

EECS 182 Homework 9

Special Participation A

Athul Krishnan with Claude Opus 4.5 (Extended Thinking)

Contents

1 Overview/Recap	2
2 Introduction	3
3 Question 1: Justifying Scaled Dot-Product Attention	4
4 Question 2: Argmax Attention	8
5 Question 3: Multi-Head Attention Implementation	12
6 Question 4: Transformer Decoding Optimization	17
7 Question 6: Kernelized Linear Attention (Part I)	22

1 Overview/Recap

This section is repeated from my Ed Post for the gradescope submission!

For Special Participation A, I evaluated Claude Opus 4.5 (Extended Thinking) on HW9, regarding transformers and attention! To do so, I fed in the entire problem PDF, as well as the following starting prompt:

User

"Hi Claude! I'd like you to walk me through each question of the deep learning assignment I've attached to this message, as an LLM enthusiastic about teaching others about transformers, your inner workings!

I will specify a specific question for you to answer, and let's work on a single question at a time."

Claude was incredibly strong, one-shotting nearly every question. I expected it to struggle with deep chains of algebra, and the occasional numerical calculation, but it did very well! I occasionally re-prompted it to see if it could make minor simplifications, and clarified some of its conceptual statements, but it took them with ease.

The only spot where it did hiccup a bit was question 3b (where even the staff solution is insufficient, I believe. This concern is also in the Ed thread for HW09 Solutions). It correctly identified the staff solution of changing the dimensions of W_o , and noticed necessary modifications to the `combine_head` function, but did not mirror those modifications to the `split_head` function. After re-prompting it to examine the `split_head` function, Claude was able to edit it as needed, to what I believe is a fully correct answer.

Overall, Claude is very strong at all aspects of these highly technical questions. It organized its thoughts well, answered each question thoroughly, and required a minimal hint to converge at the correct answer in the single case where it was incorrect. I found its greatest strength to be its conceptual insights for the math-heavy questions (e.g. Question 6), which filled in many of the intuition gaps I had when reading through the staff solutions!

I also used Claude to generate the LaTeX for the transcript!

2 Introduction

User

Hi Claude! I'd like you to walk me through each question of the deep learning assignment I've attached to this message, as an LLM enthusiastic about teaching others about transformers, your inner workings!

I will specify a specific question for you to answer, and let's work on a single question at a time. Do you understand?

Claude

Absolutely, I understand! I'd be happy to walk you through this homework on transformers and attention mechanisms—topics I find genuinely exciting since they're quite literally how I work under the hood.

The assignment covers some really elegant ideas:

I like that it's already giving me conceptual background.

- **Q1:** The mathematical justification for why we scale dot-product attention by \sqrt{d}
- **Q2:** A thought experiment on argmax attention (and why it would break training)
- **Q3:** Implementing multi-head attention in PyTorch
- **Q4:** Optimizations like Multi-Query Attention for efficient decoding
- **Q5:** Visualizing attention patterns in GPT and BERT
- **Q6:** Kernelized linear attention (connecting attention to kernel methods for efficiency)

Just let me know which question you'd like to tackle first, and we'll dig in. I'll aim to build intuition for *why* things work the way they do, not just the mechanics—which I know you appreciate from your teaching style in ME132.

3 Question 1: Justifying Scaled Dot-Product Attention

User

Let's start with question 1 in its entirety. What would the solutions to each subpart be? Be sure to show all of your work.

Claude

This question gets at something fundamental about transformer design—*why* do we divide by \sqrt{d} in attention? Let's build up the answer rigorously.

Setup: We have $q, k \in \mathbb{R}^d$ with $q, k \stackrel{iid}{\sim} N(\mu, \sigma^2 I)$. This means each component q_i and k_i is independently drawn from $N(\mu, \sigma^2)$.

Part (a): Find $\mathbb{E}[q^\top k]$ in terms of μ , σ , and d

The dot product expands as:

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

Taking the expectation and using linearity:

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

Since q and k are **independent** vectors, their components are independent of each other:

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

Summing over all d components:

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

since μ is given as both a vector and a scalar, this is correct.

Claude

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

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

Since terms across different indices are independent:

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

For a single term, recall the variance formula:

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

First term: Since q_i and k_i are independent:

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

For $X \sim N(0, 1)$, we have $\mathbb{E}[X^2] = \text{Var}(X) + (\mathbb{E}[X])^2 = 1 + 0 = 1$
So: $\mathbb{E}[(q_i k_i)^2] = 1 \cdot 1 = 1$

Second term:

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

Therefore:

$$\text{Var}(q_i k_i) = 1 - 0^2 = 1$$

Summing over all d components:

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

Claude

Part (c): Find s such that $\mathbb{E}[q^\top k/s] = 0$ and $\text{Var}(q^\top k/s) = 1$

Mean condition:

$$\mathbb{E}\left[\frac{q^\top k}{s}\right] = \frac{\mathbb{E}[q^\top k]}{s} = \frac{d \cdot 0^2}{s} = 0$$

This is satisfied for any $s \neq 0$. ✓

Variance condition:

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

Setting this equal to 1:

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

The Intuition

This is *exactly* why the attention formula uses $\frac{QK^\top}{\sqrt{d}}$!

Without scaling, as d grows large, the dot products $q^\top k$ would have variance d —meaning they'd take on very large positive or negative values. When you feed large-magnitude values into softmax, it saturates toward a one-hot vector (all attention on one key). This creates two problems:

1. **Vanishing gradients:** Softmax in saturation has near-zero gradients
2. **Lost expressivity:** You can't softly attend to multiple positions

By scaling by \sqrt{d} , we keep the pre-softmax logits in a “reasonable” range (variance ≈ 1) regardless of dimension, ensuring softmax stays in its useful, gradient-friendly regime.

great overview!

User

Fantastic work! As a minor reformulation, do you think it would possible to rewrite your answer to part (a) as the squared L2 norm of mu?

Claude

Yes, you're absolutely right to point that out! I was being sloppy with the notation. Looking back at the problem statement, $\mu \in \mathbb{R}^d$ is a **vector**, not a scalar. So each component has its own mean:

$$q_i, k_i \sim N(\mu_i, \sigma^2)$$

This means the expectation calculation should be:

$$\mathbb{E}[q^\top k] = \sum_{i=1}^d \mathbb{E}[q_i] \mathbb{E}[k_i] = \sum_{i=1}^d \mu_i \cdot \mu_i = \sum_{i=1}^d \mu_i^2$$

And that sum is precisely the squared L2 norm of μ :

$$\mathbb{E}[q^\top k] = \|\mu\|_2^2$$



This is a cleaner and more general formulation. It also makes the special case in parts (b) and (c) more natural: when $\mu = \mathbf{0}$ (the zero vector), we get $\|\mu\|_2^2 = 0$, so the expected dot product is zero regardless of dimension.

Thanks for the catch—this is the kind of notational precision that matters when you're writing things up!

User

Given that d is a variable you are allowed to use, and μ is overloaded in the given notation as both a vector and a scalar, I think the answer is fine! Let's move on to question 2: argmax attention. Same procedure as last time, what are the solutions to each subpart? Be sure to show all of your work.

4 Question 2: Argmax Attention

Claude

This is a great thought experiment that highlights *why* transformers use softmax attention. Let's work through it.

Part (a): Compute the argmax attention output

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:

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

Step 1: Compute similarity scores (dot products)

$$q^\top k_1 = (1)(1) + (1)(2) + (2)(0) = 1 + 2 + 0 = 3$$

$$q^\top k_2 = (1)(0) + (1)(3) + (2)(4) = 0 + 3 + 8 = 11$$

$$q^\top k_3 = (1)(5) + (1)(0) + (2)(0) = 5 + 0 + 0 = 5$$

$$q^\top k_4 = (1)(0) + (1)(0) + (2)(1) = 0 + 0 + 2 = 2$$

Step 2: Apply argmax

Scores: [3, 11, 5, 2]

$$\text{argmax}([3, 11, 5, 2]) = [0, 1, 0, 0]$$

The maximum score is 11, corresponding to key k_2 .

Step 3: Compute weighted sum of values

$$\text{output} = 0 \cdot v_1 + 1 \cdot v_2 + 0 \cdot v_3 + 0 \cdot v_4 = v_2$$

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

One shotted an arithmetic question!

Claude

Part (b): How does argmax affect our ability to train?

The fundamental problem is that **argmax is not differentiable**, which breaks gradient-based learning.

The gradient problem:

Consider what happens during backpropagation. The argmax function produces a one-hot vector, and its gradient is:

- **Zero almost everywhere:** Small perturbations to the input scores don't change which index is maximum, so the output doesn't change, meaning $\frac{\partial \text{output}}{\partial \text{scores}} = 0$
- **Undefined at ties:** When two scores are equal, the argmax has a discontinuous jump

Consequences for learning:

1. **No gradient signal to queries or keys:** Since $\frac{\partial \mathcal{L}}{\partial q} = \frac{\partial \mathcal{L}}{\partial \text{scores}} \cdot \frac{\partial \text{scores}}{\partial q}$, and the first term requires gradients through argmax (which are zero), we get no learning signal.
2. **Can't learn "soft" attention patterns:** The model can never learn to attend to multiple keys simultaneously—it's always all-or-nothing on a single key.
3. **No smooth optimization landscape:** The loss as a function of parameters becomes piecewise constant, with discontinuous jumps when the argmax switches. Gradient descent can't navigate this.

Why softmax fixes this:

Softmax is a **smooth, differentiable relaxation** of argmax:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

- It approximates argmax (high scores get most of the probability mass)
- But it's differentiable everywhere
- Small changes in scores produce small changes in outputs
- Gradients flow through to queries and keys, enabling learning

This is analogous to why we use cross-entropy loss with softmax for classification instead of directly optimizing 0-1 accuracy—we need smooth gradients for optimization!

User

For your bullet point "zero almost everywhere", do you mean that only the specific maximum value experiences any sort of gradient, not that the output as a whole doesn't change?

Claude

Great clarifying question—let me be more precise. Actually, the gradient is zero at all indices, including the maximum. Here's why: Consider scores [3, 11, 5, 2] producing argmax output [0, 1, 0, 0].

- If I perturb score k_1 : [3.01, 11, 5, 2] → still [0, 1, 0, 0] → **no change**
- If I perturb score k_2 (the max): [3, 11.01, 5, 2] → still [0, 1, 0, 0] → **no change**
- If I perturb score k_3 : [3, 11, 5.01, 2] → still [0, 1, 0, 0] → **no change**

i think it misinterpreted what I meant by gradient, I meant only for values. This should be apparent as argmax isolates a specific value to appear in score.

The argmax output is locally constant with respect to **all** scores, as long as the perturbations don't change which index achieves the maximum. The one-hot vector stays identical regardless of which score you nudge.

So $\frac{\partial \text{argmax}}{\partial z_i} = 0$ for all i almost everywhere.

The only place the gradient is undefined (not just zero) is exactly at tie points—like [5, 5, 3, 2]—where an infinitesimal perturbation causes a discontinuous jump in the output. Thanks for pushing on that—my original phrasing (“the output doesn’t change”) was correct but could have been clearer that this applies even to perturbations of the winning score!

User

Isn't the gradient 1 at the argmax index? like you have in your output? Or am I misinterpreting

brain fart, whoops

Claude

I think you might be conflating the **value** of the output with the **gradient** of the output.

The value: Yes, the output at the argmax index is 1. We have $\text{argmax}([3, 11, 5, 2]) = [0, 1, 0, 0]$, so the output value at index 2 is 1.

The gradient: This asks how the output *changes* when we perturb the input scores. That is, we want $\frac{\partial \text{output}_i}{\partial z_j}$.

Even at the argmax index:

- Input scores: [3, 11, 5, 2] → Output: [0, 1, 0, 0]
- Perturbed scores: [3, 11.01, 5, 2] → Output: [0, 1, 0, 0]

The output at index 2 is still 1. It didn't change from 1 to 1.01 or anything—it's stuck at exactly 1. So:

$$\frac{\partial \text{output}_2}{\partial z_2} = \lim_{\epsilon \rightarrow 0} \frac{1 - 1}{\epsilon} = 0$$

The output is a **constant** function of the scores (locally), so its derivative is zero everywhere—including at the winning index.

Does that distinction make sense?

User

Apologies, I miswrote my earlier statement. I meant, wouldn't the gradients in the backward pass flow through the one index selected by argmax?

Claude

You're absolutely right, and I was being imprecise about *where* the gradient blockage occurs. Let's trace the computation graph:

Forward pass:

1. Compute scores: $s_j = q^\top k_j$
2. Compute weights: $w = \text{argmax}(s) = [0, 1, 0, 0]$
3. Compute output: output = $\sum_j w_j v_j = v_2$

Backward pass:

For the **values**: You're correct! The output is literally just v_2 , so:

$$\frac{\partial \mathcal{L}}{\partial v_2} = \frac{\partial \mathcal{L}}{\partial \text{output}}, \quad \frac{\partial \mathcal{L}}{\partial v_{j \neq 2}} = 0 \quad \checkmark$$

Gradients absolutely flow through to the selected value vector.

The problem is with queries and keys:

To update q and k , we need:

$$\frac{\partial \mathcal{L}}{\partial q} = \frac{\partial \mathcal{L}}{\partial w} \cdot \frac{\partial w}{\partial s} \cdot \frac{\partial s}{\partial q}$$

That middle term $\frac{\partial w}{\partial s}$ is the gradient of argmax with respect to scores—which is zero. So the blockage is specifically: **we can learn the value projections, but we cannot learn which keys to attend to.** The model has no gradient signal telling it “you should have attended to a different key.”

Thanks for the correction—I should update my answer to part (b) to be more precise about this!

correct! well clarified, even with my erroneous re-prompt.

5 Question 3: Multi-Head Attention Implementation

User

Fantastic! This makes a lot of sense. Let's move onto question 3. Be sure to show all of your work!

Claude

This is a great exercise in understanding the mechanics of multi-head attention. Let me trace through the dimensions and logic carefully.

Part (a): Fill in the blanks

First, let's understand the architecture. In standard multi-head attention:

- Input has dimension `d_model`
- We split into `num_heads` heads, each with dimension `d_k = d_model // num_heads`
- Each head does attention independently
- We concatenate and project back to `d_model`

Blank (a): `d_model`

```
self.W_k = nn.Linear(d_model, ___(a)___)
```

The key projection should match the query projection—both map from `d_model` to `d_model`, which then gets split across heads.

(a) = `d_model`



Blank (b): `d_model`

```
self.W_v = nn.Linear(___(b)___, d_model)
```

The value projection takes the input (dimension `d_model`) and outputs `d_model`, same as Q and K.

(b) = `d_model`



Blank (c): Given as `d_model`

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

The output projection maps from the concatenated heads (`d_model`) back to `d_model`. This is essential for residual connections!

(c) = `d_model`



Claude

Blank (d): Q

```
attn_scores = torch.matmul(___(d)___, K.transpose(-2, -1))
```

Attention scores are computed as QK^\top . We need Q on the left, K transposed on the right. Shape check: Q is [batch, heads, seq_q, d_k], K transposed is [batch, heads, d_k, seq_k], result is [batch, heads, seq_q, seq_k]. ✓

(d) = Q ✓

Blank (e): softmax

```
attn_probs = torch.___(e)___(attn_scores, dim=-1)
```

We apply softmax to convert scores to probabilities. The `dim=-1` means we normalize over the key dimension (each query's attention weights sum to 1).

(e) = softmax ✓

Blank (f): V

```
output = torch.matmul(attn_probs, ____(f)___)
```

We compute the weighted sum of values using the attention probabilities.

Shape check: `attn_probs` is [batch, heads, seq_q, seq_k], V is [batch, heads, seq_k, d_k], result is [batch, heads, seq_q, d_k]. ✓

(f) = V ✓

Blank (g): Given as `self.num_heads`

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

We reshape from [batch, seq, d_model] to [batch, seq, num_heads, d_k], then transpose to [batch, num_heads, seq, d_k].

(g) = `self.num_heads` ✓

Claude

Blank (h): W_o

```
output = self._____ (h) _____ (self.combine_heads(attn_output))
```

After combining heads back to shape [batch, seq, d_model], we apply the output projection.

(h) = W_o



Summary Table

Blank	Answer
(a)	d_model
(b)	d_model
(c)	d_model
(d)	Q
(e)	softmax
(f)	V
(g)	self.num_heads
(h)	W_o

One-shot!

Part (b): Modifying W_v to output $2 * d_{model}$

If we change:

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

Now value vectors have dimension $2 * d_{model}$ instead of d_{model} . Let's trace through what breaks:

The problem:

1. After `split_heads`, V will have shape [batch, num_heads, seq, 2*d_k] instead of [batch, num_heads, seq, d_k]
2. After attention, output has shape [batch, num_heads, seq, 2*d_k]
3. After `combine_heads`, we get [batch, seq, 2*d_model]
4. But W_o expects input of size d_{model} !

Claude

Minimal modifications needed:

1. Update `W_o`:

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

2. Update `combine_heads`: The current implementation uses `self.d_model` to reshape, but now the value dimension is `2 * d_model`:

```
def combine_heads(self, x):  
    batch_size, _, seq_length, d_v = x.size()  
    return x.transpose(1, 2).contiguous().view(  
        batch_size, seq_length, self.num_heads * d_v)
```

Or introduce a new attribute `self.d_v = 2 * d_model` and use that.

Why the hint mentions residual connections:

In a transformer, attention blocks have residual connections:

```
output = x + attention(x)
```

This requires the attention output to have the same dimension as the input (`d_model`). Our modified `W_o` ensures this by projecting from `2 * d_model` back down to `d_model`.

User

Would you have to modify `split_heads` at all?

← *split-heads and combine-heads have to agree. This is what led me to ask this.*

Claude

Great catch—yes, you're absolutely right! Let me trace through more carefully.

Current `split_heads`:

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

The problem:

When we call `self.split_heads(self.W_v(V))`:

- Input shape: $[\text{batch}, \text{seq}, 2 * \text{d_model}]$ (since `W_v` now outputs $2 * \text{d_model}$)
- The reshape tries: `x.view(batch, seq, self.num_heads, self.d_k)`
- This expects the last dimension to equal `num_heads * d_k = d_model`
- But we have $2 * \text{d_model} \neq \text{d_model} \rightarrow \text{Reshape fails!}$

The fix:

We need to introduce a separate `d_v` attribute and modify `split_heads` to handle values differently:

```
def __init__(self, d_model, num_heads):
    ...
    self.d_k = d_model // num_heads
    self.d_v = 2 * d_model // num_heads # NEW
    ...

def split_heads(self, x, is_value=False):
    batch_size, seq_length, _ = x.size()
    head_dim = self.d_v if is_value else self.d_k
    return x.view(batch_size, seq_length,
                  self.num_heads, head_dim).transpose(1, 2)
```

Then in forward:

```
V = self.split_heads(self.W_v(V), is_value=True)
```

So the **complete list of minimal modifications** is:

1. Add `self.d_v = 2 * d_model // num_heads`
2. Modify `split_heads` to accept the value dimension
3. Modify `combine_heads` similarly
4. Update `W_o` to accept $2 * \text{d_model}$

Thanks for pushing on that—I missed a step in my original answer!

Makes sense to me.

6 Question 4: Transformer Decoding Optimization

User

Fantastic work! Let's move on to question 4. Be sure to justify all of your answers.

Claude

This question explores Multi-Query Attention (MQA), an important optimization for efficient inference. The key insight is that **all heads share the same keys and values**, while each head still has its own queries.

Part (a): What is A? (shape of W_q)

Looking at the output shape we need for q :

$q = \text{----C----} \ # \text{ needs to produce } [b, h, q]$

We need to produce separate queries for each head (that's the whole point—heads can still attend differently). Starting from x with shape $[b, d]$, to get $[b, h, q]$ we need W_q to have a head dimension.

$[h, d, q]$

Justification: This matches the MHA case because queries remain per-head in MQA. Each head needs its own query to attend to different aspects of the shared keys.

claudie
sprinkling
intuitive
justifications
really helps
to understand
It discourages
pattern matching

Part (b): What is B? (shape of W_k)

Looking at how keys are used:

`new_K = torch.cat([prev_K, k.unsqueeze(1)], dim=1) # [b, n+1, k]`

The cached keys have shape $[b, n, k]$ —**no head dimension!** This means k must have shape $[b, k]$, so W_k maps directly from d to k without any head dimension.

$[d, k]$

Justification: This is the core of MQA—keys are shared across all heads, so we don't need separate key projections per head. This reduces the weight matrix from $[h, d, k]$ to $[d, k]$.

Claude

Part (c): What is C? (einsum for q)

We need:

- Input: x with shape $[b, d]$
- Weights: W_q with shape $[h, d, q]$
- Output: q with shape $[b, h, q]$

The einsum must contract over d :

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

Justification:

- " $bd, dq \rightarrow bhq$ " is invalid—can't create h dimension from nothing
- " $bhd, dq \rightarrow bhq$ " assumes x has shape $[b, h, d]$, but x is $[b, d]$

I really appreciate its insights on how other choices are incorrect.

Part (d): What is D? (einsum for k)

We need:

- Input: x with shape $[b, d]$
- Weights: W_k with shape $[d, k]$
- Output: k with shape $[b, k]$ (shared across heads!)

I also like it tying everything to the overarching picture.

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

Justification:

- " $bd, hdk \rightarrow bkh$ " would require W_k to have shape $[h, d, k]$, but it's $[d, k]$
- " $bd, dk \rightarrow bkh$ " can't produce an h dimension from these inputs

Part (e): What is E? (einsum for v)

We need:

- Input: x with shape $[b, d]$
- Weights: W_v with shape $[d, v]$
- Output: v with shape $[b, v]$ (shared across heads!)

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

Justification: Same logic as keys—values are shared, so no head dimension. The other options either require wrong weight shapes or try to create dimensions that don't exist.

Claude

Part (f): Computational complexity of MQA

Let's count operations for each step:

Projections:

- Query: $[\mathbf{b}, \mathbf{d}] \times [\mathbf{h}, \mathbf{d}, \mathbf{q}] \rightarrow O(bd \cdot hq) = O(bd \cdot d) = O(bd^2)$ since $hq = d$
- Key: $[\mathbf{b}, \mathbf{d}] \times [\mathbf{d}, \mathbf{k}] \rightarrow O(bd \cdot k) = O(bd \cdot d/h) = O(bd^2/h)$
- Value: $[\mathbf{b}, \mathbf{d}] \times [\mathbf{d}, \mathbf{v}] \rightarrow O(bd \cdot v) = O(bd^2/h)$
- Output: $[\mathbf{b}, \mathbf{h}, \mathbf{v}] \times [\mathbf{h}, \mathbf{v}, \mathbf{d}] \rightarrow O(bhvd) = O(bd^2)$

Total projections: $O(bd^2)$

Attention:

Looking at the einsum:

```
logits = torch.einsum("bhk,bnk->bhn", q, new_K) # [b, h, n+1]
```

This is b batches, each doing h dot products of length k with n keys: $O(bh nk) = O(bhn \cdot d/h) = O(bnd)$

Weighted values: $[\mathbf{b}, \mathbf{h}, \mathbf{n}] \times [\mathbf{b}, \mathbf{n}, \mathbf{v}] \rightarrow O(bh nv) = O(bnd)$

Total attention: $O(bnd)$

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

Justification: The computational complexity is actually the **same as MHA!** Each head still computes attention over all keys—we just happen to use the same keys. The savings in MQA come from memory access, not computation.

Claude

Part (g): Memory access complexity of MQA

Read weights:

- $W_q: [h, d, q] \rightarrow O(hd \cdot d/h) = O(d^2)$
- $W_k: [d, k] \rightarrow O(d \cdot d/h) = O(d^2/h)$
- $W_v: [d, v] \rightarrow O(d^2/h)$
- $W_o: [h, v, d] \rightarrow O(d^2)$

Total weights: $O(d^2)$

Read/write current embeddings:

$x, q, k, v, o, y: O(bd)$ each $\rightarrow O(bd)$ total

Read cached keys/values:

Here's where MQA shines! In MHA, we had:

- $prev_K: [b, h, n, k] \rightarrow O(bhnd/h) = O(bnd)$
- $prev_V: [b, h, n, v] \rightarrow O(bnd)$

In MQA, we have:

- $prev_K: [b, n, k] \rightarrow O(\frac{bnd}{h}) = O(bnd/h)$
- $prev_V: [b, n, v] \rightarrow O(\frac{bnd}{h})$

Total cached access: $O(\frac{bnd}{h})$

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

this is equivalent to the final answer, so I decided to give it a slightly veiled hint to see if it would simplify!

Justification: The key insight is that MQA reduces the KV cache memory access by a factor of h . For long sequences (large n) with many heads (large h), this is a significant savings. This is why MQA is popular for efficient inference—it's memory-bandwidth-bound, not compute-bound, so reducing memory access speeds up generation.

User

Since $hv = d$, can you simplify the last term in part g?

τ given instead of κ .

Claude

You're right! Since $hv = d$ (equivalently, $v = d/h$), we have:

$$\frac{bnd}{h} = bn \cdot \frac{d}{h} = bnv$$

Similarly, since $k = d/h$, this also equals bnk .

So the memory access complexity simplifies to:

$$O(d^2 + bd + bnk) \quad \text{or equivalently} \quad O(d^2 + bd + bnv)$$

gave both answers! fantastic!

This is cleaner and also makes the comparison with MHA more transparent:

	MHA	MQA
KV cache access	$O(bnd)$	$O(bnk)$

Since $k = d/h$, MQA reduces KV cache memory access by exactly a factor of h . This notation makes it obvious—we're accessing n keys of dimension k per batch, rather than n keys of dimension k for each of h heads.

✓ great overview.

User

Fantastic! You correctly noticed that I gave you values, not keys, and identified the 1:1 correspondence for substituting it in. Finally, let's move onto question 6.

This question was where Claude seriously impressed me. One-shot

7 Question 6: Kernelized Linear Attention (Part I) **+ fantastic intuitive explanations!**

Claude

What a great question to dive into! This is one of my favorite topics because it reveals the elegant mathematical structure underlying attention mechanisms. Let me walk you through each part.

Background Setup

The key insight here is that standard softmax attention has $O(N^2)$ complexity because we compute all pairwise interactions between N positions. The kernel view provides a path to linearize this.

The generalized attention equation is:

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

This is a **weighted average** of values, where weights come from query-key similarity. ✓

Part (a): Conditions on the similarity function

Question: What conditions must $\text{sim}(q, k)$ satisfy to ensure V'_i remains finite?

Analysis:

For V'_i to be well-defined, the denominator must never be zero:

$$\sum_{j=1}^N \text{sim}(Q_{i\cdot}, K_{j\cdot}) \neq 0 \quad \forall i$$

Sufficient condition:

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

Staff

✓ Answer says $\neq 0$, but
the question only deals kernel attention
so it's correct!

If the similarity function is **strictly positive** for all inputs, then the sum of N positive terms is guaranteed to be positive, ensuring no division by zero.

Why softmax works: The exponential function $\exp(q^T k / \sqrt{D}) > 0$ for all real values, which is exactly why softmax attention is always well-defined! ✓

Claude

Part (b): Polynomial Kernel Attention

(b)(i): Derive $\text{sim}(q, k)$ for quadratic kernel attention

The polynomial kernel is defined as:

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

For **degree 2** (quadratic) with the constant set to 1:

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



(b)(ii): Derive the feature map $\phi(\cdot)$ for the quadratic kernel

We need to find $\phi(\cdot)$ such that:

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

Let's expand the right side. For $q, k \in \mathbb{R}^D$:

$$(q^T k + 1)^2 = \left(\sum_{i=1}^D q_i k_i + 1 \right)^2$$

Expanding:

$$= \left(\sum_{i=1}^D q_i k_i \right)^2 + 2 \sum_{i=1}^D q_i k_i + 1$$

The squared term expands to:

$$= \sum_{i=1}^D q_i^2 k_i^2 + 2 \sum_{i < j} q_i q_j k_i k_j + 2 \sum_{i=1}^D q_i k_i + 1$$

For this to equal $\phi(q)^T \phi(k)$, we need corresponding terms to multiply. The feature map is:

$$\phi(x) = \begin{bmatrix} 1 \\ \sqrt{2} x_1 \\ \vdots \\ \sqrt{2} x_D \\ x_1^2 \\ \vdots \\ x_D^2 \\ \sqrt{2} x_1 x_2 \\ \vdots \\ \sqrt{2} x_{D-1} x_D \end{bmatrix}$$

This jump is too big for my liking, personally.

Claude

Components:

- 1 constant term: 1
- D linear terms: $\sqrt{2}x_i$
- D squared terms: x_i^2
- $\binom{D}{2}$ cross terms: $\sqrt{2}x_i x_j$ for $i < j$

Dimension of $\phi(x)$: $1 + D + D + \frac{D(D-1)}{2} = 1 + D + \frac{D(D+1)}{2} = O(D^2)$

Verification ($D = 2$ case):

$$\begin{aligned}\phi(q)^T \phi(k) &= 1 + 2q_1 k_1 + 2q_2 k_2 + q_1^2 k_1^2 + q_2^2 k_2^2 + 2q_1 q_2 k_1 k_2 \\ &= 1 + 2(q_1 k_1 + q_2 k_2) + (q_1 k_1 + q_2 k_2)^2 = (q^T k + 1)^2\end{aligned}$$



(b)(iii): Rewrite kernel attention with feature map

Starting from equation (3) with $\text{sim}(q, k) = K(q, k) = \phi(q)\phi(k)^T$:

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

Part (c): Complexity of Softmax Attention

For computing all V'_i using equation (4):

$$V'_i = \frac{\sum_{j=1}^N \exp\left(\frac{Q_{i\cdot} K_{j\cdot}^T}{\sqrt{D}}\right) V_j}{\sum_{j=1}^N \exp\left(\frac{Q_{i\cdot} K_{j\cdot}^T}{\sqrt{D}}\right)}$$



Time Complexity

Operation	Cost
Compute Q, K, V projections	$O(NFD + NFM)$
Compute QK^T ($N \times D$ times $D \times N$)	$O(N^2D)$
Apply softmax row-wise	$O(N^2)$
Multiply attention $\times V$ ($N \times N$ times $N \times M$)	$O(N^2M)$

I love these tables, expressing everything step by step!

Claude

Total time complexity:

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



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

Space Complexity

Stored Quantity	Size
Q	$O(ND)$
K	$O(ND)$
V	$O(NM)$
Attention scores QK^T/\sqrt{D}	$O(N^2)$
Softmax output (for backprop)	$O(N^2)$

Total space complexity:

$$O(N^2 + ND + NM)$$



The $O(N^2)$ term dominates for long sequences—this is the bottleneck that motivates linear attention!

Part (d): Complexity of Linearized Kernel Attention

Here's where the magic happens! Using equation (5):

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

The key insight: We can use the **associative property** of matrix multiplication to change the order of computation!

The Trick

Instead of computing N^2 pairwise similarities, we **precompute aggregates**:

Define:

- $S = \sum_{j=1}^N \phi(K_{j\cdot}) V_{j\cdot}^T \in \mathbb{R}^{D' \times M}$ (outer products summed)
- $z = \sum_{j=1}^N \phi(K_{j\cdot}) \in \mathbb{R}^{D'}$

Then for each query i :

$$V'_i = \frac{\phi(Q_{i\cdot})^T S}{\phi(Q_{i\cdot})^T z}$$

Claude

Time Complexity Analysis

For degree-2 polynomial kernel, the feature map $\phi(x)$ has dimension $D' = O(D^2)$.

Operation	Cost
Compute $\phi(K_j)$ for all j	$O(ND^2)$
Compute $S = \sum_j \phi(K_j)^T V_j$ (each outer product is $D^2 \times D$)	$O(ND^3)$
Compute $z = \sum_j \phi(K_j)$	$O(ND^2)$
Compute $\phi(Q_i)$ for all i	$O(ND^2)$
Compute $\phi(Q_i)S$ for all i	$O(ND^3)$
Compute $\phi(Q_i)z$ for all i	$O(ND^2)$

With $M \approx D$:

Total time complexity:

$$O(ND^2M) = O(ND^3)$$



Space Complexity Analysis

We compute $\phi(Q_i)$ and $\phi(K_j)$ on the fly rather than storing them all:

Stored Quantity	Size
V	$O(NM) = O(ND)$
S (the $D^2 \times D$ aggregate)	$O(D^2M) = O(D^3)$
z (the D^2 normalizer)	$O(D^2)$

Total space complexity:

$$O(ND + D^3)$$

✓ *I was surprised it got this one. I was very confused by the wording when I went through this question.*

Summary: The Big Picture

Attention Type	Time	Space
Softmax	$O(N^2D)$	$O(N^2)$
Linear (deg-2 kernel)	$O(ND^3)$	$O(ND + D^3)$

When is linear attention better?

When $N \gg D$ (long sequences with moderate embedding dimension):

- Softmax scales **quadratically** in sequence length
- Linear attention scales **linearly** in sequence length

This is why linear attention variants are so attractive for processing long documents, books, or genomic sequences where N can be in the thousands or millions!

The trade-off: We exchanged the N^2 factor for D^3 and D^2 factors. The degree-2 polynomial kernel requires $O(D^2)$ -dimensional features, but for most practical transformers, D (typically 64–512 per head) is much smaller than N for long sequences. ✓

Any questions about the derivations or intuition? I find the kernel perspective really illuminating for understanding what attention is actually computing!

super helpful!