

Special Participation A

Shervin Goudarzi

November 29, 2025

Model Performance Summary

The table below summarizes the initial accuracy of the model's response for each question.

Question	Model Performance
1. Justifying Scaled-Dot Product Attention	Needed Correction
2. Argmax Attention	One Shot
4. Transformer Decoding Optimization	One Shot
6. Kernelized Linear Attention (Part I)	Needed Corrections

Table 1: Summary of Model Performance on Initial Question Attempts.

Conversation Transcript and Solutions

Initial Setup

User: You will be assisting an undergraduate student with a problem set on deep neural networks for a deep learning class. Do not answer any problem at this time, only when I give you a problem. The following problems will be only theory based, so no coding should be done at any point. For the solutions you need to provide short, but very clear step-by-step derivation of your solution with explanation of your final answer.

Gemini: Understood. I am ready to assist you with your deep learning theory problems. I will focus on providing clear, step-by-step mathematical derivations for your solutions, using \LaTeX for all equations and variables.

1. Justifying Scaled-Dot Product Attention

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

1. Define $\mathbb{E}[q^T k]$ in terms of μ, σ and d .
2. Considering a practical case where $\mu = 0$ and $\sigma = 1$, define $\text{Var}(q^T k)$ in terms of d .
3. 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 $\mathbb{E}[\frac{q^T k}{s}]$ to be 0, and $\text{Var}(\frac{q^T k}{s})$ to be $\sigma = 1$. What should s be in terms of d ?

Gemini:

Solution

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

Since q and k are independent random vectors, and for each component $\mathbb{E}[q_i] = \mu_i$ and $\mathbb{E}[k_i] = \mu_i$:

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

Because of independence, $\mathbb{E}[q_i k_i] = \mathbb{E}[q_i] \mathbb{E}[k_i] = \mu_i \cdot \mu_i = \mu_i^2$.

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

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

Given $\mu = \mathbf{0}$ and $\sigma = 1$, we have $q_i, k_i \sim \mathcal{N}(0, 1)$.

- $\text{Var}(q^T k) = \text{Var}\left(\sum_{i=1}^d q_i k_i\right) = \sum_{i=1}^d \text{Var}(q_i k_i)$ (due to independence of indices).
- For the product of two independent standard normal variables X, Y :

$$\text{Var}(XY) = \mathbb{E}[X^2 Y^2] - (\mathbb{E}[XY])^2$$

- Since $\mu = 0$, $\mathbb{E}[q_i k_i] = \mathbb{E}[q_i] \mathbb{E}[k_i] = 0$.
- $\mathbb{E}[q_i^2 k_i^2] = \mathbb{E}[q_i^2] \mathbb{E}[k_i^2]$. For standard normal distribution, $\mathbb{E}[x^2] = 1$.
- Thus, $\text{Var}(q_i k_i) = 1 \cdot 1 - 0 = 1$.

Summing over d dimensions:

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

1(c) Determine s

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

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

Setting $\frac{d}{s^2} = 1$:

$$s = \sqrt{d}$$

2. Argmax Attention

User: Problem: (a) Perform argmax attention with the following keys \mathbf{K} and values \mathbf{V} , using the query $\mathbf{q} = [1, 1, 2]^T$:

$$\mathbf{K} = \left\{ \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 3 \\ 4 \end{bmatrix}, \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\} \quad \mathbf{V} = \left\{ \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ 4 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \right\}$$

(b) How does replacing softmax with argmax affect our ability to usefully train models involving attention?

Solution

Gemini: 2(a) Argmax Attention Output

1. Calculate Similarity Scores $s_i = \mathbf{q}^T \mathbf{k}_i$:

$$\begin{aligned} s_1 &= [1, 1, 2] \cdot [1, 2, 0]^T = 1 + 2 + 0 = 3 \\ s_2 &= [1, 1, 2] \cdot [0, 3, 4]^T = 0 + 3 + 8 = 11 \\ s_3 &= [1, 1, 2] \cdot [5, 0, 0]^T = 5 + 0 + 0 = 5 \\ s_4 &= [1, 1, 2] \cdot [0, 0, 1]^T = 0 + 0 + 2 = 2 \end{aligned}$$

The score vector is $\mathbf{s} = [3, 11, 5, 2]$.

2. **Apply Argmax:** The maximum score is 11, corresponding to index 2 (the second key).
3. **Compute Output:** Argmax attention returns the value vector corresponding to the maximum index.

$$\text{Output} = \mathbf{v}_2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}$$

2(b) Effect on Training

The argmax operation is a step function. Its derivative is zero almost everywhere (and undefined at the jump points). During backpropagation, gradients cannot flow through the argmax operation to update the weights of the query and key projections. This prevents the model from learning how to attend to different parts of the input, rendering gradient-based training impossible for the attention mechanism.

User: 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 a dimension of d . For each of the h heads, the dimensions of query, key, and value projections are $q = d/h$, $k = d/h$, and $v = d/h$ respectively.

The following pseudocode performs the multi-head self-attention operation for a single token generation in the sequence (for each sequence in our batch).

```

1 def multihead_attention_incremental(x, prev_K, prev_V, W_q, W_k, W_v, W_o):
2     """Multi-head Self-Attention (one step).
3     Args:
4     x: a tensor with shape [b, d] - current token embedding
5     prev_K: tensor with shape [b, h, n, k] - cached keys from prev. tokens
6     prev_V: tensor with shape [b, h, n, v] - cached values from prev. tokens
7     W_q: a tensor with shape [h, d, q] - query projection weights
8     W_k: a tensor with shape [h, d, k] - key projection weights
9     W_v: a tensor with shape [h, d, v] - value projection weights
10    W_o: a tensor with shape [h, v, d] - output projection weights
11 Returns:
12 y: a tensor with shape [b, d] - output embedding
13 """
14 # Project inputs to query, key, and value
15 q = torch.einsum("bd,hdq->bhq", x, W_q)    # [b, h, q]
16 k = torch.einsum("bd,hdk->bhk", x, W_k)    # [b, h, k]
17 v = torch.einsum("bd,hdv->bhv", x, W_v)    # [b, h, v]
18
19 # Append new key and value to previous cached states
20 new_K = torch.cat([prev_K, k.unsqueeze(2)], dim=2) # [b, h, n+1, q]
21 new_V = torch.cat([prev_V, v.unsqueeze(2)], dim=2) # [b, h, n+1, v]
22
23 # Compute attention scores, apply softmax, and get weighted values
24 logits = torch.einsum("bhq,bhnq->bhn", q, new_K)    # [b, h, n+1]
```

```

25 weights = torch.nn.functional.softmax(logits, dim=-1) # [b, h, n+1]
26 o = torch.einsum("bhn,bhnv->bhv", weights, new_V)      # [b, h, v]
27
28 # Project back to output dimension
29 y = torch.einsum("bhv,hvd->bd", o, W_o) # [b, d]
30
31 return y

```

Multi-Query Attention (MQA): Multi-query attention is a variant of multi-head attention where all heads share the same keys and values, reducing memory access and improving efficiency for incremental decoding.

```

1 def multiquery_attention_incremental(x, prev_K, prev_V, W_q, W_k, W_v, W_o):
2     """Multi-query Self-Attention (one step).
3     Args:
4     x: a tensor with shape [b, d] - current token embedding
5     prev_K: tensor with shape [b, n, k] - cached keys (shared across heads)
6     prev_V: tensor with shape [b, n, v] - cached values (shared across heads)
7     W_q: a tensor with shape _____A_____ - query projection weights
8     W_k: a tensor with shape _____B_____ - key projection weights
9     W_v: a tensor with shape [d, v] - value projection weights
10    W_o: a tensor with shape [h, v, d] - output projection weights
11    Returns:
12    y: a tensor with shape [b, d] - output embedding
13    """
14    # Project input to queries (per head), keys and values (shared)
15    q = _____C_____
16    k = _____D_____
17    v = _____E_____
18
19    # Append new key and value to previous cached states
20    new_K = torch.cat([prev_K, k.unsqueeze(1)], dim=1) # [b, n+1, k]
21    new_V = torch.cat([prev_V, v.unsqueeze(1)], dim=1) # [b, n+1, v]
22
23    # Compute attention scores, apply softmax, and get weighted values
24    logits = torch.einsum("bhk,bnk->bhn", q, new_K)      # [b, h, n+1]
25    weights = torch.nn.functional.softmax(logits, dim=-1) # [b, h, n+1]
26    o = torch.einsum("bhn,bnv->bhv", weights, new_V)      # [b, h, v]
27
28    # Project back to output dimension
29    y = torch.einsum("bhv,hvd->bd", o, W_o) # [b, d]
30
31    return y

```

First, fill in the blanks in the code/comments above by picking one:

- (a) (3pts) What is A?
☐ [h, d, q] ☐ [d, q] ☐ [h, q]
- (b) (3pts) What is B?
☐ [h, d, k] ☐ [d, k] ☐ [h, k]

(c) (4pts) 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) (4pts) What is D?

- ☐ `torch.einsum("bd,dk->bk", x, W_k)`
- ☐ `torch.einsum("bd,hdq->bhq", x, W_k)`
- ☐ `torch.einsum("bd,dk->bhk", x, W_k)`

(e) (4pts) 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) (4pts) Recall that for generating a batch of single tokens in multi-head-attention with a cache of n previous tokens, the operations have computational complexity $O(bd^2 + bnd)$ since it is $O(bd^2)$ to do the projections (query, key, value, output) and $O(bnd)$ from attention (dot products with n cached keys/values for each head). The memory access complexity is $O(d^2 + bd + bnd)$ since it is $O(d^2)$ to read weights, $O(bd)$ to read/write current token embeddings (x, q, k, v, o, y) , and $O(bnd)$ to read cached keys/values. The MHA arithmetic intensity is thus $\frac{O(bd^2 + bnd)}{O(d^2 + bd + bnd)}$.

- Both terms are kept; which one dominates depends on the relative size of n and d .
- If n is small, projections dominate; if n is large, memory access for cached keys/values dominates.

Now consider MQA. For generating a single token with a cache of n previous tokens, what is the computational complexity (in Big-O notation)?

(g) (4pts) For MQA generating a single token with a cache of n previous tokens, **what is the memory access complexity (in Big-O notation)?**

Gemini: Solution

4(a)-(e) Shapes and Operations

- (a) W_q shape: $[\mathbf{h}, \mathbf{d}, \mathbf{q}]$ (Multi-head projection).
- (b) W_k shape: $[\mathbf{d}, \mathbf{k}]$ (Single shared key projection).
- (c) Query calc: `torch.einsum("bd,hdq->bhq", x, W_q)`.
- (d) Key calc: `torch.einsum("bd,dk->bk", x, W_k)`.
- (e) Value calc: `torch.einsum("bd,dv->bv", x, W_v)`.

4(f) Computational Complexity (One Token)

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

Where bd^2 is the projection cost and bnd is the attention scoring cost.

4(g) Memory Access Complexity

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

The term $\frac{bnd}{h}$ reflects the reduced KV cache size in Multi-Query Attention (MQA).

User: 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 this¹.

Under this context, this question presents a formulation of attention via the lens of the kernel. A large portion of the context is adopted from Tsai et al.². In particular, attention can be seen as applying a kernel over the inputs with the kernel scores being the similarities between inputs. This formulation sheds light on individual components of the transformer's attention, and helps introduce some alternative attention mechanisms that replaces the "softmax" with linearized kernel functions, thus reducing the $O(N^2)$ complexity in memory and computation.

We first review the building block in the transformer. Let $x \in \mathbb{R}^{N \times F}$ denote a sequence of N feature vectors of dimensions F . A transformer³ is a function $T : \mathbb{R}^{N \times F} \rightarrow \mathbb{R}^{N \times F}$ defined by the composition of L transformer layers $T_1(\cdot), \dots, T_L(\cdot)$ as follows,

$$T_l(x) = f_l(A_l(x) + x). \quad (1)$$

The function $f_l(\cdot)$ transforms each feature independently of the others and is usually implemented with a small two-layer feedforward network. $A_l(\cdot)$ is the self attention function and is the only part of the transformer that acts across sequences.

We now focus on the self attention module which involves softmax. The self attention function $A_l(\cdot)$ computes, for every position, a weighted average of the feature representations of all other positions with a weight proportional to a similarity score between the representations. Formally, the input sequence x is projected by three matrices $W_Q \in \mathbb{R}^{F \times D}$, $W_K \in \mathbb{R}^{F \times D}$ and $W_V \in \mathbb{R}^{F \times M}$ to corresponding representations Q, K and V . The output for all positions, $A_l(x) = V'$, is computed as follows,

$$Q = xW_Q, \quad K = xW_K, \quad V = xW_V, \quad (2)$$

$$A_l(x) = V' = \text{softmax} \left(\frac{QK^T}{\sqrt{D}} \right) V. \quad (3)$$

Note that in the previous equation, the softmax function is applied rowwise to QK^T . Following common terminology, the Q, K and V are referred to as the "queries", "keys" and "values" respectively.

Equation (2) implements a specific form of self-attention called softmax attention where the similarity score is the exponential of the dot product between a query and a key. Given that subscripting a

¹<https://huggingface.co/blog/long-range-transformers>

²Tsai, Yao-Hung Hubert, et al. "Transformer dissection: a unified understanding of transformer's attention via the lens of kernel" (2019).

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

matrix with $i \cdot$ returns the i -th row as a vector and subscripting with $\cdot j$ returns the j -th column, we can write a generalized attention equation for any similarity function as follows,

$$V'_i = \frac{\sum_{j=1}^N \text{sim}(Q_{i\cdot}, K_{j\cdot}) V_{j\cdot}}{\sum_{j=1}^N \text{sim}(Q_{i\cdot}, K_{j\cdot})}. \quad (4)$$

Equation (3) is equivalent to equation (2) if we substitute the similarity function with $\text{sim}_{\text{softmax}}(q, k) = \exp\left(\frac{q^T k}{\sqrt{D}}\right)$. This can lead to

$$V'_i = \frac{\sum_{j=1}^N \exp\left(\frac{Q_{i\cdot} K_{j\cdot}^T}{\sqrt{D}}\right) V_{j\cdot}}{\sum_{j=1}^N \exp\left(\frac{Q_{i\cdot} K_{j\cdot}^T}{\sqrt{D}}\right)}. \quad (5)$$

For computing the resulting self-attended feature $A_l(x) = V'$, we need to compute all V'_i ($i \in \{1, \dots, N\}$) in equation (4).

- (a) Identify the conditions that needs to be met by the sim function to ensure that V'_i in Equation (4) remains finite (the denominator never reaches zero).
- (b) The definition of attention in equation (3) is generic and can be used to define several other attention implementations.
 - (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 . 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 $\phi(\cdot)$. 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 $\phi(\cdot)$.
- (c) We can rewrite the softmax attention in terms of equation (3) as equation (4). For all the V'_i ($i \in \{1, \dots, N\}$), derive the time complexity (asymptotic computational cost) and space complexity (asymptotic memory requirement) of the above softmax attention in terms of sequence length N, D and M .
NOTE: for memory requirement, we need to store any intermediate results for backpropagation, including all Q, K, V .
- (d) Assume we have a kernel K as the similarity function and the kernel can be represented with a feature map $\phi(\cdot)$, we can rewrite equation (3) with $\text{sim}(x, y) = K(x, y) = (\phi(Q_{i\cdot})^T \phi(K_{j\cdot}))$ in part (b). We can then further simplify it by making use of the associative property of matrix multiplication to

$$V'_i = \frac{\phi(Q_{i\cdot}) \sum_{j=1}^N \phi(K_{j\cdot})^T V_{j\cdot}}{\phi(Q_{i\cdot}) \sum_{j=1}^N \phi(K_{j\cdot})^T}. \quad (6)$$

Note that the feature map $\phi(\cdot)$ is applied row-wise to the matrices Q and K .

Considering using a linearized polynomial kernel $\phi(x)$ of degree 2, and assume $M \approx D$, derive the computational cost and memory requirement of this kernel attention as in...

Gemini: Solution

6(a) Conditions for finite V'_i . For the output to be finite, the normalization term (denominator) must be non-zero:

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

Since we are using a squared kernel (which produces non-negative values), we strictly require that the sum is strictly positive.

6(b) Kernelized Attention Definitions

(i) **Kernel:**

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

(ii) **Feature Map $\phi(q)$:** We need a map such that $\phi(q)\phi(k)^T = (q^T k + 1)^2 = (q^T k)^2 + 2q^T k + 1$. For a vector $q \in \mathbb{R}^D$, the explicit feature map $\phi(q)$ is a vector of dimension approximately $D^2/2$:

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

(iii) **Linearized Equation:** Using the associative property of matrix multiplication, we compute $\phi(K)^T V$ first. Note that the transpose is on $\phi(K)$ in the numerator and denominator sums.

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

6(c) Complexity and Pseudocode

Standard Softmax Attention:

- Time: $O(N^2 D)$
- Space: $O(N^2)$ (to store the attention matrix)

Linearized Kernel Attention:

- Time: $O(ND^3)$ (assuming the feature map size corresponds to D^2).
- Space: $O(ND^2 + D^3)$ (Storing feature maps and aggregated KV).

Pseudocode for Linearized Attention

```
def linearized_kernel_attention(Q, K, V):  
    """  
    Linearized Kernel Attention with Associative Property  
    V' = phi(Q) @ [phi(K).T @ V] / phi(Q) @ [phi(K).T @ ones]  
    """  
    N = Q.shape[0]
```

```

# 1. Apply Feature Map: Maps  $[N, D] \rightarrow [N, D^2]$ 
phi_Q = feature_map(Q)
phi_K = feature_map(K)

# 2. Compute Aggregated Key-Values (The "Linear" trick)
#  $[D^2, N] @ [N, M] \rightarrow [D^2, M]$ 
# This matrix is computed ONCE for the whole sequence
kv_agg = phi_K.T @ V

# 3. Compute Normalizer Aggregation
#  $[D^2, N] @ [N, 1] \rightarrow [D^2, 1]$ 
z_agg = phi_K.T @ torch.ones(N, 1)

# 4. Compute Numerator
#  $[N, D^2] @ [D^2, M] \rightarrow [N, M]$ 
numerator = phi_Q @ kv_agg

# 5. Compute Denominator
#  $[N, D^2] @ [D^2, 1] \rightarrow [N, 1]$ 
denominator = phi_Q @ z_agg

# 6. Final Division
output = numerator / (denominator + 1e-6)

return output

```