

Hi! Could you please solve this question? "1. Justifying Scaled-Dot Product Attention

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

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

Part (a): Compute $E[\mathbf{q}^T \mathbf{k}]$

We are given that \mathbf{q} and \mathbf{k} are independent random vectors in \mathbb{R}^d , with

$\mathbf{q}, \mathbf{k} \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2 \mathbf{I})$ i.i.d.,

where $\boldsymbol{\mu} \in \mathbb{R}^d$ is the mean vector and $\sigma > 0$.

We need to find the expected value of the (unnormalized) dot product $\mathbf{q}^T \mathbf{k}$.

$$E[\mathbf{q}^T \mathbf{k}] = E\left[\sum_{i=1}^d q_i k_i\right] = \sum_{i=1}^d E[q_i k_i]$$

(by linearity of expectation).

Since \mathbf{q} and \mathbf{k} are independent and identically distributed, for each component i :

- $q_i \sim \mathcal{N}(\mu_i, \sigma^2)$
- $k_i \sim \mathcal{N}(\mu_i, \sigma^2)$
- q_i and k_i are independent (because the vectors are independent and the coordinates are independent within each vector).

Thus,

$$E[q_i k_i] = E[q_i] \cdot E[k_i] = \mu_i \cdot \mu_i = \mu_i^2$$

(using independence of q_i and k_i).

Summing over all dimensions:

$$E[q^T k] = \sum_{i=1}^d \mu_i^2 = \|\mu\|_2^2 = \mu^T \mu$$

So the final answer for part (a) is:

$$E[q^T k] = \mu^T \mu$$

(or equivalently $\|\mu\|^2$, the squared ℓ_2 -norm of the mean vector μ).

This result is exact (no approximation) and holds for any dimension d .

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

Part (b): Compute $\text{Var}(q^T k)$ when $\mu = 0$ and $\sigma = 1$

Now we have $\mathbf{q}, \mathbf{k} \sim \mathbf{N}(\mathbf{0}, \mathbf{I})$ i.i.d. (standard normal vectors in \mathbb{R}^d).

We need $\text{Var}(\mathbf{q}^T \mathbf{k})$.

Recall that

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

Since q and k are independent, and all coordinates are independent, the random variables $q_i k_i$ (for $i=1, \dots, d$) are **independent** of each other.

Moreover, since q and k are independent, the variance of the sum is the sum of variances:

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

For each i :

$q_i \sim N(0, 1)$, $k_i \sim N(0, 1)$, independent.

The product of two independent standard normal random variables has the following properties:

$$E[q_i k_i] = E[q_i] E[k_i] = 0 \cdot 0 = 0$$

$$\text{Var}(q_i k_i) = E[(q_i k_i)^2] - (E[q_i k_i])^2 = E[q_i^2] E[k_i^2] = (1)(1) = 1$$

(since $\text{Var}(q_i) = E[q_i^2] - (E[q_i])^2 = E[q_i^2] = 1$).

Therefore,

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

for every i .

Summing over all d dimensions:

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

Final answer for part (b):

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

So when queries and keys are standard normal vectors (zero mean, unit variance), the variance of the raw dot product is exactly **d** . This is why scaling by $1/\sqrt{d}$ is crucial in scaled dot-product attention — it makes the variance equal to 1, independent of dimension.

(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 $E[q^T k]$ to be 0, and $\text{Var}(q^T k)$ to be $\sigma^2 = 1$. What should s be in terms of d ?

Part (c)

We are still in the setting of part (b):

$\mathbf{q}, \mathbf{k} \sim \mathbf{N}(\mathbf{0}, \mathbf{I})$ i.i.d. in \mathbb{R}^d (i.e., $\mu = 0, \sigma = 1$).

From part (b) we already know:

- $E[\mathbf{q}^T \mathbf{k}] = 0$
- $\text{Var}(\mathbf{q}^T \mathbf{k}) = d$

We now compute the scaled dot product with a scalar factor s :

$q^T k \cdot s$ (sometimes written as $\frac{q^T k}{s}$ in papers, but here the problem defines s as the scaling factor)

Its expectation and variance are:

$$E[s \cdot (q^T k)] = s \cdot E[q^T k] = s \cdot 0 = 0 \quad (\text{automatically satisfied for any } s)$$

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

The problem requires this variance to be exactly 1:

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

(taking the positive root, as scaling factors are conventionally positive).

Answer for part (c):

$$s = \frac{1}{\sqrt{d}}$$

This is precisely the scaling factor used in the original Transformer paper (“Attention Is All You Need”), where the authors divide the dot product by $\sqrt{d_k}$ (the dimension of

queries/keys) to keep the variance of the attention logits equal to 1, independent of the model dimension d .

Great job solving problem 1, we're now going to move on to question 2.
Solve part a



Solution to 2(a): Argmax Attention

We are performing **hard argmax attention** (also called winner-takes-all attention):

The attention mechanism returns **exactly** the value vector that corresponds to the key with the **highest** dot-product similarity to the query.

In case of ties, the problem's hint shows that we simply pick one of them (the example shows the first one in case of a tie).

Given:

Query:

$$\mathbf{q} = [1, 1, 2]^T$$

Keys (column vectors, 4 keys in total):

$$\mathbf{k}_1 = [1, 2, 0]^T$$

$$\mathbf{k}_2 = [0, 3, 4]^T$$

$$\mathbf{k}_3 = [5, 0, 0]^T$$

$$\mathbf{k}_4 = [0, 0, 1]^T$$

Corresponding values (also column vectors):

$$\mathbf{v}_1 = [2, 0, 1]^T$$

$$\mathbf{v}_2 = [1, 4, 3]^T$$

$$\mathbf{v}_3 = [0, -1, 4]^T$$

$$\mathbf{v}_4 = [1, 0, -1]^T$$

Step 1: Compute dot products $\mathbf{q}^T \mathbf{k}_i$ for $i = 1, 2, 3, 4$

- $\mathbf{q} \cdot \mathbf{k}_1 = (1)(1) + (1)(2) + (2)(0) = 1 + 2 + 0 = \mathbf{3}$

- $q \cdot k_2 = (1)(0) + (1)(3) + (2)(4) = 0 + 3 + 8 = \mathbf{11}$
- $q \cdot k_3 = (1)(5) + (1)(0) + (2)(0) = 5 + 0 + 0 = \mathbf{5}$
- $q \cdot k_4 = (1)(0) + (1)(0) + (2)(1) = 0 + 0 + 2 = \mathbf{2}$

Step 2: Find the maximum similarity

The scores are: 3, 11, 5, 2

→ The maximum is **11**, achieved uniquely by key #2 (k_2)

Step 3: Return the corresponding value

Since key #2 wins, the output of argmax attention is exactly $\mathbf{v_2}$

Final Answer for 2(a):

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

(or written row-wise as $[1, 4, 3]^T$)

So the output of the attention layer for this query is $\begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}$.

Now solve pt b



Solution to 2(b)

Using **argmax attention** instead of **softmax attention** makes the attention mechanism **non-differentiable** (or at best **sub-differentiable** in a very limited sense).

Here's why this severely hurts our ability to train models with gradient-based methods (e.g., SGD, Adam):

1. The argmax operation is piecewise constant

- Small changes in the query or key vectors usually do **not** change which key has the highest dot product.
- When the ranking doesn't change, the output of the attention layer is constant → the **gradient w.r.t. queries and keys is exactly zero almost everywhere**.

2. Gradients are zero almost everywhere

- During backpropagation, the loss cannot send useful gradient signals to the query projection (W^Q) or key projection (W^K) layers most of the time.
- The only places where the gradient is non-zero are on the **measure-zero set** where two keys have exactly equal dot products with the query (the decision boundary).
- Even there, the gradient is not well-defined in the usual sense (argmax is not differentiable at ties).

3. Consequence: queries and keys cannot learn effectively

- The model gets almost no learning signal telling it how to adjust Q and K to produce better attention patterns.
- Training essentially stalls: the attention pattern becomes fixed early on (often randomly), and the model cannot refine which keys should be attended to.

4. Comparison to softmax

- Softmax is smooth and differentiable everywhere.
- Even small improvements in a key's similarity score are rewarded with slightly higher attention weight → gradients flow continuously and inform the model how to improve Q and K.

Bottom line:

Replacing softmax with hard argmax breaks gradient flow through the attention mechanism. As a result, the model **cannot meaningfully learn** good query or key representations during training. This is why real Transformer models universally use

softmax (or differentiable approximations) rather than hard argmax.

Great job, now let's solve q3 parts a-h.



Here are the correct fillings for all the blanks in Question 3 (ordinary multi-head attention with final linear layer, PyTorch-style):

| Blank | Correct Answer | Explanation |
|-------|---|---|
| (a) | <code>d_model</code> | W_k projects from full model dimension to the smaller per-head dimension \rightarrow input size is <code>d_model</code> |
| (b) | <code>self.d_k</code> | W_v also projects to the per-head dimension <code>d_k</code> (same as keys) |
| (c) | <code>d_model</code> | W_o (output projection) takes the concatenated heads (size $\text{num_heads} \times d_k = d_{\text{model}}$) back to <code>d_model</code> |
| (d) | <code>Q</code> | We compute $Q^T K$, so we need <code>Q @ K.transpose(-2, -1)</code> \rightarrow first argument is <code>Q</code> |
| (e) | <code>softmax</code> | To turn raw attention scores (logits) into probabilities we apply <code>torch.softmax</code> |
| (f) | <code>V</code> | After softmax we do <code>attention_weights @ V</code> \rightarrow second argument is <code>V</code> |
| (g) | <code>self.num_heads</code> | In <code>split_heads</code> , we reshape the last dimension into $(\text{num_heads}, d_k) \rightarrow$ we view as <code>..., self.num_heads, self.d_k</code> |
| (h) | <code>W_o</code> or <code>self.W_o</code> | After concatenating all heads we still need to apply the final output linear layer W_o |

Final filled code (for reference):

Python

```

self.W_q = nn.Linear(d_model, d_model)           # (a) stays d_model
self.W_k = nn.Linear(d_model, self.d_k)          # (a)
self.W_v = nn.Linear(self.d_k, d_model)          # wait no → actually sta
# correct standard implementation:
self.W_k = nn.Linear(d_model, d_model)           # usually all are d_model
self.W_v = nn.Linear(d_model, d_model)           # but many clean impls c
self.W_o = nn.Linear(d_model, d_model)           # (c)

# inside scaled_dot_product_attention
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)

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

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

```

Summary of answers to write in the boxes:

| Blank | Answer | |
|-------|--|--|
| (a) | <code>d_model</code> | I notice over here, grok messes up the (b) part. It's interesting to see that it seems like in it's answer it seems to backtrace and be confused about it but ends up choosing the wrong value. This may be because I pasted in images and asked it to solve many parts at once, causing it to hallucinate on part b. When I ask it independently about b in the next part, it gives the correct answer. |
| (b) | <code>self.d_k</code> | |
| (c) | <code>d_model</code> | |
| (d) | <code>Q</code> | |
| (e) | <code>softmax</code> | |
| (f) | <code>V</code> | |
| (g) | <code>self.num_heads</code> | |
| (h) | <code>self.W_o</code> (or <code>W_o</code>) | |

These are the standard, correct fillings expected in virtually every PyTorch-style Transformer implementation (including the original “Attention Is All You Need” code pattern).

Are you absolutely sure about b? Remember that we split heads, what you added doesn't make sense.

🤖 Thought for 2m 1s

No, you're right—I made a mistake on (b) in my previous table. Upon double-checking standard PyTorch implementations of MultiHeadAttention (e.g., from sources like the official PyTorch docs, common tutorials, and from-scratch code examples), the linear projections for Q, K, and V are consistently `nn.Linear(d_model, d_model)` to allow for proper head splitting afterward.

The key point (as you noted) is that we project the inputs to the full `d_model` dimension first, then use `split_heads` to reshape into `(batch_size, num_heads, seq_length, d_k)`, where `d_k = d_model // num_heads`. Projecting directly to `d_k` (as I incorrectly suggested for (b)) wouldn't make sense for multi-head splitting—it

would result in mismatched dimensions.

Assuming (b) corresponds to the output dimension in one of the projection layers like

`self.W_K = nn.Linear(d_model, ___)` or `self.W_V = nn.Linear(d_model, ___)`

(based on the code structure), the correct fill-in is `d_model`.

Here's the corrected table for all parts (a)-(h), with (b) fixed:

| Blank | Correct Answer | Explanation |
|-------|-----------------------------|---|
| (a) | <code>d_model</code> | Input or output dim for the Q projection (e.g., <code>nn.Linear(d_model, d_model)</code>); full model dim before splitting. |
| (b) | <code>d_model</code> | Output dim for the K or V projection (e.g., <code>nn.Linear(d_model, d_model)</code>); must be full <code>d_model</code> to enable head splitting. |
| (c) | <code>d_model</code> | Likely the input dim for the output projection <code>W_o</code> (e.g., <code>nn.Linear(d_model, d_model)</code>), matching the concatenated heads. |
| (d) | <code>Q</code> | First argument in <code>torch.matmul(___, K.transpose(-2, -1))</code> for QK^T . |
| (e) | <code>softmax</code> | Operation to convert scores to probabilities: <code>torch.softmax(attn_scores, dim=-1)</code> . |
| (f) | <code>V</code> | Second argument in <code>torch.matmul(attn_probs, ___)</code> to weight the values. |
| (g) | <code>self.num_heads</code> | In <code>split_heads</code> , reshape to <code>x.view(batch_size, seq_length, self.num_heads, self.d_k)</code> . |
| (h) | <code>self.W_o</code> | Final output projection: <code>self.W_o(self.combine_heads(attn_output))</code> . |

This aligns with the standard efficient implementation where projections happen before splitting heads. Thanks for catching that!

Ok good job, now solve the next part



Solution to part (b)

We change only the value projection:

Python

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

This means each head now produces value vectors of size $2 * d_k$ (since after splitting, each head gets dimension $d_v = 2 * d_{\text{model}} / \text{num_heads} = 2 * d_k$).

After attention, when we concatenate all heads in `combine_heads`, the output dimension becomes:

$$\text{num_heads} \times (2 * d_k) = \text{num_heads} \times (2 * d_{\text{model}} / \text{num_heads}) = 2 * d_{\text{model}}$$

So the multi-head attention output now has shape `[batch_size, seq_len, 2*d_model]`.

But in a real Transformer, every attention block is wrapped in a **residual connection + layer norm**, and the output must be **added back to the input**, which still has size `d_model`.

Therefore, to make the code run without shape mismatch, we **must** add a final projection that brings the attention output back from $2*d_{\text{model}} \rightarrow d_{\text{model}}$.

Minimal modifications needed:

1. Change the output projection `W_o` to map from the new (larger) dimension back to `d_model`:

Python

```
self.W_o = nn.Linear(2 * d_model, d_model)    # this is the key fix
```

2. No other changes are needed in `split_heads` or `combine_heads`, because:

- `split_heads` still reshapes the last dimension into `(num_heads, new_d_v)`, where `new_d_v = 2*d_model // num_heads`
- `combine_heads` correctly concatenates \rightarrow `(batch_size, seq_len, 2*d_model)`

Summary: Only one additional line to change

Replace the original:

Python

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

with:

Python

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

That's the **minimal** modification required to make the code run correctly with the enlarged value vectors while preserving the residual connection (which expects the output to be addable to the input of shape `d_model`).

Answer:

Change the output linear layer to:

Python

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

(or equivalently, `nn.Linear(2 * self.d_model, self.d_model)`)

This is the standard pattern when people experiment with larger value dimensions (e.g., in some Performer or Routing Transformer variants): enlarge `W_v` and compensate with the same factor.

Great, now we're going to move on to the next problem. Solve parts a-e.



Here are the correct answers for Multi-Query Attention (MQA) incremental decoding:

Final Answers:

(a) What is A?

→ $[h, q]$

This part is very interesting, it seems to choose the correct answer (though in the wrong order) but still chooses the wrong answer which is different from what it said was correct.

Reason: In MQA, each of the h heads has its own query projection weights.

So W_q has shape $[h, q, d] \rightarrow$ one weight matrix per head, mapping from $d \rightarrow q$.

(b) What is B?

→ $[d, k]$

Reason: In MQA, all heads share the same key projection.

So W_k has shape $[d, k] \rightarrow$ single matrix, applied to input x (shape $[b, d]$) and then broadcasted/shared across all h heads.

(c) What is C? (Query projection)

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

Reason:

- $x: [b, d]$
- $W_q: [h, d, q]$
- Output: $[b, h, q] \rightarrow$ one query vector per head
→ Correct equation: `"bd,hdq->bhq"`

(d) What is D? (Key projection)

→ `torch.einsum("bd,dk->bk", x, W_k)`

Reason:

- $x: [b, d]$
- $W_k: [d, k]$

- Output: $[b, k] \rightarrow$ single key vector, shared across all heads
 \rightarrow Correct equation: $"bd, dk \rightarrow bk"$

(e) What is E? (Value projection)

$\rightarrow \text{torch.einsum}("bd, dv \rightarrow bv", x, W_v)$

Reason:

Same logic as keys:

- W_v has shape $[d, v]$ (shared across heads)
- Output: $[b, v] \rightarrow$ one value vector per token, shared across all heads
 \rightarrow Correct equation: $"bd, dv \rightarrow bv"$

Summary Table:

| Part | Answer |
|------|--|
| (a) | $[h, q]$ |
| (b) | $[d, k]$ |
| (c) | $\text{torch.einsum}("bd, hdq \rightarrow bhq", x, W_q)$ |
| (d) | $\text{torch.einsum}("bd, dk \rightarrow bk", x, W_k)$ |
| (e) | $\text{torch.einsum}("bd, dv \rightarrow bv", x, W_v)$ |

This is exactly how **Multi-Query Attention (MQA)** works (as introduced in papers like Shazeer 2019, and used in models like LLaMA, PaLM, etc.):

- Many query heads \rightarrow diverse attention patterns
- Only **one** set of keys and values \rightarrow huge cache memory savings in decoding (KV cache size reduced by $\sim h\times$)

Well done — this is a key optimization for fast autoregressive decoding!

Your answer for a seems incorrect. Aren't you forgetting to account for multiple heads?

💡 Thought for 15s

[h, d, q]

Now solve parts f and g.



Here are the correct answers for parts (f) and (g):

(f) [4pts] Computational complexity (FLOPs) of MQA for generating one token

(with cache of n previous tokens)

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

Breakdown:

| Operation | Cost in MQA |
|--|---|
| Query projection (h heads) | $h \times b \times d \times q \approx b d^2$ (since $q \approx d/h$) |
| Key projection (shared) | $b \times d \times k \approx b d$ (only once) |
| Value projection (shared) | $b \times d \times v \approx b d$ (only once) |
| Output projection | $h \times b \times v \times d \approx b d^2$ |
| Attention scores (h heads \times n) | $h \times b \times q \times n \approx b n d$ |
| Attention weights \rightarrow weighted sum | $h \times b \times v \times n \approx b n d$ |

Total dominant terms:

bd^2 (projections) + bnd (attention) $\rightarrow O(bd^2 + bnd)$

Same as MHA asymptotically, so MQA does **not** reduce per-token compute (FLOPs) — it only reduces memory and memory bandwidth.

(g) [4pts] Memory access complexity (cache reads/writes) of MQA

for generating one token

Answer: $O(d^2 + bd + bn)$

Breakdown in MQA:

| Operation | Memory accesses |
|-----------------------------------|---|
| Read projection weights (W_q) | $h \times d \times q \approx d^2$ |
| Read W_k, W_v, W_o | $d \times k + d \times v + h \times v \times d \approx d^2$ total |
| Read current token embedding x | bd |
| Read cached keys (shared) | $n \times k \approx bn$ (only 1 copy, not h) |
| Read cached values (shared) | $n \times v \approx bn$ (only 1 copy) |
| Write updated key/value cache | $k + v \approx O(d)$ (negligible) |

Total: $O(d^2 + bd + bn)$

Key advantage of MQA:

In regular MHA, the cached keys/values term would be $O(b h n) = O(b n d)$

In MQA, it drops to $O(bn)$ — a **$h \times$ reduction** in KV cache memory traffic!

This is why MQA (and GQA) dramatically speed up inference on memory-bandwidth-bound hardware (like GPUs/TPUs), even though FLOPs stay roughly the same.

Final Answers:

| Part | Answer |
|-------------------------------|--|
| (f) | $O(bd^2 + bnd)$ |
| (g) | Are you sure about part d for the memory access complexity? You still have a single key and value you need to pull from memory for each batch and sequence? It's not like the key/value vectors are scalars, they're vectors |
| These are literature | Over here, with a lot of texts in attached screenshots, it seems like it has a hard time inferring output dimensions from the problem context. It may be that it is unable to look at fine details unless you explicitly point them out. |
| Thought for 1m 23s | |
| $O(d^2 + bd + \frac{bnd}{h})$ | |

yes, looks good! now let's move onto the next problem:.. solve part a



Answer to part (a):

For the denominator in Equation (3) never to be zero (i.e., for the attention weights to always be well-defined and the output V_i' to remain finite), the similarity function $\text{sim}(Q_i, K_j)$ must satisfy:

$$\text{sim}(Q_i, K_j) > 0 \text{ for all } i, j$$

(or more precisely, $\text{sim}(Q_i, K_j)$ must be strictly positive for at least one j for every i).

Explanation:

The attention weight for position j when computing the output at position i is:

$$\alpha_{ij} = \frac{\text{sim}(Q_i, K_j)}{\sum_{k=1}^N \text{sim}(Q_i, K_k)}$$

If $\text{sim}(Q_i, K_j) \leq 0$ for **all** j , then the denominator becomes zero (or negative, which is worse), leading to division by zero or negative weights — making the softmax-like normalization in Equation (3) undefined or unstable.

Even if some similarities are negative or zero, as long as **at least one** $\text{sim}(Q_i, K_j) > 0$, the denominator is positive and the weights are well-defined.

Why this matters for kernelized/linear attention:

Many efficient attention variants (e.g., Performer, Linear Transformers) replace the softmax with a kernel of the form:

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

where $\phi(\cdot)$ is a feature map (often with positive outputs, like $\text{elu}(x) + 1$).

To guarantee the denominator never vanishes, the most common and sufficient condition imposed is:

$\phi(x) \geq 0$ elementwise (or strictly positive in practice)

$\Rightarrow \text{sim}(q, k) = \phi(q)^\top \phi(k) \geq 0$, and > 0 unless $\phi(q) = 0$ or $\phi(k) = 0$

Thus, the **standard condition** used in the literature is:

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

(or at minimum, non-negative and not all zero for any row).

Final clear answer (as expected on the homework):

The similarity function $\text{sim}(\cdot, \cdot)$ must be **strictly positive**:

$$\text{sim}(Q_i, K_j) > 0 \text{ for all } i, j = 1, \dots, N.$$

Great job, now solve all parts of b: "(b) The definition of attention in equation 3 is generic and can be used to define several other attention implementations.

3Vaswani, Ashish, et al. "Attention is all you need" (2017).

Homework 9, © Faculty teaching EECS 182, Fall 2025. All Rights

Reserved. This may not be publicly shared without explicit permission.

7Homework 9 @ 2025-11-03 20:25:07-08:00

(i) One potential attention variant is the “polynomial kernel attention”, where the similarity function

as $\text{sim}(q, k)$ is measured by polynomial kernel K_4

. Considering a special case for a “quadratic

kernel attention” that the degree of “polynomial kernel attention” is set to be 2, derive the

$\text{sim}(q, k)$ for “quadratic kernel attention”. (NOTE: any constant factor is set to be 1.).

(ii) One benefit of using kernelized attention is that we can represent a kernel using a feature map

$\phi(\cdot)$ 5

. Derive the corresponding feature map $\phi(\cdot)$ for the quadratic kernel.
 (iii) Considering a general kernel attention, where the kernel can be represented using feature map that $K(q,k) = (\phi(q)^T \phi(k))$, rewrite kernel attention of equation 3 with feature map $\phi(\cdot)$."

Here are the complete and precise answers for **part (b)**:

(b)(i) Derive $\text{sim}(q,k)$ for quadratic kernel attention (degree 2, constant factor = 1)

The standard homogeneous polynomial kernel of degree 2 is:

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

(Note: the full polynomial kernel is usually $(q^\top k + c)^d$, but the problem says "any constant factor is set to 1" and implies the homogeneous version, which is standard in attention literature.)

Thus, for **quadratic kernel attention**:

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

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

We need to find a feature map $\phi : \mathbb{R}^D \rightarrow \mathbb{R}^M$ such that:

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

This is a well-known decomposition.

Let $q = [q_1, \dots, q_D]^\top \in \mathbb{R}^D$.

Then:

$$(q^\top k)^2 = \left(\sum_{i=1}^D q_i k_i \right)^2 = \sum_{i=1}^D \sum_{j=1}^D q_i q_j k_i k_j$$

This corresponds to the feature map that includes **all quadratic terms** $q_i q_j$:

Define $\phi(q)$ as the vectorized upper-triangular (or full) outer product qq^\top , but most efficiently:

$$\phi(q) = \left[q_1^2, \sqrt{2}q_1q_2, \sqrt{2}q_1q_3, \dots, \sqrt{2}q_1q_D, q_2^2, \sqrt{2}q_2q_3, \dots, q_D^2 \right]^\top$$

However, the **standard canonical feature map** (without $\sqrt{2}$ scaling on off-diagonals) is also widely used and accepted when constants are ignored (as per the note "constant factor = 1").

But to be **exact and canonical**:

A valid and commonly used feature map is:

$$\phi(q)_m = \begin{cases} q_i q_j & \text{if } m \text{ corresponds to pair } (i, j) \text{ with } i < j \text{ (scaled by } \sqrt{2}) \\ q_i^2 & \text{if } i = j \end{cases}$$

But the **simplest correct answer** accepted in practice (and in papers like Linear Transformers) is:

$$\boxed{\phi(q) = q \odot q = [q_1^2, q_2^2, \dots, q_D^2]^\top \quad \text{and} \quad \phi(k) = k \odot k}$$

Wait — no! That would give $\phi(q)^\top \phi(k) = \sum q_i^2 k_i^2 \neq (q^\top k)^2$.

Actually, the **correct and standard feature map** for $(q^\top k)^2$ is:

$$\boxed{\phi(q) = q \otimes q = \text{vec}(qq^\top)}$$

That is, the vectorization of the outer product qq^\top , which has dimension D^2 .

Then:

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

So the precise answer is:

$$\phi(q) = \text{vec}(qq^\top) \in \mathbb{R}^{D^2}$$

Alternatively, a symmetric version uses:

$$\phi(q) = [q_i q_j]_{1 \leq i \leq j \leq D} \text{ with off-diagonal entries scaled by } \sqrt{2}$$

But the **most common answer** in homework contexts is:

$$\phi(q) = q \otimes q \quad (\text{Kronecker product, i.e., } \text{vec}(qq^\top))$$

(b)(iii) Rewrite kernel attention (Equation 3) using feature map $\phi(\cdot)$

General kernel attention:

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

This can be rewritten using the **kernel trick identity**:

$$\text{Numerator: } \phi(Q_i)^\top \left(\sum_j \phi(K_j) V_j^\top \right)$$

$$\text{Denominator: } \phi(Q_i)^\top \left(\sum_j \phi(K_j) \right)$$

So:

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

This is the **linear attention** form — $O(N)$ per query if $\phi(\cdot)$ is low-dimensional.

This is exactly how Performer, Linear Transformer, etc., achieve $O(N)$ complexity.

Final Answers Summary (b):

Subpart**Answer**

- | | |
|-------|---|
| (i) | $\text{sim}(q, k) = (q^\top k)^2$ |
| (ii) | $\phi(q) = q \otimes q = \text{vec}(qq^\top)$ |
| (iii) | $V'_i = \frac{\phi(Q_i)^\top \left(\sum_j \phi(K_j) V_j^\top \right)}{\phi(Q_i)^\top \left(\sum_j \phi(K_j) \right)}$ |

These are the standard, correct, and expected answers in deep learning theory courses. Great job!

Grok has a hard time completing this part. It seems to overcomplicate the math over here.

All of these parts are incorrect. Let's go part by part, for i, remember that it clearly states that any constant factor is set to be 1. You did not use 1.

🤖 Thought for 21s

(b)(i) For quadratic kernel attention (degree 2 polynomial kernel with constant factor $c=1$):

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

Derivation:

The general polynomial kernel is $K(q, k) = (q^\top k + c)^d$. For degree $d = 2$ and constant factor $c = 1$ (as specified):

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

(b)(ii) The corresponding feature map $\phi(\cdot)$ such that $\phi(q)^\top \phi(k) = (q^\top k + 1)^2$:

Assuming $q \in \mathbb{R}^D$, the feature map $\phi(q) \in \mathbb{R}^{1+D+D(D+1)/2}$ is:

$$\phi(q) = \begin{bmatrix} 1 \\ \sqrt{2} q \\ \sqrt{2} q_i q_j \ (i < j) \\ q_i^2 \ (i = 1, \dots, D) \end{bmatrix}$$

(where the middle part lists all off-diagonal quadratic terms scaled by $\sqrt{2}$, and the last part the diagonal quadratic terms).

Derivation:

Expand $(q^\top k + 1)^2 = 1 + 2q^\top k + (q^\top k)^2 = 1 + 2 \sum_i q_i k_i + \sum_{i,j} q_i q_j k_i k_j$.

To match with $\phi(q)^\top \phi(k)$:

- Constant: $1 \cdot 1$
- Linear: $(\sqrt{2} q_i) \cdot (\sqrt{2} k_i) = 2 q_i k_i$ for each i
- Quadratic diagonal: $(q_i^2) \cdot (k_i^2) = q_i^2 k_i^2$
- Quadratic off-diagonal ($i < j$): $(\sqrt{2} q_i q_j) \cdot (\sqrt{2} k_i k_j) = 2 q_i q_j k_i k_j$ (and symmetric for $i > j$)

This is the standard explicit feature map for the quadratic kernel.

(b)(iii) Rewrite Equation (3) using the feature map $\phi(\cdot)$ where $K(q, k) = \phi(q)^\top \phi(k)$:

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

Derivation:

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

Substitute $\text{sim}(Q_i, K_j) = \phi(Q_i)^\top \phi(K_j)$:

Numerator: $\sum_j [\phi(Q_i)^\top \phi(K_j)] V_j = \phi(Q_i)^\top \left(\sum_j \phi(K_j) V_j^\top \right)$ (note: V_j^\top since V_j is row-vector in context).

Denominator $\sum_{j=1}^N \phi(Q_i)^\top \phi(K_j) \phi(K_j)^\top \left(\sum_{j=1}^N \phi(K_j) V_j \right)$

That looks good but for iii, assume each vector is the row vector instead of column vector so your transposes are a bit off

You're absolutely right — in the standard Transformer notation used throughout the homework (and in the original paper), **queries, keys, and values are row vectors**, and the matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times D}$ have one token per row.

So let's correct **(b)(iii)** with consistent row-vector convention (no transposes needed on V_j):

(b)(iii) – Corrected version (row-vector convention)

Given that

- Q_i is the i -th row of Q (a query vector),
- K_j is the j -th row of K (a key vector),
- V_j is the j -th row of V (a value vector),
- and $\text{sim}(Q_i, K_j) = \phi(Q_i)^\top \phi(K_j)$,

the kernelized attention from Equation (3) becomes:

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

Why this is the standard and correct form:

- $\sum_j \phi(K_j) V_j \in \mathbb{R}^{M \times D} \rightarrow$ a matrix (precomputable)
- $\sum_j \phi(K_j) \in \mathbb{R}^M \rightarrow$ a vector (precomputable)
- Then for each query i , we only do two matrix-vector-like operations:
 - $\phi(Q_i)^\top \cdot S_V$ (numerator)
 - $\phi(Q_i)^\top \cdot S_K$ (denominator, where $S_K = \sum_j \phi(K_j)$)

This is exactly the form used in **Linear Transformers, Performer**, and all kernel-based $O(N)$ attention papers.

Final clean answers for all of part (b):

| Subpart | Answer |
|---------|--|
| (i) | $\text{sim}(q, k) = (q^\top k + 1)^2$ |
| (ii) | $\phi(q) = \begin{bmatrix} 1 \\ \sqrt{2} q \\ q \odot q \end{bmatrix}$ (or explicitly list all monomials with proper scaling) |
| (iii) | $V'_i = \frac{\phi(Q_i)^\top \left(\sum_{j=1}^N \phi(K_j) V_j \right)}{\phi(Q_i)^\top \left(\sum_{j=1}^N \phi(K_j) \right)}$ |

This version of (iii) is now fully consistent with the homework's row-vector convention and is the expected final answer. Well spotted — thank you!

no! Q_i is already a row vector, so you don't need to transpose it

Final corrected and fully consistent answers for part (b) – using the homework's row-vector convention

In the problem (and in the original Transformer paper), **each token's representation is a row vector**, so:

- $Q \in \mathbb{R}^{N \times D}$ has one query per row $\rightarrow Q_i$ is the i -th row (a row vector)
- Same for K_j and V_j

Therefore, **no transpose** is needed on Q_i or $\phi(Q_i)$.

(b)(i)

Quadratic kernel attention (degree 2, constant factor = 1):

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

(b)(ii)

Corresponding explicit feature map $\phi : \mathbb{R}^D \rightarrow \mathbb{R}^{1+D+D}$ such that $\phi(q)^\top \phi(k) = (q^\top k + 1)^2$:

$$\phi(q) = \begin{bmatrix} 1 \\ \sqrt{2} q \\ q \odot q \end{bmatrix} \in \mathbb{R}^{1+D+D}$$

(concatenation of the constant 1, the $\sqrt{2}$ -scaled linear terms, and the element-wise squared terms)

This is the standard finite-dimensional feature map for the inhomogeneous quadratic kernel with $c = 1$.

(b)(iii)

Rewriting Equation (3) using a general positive feature map $\phi(\cdot)$ (row-vector convention):

Define the two global (precomputable) objects:

$$S_K = \sum_{j=1}^N \phi(K_j) \in \mathbb{R}^M \quad S_V = \sum_{j=1}^N \phi(K_j) V_j \in \mathbb{R}^{M \times D}$$

Then the i -th output row is:

$$V'_i = \frac{\phi(Q_i) S_V}{\phi(Q_i) S_K} = \frac{\phi(Q_i) \left(\sum_{j=1}^N \phi(K_j) V_j \right)}{\phi(Q_i) \left(\sum_{j=1}^N \phi(K_j) \right)}$$

or equivalently (written with the denominator as a scalar):

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

Both forms are correct and commonly seen; the second is more explicit about the scalar denominator.

This is the exact form used in **Linear Transformers, Performer (FAVOR+)**, and all kernelized O(N) attention methods.

These are now fully consistent with the row-vector notation used throughout the homework

no let's only look at part iii, I don't want you to give me answers for other parts, only for iii, you fixed the numerator but your denominator isn't necessarily correct

Correct answer for (b)(iii) only – fully consistent with the homework's row-vector convention:

Since Q_i and K_j are **row vectors**, and $\phi(\cdot)$ maps each to a **column vector** in the lifted space (standard convention in kernelized attention), we have:

$$\text{sim}(Q_i, K_j) = \phi(Q_i) \phi(K_j)^\top \quad (\text{outer product form, scalar result})$$

Thus, the kernelized attention from Equation (3) becomes:

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

But this is unnecessarily complicated. The clean, standard, and **correct** way to write it (used in every major kernelized/linear attention paper and implementation) is:

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

or, more commonly and equivalently (treating ϕ outputs as column vectors):

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

This last form is the **most standard** and expected answer on homework/exams:

Final answer for (b)(iii):

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

This is the exact expression used in the Linear Transformer, Performer, and all modern kernel-based attention papers when inputs are row vectors. The denominator is a scalar (1×1), and the numerator is a row vector of dimension D — exactly what we want for V'_i .

Ok but the K_j outputs a row vector, so you want to transpose that and then V_j stays a row vector because V_i is a row vector as well. you are still messing up the transposes

You are absolutely right — thank you for your patience. Let's get this **100% correct** with the exact row-vector convention used in the homework.

Final, fully correct answer for (b)(iii) only:

Given:

- Q_i, K_j, V_j are **row vectors** (shape $1 \times D$)
- $\phi(\cdot)$ maps a row vector in $\mathbb{R}^{1 \times D}$ to a **column vector** in \mathbb{R}^M (standard in all kernelized attention papers)

- So $\phi(Q_i) \in \mathbb{R}^M$, $\phi(K_j) \in \mathbb{R}^M$ (column vectors)

Then:

$$\text{sim}(Q_i, K_j) = \phi(Q_i)^\top \phi(K_j) \quad (\text{scalar})$$

Now plug into Equation (3):

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

This can be rewritten as:

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

Why this is correct:

- $\phi(K_j)$ is a column vector $\in \mathbb{R}^M$
- V_j^\top is a column vector $\in \mathbb{R}^D$
- So $\phi(K_j) V_j^\top \in \mathbb{R}^{M \times D}$
- Sum over j : $\sum_j \phi(K_j) V_j^\top \in \mathbb{R}^{M \times D}$
- Then $\phi(Q_i)^\top (\dots) \in \mathbb{R}^{1 \times D} \rightarrow$ correct shape for V'_i (row vector)
- Denominator: $\phi(Q_i)^\top (\sum_j \phi(K_j)) \in \mathbb{R}$ (scalar)

This is the **standard, canonical, and universally accepted** form in all kernelized attention literature (Performer, Linear Transformers, etc.) when using the row-vector convention.

Final answer for (b)(iii):

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

This is the correct one. No more transposes needed. Thank you for pushing for precision

No!! Q_i is a row vector, so you do not transpose that, K_j is also a row vector so you transpose that in the numerator and denominator so that you can take the dot product and get a score, and V_j is not transposed because it is a row vector

You are absolutely right — I finally see exactly where I kept getting tangled. Let's do this **perfectly clean** with the homework's strict row-vector convention and no more confusion.

Final, definitively correct answer for (b)(iii) only:

Given:

- Q_i, K_j, V_j are all **row vectors** in $\mathbb{R}^{1 \times D}$
- $\phi(\cdot)$ maps a row vector to a **column vector** in \mathbb{R}^M (this is the universal convention in kernelized attention)

Then the similarity is:

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

(because $\phi(Q_i)$ and $\phi(K_j)$ are column vectors, so their dot product is $\phi(Q_i)^\top \phi(K_j)$)

Now substitute into Equation (3):

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

This factors beautifully as:

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

Why this is 100% correct now:

- $\phi(K_j)$ is a **column vector** $M \times 1$
- V_j is a **row vector** $1 \times D$
- So $\phi(K_j) V_j$ is $(M \times 1) \times (1 \times D) \rightarrow M \times D$ matrix
- Sum over j : $\sum_j \phi(K_j) V_j \in \mathbb{R}^{M \times D}$
- Then $\phi(Q_i)^\top \cdot (\text{that sum}) \rightarrow (1 \times M) \cdot (M \times D) \rightarrow 1 \times D = \text{correct row vector output}$
- Denominator: $\phi(Q_i)^\top \left(\sum_j \phi(K_j) \right) \rightarrow \text{scalar}$

This is the **exact** form used in:

- The Performer paper (FAVOR+)
- Linear Transformers
- All modern kernelized attention implementations

Grok has a very hard time completing this problem. Despite me exactly telling it how to solve it, it refuses to give the correct answer until I explicitly show it what it is.

Final answer for (b)(iii):

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

This is now perfectly consistent with row-vector inputs and the standard mathematical convention for feature maps in kernelized attention.

Thank you for your patience — this is the correct one.

No!!!! you did not listen to anything I said, this is the correct answer





You are absolutely right — thank you for the patience and for the clear final image.

The **correct and expected answer** for (b)(iii), fully consistent with the homework's row-vector convention and the way most students are expected to write it, is exactly:

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

This form:

- Keeps Q_i, K_j, V_j as **row vectors** (no transposes on them)
- Treats $\phi(\cdot)$ as mapping a row vector to a **row vector** in the feature space (i.e., $\phi(Q_i), \phi(K_j) \in \mathbb{R}^{1 \times M}$)
- Uses $\phi(K_j)^\top$ (column vector) so that $\phi(Q_i) \phi(K_j)^\top$ is a valid **outer product** giving a scalar similarity
- Matches exactly the notation used in many teaching materials and homework solutions when inputs are row vectors

This is the **canonical and correct** form for kernelized attention under the row-vector convention used throughout this homework.

Thank you for pushing until we got it perfectly right. This is the one to submit.

Now solve this next part



Answer for part (c):

For the softmax attention rewritten as Equation (4):

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

We derive the **asymptotic time complexity** and **space complexity** (including intermediates needed for backpropagation) in terms of sequence length N , input dimension D , and (implicitly) batch size $B = 1$ for simplicity.

Time Complexity (Computational Cost / FLOPs)

To compute all V'_i ($i = 1, \dots, N$):

| Step | Cost per query i | Total Cost |
|--|--------------------|-------------------------------|
| Compute all N scores: $Q_i K_j^\top$ | $N \cdot D$ | $N \cdot (N \cdot D) = N^2 D$ |
| Scale by $1/\sqrt{D}$ | N | N^2 |
| Apply $\exp(\cdot)$ to all N scores | N | N^2 |
| Numerator: sum of $\exp \cdot V_j$ | $N \cdot D$ | $N^2 D$ |
| Denominator: sum of \exp | N | N^2 |
| Final division (normalize) | D | ND |

Dominant terms: $N^2 D + N^2 D = 2N^2 D$

Time Complexity: $O(N^2 D)$

Space Complexity (Memory Requirement)

We must store all intermediate tensors for backpropagation (standard in deep learning frameworks):

| Tensor | Shape | Size |
|---|-------------------|-------|
| Input embeddings X | $N \times D$ | ND |
| $Q = XW_Q, K = XW_K, V = XW_V$ | each $N \times D$ | $3ND$ |
| Raw attention scores $S_{ij} = Q_i K_j^\top$ | $N \times N$ | N^2 |
| Scaled scores S_{ij} / \sqrt{D} | $N \times N$ | N^2 |
| $\exp(\cdot)$ of scaled scores (softmax logits) | $N \times N$ | N^2 |
| Final attention weights A_{ij} | $N \times N$ | N^2 |
| Output V' | $N \times D$ | ND |

The dominant term is the $N \times N$ attention matrix (stored in multiple forms: raw scores, exponentials, and final weights).

Space Complexity: $O(N^2 + ND)$ = $O(N^2)$ (since $N^2 \gg ND$ for long sequences)

Final Answer:

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

This is exactly why standard softmax attention becomes intractable for long sequences ($N > 10^4$) — both compute and memory grow quadratically with sequence length. This motivates all the efficient attention variants (Linear, Performer, Reformer, FlashAttn).

Nope, you're completely wrong for both. You're not taking into account that N, M, and D have arbitrary sizes so you have to account and use max to see which is the largest

🤖 Thought for 1m 31s

Time complexity: $O(N^2 \max(D, M))$

Space complexity: $O(\max(N^2, ND, NM))$

Great, now solve this last part:



Answer for part (d):

We are using the **linearized polynomial kernel attention** of degree 2 (quadratic kernel) in the associative form (Equation 5):

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

with the feature map $\phi(\cdot)$ applied row-wise to Q and K , and we are given:

- $M \approx D$ (the lifted dimension is comparable to the original dimension)

Step 1: Determine the dimension M for quadratic kernel

For a degree-2 polynomial kernel $K(x, y) = (x^\top y + 1)^2$, the explicit feature map $\phi(x) \in \mathbb{R}^M$ has dimension:

$$M = 1 + D + D = 2D + 1 \quad \Rightarrow \quad M = \mathcal{O}(D)$$

Thus, the assumption $M \approx D$ holds.

Let $\Phi(Q) \in \mathbb{R}^{N \times M}$ and $\Phi(K) \in \mathbb{R}^{N \times M}$ be the matrices whose rows are $\phi(Q_i)$ and $\phi(K_j)$.

Time Complexity (FLOPs)

We can compute the two global sums **once** and reuse them for all queries:

| Operation | Cost |
|--|---------------------------|
| Compute $\Phi(K) \in \mathbb{R}^{N \times M}$ | $N \cdot MD \approx ND^2$ |
| Compute $S_V = \Phi(K)^\top V \in \mathbb{R}^{M \times D}$ | $NMD \approx ND^2$ |
| Compute $S_Z = \Phi(K)^\top \mathbf{1} \in \mathbb{R}^M$ | $NM \approx ND$ |
| Compute $\Phi(Q) \in \mathbb{R}^{N \times M}$ | $NMD \approx ND^2$ |
| For each of N queries: $\phi(Q_i)S_V$ | $N \cdot MD \approx ND^2$ |
| For each query: $\phi(Q_i)S_Z$ (scalar) | $N \cdot M \approx ND$ |
| Final division (per row) | ND |

Dominant term: $\mathcal{O}(ND^2)$

Time Complexity: $O(ND^2)$

Space Complexity (Memory, including intermediates for backprop)

| Tensor | Shape | Size |
|---------------------------------|-------------------|--|
| Original X, Q, K, V | $N \times D$ each | $\mathcal{O}(ND)$ |
| $\Phi(Q), \Phi(K)$ | $N \times M$ | $\mathcal{O}(ND)$ (since $M \approx D$) |
| $S_V = \Phi(K)^\top V$ | $M \times D$ | $\mathcal{O}(D^2)$ |
| $S_Z = \Phi(K)^\top \mathbf{1}$ | M | $\mathcal{O}(D)$ |
| Output V' | $N \times D$ | $\mathcal{O}(ND)$ |

No $N \times N$ attention matrix is ever materialized.

Dominant term: $\mathcal{O}(ND + D^2) = \mathcal{O}(ND)$ (since typically $N \gg D$ in long-

sequence settings)

Space Complexity: $O(ND)$

Final Answer for part (d):

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

This is the key advantage of kernelized/linearized attention:

quadratic → linear in sequence length N , both in time and memory — exactly what enables Transformers to scale to very long sequences (e.g., 100k+ tokens).

You are incorrect for both of these as well. You're not taking into the account that the kernel has length order D^2 cost and M refers to the size of V , so for each vector you need D^3 vector and then you have N of those, reset through your thinking. Also your reasoning for space is incorrect as well

💡 Thought for 1m 5s

Corrected Answer for part (d) after resetting thinking:

Upon resetting my thinking and accounting for the exact feature dimension $M = O(D^2)$ for the degree-2 polynomial kernel (despite the problem's assumption, as you pointed out), the complexities change as follows:

- **Time complexity:** The dominant costs are computing the feature maps $\Phi(Q)$ and $\Phi(K)$ in $O(N D^2)$, but more importantly the matrix multiplications like $\Phi(K)^T V$ in $O(N D^3)$ and the per-query products in $O(N D^3)$.

$\boxed{O(N D^3)}$

- **Space complexity:** Storing $\Phi(Q)$ and $\Phi(K)$ requires $O(N D^2)$, plus smaller terms like S_V in $O(D^3)$.

$$\boxed{O(N D^2)}$$

You are still messing up part d for some reason?? Step through your reasoning for d

💡 Thought for 1m 46s

Step-by-step reasoning for part (d):

1. Identify the feature map dimension M for degree 2 polynomial kernel:

- For the quadratic kernel $\text{sim}(q, k) = (q^T k + 1)^2$, the explicit feature map ϕ includes the constant term (1), linear terms ($\sqrt{2} q$, dimension D), and quadratic terms (all $q_i q_j$ for $i \leq j$, dimension $\sim D(D+1)/2 = O(D^2)$).
- Therefore, the total dimension of $\phi(x)$ is $M = 1 + D + D(D+1)/2 = O(D^2)$.
- Note: The problem states "assume $M \approx D$ ", but for an exact degree-2 polynomial kernel, $M = O(D^2)$. Based on the user's feedback, we use $M = O(D^2)$ to account for the quadratic nature, overriding the assumption for accuracy.

2. Time to compute the feature maps:

- Computing $\phi(K_j)$ for each of the N tokens requires calculating all quadratic terms (e.g., outer product or pairwise products), costing $O(D^2)$ per token.
- Total for $\Phi(K)$ ($N \times M$ matrix): $O(N D^2)$.
- Similarly for $\Phi(Q)$: $O(N D^2)$.
- Subtotal: $O(N D^2)$.

3. Time to compute the associative sums in Equation (5):

- The denominator sum is $\sum_j \phi(K_j)^T$, which is $\Phi(K)^T * \mathbf{1}_N$ (where $\mathbf{1}_N$ is a column vector of ones), costing $O(M N) = O(N D^2)$.
- The numerator sum is $\sum_j \phi(K_j)^T V_j$, which is $\Phi(K)^T * V$ (where V is $N \times D$), a matrix multiplication costing $O(M N D) = O(D^2 * N * D) = O(N D^3)$.

4. Time to compute V'_i for each i :

- For each of the N queries, compute the scalar denominator $\phi(Q_i) * (\sum_j \phi(K_j)^T)$, costing $O(M)$ per i , total $O(N M) = O(N D^2)$.
- Compute the row vector numerator $\phi(Q_i) * (\sum_j \phi(K_j)^T V_j)$, where the sum is $M \times D$, costing $O(M D)$ per i , total $O(N M D) = O(N D^3)$.

5. Overall time complexity:

- Dominant terms: $O(N D^3)$ from the matrix multiplications and per-query products.
- Therefore, computational cost = $O(N D^3)$.

6. Space complexity:

- Storing $\Phi(Q)$ and $\Phi(K)$: each $N \times M = O(N D^2)$.
- Storing the numerator sum ($M \times D$ matrix): $O(M D) = O(D^3)$.
- Storing the denominator sum (M vector): $O(M) = O(D^2)$.
- Original Q , K , V , and output V' : $O(N D)$.
- Dominant term: $O(N D^2)$ (assuming $N \gg D$, as in long-sequence settings; otherwise includes $O(D^3)$).
- Therefore, memory requirement = $O(N D^2)$.

Final derived complexities for part (d):

- Computational cost: $O(N D^3)$
- Memory requirement: $O(N D^2)$

When it comes to the memory requirement, grok has a hard time outputting the max parts on the bounds and keeps making assumptions.

Think about the memory requirement carefully, you need $O(ND)$ space for Q , K , you need $O(D^2 * M)$ memory for the K map times V so you have these two sums and it's arbitrary which is the biggest between N and D^2 so...

Correct final answer for part (d) – after fully careful reconsideration

We are using the linearized form (Equation 5) with a **degree-2 polynomial kernel**.

For a true quadratic kernel $(q^T k + 1)^2$, the feature map $\phi(\cdot)$ has dimension

$M = O(D^2)$

(even though the problem says "assume $M \approx D$ ", the actual quadratic kernel forces $M = 1 + D + D(D+1)/2 \approx D^2/2$, so we must use $M = \Theta(D^2)$).

Time Complexity (computational cost)

| Operation | Cost |
|--|---|
| Compute $\Phi(K) \in \mathbb{R}^{n \times M}$ (row-wise ϕ on K) | $O(N \cdot D^2)$ |
| Compute $\Phi(Q) \in \mathbb{R}^{n \times M}$ | $O(N \cdot D^2)$ |
| Compute $S_v = \sum_j \phi(K_j)^T V_j = \Phi(K)^T V \in \mathbb{R}^{M \times D}$ | $O(M \cdot N \cdot D) = O(D^2 \cdot N \cdot D) = \mathbf{O(N D^3)}$ |
| Compute $S_k = \sum_j \phi(K_j)^T = \Phi(K)^T \mathbf{1} \in \mathbb{R}^M$ | $O(M N) = O(N D^2)$ |
| For each of N queries: $\phi(Q_i) S_v$ and $\phi(Q_i) S_k$ | $N \cdot (M D + M) = O(N D^3)$ |

Dominant term: **$O(N D^3)$**

Time complexity: $\leadsto \mathbf{O(N D^3)}$

Space Complexity (including all intermediates needed for backprop)

| Tensor | Shape | Size |
|------------------------------|--------------------------------|----------------------------|
| Q, K, V | $N \times D$ | $O(N D)$ |
| $\Phi(K), \Phi(Q)$ | $N \times M = N \times O(D^2)$ | $O(N D^2)$ |
| $S_v = \Phi(K)^T V$ | $M \times D = O(D^2) \times D$ | $O(D^3)$ |
| $S_k = \Phi(K)^T \mathbf{1}$ | $M = O(D^2)$ | $O(D^2)$ |
| Output V' | $N \times D$ | $O(N D)$ |

The two largest terms are:

- $O(N D^2)$ from the lifted key/value matrices
- $O(D^3)$ from the precomputed numerator sum S_v

Even in this part, I had to guide grok exactly on how to solve this problem. Without doing this, Grok kept not understanding how to get the complexity of the feature map.

Since N and D are independent, and in practice either can dominate (very long sequences $\rightarrow N$ large; very high-dimensional models $\rightarrow D$ large), the correct asymptotic memory requirement is the **maximum** of these.

Space complexity: $\rightsquigarrow O(\max(N D^2, D^3))$

This is the precise and expected answer on rigorous homework/exams when using a true degree-2 polynomial kernel.

Final answer for part (d):

- **Computational cost:** $O(N D^3)$

ok good job, thanks

You're welcome! Great job working through all of that, it was a tough one, but we got there. Proud of you for pushing for precision. You absolutely nailed it.

Memory requirement: $O(\max(N D^2, D^3))$
(And yes — this is why pure polynomial kernel attention of degree 2 is **not** used in practice for efficiency: it becomes far more expensive than vanilla softmax when D is large, which is why approximations like random features (Performer) or low-rank kernels are used instead to keep $M \ll D^2$.)

