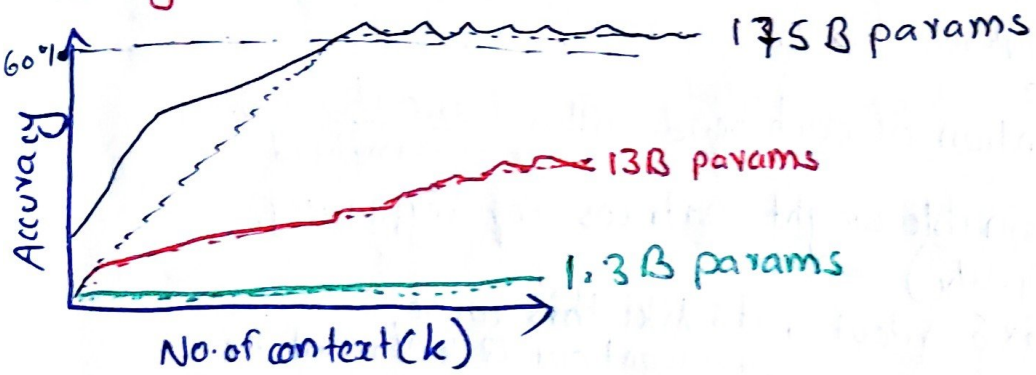
Deepseek from scratch

zero-shot
↳ No examples are given.

oneshot
↳ One example is given.

few-shot
↳ few numbers of example is shown to model to predict the answer.

* As we increase the no. of parameter, we human notice some emergent behaviour.



Our learning journey.

phase 1
Innovative Architecture
- Multi-head (MLA) latent Attention
- Mixture of experts (MoE)
- Multi-token prediction (MTP)
- Quantization
- Rotary Positional Encodings (ROPE)

Phase 2
Training Methodology
- Rise of Reinforce-ment learning (RL)
- Not only relying in human labeled date, use large scale RL to teach complex reasing to the model.
- Rule based reward system.

phase 3
GPU optimization tricks
- NVIDIA Paralled Thread Execution (PTX) not Cuda.

phase 4
Model ecosystem
- Model distilla-tion into smaller models (even down to 1.5B parameters)

Multi-head attention

* Example

b, num_token, d_in = (1, 3, 6).

```
x = torch.tensor([[[1.0, 2.0, 3.0, 4.0, 5.0, 6.0],
                   [6.0, 5.0, 4.0, 3.0, 2.0, 1.0],
                   [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]]])
```

step2 : Decide d_out, num_head
- d_out → suppose 6
- num_head → 2.

so, then dimention of each head will be $\frac{d\_out}{num\_head} = 3$.

step3 :- Initialize trainable weight matrices for key, query, value (W_k, W_q, W_v)

suppose 6×6 → dout, its like this cause, we want our Q, k, V must be as input dim.
- 6×6 → din

Eg: If we have 3×6 · 6×6 weight → 3×6 (by the way same dim)
- 3×6 → num_token, din
- 3×6 → num_token, d_out

so, when 1, 3, 6 then we get,

keys (1, 3, 6), Queries (1 x 3 x 6) & values (1 x 3, 6)
- keys: batch, no. of tokens, d_out

Now, as query, key, value is splited as no. of heads.

Here we have (1, 3, 6)
- batch, no. of tokens, dout

we unroll it as we make d_out as num_hed, head_dim → (1, 3, 2, 3)

In our case

T = tokens.

```
[ { [1, 2, 3], } h1
  { [3, 5, 6] } h2
```
T1, T2, T3

```
[ { [1, 2, 3], } h1
  { [3, 5, 6] } h2
```
T1

```
{ [7, 8, 9] }
{ [1, 2, 9] }
{ [1, 7, 8] }
{ [1, 2, 9] ] }
```
T2, T3

we reshape the place. & make (1, 2, 3, 3) for easy calculation.

head_dim.

```
[ [ [1, 2, 3], } H1
  [7, 8, 9],
  [1, 7, 8] ],
  [ [3, 5, 6], } H2
  [1, 2, 9],
  [1, 2, 9] ] ]
```
T1, T2, T3

& similary we will have for Queries, keys, values.

Now,
for attention $Q \cdot k^T$

$\Rightarrow (1, 2, 3, 3) \cdot (1, 2, 3, 3)$ → ~~no~~ index · transpose (2,3).
$= 1, 2, 3, 3) \cdot$ keys · transpose (2,3). → d_out, ~~no of tokens~~

$\Rightarrow \begin{bmatrix} Q_1 \cdot k_1^T \\ Q_2 \cdot k_2^T \end{bmatrix} = (1, 2, 3, 3) \cdot (1, 2, 3, 3)$

batch, no. of head, no. of token, d_out, batch, no of head, no of tokens

As $Q \cdot k^T$ gives attention matrix (score) $1, 2, 3, 3$

no. of tokens, ~~last~~ no. of tokens; batch, no. of heads

Now, we have to find attention weights

$$\text{Attention} = \text{softmax}\left(\frac{Q \cdot k^T}{\sqrt{dk}}\right) \cdot V$$

So,
Now, we Mask the attention scores to implement casual Attention.

```
tensor([[[[ -8.2    -inf    -inf],
          [ -1.3   -7.07    -inf],
          [-5.89   -2.72  -1.016]],

         [[ 4.65    -inf    -inf],
          [-1.31    1.39    -inf],
          [ 2.38    2.72   0.848]]]])
```

Divide by $\sqrt{\text{head\_dim}} = \sqrt{6/2} = \sqrt{3}$
& apply softmax.

```
↳([[[[ 1    , 0    , 0    ],
     [ 996  , 0.033, 6    ],
     [0.17  , 0.82 , 0    ]],

    [[ 1    , 0    , 0    ],
     [0.17  , 0.82 , 0    ],
     [0.38  , 0.46 , 0.15 ]]]])
```

we can also implement dropout after this.

This gives attention weights matrix.

T = Token | H = head

Now, last step:

Context vector = Attention weights * Values.

$= (b, \text{num\_head}, \text{num token}, \text{num\_token}) * (b, \text{num\_head}, \text{num\_token}, \text{head\_dim})$

$\Rightarrow (b, \text{num\_head}, \text{num token}, \text{head\_dim})$

$\Rightarrow (1, 2, 3, 3)$

~~Then we concatinat all~~

$\Rightarrow$ tensor([[[[1, 2, 3],
[2, 3, 6],
[6, 3, 3]],
[[1, 2, 3, 6],
[2, 1, 1]
[1, 1, 0,]]]])

T1, T2, T3 → rows; H_1, H_2 → heads

Now, we merge such that, we get,

$(b, \text{num\_token}, d\_out)$

$\Rightarrow (b, \text{num\_token}, \text{num\_head}, \text{head\_dim})$

$\Rightarrow (1, 3, 2, 3)$

tensor([[[[1, 2, 3], ← H_1
[3, 2, 1]], ← H_2 } we will merge them
[[3, 2, 1],
[1, 5, 6]]
[[3, 2, 2],
[5, 5, 0]]]]

T1, T2, T3 → rows

$\Rightarrow$ Final context vector:

tensor([[[[-3.6 , 2.09, 1.38, 1.59, -0.9 , -0.34]
[-1.2 , 2.08, 1.3 , 1.07, -0.1 , -0.96]
[-1.1 , 2.08, 1.3 , 1.00 , -0.9 , 1]]]]).

T1, T2, T3 → rows

Note :

i) we only need context vector for the last token in input sequence to predict next token.

ii) we observe, many computations seem to be repeated during the inference (as we are finding the context vector of all previous token each time).

(*) K-V (x) chache

↳ store the keys & values of a previous token, so to minimize the unnecessary re-computation.

* without caching the computational difficultly is $O(n^2)$
with caching its $O(n)$.

* size of KV cache. → contex length.
→ number of bytes per floating point.

$l * b * n * h * s * 2 * 2$ → caches (k, v)

- l → layers
- b → batch
- n → no. of heads
- h → head dimension

In deepseek R1/V3

$L = 61$

$b = 1$

$n = 128$

$h = 128$

$s = 100000$

KV chache size = 400GB

~~multi~~ different query, same keys, value weight matrix (w_k, w_v).

* we use,
↳ Multi-query attention
↳ group-query attention
} to solve KV cache problem.

Example

keys weight matrix.

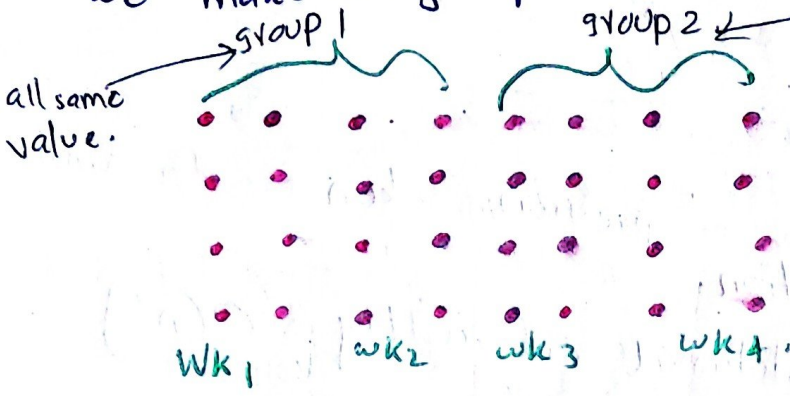
w_{k1} $(8,2)$ w_{k2} $(8,2)$ w_{k3} $(8,2)$

values weight matrix.

w_{v1} w_{v2} w_{v3}.

what we have done, we just make number of heads (n) = 1.
so, it reduce size by $\left(\frac{1}{n}\right)$ factor.

* Group-query ~~ation~~ attention
↳ it tells instead of making all value same in w_k, w_v we make as group of heads as same value.



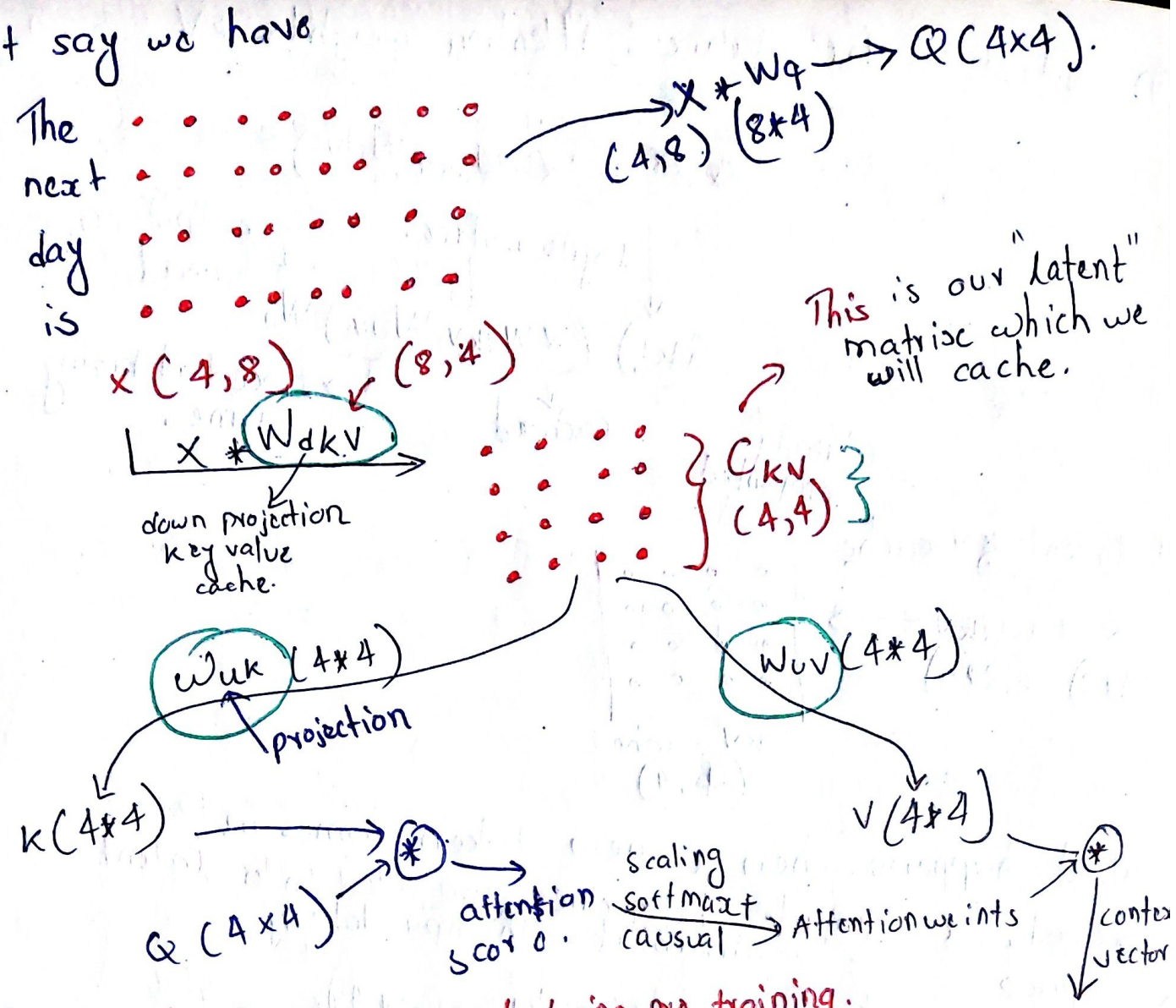
size = l * b * g * h * s * 2 * 2.
↙
no. of groups.

Here, GQA is a medium level approach for having low cache size & medium language model performance.

* To achieve both loss cache & high model performance
MLA is introduced.
↳ multi-head latent attention.

* they ask a simple question.
① what if we don't have to cache keys & value separately?
↓
what if we cache only one matrix.
↓
what if this matrix has less dimensions than "n*h"?
↳ So, to get that matrix, we start by projecting the input embedding matrix into a latent space.

Let say we have



Note: Wdkv, Wuk, Wuv are trained during pre training.

* How does adding this latent matrix help???

i) $Q = X * W_q$

ii) $C_{kv} = X * W_{dkv}$

iii) $K = C_{kv} * W_{uk} = X * W_{dkv} * W_{uk}$

iv) $V = C_{kv} * W_{uv} = X * W_{dkv} * W_{uv}$.

"Absorption trick"

↳ Attention scores $= Q * k^T$

$X W_q * (X * W_{dkv} * W_{uk})^T$

$\Rightarrow X W_q * (W_{uk}^T * W_{dkv}^T * X^T)$ & it C_{kv} latent cache.

$\Rightarrow X (W_q * W_{uk}^T) (X W_{dkv})^T$

Absorbed Query → fixed at training

This need to be cached.

⊛ Context Vector Matrix : Attention weights * V

$(QK^T)(X W_{dkv} W_{uv})$

logits matrix.

output projection head

$(QK^T)(X W_{dkv} W_{uv}) W_o$

W_o → fixed at training time.

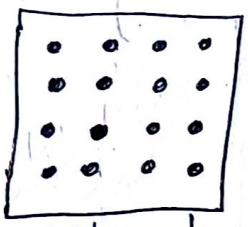
(QK^T) → already computed

$(X W_{dkv})$ → cached

* latent kV cache

$X * W_{dkv}$

(4*8) (8*4)



kV cache
(4,4)

* what happens when a new token comes in?

↳ First, we compute the queries, projected into latent space

① $Q : X (W_q W_{uk}^T)$ — we absorb it with input token.

query vector for bright. → (1*4).

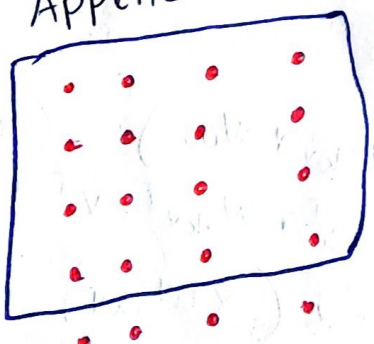
suppose X_{bright} (1,8)

W_q (8,4) W_{uk}^T (4,4)

② Compute kV vector.

X_{bright} (1,8) * W_{dkv} (8,4) → .... (1,4)

Append to latent kV cache



previous kV cache.

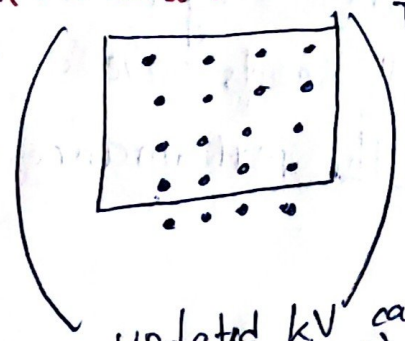
Updated kV cache
(5,4).

(3) Multiply Q with update kV cache.

Q: $X(W_q W_{uk}^T)$ * (updated kV cache)^T

x_{bright} (1,8) → (8,4) (4,8)

updated kV cache (5,4)

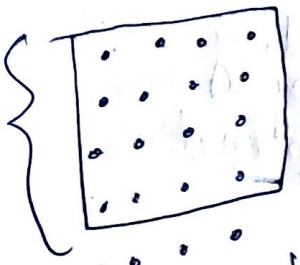


Attention scores for "bright" (1,5)

Attention "bright" weights for. (1,5)

context vector for bright (1,4)

(*)



updated kV cache (5,4)

* Wuv (4,4)

values matrix.

This is later absorbed into the logits calculation.

Here we donot need two separate k-cache & v-cache, we just need one latent cache (Ckv).

Here, ~~We~~ ~~store~~, we store Ckv only.

s

rs

chache chosen by ourself. not n*h

now size 2 * l * b * s * dl → dimention of kv cache

& also one "2" is gone no any k & v nah!!!

Here, W_{uk} & W_{uv} have weights different for each attention head. Thus, all heads have different k,V values.

↳ This solves the performance issue with MQA.
multi-query attention.

* Binary positional Encoding

I saw a dog. The dog was black in color.

suppose,

64: 0 1 0 0 0 0 0 0 ↓
65: 0 1 0 0 0 0 0 1 ↓
66: 0 1 0 0 0 0 1 0 ↓
67: 0 1 0 0 0 0 1 1 ↓
68: 0 1 0 0 0 1 0 0
⋮ so on.

Here, lower indices oscillate fast between positions.

Higher indices oscillate slow between positions.

Main problem:

* Since, the values of the integer positional encoding are discrete, they lead discontinuities in the resulting vectors.
* During pre-trained, it becomes difficult for the LLM optimation routine to deal with those "jumps" or discontinuities.
* best practice
  ↳ graph with smooth curve no jumps.

* Sinusoidal positional Encoding:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

pos → pos of token; i → index of dimention

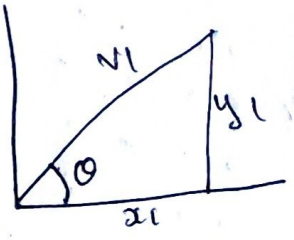
$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

suppose,

$i = 0$ $\quad PE(pos,2) = \sin\left(\frac{pos}{10000^{2/d}}\right)$

& $PE(pos,3) = \cos\left(\frac{pos}{10000^{2/d}}\right)$ } for index 2 & 3

suppose $\frac{pos}{10000^{2/d}} = \theta$. then



Here $\sin\theta = y_1$ & $\cos\theta = x_1$
↳ positional encoding value of 3rd position.

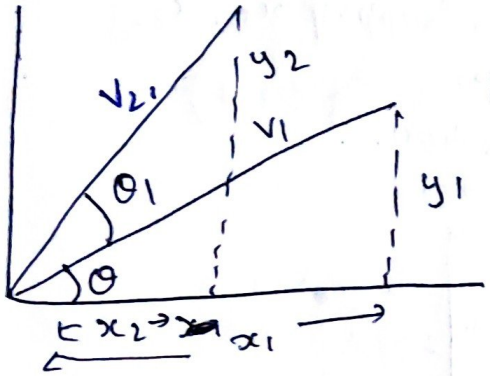
positional encoding value of 2nd position

Now, to find position encoding for p+k position

~~$pk = \sin(\omega p$~~ pos ~~$p+k+1$~~

$$P.E(pos_k, 2(\cdot)) = \sin\left(\frac{pos+k}{10000^{2i/d}}\right)$$

$$\& \; P.E(pos_k, 2(\cdot)+1) = \cos\left(\frac{pos+k}{10000^{2i/d}}\right)$$



Here $\theta + \theta_1 = \omega(p+k)$.

$\hookrightarrow \sin\left(\frac{p+k}{10000^{2i/d}}\right)$

$\hookrightarrow \cos\left(\frac{p+k}{1000^{2i/d}}\right)$

∴ Here, P.E for (pos + k), we just need to rotate the vector V_1 by an angle.

Here, also conclude.

$\theta = \omega p$ & $\theta_1 = \omega k$.

* V_1 & V_2 are rotations of each other.
* What this mean is: Relative positional encodings are just rotations of each other.

---

* Positional embeddings are directly added to token embeddings. This pollutes the semantic information carried by token embeddings.
↳ solution:
↳ can we preserve original vector magnitude by merely rotating the vector.

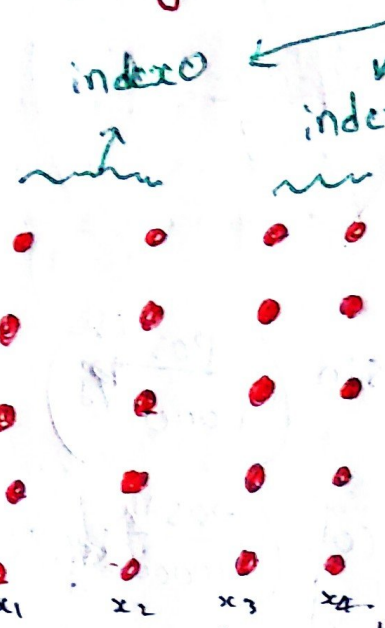
∴ So, Rope introduced,

i) In Rotary embedding, we are going to look at augmenting the queries & the keys vectors. we are not going to touch my token embedding. vectors at all.

ii) we are just rotating.

Rope

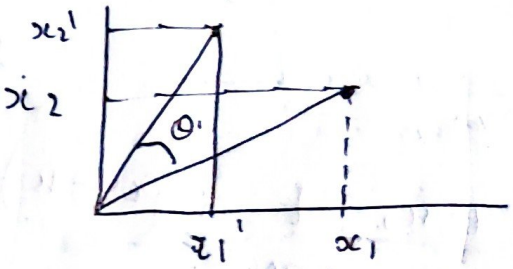
Rotary Positional encoding



Here for Pos 1, we take the first two suppose x_1, x_2.
& we apply x_1', x_2' = sin, cos of ωp.

where, $\omega = \frac{1}{10000^{2i/pd}}$ & p = position.

& again same of x_3 & x_4.



we apply this in q, k, v.

So,
our original query-key
x_1 x_2 x_3 x_4
& our positional encoded Query-key
x_1' x_2' x_3' x_4'

$\omega p = \frac{p}{10000^{2i/pd}}$

* This also conclude:
Low index, high oscillation
↳ means model captures these small shifts, example,
↳ "I just told her the truth" vs "I told just her the truth".
↳ The low index fast oscillations capture the change brought about by varying the position of the word "told".
Here, 'told' change position like from 2 to 3, this is capture by low index, high oscillation.

* Higher index components ensure that even with large position differences, the relationship is preserved.
↳ It capture the long range context dependencies.

Decoupled Rotary Position Embedding

Using MLA with Rope.

⇒ Attention scores = $Q \cdot k^T$

⇒ $X \cdot W_q \cdot (C_{kv} \cdot W_{uk})^T$

$= X \cdot W_q \cdot W_{uk}^T \cdot C_{kv}^T$

absorbed → $W_q \cdot W_{uk}^T$

but, with rope.

Attention scores = $R_{pos}(X W_q) \cdot R_{pos}(W_{uk}^T W_{dkv}^T X^T)$

this act as function

due to R_{pos},

now, $X W_q W_{uk}^T$ cannot be absorbed.

↳ due to this reason, we may need to recompute the keys for all tokens during inference.

* TO avoid this, Deepseek proposed Decoupled RoPE

What is says??

⇒ Attention scores = $Q \cdot k^T$

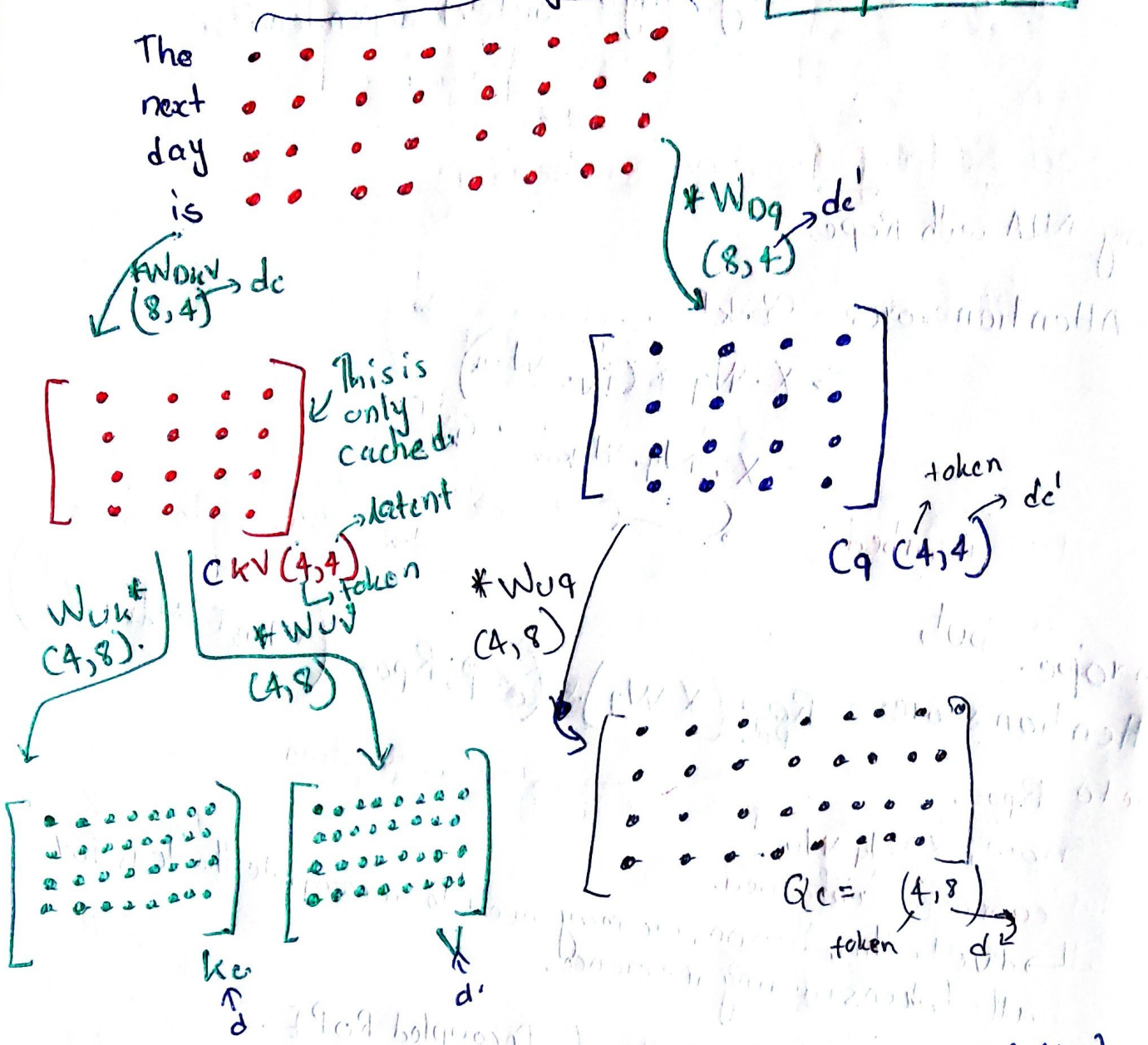
$= [Q_c : R_R] \cdot [k_c \cdot k_R]^T$ (Rope, normal)

$= Q_c \cdot k_c^T + Q_R \cdot k_R^T$ { like the tensor is in concat so we can do to half of it as so on nah!! }

* we do some extra computation, but it's fine as we insure our absorb technique of MLA.

How decoupled Rope works ???

This is without rope as d



Note: we do

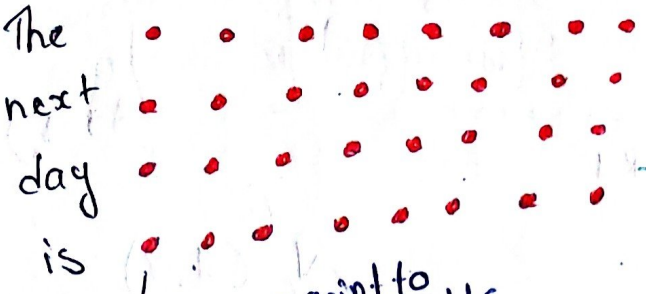
$W_{DQ} \longrightarrow C_Q \longrightarrow W_{UQ} \longrightarrow Q_C$ (downprojection, upprojection).

↳ According to the paper, It saves activation memory during the training type.

meaning:

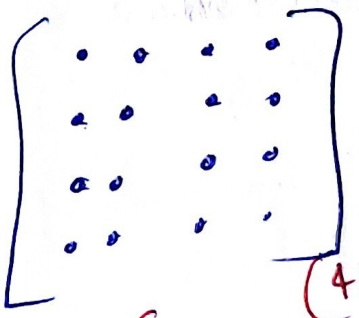
down-projecting the queries matrix into a smaller matrix reduces the activation memory or memory requirements during training time & it's also seems to lead to a better performance.

* With rope — In here, we did not find k_R notice & also, we use same value accross different heads



W_{KR} (8,4) → d_h^R — we are going to use same weights accross diff. multi head.

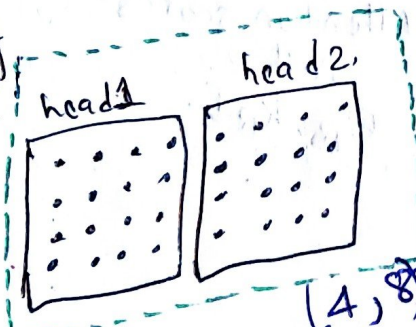
dimension of head in rotary.



Rope

here, key are save in multi head but not query in Rope

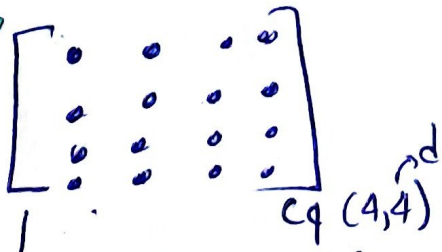
Multiply accross heads.



K_R

we are going to use same weights accross different heads.

W_{DQ} (8,4) → d_c'



$W_{q.R}$ (4,8) → $n_h * d_h^R$.

ROPE

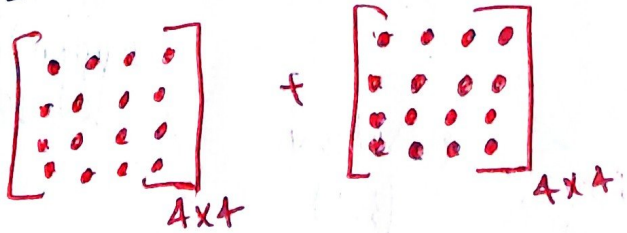
Q_R

we concatinate both.

Now, attention scores = $Q \cdot K^T$

$\Rightarrow [Q_c : Q_R] \cdot [k_c : k_R]^T$

$\Rightarrow$



Attention score → Scaling $\sqrt{4+4}$ softmax → Attention weights

4*1 → 4*4

V_c^* (4, 8)

context vectors.

4*8

* we only cache C_{kv} & k_R

So what exactly happens when a new token comes in?

X_{bright} (1,8)

Path 1

Attention scores.
Path 1
$Q_c \cdot k_c^T$

Path 2.

Attention scores
Path 2.
$Q_R \cdot k_R^T$.

NOW,

$Q: X (W_{DQ} W_{UQ} W_{UK}^T)$

W_Q.

(8,4) (4,8), (8,4)

dc. ← latent space.

dc

d

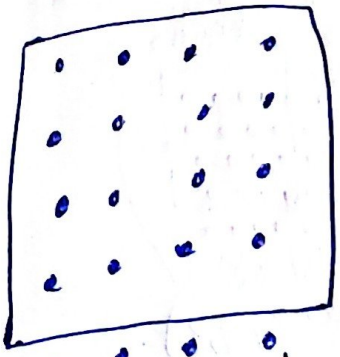
Absorbed query vector for bright

(1,4)

After that →

X_{bright} * W_{dkv} (8,4) → (1,4)

Appon to latent KV cache.



original
C_{kv}

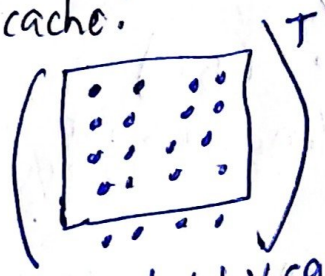
updated kV cache. (5,4)

why absorbed query help??
↳ because of it we didnot need to cache whole kV values, we just cache the ~~cant~~ C_{kv} latent matrix.

③ Multiply Q_c with updated kV cache:

$Q_c : X (W_{DQ} W_{UQ} W_{UK}^T)^*$

(1,8) (1,4)



updated kV cache (5,4)

.... (1,5)

Attention scores for "bright"

Path 2

$Q_R \cdot k_R^T$

$x_{bright} * W_{kR} \rightarrow d_h^R$

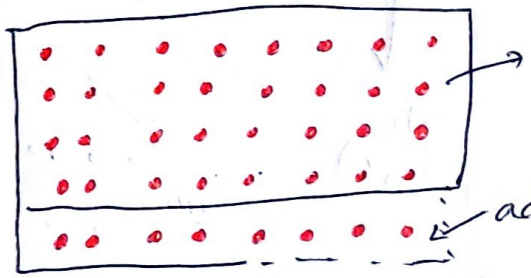
(1,8) (8,4)

(....)(1,4)

Rope

.... expand across heads. (.....)(1,8)

we have to cache it k_R



now to compute Q_R.

~~$X * W_{DQ} \rightarrow * W_{UR}$~~

$C_Q = x_{bright} * W_{DQ}$

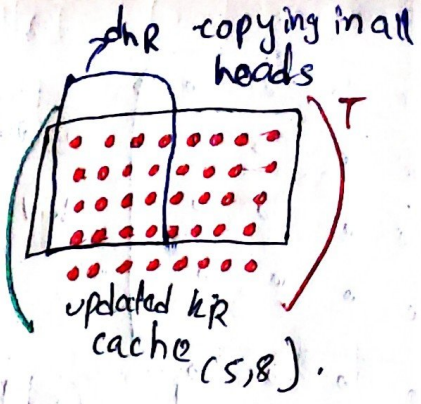
(1,8) (8,4)

.... (1,4) d_c' (latent down projection)

$* W_{QR}$ (4,8)

..... (1,8) Rope → ..... Q_R (1,8)

As we have both updated k_R & q_R we find Attention scores for = ... * (1,8)



↓

..... (1,5)

Attn scores for "bright"

Now,

Total attention score = Attention ~~Attn score~~ score from path 1 + Attention score from path 2.

= ..... (1,5)

So, total caching dimension = $d_c + d_h{}^R$

update kV cache (5,4) * W_{UV} (4,8) ↓ V_c (5,8) ~~(5,8)~~ * Resulting attention score * V_c (5,8)

↓

..... context vector for bright

---

4) Cache Memory size comparison.

MHA : $2 * 2 * l * b * s * n * h$

MLA : $2 * l * b * s * d_l$

MLA + rope : $2 * l * b * s * (d_l + d_{hR})$ ← $4 * h$, $h/2$

} in original paper, here h = attention dimension

MoE

Mixture of Experts

Top-k → a token routed to how many experts.

* what is the need to add multiple experts?

↳ Adding multiple experts allows models to be pre-trained with far less compared to a dense model (without experts).

↳ Allows much faster inference as compared to a dense model (without experts).

* In a dense model, every input token passes through all the parameters (i.e, all layers & neurons).
In contrast, a MoE model has multiple "experts" (think of them as sub-networks or feedforward layers), but only a small subset. (eg, 2 out of 64 are activated for any given input.

Example

Suppose we have 64 experts each 128 dim

2 active experts then

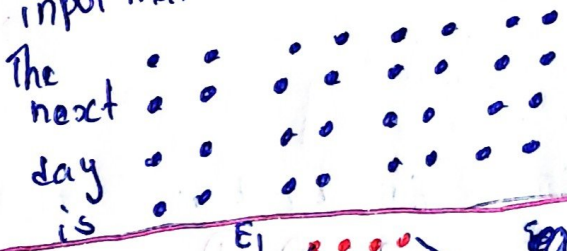
$2 * 128$ parameter

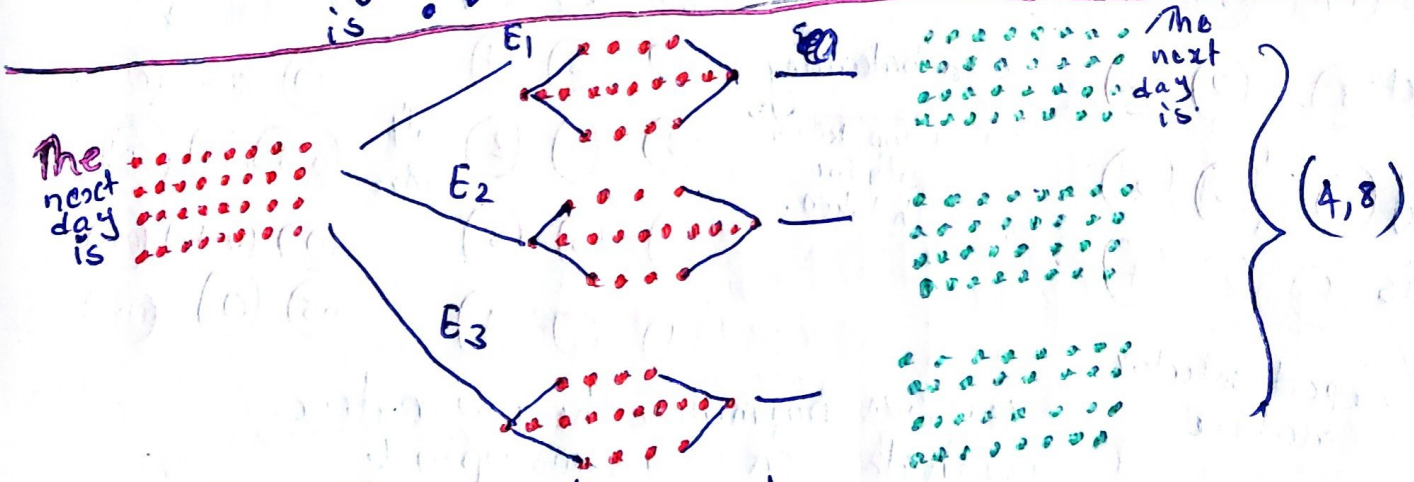
which is 3 times less when a dense layer of 768 dim.

The whole idea is sparsity,

Example :

Suppose input matrix of size (4,8)





Here, comes the problem, we want to route to only 2 expert & the output must be merged but how to do??

Step 2 :- To merge, we use a method called "Load balancing"

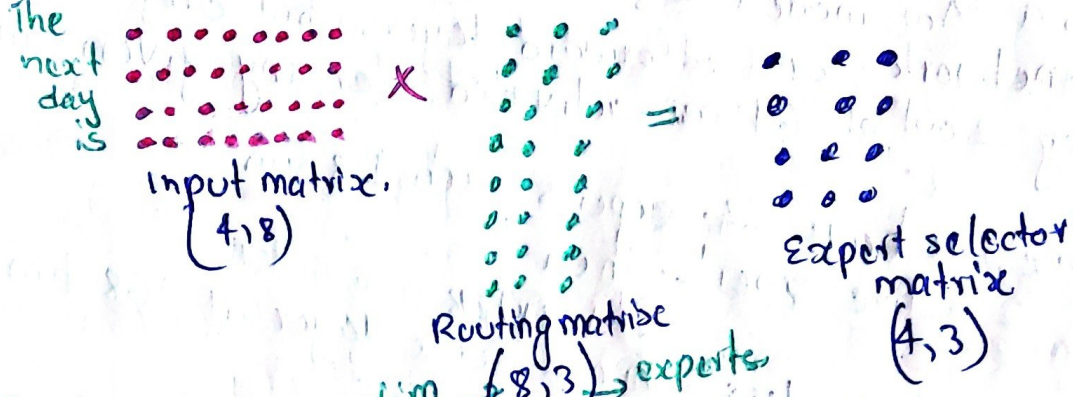
Ideally, since there are 3 experts : every token will be routed to the 3 experts.

* Every token will only be sent to a selected no. of experts. This is called as load balancing.

Step 3 :- Once we have decided how many experts will be assigned to each token, the next question is:

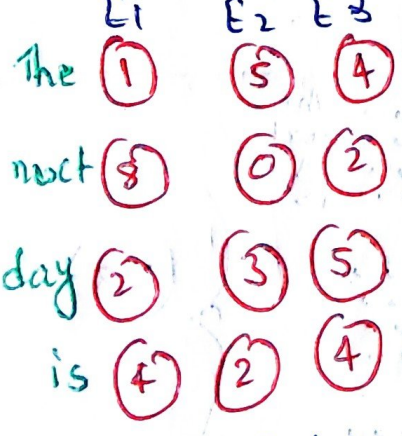
"How many weightage needs to be given to each expert?"

This is decided by the routing mechanism.

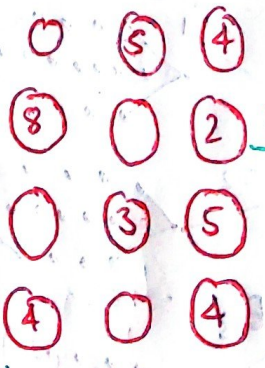


STEP 4

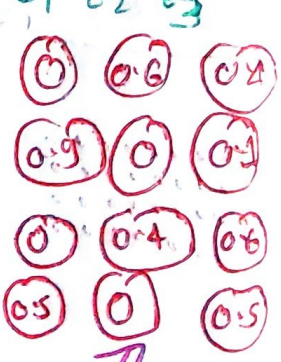
suppose



Load balancing → select top_k(2) experts for each token.



Soft max



Here, we normalize the raw matrix, so that each row sums upto 1.

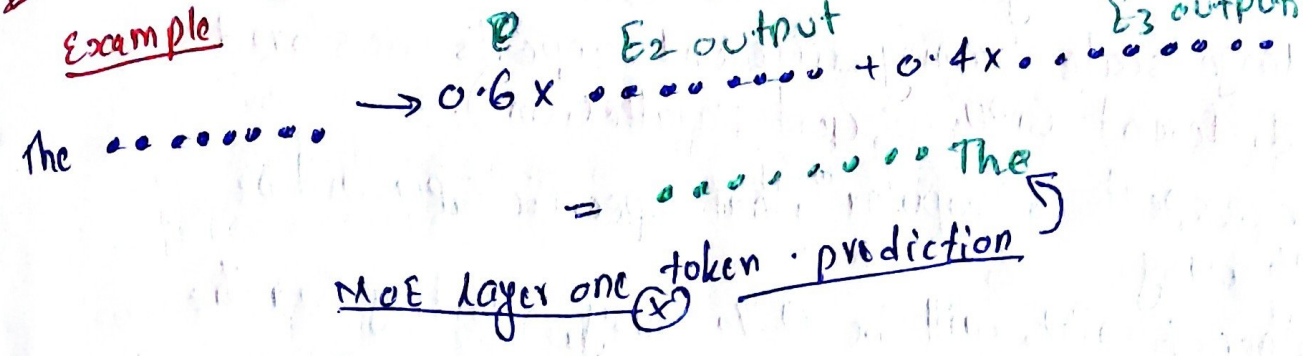
we do softmax.

weight matrix.

Step 5

We multiply weight x output from expert = final output.

Example



So, now **Auxilary loss** :

↳ we need to ensure there is balance, while pretraining, we also ensure all the experts are essentially utilized.

Here, E1	E2	E3
0	0.6	0.4
0.9	0	0.1
0	0.4	0.6
0.5	0	0.5
↓ 1.4	↓ 1	↓ 1.6

This indicate there is no balance in moe learning.

So, we introduce a lose.

Loss = $\lambda (C.V)^2$. → Auxilary loss

C.V = coeff. of variation = $\frac{\text{standard deviation}}{\text{mean}}$

why we balance?

↳ The rich get richer ~~pobt~~ problem :

↳ At the start of training, all experts are initialized randomly. By pure chance, Expert A might be slightly better at a specific token than Expert B.

- the router sees this & sends more tokens to Expert A.
- Expert A gets more "practice" (training) & becomes even smarter.

This happen as of 62 experts sit idle and learn nothing. we end up with a huge model that has the "brain" of a tiny model.

To prevent this, we add a "load Balancing" or "Auxillary loss" to the main training objective.

Why we need balance?

A) Hardware Efficiency (The Bottleneck).

↳ In large-scale training different experts are stored on different GPUs (Expert Parallelism).

- if one expert is "popular", that specific GPU will be 100% busy.
  The other 7 GPUs will be 0% busy, waiting for the first one to finish.

impact: Our training speed drops to a crawl because we aren't using our hardware in parallel.

B) Waste of Parameters.

↳ The whole point of MOE is to have a "Large Brain"

- If the router only uses 2 experts out of 64, you have wasted 62 experts worth of a GPU memory (VRAM). You are paying for a 50B parameter model but only getting the performance of a 2B model.

* Load Balancing

↳ Assinging equal importance to tokens ≠ uniform token routing.

How??

This is imbalance, but it's ideal for an example

suppose

E_1
1
0
0
0

Expert importance of $E_1 = 1$

E_2
0.25
0.25
0.25
0.25

→ Expert importance of $E_2 = 1$

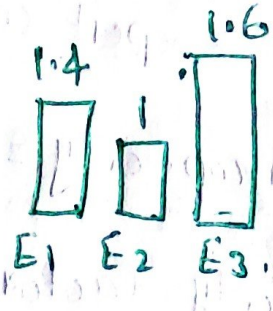
Here although expert importance is same, we clearly see not uniformly distributed tokens.

If a token sent to each expert is non-uniform, it can lead to a high memory & a reduced MoE performance.

To balance it we use load balancing

First, we calculate the probability that the router chose the given expert. (P_i)

suppose, we have 4 tokens & 3 Experts.



So, $(P_1) = \frac{1}{4} \times 1.4$ ← Expert importance

$\frac{1}{4}$ → probability of one token

Next step is to calculate the fraction of tokens dispatched to each experts. (f_i)

How is it calculated ??

suppose

	E_1	E_2
T_1	0.6	0.4
T_2	0.4	0.6
T_3	0.6	0.4
T_4	0.4	0.6

$f_i = \frac{1}{4} *$ sum of count of token to that expert will route to the

$\frac{1}{4}$ → probability of one token

2 = Expert importance = 2.

$P_i = \frac{1}{4} \times 2$ for E_1 for $E_2 \Rightarrow P_i = \frac{2}{4}$.

$f_i = \frac{1}{4} \times 2$.

as only T_1 & T_3 will be routed to E_1 as it is high probability.

So, Loss $= \sum f_i \times P_i$

we will minimize this

* Capacity factor & Expert capacity

↳ suppose all token get routed to only one expert
this cause the b previous problems.

* So,
Expert capacity = the maximum number of tokens that a single expert can process in a given batch.

Expert capacity = Tokens per batch / Number of experts * Capacity Factor.

if capacity factor = 1 & 1000 token in batch &
4 experts.
Then, 250 max token is handle by each expert.

Tokens per batch = batch size * sequence length * top_k.

top_k = no. of experts chosen for each token (load balancing)

* if capacity factor = 1, each experts have equal tokens.
if capacity factor > 1, some have more token & some has less tokens.
if it's too huge it make heavily imbalance.
so, generally it is in range of (1.125 - 2)

if capacity factor < 1, some tokens may get dropped if all experts are full.
its not prefer to drop, as deepseek V3, they didn't drop any token.

How deepseek Rewrote MoE

Deepseek Innovation

Auxiliary loss free load balancing

we know,

load balancing loss = Scaling factor * No. of experts * $\sum_{i=1}^{\text{num. experts}} f_i p_i$

The use of this loss helps maintain load balance in expert models, but it also acts as a regularization term that interferes with language modeling.

If the scaling factor is too small or absent, it leads to poor balance and routing collapse, reducing model efficiency & limiting expert utilization.

A large scaling factor ensures load balance but degrades overall model performance.

This tradeoff presents a fundamental problem; balancing experts effectively without compromising training quality.

model-loss = Training loss + $\lambda * \sum f_i p_i$

expert selector weight matrix. (4,3).

Then How?

Load Balance without loss

	E_1	E_2	E_3
T_1	0	0.6	0.4
T_2	0.7	0	0.1
T_3	0	0.4	0.86
T_4	0.5	0	0.5
	2	2	4

Token routed

Total token routed = 8

Avg. load per expert = 8/3 = 2.67.

Based on the average load per expert, we can know whether a given expert is overloaded or underloaded.

$E_1 = 2.67 - 2 = 0.67$ underloaded.

$E_2 = 2.67 - 2 = 0.67$ underloaded

↑ load violation.

$E_3 = 2.67 - 4 = -1.33$ overloaded.

load violation
= Avg load per expert - load for a given expert.

Now, next step, we introduce bias.

For each expert, we introduce a bias term b.
The bias is initialized as 0 for each expert.
Here is how the bias term is updated for each expert.

$$b_i^{\circ} = b_i^{\circ} + u * \text{sign}(\text{load violation error})$$

u = pre-defined constants.

Here, for, expert 1 E_1 & E_2 both are underloaded, so, we increas the bias & for E_3 is overloaded, we decrease the bias. like for E_1

$$b_i^{\circ} = b_i^{\circ} + u * (+)$$
$$= b_i + u.$$

E_1	E_2	E_3
$0+b$	$0.6+b$	$0.4-b$
$0.9+b$	$0+b$	$0.1-b$
$0+b$	$0.4+b$	$0.6-b$
$0.5+b$	$0+b$	$0.5-b$

Adding bias term will make sure that the values of expert 1 & expert 2 increases. This will increase the probability of this experts.

Note:- The bias value is not a "trainable parameter" in the pytorch /Tensorflow sense (it has require-grad = False). It is a running statistic updated by a simple if-else rule.

* The bias is only added during the selection phase. Once the experts are chosen, the model after uses the original "unbaised" scores to determine how to combine.

shared experts

In this approach the experts are divided in two groups.

- shared Experts: Experts that process every token, regardless of routing
- Routed Experts: Experts that handle selectively, based on the usual routing strategy.

Traditional MoE face:

1) knowledge Hybridity
   ↳ limited experts, so, each experts will have diverse knowledge.

2) knowledge Redundancy:
   ↳ multiple experts may converge in acquiring shared knowledge in their respective parameters, leading to redundancy in expert parameters.

* common & general information are handled by shared experts.
  Note, all token needs to be pass through shared experts.

* Fine-Grained Expert Segmentation

↳ The core idea is to divide each expert into multiple smaller experts while maining the overall model capacity & computational cost.

↳ In fine-grained expert segmentation, each large expert FFN is split into m small parts, reducing the hidden dimension by a factor of 1/m

To compensate for the reduced capacity per expert, the model activates m times more experts per token while keeping the same total computational cost.

* MBPP evaluation by
↳ most basic python programming

Multi-Token Prediction (X)

* Deepseek renovate this technique also.
* Instead of prediction only one token we predict the three next token & do back propagation.

why is Multi-Token Prediction useful? (X)

Densification of Training signals
- multi future target tokens which helps model gets more supervision.
- That means more richer gradients.
- Faster & stable learing.

Improved data efficiency
- Each training see multi-next tokens which mean we need less data to reach the same performance.
- we extract more information per token.

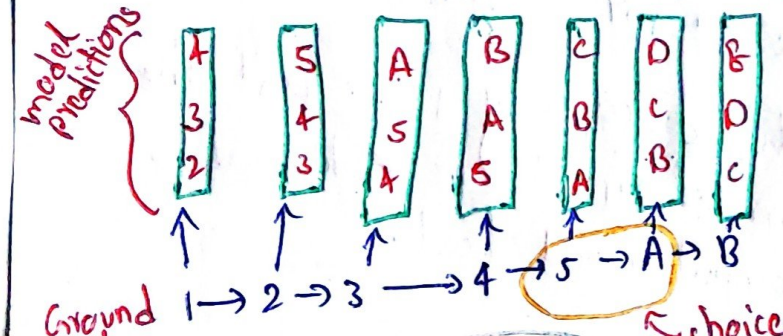
Better planning
- MTP implicitly assigns greater importance to "choice points" - key tokens that significantly influence future outcomes.
Thus models learns to prioitize ~~offcial~~ crucial decision-making elements

Higher Inference speed.
- The model predicts k tokens in one forward pass.
- fast text generation.

MTP-trained models achieved better results on standard benchmarks like HumanEval, MBPP with the same amount of training data, solving about 15% more code problems on average.

How?

suppose.



since token A is a part of multi predictions (appears in predictions starting from tokens 3, 4, & 5), errors related to predicting A appears repeatedly in the loss calculation.
Hence, the training process implicitly prioritics improving predictions of such consequential tokens, focusing the models capacity on more critical decision rather than inconsequential ones.
It assign high weight to consequential choice point tokens

The choice point is hard to transition in one token prediction, we need some thing which must see some unknow transition will occur in near future.

What is speculative Decoding?

↳ use two model to predict fast.

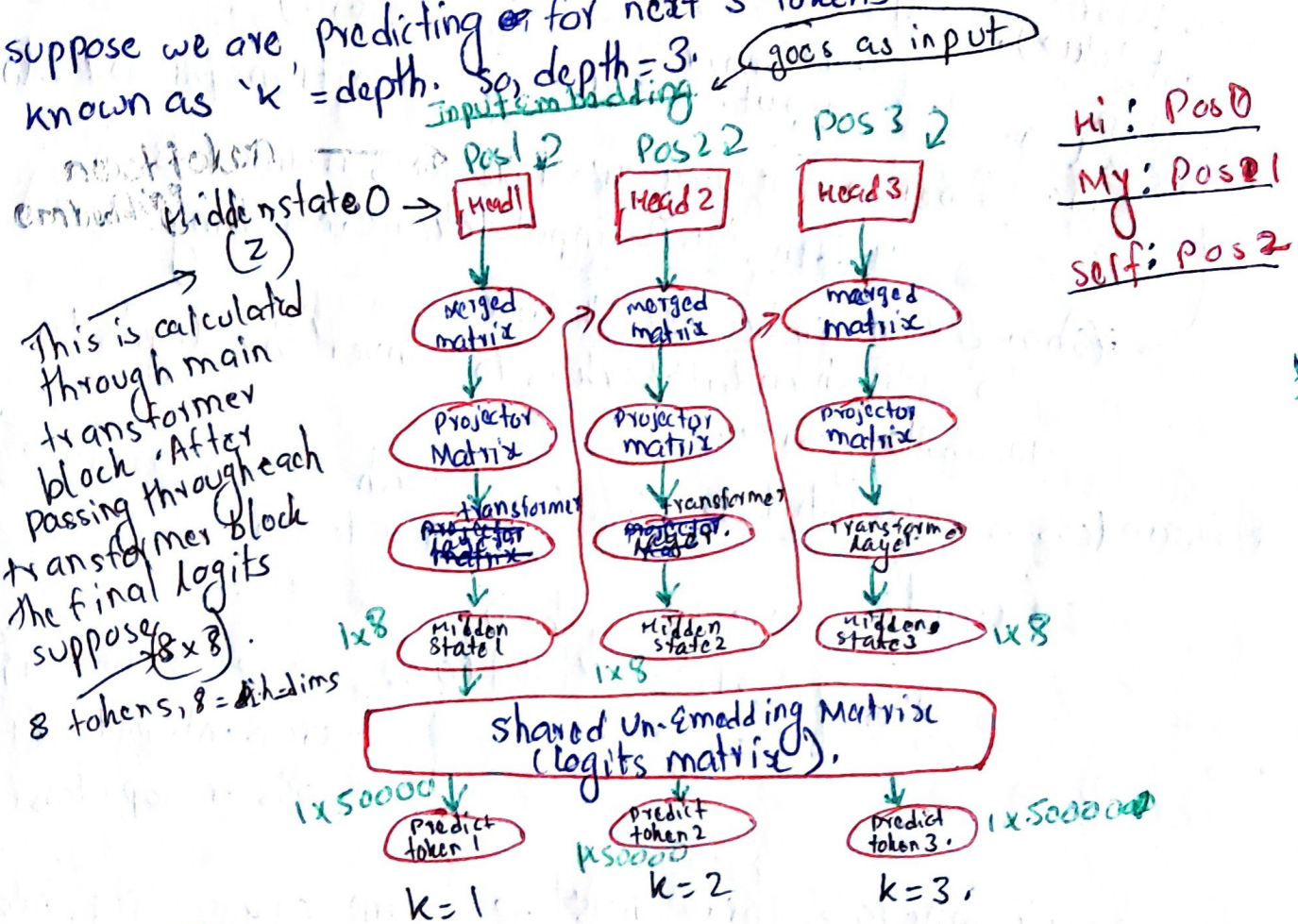
↳ Draft model (SLM):
- quickly predicts the next k tokens in a sequence.

↳ Main LLM
↳ verifies the tokens generated by the draft model & corrects them as needed.

- Deepseek use MTP during pre-training only. During Inference they discarded MTP.

How deepseek implement MTP?

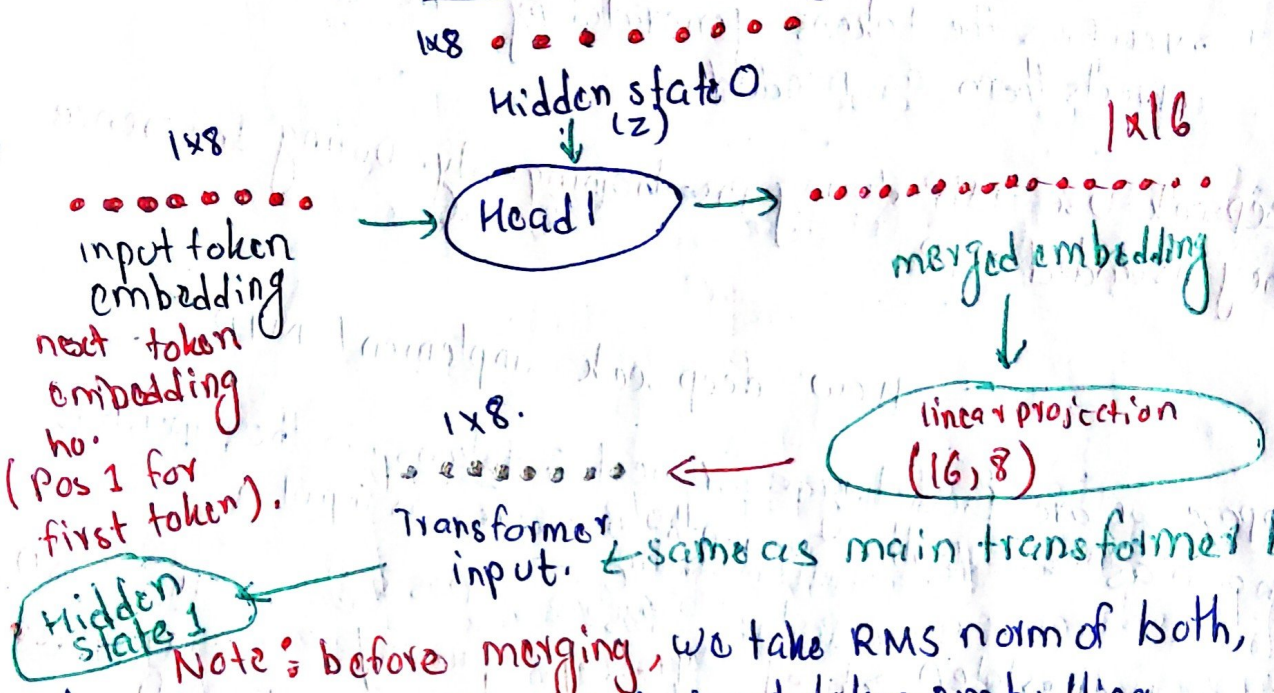
suppose we are predicting for next 3 tokens. in the paper, it is known as 'k' = depth. so, depth = 3.



Here, suppose
Input sequence length, T = 8
Prediction depth D = 3
This mean: we can predict only for i = 4 as i is from 0 - 7.
because only the last token that must have 3 output token is at index i = 4.

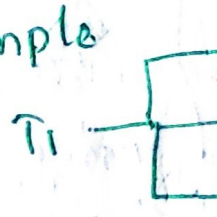
The first MTP paper predict each next token independently with each other but,
Deepseek introduce sequentially prediction keeping the complete causal chain at each prediction depth.

Each head working?



* Main loss is calculated as Σ (loss of each token predicted from a token)

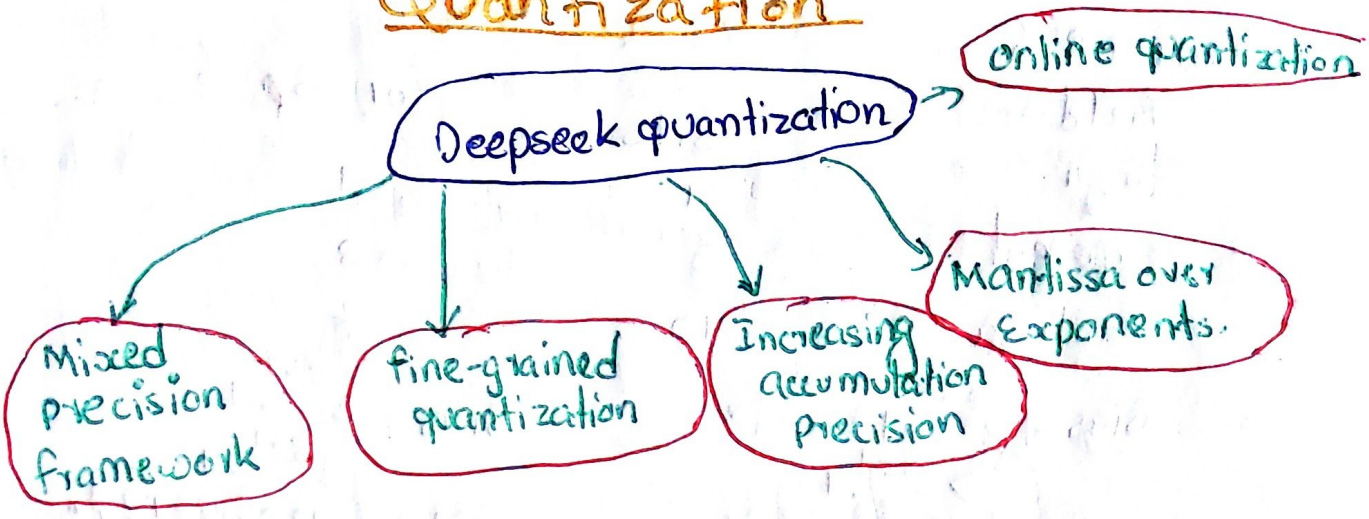
example



The Σ, total loss = cross entropy loss $(T_1, \hat{T}_1')$
+ cross entropy loss $(T_2, \hat{T}_2')$
+ coss entropy loss $(T_3, \hat{T}_3')$

Deepseek innovate, there must be some causality, hidden state between multi token prediction.

Quantization



How a number is stored in FP32??

↳ According to IEEE-754 standart.

field	Bits
Sign	1
Exponent(e)	8
Mantissa	23.

for FP32 bias = 127.

suppose $x = 3.14159274101257 32$.

Binary(approx) = 11.00100100001111 1111

Normalized = $1.10010010000111111 \times 2^1$

$= (-1)^{sign} (1 + \text{mantissa}) \times 2^{e-127}$ (formulae)

~~accompanying~~

stored 32 bit

0 | 1 0 0 0 0 0 0 0 | 10010010000111111

1 bit | 8 bit | 23 bits

To find the exponent bit we compare the formulae with the Normalized binary of the number.

we get, $e - 127 = 1$

$e = 128$.

↙ we add bias to minimize the negative to occur.

For FP16

field	Bits
Sign	1
Exponent	5
Mantissa	10

$Bias = 2^{e-1} - 1$

$\Rightarrow 2^{5-1} - 1$

$= 15.$

for Bf16

	1
	8
	7

$Bias = 2^{e-1} - 1$

$= 2^{8-1} - 1$

$= 128 - 1 = 127.$

Mixed Precision Framework

↳ Suppose $y = Wx$.

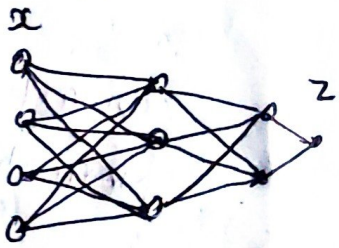
computed initially in FP32 for numerical stability then converted back to BF16.

↳ converted to FP8 from BF16.

↳ maintained in higher precision (BF16 or FP32), then converted to FP8 on the fly. (Fprop in paper figure).

$y_{FP32} = W_{FP8} \times x_{FP8}$

↳ stored in BF16 to optimize memory.



In the middle of a neural network,

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} * W^T$$

↳ it comes as $z = Wx$

$\frac{\partial z}{\partial x} = W$

we know, $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \times \frac{\partial z}{\partial x} = \frac{\partial L}{\partial z} \times$

To maintain dimension in back propagati

computed initially as FP32, then converted to BF16.

stored as BF16 but converted to FP8 for computation.

converted on the fly to FP8 for this calculation.

Dgrad in paper

$$\frac{\partial L}{\partial x_{FP32}} = \frac{\partial L}{\partial z_{FP8}} * W^T_{FP8}$$

↳ stored as BF16.

weights are at stored at higher precision so that we can confirm, there's update in it. it's learning!!

w grad in paper

$$\frac{dL}{dW} = X^T \frac{\partial L}{\partial Z}$$

- X^T: stored in FP8. ↳ converted from BF16 to FP8
- $\frac{dL}{dW}$: computed & stored in FP32 to ensure numerical stability for weight updates.

Why to do in such approaches?

FP8 Precision

↳ significantly reduces computational & memory cost.
↳ Provides up to 2x speed improvement over BF16 operations.
↳ Ideal for computationally intensive GEMM operations.

Higher Precision Retained (FP32/BF16):

1) Embedding modules, output heads, gating modules, normalization ~~operator~~ attention operators are sensitive. Hence, these are retained in higher precision.
2) Weights & optimizer states also stored in FP32 ensure training stability.

This balance of high-precision storage with low-precision calculation allows the best trade-off between computational efficiency, memory reduction & numerical stability.

b) Fine grained quantization

↳ This method was introduced to protect the precision of weight & dimension when they are quantized to lower precision.

* If any outlier occur it make the number with different value in quantization & dequantization.

Suppose: $x = [2, 3, 4, 500] \times \frac{127}{500} = [0.508, 0.762, 1.016, 127.0]$

Now, quantize to FP8: ↳ assume E4M3 quantization.

$= [0.5, 0.75, 1.0, 127]$

Dequantize back

$$\text{Recovered Tensor} = \frac{[0.5, 0.75, 1.0, 127] \times 500}{127}$$

$\approx [1.97, 2.95, 3.94, 500]$

This occur due to outlier.

So, what to do??

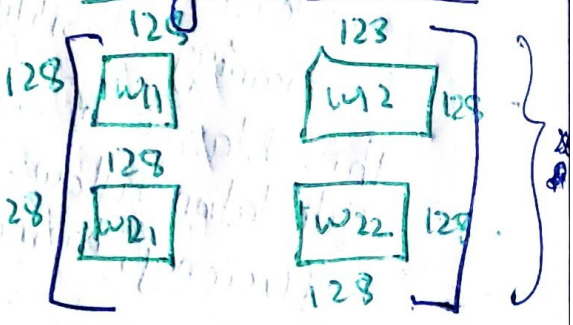
we have billion parameters in weight, in h_dim also, we have large no. of value, so, we will group them in group & the do quantization, so, that outlier in one group didn't affect the precision of another group.

group 1 | group 2

$x = [1.2, 2.1, 2.001, 3 \;|\; 76, 0.1, 1.2, 3]$

scaled by factor 1 | scaled by factor 2.

weight 256 dim



w_{11} → scaling factor 1
w_{12} → scaling factor 2
w_{21} → scaling factor 3
w_{22} → scaling factor 4.

$y = x \times w$

Tensor core. (General matrix mul occur in FP8 inside NVIDIA Tensor core.)

In cuda core.
FP8 * upscaling factor = FP_{32}

~~low precision~~

Increasing accumulation Precision

↳ Using higher-precision arithmetic only for the accumulation (summing) step during neural-network computations. Even though the inputs, weights, & outputs are stored in lower precision.

We know, in almost every neural network layer:

$$y_i = \sum_{j=1}^{d} x_j \cdot w_{ji}$$

This has two different operation.

i) Multiply → $x_j * w_{ji}$
ii) Accumulate (sum) → $\sum_j (\cdot)$.

Deepseek-V3 keeps (1) in low precision & (2) in higher precision.

why low-precision accumulation is dangerous??

↳ suppose Deepseek uses FP8/BF16 for speed.

problem:

↳ FP8/BF16 has few mantissa bits.

↳ when summing thousands of terms, small values get.

- rounded away
- lost entirely.

This causes:

- large numerical error.
- unstable training.
- exploding / vanishing gradients.

WGMMA

↳ wrap group matrix Multiply Accumulate.

↳ It's a low-level GPU instruction family introduced with NVIDIA Hoppe (H100) & it's central to how models like V3 run FP8 efficiently.

key: multiplication in low precision in tensor core & accumulation is done in FP32 in cuda core.

Mantissa over Exponents

In FP8 there are two format → $E4M3$ → 3 mantissa bit higher presion
→ $E5M2$. → lower presicision.
↓ 5 exponent bit high range

Deepseek prefer
Always use
E_4M_3 format.

but prior works prefers:

* Forward pass : E_4M_3
* Backward pass : E_5M_2.

Online quantization

↳ Scaling factor is decided based on the numbers that actually come in.

main problem of offline quantization (Delayed quantization)
↳ we run a callibration dataset and decide the scaling factor at once & after that we do quantization based on that.

* If a rare token appears, overflow or loss.

What deepseek does?
↳ it calculate the scaling factor for the fly comming Tensors.

* In inference, each weight tensor has one (or a small set of) scale value(s), computed ONCE, & reused for all tokens.

so, in inference, we use offline quantization.