

[Homework 09 Question...](#)  
[PDF](#)

can you please solve problem 1 in this PDF?

Please restate the problem exactly as written, and then solve the problem. Solve all parts of that specific question unless I specify a, b, c, etc

Thought for 12s >

### Problem 1 — Justifying Scaled-Dot Product Attention

(quoted verbatim from the PDF) [Homework 09 Question](#)

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  $\mu$  and standard deviation  $\sigma$ , and the same is true for  $k$ .

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

(b) Considering a practical case where  $\mu = 0$  and  $\sigma = 1$ , define  $\text{Var}(q^T k)$  in terms of  $d$ .

(c) 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  $\mathbb{E}\left[\frac{q^T k}{s}\right] = 0$  and

$\text{Var}\left(\frac{q^T k}{s}\right) = \sigma = 1$ .

What should  $s$  be in terms of  $d$ ?

When I prompted the LLM with Question 1, it restated the setting correctly:  $q$  and  $k$  are iid Gaussian vectors in  $\mathbb{R}^d$  with mean  $\mu$  and variance  $\sigma^2$ . For part (a), the LLM derived  $\mathbb{E}[q^T k]$  by summing  $\mathbb{E}[q_i k_i]$ , concluding that each expectation equals  $\mu^2$ , giving  $\mathbb{E}[q^T k] = \|\mu\|^2$ . This matches the solution

. For part (b), with  $\mu=0$  and  $\sigma=1$ , the LLM computed  $\text{Var}(q^T k)$  by noting that the variance of the product of two independent  $N(0,1)$  variables is 1, giving  $\text{Var}(q^T k) = d$ . This is again identical to the official answer. For part (c), the LLM reasoned that scaling by  $s$  makes  $\text{Var}((q^T k)/s) = d/s^2$ , leading to  $s = \sqrt{d}$ . The official solution shows the same reasoning and the same value of  $s$ , so the LLM's work for the entire problem was fully correct and aligned.

### Solution

(a)  $\mathbb{E}[q^T k]$

Write the dot product as a sum of coordinate-wise products:

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

Because the vectors are drawn independently and every pair  $(q_i, k_i)$  is independent with

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

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

Hence

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

**(b)  $\text{Var}(q^\top k)$  for  $\mu = 0, \sigma = 1$**

With zero mean, the expectation from part (a) is 0; variance is therefore

$$\text{Var}(q^\top k) = \mathbb{E}[(q^\top k)^2].$$

Because different coordinates are independent and  $\mathbb{E}[q_i k_i] = 0$ , all cross-terms vanish:

$$\begin{aligned} \mathbb{E}[(q^\top k)^2] &= \sum_{i=1}^d \mathbb{E}[q_i^2 k_i^2] = \sum_{i=1}^d \mathbb{E}[q_i^2] \mathbb{E}[k_i^2] \\ &= \sum_{i=1}^d (1)(1) = d. \end{aligned}$$

Thus

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

**(c) Scaling factor  $s$  so that the variance equals 1**

We need

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

Solving for  $s$ :

$$s = d.$$

Therefore

$$s = d$$

This is exactly the  $\frac{1}{d}$  scaling used in the “scaled-dot-product attention” found in Transformers.



now solve 2

Thought for 20s >

## Problem 2 — Argmax Attention

(quoted verbatim from the PDF)

### 2. Argmax Attention

Recall from lecture that we can think about attention as being queryable softmax pooling. In this problem, we ask you to consider a hypothetical **argmax** version of attention where it returns exactly the value corresponding to the key that is most similar to the query, where similarity is measured using the traditional inner-product.

**(a)** Perform argmax attention with the following keys and values:

Keys:

$$\left\{ \begin{array}{cccc} 1 & 0 & 5 & 0 \\ 2 & , & 3 & , \\ 0 & 4 & 0 & 1 \end{array} \right\}$$

Corresponding Values:

$$\left\{ \begin{array}{cccc} 2 & 1 & 0 & 1 \\ 0 & , & 4 & , \\ 1 & 3 & 4 & -1 \end{array} \right\}$$

using the following query

$$q = \begin{matrix} 1 \\ 2 \end{matrix} .$$

What would be the output of the attention layer for this query?

**(b)** Note that instead of using softmax we used **argmax** to generate outputs from the attention layer. How does this design choice affect our ability to usefully train models involving attention? (Hint: think about how the gradients flow through the network in the backward pass. Can we learn to improve our queries or keys during the training process?) 2:file1<sup>†</sup>L29-L33    2:file1<sup>†</sup>L34-L73    2:file1<sup>†</sup>L77-L116  
2:file1<sup>†</sup>L120-L130    2:file0<sup>†</sup>L3-L6

## Solution

### (a) Compute the argmax-attention output

1. Dot-product similarities between the query  $q$  and each key  $k_i$ :

$$\begin{aligned}s_1 &= q^\top k_1 = (1)(1) + (1)(2) + (2)(0) = 3, \\ s_2 &= q^\top k_2 = (1)(0) + (1)(3) + (2)(4) = 11, \\ s_3 &= q^\top k_3 = (1)(5) + (1)(0) + (2)(0) = 5, \\ s_4 &= q^\top k_4 = (1)(0) + (1)(0) + (2)(1) = 2.\end{aligned}$$

2. Identify the maximum score.

$\max_i s_i = 11$  (achieved by key 2).

3. Return the value associated with that key.

$$\text{Output} = \begin{matrix} 1 \\ 4 \\ 3 \end{matrix}$$

### (b) Why argmax hurts end-to-end training

The **argmax** operator is *non-differentiable* and piece-wise constant. During back-propagation its gradient is zero almost everywhere, so errors cannot flow through it to update the query, key, or value projection parameters. In other words, the model cannot

learn better similarity representations using standard gradient-based optimization. Softmax, by contrast, is smooth and provides useful gradients that let the network adjust queries and keys during training. Therefore replacing softmax with argmax effectively blocks learning unless special tricks (e.g.\ straight-through estimators or reinforcement-learning objectives) are introduced.

When I asked the LLM to perform argmax attention, it computed all four inner products between the query [1,1,2] and the four keys and obtained 3, 11, 5, and 2, respectively—the same values as the sols. It then applied argmax correctly and selected the second key, returning the value vector [1,4,3]. This is the same output in the official solution. For part (b), the LLM stated that argmax attention does not allow gradients to flow back through most parameters because argmax is insensitive to small changes, making gradients zero for almost all keys and queries. The official solution explains the same reasoning: almost no parameters update because the selected index does not change under infinitesimal perturbations. So the LLM's reasoning and conclusions were fully consistent with the official answer.

Solve 3

Thought for 42s >

### Problem 3 — Ordinary Softmax Multi-Head Attention Implementation (quoted verbatim from the PDF)

#### 3. “Ordinary Softmax Multihead Attention Implementation

(a) Complete the incomplete lines in the implementation of multi-head attention (with a final linear layer on top) by filling in (on the following page) what (a), (b), (c), etc. should be in the code below.”

python

 Copy code

```
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)__)
    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

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)

def combine_heads(self, x):
    batch_size, _, seq_length, d_k = x.size()
    return x.transpose(1, 2).view(batch_size,
                                  seq_length, self.d_model)

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

```

*Fill in the missing blanks...*

(b) Suppose that you want to update the initialization `W_v` in this way:

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

This increases the length of the value vectors that we get. **What other minimal modifications** should you apply to the code to make the code run? (*Hint: remember that in transformers we typically use attention blocks with residual connections around them. What does that require?*)

## Solution

### (a) Filled-in blanks

| blank | correct code / value         | reasoning  |
|-------|------------------------------|--|
| (a)   | <code>d_model</code>         | keys have the same dimension as the model                            |
| (b)   | <code>d_model</code>         | the value projection takes $d_{\text{model}}$ -wide vectors as input |
| (c)   | <code>d_model</code> (given) | keeps the output in the residual path's dimension                    |

| blank | correct code / value   | reasoning   |
|-------|------------------------|---|
| (d)   | Q                      | attention scores are $QK^\top$  |
| (e)   | softmax                | convert scores to probabilities   |
| (f)   | V                      | probabilities weight the value vectors                                    |
| (g)   | self.num_heads (given) | one slice per head when reshaping   |
| (h)   | W_o                    | final linear layer projects concatenated heads back to $d_{\text{model}}$ |

With the blanks filled, the critical lines become:

python Copy code

```

self.W_k = nn.Linear(d_model, d_model)          # (a)
self.W_v = nn.Linear(d_model, d_model)          # (b)
...
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)
...
return x.view(batch_size, seq_length,
             self.num_heads, self.d_k).transpose(1, 2)  # (g)
...
output = self.W_o(self.combine_heads(attn_output))   # (h)

```

## (b) Making the doubled-value variant work

Changing

python Copy code

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

means each head now emits **twice as many value features**. Two shape-dependent spots must be updated so the tensor sizes still line up with the residual connection:

1. **Output projection:** it must now accept a vector of length  $2d_{\text{model}}$  and map it back to  $d_{\text{model}}$  so that the block's output can still be added to the residual input.

python

Copy code

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

2. **Head recombination:** after splitting and transposing, each head has  $2d_k$  features. Replace the hard-coded reshape in `combine_heads` (or introduce `self.d_v = 2 * d_model // num_heads`) so the concatenation produces  $2d_{\text{model}}$  features:

python

Copy code

```
def combine_heads(self, x):
    batch_size, _, seq_length, d_v = x.size()    # d_v = 2 * self.d_
    return x.transpose(1, 2).reshape(batch_size,
                                    seq_length, 2 * self.d_model)
```

These two minimal adjustments preserve dimensional consistency, allowing the forward pass (and the surrounding residual connection) to run without further changes.

For the multi-head attention code in PyTorch, the LLM filled in all blanks: `W_k` takes input `d_model`, `W_v` takes input `d_model`, `W_o` outputs `d_model`, the attention uses `Q` in the scores, softmax in the probability conversion, `V` in the weighted sum, `self.num_heads` in the reshape, and `W_o` in the final projection. Every one of these matches the official sols

. In part (b), the LLM originally proposed several dimension adjustments when `W_v` is expanded to output  $2d_{\text{model}}$ , but the official answer clarifies that the minimal change required is simply updating `W_o` to take  $2d_{\text{model}}$  as input. The LLM's solution wasn't incorrect, but it wasn't minimal. Once pointed out, though, it easily recognized the simpler adjustment.

solve 4

Thought for 51s >

## Problem 4 — Transformer Decoding Optimization

(quoted verbatim from the PDF)

### 4. Transformer Decoding Optimization

Consider a multi-head attention (MHA) layer in a transformer decoder performing **incremental decoding**.

The attention layer operates on a batch of  $b$  sequences, each with dimension  $d$ .

For each of the  $h$  heads the query, key and value projection sizes are

$$q = k = v = \frac{d}{h}.$$

The following pseudocode shows the one-step MHA used during decoding:

python

Copy code

```
def multihead_attention_incremental(x, prev_K, prev_V, W_q, W_k, W_v):
    """Multi-head Self-Attention (one step)."""
    q = torch.einsum("bd,hdq->bhq", x, W_q) # [b,h,q]
    k = torch.einsum("bd,hdk->bhk", x, W_k) # [b,h,k]
    v = torch.einsum("bd,hdv->bhv", x, W_v) # [b,h,v]
    new_K = torch.cat([prev_K, k.unsqueeze(2)], dim=2) # [b,h,n+1,k]
    new_V = torch.cat([prev_V, v.unsqueeze(2)], dim=2) # [b,h,n+1,v]
    logits = torch.einsum("bhq,bhnk->bhn", q, new_K) # [b,h,n+1]
    weights = torch.nn.functional.softmax(logits, dim=-1)
    o = torch.einsum("bhn,bhnv->bhv", weights, new_V) # [b,h,v]
    y = torch.einsum("bhv,hvd->bd", o, W_o) # [b,d]
    return y
```

*Multi-Query Attention (MQA)* uses a **single shared** key and value stream for all heads:

python

Copy code

```
def multiquery_attention_incremental(x, prev_K, prev_V, W_q, W_k, W_v):
    """Multi-query Self-Attention (one step)"""
    # shapes to fill in ↓↓↓
    W_q: ____A____           W_k: ____B____
    # projections ↓↓↓
    q = ____C____             k = ____D____             v = ____E____
    new_K = torch.cat([prev_K, k.unsqueeze(1)], dim=1) # [b,n+1,k]
    new_V = torch.cat([prev_V, v.unsqueeze(1)], dim=1) # [b,n+1,v]
    logits = torch.einsum("bkh,bnk->bhn", q, new_K)
    weights = torch.nn.functional.softmax(logits, dim=-1)
    o = torch.einsum("bhn,bh nv->bhv", weights, new_V)
    y = torch.einsum("bhv,hvd->bd", o, W_o)
    return y
```

First, fill in the blanks above by picking one option for each item.

(a) What is A?  [h, d, q]  [d, q]  [h, q]

(b) What is B?  [h, d, k]  [d, k]  [h, k]

- (c) What is C?  `torch.einsum("bd,dq->bhq", x,W_q)`  
 `torch.einsum("bd,hdq->bhq", x,W_q)`  
 `torch.einsum("bhd,dq->bhq", x,W_q)`

- (d) What is D?  `torch.einsum("bd,dk->bk", x,W_k)`  
 `torch.einsum("bd,hdk->bhk", x,W_k)`  
 `torch.einsum("bd,dk->bhk", x,W_k)`

- (e) What is E?  `torch.einsum("bd,hdv->bhv", x,W_v)`  
 `torch.einsum("bd,dv->bhv", x,W_v)`  
 `torch.einsum("bd,dv->bv", x,W_v)`

(f) For MHA, generating one token with a cache of  $n$  tokens costs  $O(bd^2 + bnd)$ .

What is the computational complexity for MQA?

- (g) What is the memory-access complexity for MQA?

## Solution

| item | correct answer                                      | explanation  |
|------|---|--|
| A    | <code>[h, d, q]</code>                              | we still learn a distinct query projection per head, so the weight tensor carries an $h$ dimension.        |
| B    | <code>[d, k]</code>                                 | keys are shared across heads, so only one projection matrix of size $d \times k$ is required.              |
| C    | <code>torch.einsum("bd,hdq-&gt;bhq", x, W_q)</code> | multiplies every token $(b,d)$ with the per-head weights $(h,d,q)$ , yielding queries of shape $(b,h,q)$ . |
| D    | <code>torch.einsum("bd,dk-&gt;bk", x, W_k)</code>   | produces a single key vector per token of shape $(b,k)$ , matching B.                                      |
| E    | <code>torch.einsum("bd,dv-&gt;bv", x, W_v)</code>   | likewise yields a single value vector per token $(b,v)$ .  |

### (f) Computational complexity for MQA

#### *Projections*

- Queries:  $O(bd^2)$  (same as MHA).
- Keys + Values: each  $O(bd^2/h)$  because they are computed once, not per head.
- Output projection:  $O(bd^2)$ .

Dominant term:  $O(bd^2)$ .

#### *Attention*

The dot-product and weighting still require  $O(bhnk) = O(bnd)$ .

Hence

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

(the same big-O as MHA, though constant factors are lower for keys/values).

### (g) Memory-access complexity for MQA

- Weight reads:  $O(d^2)$  (unchanged).
- Current-token tensors (x, q, k, v, o, y):  $O(bd)$ .
- Cached keys/values: now **shared** across heads  $\rightarrow O(bnd/h)$ .

Therefore

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

Compared with MHA's  $O(d^2 + bd + bnd)$ , the cache-related term is reduced by the factor  $1/h$ , giving MQA its memory-efficiency advantage while leaving arithmetic cost nearly unchanged.



for g) just to confirm, the solution is  $O(d^{22} + bd + bnk)$   
is that the same as your answer with them just making  
 $k=b/h$ ?

Thought for 5s >

Yes—we're saying the same thing.

- In my notation the cache term was  $O\left(\frac{bnd}{h}\right)$ .
- If you denote the per-head key (and value) width by  $k \equiv d/h$ , that term is  $O(bnk)$ .

So your expression

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

is exactly equivalent (ignoring your extra "2" on  $d^2$ , which looks like a typo).

For the decoding-optimization question, the LLM read the pseudocode and selected dimensions for A, B, C, D, and E. It chose A = [h, d, q] and C = torch.einsum("bd,hdq->bhq"), which match the official sols

. It also identified B = [d, k], D = torch.einsum("bd,dk->bk"), and E = torch.einsum("bd,dv->bv", which again align exactly with the official answers. On the computational and memory complexity questions, the LLM independently derived forms equivalent to those in the official solution— $O(b d^2 + b n d)$  for compute and  $O(d^2 + b d + b n k)$  for memory. Even though it used slightly different notation at first, after substituting  $k = d/h$ , both matched the official expressions perfectly.

now do question 6 (skip 5)

Thought for 34s >

## Problem 6 — Kernelized Linear Attention (Part I)

(quoted verbatim from the PDF)

### 6. "Kernelized Linear Attention (Part I)

The softmax attention is widely adopted in transformers, however the  $O(N^2)$  ( $N$  stands for the sequence length) complexity in memory and computation often makes it less desirable for processing long document like a book or a passage, where the  $N$  could be beyond thousands. There is a large body of the research studying how to resolve this."

Under this context, this question presents a formulation of attention via the lens of the kernel .

We first review the building block in the transformer ... (equations for soft-max attention are introduced) ...

$$Q = xW_Q, \quad K = xW_K, \quad V = xW_V, \quad A(x) = V' = \text{softmax}\left(\frac{QK^\top}{D}\right)(2)$$

A more general attention can be written as

$$V'_i = \frac{\sum_{j=1}^N \text{sim}(Q_{i\cdot}, K_{j\cdot}) V_{j\cdot}}{\sum_{j=1}^N \text{sim}(Q_{i\cdot}, K_{j\cdot})}. \quad (3)$$

(a) Identify the conditions that need to be met by the `sim` function to ensure that the denominator in (3) never reaches zero .

(b) The definition in (3) is generic.

(i) For a **quadratic kernel attention** (degree 2 polynomial kernel), derive  $\text{sim}(q, k)$

(ii) Derive the corresponding feature map  $\phi(\cdot)$ .

(iii) Rewrite (3) using the feature map  $\phi$  .

(c) For the soft-max attention in (4), derive the **time complexity and space complexity** (memory needed for forward + back-prop) in terms of  $N, D, M$  .

(d) Using the linearized polynomial kernel of degree 2 with  $M \approx D$ , rewrite attention as in equation (5) and derive its computational cost and memory requirement .

## Solution

For part (a) on kernelized attention, the LLM stated that the denominator must be strictly positive, and one sufficient condition is that  $\text{sim}(q, k)$  is always non-negative and at least one term is positive. The official solution on page 10 says that the denominator must never be zero and that the similarity function must keep the same sign—particularly considering kernels that produce positive values only. These two statements are aligned: the LLM's sufficient condition is simply a stricter version of the official's requirement. So part (a) was correct.

### (a) Conditions on sim for a finite denominator

For every query  $q = Q_i$ . the denominator

$$\sum_{j=1}^N \text{sim}(q, K_{j.})$$

must be **strictly positive**.

A sufficient set of conditions is

1. **Non-negativity:**  $\text{sim}(q, k) \geq 0$  for all  $q, k$ .
2. **Positivity for at least one key:** for every  $q$  there exists some  $k$  such that  $\text{sim}(q, k) > 0$ .

These ensure the sum is never zero, so  $V'_i$  remains finite.

### (b) Quadratic-kernel attention

#### (i) Similarity function

A degree-2 homogeneous polynomial kernel is

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

**(ii) Feature map**

Because  $(q^T k)^2 = \sum_a \sum_b q_a q_b k_a k_b$ ,

define

$$\phi(q) = \{ q_a q_b \mid 1 \leq a \leq b \leq D \} \in \mathbb{R}^{D(D+1)/2},$$

(using a  $\sqrt{2}$  scale on off-diagonal components if one prefers an orthonormal basis).

Then

$$\text{sim}(q, k) = \phi(q)^T \phi(k).$$

**(iii) Kernel-attention form**

Plugging  $K(q, k) = \phi(q)^T \phi(k)$  into (3):

$$V'_i = \frac{\phi(Q_{i\cdot})^T \sum_{j=1}^N \phi(K_{j\cdot}) V_{j\cdot}}{\phi(Q_{i\cdot})^T \sum_{j=1}^N \phi(K_{j\cdot})}.$$

**(c) Complexity of soft-max attention**

*Forward time*

- Computing  $QK^T$ :  $O(N^2 D)$
- Soft-max over each row:  $O(N^2)$
- Multiplying by  $V$ :  $O(N^2 M)$

Total:  $O(N^2(D + M))$ .

*Memory (activations kept for back-prop)*

- $Q, K, V$ :  $O(N(D + D + M))$
- Attention logits / probs:  $O(N^2)$

Dominant term:  $O(N^2 + N(D + M))$ .

**(d) Quadratic-kernel (degree 2) with linearization**

For the quadratic kernel  $P := \dim \phi = O(D^2)$ .

**Pre-computations (once per sequence)**

$$S = \sum_{j=1}^N \phi(K_j)^\top V_j \text{ (shape } P \times M), \quad z = \sum_{j=1}^N \phi(K_j) \text{ (shape } P).$$

Cost:  $O(NPM) = O(ND^2M)$ .

**Per-query work (for all  $N$  positions)**

Each  $i$  needs  $\phi(Q_i)^\top S$  and  $\phi(Q_i)^\top z$ :  
 $O(PM + P)$  per token  $\rightarrow O(ND^2M)$ .

**Total time:**

$$O(ND^2M) \quad (\text{with } M \approx D, \text{ this is } O(ND^3)).$$

**Memory:**

- Store  $S: O(PM) = O(D^3)$ .
- Store  $z: O(P) = O(D^2)$ .
- Store  $Q, K, V: O(N(D + M))$ .

Dominant term:  $O(D^3 + ND)$ .

Compared with soft-max, the quadratic kernel replaces the  $N^2$  dependence with linear  $N$  but introduces a  $D^3$  factor, which is advantageous when  $N \gg D^2$ .



You previously solved a homework problem involving kernelized attention, but I need you to reconsider two parts of your answer.

Do not give me new solutions immediately. Instead, walk through your reasoning and correct any issues you find.

Please do the following:

Revisit part (b)(i):

The problem statement refers to polynomial kernel

attention and then a quadratic kernel attention specifically.

Make sure you are using the standard degree-2 polynomial kernel used in machine learning textbooks—not just the homogeneous kernel.

Reflect on whether you included the correct constant/bias term that appears in the usual polynomial kernel definition.

Do your earlier assumptions match the standard definition?

Revisit part (b)(ii):

Once you settle the correct kernel form from step (1), derive the corresponding feature map.

Ensure the dimensionality and structure match the known expansion of a degree-2 polynomial kernel.

Carefully check whether your earlier feature map would expand to the kernel you are now using.

After you reason through both steps, state what parts of your earlier answer would need modification

Thought for 11s >

Re-examining my earlier answer for part (b)

**(1) Polynomial-kernel definition I should have used  
Standard reference (textbook) form**

For a degree-2 polynomial kernel the standard form is

$$K(q, k) = (q^T k + c)^2,$$

where the bias  $c$  is usually set to 1. Thus

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

Expanding gives

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

Where my previous answer differed

Earlier I wrote the *homogeneous* version  $(q^T k)^2$  (i.e. I implicitly set  $c = 0$ ).

That omits the constant and linear-interaction terms and is **not** the canonical polynomial kernel.

So part (b)(i) needs to be corrected to include the "+1" bias term.

## (2) Consequences for the feature map

A feature map  $\phi(\cdot)$  must satisfy

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

With  $K(q, k) = (q^T k + 1)^2$  we need  $\phi$  to produce all three pieces:

1. Constant term: 1
2. Linear terms:  $2 q^T k$
3. Quadratic terms:  $(q^T k)^2$

A common explicit map is

$$\phi(q) = [1; 2q_1, \dots, 2q_D; q_1^2, 2q_1q_2, \dots, 2q_{D-1}q_D, q_D^2]^T.$$

- The **first component "1"** reproduces the constant +1.
- The  $2q_i$  components ensure  $\phi(q)^T \phi(k)$  yields  $2q^T k$ .
- The **quadratic block** (diagonals un-scaled, cross-terms scaled by 2) reproduces  $(q^T k)^2$ .

This map has dimensionality

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

whereas my earlier answer contained only the  $\frac{D(D+1)}{2}$  quadratic part.

Therefore part (b)(ii) also needs revision: the constant and linear components (with the 2 scaling) must be added, and the stated dimensionality updated accordingly.

## Summary of required modifications

| Section | What must change   | Why  |
|---------|--|--|
| (b)(i)  | Replace $\text{sim}(q, k) = (q^T k)^2$ with $(q^T k + 1)^2$ .  | The standard degree-2 polynomial kernel includes the bias term $+1$ .  |
| (b)(ii) | Expand the feature map to include:<br><ul style="list-style-type: none"> <li>• a constant "1" component,</li> <li>• <math>2 q_i</math> linear components,</li> <li>• quadratic components <math>q_i q_j</math> (off-diagonals scaled by <math>\frac{1}{2}</math>)</li> </ul> | These extra components are required so that $\phi(q)^T \phi(k)$ reproduces the full $(q^T k + 1)^2$ expansion. |

Its main mistake was in part (b). It had used the homogeneous quadratic kernel  $K(q, k) = (q^T k)^2$  instead of the standard polynomial kernel  $K(q, k) = (q^T k + 1)^2$ . Because of that, its feature map only included the quadratic terms like  $q_i q_j$  and completely missed the constant term and the linear terms that come from expanding  $(q^T k + 1)^2 = (q^T k)^2 + 2(q^T k) + 1$ . To fix this without giving away the answer, I wrote a prompt designed to get the LLM to question whether it had used the correct polynomial kernel definition and to check whether its feature map truly matched the kernel it claimed.

After I fed that corrective prompt to the LLM, it re-examined its reasoning and identified the exact issues on its own. It recognized that the kernel should have been  $(q^T k + 1)^2$  and therefore the feature map needed to include the constant 1, the linear components  $2 q_i$ , and the quadratic components  $q_i^2$  and  $2 q_i q_j$ . Once it reconstructed the full feature map and stated the correct dimensionality  $(D+1)(D+2)/2$ , I checked that everything matched the official solution. Throughout the process, I relied on prompting strategies that pushed the model toward self-correction—having it revisit assumptions, compare definitions, and inspect whether its earlier expressions could actually expand into the form required—without ever directly supplying the missing terms.