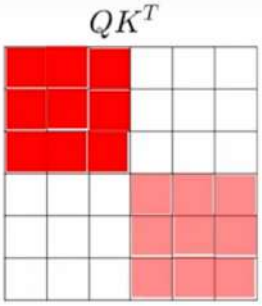
## (5) Global + Random + Local Attention



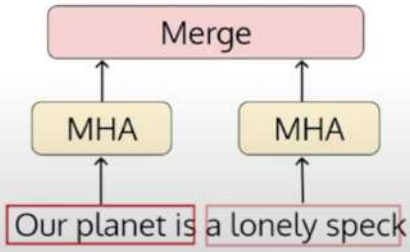

Similar to the previous, but for each word you pay attention to some random words, you may also not select any word, ie random words pick are between (0-T)

Complexity for the above is
$O((3T + T + T)d)$

## (6) Local Block Attention




Computational complexity $O(\frac{T^2}{2})$

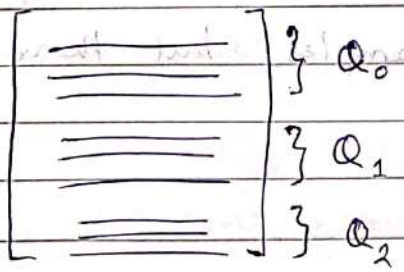



Here as we can see, we have divided the context length into 2 blocks

Thus, $n = 2$ in this case.
In the real word if $n = 2, 3, 4$
4 context length was 512
the block sizes would be
256, 128, 64

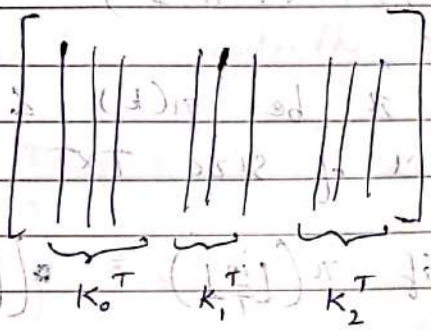
Lets understand in a bit more detail. Lets say we have the following matrices

Q =




$Q \in R^{T \times d_k}$

$K^T$ =




$K^T \in R^{d_k \times T}$

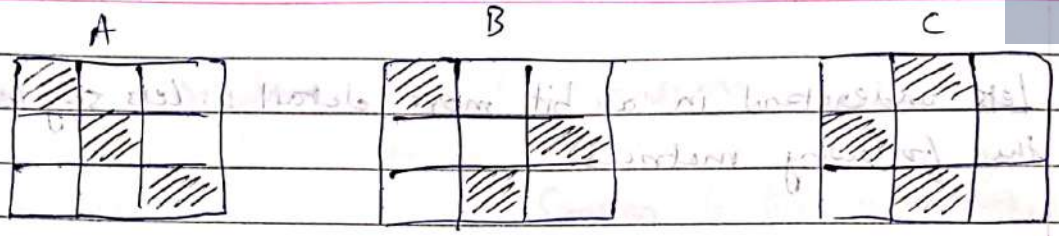
As seen earlier we divided both matrix Q & $K^T$ into n, blocks where n = 3, thus the attention matrix that will be formed will have $n^2$, ie 9 blocks

$Q K^T$

| | | |
|---|---|---|
| $Q_0 K_0^T$ | $Q_0 K_1^T$ | $Q_0 K_2^T$ |
| $Q_1 K_0^T$ | $Q_1 K_1^T$ | $Q_1 K_2^T$ |
| $Q_2 K_0^T$ | $Q_2 K_1^T$ | $Q_2 K_2^T$ |

Note - We will use similar blocks for V matric as K $Q_1 K_0^T$ will be multiplied by $V_0$ & same for others as well

Now, for each Q we can pick any K and this results into different combinations of the above metric




Above are some examples, but there can be n! such permutations

Lets formulate

$\pi$ = perm ( 0, 1, . . . . n-1 )

let $k^{th}$ element of $\pi$ be $\pi(k)$ &
masking matrix be M of size TXT

$$M_{ij} = \begin{cases} 1 & \text{if } \pi\left(\left\lfloor \frac{i}{T} \right\rfloor\right) = \pi\left(\left\lfloor \frac{j}{T} \right\rfloor\right) \\ 0 & \text{otherwise} \end{cases}$$

row number, col number

$$Z = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} \odot M\right)$$

$\odot$ : hadamard product

Now lets say

A has permutation $\pi = (0, 1, 2)$
B has permutation $\pi = (0, 2, 1)$
C has permutation, $\pi = (1, 0, 1)$

We will be creating mask matrix for all the above cases using the formula. The mask matrix obtained will have 1 inplace of highlighted regions & 0 otherwise

Note - In the figure of A, B, C the highlighted regions are actually blocks and in practice each block has multiple rows & colums. The masked matrix is not formed at a block level but at the row/col level.

Note - In practice ~~these~~ different such permutations are used in different attention heads to obtain maximum coverage.

Lets discuss the complexity for block local attention

$$= O\left(\frac{T}{n} \times \frac{T}{n} \times n \times d\right)$$

$$= O\left(\frac{T^2 d}{n}\right)$$

## (7) Linear Attention (Low Rank Approximation)

As we know, following is the formula for full attention

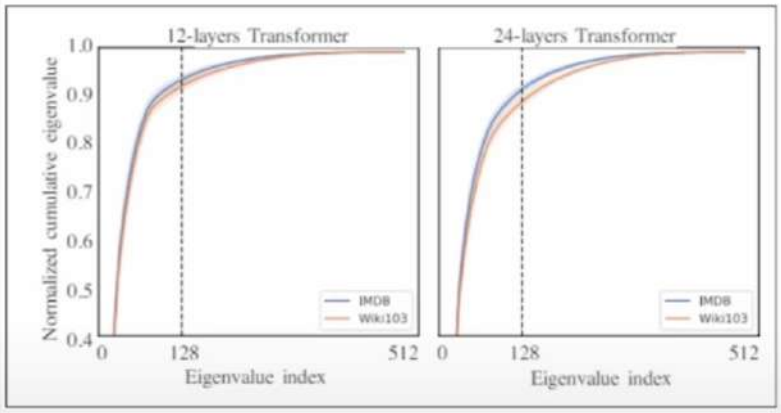
$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

$$Z = A.V$$

Lets, ignore Z for now. $A \in R^{T \times T}$ where T is the sequence length

Lets say $T = 512$, thus we have A which is $512 \times 512$ dimensional matrix.

We do SVD on this 512 X 512 matrix, this means we get 512 eigen values in diagonal matrix. normalizing these values in 0-1 scale such that all values add upto 1 & plotting cummulative eigen values




The graph clearly shows that top 128 eigen values contribute 90%, Thus a lower dimensional / ranked matrix can represent this 512x512 matrix.

$$Q \in \mathbb{R}^{T \times d} \quad K \in \mathbb{R}^{T \times d} \quad V \in \mathbb{R}^{T \times d}$$

Introduce two learnable linear projection matrices

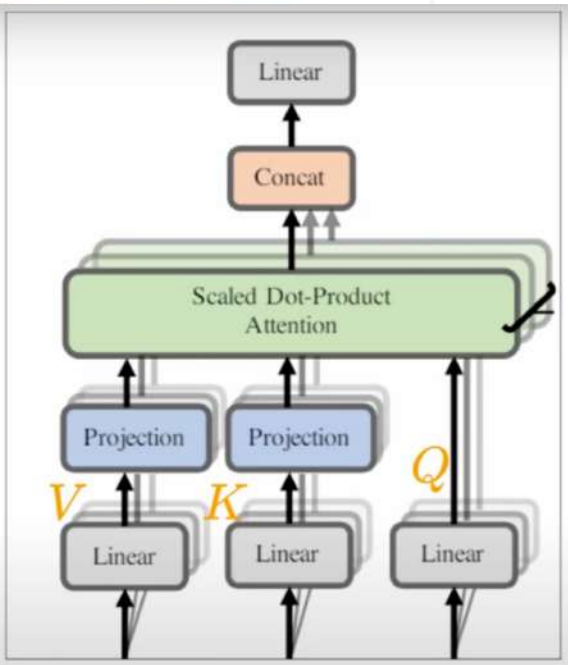
$$E, F \in \mathbb{R}^{k \times T}$$

$$A = softmax(\frac{Q(EK)^T}{\sqrt{d}})$$

$$A \in \mathbb{R}^{T \times k}$$

$$\boxed{O = AFV}$$

The computational complexity is $O(kTd)$, where $k$ is the rank and can be set according to the error $\epsilon$



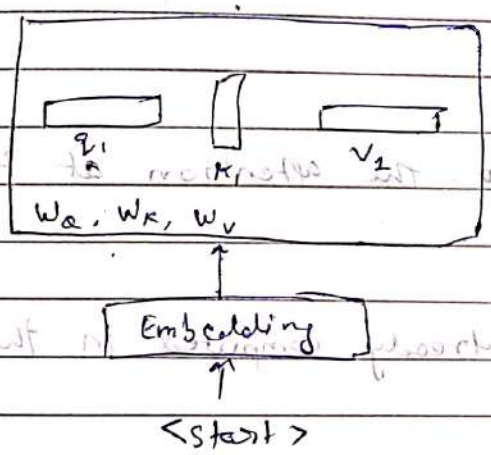
# * Inference Mechanisms

## → KV Cache

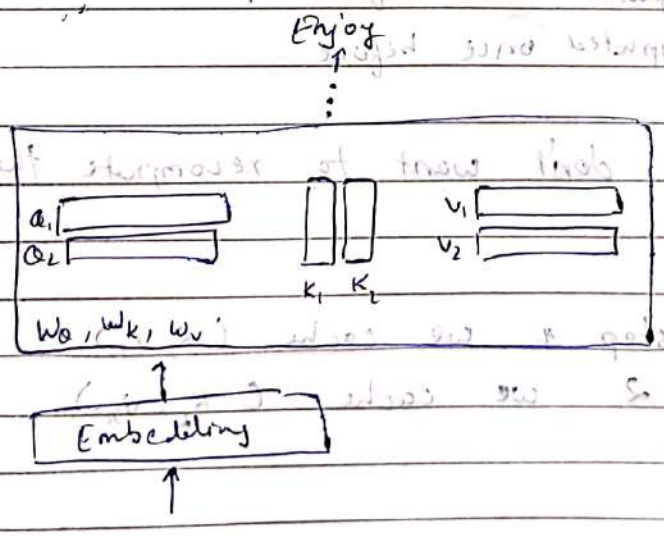
The concept of KV cache is mainly used in transformer decoder

Let's say we are auto-regressively predicting tokens at inference

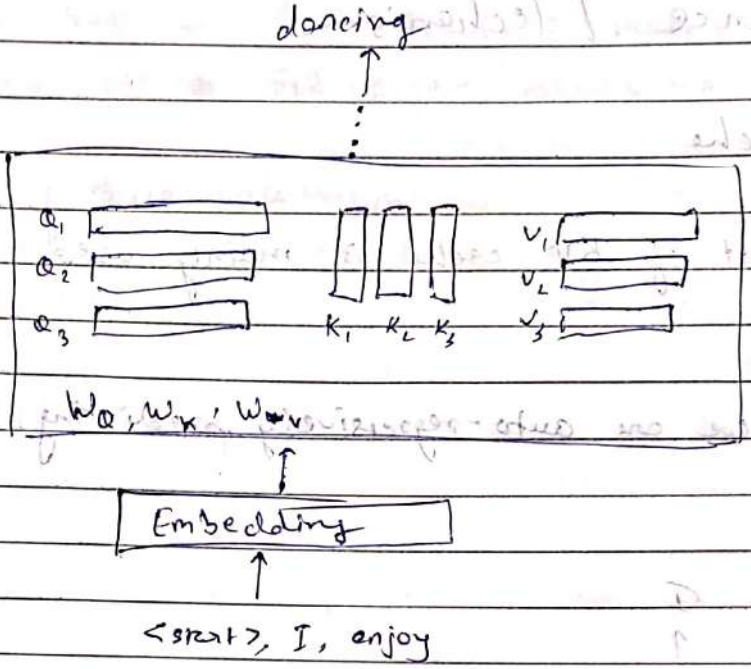
time step 1




time step 2



timestep 3




and so on.

Now, if we zoom into the attention at time step 3

- $K_1$, $K_2$, $V_1$, $V_2$ are already computed in the previous timesteps.

$(K_1, V_1)$ were computed twice before
$(K_2, V_2)$ were computed once before

idea is that we don't want to recompute them, so we cache them

So at timestep 1 we cache $(K_1, V_1)$
at timestep 2 we cache $(K_2, V_2)$

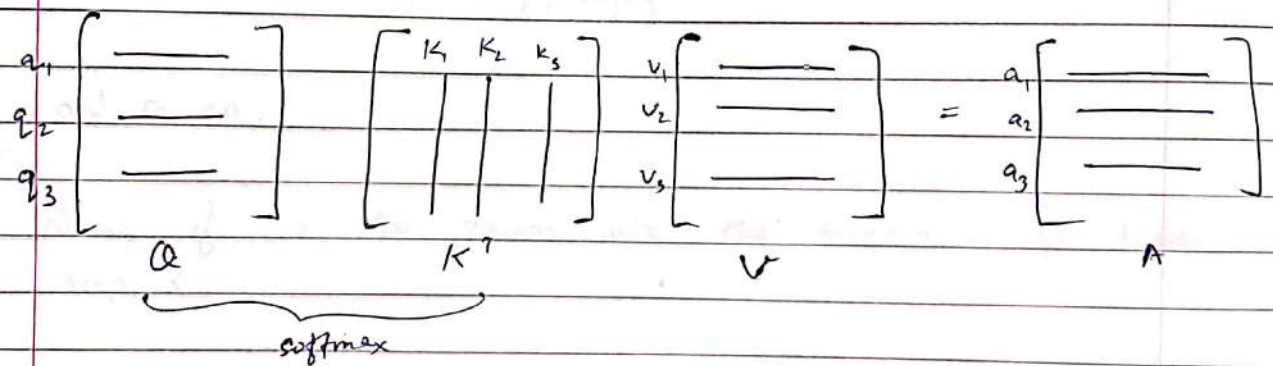
and so on ...

This is called KV cache

- Can we also do something for $q_1, q_2, q_3$ because they also keep consuming memory in addition to KV cache

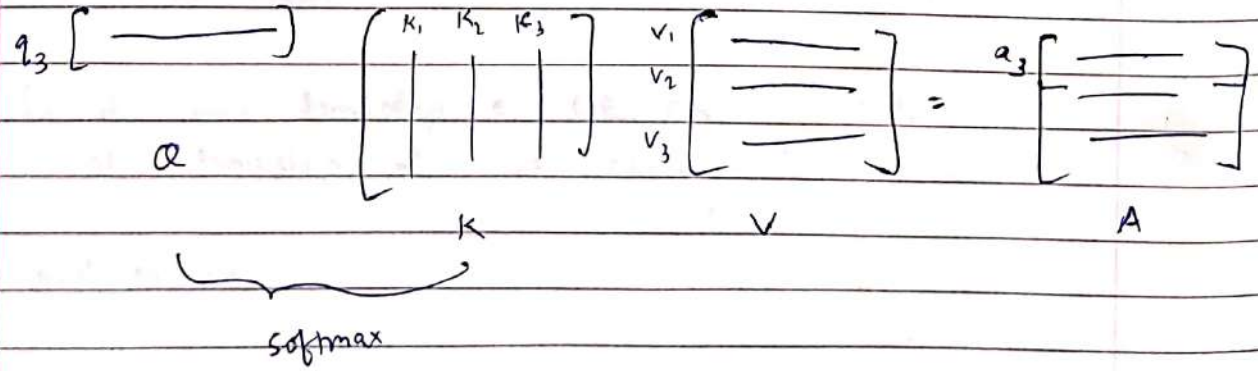
at time step 1 we have $q_1$
at time step 2 we have $q_1, q_2$
:
and so on

The answer is only use $q_i$ at $i^{th}$ time step
other q's are not needed




If we observe closely $a_3$ is the output of the timestep 3 it can also be called $q_4$ (dancing) in this case

So for computing $a_3$, we only need $K_1, K_2, K_3, V_1, V_2, V_3$



So KV cache reduces computation complexity from $O(n^2)$ to $O(n)$ but it increases memory requirement

Lets compute memory requirement for KV cache

KV cache per token in bytes =

$$2 \times \text{num\_layers} \times (\text{num\_heads} \times \text{dim\_head}) \times \text{precision\_in\_bytes}$$

- GPT-3

num_layers = 96
num_heads = 96
dim_head = 128
precision_in_bytes = 4 = 32 bit precision

KV cache per token = $2 \times 96 \times (96 \times 128) \times 4$
= 9.4 MB

if context length = 2000

KV cache = $2000 \times 9.4$ MB = 19.2 GB

This is huge, so we need some modifications to deal with this

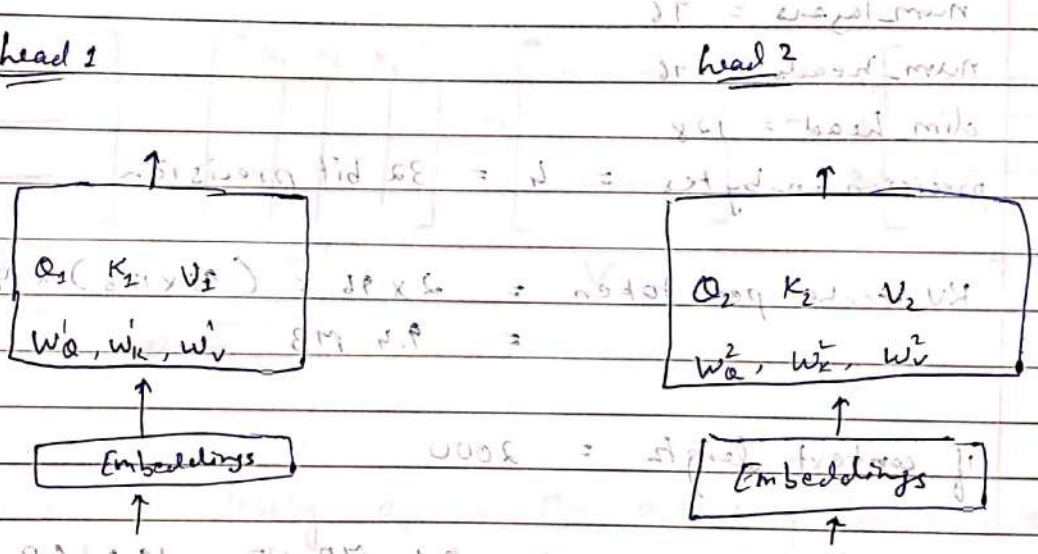
→ Multi-Query Attention (MQA)

KV cache per token in Bytes =

$$2 \times \text{num\_layers} \times (\text{num\_heads} \times \text{dim\_head}) \times \text{precision in bytes}$$

The idea is to share K,V across all heads & thus reduce num_heads to 1

Lets see with example




So here for KV weights $W_K'$, $W_V'$ & $W_K^2$, $W_V^2$ are different in different heads. Thus they produce different $K_1, V_1$ & $K_2, V_2$

If we share $W_K'$ & $W_K^2$, $W_V'$ & $W_V^2$ for all heads we can compute K, V only once and use it with $Q_i$'s in all heads.

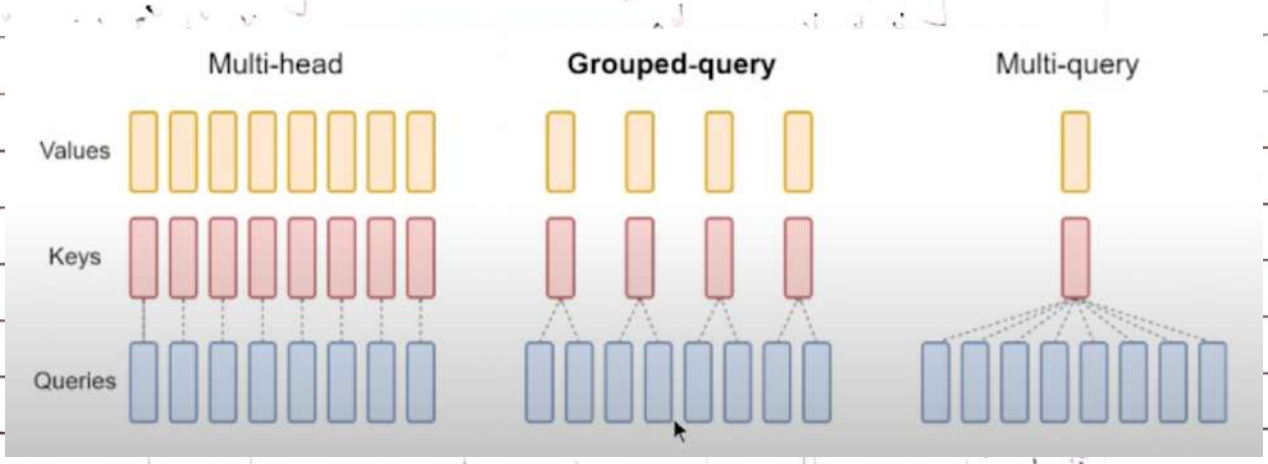
We are reducing parameters thus accuracy will degrade & we will also have to retrain of more data from training

→ Grouped Query Attention (GQA)

This is a ~~not~~ middle ground between, multi-head attention & multi-query attention.

The idea here is ~~to~~ to share KV weights & thus KV values for "n" number of queries

Thus we would have groups of queries working with different KV values.



Date: / /

* Decoding Strategies (these strategies are used during inference)

Deterministic
- Exhaustive Search
- Greedy Search
- Beam Search

Stochastic

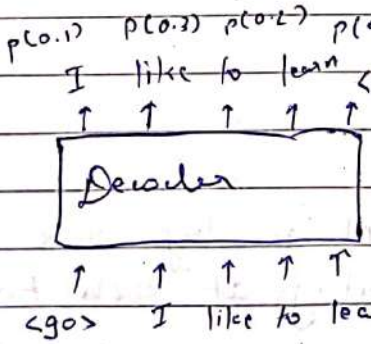
→ Deterministic Decoding (same output everytime)

- Exhaustive Search

Lets say we want a output from the decoder that contains 5 words. Our vocabulary size is 40k words. Then the total number of unique sequences that can be formed

$$|V|^5 \approx (\text{vocabulary size})^5 \approx 40{,}000^5$$

The number of permutations is huge & thus exhaustive search is practically not possible.




Example of how exhaustive search works
(for this sequence only we will end up running decoder 5 times)

Now, we calculate the probability of the above sequence

$$p = p(I)\ p(\text{like} \mid I)\ p(\text{to} \mid \text{like}, I) \ldots\ldots$$

$$= 0.1 \times 0.3 \times 0.2 \ldots\ldots \times 0.9 \times 0.7$$

Date: / /

Similarly we calculate the probabilities of all possible sequences and pick the sequence with highest probability

If we talk about time complexity you will end up running the transformer decoder $|V|^5$ times for this case

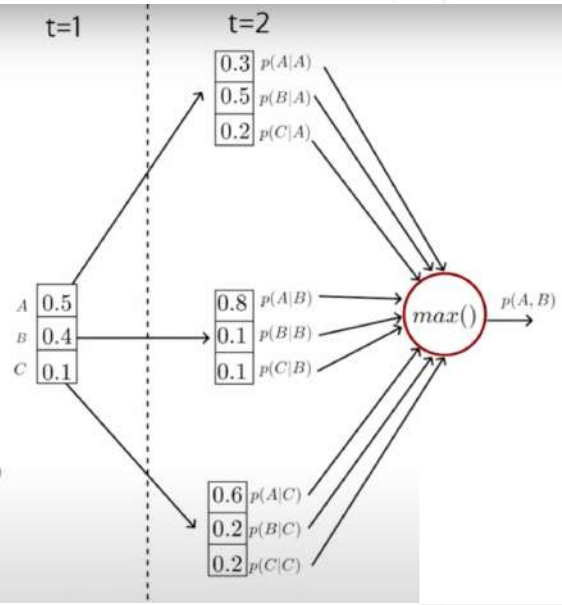
* **Illustration**

Exhaustive search for a sequence of length 2 with the vocabulary of size 3

At time step-1, the decoder outputs probability for all 3 tokens

At time step-2, we need to run the decoder three times independently conditioned on all the three predictions from the previous time step

At time step-3 we will have to run the decoders 9 times




- Greedy Search

As the name suggest, we will greedily pick the word with highest probability at each time step

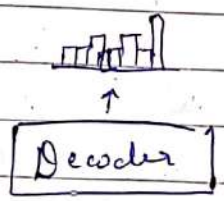
The time step here means every time we run the decoder to get the next word

Ishan Modi

Date: / /

Taking the same example of predicting a sequence of 5 words

timestep 1

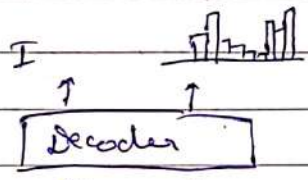



} probability distribution accross vocabulary (V)

since it is greedy search out of all probabilities in the distribution we pick word with highest probability

lets say at timestamp 1 "I" had highest probability
P(I) = 0.9

timestep 2




} probability distribution accross vocabulary (V)

again we pick word with highest probability.

lets say at timestep 2 "Like" had highest probability
P(Like | I) = 0.8

Now, we carry on same thing accross 5 time steps and pick highest probability words at each step given previous context. Thus you obtain the sequence

So, if we talk about time complexity, we will endup running the decoder 5 times, which is equal to the words in the sequence

↓
number of

Date: / /

Note - Unlike exhaustive search, greedy search doesn't guarantee that the sequence that is selected has the highest overall probability

eg - $w_1 \quad w_2 \quad w_3 \quad w_4 \quad w_5$

$P(s_1) \Rightarrow 0.8 \times 0.4 \times 0.25 \times 0.3 \times 0.22$ (sequence 1 probs)

$w_1' \quad w_2' \quad w_3' \quad w_4' \quad w_5'$

$P(s_2) \Rightarrow 0.19 \times 0.65 \times 0.82 \times 0.65 \times 0.76$ (sequence 2 probs)

Here, $P(s_2) > P(s_1)$

What we have actually done, at the first time step we picked the ~~second~~ word with second highest probability. Thus all the subsequent distributions changed

- Beam Search

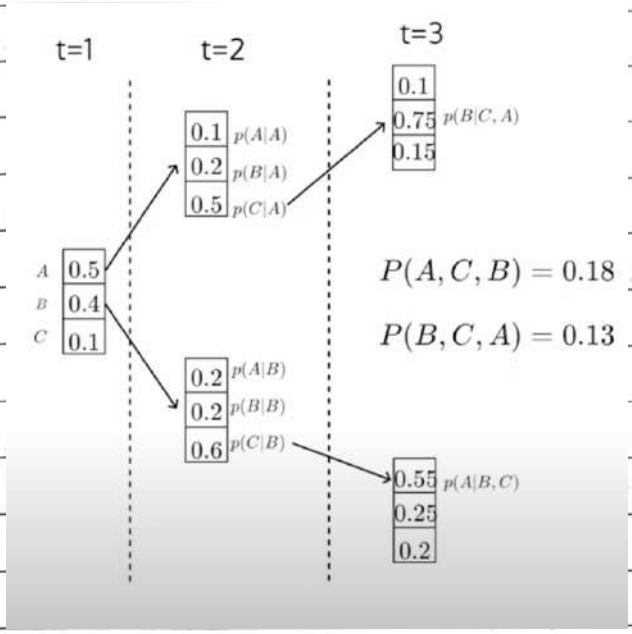
Beam search is some where in between exhaustive & greedy search

- Exhaustive search looks at all probabilities at all time steps
- Greedy search looks at highest probabilities at all time steps autoregressively
- Beam search looks at "top b" probabilities at all time steps where b => beam size.

Lets understand it with the following example

here vocab size $V = 3$ & beam size $b = 2$

Date: / /




As seen in the figure, at first timestep decoder is called only once & for all other timesteps it is called "b" (beam size) no. number of times.

time step 1 → we pick the top 2 words with highest probabilities

time step 2

$p(A)\ p(A|A) = 0.5 \times 0.1 = 0.05$
$p(A)\ p(B|A) = 0.5 \times 0.2 = 0.01$
$p(A)\ p(C|A) = 0.5 \times 0.5 = \boxed{0.25}$
$p(B)\ p(A|B) = 0.4 \times 0.2 = 0.08$
$p(B)\ p(B|B) = 0.4 \times 0.2 = 0.08$
$p(B)\ p(C|B) = 0.4 \times 0.6 = \boxed{0.24}$

we pick the top 2 words with highest probabilities

We carry on same till desired words in the sequence & the out of b sequences we pick sequence with highest probability

Date: / /

Note - Beam search with $b = 1$ is equivalent to greedy search

Beam search with $b = (V)$ is equivalent to exhaustive search

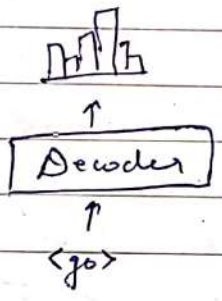
→ Sampling Based Non Deterministic Decoding (also called Stochastic Decoding) (output is not same everytime)

- Top-K Sampling

This approach is similar to beam search, but we add some random sampling at each time step.

K ⇒ the number of words to pick based on highest probabilities

timestep 1




} lets say $k = 3$ and we pick top three words with highest probabilities

$w_1 = 0.31$, $w_2 = 0.25$, $w_3 = 0.15$

Now, we normalize all of the selected words

$$w_1 = \frac{w_1}{w_1 + w_2 + w_3} = \frac{0.31}{0.31 + 0.25 + 0.15} = 0.436$$

Date: / /

$w_2 = \frac{0.25}{0.71} = 0.352$

$w_3 = \frac{0.15}{0.71} = 0.211$

Now, we choose a random number between 0 & 1

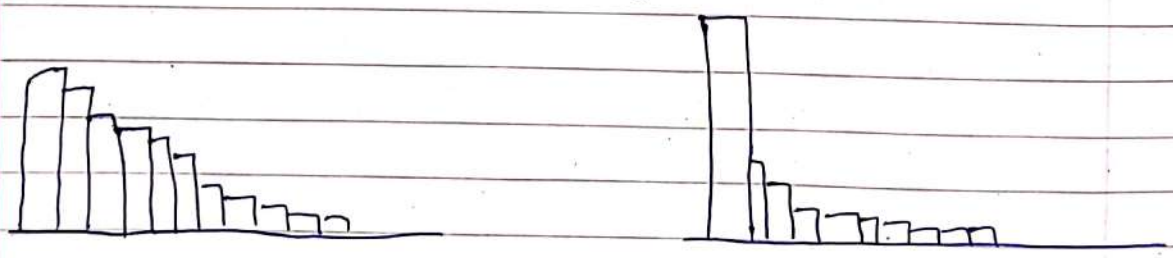
- interval $w_1$ = $0 \leftrightarrow 0.436$
- interval $w_2$ = $0.436 \leftrightarrow (0.436 + 0.352)$
  $0.436 \leftrightarrow 0.788$
- interval $w_3$ = $0.788 \leftrightarrow 1$

We pick the word within whoose interval the random number lies

Follow the same thing for all the other time steps & you will get random sequences

- Temperature Sampling

Lets say we have the following two distributions



A B

Date: / /

Each bar in the distributions represents the probability of a word

The distribution A has probabilities evenly distributed among words, whereas B is peaky distribution

If we pick top-k samples for distribution B it can lead to unwanted results because the probabilities in the k words other than the peak is very less. But the same thing would work fine with distribution A.

To deal with this something known as temperature is introduced

Instead of applying softmax to logits to get probability we can apply softmax with temperature

$$\text{sftmx with temperature} = \frac{\exp(\frac{x}{T})}{\sum \exp(\frac{x}{T})}$$

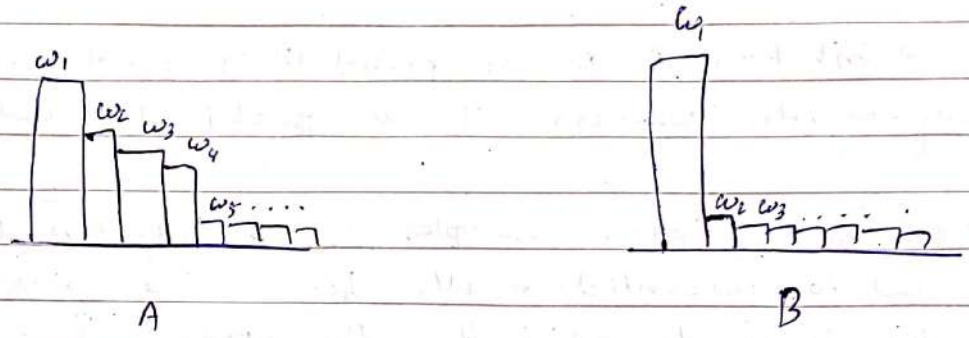
The default value of $T = 1$

Now, the property of the above function is

- smaller the value of T, it makes larger softmax even larger & smaller softmax smaller. Thus it makes distribution peakier & less no. of words means less creative
- higher the value of T, it makes smaller softmax ~~larger~~ large & larger softmax small. Thus flattens the distribution & leads to more words & more creativity

Date: / /

- Top- P (Nucleus) Sampling




Lets say A & B are two distributions at 1st time step

Now as per top-k sampling we pick the fixed number of words k.

The idea of top-p (nucleus) sampling is to pick dynamic value of k at each time step

Now, we calculate cummulative distribution (CDF) starting from $w_1$ till CDF becomes less than a threshold p, lets say $p = 0.6$

Examples — For distribution A

$w_1$ — 0.3
$w_1 + w_2$ — 0.45
$w_1 + w_2 + w_3$ — 0.55
$w_1 + w_2 + w_3 + w_4$ — 0.59
$w_1 + w_2 + w_3 + w_4 + w_5$ — 0.62

So we include all words below threshold 0.6

$\therefore k = 4$ in this case and $w_1, w_2, w_3, w_4$ move to next time step

Date: / /

- For distribution B

$w_1 - 0.8$ } Since CDF > threshold (0.6) & there is no word

$\therefore k=1$ & $w_1$ goes to next time step

We do sm similar for all time steps & get randomised sequences