Discussion 11

Note: *Your TA will probably not cover all the problems on this worksheet. The discussion worksheets are not designed to be finished within an hour. They are deliberately made slightly longer so they can serve as resources you can use to practice, reinforce, and build upon concepts discussed in lectures, discussions, and homework.*

This Week's Cool AI Demo/Video:
Anthropic credit research: `https://www.youtube.com/watch?v=Y6wiWlcH5jM`

Why aren't LLMs deterministic with temperature 0?:

`https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/`

1 Positional Encoding

(a) Why do we need positional encoding? Describe a situation where word-order information is necessary for the task performed.

(b) What is *relative* positional encoding? How is it different from *absolute* positional encoding?

(c) **Rotary positional embeddings (RoPE)** are a way of encoding token positions by rotating pairs of embedding dimensions by an angle that grows with the position index, so that inner products between tokens automatically depend only on their relative positions. They've become a de-facto default in many modern LLMs.

Consider a sequence of N two-dimensional vectors

$$x_n := \left(x_n^{(1)}, x_n^{(2)}\right), \quad n = 1, 2, \ldots, N.$$

For a parameter $\theta \in \mathbb{R}$, the RoPE encoding at position n is

$$\mathrm{RoPE}(x_n, n) = R(n\theta)x_n, \qquad \text{where} \qquad R(\alpha) := \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}.$$

Show that the dot product between two RoPE-encoded vectors depends only on their *relative* positions. In particular, prove that for any integers m, n, and k, the following holds:

$$\mathrm{RoPE}(x, m)^\top \, \mathrm{RoPE}(y, n) \;=\; \mathrm{RoPE}(x, m+k)^\top \, \mathrm{RoPE}(y, n+k).$$

(a) Why do we need positional encoding? Describe a situation where word-order information is necessary for the task performed.

- input embeddings from input tokens don't capture position of token in sequence
- self-attention mechanism is permutation invariant
- sometimes word order changes the meaning, e.g.,
  a. Only John texted me on my birthday
  b. John only texted me on my birthday

(b) What is *relative* positional encoding? How is it different from *absolute* positional encoding?

Consider example (a) from previous part:

absolute → we encode "only" as first word, "John" as second word, + so on

relative → we encode "only" is one word before "John" + such
* more generalizable

In lecture, we discussed absolute encodings (like from Attention paper), where x_n is an input embedding, r_n is that token's position, + we combine them as $\tilde{x}_n = x_n + r_n$.

(c) **Rotary positional embeddings (RoPE)** are a way of encoding token positions by rotating pairs of embedding dimensions by an angle that grows with the position index, so that inner products between tokens automatically depend only on their relative positions. They've become a de-facto default in many modern LLMs.

→ similarity score QK^T is composed of inner products

Consider a sequence of N two-dimensional vectors

$$x_n := \left(x_n^{(1)}, x_n^{(2)}\right), \quad n = 1, 2, \ldots, N.$$

For a parameter $\theta \in \mathbb{R}$, the RoPE encoding at position n is — rotation matrix

$$\mathrm{RoPE}(x_n, n) = R(n\theta)x_n, \qquad \text{where} \qquad R(\alpha) := \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}.$$

Show that the dot product between two RoPE-encoded vectors depends only on their *relative* positions. In particular, prove that for any integers m, n, and k, the following holds:

$$\mathrm{RoPE}(x, m)^\top \mathrm{RoPE}(y, n) = \mathrm{RoPE}(x, m+k)^\top \mathrm{RoPE}(y, n+k).$$

$x = W_K \tilde{x} \quad y = W_Q \tilde{y}$

$\mathrm{RoPE}(x,m) = R(m\theta)x \qquad \mathrm{RoPE}(x,m+k) = R((m+k)\theta)x$

$\mathrm{RoPE}(y,n) = R(n\theta)y \qquad \mathrm{RoPE}(y,n+k) = R((n+k)\theta)y$

$\mathrm{RoPE}(x,m)^T \mathrm{RoPE}(y,n) = x^T R(m\theta)^T R(n\theta) y$

$\mathrm{RoPE}(x,m+k)^T \mathrm{RoPE}(y,n+k) = x^T R((m+k)\theta)^T R((n+k)\theta) y$

$$R(\alpha)^T R(\beta) = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{bmatrix}$$

$$= \begin{bmatrix} \cos\alpha\cos\beta + \sin\alpha\sin\beta & -\cos\alpha\sin\beta + \sin\alpha\cos\beta \\ -\sin\alpha\cos\beta + \cos\alpha\sin\beta & \sin\alpha\sin\beta + \cos\alpha\cos\beta \end{bmatrix}$$

we need the product to sum formulas from trig:

$$\sin(\alpha)\sin(\beta) = \frac{1}{2}[\cos(\alpha-\beta) - \cos(\alpha+\beta)]$$
$$\cos(\alpha)\cos(\beta) = \frac{1}{2}[\cos(\alpha-\beta) + \cos(\alpha+\beta)]$$
$$\sin(\alpha)\cos(\beta) = \frac{1}{2}[\sin(\alpha+\beta) + \sin(\alpha-\beta)]$$
$$\cos(\alpha)\sin(\beta) = \frac{1}{2}[\sin(\alpha+\beta) - \sin(\alpha-\beta)]$$

$$R(\alpha)^T R(\beta) = \begin{bmatrix} \cos(\alpha-\beta) & \sin(\alpha-\beta) \\ -\sin(\alpha-\beta) & \cos(\alpha-\beta) \end{bmatrix}$$

B/c cosine is even $\cos(-\theta) = \cos\theta$

B/c sine is odd $\sin(-\theta) = -\sin\theta$

$$R(\alpha)^T R(\beta) = \begin{bmatrix} \cos(\beta-\alpha) & -\sin(\beta-\alpha) \\ \sin(\beta-\alpha) & \cos(\beta-\alpha) \end{bmatrix} = R(\beta-\alpha)$$

$$\begin{aligned} RoPE(x,m)^T RoPE(y,n) &= x^T R(m\theta)^T R(n\theta)\, y \\ &= x^T R(n\theta - m\theta)\, y \end{aligned}$$

$$\begin{aligned} RoPE(x,m+k)^T RoPE(y,n+k) &= x^T R((m+k)\theta)^T R((n+k)\theta)\, y \\ &= x^T R((n+k)\theta - (m+k)\theta)\, y \\ &= x^T R(n\theta - m\theta)\, y \end{aligned}$$

$\therefore$ Shifting the position of both tokens by the same amount, k, won't change the dot product of embeddings

2 Matching Similarity Matrices to Attention Heatmaps

Recall that in self-attention mechanisms, we compute the key-query similarity matrix QK^T, scale the matrix by $\frac{1}{\sqrt{D_k}}$, and then apply the softmax function row-wise to obtain attention scores. Understanding how pre-softmax similarity scores translate to post-softmax attention weights is crucial for interpreting attention patterns. Below are four 4×4 pre-softmax similarity matrices and four corresponding post-softmax attention heatmaps. Your task is to match each pre-softmax matrix to its corresponding heatmap and explain the transformation.

Pre-softmax matrices:

QK^T

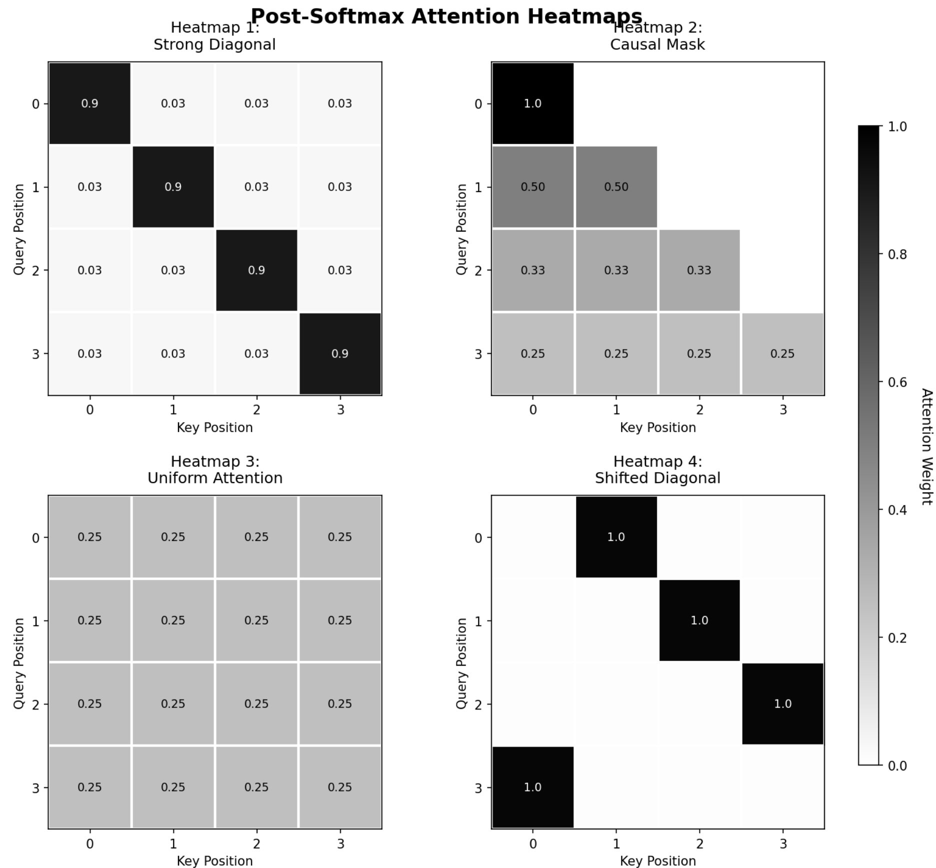
$$\text{Matrix A: } \begin{bmatrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix} \qquad \text{Matrix B: } \begin{bmatrix} 10 & 1 & 1 & 1 \\ 1 & 10 & 1 & 1 \\ 1 & 1 & 10 & 1 \\ 1 & 1 & 1 & 10 \end{bmatrix}$$

$$\text{Matrix C: } \begin{bmatrix} 5 & -\infty & -\infty & -\infty \\ 5 & 5 & -\infty & -\infty \\ 5 & 5 & 5 & -\infty \\ 5 & 5 & 5 & 5 \end{bmatrix} \qquad \text{Matrix D: } \begin{bmatrix} -\infty & 5 & -\infty & -\infty \\ -\infty & -\infty & 5 & -\infty \\ -\infty & -\infty & -\infty & 5 \\ 5 & -\infty & -\infty & -\infty \end{bmatrix}$$

Post-softmax heat maps (darker = higher attention weight):

$\text{SoftMax}_{row}\left(\frac{QK^T}{\sqrt{D_k}}\right)$

D_k = key/query dimension



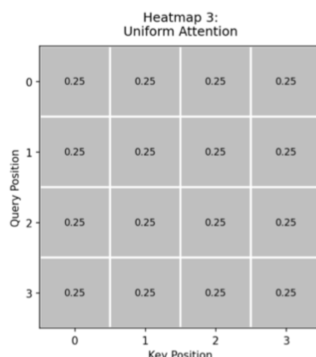
(a) Match each pre-softmax matrix (A-D) to its corresponding heatmap (1-4). Provide brief reasoning for each match.

(b) Explain why Matrix C represents a causal attention mask. In what type of model would you typically see this pattern?

(c) Consider a temperature-scaled softmax: $\text{softmax}(x/T)$ where T is temperature. How would changing T affect the attention distribution for Matrix B? What happens as $T \to 0$ and $T \to \infty$?

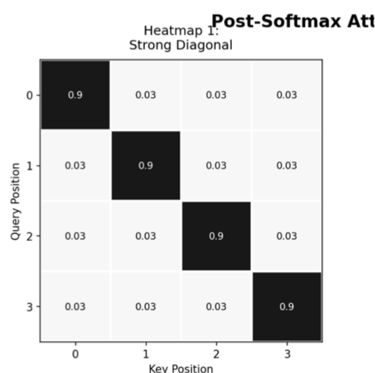
(a) Match each pre-softmax matrix (A-D) to its corresponding heatmap (1-4). Provide brief reasoning for each match.

Matrix A: $\begin{bmatrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix}$



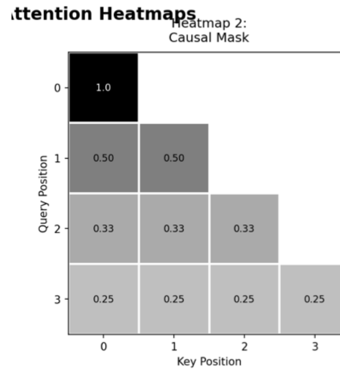
why are all the values 0.25?
we apply softmax row-wise, so each <u>row</u> sums to 1.

Matrix B: $\begin{bmatrix} 10 & 1 & 1 & 1 \\ 1 & 10 & 1 & 1 \\ 1 & 1 & 10 & 1 \\ 1 & 1 & 1 & 10 \end{bmatrix}$



How did we get 0.9 along diagonal?
we scale QK^T (matrix B) by $1/\sqrt{D_k}$ (which we don't provide) & then apply softmax row-wise.

Matrix C: $\begin{bmatrix} 5 & -\infty & -\infty & -\infty \\ 5 & 5 & -\infty & -\infty \\ 5 & 5 & 5 & -\infty \\ 5 & 5 & 5 & 5 \end{bmatrix}$



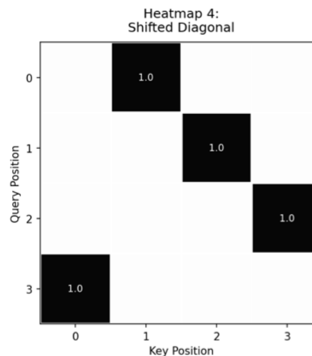
How did we get zeros in upper triangle?

$$\frac{e^{-\infty}}{ne^{-\infty} + (4-n)e^{5}} = 0$$ for any finite integer n

What about the lower triangle?

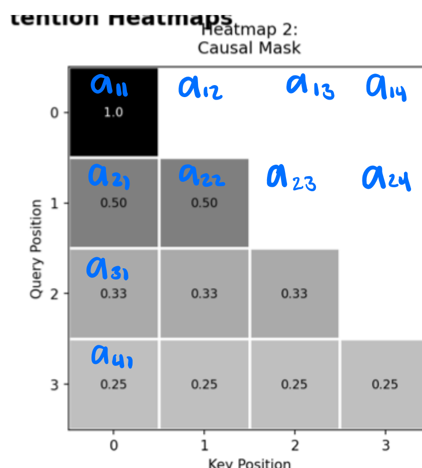
After setting upper triangle to 0, rows must sum to 1.

Matrix D: $\begin{bmatrix} -\infty & 5 & -\infty & -\infty \\ -\infty & -\infty & 5 & -\infty \\ -\infty & -\infty & -\infty & 5 \\ 5 & -\infty & -\infty & -\infty \end{bmatrix}$



(b) Explain why Matrix C represents a causal attention mask. In what type of model would you typically see this pattern?

Matrix C: $\begin{bmatrix} 5 & -\infty & -\infty & -\infty \\ 5 & 5 & -\infty & -\infty \\ 5 & 5 & 5 & -\infty \\ 5 & 5 & 5 & 5 \end{bmatrix}$



By setting the upper triangle to $-\infty$, we force the attention values in those positions to be 0.

Recall a_{ni} is attention that token n places on token i. Why would we want $a_{ni} = 0$ for $n < i$?

Consider the example from discussion mini lecture. During training, we have the following:

<u>Inputs</u>: where is the library <eos>

<u>Outputs</u>: <sos> Dónde está la biblioteca

<u>Targets</u>: Dónde está la biblioteca <eos>

During inference, we start w/ the following:

<u>Inputs</u>: where is the library <eos>

<u>Outputs</u>: <sos>

And we build the outputs one token at a time from our trained transformer's predictions.

To train a <u>good</u> (i.e., generalizable) model that can perform well during inference time, we don't want it to "cheat" during training. Token 1 should not be able to attend to token 2 b/c it won't be able to do this during inference. But token 4 can look at tokens 1-3 b/c it will have this info available during inference.

(c) Consider a temperature-scaled softmax: softmax(x/T) where T is temperature. How would changing T affect the attention distribution for Matrix B? What happens as $T \to 0$ and $T \to \infty$?

row-wise

Matrix B: $\begin{bmatrix} 10 & 1 & 1 & 1 \\ 1 & 10 & 1 & 1 \\ 1 & 1 & 10 & 1 \\ 1 & 1 & 1 & 10 \end{bmatrix} * \frac{1}{T}$, where $T = \sqrt{D_K}$

QK^T

Along the diagonal: $a_{diag} = \dfrac{e^{10/T}}{e^{10/T} + 3e^{1/T}}$

Off of diagonal: $a_{off} = \dfrac{e^{1/T}}{e^{10/T} + 3e^{1/T}}$

Low temperature $(T \to 0^+)$

i.e., low key/query dimension, D_K:

$$\lim_{T \to 0^+} \frac{e^{10/T}}{e^{10/T} + 3e^{1/T}} \cdot \frac{e^{-10/T}}{e^{-10/T}} = \lim_{T \to 0^+} \frac{1}{1 + 3e^{-9/T}} = \frac{1}{1+3(0)} = 1$$

$a_{diag} + 3a_{off} = 1$

$1 + 3a_{off} = 1 \to a_{off} = 0$

$$\lim_{T \to 0^+} \frac{e^{1/T}}{e^{10/T} + 3e^{1/T}} \cdot \frac{e^{-10/T}}{e^{-10/T}} = \lim_{T \to 0^+} \frac{e^{-9/T}}{1 + 3e^{-9/T}} = \frac{0}{1+3(0)} = 0$$

∴ The diagonal terms dominate off-diagonal ones.

High temperature ($T \to +\infty$)

i.e., high key/query dimension, D_k:

$$\lim_{T \to \infty} \frac{e^{10/T}}{e^{10/T} + 3e^{1/T}} = \frac{1}{1+3(1)} = \frac{1}{4}$$

$$a_{diag} + 3a_{off} = 1$$

$$\frac{1}{4} + 3a_{off} = 1 \rightarrow a_{off} = \frac{1}{4}$$

$$\lim_{T \to \infty} \frac{e^{1/T}}{e^{10/T} + 3e^{1/T}} = \frac{1}{1+3(1)} = \frac{1}{4}$$

$\therefore$ All of the terms get equal weight (i.e., attention becomes uniform)

3 KV Caching in Autoregressive Generation

In autoregressive language models like GPT, we generate tokens one at a time, conditioning on all previously generated tokens. This problem explores how KV caching dramatically improves the runtime of this process.

The Naive Approach:

Consider generating a sequence of length N with a transformer that has:

- Model (hidden) dimension, D_{model}
- Number of attention heads, H
- Number of attention layers, L

Without caching, at iteration n when generating token y_n, we compute attention as:

$$\text{Attention}(Q_n, K_{1:n}, V_{1:n}) = \text{SoftMax}_{row}\left(\frac{Q_n K_{1:n}^T}{\sqrt{D_{model}/H}}\right) V_{1:n}$$

where $K_{1:n}$ and $V_{1:n}$ are the keys and values for all tokens from position 1 to n. Notice that we recompute K_i and V_i for all $i < n$ at every step, even though these values don't change.

The KV Cache Solution: We store the computed K and V matrices from previous iterations and reuse them, only computing the new K_n and V_n for the current token.

(a) **Without KV caching**, how many times do we compute the key and value projections for the **first** token y_1 throughout the entire generation of a sequence of length N?

(b) For a single layer, what is the total number of matrix multiplications needed to compute **all** key projections $K_{1:N}$ throughout the entire generation process **without KV caching**? (Assume we compute each K_i individually at each iteration, i.e. $K_i = y_i W_k$, instead of stacking tokens i up to n into an $n \times D_{model}$ matrix to compute $K = XW_k$ at each iteration.) Express your answer in terms of N.

(c) **With KV caching**, how many matrix multiplications are needed to compute all key projections throughout the **entire** generation?

(d) Briefly explain why KV caching is particularly important for applications like chatbots or code assistants that involve multiple rounds of interaction.

Without caching, at iteration n when generating token y_n, we compute attention as:

$$\text{Attention}(Q_n, K_{1:n}, V_{1:n}) = \text{SoftMax}_{row}\left(\frac{Q_n K_{1:n}^T}{\sqrt{D_{model}/H}}\right) V_{1:n}$$

where $K_{1:n}$ and $V_{1:n}$ are the keys and values for all tokens from position 1 to n. Notice that we recompute K_i and V_i for all $i < n$ at every step, even though these values don't change.

(a) **Without KV caching**, how many times do we compute the key and value projections for the **first** token y_1 throughout the entire generation of a sequence of length N?

Every token in our sequence of length N will use the key/value of y_1 in the attention mechanism, so $K_1 + V_1$ would need to be computed N times if we don't cache them.

(b) For a single layer, what is the total number of matrix multiplications needed to compute **all** key projections $K_{1:N}$ throughout the entire generation process **without KV caching**? (Assume we compute each K_i individually at each iteration, i.e. $K_i = y_i W_k$, instead of stacking tokens i up to n into an $n \times D_{model}$ matrix to compute $K = XW_k$ at each iteration.) Express your answer in terms of N.

At iteration 1, we need K_1, so we have 1 mat. mul.

At iteration 2, we need $K_1 + K_2$, so we have 2 m.m.

⋮

At iteration N, we need $K_1, K_2, \ldots, K_N$, so we have N

In total, we have $\sum_{n=1}^{N} n = \frac{1}{2} N(N+1) = O(N^2)$

matrix multiplications

(c) **With KV caching**, how many matrix multiplications are needed to compute all key projections throughout the **entire** generation?

At iteration 1, we need k_1, so we have 1 mat. mul.

At iteration 2, we need k_1 & k_2, but we cached k_1, so we only have 1 mat. mul. for k_2

⋮

At iteration N, we need $k_1, k_2, \ldots, k_N$, but we cached $k_1, k_2, \ldots, k_{N-1}$, so we only have 1 mat. mul. for k_N

In total, we have exactly N matrix multiplications.

(d) Briefly explain why KV caching is particularly important for applications like chatbots or code assistants that involve multiple rounds of interaction.