

# Deepseek Chat

To ensure that DeepSeek is reading the question accurately, I have it restate the question. This not only helps me verify that it didn't hallucinate the problems, but also helps me verify that it read little details like superscripts correctly from the PDF.



File

Shared files can't be displayed

Help me solve all non-coding questions from this deep learning homework PDF. We will go one question at a time. For each problem, restate the problem first and then solve it. Show all work.

---

Start with Question 1. First, you must rewrite Question 1. Then, solve all parts to it.

## Question 1 – Justifying Scaled-Dot Product Attention

Suppose  $q, k \in \mathbb{R}^d$  are two random vectors, each drawn from  $N(\mu, \sigma^2 I)$ , where  $\mu \in \mathbb{R}^d$  and  $\sigma \in \mathbb{R}^+$ .

Each component  $q_i, k_i$  is i.i.d. normal with mean  $\mu$  and variance  $\sigma^2$ .

### Part (a)

Define  $\mathbb{E}[q^T k]$  in terms of  $\mu, \sigma$  and  $d$ .

**Solution:**

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

Since  $q_i$  and  $k_i$  are independent for each  $i$ :

$$\mathbb{E}[q_i k_i] = \mathbb{E}[q_i] \mathbb{E}[k_i] = \mu \cdot \mu = \mu^2$$

Thus:

Interestingly, DeepSeek already makes a minor error here, because it treats mu as a scalar despite defining it as a vector when restating the question.

$$\mathbb{E}[q^T k] = \sum_{i=1}^d \mu^2 = d\mu^2$$

$\boxed{\mathbb{E}[q^T k] = d\mu^2}$

---

## Part (b)

Assume  $\mu = 0, \sigma = 1$ .

Define  $\text{Var}(q^T k)$  in terms of  $d$ .

**Solution:**

With  $\mu = 0$ , each  $q_i, k_i \sim N(0, 1)$  independent.

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

Because  $q_i$  and  $k_i$  are independent, zero-mean:

$$\text{Var}(q_i k_i) = \mathbb{E}[q_i^2 k_i^2] - (\mathbb{E}[q_i k_i])^2$$

We know  $\mathbb{E}[q_i k_i] = 0$  and  $q_i^2, k_i^2$  are independent, each having mean 1 (since variance is 1 for standard normal, and  $\mathbb{E}[z^2] = 1$ ).

$$\mathbb{E}[q_i^2 k_i^2] = \mathbb{E}[q_i^2] \mathbb{E}[k_i^2] = 1 \cdot 1 = 1$$

So:

$$\text{Var}(q_i k_i) = 1 - 0 = 1$$

Since  $q_i k_i$  are independent across  $i$ , variances add:

$$\text{Var}(q^T k) = \sum_{i=1}^d \text{Var}(q_i k_i) = d \cdot 1 = d$$

$\boxed{\text{Var}(q^T k) = d}$

---

## Part (c)

Continue with  $\mu = 0, \sigma = 1$ .

Let  $s$  be the scaling factor on the dot product.

We want:

$$\mathbb{E} \left[ \frac{q^T k}{s} \right] = 0, \quad \text{Var} \left( \frac{q^T k}{s} \right) = 1$$

Find  $s$  in terms of  $d$ .

**Solution:**

From part (b),  $\text{Var}(q^T k) = d$ .

Scaling by  $s$ :

$$\text{Var} \left( \frac{q^T k}{s} \right) = \frac{\text{Var}(q^T k)}{s^2} = \frac{d}{s^2}$$

We want this to be 1:

$$\frac{d}{s^2} = 1 \quad \Rightarrow \quad s^2 = d \quad \Rightarrow \quad s = \sqrt{d}$$

(We take positive root since  $s > 0$  typically.)

Mean condition:  $\mathbb{E}[q^T k / s] = \frac{1}{s} \mathbb{E}[q^T k] = 0$  already satisfied since mean is 0.

$$s = \sqrt{d}$$

This explains the  $\sqrt{d}$  scaling in scaled dot-product attention: to keep variance of attention logits near 1 for stability.

Now restate Question 2 and solve all parts to it.

## Question 2 – Argmax Attention

Recall that we can think of attention as *queryable softmax pooling*. Here we consider a hypothetical **argmax version** of attention: it returns exactly the

value corresponding to the key most similar to the query, where similarity is measured using the traditional inner product.

## Part (a)

Perform **argmax attention** with the given keys, values, and query:

**Keys  $K$ :**

Here, DeepSeek misreads the PDF and makes the key matrix have only 3 columns instead of 4. This is why having it restate the problem was a good strategy for catching hallucinations.

$$K = \begin{bmatrix} 1 & 0 & 5 \\ 2 & 3 & 0 \\ 0 & 4 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

I interpret this as  $K$  being  $3 \times 3$  with an extra column? Let's check dimensions — based on the given, maybe it's  $3 \times 4$ ?

Looking carefully:

It is written as:

$$\begin{bmatrix} 1 & 0 & 5 \\ 2 & 3 & 0 \\ 0 & 4 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

But that matrix multiplication would give a  $3 \times 1$ , not a key matrix.

I suspect instead the **Keys** matrix is:

$$K = \begin{bmatrix} 1 & 0 & 5 \\ 2 & 3 & 0 \\ 0 & 4 & 0 \end{bmatrix}$$

(3 keys, each of dimension 3) and possibly the second matrix  $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$  is a separate key? But that would make 4 keys total (3 in first matrix, 1 extra).

The "Corresponding Values" are given similarly:

Interestingly, DeepSeek verbalizes its confusion about misreading the matrix and makes a best guess about what the actual matrix would be compared to its original guess

$$V = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 4 & -1 \\ 1 & 3 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

This suggests 3 values (each in  $\mathbb{R}^3$ ) plus a 4th value.

**Let's assume** the Keys matrix  $K$  is  $3 \times 4$ : each column is a key (so 4 keys), each key is length 3.

Thus:

Here, DeepSeek correctly fixes its transcription of the problem

$$K = \begin{bmatrix} 1 & 0 & 5 & 0 \\ 2 & 3 & 0 & 0 \\ 0 & 4 & 0 & 1 \end{bmatrix}$$

Columns:

$$k_1 = [1, 2, 0]^T, k_2 = [0, 3, 4]^T, k_3 = [5, 0, 0]^T, k_4 = [0, 0, 1]^T.$$

**Values**  $V$  is also  $3 \times 4$ :

$$V = \begin{bmatrix} 2 & 1 & 0 & 1 \\ 0 & 4 & -1 & 0 \\ 1 & 3 & 4 & -1 \end{bmatrix}$$

Columns:

$$v_1 = [2, 0, 1]^T, v_2 = [1, 4, 3]^T, v_3 = [0, -1, 4]^T, v_4 = [1, 0, -1]^T.$$

**Query**  $q = [1, 1, 2]^T$ .

**Step 1:** Compute similarities  $q^T k_j$ :

- $q^T k_1 = 1 \cdot 1 + 1 \cdot 2 + 2 \cdot 0 = 3$
- $q^T k_2 = 1 \cdot 0 + 1 \cdot 3 + 2 \cdot 4 = 0 + 3 + 8 = 11$
- $q^T k_3 = 1 \cdot 5 + 1 \cdot 0 + 2 \cdot 0 = 5$
- $q^T k_4 = 1 \cdot 0 + 1 \cdot 0 + 2 \cdot 1 = 2$

Similarities:  $[3, 11, 5, 2]$ .

**Step 2:** Argmax: the maximum similarity is 11 at index 2 (if indexing starts at 1: key 2).

Argmax  $\rightarrow$  one-hot vector:  $[0, 1, 0, 0]$ .

**Step 3:** Output =  $\sum_j (\text{one-hot}_j) v_j = v_2$ .

Prompting DeepSeek to show all work helped it not make any algebra errors in the attention calculations

$$v_2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}$$

---

## Part (b)

How does using **argmax** instead of **softmax** affect our ability to train models involving attention?

**Answer:**

The key issue is **differentiability**.

- In softmax attention, the attention weights are a continuous, differentiable function of the query-key dot products. Gradients can flow back through the softmax to update queries and keys based on the loss.
- In argmax attention, the output is the single value corresponding to the maximum similarity. The operation is discontinuous and has zero gradient almost everywhere with respect to the similarity scores (except at the boundary where the maximum changes, but that's a non-differentiable point).

Therefore, **gradients cannot propagate** through the argmax operation to update queries or keys during training.

Consequences:

- Queries and keys cannot be learned via gradient descent, because argmax provides no useful gradient signal for improving similarity scores.
- The model would either not train at all, or would require alternative training methods (like reinforcement learning with a discrete-action policy gradient) which are more complex and less stable.

Argmax breaks gradient flow, preventing useful learning of queries and keys.

Now restate Question 3 and solve all parts to it.

## Question 3 – Ordinary Softmax Multi-head Attention Implementation

We are given a partial implementation of the `MultiHeadAttention` class and asked to fill in missing blanks.

### Part (a)

Fill in blanks (a) through (h) in the code.

The class definition:

```
python

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, ____(a)__)
        self.W_v = nn.Linear(___(b)__, d_model)
        self.W_o = nn.Linear(d_model, ___(c)__)


```

### Solution for (a):

In standard multi-head attention, each head gets its own key projection of size  $d_k$ . Since  $d_k = d_{\text{model}} // \text{num\_heads}$ , the total output dimension for keys across all heads should be  $d_{\text{model}}$  (because we concatenate heads at the end).

Actually, looking at `self.W_k`: it projects from  $d_{\text{model}}$  to  $d_{\text{model}}$  (same as `W_q`). Yes, because `split_heads` will later split  $d_{\text{model}}$  into `num_heads` heads, each of size  $d_k$ .

So `W_k` output dim =  $d_{\text{model}}$ .

Thus:

$$(a) = d_{\text{model}}$$

We can see that DeepSeek sometimes questions itself when answering almost as a double-check for more accurate results despite this not being Thinking mode.

### Solution for (b):

$W_v$  projects from  $d_{model}$  to  $d_{model}$  (same logic).

Actually, in some implementations,  $W_v$  also outputs  $d_{model}$ , then split into heads.

Thus:

(b) =  $d_{model}$

---

### Solution for (c):

Given in the problem: (c) =  $d_{model}$

---

Now the `scaled_dot_product_attention` method:

```
python

def scaled_dot_product_attention(self, Q, K, V, mask=None):
    attn_scores = torch.matmul(__(d)__, K.transpose(-2, -1))
    attn_scores = attn_scores / math.sqrt(self.d_k)
    if mask is not None:
        attn_scores = attn_scores.masked_fill(mask == 0, -1e9)
    attn_probs = torch.__(e)__(attn_scores, dim=-1)
    output = torch.matmul(attn_probs, __(f)__)
    return output
```

### (d):

We compute  $Q @ K^T$ . So  $Q$  goes in first argument.

Thus:

(d) =  $Q$

---

### (e):

We go from scores to probabilities via softmax.

Thus:

(e) = `softmax`

---

**(f):**

We multiply attention probabilities by  $V$ .

Thus:

$(f) = V$

---

Now `split_heads` method:

```
python

def split_heads(self, x):
    batch_size, seq_length, d_model = x.size()
    return x.view(batch_size, seq_length,
                  ____(g)__, self.d_k).transpose(1, 2)
```

**(g):**

Given in problem:  $(g) = \text{self.num\_heads}$

---

Now `forward` method:

```
python

def forward(self, Q, K, V, mask=None):
    Q = self.split_heads(self.W_q(Q))
    K = self.split_heads(self.W_k(K))
    V = self.split_heads(self.W_v(V))

    attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
    output = self.__(h)__(self.combine_heads(attn_output))
    return output
```

**(h):**

After combining heads, we apply the final output linear projection  $W_o$ .

Thus:

$(h) = W_o$

---

**Final answers for part (a):**

- (a)  $d_{\text{model}}$
- (b)  $d_{\text{model}}$
- (c)  $d_{\text{model}}$
- (d)  $Q$
- (e) softmax
- (f)  $V$
- (g) self.num\_heads
- (h)  $W_o$

## Part (b)

Suppose we change the initialization of `W_v` to:

```
python
self.W_v = nn.Linear(in_features=d_model, out_features=2 * d_model)
```

This increases the length of value vectors.

What other minimal modifications should we apply to make the code run?

**Reasoning:**

If `W_v` outputs `2 * d_model`, then after splitting heads, each head's value dimension `d_v` would be `(2 * d_model) / num_heads`.

But attention's output after `torch.matmul(attn_probs, V)` will have shape `[batch, num_heads, seq_len, d_v]`.

Combining heads via `combine_heads` will produce shape `[batch, seq_len, 2 * d_model]`.

Now the final linear layer `W_o` currently expects input `d_model`, but we give it `2 * d_model`. So we must change `W_o`'s `in_features` to `2 * d_model` (or leave `W_o` unchanged but then dimensions mismatch).

But the problem says *minimal modifications*: likely we keep `W_o` input size = `d_model` (since residual connection adds original input of size `d_model`). So maybe we change `W_v` back? But they explicitly said we increase length of value vectors, so maybe we also change `W_o` to accept `2*d_model` and output `d_model`.

Once again, I find it interesting to see DeepSeek doubting itself and double checking with the question despite this not being Thinking Mode.

But the residual connection requires the attention block output to match input dimension `d_model`. So we must ensure final output dimension is `d_model`. Thus `w_o` should be `nn.Linear(2*d_model, d_model)`.

Also, `self.W_v` in `__init__` already changed to `(d_model, 2*d_model)`. `self.W_k` and `self.W_q` unchanged.

In `forward`, `split_heads` will split `2*d_model` into `num_heads` heads, each head's `d_v = (2*d_model)/num_heads`. `combine_heads` will combine back to `2*d_model`. Then `w_o` maps `2*d_model → d_model`.

Thus minimal modifications:

1. Change `w_o` to `nn.Linear(2*d_model, d_model)`.
2. Possibly adjust `d_k` if needed? No, `d_k` for Q and K unchanged.

Also ensure `scaled_dot_product_attention` works with different `d_v` vs `d_k` — it does, because matmul uses `d_k` only for scaling, and `V`'s last dimension is `d_v`.

DeepSeek was able to one-shot the entirety of Change  $W_o$  to `nn.Linear(2 \times d_{model}, d_{model})`  
Question 3 with no errors which was the longest so far

Now solve Question 4

## Question 4 – Transformer Decoding Optimization

We are given pseudocode for **Multi-Head Attention (MHA)** incremental decoding and **Multi-Query Attention (MQA)** incremental decoding. We need to fill in blanks and analyze complexity.

### Part (a) – (e): Fill in the blanks for MQA code

The MQA function signature:

`python`

For Q4, I do not tell it to restate the problem as it is a very long problem and I was worried that it would inflate the token count and lead to worse performance. This was fine as the model indirectly restates the problems in its answers as it goes through each part 1 by 1.

```
def multiquery_attention_incremental(x, prev_K, prev_V, W_q, W_k, W_v,  
W_o):
```

We're told:

- `prev_K` : shape `[b, n, k]` cached keys (shared across heads)
  - `prev_V` : shape `[b, n, v]` cached values (shared across heads)
  - `W_q` : shape **A** – query projection weights
  - `W_k` : shape **B** – key projection weights
  - `W_v` : shape `[d, v]` – value projection weights
  - `W_o` : shape `[h, v, d]` – output projection weights
- 

### (a) Shape of `W_q` (**A**):

In MQA, each head has its own query projection, but keys and values are shared.

If we have `h` heads, `d` model dim, `q = d/h` query dim per head, then `W_q` should be `[h, d, q]`.

Answer: ○ **[h,d,q]**

$$[h, d, q]$$

---

### (b) Shape of `W_k` (**B**):

Keys are shared across heads → only one key projection for all heads.

Input dim `d`, output dim `k = d/h`.

So shape is `[d, k]`.

Answer: ○ **[d,k]**

$$[d, k]$$

---

### (c) Code for `q` (**C**):

We compute per-head queries:

`x` shape `[b, d]`, `W_q` shape `[h, d, q]`.

We want output [b, h, q].

Using einsum: "bd,hdq->bhq".

Matches: O `torch.einsum("bd,hdq->bhq", x, W_q)`

```
torch.einsum("bd,hdq->bhq", x, W_q)
```

---

#### (d) Code for k (D):

Keys are shared:

x shape [b, d], W\_k shape [d, k].

We want output [b, k] (since shared across heads, no head dim yet).

Using einsum: "bd,dk->bk".

Matches: O `torch.einsum("bd,dk->bk", x, W_k)`

```
torch.einsum("bd,dk->bk", x, W_k)
```

---

#### (e) Code for v (E):

Values are shared:

x shape [b, d], W\_v shape [d, v].

We want output [b, v].

Using einsum: "bd,dv->bv".

Matches: O `torch.einsum("bd,dv->bv", x, W_v)`

```
torch.einsum("bd,dv->bv", x, W_v)
```

---

## Part (f) – (g): Complexity analysis

---

### (f) Computational complexity of MQA for generating a single token with cache of n previous tokens

We reason step-by-step:

#### 1. Projections

- Query:  $b * d * (d/h) * h = b * d^2$  (since  $q = d/h$ ,  $h$  heads  $\rightarrow$  total  $b * d * d$ ). Actually:  $W_q$  shape  $[h, d, q]$ , computing  $q = b * d * q * h = b * d * (d/h) * h = b * d^2$ .
- Key:  $b * d * k$  where  $k = d/h \rightarrow b * d^2 / h$ .
- Value:  $b * d * v$  where  $v = d/h \rightarrow b * d^2 / h$ .  
But in MQA, keys and values are shared, so their projections are computed once, not per head.

Wait carefully:

- $q : b * d * q * h = b * d^2$ .
- $k : b * d * k = b * d^2 / h$ ? Let's check:  $k = d/h$ , so  $b * d * k = b * d * (d/h) = b * d^2 / h$ .
- $v : same as k : b * d^2 / h$ .

So total projection cost:

$$bd^2 + b\frac{d^2}{h} + b\frac{d^2}{h} = bd^2 \left(1 + \frac{2}{h}\right) = O(bd^2)$$

since  $h$  is constant w.r.t  $d$ .

## 2. Attention computation

- $logits : q$  shape  $[b, h, q]$ ,  $new\_K$  shape  $[b, n+1, k]$ .  
Einsum "bhq, bnk -> bhn" cost:  $b * h * q * (n+1)$ .  
But  $q = d/h$ , so  $b * h * (d/h) * n = b * d * n$ .
- $weights$ : softmax over  $n+1 : O(b * h * n) = O(b * d * n / h)$ ? Wait,  $h \sim$  constant, so  $O(b * d * n)$ .
- $o$ : weights  $[b, h, n+1]$  with  $new\_V$   $[b, n+1, v]$ , einsum "bhn, bnv -> bhv":  $b * h * n * v$  with  $v = d/h \rightarrow b * h * n * (d/h) = b * d * n$ .

So attention step total:  $O(b * d * n)$ .

## 3. Output projection

- $o$  shape  $[b, h, v]$ ,  $W_o$  shape  $[h, v, d]$ , einsum "bhv, hvd -> bd": cost  $b * h * v * d$ .  
 $v = d/h$ , so  $b * h * (d/h) * d = b * d^2$ .

**Total compute** = projections + attention + output projection  
 $= O(b * d^2) + O(b * d * n) + O(b * d^2) = O(b * d^2 + b * d * n)$ .

This is same as MHA compute complexity. But note: in MQA, key/value projections are cheaper by factor  $h$ , but it's still  $O(b \cdot d^2)$  because query and output projections dominate.

So answer:

$$O(bd^2 + bdn)$$


---

### (g) Memory access complexity for MQA

We consider data read/write:

#### 1. Weights:

- o  $W_q : [h, d, q]$  size  $h * d * (d/h) = d^2$ .
- o  $W_k : [d, k]$  size  $d * (d/h) = d^2/h$ .
- o  $W_v : [d, v]$  size  $d^2/h$ .
- o  $W_o : [h, v, d]$  size  $h * (d/h) * d = d^2$ .

Total weights:  $O(d^2)$ .

#### 2. Activations for current step (read/write $x, q, k, v, o, y$ ):

- o  $x : [b, d] \rightarrow O(b \cdot d)$ .
- o  $q : [b, h, q] = b * h * (d/h) = b \cdot d$ .
- o  $k : [b, k] = b * (d/h) = b \cdot d/h$ .
- o  $v$ : same as  $k \rightarrow b \cdot d/h$ .
- o  $o : [b, h, v] = b \cdot d$ .
- o  $y : [b, d] = b \cdot d$ .

Total activations:  $O(b \cdot d)$  (since constants like  $h$  ignored in big-O).

#### 3. Cached keys/values:

- o  $prev\_K : [b, n, k] = b \cdot n \cdot (d/h) \rightarrow O(b \cdot n \cdot d / h)$ .
  - o  $prev\_V$ : same  $\rightarrow O(b \cdot n \cdot d / h)$ .
- But  $h$  is constant w.r.t scaling, so  $O(b \cdot n \cdot d)$ .

So total memory access:

$$O(d^2 + bd + bnd)$$

$$O(d^2 + bd + bnd)$$

### Summary for Question 4:

DeepSeek was able to one-shot everything but part (g) correctly, so I try to guide it towards the right answer and it double-checks its work.

- (a)  $[h, d, q]$
- (b)  $[d, k]$
- (c) `torch.einsum("bd, hdq -> bhq", x, Wq)`
- (d) `torch.einsum("bd, dk -> bk", x, Wk)`
- (e) `torch.einsum("bd, dv -> bv", x, Wv)`
- (f)  $O(bd^2 + bdn)$
- (g)  $O(d^2 + bd + bnd)$

For part (g), can the third term be reduced from  $O(bnd)$  in MHA due to shared K/V?

You're absolutely right to question that. Let's carefully compare MHA and MQA for memory access of cached K/V:

#### MHA (multi-head attention):

- `prev_K` shape:  $[b, h, n, k]$  where  $k = d/h$
- Memory for cached keys =  $b * h * n * (d/h) = b \cdot n \cdot d$
- Same for values:  $b \cdot n \cdot d$
- Total for K/V cache:  $O(b \cdot n \cdot d)$

#### MQA (multi-query attention):

- `prev_K` shape:  $[b, n, k]$  where  $k = d/h$
- Memory for cached keys =  $b * n * (d/h) = b \cdot n \cdot d / h$
- Same for values:  $b \cdot n \cdot d / h$
- Total for K/V cache:  $O(b \cdot n \cdot d / h)$

Since  $h$  is a constant (e.g., 8, 16, 32) in big-O notation, we might say  $O(b n d)$ . But if we're comparing MHA vs MQA, the constant factor matters: MQA reduces cache memory by factor  $1/h$ .

But the question asks for "memory access complexity (in Big-O notation)". In asymptotic analysis,  $h$  is constant w.r.t  $b$ ,  $n$ ,  $d$ , so  $O(b n d / h) = O(b n d)$ .

However, looking at their provided note in MHA complexity:

"The memory access complexity is  $O(d^2 + bd + bnd)$  since it is  $O(d^2)$  to read weights,  $O(bd)$  to read/write current token embeddings..., and  $O(bnd)$  to read cached keys/values."

For MQA, since keys/values are shared, the cache size is smaller by factor  $1/h$ . But big-O hides constants, so it's still  $O(b n d)$ .

**But wait:** They're asking specifically about **MQA** in part (g). We should report what's in the code:

- `prev_K`:  $[b, n, k]$  with  $k = d/h$
- `prev_V`:  $[b, n, v]$  with  $v = d/h$
- Memory per cache:  $b * n * (d/h)$
- For both caches:  $2 * b * n * (d/h) = (2/h) * b n d$

So the term is  $O(b n d / h)$ . Since  $h$  is fixed w.r.t scaling of  $b$ ,  $n$ ,  $d$ , this is  $O(b n d)$ .

But if they want to show **improvement** over MHA, they might write it as  $O(b n d / h)$ .

Given their MHA complexity was given as  $O(d^2 + bd + bnd)$  (without  $1/h$  factor), and they said MQA reduces memory access, I think we should write:

$$O(d^2 + bd + bnd/h)$$

But since the question says "in Big-O notation", and constants are dropped, either is technically acceptable. However, since they explicitly mentioned  $h$  in weight shapes, it might be fine to keep the  $/h$ .

Actually, let's check their own comparison: They said MHA arithmetic

$$\text{intensity} = \frac{O(bd^2 + bnd)}{O(d^2 + bd + bnd)}.$$

For MQA, the denominator would change: weights still  $O(d^2)$ , activations

$O(bd)$ , cache  $O(b \cdot n \cdot d / h)$ . So total =  $O(d^2 + bd + b \cdot n \cdot d / h)$ .

I'll put:

$$O(d^2 + bd + bnd/h)$$

to show the improvement from MHA.

---

#### Final Answer for 4(g):

After 1 guidance prompt, the model gets the new correct answer, although with a lot of tokens and thought.

$$O(d^2 + bd + \frac{bnd}{h})$$

Now solve Question 6a

---

## Question 6 – Kernelized Linear Attention (Part I)

We are given the generalized attention equation:

Given the Q6 is the longest

question, I split it into 1 prompt

per part rather than 1 prompt for the whole question to ensure less errors

$$V'_i = \frac{\sum_{j=1}^N \text{sim}(Q_i, K_j) V_j}{\sum_{j=1}^N \text{sim}(Q_i, K_j)}$$

where  $Q_i, K_j \in \mathbb{R}^D$  and  $V_j \in \mathbb{R}^M$ .

---

### Part (a)

Identify the conditions that need to be met by the sim function to ensure  $V'_i$  remains finite (denominator never reaches zero).

#### Reasoning:

The denominator is  $S_i = \sum_{j=1}^N \text{sim}(Q_i, K_j)$ .

If  $S_i = 0$ , then  $V'_i$  is undefined (division by zero).

To keep  $V'_i$  finite, we require:

1.  $\text{sim}(Q_i, K_j) \geq 0$  for all  $i, j$  (non-negativity), because if some terms are negative and could sum to zero, we get division by zero.
2.  $\sum_{j=1}^N \text{sim}(Q_i, K_j) > 0$  for all  $i$  (strictly positive total similarity).

A stronger sufficient condition is:

$$\text{sim}(q, k) > 0 \quad \forall q, k$$

Then each term is positive, so sum is positive.

For example, in softmax attention:

$$\text{sim}(q, k) = \exp\left(\frac{q^T k}{\sqrt{D}}\right) > 0$$

which ensures denominator  $> 0$ .

Thus the key condition: **The similarity function must output positive values for all inputs.**

---

**Answer:**

$$\boxed{\text{sim}(Q_i, K_j) > 0 \quad \forall i, j}$$

Equivalently: sim must be a positive-definite kernel function (in the sense of producing positive values, not necessarily in the Mercer sense).

Now solve part b of Question 6

---

## Question 6 – Part (b)

We consider kernelized attention variants.

---

### (i) Polynomial kernel attention with degree 2

Given:

Polynomial kernel  $\mathcal{K}(x, y) = (x^T y + c)^d$ , where  $c$  is a constant.

In the problem: "any constant factor is set to be 1", and degree  $d = 2$ . So  $c = 1$ ? Wait, "any constant factor is set to be 1" might mean  $c = 1$ .

Standard polynomial kernel:  $\mathcal{K}(x, y) = (x^T y + 1)^2$  for degree 2.

But the problem says "degree of polynomial kernel attention is set to be 2" and "any constant factor is set to be 1". That could mean  $c = 1$ , or perhaps they mean no constant (so  $c = 0$ ) and just  $(x^T y)^2$ .

Given typical attention, we want similarity, and if  $c = 1$ , then even if  $x^T y = 0$ , similarity = 1, which is fine. But sometimes in attention literature, polynomial kernel is just  $(x^T y)^d$  for simplicity.

Let's check: They say "any constant factor is set to be 1". I interpret: The kernel is  $\mathcal{K}(q, k) = (q^T k + 1)^2$ .

Thus:

$$\text{sim}(q, k) = (q^T k + 1)^2$$

$$\boxed{\text{sim}(q, k) = (q^T k + 1)^2}$$

---

## (ii) Feature map for quadratic kernel

For  $\mathcal{K}(q, k) = (q^T k + 1)^2$ , expand:

Let  $q, k \in \mathbb{R}^D$ . Then:

$$(q^T k + 1)^2 = (q^T k)^2 + 2(q^T k) + 1$$

We can write as  $\phi(q)^T \phi(k)$  where:

$$\phi(x) = \begin{bmatrix} 1 \\ \sqrt{2}x_1 \\ \vdots \\ \sqrt{2}x_D \\ x_1^2 \\ x_1x_2 \\ \vdots \\ x_D^2 \end{bmatrix}$$

More systematically:

The feature map includes all monomials of degree  $\leq 2$ :

- Bias term: 1
- Linear terms:  $\sqrt{2}x_i$  for  $i = 1, \dots, D$
- Quadratic terms:  $x_i x_j$  for  $1 \leq i \leq j \leq D$

Thus dimension of  $\phi(x)$  is  $1 + D + \frac{D(D+1)}{2} = \frac{D^2+3D+2}{2}$ .

$$\phi(x) = \begin{bmatrix} 1 \\ \sqrt{2}x \\ \text{vec}(xx^T)_{\text{upper-tri}} \end{bmatrix}$$

where  $\text{vec}(xx^T)_{\text{upper-tri}}$  denotes the vector of all monomials  $x_i x_j$  for  $i \leq j$ .

### (iii) Rewrite kernel attention with feature map

Given  $\mathcal{K}(q, k) = \phi(q)^T \phi(k)$ , rewrite:

$$V'_i = \frac{\sum_{j=1}^N (\phi(Q_i)^T \phi(K_j)) V_j}{\sum_{j=1}^N \phi(Q_i)^T \phi(K_j)}$$

Factor  $\phi(Q_i)^T$  out of sums (since it doesn't depend on  $j$ ):

$$V'_i = \frac{\phi(Q_i)^T \sum_{j=1}^N \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^N \phi(K_j)}$$

Note:  $V_j$  is a row vector (size  $M$ ), so  $\phi(K_j) V_j^T$  is outer product? Wait careful:  $\phi(K_j)$  is column vector of length  $L$ ,  $V_j$  is row vector of length  $M$ , so

$\phi(K_j)V_j$  is  $L \times M$  matrix.

Thus  $\sum_j \phi(K_j)V_j$  is  $L \times M$  matrix.

Then numerator:  $\phi(Q_i)^T [\text{ } L \times M \text{ matrix}]$  gives  $1 \times M$  vector.

Denominator:  $\phi(Q_i)^T \sum_j \phi(K_j)$  (vector length  $L$ ) gives scalar.

So equation:

$$V'_i = \frac{\phi(Q_i)^T \left( \sum_{j=1}^N \phi(K_j)V_j \right)}{\phi(Q_i)^T \left( \sum_{j=1}^N \phi(K_j) \right)}$$

The model tells itself to be careful when multiplying matrices/vectors by checking that the dimensions add up, which is good practice for ensuring correctness.

$$V'_i = \boxed{\frac{\phi(Q_i)^T \left( \sum_{j=1}^N \phi(K_j)V_j \right)}{\phi(Q_i)^T \left( \sum_{j=1}^N \phi(K_j) \right)}}$$

Now solve part c of Question 6

## Question 6 – Part (c)

We analyze time and space complexity for **softmax attention** from Equation (4):

$$V'_i = \frac{\sum_{j=1}^N \exp \left( \frac{Q_i^T K_j}{\sqrt{D}} \right) V_j}{\sum_{j=1}^N \exp \left( \frac{Q_i^T K_j}{\sqrt{D}} \right)}$$

for all  $i \in \{1, \dots, N\}$ .

Given:  $Q, K \in \mathbb{R}^{N \times D}, V \in \mathbb{R}^{N \times M}$ .

### Time complexity (computational cost)

Steps:

1. Compute  $QK^T$ :

- $Q$  is  $N \times D$ ,  $K^T$  is  $D \times N$

- Matrix multiplication cost:  $O(N \cdot D \cdot N) = O(N^2D)$
2. **Scale by  $1/\sqrt{D}$**  and apply **exp** element-wise:
    - $O(N^2)$  operations.
  3. **Row-wise softmax normalization:**
    - For each of  $N$  rows, compute sum of  $N$  exponentials:  $O(N^2)$
    - Then divide each of  $N^2$  entries:  $O(N^2)$
    - Total for softmax:  $O(N^2)$
  4. **Multiply attention weights  $A$  (size  $N \times N$ ) by  $V$  (size  $N \times M$ ):**
    - Matrix multiplication:  $O(N \cdot N \cdot M) = O(N^2M)$

**Total time complexity:**

$$O(N^2D) + O(N^2) + O(N^2) + O(N^2M) = O(N^2(D + M))$$

Since typically  $D \approx M$  in transformers, this is  $O(N^2D)$ .

$O(N^2(D + M))$

or  $O(N^2D)$  if  $D \approx M$ .

---

## Space complexity (memory for backpropagation)

We need to store intermediate tensors for gradient computation:

1. **Inputs:**  $Q, K, V$  each:  $N \times D$  (or  $N \times M$  for  $V$ )
  - But  $Q = XW_Q, K = XW_K, V = XW_V$  from input  $X$ , so storing  $Q, K, V$  is  $O(N(D + D + M)) = O(N(D + M))$ .
2. **Attention scores**  $S = QK^T/\sqrt{D}: N \times N \rightarrow O(N^2)$
3. **Exponential of scores**  $\exp(S): N \times N \rightarrow O(N^2)$
4. **Attention weights (softmax output):**  $N \times N \rightarrow O(N^2)$
5. **Output  $V'$ :**  $N \times M \rightarrow O(NM)$

The **dominating term** is  $O(N^2)$  from attention matrices.

Thus:

$$O(N^2 + N(D + M))$$

or simply  $O(N^2)$  when  $N$  is large.

---

**Final answer for part (c):**

- **Time complexity:**  $O(N^2(D + M))$
- **Space complexity:**  $O(N^2 + N(D + M))$

Now solve part d of Question 6

---

## Question 6 – Part (d)

We use the **linearized polynomial kernel** of degree 2 with feature map  $\phi(\cdot)$ , and the reformulated attention:

$$V'_i = \frac{\phi(Q_i)^T \sum_{j=1}^N \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^N \phi(K_j)}$$

We assume  $M \approx D$ .

---

### Step 1: Dimensions of $\phi(x)$

From part (b)(ii), for quadratic kernel  $(q^T k + 1)^2$ ,

$$\phi(x) \in \mathbb{R}^L \quad \text{where} \quad L = 1 + D + \frac{D(D+1)}{2} = \frac{D^2 + 3D + 2}{2} = O(D^2)$$


---

### Step 2: Compute complexity of kernel attention

We precompute two aggregates **once** for all  $i$ :

$$S_V = \sum_{j=1}^N \phi(K_j) V_j^T \quad (L \times M \text{ matrix})$$

$$S_\phi = \sum_{j=1}^N \phi(K_j) \quad (L \times 1 \text{ vector})$$

**Precomputation cost:**

- Computing all  $\phi(K_j)$  for  $j = 1, \dots, N$ : each  $\phi$  costs  $O(D^2)$  (since it forms all monomials up to degree 2).  
Total:  $O(ND^2)$ .
- Forming  $S_V$ : For each  $j$ , multiply  $\phi(K_j)$  (size  $L = O(D^2)$ ) by  $V_j^T$  (size  $M \approx D$ )  $\rightarrow O(D^3)$  per  $j$  if done naively.  
But careful:  $\phi(K_j)$  is  $L$ -dim,  $V_j$  is  $M$ -dim, outer product is  $L \times M$ .  
Each outer product costs  $O(LM) = O(D^2 \cdot D) = O(D^3)$ .  
Summing over  $N$  terms:  $O(ND^3)$  — too large? Wait, maybe we can do better.

Actually, we don't need to compute full outer products separately then sum.

We can accumulate:

$$S_V = \sum_j \phi(K_j) V_j^T$$

For each  $j$ ,  $\phi(K_j)V_j^T$  is  $L \times M$ , costs  $O(LM) = O(D^3)$ . Over  $N$  steps:  $O(ND^3)$ .

Similarly,  $S_\phi$ : sum of  $L$ -dim vectors over  $N$  steps:  $O(NL) = O(ND^2)$ .

So total precomputation:  $O(ND^3)$  (dominated by  $S_V$  computation).

---

**Then, for each  $i$ :**

1. Compute  $\phi(Q_i)$ :  $O(D^2)$
2. Numerator:  $\phi(Q_i)^T S_V \rightarrow O(LM) = O(D^2 \cdot D) = O(D^3)$
3. Denominator:  $\phi(Q_i)^T S_\phi \rightarrow O(L) = O(D^2)$
4. Division:  $O(M) = O(D)$

Per  $i$  cost:  $O(D^3)$ . For  $N$  queries:  $O(ND^3)$ .

But wait — we already computed  $\phi(Q_i)$  for all  $i$  when processing queries.

Let's separate:

**Total complexity:**

- Compute  $\phi(Q_i)$  for all  $i$ :  $O(ND^2)$
- Compute  $\phi(K_j)$  for all  $j$ :  $O(ND^2)$
- Compute  $S_V$  and  $S_\phi$ :  $O(ND^3)$  (from accumulating outer products)
- For each  $i$ , compute numerator/denominator:  $O(D^3)$  each  $\rightarrow$  total  $O(ND^3)$

So overall time complexity:  $O(ND^3)$ .

But typical  $D$  (e.g., 64, 512) is fixed,  $N$  grows, so  $O(ND^3)$  is **linear in  $N$** .

---

### Step 3: Memory requirement

We need to store:

1.  $\phi(Q_i)$  for all  $i$ :  $N \times L \rightarrow O(ND^2)$
2.  $\phi(K_j)$  for all  $j$ :  $N \times L \rightarrow O(ND^2)$
3.  $S_V: L \times M \rightarrow O(D^2 \cdot D) = O(D^3)$
4.  $S_\phi: L \rightarrow O(D^2)$
5.  $V: N \times M \rightarrow O(ND)$
6. Output  $V': N \times M \rightarrow O(ND)$

The model was more in-depth with its answer explanations at first when answering the time complexity, but did not explain its answers with the space complexity towards the end. This could possibly explain why it got a wrong answer for the memory complexity part.

Dominant term:  $O(ND^2)$  (from storing  $\phi(Q)$  and  $\phi(K)$ ).

---

### Step 4: Final answer

Assuming  $M \approx D$ :

- **Computational cost:**  $O(ND^3)$
- **Memory requirement:**  $O(ND^2)$

Time: $O(ND^3)$ , Memory: $O(ND^2)$
-------------------------------------

**Note:** This is linear in  $N$  for computation and memory, compared to softmax attention's  $O(N^2)$ , but has higher dependence on  $D$ .

For the memory complexity, isn't the  $D^3$  an important term? Also don't the  $K$  and  $Q$  just take up  $O(ND)$  memory rather than  $O(ND^2)$ ?

You're absolutely right — let's carefully re-examine the memory requirements.

---

## Step 1: Do we need to store all $\phi(Q_i)$ and $\phi(K_j)$ ?

In the **kernel attention formulation (Equation 5):**

We have:

$$V'_i = \frac{\phi(Q_i)^T \sum_{j=1}^N \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^N \phi(K_j)}$$

We can **precompute** once:

$$S_V = \sum_{j=1}^N \phi(K_j) V_j^T \quad (L \times M)$$

$$S_\phi = \sum_{j=1}^N \phi(K_j) \quad (L \times 1)$$

---

**Crucially:**

- We **do not need to store all  $\phi(K_j)$  individually** after computing the sums  $S_V$  and  $S_\phi$ .
- We can compute  $S_V$  and  $S_\phi$  in an **online** manner, processing one  $K_j$  at a time, then discarding  $\phi(K_j)$ .
- Similarly, for each  $Q_i$ , we compute  $\phi(Q_i)$  on the fly when computing  $V'_i$ , then discard it.

So the **persistent storage** is:

1. **Original  $Q, K, V$**  for backpropagation:

$$Q, K: N \times D \text{ each} \rightarrow O(ND)$$

$$V: N \times M \rightarrow O(NM) = O(ND)$$

2. **Precomputed aggregates  $S_V$  and  $S_\phi$ :**

$$S_V: L \times M \text{ where } L = O(D^2), M \approx D \rightarrow O(D^3)$$

$$S_\phi: L \times 1 \rightarrow O(D^2)$$

3. **Output  $V'$ :**  $N \times M \rightarrow O(ND)$

4. **Intermediate  $S = QK^T$  not stored** in kernel formulation — we bypass it.

Thus **dominant terms**:

- If  $N$  is very large, the  $O(ND)$  terms dominate:  $O(ND)$ .
- The  $D^3$  term is **constant in  $N$**  but could be large if  $D$  is large.

So **memory complexity** =  $O(ND + D^3)$ .

---

But let's check: For backpropagation, we need  $\phi(Q_i)$  and  $\phi(K_j)$  to compute gradients w.r.t  $Q_i, K_j, V_j$ .

We can recompute  $\phi(Q_i)$  and  $\phi(K_j)$  during backward pass if needed, or store them.

**If we store them:**

- $\phi(Q): N \times L = O(ND^2)$
- $\phi(K): N \times L = O(ND^2)$

That's much larger.

But in kernel attention papers (e.g., Linear Transformers), they often **avoid storing them** by recomputing in backward, trading compute for memory.

The problem likely assumes we store **minimal** intermediates. The explicit aggregates  $S_V, S_\phi$  are  $O(D^3)$  and  $O(D^2)$ .

So **memory** =  $O(ND + D^3)$ .

Given  $N \gg D$  in long sequences, the  $ND$  term dominates.

---

**Wait:** The problem says: "for memory requirement, we need to store any intermediate results for backpropagation, including all  $Q, K, V$ ". So we must store  $Q, K, V$ : that's  $O(ND + NM) = O(ND)$ .

But we don't necessarily need to store  $\phi(Q), \phi(K)$  if we can recompute. However, during **forward pass**, we need  $\phi(Q_i)$  for each  $i$  — but we compute and discard.

Thus:

$$\begin{aligned}\text{Memory} &= O(ND) \text{ (for } Q, K, V) + O(D^3) \text{ (for } S_V) + O(D^2) \text{ (for } S_\phi) \\ &= O(ND + D^3)\end{aligned}$$

If  $N \gg D$ , then  $O(ND)$  dominates.

---

Given  $M \approx D$ , final:

$\text{Memory: } O(ND + D^3)$

---

**So corrected final for (d):**

- **Computational cost:**  $O(ND^3)$  (from forming  $S_V$  and per-query multiplies)
- **Memory requirement:**  $O(ND + D^3)$

After thinking out loud, the model is able to fix its answer. The “out loud thinking” is almost like deep thinking mode as it helps it catch flaws in its logic.