

ChatGPT 5.1 ~

[↑ Share](#)
[🔗 Add people](#)
...
Question 1:

My prompt approach was to have the LLM follow the steps I take when approaching problems. Also, I chose to attach screenshots of the problem to prevent formatting issues and so I could in problems one by one.

1. Justifying Scaled-Dot Product Attention

Suppose $q, k \in \mathbb{R}^d$ are two random vectors with $q, k \stackrel{\text{iid}}{\sim} \mathcal{N}(\mu, \sigma^2 I)$, where $\mu \in \mathbb{R}^d$ and $\sigma \in \mathbb{R}^+$. In other words, each component q_i of q is drawn from a normal distribution with mean μ and standard deviation σ , and the same is true for k .

- Define $\mathbb{E}[q^T k]$ in terms of μ, σ and d .
- Considering a practical case where $\mu = 0$ and $\sigma = 1$, define $\text{Var}(q^T k)$ in terms of d .
- Continue to use the setting in part (b), where $\mu = 0$ and $\sigma = 1$. Let s be the scaling factor on the dot product. Suppose we want $\mathbb{E}[q^T k]$ to be 0, and $\text{Var}(q^T k)$ to be σ^2 . What should s be in terms of d ?

I'm working on this problem. For each subpart, please: 1. Restate the problem in your own words. 2. Explain the main idea / plan before doing any calculations. 3. Show the full step-by-step solution, with no big jumps. 4. Briefly summarize the final answer and why it's correct. Keep explanations clear and intuitive. Make sure to include thorough explanations. Remember to solve all subparts associated with the problem and give your final answer. I have attached screenshots of each problem.



Thought for 46s >

Let's go through each subpart carefully.

We're given

- $q, k \in \mathbb{R}^d$
- $q, k \stackrel{\text{iid}}{\sim} \mathcal{N}(\mu, \sigma^2 I)$.

That means each coordinate q_i and k_i is independent and $q_i \sim \mathcal{N}(\mu, \sigma^2)$, $k_i \sim \mathcal{N}(\mu, \sigma^2)$, and q and k are independent of each other.

Also

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

(a) Compute $\mathbb{E}[q^\top k]$ in terms of μ, σ, d .**1. Restate**

Find the expected value of the dot product of these two random vectors.

2. Plan

Use linearity of expectation and independence:

1. Write $q^\top k = \sum_{i=1}^d q_i k_i$.
2. Compute $\mathbb{E}[q_i k_i]$ for a single coordinate.
3. Sum over all coordinates.

3. Steps

This part is very concise and clear. However, it is slightly misaligned with the notation of the problem.

Should say "we compute $\mathbb{E}[q_i k_i] = \mathbb{E}[q_i] \mathbb{E}[k_i] = \mu_i^2$ for each coordinate, then sum."

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

By linearity of expectation:

$$\mathbb{E}[q^\top k] = \sum_{i=1}^d \mathbb{E}[q_i k_i].$$

Since q and k are independent, each pair (q_i, k_i) is independent, so

$$\mathbb{E}[q_i k_i] = \mathbb{E}[q_i] \mathbb{E}[k_i].$$

Each coordinate has mean μ , so

$$\mathbb{E}[q_i] = \mu, \quad \mathbb{E}[k_i] = \mu$$

and thus

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

Now sum over all dimensions:

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

(If you treat μ as a mean vector $\mu \in \mathbb{R}^d$ with components μ_i , the same derivation gives $\sum_i \mu_i^2 = \|\mu\|^2$.)

4. Summary

The expected dot product is

This should have been given
as the main answer.
↓

$$\boxed{\mathbb{E}[q^\top k] = d\mu^2 (\text{=} \|\mu\|^2 \text{ if you use a mean vector})}.$$

It doesn't depend on σ because expectation only depends on the means.

(b) With $\mu = 0, \sigma = 1$, find $\text{Var}(q^\top k)$ in terms of d .

1. Restate

Now each coordinate is standard normal: $q_i, k_i \sim \mathcal{N}(0, 1)$. We need the variance of the dot product $q^\top k$.

2. Plan

1. Again write $q^\top k = \sum_{i=1}^d q_i k_i$.
2. Compute $\text{Var}(q_i k_i)$ for one coordinate.
3. Argue that different coordinates' products are independent, so $\text{Var}(\sum_i q_i k_i) = \sum_i \text{Var}(q_i k_i)$.
4. Multiply by d .

The high level plan is
correct and very clear.
It is well-structured.

3. Steps

Let

$$Z_i = q_i k_i, \quad \text{so} \quad q^\top k = \sum_{i=1}^d Z_i.$$

First compute $\text{Var}(Z_i)$.

Because $\mu = 0$, we have

$$\mathbb{E}[q_i] = 0, \quad \mathbb{E}[k_i] = 0.$$

Since q_i and k_i are independent:

$$\mathbb{E}[Z_i] = \mathbb{E}[q_i k_i] = \mathbb{E}[q_i] \mathbb{E}[k_i] = 0 \cdot 0 = 0.$$

So

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

Now,

$$Z_i^2 = (q_i k_i)^2 = q_i^2 k_i^2.$$

Again using independence,

$$\mathbb{E}[Z_i^2] = \mathbb{E}[q_i^2 k_i^2] = \mathbb{E}[q_i^2] \mathbb{E}[k_i^2].$$

The derivation differs slightly when computing $\text{Var}(q_i k_i)$. The official solution uses the general Variance expansion. This solution uses independence. However, both are valid approaches.

For a standard normal $X \sim \mathcal{N}(0, 1)$, $\mathbb{E}[X^2] = \text{Var}(X) = 1$. Thus

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

so

$$\mathbb{E}[Z_i^2] = 1 \cdot 1 = 1,$$

and therefore

$$\text{Var}(Z_i) = 1.$$

Next, relate $\text{Var}(q^\top k)$ to $\text{Var}(Z_i)$.

The pairs (q_i, k_i) are independent across different coordinates i , so the products Z_i are independent across i . For a sum of independent random variables,

$$\text{Var}\left(\sum_{i=1}^d Z_i\right) = \sum_{i=1}^d \text{Var}(Z_i).$$

Hence

$$\text{Var}(q^\top k) = \sum_{i=1}^d \text{Var}(Z_i) = \sum_{i=1}^d 1 = d.$$

4. Summary

With zero mean and unit variance components, each coordinate contributes variance 1 to the dot product, and they add up:

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

The final answer matches the official solutions.

So as the dimension d increases, the variability of the dot product grows linearly with d .

(c) Choose a scaling factor s so that $\mathbb{E}\left[\frac{q^\top k}{s}\right] = 0$ and $\text{Var}\left(\frac{q^\top k}{s}\right) = 1$.

We stay in the setting $\mu = 0, \sigma = 1$.

1. Restate

We scale the dot product by a factor s . We want the scaled random variable $\frac{q^\top k}{s}$ to have mean 0 and variance 1. Find s in terms of d .

2. Plan

We already know from (b):

- $\mathbb{E}[q^\top k] = 0$,
- $\text{Var}(q^\top k) = d$.

The plan is very simple and clear. It mentions what we are using from part b which is helpful.

For any constant s ,

- $\mathbb{E}\left[\frac{X}{s}\right] = \frac{1}{s}\mathbb{E}[X]$,
- $\text{Var}\left(\frac{X}{s}\right) = \frac{1}{s^2}\text{Var}(X)$.

Apply this with $X = q^\top k$, then solve for s so that the variance becomes 1.

3. Steps

Let

$$Y = \frac{q^\top k}{s}.$$

Mean

$$\mathbb{E}[Y] = \mathbb{E}\left[\frac{\mathbf{q}^\top \mathbf{k}}{s}\right] = \frac{1}{s}\mathbb{E}[\mathbf{q}^\top \mathbf{k}].$$

From (b), $\mathbb{E}[\mathbf{q}^\top \mathbf{k}] = 0$. So

$$\mathbb{E}[Y] = \frac{1}{s} \cdot 0 = 0.$$

So the mean is 0 automatically for any s . That condition is already satisfied.

Variance

$$\text{Var}(Y) = \text{Var}\left(\frac{\mathbf{q}^\top \mathbf{k}}{s}\right) = \frac{1}{s^2} \text{Var}(\mathbf{q}^\top \mathbf{k}) = \frac{1}{s^2} \cdot d.$$

These steps match the approach the solutions take, but in more detail.

We want $\text{Var}(Y) = 1$, so set

$$\frac{d}{s^2} = 1.$$

Solve for s :

$$s^2 = d \Rightarrow s = \sqrt{d} \quad (\text{take } s > 0 \text{ as a scaling factor}).$$

The summary of why we need the scaling factor is very helpful in understanding the general intuition of the problem.

4. Summary

To make the scaled dot product have variance 1 (and mean 0), we must divide by \sqrt{d} .

$s = \sqrt{d}$. The final answer matches the official solutions.

This is exactly why scaled dot-product attention uses $\frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{d}}$: without this scaling, the variance of the scores grows with dimension d , making them too large and leading to very peaked softmax distributions as d increases.

◻ ↗ ⚡ ⌄ ↘ ⌂ ...

Question 2:

I used the same prompting style for the second question.

I'm working on this problem. For each subpart, please: 1. Restate the problem in your own words. 2. Explain the main idea / plan before doing any calculations. 3. Show the full step-by-step solution, with no big jumps. 4. Briefly summarize the final answer and why it's correct. Keep explanations clear and intuitive. Make sure to include thorough explanations. Remember to solve all subparts associated with the problem and give your final answer. I have attached screenshots of each problem.

Thought for 1m 13s >

Let's work through both parts carefully.

Setup: What is argmax attention?

We have:

- Keys $k_1, k_2, k_3, k_4 \in \mathbb{R}^3$
- Corresponding values $v_1, v_2, v_3, v_4 \in \mathbb{R}^3$
- A query $q \in \mathbb{R}^3$

Argmax attention rule:

1. Compute the similarity scores $s_i = q^\top k_i$ for each key.
2. Find the index $j = \arg \max_i s_i$.
3. Output the value v_j corresponding to the winning key.

So instead of a weighted average (like softmax attention), we directly pick one value.

(a) Compute the output of argmax attention for the given q , keys, and values.**1. Restate**

We're given:

Keys

$$k_1 = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}, \quad k_2 = \begin{bmatrix} 0 \\ 3 \\ 4 \end{bmatrix}, \quad k_3 = \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix}, \quad k_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Values

$$v_1 = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}, \quad v_3 = \begin{bmatrix} 0 \\ -1 \\ 4 \end{bmatrix}, \quad v_4 = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

Query

$$q = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

We must run argmax attention: find which key is most similar to q (by dot product), then return that key's value.

2. Plan

1. Compute the dot products $s_i = q^\top k_i$ for $i = 1, 2, 3, 4$.
2. Identify which score is largest (that index is the argmax).
3. Output the corresponding value vector v_i .

The setup focuses on the required computations. It is clear, correct, and complete.

3. Step-by-step calculations

Dot product with k_1 :

$$s_1 = q^\top k_1 = [1 \ 1 \ 2] \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} = 1 \cdot 1 + 1 \cdot 2 + 2 \cdot 0 = 1 + 2 + 0 = 3$$

Dot product with k_2 :

$$s_2 = q^\top k_2 = [1 \ 1 \ 2] \begin{bmatrix} 0 \\ 3 \\ 4 \end{bmatrix} = 1 \cdot 0 + 1 \cdot 3 + 2 \cdot 4 = 0 + 3 + 8 = 11$$

Dot product with k_3 :

All 4 dot products are computed correctly.

$$s_3 = q^\top k_3 = [1 \ 1 \ 2] \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix} = 1 \cdot 5 + 1 \cdot 0 + 2 \cdot 0 = 5$$

Dot product with k_4 :

$$s_4 = q^\top k_4 = [1 \ 1 \ 2] \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 1 \cdot 0 + 1 \cdot 0 + 2 \cdot 1 = 2$$

So the similarity scores are:

$$(s_1, s_2, s_3, s_4) = (3, 11, 5, 2).$$

The largest score is 11, which is s_2 .

Therefore:

$$\arg \max_i s_i = 2.$$

So argmax attention chooses key 2, and we output value 2:

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

One difference is that the official solutions explains why the "weighted sum" is trivial, but this solution explicitly assumes it. It would be clearer if mentioned.

4. Summary for (a)

The key most similar to the query q is k_2 , because its dot product with q (11) is the largest.

Therefore, the argmax attention output is the corresponding value:

$\text{output} = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}.$

The output is correct.

(b) How does using argmax instead of softmax affect training?

1. Restate

We now replace the usual *softmax* in attention with *argmax*. The layer's output is exactly one value vector, with no mixing.

Question: **How does this design choice affect our ability to train models with attention using gradient-based methods (backpropagation)?**

2. Plan

Think about:

The plan gives a good hint for the conceptual issue.

- What the softmax does mathematically to the scores.
- Why softmax is good for gradients.
- What argmax does instead, and how that affects gradients w.r.t. the scores, and thus w.r.t. queries/keys.

3. Explanation

Softmax attention:

- We compute scores $s_i = q^\top k_i$.
- We apply softmax:

$$\alpha_i = \frac{e^{s_i}}{\sum_j e^{s_j}}$$

This matches the correct reasoning and aligns with the official solution.

- We output a smooth weighted sum:

$$\text{out} = \sum_i \alpha_i v_i.$$

- Each α_i is a differentiable function of the scores s_i , and each s_i is a differentiable function of q and k_i .
- So when we backpropagate, **every key and the query receives some gradient**, and the gradient varies smoothly as scores change. This lets the model gradually adjust keys and queries so that "better" keys get higher scores.

Argmax attention:

- We still compute scores s_i .
- Then we take

This solution goes into more detail in comparison to the official solution. It is more step-by-step and gives mechanical insight.

$$j = \arg \max_i s_i, \quad \text{out} = v_j.$$

- The output is *piecewise constant* with respect to the scores: as long as the same key stays the argmax, small changes in scores don't change which value we output.
- Mathematically, the argmax function is:
 - **Non-differentiable at ties** (when two scores are equal),
 - **Zero gradient almost everywhere else** (because the output index stays constant under small perturbations).

In terms of mathematical accuracy, this explicitly states derivative issues, explains why argmax is non-differentiable, and is more rigorous.

So in backprop:

- The loss depends on $\text{out} = v_j$.
- But changing scores slightly (changing q or keys k_i) typically doesn't change which index is chosen — until one score overtakes another, at which point it jumps discontinuously.
- The derivative of argmax w.r.t. scores is effectively zero almost everywhere, and undefined at the jump.
- Therefore, **no useful gradient flows back into the query or keys**. Gradient descent has nothing to work with to improve them.

We can still learn the values v_j after the argmax (since they're directly in the forward path), but we cannot learn how to *select* better keys or tune the queries using standard backprop.

To train such a model, we'd need non-standard tricks (e.g., reinforcement learning, straight-through estimators, Gumbel-softmax, etc.), which is exactly why real transformers use softmax instead of argmax.

4. Summary for (b)

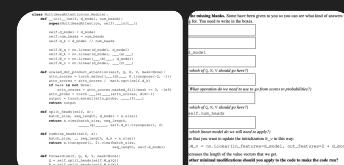
Using **argmax** instead of **softmax** makes the attention operation *non-differentiable* with respect to the similarity scores (and thus with respect to queries and keys). The output changes only when the winning key index changes, so its gradient is zero almost everywhere and undefined at the boundaries. As a result, gradients cannot flow back through the attention scores, so we can't effectively learn or improve the queries and keys with standard backpropagation. Softmax avoids this problem by providing smooth, differentiable weights that let gradients flow to all keys.

Summary is very clear and highlights main issue.



Question 3:
Again, same prompt

I'm working on this problem. For each subpart, please: 1. Restate the problem in your own words. 2. Explain the main idea / plan before doing any calculations. 3. Show the full step-by-step solution, with no big jumps. 4. Briefly summarize the final answer and why it's correct. Keep explanations clear and intuitive. Make sure to include thorough explanations. Remember to solve all subparts associated with the



problem and give your final answer. I have attached screenshots of each problem.

Thought for 2m 32s >

Let's go through this one carefully and fill in all the blanks, then handle part (b).

Part (a) – Fill in the missing pieces of the multi-head attention code

1. Restate

We're given a partial PyTorch implementation of *ordinary softmax multi-head attention*:

- `__init__` sets up linear layers for W_q , W_k , W_v , W_o .
- `scaled_dot_product_attention` does the softmax attention for a single multi-head tensor.
- `split_heads` and `combine_heads` handle reshaping.
- `forward` wires everything together.

There are blanks labeled (a)–(h) that we must fill so the code matches the standard multi-head attention behavior.

2. Main idea / plan

Recall standard multi-head attention for model size `d_model` and `num_heads` heads:

- We first project the input to queries/keys/values of size `d_model` using linear layers `W_q`, `W_k`, `W_v`.
- Then we reshape each into `(batch, num_heads, seq_len, d_k)` where
`d_k = d_model // num_heads`.
- Scaled dot-product attention:

$$\begin{aligned} \text{scores} &= Q @ K^T / \sqrt{d_k} \\ \text{probs} &= \text{softmax}(\text{scores}) \\ \text{output} &= \text{probs} @ V \end{aligned}$$
- Finally we recombine heads and pass through a final linear layer `W_o` (still outputting size `d_model`).

So we'll choose each blank so that:

- All linear layers take/produce tensors of dimension `d_model`.
- Attention uses `Q` in QK^T and multiplies probabilities by `V`.
- We use `softmax` to convert scores to probabilities.
- We split into `self.num_heads` heads.
- After attention, we apply `W_o`.

The plan correctly outlined that each blank corresponds to a standard multi-head attention component. The plan mentions the relevant transformer formulas.

3. Fill each blank, step by step

I'll rewrite the relevant bits with the blanks filled and explain each.

In `__init__`

```
python
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)__)


```

Copy code

For standard transformers, all of these transform from `d_model` to `d_model`, so:

- (a) = `d_model` – output of `W_k` has size `d_model`.
- (b) = `d_model` – input to `W_v` is `d_model`.

- (c) is already given as `d_model` – output of `W_o` is `d_model`.

So:

python

Copy code

```
self.W_k = nn.Linear(d_model, d_model)          # (a)
self.W_v = nn.Linear(d_model, d_model)          # (b)
self.W_o = nn.Linear(d_model, d_model)          # (c)
```

In `scaled_dot_product_attention`

python

Copy code

```
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)__)


```

- To compute attention scores we need QK^T , so (d) must be `Q`.
- To convert scores to probabilities we use `softmax`, so (e) is `softmax`.
- After probabilities we multiply by `V`, so (f) must be `V`.

Thus:

python

Copy code

```
attn_scores = torch.matmul(Q, K.transpose(-2, -1))      # (d)
...
attn_probs = torch.softmax(attn_scores, dim=-1)        # (e)
output = torch.matmul(attn_probs, V)                   # (f)
```

Each blank is matched to the correct variable / function. Each step is also connected to the actual multi-head attention equations. The mapping from pseudocode to math is clear.

In `split_heads`

python

Copy code

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

We want to reshape from `(batch, seq, d_model)` to `(batch, num_heads, seq, d_k)`, so:

- `(g) = self.num_heads`.

python

Copy code

```
return x.view(batch_size, seq_length,
              self.num_heads, self.d_k).transpose(1, 2)  # (g)
```

In `forward`

python

Copy code

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

After we recombine heads, we must apply the final linear layer `W_o`. So:

- $(h) = W_o$.

python

 Copy code

```
output = self.W_o(self.combine_heads(attn_output)) # (h)
```

4. Summary of part (a)

The completed blanks are:

- (a) d_{model}
- (b) d_{model}
- (c) d_{model} (given)
- (d) Q
- (e) softmax
- (f) V
- (g) $self.num_heads$ (given)
- (h) W_o

All final blanks were filled in correctly.

This makes the implementation match standard multi-head scaled dot-product attention: queries attend to keys via QK^T , we softmax to get probabilities, weight the values V , then recombine heads and project back with W_o .

Part (b) – Changing W_v to output length $2 \cdot d_{model}$

1. Restate

Now we change the value projection:

python

 Copy code

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

So each value vector is twice as long as before.

Question: What minimal additional modifications are needed so the code still runs correctly, especially remembering that attention blocks sit inside residual connections?

2. Main idea / plan

Key consequences:

1. After we compute $V = W_v(x)$, its last dimension is now $2 * d_{model}$ instead of d_{model} .
2. After splitting into heads and recombinining them, the concatenated per-position vector will also be of length $2 * d_{model}$.
3. But the residual connection around the attention block expects the final output of the block to have dimension d_{model} so that it can be added back to the original input.

So:

- Inside the block, we can allow values to be longer ($2 \cdot d_{model}$).
- But before we add the residual, we must project back down to d_{model} .

This means:

- The final linear layer W_o must take $2 * d_{model}$ -dimensional input and output d_{model} -dimensional vectors.

We also must make sure the recombined tensor given to W_o actually has last dimension $2 * d_{model}$ (which it will once we adjust the reshaping to match the new value size).

Summarizes the main problem of changing W_v changes the downstream dimensionality of the concatenated heads. The plan explicitly lists the requirement of output dim = d_{model}

This explains the reasoning behind why only W_o must adjust.

3. Concrete modifications

1. Update the final output projection w_o :

python

 Copy code

```
# before:  
self.W_o = nn.Linear(d_model, d_model)  
  
# after change to W_v:  
self.W_o = nn.Linear(2 * d_model, d_model)
```

Now `w_o` takes the longer concatenated value vectors and maps them back down to `d_model`, so the residual connection (input + attention_output) still matches in shape.

2. Ensure `combine_heads` reshapes to match `2 * d_model`:

After changing `w_v`, each position's concatenated heads will have length $2 * d_{\text{model}}$. So in `combine_heads`, the `view` call should be consistent with that. In practice this means updating the last dimension in `view` to match the actual product of `num_heads` and per-head value dimension (which is now $2 * d_{\text{model}}$), so that what we feed into `w_o` has size `(batch_size, seq_length, 2 * d_model)`.

Conceptually:

python

 Copy code

```
# combine_heads should produce (... , 2 * d_model)
x = x.transpose(1, 2).contiguous()
batch_size, seq_length, combined_dim = x.size()
# here combined_dim = 2 * d_model
```

The walkthrough of how head concatenation sizes change was correct and helpful. The logic for fixing residual connections is clear.

Then W_o maps $2 * d_{model} \rightarrow d_{model}$

4. Summary of part (b)

Because \mathbf{W}_V now outputs vectors of length $2d_{\text{model}}$, the concatenated multi-head output also has length $2d_{\text{model}}$. To keep the transformer block compatible with a residual connection around it (which requires the block output to have the same dimension as its input, d_{model}):

- Modify the final projection layer to

$$W_e : \mathbb{R}^{2d_{\text{model}}} \rightarrow \mathbb{R}^{d_{\text{mode}}}$$

i.e.

The final answer exactly matches the official answer and includes justification.

python

 Copy code

```
self.W_o = nn.Linear(2 * d_model, d_model)
```

- And ensure the recombined heads tensor fed into `W_o` is reshaped to have last dimension `2 * d_model`.

This way, internal value vectors are longer, but the final attention output still has dimension `d_model`, so the residual addition `x + attention_output` is well-defined.

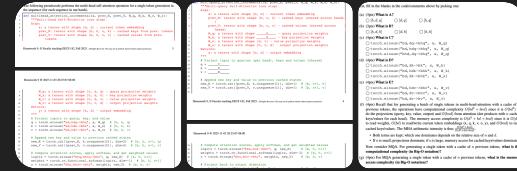


Question 4:

The first time I prompted the LLM caused it to crash.

I'm working on this problem. For each subpart, please: 1. Restate the problem in your own words. 2. Explain the main idea / plan before doing any calculations. 3. Show the full step-by-step solution, with no big jumps. 4. Briefly summarize the final answer and why it's correct. Keep explanations clear and intuitive. Make sure to include thorough explanations. Remember to solve all subparts associated with the problem and give your final answer. I have attached screenshots of each problem.

Stopped thinking >



I tried re-promting with the same approach again.

I'm working on this problem. For each subpart, please: 1. Restate the problem in your own words. 2. Explain the main idea / plan before doing any calculations. 3. Show the full step-by-step solution, with no big jumps. 4. Briefly summarize the final answer and why it's correct. Keep explanations clear and intuitive. Make sure to include thorough explanations. Remember to solve all subparts associated with the problem and give your final answer. I have attached screenshots of each problem.

Thought for 5m 7s >

Let's decode what's going on and then fill in all the blanks.

We have:

- A **multi-head attention incremental** function (already written).
- A **multi-query attention incremental** function that we have to finish.
- Then multiple-choice questions (A–E) about shapes / einsums, and (f,g) about computational and memory complexity.

In MQA all heads share the *same* keys and values; only queries are per-head.

(a) What is A? (shape of W_q in MQA)

1. Restate

We need the shape of the query projection weights W_q in MQA.

Options:

- $[h, d, q]$
- $[d, q]$
- $[h, q]$

2. Idea

Queries are still **per head**, exactly like in regular MHA: one projection matrix per head from a d -dim embedding to a q -dim vector (usually $q = d/h$). So we need an extra head dimension h .

3. Reasoning

This solution correctly recognized the shape of W_q , but could have explained that it is because it maintains separate weights per head.

- Input x has shape $[b, d]$.
- We want queries q with shape $[b, h, q]$ (one query per head).
- The einsum we'll use later is "bd,hdq->bhq", which requires W_q to have indices (h, d, q) .

So the only compatible shape is $[h, d, q]$.

4. Answer

$$A = [h, d, q]$$

Final answer is correct
and matches the official
output exactly.

(b) What is B? (shape of W_k in MQA)

1. Restate

Find the shape of the **shared** key projection weights W_k .

Options:

- $[h, d, k]$
- $[d, k]$
- $[h, k]$

2. Idea

Keys are **shared across heads**, so we do *not* want a separate matrix per head. That means no leading h dimension.

This correctly explains why
there is no h dim.

3. Reasoning

- Input is still $x \in \mathbb{R}^{b \times d}$.
- We want keys k with shape $[b, k]$ (no head dim).
- The einsum we will use is "bd,dk->bk", which expects W_k of shape $[d, k]$.

So B must be $[d, k]$.

4. Answer

$$B = [d, k]$$

This is fully correct.

(c) What is C? (expression to compute q)

We want to fill in:

python

Copy code

q = ____C____

Options:

1. `torch.einsum("bd,dq->bhq", x, W_q)`
2. `torch.einsum("bd,hdq->bhq", x, W_q)`
3. `torch.einsum("bhd,dq->bhq", x, W_q)`

1. Restate

Choose the einsum that gives queries of shape $[b, h, q]$ from $x : [b, d]$ and $W_q : [h, d, q]$.

2. Idea

Einsum subscripts must match the shapes:

- x : indices `b d`.
- W_q : indices should be `h d q`.
- Result: `b h q`.

3. Check options

- (1) uses "bd,dq->bhq" : W_q would be $[d, q]$, and result would only have indices $b \ h \ q$ if we manufacture an extra h , which we don't have. So wrong.
- (2) "bd,hdq->bhq" : matches exactly:
 - $bd \leftrightarrow x$,
 - $hdq \leftrightarrow W_q$,
 - result bhq . ✓
- (3) "bhd,dq->bhq" : would require x to be $[b, h, d]$, which it is not.

*Einsum reasoning is correct.
The solution walks through
each option which is helpful.*

4. Answer

$$C = \text{torch.einsum}("bd,hdq->bhq", x, W_q)$$

*This matches the
official solutions.*

This produces q with shape $[b, h, q]$.

(d) What is D? (expression to compute k)

We fill in:

python

Copy code

$$k = \underline{\quad} D \underline{\quad}$$

Options:

1. `torch.einsum("bd,dk->bk", x, W_k)`
2. `torch.einsum("bd,hdk->bhk", x, W_k)`
3. `torch.einsum("bd,dk->bhk", x, W_k)`

1. Restate

Compute shared keys from $x : [b, d]$ and $W_k : [d, k]$ so that k has shape $[b, k]$.

*The solution recognizes that
keys are projected exactly
like queries, but using W_k .*

2. Idea

We want result indices $b \ k$ and we know W_k has $d \ k$.

3. Check options

- (1) "bd,dk->bk" :
 - $bd \leftrightarrow x$, $dk \leftrightarrow W_k$, result $bk \rightarrow$ shape $[b, k]$. ✓
- (2) would need W_k shape $[h, d, k]$ and produce bhk , which we don't want in MQA.
- (3) makes result bhk but our einsum has no h on inputs.

*Could mention shape reasoning
explicitly. $[b,d] \times [d,k] \rightarrow [b,k]$, then
heads added $\rightarrow [b,h,k]$*

4. Answer

$$D = \text{torch.einsum}("bd,dk->bk", x, W_k)$$

The solution is correct.

This gives k with shape $[b, k]$, which we then `unsqueeze(1)` and concatenate into `prev_K` of shape $[b, n, k]$.

(e) What is E? (expression to compute v)

We fill:

python

Copy code

$$v = \underline{\quad} E \underline{\quad}$$

Options:

1. `torch.einsum("bd,hdv->bhv", x, W_v)`
2. `torch.einsum("bd,dv->bhv", x, W_v)`

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

1. Restate

Compute shared values from $x : [b, d]$ and $W_v : [d, v]$ so that v has shape $[b, v]$.

2. Idea

Like keys, values are shared across heads; so we want result $b \cdot v$.

3. Check options

- (1) wants W_v with $h \cdot d \cdot v$ and would give $b \cdot h \cdot v$; that's per-head, not shared.
- (2) "bd,dv->bhv": indices have b on first input and d, v on second, but the result $b \cdot h \cdot v$ includes an h that isn't present anywhere → invalid.
- (3) "bd,dv->bv":
 - $bd \leftrightarrow x, dv \leftrightarrow W_v$, result $b \cdot v \rightarrow [b, v]$. ✓

The answer matches the official solution and the explanation is clear.

4. Answer

$$E = \text{torch.einsum("bd,dv->bv", x, W_v)}$$

Then we `unsqueeze(1)` and concatenate into `prev_V: [b, n, v]`.

(f) Computational complexity for MQA (one new token, cache length n)

1. Restate

For multi-head attention (MHA) they gave:

- Compute cost: $O(bd^2 + bnd)$
- (from projections q, k, v, o : $O(bd^2)$, plus attention over n cached tokens: $O(bnd)$)

We must compute the same for MQA, generating one token with a cache of n previous tokens.

2. Idea

Compare work done in MQA to MHA:

The breakdown is very clear with computing all projections first, then dot products with cached k/v.

- Projections:
 - q : still per-head $W_q : [h, d, q] \rightarrow$ cost $O(bd^2)$ (same).
 - k, v : now shared $W_k : [d, k], W_v : [d, v] \rightarrow$ each costs $O(bdk) \sim O(bd^2/h)$ instead of $O(bd^2)$. So projections are **slightly cheaper**, but still $O(bd^2)$ overall because of q and W_o .
- Attention over cache:
 - We still compute dot products between each head's query and the $n + 1$ cached keys: einsum "bkh, bnk->bhn" uses $O(bhkn) = O(bnd)$ ops, same as MHA.

So the asymptotic compute cost (ignoring constant factors and the $1/h$ savings on k, v) is **unchanged**.

3. Computation

The projection and attention cost is clearly separated, which helps with organization.

Total MQA operations per token:

$$O(\underbrace{bd^2}_{q + o \text{ proj (dominant)}} + \underbrace{bnd}_{\text{attention}}) = O(bd^2 + bnd).$$

4. Answer

$$\boxed{\text{MQA compute complexity} = O(bd^2 + bnd)}$$

The complexity matches the official solution.

The constants improve slightly, but the Big-O form stays the same as standard MHA.

(g) Memory access complexity for MQA (one new token, cache length n)

1. Restate

For MHA they gave memory access complexity:

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

(from reading weights, current token embeddings, and cached keys/values across all heads).

We must now compute the analogous complexity for MQA, where keys and values are *shared*.

2. Idea

Break down memory reads/writes in MQA:

1. Weights:

- $W_q : [h, d, q], W_k : [d, k], W_v : [d, v], W_o : [h, v, d]$.
- Total is still $O(d^2)$ in terms of asymptotics.

2. Current token embeddings / intermediates:

- We read/write x, q, k, v, o, y : total $O(bd)$ just like MHA.

3. Cached keys & values:

- In MHA, caches had shape $[b, h, n, k]$ and $[b, h, n, v] \rightarrow O(bnd)$.
- In MQA, caches are shared: $[b, n, k]$ and $[b, n, v]$.

Each has only d/h in the last dimension, so total cached size is

$$O(bn(d/h)) = O\left(\frac{bnd}{h}\right).$$

So the only term that shrinks is the cache term, by a factor of $1/h$.

3. Combine terms

$$\text{memory complexity (MQA)} = O(d^2) + O(bd) + O\left(\frac{bnd}{h}\right) = O\left(d^2 + bd + \frac{bnd}{h}\right).$$

4. Answer

$$\boxed{\text{MQA memory access complexity} = O\left(d^2 + bd + \frac{bnd}{h}\right)}$$

Intuitively: we still have to read weights and current token, but the cached keys/values are h times smaller, which is where MQA gains its memory-access advantage.

This solution identifies the 3 components and correctly breaks down the complexity for each part.

The official solutions are $O(d^2 + bd + bnd)$ which is equivalent to this because $k \leq d/h$.

Final collected answers

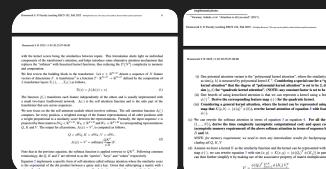
- (a) $A = [h, d, q]$
- (b) $B = [d, k]$
- (c) $C = \text{torch.einsum("bd,hdq->bhq", x, W_q)}$
- (d) $D = \text{torch.einsum("bd,dk->bk", x, W_k)}$
- (e) $E = \text{torch.einsum("bd,dv->bv", x, W_v)}$
- (f) MQA computation: $\boxed{O(bd^2 + bnd)}$
- (g) MQA memory access: $\boxed{O\left(d^2 + bd + \frac{bnd}{h}\right)}$.

Each choice matches the required tensor shapes and shows how MQA shares keys/values while keeping per-head queries.

□ ↗ ↘ ↕ ↖ ↙ ⋮

The LLM consistently reasoned about projection dimensions, einsum axes, and broadcasting across heads. Also, the compute and memory complexities were broken down by component.

In general, all student solutions matched the official solutions. The LLM solution included more detailed reasoning and explained why each answer is correct. It follows the general structure from the prompt, but also adapts it to better fit the needs of the question.



Question 6:

Again, the same prompt was used.

I'm working on this problem. For each subpart, please: 1. Restate the problem in your own words. 2. Explain the main idea / plan before doing any calculations. 3. Show the full step-by-step solution, with no big jumps. 4. Briefly summarize the final answer and why it's correct. Keep explanations clear and intuitive. Make sure to include thorough explanations. Remember to solve all subparts associated with the problem and give your final answer. I have attached screenshots of each problem.

Thought for 4m 27s >

Nice, this is a big one but very structured. I'll go part by part.

First, let me rewrite the **generic attention** equation (3), since everything builds on it:

$$V'_i = \frac{\sum_{j=1}^N \text{sim}(Q_i, K_j) V_j}{\sum_{j=1}^N \text{sim}(Q_i, K_j)}. \quad (3)$$

Here:

- Q_i is the i -th query vector,
- K_j is the j -th key,
- V_j is the corresponding value,
- $\text{sim}(\cdot, \cdot)$ is a generic similarity function.

(a) Conditions on `sim` so V'_i is always finite**1. Restate**

We need conditions on the similarity function $\text{sim}(q, k)$ so that, for every query Q_i , the denominator

$$\sum_{j=1}^N \text{sim}(Q_i, K_j)$$

never becomes zero, so that V'_i remains finite.

2. Idea

The fraction blows up (is undefined) if the denominator is zero. So we want conditions that guarantee:

- the denominator is **never 0**, and ideally
- all terms are **finite**, so we don't get infinities.

The LLM correctly focuses on the denominator.

A simple sufficient condition is:

"similarities are always non-negative, and for each query at least one similarity is strictly positive."

3. Steps

For a fixed query Q_i , define

$$Z_i = \sum_{j=1}^N \text{sim}(Q_i, K_j).$$

To ensure V'_i is finite, we need:

1. Each $\text{sim}(Q_i, K_j)$ is **finite**.

So sim should not return $\pm\infty$.

2. $Z_i \neq 0$ for all i . A simple way to ensure this is:

- $\text{sim}(q, k) \geq 0$ for all q, k (non-negative outputs), and
- for each i there exists some j such that $\text{sim}(Q_i, K_j) > 0$.

The official solutions phrase a more general condition first ($\sum \neq 0$), then specializes to positive kernels.

The steps are mathematically correct: non-negative plus at least one positive term $\Rightarrow \sum > 0$. The solution also required that sim never be int, which is reasonable.

Then every term in the sum is ≥ 0 , and at least one is > 0 , so

$$Z_i = \sum_j \text{sim}(Q_i, K_j) > 0.$$

These conditions are satisfied for standard softmax attention, where $\text{sim}(q, k) = \exp(q^\top k / \sqrt{D}) > 0$.

4. Final answer

A sufficient set of conditions is:

- $\text{sim}(q, k)$ is finite for all q, k ,
- $\text{sim}(q, k) \geq 0$ for all q, k ,
- for every query Q_i , there exists at least one key K_j such that $\text{sim}(Q_i, K_j) > 0$.

For the restricted setting of positive kernels, the final answer condition is equivalent. However, it is more restrictive than needed in general case because of the non negativity.

Then the denominator $\sum_j \text{sim}(Q_i, K_j)$ is strictly positive, so V'_i is always finite.

(b)(i) Polynomial / quadratic kernel attention: what is $\text{sim}(q, k)$?

1. Restate

We consider "polynomial kernel attention," where

$$\text{sim}(q, k) = K^c(q, k)$$

is given by a polynomial kernel of degree c . For the special case **quadratic kernel attention** with degree $c = 2$, we must derive $\text{sim}(q, k)$. Any constant factor is set to 1.

2. Idea

The standard polynomial kernel (with no extra constant) is

$$K^c(q, k) = (q^\top k)^c.$$

The solution missed the part in instruction that says constant factor is set to 1.

So for $c = 2$ we just square the dot product.

3. Steps

For degree $c = 2$,

$$K^2(q, k) = (q^\top k)^2.$$

So the similarity function is

$$\text{sim}(q, k) = (q^\top k)^2.$$

The derivation that $(q^\top k)^2$ is a valid similarity function was mathematically fine. However, it is not the kernel the problem asked for. The "+1" was dropped.

4. Final answer

For quadratic kernel attention,

$$\boxed{\text{sim}(q, k) = (q^\top k)^2.}$$

The final answer is right for the homogeneous Kernel instead of the polynomial Kernel. This final answer is therefore incorrect.

(b)(ii) Feature map $\phi(\cdot)$ for the quadratic kernel

1. Restate

We want a feature map $\phi(\cdot)$ such that the quadratic kernel satisfies

$$K(q, k) = (q^\top k)^2 = \phi(q)^\top \phi(k).$$

Find such a $\phi(\cdot)$.

2. Idea

For any vectors $q, k \in \mathbb{R}^D$,

$$(q^\top k)^2 = \left(\sum_{r=1}^D q_r k_r \right)^2 = \sum_{r=1}^D \sum_{s=1}^D q_r q_s k_r k_s.$$

We can recognize this as an inner product between the **Kronecker products**:

$$(q \otimes q)^\top (k \otimes k) = \sum_{r,s} q_r q_s k_r k_s = (q^\top k)^2.$$

The high-level idea of using a feature map built from quadratic monomials to represent a quadratic kernel. However, the wrong kernel is used here as well.

So a very natural feature map is $\phi(x) = x \otimes x$, i.e., all pairwise products of coordinates.

3. Steps

- Define

For the kernel $(q^\top k)^2$, this feature map is valid, but lacks the constant "1" component and $\sqrt{2}$ scaling.

$$\phi(x) = \text{vec}(xx^\top) \in \mathbb{R}^{D^2},$$

i.e. we take the $D \times D$ matrix xx^\top and flatten it into a vector.

- Then:

$$\phi(q)^\top \phi(k) = \text{vec}(qq^\top)^\top \text{vec}(kk^\top) = \text{trace}(qq^\top kk^\top) = (q^\top k)^2.$$

So this ϕ reproduces the quadratic kernel.

4. Final answer

One valid feature map is

$$\boxed{\phi(x) = \text{vec}(xx^\top) = x \otimes x.}$$

The LRU gave a valid Φ for $(q^\top k)^2$, not for $(q^\top k + 1)^2$, making this final answer incorrect.

Then $\phi(q)^\top \phi(k) = (q^\top k)^2$.

(Any equivalent map that lists all degree-2 monomials of the components of x is fine.)

(b)(iii) Rewrite kernel attention (equation 3) using a feature map ϕ

1. Restate

We consider a **general** kernel attention where the kernel can be written as

$$K(q, k) = \phi(q)^\top \phi(k).$$

We must rewrite the attention equation (3) using this feature map.

2. Idea

Just plug $\text{sim}(Q_i, K_j) = K(Q_i, K_j) = \phi(Q_i)^\top \phi(K_j)$ into (3).

The plan is correct and concise.

3. Steps

Start from (3):

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

The algebra is straightforward and matches the official answer.

Set

$$\text{sim}(Q_i, K_j) = K(Q_i, K_j) = \phi(Q_i)^\top \phi(K_j).$$

Then

$$V'_i = \frac{\sum_{j=1}^N (\phi(Q_i)^\top \phi(K_j)) V_j}{\sum_{j=1}^N \phi(Q_i)^\top \phi(K_j)}.$$

The official solution has $\Phi(Q_i)\Phi(K_j)^\top$, which is a row vector times a column vector while $\Phi(Q_i)^\top \Phi(K_j)$ is a column vector times a row vector. Both are equal to the dot product and are just vector notation differences.

That's the kernelized attention written explicitly with ϕ .

4. Final answer

Kernel attention with feature map ϕ is

$$V'_i = \frac{\sum_{j=1}^N (\phi(Q_i)^\top \phi(K_j)) V_j}{\sum_{j=1}^N \phi(Q_i)^\top \phi(K_j)}.$$

The final answer is correct.

(The later parts of the problem will algebraically rearrange this.)

(c) Time and space complexity of softmax attention (equation 4)

Equation (4) is the usual softmax attention form:

$$V'_i = \frac{\sum_{j=1}^N \exp\left(\frac{Q_i^\top K_j}{\sqrt{D}}\right) V_j}{\sum_{j=1}^N \exp\left(\frac{Q_i^\top K_j}{\sqrt{D}}\right)}. \quad (4)$$

We want the **asymptotic time complexity** and **space complexity** to compute all V'_i for $i = 1, \dots, N$, in terms of:

- sequence length N ,
- model dimension D for Q, K ,
- value dimension M for V .

We also must include storing Q, K, V for backprop.

Time complexity

1. Restate

Compute all outputs V'_i for $i = 1, \dots, N$ using softmax attention, and find the big-O computational cost.

2. Idea

Main stages:

1. Compute score matrix $S = \frac{QK^\top}{\sqrt{D}}$ ($N \times N$).
2. Apply softmax row-wise to get attention weights A ($N \times N$).
3. Multiply AV to get $V' \in \mathbb{R}^{N \times M}$.

The plan clearly separates into score computation, softmax, and weighted sum.

We focus on dominant terms.

3. Steps

1. Scores $S = QK^\top$:
 - Q is $N \times D$, K^\top is $D \times N$.
 - Matrix multiplication cost: $O(N^2D)$.
2. Softmax:
 - For each of N rows, we do exponentials + normalizations over N elements.
 - Cost: $O(N^2)$ (lower order than N^2D if D is reasonably large).
3. Weighted sum $V' = AV$:
 - A is $N \times N$, V is $N \times M$.
 - Matrix multiplication cost: $O(N^2M)$.

The steps use the matrix multiply viewpoint instead of explicit loops, but save big-O reasoning. However, the pseudocode given in the official solutions make the costs more intuitive and clear.

Total time:

$$T_{\text{softmax}} = O(N^2D + N^2M) = O(N^2(D + M)).$$

(If D and M are same order, this is the usual $O(N^2D)$ complexity.)

4. Final time answer

$$T_{\text{softmax}} = O(N^2(D + M)).$$

This matches the official solution.

Space (memory) complexity

1. Restate

We must store everything needed for backprop, including Q, K, V and intermediate matrices.

2. Idea

We need memory for:

- $Q: N \times D$,
- $K: N \times D$,
- $V: N \times M$,
- scores / attention matrix S or $A: N \times N$ (we must keep something of that size).

3. Steps

Sizes:

- $Q: O(ND)$,
- $K: O(ND)$,
- $V: O(NM)$,
- attention scores/weights: $O(N^2)$.

Ignoring smaller overheads, total:

$$\text{Mem}_{\text{softmax}} = O(ND + ND + NM + N^2) = O(N(D + M) + N^2).$$

4. Final space answer

$\text{Mem}_{\text{softmax}} = O(N(D + M) + N^2).$

This matches the official solution.

The N^2 term from the attention matrix is what hurts for long sequences.

(d) Complexity of kernel attention in equation (5) (quadratic, linearized kernel)

Equation (5) (after using the feature map and associativity) is:

$$V'_i = \frac{\phi(Q_i)^\top \left(\sum_{j=1}^N \phi(K_j) V_j^\top \right)}{\phi(Q_i)^\top \left(\sum_{j=1}^N \phi(K_j) \right)}. \quad (5)$$

Note:

- $\phi(Q_i), \phi(K_j) \in \mathbb{R}^M$ (feature dimension M),
- $V_j \in \mathbb{R}^M$ as before.

We are told:

- use a **linearized polynomial kernel** $\phi(x)$ of degree 2,
- assume $M \approx D$,
- find **computational cost** and **memory requirement** of this kernel attention.

1. Restate

We must:

1. Consider kernel attention computed as in (5).
2. Assume ϕ is the degree-2 polynomial feature map (dimension M), with $M \sim D$.
3. Derive big-O time and memory in terms of N, D, M .

2. Idea

Break computation into stages:

1. Compute $\phi(Q_i)$ and $\phi(K_j)$ for all tokens.
2. Compute the **global** quantities:
 - $S = \sum_{j=1}^N \phi(K_j) V_j^\top$ (an $M \times M$ matrix),
 - $u = \sum_{j=1}^N \phi(K_j)$ (an M -dim vector).
3. For each query i , compute numerator and denominator:

The solution assumes the feature dimension is equals M or D , instead of using a separate symbol C and recognizing that for the quadratic polynomial kernel, $C = O(D^2)$. This means the cost was underestimated.

- numerator: $\phi(Q_i)^\top S \rightarrow$ an M -dim row vector,
- denominator: $\phi(Q_i)^\top u \rightarrow$ a scalar.

4. Divide elementwise: $V'_i = \text{numerator}/\text{denominator}$.

We count ops and memory for these.

3. Time complexity

(i) Compute feature maps $\phi(Q)$ and $\phi(K)$

A degree-2 polynomial feature map essentially produces all degree-2 monomials; we'll treat its cost as $O(DM)$ per vector.

- For all queries: N vectors \rightarrow cost $O(NDM)$.
- For all keys: N vectors \rightarrow cost $O(NDM)$.

Total feature-map cost: $O(NDM)$ (up to constant factor 2).

(ii) Compute global sums S and u

- For each j :
 - $\phi(K_j)V_j^\top$ is an outer product of two M -dim vectors $\rightarrow O(M^2)$.
 - Sum over N keys: cost $O(NM^2)$.
- For $u = \sum_j \phi(K_j)$:
 - Summing N M -dim vectors: $O(NM)$, which is dominated by $O(NM^2)$.

Total here: $O(NM^2)$.

(iii) Compute each V'_i using (5)

For each i :

- Numerator: $\phi(Q_i)^\top S$:
 - multiply $1 \times M$ by $M \times M \rightarrow O(M^2)$.
- Denominator: $\phi(Q_i)^\top u$:
 - $1 \times M$ by $M \times 1 \rightarrow O(M)$.

For all N queries:

- Numerators: $O(NM^2)$,
- Denominators: $O(NM)$.

Total, dominated by $O(NM^2)$.

Combine all steps

Total time:

$$T_{\text{kernel}} = O(NDM) + O(NM^2) + O(NM^2) = O(NDM + NM^2).$$

Given the assumption $M \approx D$, we can say M and D are the same order, so:

$$T_{\text{kernel}} = O(ND^2).$$

(Compare this with $O(N^2D)$ for standard softmax attention when $D \sim M$ — this is linear in N instead of quadratic.)

4. Space complexity

We must store:

- Q and K : each $N \times D \rightarrow O(ND)$,
- $V: N \times M \rightarrow O(NM)$,
- $\phi(Q): N \times M \rightarrow O(NM)$,
- $\phi(K): N \times M \rightarrow O(NM)$,
- $S: M \times M \rightarrow O(M^2)$,
- $u: M \rightarrow O(M)$.

All computation is correct, but the dimension mistake carries throughout the solution.

Ignoring constants and lower-order terms:

$$\text{Mem}_{\text{kernel}} = O(ND + NM + M^2).$$

With $M \approx D$:

$$\text{Mem}_{\text{kernel}} = O(ND + D^2).$$

In many realistic cases $N \gg D$, so this behaves like $O(ND)$, i.e. **linear in sequence length** rather than quadratic.

5. Final answer for (d)

- Time complexity of kernel attention with a degree-2 linearized polynomial kernel and $M \approx D$:

$$T_{\text{kernel}} = O(ND^2).$$

- Memory requirement:

$$\text{Mem}_{\text{kernel}} = O(ND + D^2).$$

Both the time and space complexities are too optimistic by a factor of 0, so this final answer is incorrect.

I guided the LLM by mentioning the constant.

For bi and bii, the constant should be included. Please give me answers accordingly.

Thought for 14s >

Gotcha — let's redo (b.i) and (b.ii) *including the constant term* in the polynomial kernel. I'll keep the same 4-step structure.

(b.i) Quadratic kernel attention with constant term

1. Restate

We're using **polynomial kernel attention** where the similarity is a polynomial kernel K .

For **quadratic kernel attention** (degree 2), we now need to write the similarity

$$\text{sim}(q, k)$$

including the constant term of the polynomial kernel.

2. Main idea

The standard degree-2 polynomial kernel is

The correct kernel is now being used.

$$K(q, k) = (q^\top k + c)^2.$$

The problem statement says any *overall constant factor* can be treated as 1, but we **should not drop the +1 inside** — that's part of the kernel definition, not just a scaling.

So we'll take $c = 1$, no extra scaling.

3. Step-by-step

Define

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

If you expand it you get

$$(q^\top k + 1)^2 = (q^\top k)^2 + 2q^\top k + 1,$$

which is still a valid similarity function and can be expressed using a feature map later.

4. Final answer

For quadratic polynomial kernel attention **with constant term**:

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

This is correct and aligns with the official solution.

(b.ii) Feature map $\phi(\cdot)$ for the quadratic kernel with constant

We now want $\phi(\cdot)$ such that

$$K(q, k) = (q^\top k + 1)^2 = \phi(q)^\top \phi(k).$$

1. Restate

We must build an explicit feature map $\phi : \mathbb{R}^D \rightarrow \mathbb{R}^M$ whose inner product reproduces the **full** quadratic polynomial kernel $(q^\top k + 1)^2$, including constant and linear terms.

2. Main idea

- Expand $(q^\top k + 1)^2$ into constant, linear, and quadratic pieces.
- Build $\phi(x)$ whose coordinates are:
 - a constant 1,
 - all original coordinates x_i (linear terms),
 - all degree-2 monomials in x : x_i^2 and $x_i x_j$.
- Choose $\sqrt{2}$ scaling on the linear and cross terms so that the inner product matches all coefficients exactly.

3. Step-by-step

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

First expand the kernel:

$$(q^\top k + 1)^2 = (q^\top k)^2 + 2q^\top k + 1.$$

Write $q^\top k$ as a sum:

$$q^\top k = \sum_{d=1}^D q_d k_d.$$

The steps here use the right kernel and are very clear.

Then

$$(q^\top k)^2 = \left(\sum_{d=1}^D q_d k_d \right)^2 = \sum_{d=1}^D q_d^2 k_d^2 + 2 \sum_{1 \leq i < j \leq D} q_i q_j k_i k_j.$$

So altogether:

$$(q^\top k + 1)^2 = 1 + 2 \sum_{d=1}^D q_d k_d + \sum_{d=1}^D q_d^2 k_d^2 + 2 \sum_{1 \leq i < j \leq D} q_i q_j k_i k_j.$$

We want to express this as an inner product $\phi(q)^\top \phi(k)$.

Define $\phi(x)$ to contain:

1. A **constant** component: 1.
2. All **linear** components scaled by $\sqrt{2}$: $\sqrt{2} x_d$.
3. All **squared** components: x_d^2 .
4. All **cross** components scaled by $\sqrt{2}$: $\sqrt{2} x_i x_j$ for $i < j$.

Explicitly,

$$\phi(x) = \begin{bmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \vdots \\ \sqrt{2}x_D \\ x_1^2 \\ x_2^2 \\ \vdots \\ x_D^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2}x_1x_3 \\ \vdots \\ \sqrt{2}x_{D-1}x_D \end{bmatrix} \in \mathbb{R}^M, \quad M = 1 + D + D + \frac{D(D-1)}{2} = 1 + D + \frac{D(D+1)}{2} = O(D^2).$$

Now compute the inner product:

$$\phi(q)^\top \phi(k) = 1 \cdot 1 + \sum_{d=1}^D (\sqrt{2}q_d)(\sqrt{2}k_d) + \sum_{d=1}^D q_d^2 k_d^2 + \sum_{i < j} (\sqrt{2}q_i q_j)(\sqrt{2}k_i k_j).$$

That is

$$\phi(q)^\top \phi(k) = 1 + 2 \sum_{d=1}^D q_d k_d + \sum_{d=1}^D q_d^2 k_d^2 + 2 \sum_{i < j} q_i q_j k_i k_j = (q^\top k + 1)^2.$$

So this ϕ correctly reproduces the polynomial kernel with constant term.

4. Final answer

A valid feature map for the quadratic polynomial kernel with constant term is

$$\boxed{\phi(x) = (1, \sqrt{2}x_1, \dots, \sqrt{2}x_D, x_1^2, \dots, x_D^2, \sqrt{2}x_1x_2, \dots, \sqrt{2}x_{D-1}x_D)^\top}$$

with dimension

$$M = 1 + D + \frac{D(D+1)}{2} = O(D^2),$$

This final answer is correct.

and it satisfies

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

If you'd like, I can also quickly rewrite (b.iii) in terms of this new ϕ , but the algebraic form is identical — only the internal definition of ϕ changes.

◻ ⊕ ⊖ ↗ ↙ ⋯

I also corrected part d by explicitly saying quadratic kernel.

For d, the quadratic kernel gives us C = O(D^2). Please give me the updated answer accordingly.

Thought for 54s >

Nice catch about the quadratic feature map **30** — let's redo (d) assuming the **full quadratic kernel feature map** from (b.ii), whose dimension is

$$C = \dim \phi(x) = \Theta(D^2).$$

I'll go through the four steps again just for part (d).

(d) Complexity with quadratic kernel, $C = O(D^2)$ **1. Restate**

We use kernel attention in the form

$$V'_i = \frac{\phi(Q_i)^\top \left(\sum_{j=1}^N \phi(K_j) V_j^\top \right)}{\phi(Q_i)^\top \left(\sum_{j=1}^N \phi(K_j) \right)}, \quad (5)$$

where:

- $Q_i, K_j \in \mathbb{R}^D$ (query/key dim = D)
- $V_j \in \mathbb{R}^M$ (value dim = M)
- $\phi : \mathbb{R}^D \rightarrow \mathbb{R}^C$ is the **quadratic kernel feature map**, with

$$C = \Theta(D^2).$$

We want the **asymptotic time and memory** of computing all V'_1, \dots, V'_N under this assumption.

2. Main idea / plan

1. Express the computation of (5) in two phases:

- Precompute global sums over keys/values:

$$Z = \sum_{j=1}^N \phi(K_j) \in \mathbb{R}^C, \quad S = \sum_{j=1}^N \phi(K_j) V_j^\top \in \mathbb{R}^{C \times M}.$$

- For each query Q_i , compute $\phi(Q_i)$, then numerator and denominator using S and Z .

2. Count time in terms of N, C, M , then plug in $C = \Theta(D^2)$.

3. Count memory needed to store Q, K, V and the kernel-specific intermediates (Z, S , and optionally $\phi(K_j)$ for backprop).

3. Step-by-step derivation**Precomputation over keys/values**

For each $j = 1, \dots, N$:

1. Compute $\phi(K_j) \in \mathbb{R}^C$.
 - Cost: $O(C)$ per token (we're explicitly forming all C monomials).
2. Update the vector sum:

$$Z \leftarrow Z + \phi(K_j)$$

- Cost: $O(C)$.

3. Update the matrix sum:

$$S \leftarrow S + \phi(K_j) V_j^\top$$

- This is an outer product in $\mathbb{R}^{C \times M}$.
- Cost: $O(CM)$ per token.

The steps are shown clearly, but having pseudocode like the official solutions would be helpful.

So per key position j : $O(C + CM) = O(CM)$.

For all N keys:

$$T_{\text{keys}} = O(NCM).$$

(We're ignoring lower-order NC compared to NCM if M is not tiny.)

Per-query computation

For each query $i = 1, \dots, N$:

1. Compute $\phi(Q_i) \in \mathbb{R}^C$: $O(C)$.
2. Numerator:

$$\text{num}_i = \phi(Q_i)^\top S \in \mathbb{R}^{1 \times M}.$$

- Multiply $1 \times C$ by $C \times M: O(CM)$.

3. Denominator:

$$\text{den}_i = \phi(Q_i)^\top Z$$

- Dot product: $O(C)$.

4. Final output:

$$V'_i = \frac{\text{num}_i}{\text{den}_i},$$

- Elementwise divide: $O(M)$.

Per query, the dominant term is $O(CM)$.

For all N queries:

$$T_{\text{queries}} = O(NCM).$$

Total time

Adding both phases:

$$T_{\text{kernel}} = O(NCM) + O(NCM) = O(NCM).$$

Now substitute $C = \Theta(D^2)$:

$$T_{\text{kernel}} = O(ND^2M).$$

If you take the common assumption $M \approx D$, then

$$T_{\text{kernel}} = O(ND^3).$$

Memory complexity

We must store for backprop:

- Queries $Q: N \times D \rightarrow O(ND)$
- Keys $K: N \times D \rightarrow O(ND)$
- Values $V: N \times M \rightarrow O(NM)$
- Vector $Z: \text{size } C \rightarrow O(C)$
- Matrix $S: \text{size } C \times M \rightarrow O(CM)$
- Optionally, all $\phi(K_j): N \times C \rightarrow O(NC)$ (likely needed if you backprop through ϕ).

So total:

$$S_{\text{kernel}} = O(ND) + O(ND) + O(NM) + O(NC) + O(C) + O(CM).$$

Dominant terms are $O(NC)$ and $O(CM)$.

Substitute $C = \Theta(D^2)$:

$$S_{\text{kernel}} = O(ND + NM + ND^2 + D^2M).$$

This is concise
and clear.

If again $M \approx D$:

$$S_{\text{kernel}} = O(ND^2 + D^3).$$

4. Final summary

With the **full quadratic kernel feature map** (dimension $C = \Theta(D^2)$), kernel attention in the form of (5) has:

- **Time complexity:**

$$T_{\text{kernel}} = O(NCM) = O(ND^2M) \approx O(ND^3) \text{ if } M \approx D.$$

- **Memory complexity** (including Q, K, V , sums, and feature-map intermediates):

$$S_{\text{kernel}} = O(ND + NM + ND^2 + D^2M) \approx O(ND^2 + D^3) \text{ if } M \approx D.$$

So once you take the *true* quadratic mapping with $C = O(D^2)$, the kernelized attention no longer has the nice $M \approx D$ scaling used in the “linearized” case; instead, cost scales roughly like ND^3 in time and $ND^2 - D^3$ in memory.

The answer now
aligns with the
official solutions.