Assumptions are written in red

PROBLEM 1

KV cache, batch size, and arithmetic intensity calculation: Write a script (in any language of your choice), that given model, hardware, and input specs enumerated below, computes the size of the KV cache per request and the maximum permissible batch size on the given hardware.

The model is considered with the following specifications (based on a LLaMA3-8B):

- Number of Layers: $L = 32$
- Number of query heads: $Q = 32$
- Number of KV heads: $K = 8$
- Embedding Dimension: $h_1 = 4096$
- FFN Inner Dimension: $h_2 = 14336$
- Vocabulary Size: $V = 128256$
- GPU Memory: 80 GB
- Precision: p = 2 bytes per parameter (FP16, same for weight and kv cache bytes)
- Context Length: $s = 1024$ (**Assumed**)

File name

```
question1.py
```

1(a) Find the maximum batch size that can fit on a GPU given the above specifications. Please output a table in the below format and show calculation for how you find KV cache per request.

Solution -

The number of KV heads 8 isn't explicitly used because the calculations are done on the full embedding dimension h_1 . Splitting into heads only redistributes h_1 into smaller chunks, so the overall parameter count and memory (which depend on h_1) remain unchanged.

Output of the program

```
PS C:\Users\Lenovo\Downloads\SML_Ass4> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python38/python.exe c:/User
s/Lenovo/Downloads/SML_Ass4/question1.py
----- Question 1(a) Results -----
Total model parameters: 6.43 Billion
Memory in bytes:  12861833216
Total parameter memory: 11.98 GB
KV cache per request: 0.50 GB
Max batch size possible: 136
```

1. Total model parameters - The total model parameters of transformer-based models come from three sources:

i. **Self-Attention Mechanism:** In each layer, self-attention projects input embeddings into Query (Q), Key (K), and Value (V) vectors. These projections are done using linear transformations, and then the output projection combines the results. Even though the model splits the computation into multiple attention heads for parallel processing and improved expressiveness, the total number of parameters remains the same as if it were computed in one block.

Attention Block per Layer:

$$\text{attn\_params} = 4 \times h_1^2 = 4 \times 4096^2$$

$$= 4 \times 16{,}777{,}216 = 67{,}108{,}864$$

ii. **Feedforward Network (FFN):** The FFN typically comprises two linear layers. The first (up-projection) increases the dimensionality from h_1 to h_2 to capture complex features, and the second (down-projection) reduces it back to h_1.

Feedforward Block per Layer:

$$\text{ffn\_params} = 2 \times h_1 \times h_2$$

$$= 2 \times 4096 \times 14336 = 117{,}440{,}512$$

Total Parameters per Layer (Attention + FFN):

$$\text{params\_per\_layer} = \text{attn\_params} + \text{ffn\_params}$$

$$= 67{,}108{,}864 + 117{,}440{,}512 = 184{,}549{,}376$$

For All Layers:

$$\text{layers\_params} = L \times \text{params\_per\_layer}$$

$$= 32 \times 184{,}549{,}376 = 5{,}905{,}580{,}032$$

iii. **Embedding Layer:** The embedding layer converts discrete token IDs into continuous vectors of dimension h_1.

$$\text{embed\_params} = V \times h_1$$

$$= 128256 \times 4096 = 525{,}336{,}576$$

Total Model Parameters:

$$N_{\text{params}} = \text{layers\_params} + \text{embed\_params}$$

$$= 5{,}905{,}580{,}032 + 525{,}336{,}576$$

$$= 6{,}430{,}916{,}608$$

$$\approx 6.43 \text{ Billion}$$

2. Total Parameter Memory - Since each parameter is stored using 2 bytes, using the formula:

$$\text{Parameter Memory} = N_{\text{params}} \times \textit{precision}$$

we get,

$$\text{Parameter Memory} = 6{,}430{,}916{,}608 \times 2 \text{ bytes} = 12{,}861{,}833{,}216 \text{bytes}$$

$$= \frac{12{,}861{,}833{,}216}{1024^3} \approx 11.98 \text{ GB}$$

Here, we have taken $1024^3 = 1GB$. If we had taken $10^9 = 1GB$, then total memory would be $12.86GB$. This represents the memory footprint solely for storing the model parameters.

3. KV Cache Memory per Request - For efficient autoregressive inference, the model caches the Key (K) and Value (V) matrices from each layer to avoid re-computation when generating each new token.

$$\text{KV Cache} = 2 \times \textit{precision} \times L \times h_1 \times s$$

where:

- The factor 2 is because both Key and Value matrices are stored
- $s = 1024$ tokens (context length)

Substituting, we have: KV Cache $= 2 \times 2 \times 32 \times 4096 \times 1024$

$$\text{KV Cache} = 4 \times 32 \times 2^{12} \times 2^{10}$$

$$= 2^{29} \text{ bytes}$$

$$\approx 512 \text{ MB}$$

$$\approx 0.5 \text{ GB}$$

4. Maximum Batch Size - The maximum batch size is determined by how many requests can be processed concurrently given the available GPU memory. The available memory for KV cache is the total GPU memory minus the model memory:

$$\text{Available Memory} = 80 \text{ GB} - 11.98 \text{ GB} = 68.02 \text{ GB}$$

The maximum batch size is given by:

$$B_{\max} = \frac{\text{Available Memory}}{\text{KV Cache per Request}}$$

we have:

$$B_{\max} = \frac{67.14 \text{ GB}}{0.5 \text{ GB}} \approx 136.04$$

Taking the floor (since the batch size must be an integer), the maximum batch size is:

$$B_{\max} = 136$$

Final Answer

Total model parameters	6.43 Billion
Total parameter memory (GB)	11.98 GB
KV cache per request (GB)	0.5 GB
Max batch size possible	136

1(b) Repeat the same exercise for a given Tensor Parallelism (TP) dimension 'X'. Assume the model and KV cache memory requirements are divided equally among X GPUs, and recalculate the maximum batch size.

In Tensor Parallelism (TP) with dimension X, when the model parameters and the KV cache are partitioned equally among X GPUs, each GPU stores only a $1/X$ fraction of the model and handles $1/X$ of the KV cache for every request.

Output of the program:

```
----- Question 1(b) Results -----
For Tensor Parallelism (TP) dimension X = 2:
Max batch size possible per GPU: 296
```

The key idea is that the total memory required for the model and the KV cache is divided among X GPUs. Thus:

1. **Model Memory per GPU:**

$$\text{Model Memory per GPU} = \frac{\text{Total Model Memory}}{X}$$

$$= \frac{11.98 \text{ GB}}{X}$$

2. **KV Cache per Request per GPU:** The KV cache (which is 0.5 GB per request in the single-GPU case) is also partitioned, so each GPU requires:

$$\text{KV Cache per Request per GPU} = \frac{0.5 \text{ GB}}{X}$$

3. **Available Memory per GPU for KV Cache:** Each GPU has 80 GB total, so after accounting for the partitioned model memory:

$$\text{Available Memory per GPU} = 80 \text{ GB} - \frac{11.98 \text{ GB}}{X}$$

Maximum Batch Size Calculation:

For each request, the KV cache memory required on one GPU is $\frac{0.5}{X}$ GB. Therefore, the maximum number of requests (i.e. the maximum batch size $B_{\max}$) that can be processed concurrently on each GPU is:

$$B_{\max} = \frac{\text{Available Memory per GPU}}{\text{KV Cache per Request per GPU}} = \frac{80 - \left(\frac{11.98}{X}\right)}{\frac{0.5}{X}}$$

Simplifying the fraction:

$$B_{\max} = \left(80 - \frac{11.98}{X}\right) \times \frac{X}{0.5} = \frac{X \times 80 - 11.98}{0.5}$$

$$B_{\max} = 2\,(80X - 11.98) = 160X - 23.96$$

Since the batch size must be an integer, we take the floor:

$$\boxed{B_{\max} = \lfloor 160X - 23.96 \rfloor}$$

Example:
For $X = 1$ (i.e. single GPU), the formula gives:

$$B_{\max} = \lfloor 160(1) - 23.96 \rfloor = \lfloor 136.04 \rfloor = 136$$

which matches our result from part (a).
For $X = 2$**:**

$$B_{\max} = \lfloor 160(2) - 23.96 \rfloor = \lfloor 320 - 23.96 \rfloor = \lfloor 296.04 \rfloor = 296$$

Final Answer (with TP)

Metric	**Value**
Total model parameters	6.43 Billion
Total parameter memory (GB)	11.98 GB
KV cache per request (GB)	0.5 GB
Max batch size (for TP dimension X)	$\lfloor 160X - 23.96 \rfloor$

1(c) Write down the detailed formula for total flops and total memory access(bytes) to calculate arithmetic intensity(flops/memory_bytes) for prefill and decode attention function. Assume the following attention implementation. Head dimension is d and context length is N.

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.
1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Write your calculations, specifically how you arrive at the total flops and memory access for both prefill and decode attention in terms of N and d.

Solution - We consider a standard single-head attention mechanism with head dimension d and context length N. We assume one multiply-add is counted as 2 FLOPs.
(i) Prefill Attention (Processing all N tokens at once):

FLOPs Calculation (Prefill):

1. **Compute $\mathbf{S} = \mathbf{QK}^\top$**:

$$Q \in \mathbb{R}^{N\times d}, \quad K \in \mathbb{R}^{N\times d}, \quad S \in \mathbb{R}^{N\times N}$$

   - To compute one entry in $S \in \mathbb{R}^{N\times N}$, we perform a dot product of two d-dimensional vectors. A dot product requires d multiplications and $d-1$ additions, which is $2d$ FLOPs (approximately, since we treat a multiply-add as 2 FLOPs).
   - Total cost:

$$\text{S\_flops} \approx 2\,N^2\,d$$

2. **Compute $P = \text{softmax}(S)$:** We apply exponentiation, sum, and division across $N \times N$ elements. This is typically smaller than large matrix multiplications, but we can denote it as $\alpha \times N^2$ for some constant α.

3. **Compute $\mathbf{O} = \mathbf{PV}$**:

$$P \in \mathbb{R}^{N\times N}, \quad V \in \mathbb{R}^{N\times d}, \quad O \in \mathbb{R}^{N\times d}$$

   - Each entry in $O \in \mathbb{R}^{N\times d}$ is computed by a dot product (again $2d$ FLOPs per entry).
   - Total cost:

$$\text{PV\_flops} \approx 2\,N^2\,d.$$

Thus, the **total FLOPs**(dominant) for the prefill attention are:

$$\boxed{\text{prefill\_flops} \approx 2\,N^2\,d + 2\,N^2\,d = 4\,N^2\,d}$$

(If softmax costs are included, they add a term on the order of $O(N^2)$, but for large d the matmul terms dominate.)

Memory Access Calculation (Prefill): We count the total number of elements read from or written to HBM(high-bandwidth memory) then multiply by bytes per element(BPE).

1. **Read:**

   - Q: $N \times d$ elements
   - K: $N \times d$ elements
   - V: $N \times d$ elements
   - P: $N \times N$ elements
   - Reading S for softmax: $N \times N$ elements

   Total read elements:

$$2Nd \text{ (for } Q \text{ and } K) + Nd \text{ (for } V) + 2N^2\text{(for P and S)} = 3Nd + 2N^2$$

2. **Write:**

   - Write S: N^2 elements.
   - Write softmax output P: N^2 elements.
   - Write output O: $N \times d$ elements.

   Total write elements:

$$N^2 + N^2 + Nd = 2N^2 + Nd$$

Thus, the total elements transferred are:

$$(3Nd + 2N^2) + (2N^2 + Nd) = 4N^2 + 4Nd$$

Multiplying by BPE, the total memory access is:

$$\boxed{\text{prefill\_memory} = (4N^2 + 4Nd) \times \text{BPE}}$$

For FP16, using BPE = 2, this becomes $(4N^2 + 4Nd) \times 2$ bytes.

(ii) Decode Attention (Generating One Token at a Time):

In decode mode, we generate one new token at a time. Let $Q \in \mathbb{R}^{1\times d}$ be the query for the new token, while $K, V \in \mathbb{R}^{N\times d}$ represent previously stored states.

FLOPs Calculation (Decode):

1. **Compute $\mathbf{S} = \mathbf{QK}^\top$:**
   - $S \in \mathbb{R}^{1\times N}$ requires a dot product for each of its N entries
   - Each dot product: $2d$ FLOPs
   - Total cost:
   $$\text{S\_flops} \approx 2\,N\,d$$
2. **Compute $\mathbf{P} = \text{softmax}(S)$:** This is over N elements, so $\alpha \times N$ FLOPs, typically small.
3. **Compute $\mathbf{O} = \mathbf{PV}$:**
   - $P \in \mathbb{R}^{1\times N}$ and $V \in \mathbb{R}^{N\times d}$ produce $O \in \mathbb{R}^{1\times d}$
   - Total cost:
   $$\text{PV\_flops} \approx 2\,N\,d$$

Thus, the total FLOPs(dominant) for decode attention are:

$$\boxed{\text{decode\_flops} \approx 2\,N\,d + 2\,N\,d = 4\,N\,d}$$

Memory Access Calculation (Decode): Read Q and K, write S, then read S for softmax, write P, then read P and V and write O

1. **Read:**
   - Q: $1 \times d$ elements
   - K: $N \times d$ elements
   - V: $N \times d$ elements
   - P: $(1 \times N)$
   - Reading S for softmax: $1 \times N$ elements

   Total read elements:
   $$d + Nd + Nd + 2N = d + 2Nd + 2N$$
2. **Write:**
   - Write S: $1 \times N$ elements
   - Write softmax output P: $1 \times N$ elements
   - Write output O: $1 \times d$ elements

   Total write elements:
   $$N + N + d = 2N + d$$

Thus, total elements transferred:

$$\big[d + 2Nd + 2N\big] + \big[2N + d\big] = 2d + 2Nd + 4N$$

Multiplying by BPE, the memory access is:

$$\boxed{\text{decode\_memory} = \Big(2d + 2Nd + 4N\Big) \times \text{BPE}}$$

(iii) Arithmetic Intensity (AI): Arithmetic Intensity (FLOPs per byte) is defined as:

$$\text{AI} = \frac{\text{Total FLOPs}}{\text{Total Memory Access (in bytes)}}$$

Thus, for prefill case:

$$\text{AI}_{\text{prefill}} = \frac{4\,N^2\,d}{(4N^2 + 4Nd)\,\text{BPE}}$$

Decode case:

$$\mathrm{AI}_{\text{decode}} = \frac{4\,N\,d}{\left(2d + 2Nd + 4N\right)\,\mathrm{BPE}}$$

Final Formulas

$$\begin{aligned}
\text{prefill_flops} &\approx 4\,N^2\,d \\
\text{prefill_memory} &\approx (4\,N^2 + 4\,N\,d)\,\mathrm{BPE} \\
\mathrm{AI}_{\text{prefill}} &= \frac{4\,N^2\,d}{(4\,N^2 + 4\,N\,d)\,\mathrm{BPE}} \\
\\
\text{decode_flops} &\approx 4\,N\,d \\
\text{decode_memory} &\approx \left[\,2\,d\,(N+1) + 4N\,\right] \times \mathrm{BPE} \\
\mathrm{AI}_{\text{decode}} &= \frac{4\,N\,d}{\left[2\,d\,(N+1) + 4N\right]\,\mathrm{BPE}}
\end{aligned}$$

1(d) Calculate arithmetic intensity using the formula above, given $N = 1024$ and $d = 128$. Plot the arithmetic intensity as context length (N) varies (for prefill, batch size 1) and as batch size varies (for decode). Crisply explain your observations and reason.

We compute arithmetic intensity (AI) for both **prefill** and **decode** attention under the following assumptions:

- Head dimension $d = 128$ (given)
- Context length N can vary from 256 to 8192 for the prefill plot
- For the decode plot, we fix $N = 1024$ and vary the batch size from 1 to 100
- Precision: FP16 (BPE = 2 bytes/element)
- Each multiply–add is counted as 2 FLOPs, so a dot product of length d is $2d$ FLOPs

Output of the program

```
----- Question 1(d) Arithmetic Intensity -----
Arithmetic Intensity (Prefill) for N=1024, d=128: 56.89 FLOPs/byte
Arithmetic Intensity (Decode) for N=1024, d=128: 0.98 FLOPs/byte
```

Prefill Attention (Processing N Tokens at Once):

$$\mathrm{AI}_{\text{prefill}} = \frac{4\,N^2\,d}{(4\,N^2 + 4\,N\,d)\,\mathrm{BPE}} = \frac{4 \times (1024)^2 \times 128}{\left(4 \times (1024)^2 + 4 \times 1024 \times 128\right) \times 2}$$

$$\approx 56.89 \text{ FLOPs/byte.}$$

Decode Attention (Generating One Token at a Time):

$$\mathrm{AI}_{\text{decode}} = \frac{4\,N\,d}{(2\,d + 2\,N\,d + 4\,N)\,\mathrm{BPE}} = \frac{4 \times 1024 \times 128}{\left(2 \times 128 + 2 \times 1024 \times 128 + 4 \times 1024\right) \times 2} \approx 0.984 \text{ FLOPs/byte}$$

Graphical representation

Prefill: Arithmetic Intensity vs. Context Length.

- The plot shows $\mathrm{AI}_{\text{prefill}}$ rising from about 42 FLOPs/byte at small N to nearly 64 FLOPs/byte at large N.
- As N grows larger than $d = 128$, the term $\frac{N}{N+d}$ approaches 1, so AI approaches $\frac{d}{\mathrm{BPE}} = \frac{128}{2} = 64$ FLOPs/byte.

- The curve flattens around $N \approx 4000$–8000, confirming the saturation near 64.

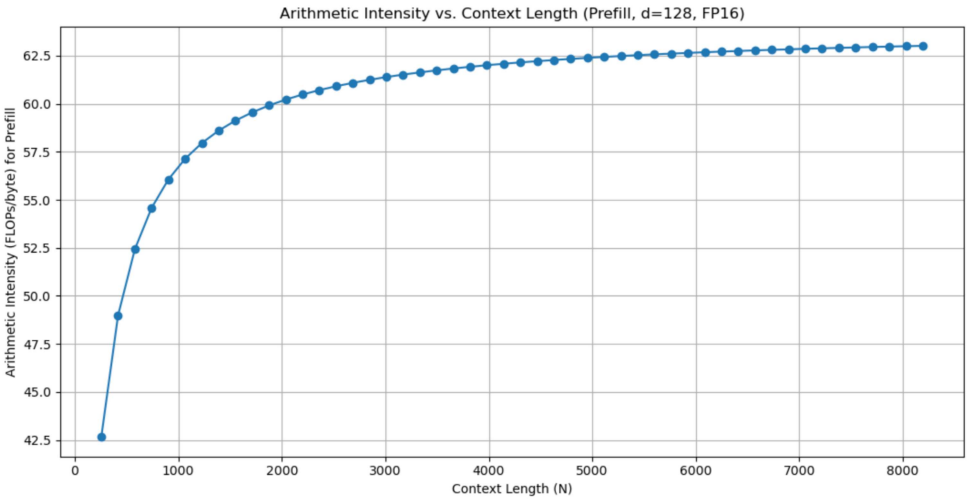

Figure 1: Prefill: arithmetic intensity vs context length

Decode: Arithmetic Intensity vs. Batch Size.

- The plot below keeps $N = 1024$ fixed and plots AI as batch size goes from 1 to 100.
- The line is effectively flat, around ≈ 0.98–1.0 FLOPs/byte, indicating a memory-bound regime.
- Even if more requests are processed concurrently, the ratio of FLOPs to memory bytes does not improve.

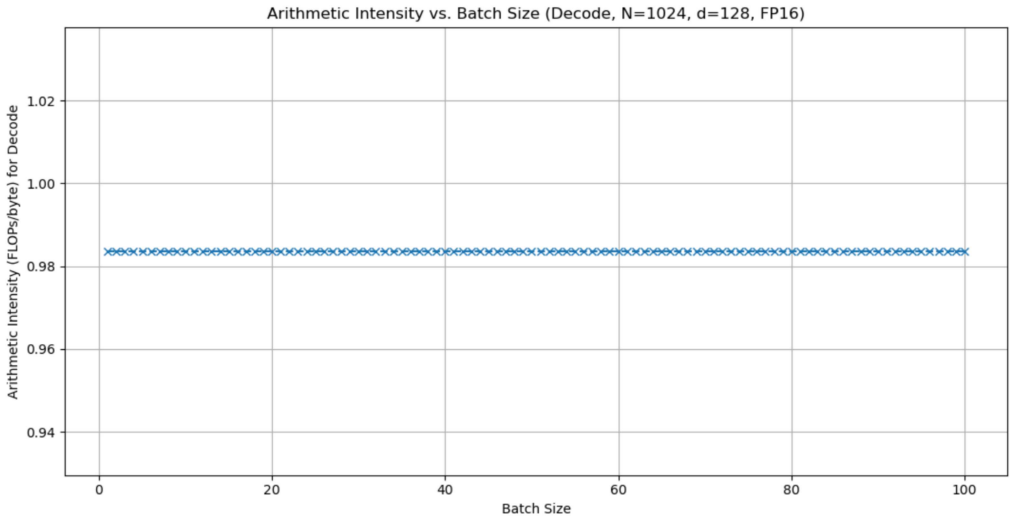

Figure 2: Decode: arithmetic intensity vs batch size

Observations and Reasoning

1. **Prefill Attention:**
   The arithmetic intensity (AI) for prefill attention is given by

$$\mathrm{AI}_{\text{prefill}} = \frac{4\,N^2\,d}{(4N^2 + 4N\,d)\,\mathrm{BPE}}$$

As the context length N increases, the term $\frac{N^2}{N^2+N\,d}$ approaches 1 when $N \gg d$. Therefore, AI starts from about 42 FLOPs/byte at small N and for very large N the AI saturates near

$$\frac{d}{\text{BPE}}$$

which in our case (with $d = 128$ and BPE $= 2$) is about 64 FLOPs/byte. This high AI implies that, for large contexts, the prefill attention computation shifts toward being **compute-bound** rather than limited by memory bandwidth.

2. **Decode Attention:**
   For decode attention, the arithmetic intensity is expressed as

$$\text{AI}_{\text{decode}} = \frac{4\,N\,d}{\big(2\,d + 2\,N\,d + 4\,N\big)\,\text{BPE}}$$

   In this mode, even when we include batch size B the B cancels out in numerator and denominator. With typical values (e.g., $N = 1024$ and $d = 128$), the resulting AI is very low (around 1 FLOP/byte). This low AI indicates that decode attention is strongly **memory-bound**; that is, most of the time is spent transferring data (e.g. loading all keys and values) rather than performing arithmetic operations.

3. **Effect of Batch Size:**
   In the decode phase, increasing the batch size B scales both the total number of FLOPs and the total memory accesses linearly, so the arithmetic intensity (FLOPs per byte) remains constant. This observation is critical because it shows that simply processing more requests concurrently does not improve the compute-to-memory ratio in the decode phase.

Conclusion:
For large context lengths, prefill attention can achieve a high arithmetic intensity (up to around 64 FLOPs/byte in our example), suggesting that its performance could be limited more by compute capability. In contrast, decode attention shows a very low arithmetic intensity (around 1 FLOP/byte), making it inherently memory-bound, which explains why optimizations in data movement (rather than computation) are important for improving decode performance.

PROBLEM 2

Implement a simple 2 layer decoder-only transformer model in Python, using the provided skeleton code.

File name

```
transformer_skeleton.py
```

We use an absolute tolerance of 1×10^{-2} (atol=1e-2) to accommodate minor floating-point precision differences during evaluation.

2(a) Implement all the TODO sections in the provided code to create a functioning transformer model with the ability to autoregressively generate 'n' tokens.

Implementation Details

1. class SingleHeadAttention

- *Projection Layers:* Implemented four linear projections (W_q, W_k, W_v, W_o) that map the input from d_{model} to d_{model}, with the query, key, and value layers set without bias.
- *Scaled Dot-Product:* Computed attention scores using

$$\text{scores} = \frac{QK^T}{\sqrt{d_k}}$$

  ensuring that the scaling by $\sqrt{d_k}$ prevents gradient instability.
- *Causal Masking:* Applied an upper-triangular mask using `torch.triu(..., diagonal=1)` to the scores to block future tokens, enforcing autoregressive behavior.
- *KV Caching:* Added logic to check if caching is enabled. When use_cache is true, new key (K) and value (V) tensors are concatenated with previously cached ones.
- Used softmax over the last dimension to get attention weights, then computed

$$\mathbf{C} = \text{softmax}(\text{scores}) \times \mathbf{V}$$

- Finally projected the context back to d_{model} dimensions using $\mathbf{W_o}$.

The self-attention mechanism allows each token to attend only to prior tokens, which is mathematically expressed by the scaled dot-product formula. Causal masking ensures that the model adheres to the autoregressive paradigm during both training and inference.

2. class FeedForward

- *Two Linear Layers:* Designed as a position-wise feed-forward network that first projects the input from d_{model} to a higher dimensional space d_{ff} and then back to d_{model}.

$$\text{fc1} : d_{\text{model}} \to d_{\text{ff}}, \quad \text{fc2} : d_{\text{ff}} \to d_{\text{model}}$$

- *Activation Function:* A ReLU activation is applied between the two linear transformations:

$$\text{FFN}(x) = \text{Linear}_2\big(\text{ReLU}(\text{Linear}_1(x))\big)$$

This structure introduces non-linearity and increases the model's capacity to capture complex relationships. The position-wise feed-forward network is key to transforming the attention outputs into higher-level representations.

3. class DecoderLayer

- Composed of:
  - A `SingleHeadAttention` layer
  - A `FeedForward` layer
  - Two `LayerNorm`s and a dropout
- Follows the pattern:

$$x \to \text{LayerNorm} \to \text{Self-Attention} \to \text{Dropout} \to \text{Residual connection}$$

$$x \to \text{LayerNorm} \to \text{FeedForward} \to \text{Dropout} \to \text{Residual connection}$$

- *Layer Normalization:* Implemented normalization before both the attention and the feed-forward submodules to ensure stable distributions.
- *Residual Connections and Dropout:* Integrated skip connections after both the attention and feed-forward blocks:

$$x \leftarrow x + \text{Dropout}(\text{SubLayer}(x))$$

  which facilitate gradient flow and mitigate overfitting.
- *KV Caching Integration:* Ensured that the attention module passes updated key-value pairs (K, V) to subsequent layers when caching is enabled.

Residual connections and layer normalization help stabilize training and allow deeper networks. Dropout mitigates overfitting. Passing KV pairs forward supports efficient decoding.

4. class DecoderOnlyTransformer

The implementation consists of two primary methods: the `forward` method and the `generate` method.

`forward` **Method:**

- **Embedding Layers:**
  - `token_embedding` converts input token IDs to vectors of size d_model.
  - `pos_embedding` provides positional embeddings of size max_seq_len.

  Their sum is computed as:

$$x = \text{token\_embedding}(\text{input\_ids}) + \text{pos\_embedding}(\text{positions})$$

  where positions is generated based on the input sequence length.
- **Stacking Decoder Layers:** The embedded input is passed sequentially through each decoder layer in a loop. For each layer, the method passes:
  - past_kv$[i]$ if KV caching is enabled, or
  - `None` if caching is not used.
- **Final Normalization and Projection:** After processing through all layers, a final layer normalization is applied. The resulting hidden states are then projected via the `output_projection` layer to produce output logits over the vocabulary.

`generate` **Method:** This method implements autoregressive token generation as follows-

- **Iterative Decoding:** The generation starts with the initial input sequence. For each new token, if `use_cache` = True, only the last generated token is used as input along with the cached keys and values.
- **Temperature Scaling:** The logits corresponding to the last token are scaled by a temperature T:

$$\text{logits}_{\text{scaled}} = \frac{\text{logits}}{T}$$

- **Sampling and Sequence Update:** The scaled logits are converted into probabilities via softmax. The next token is then sampled using `torch.multinomial` and appended to the generated sequence. The KV cache is updated for subsequent steps.

Results

```
Successfully loaded model state dictionary from model_state_dict.pt
Weights loaded from model_state_dict.pt
Weights match for layer 0 Wq: True
Weights match for layer 1 Wk: True
Test cases loaded from test_cases.json
First test case:
dict_keys(['input_ids', 'expected_logits_no_cache', 'expected_logits_with_cache', 'expected_logits_sequential'])
input_ids torch.Size([1, 11])
expected_logits_no_cache torch.Size([1, 11, 1000])
expected_logits_with_cache torch.Size([1, 11, 1000])
expected_logits_sequential torch.Size([1, 11, 1000])

Evaluating model...
Evaluation Results:
  All tests passed: True
  Pass rate: 100.00% (10/10)

Benchmarking performance...
Performance Results:
  Without KV cache: 0.0458 seconds
  With KV cache: 0.0208 seconds
  Speedup: 2.20x
```

Figure 3: Console output for Q2a

Observations

- Residual connections and normalization stabilize training and allow deeper networks.
- Causal masking ensures the model cannot attend to future tokens.
- Temperature scaling influences the randomness of sampling: a higher temperature yields more diverse outputs.

2(b) Correct implementation of the model with logits generated after the initial prompt processing matching the expected values for a given model weight file. Run the file with mode - -evaluate

We verify the correct implementation of the model. We compare the logits produced by the model against the expected logits provided in the test cases.This test ensures that the model's forward pass is implemented correctly and produces consistent outputs when running with the pre-trained weights.

Command

To execute this test, we run the file with mode `--evaluate` using the following command:

```
python transformer_skeleton.py --mode evaluate
```

Results

When executed in `--evaluate` mode, the console output confirms that the logits generated after processing the initial prompt match the expected values. The sample console output below shows that all 10 test cases passed at 100% accuracy, demonstrating the correctness of the model:

```
Successfully loaded model state dictionary from model_state_dict.pt
Test cases loaded from test_cases.json
Evaluation Results:
  Num test cases: 10
  All tests passed: True
  Pass rate: 100.00% (10/10)
```

Figure 4: Console output for Q2(b) showing 10/10 tests passed at 100% accuracy.

All tests passed: True
Pass rate: 100.00%
This confirms that the model implementation is correct, as the logits match the expected values for the given weight file.

2(c) Implement KV caching to cache the generated KV from previous iterations in every subsequent iter instead of naively processing the entire sequence in each step. Validate that the logits with and without KV caching match. Run the file with mode - -kv_evaluate

We implement KV caching so that during token generation, the model caches the previously computed keys and values (KV) from earlier iterations. This allows each subsequent iteration to process only the most recent token rather than the entire sequence, significantly improving computational efficiency.

To validate the correctness of the caching mechanism, the logits produced with KV caching are compared against those generated without caching. The test harness ensures that the two approaches yield equivalent results within a specified numerical tolerance.

In the code, the key changes for KV caching include:

- In the `SingleHeadAttention.forward` method, an additional parameter `use_cache` is introduced. When enabled and if a previous KV cache (`past_kv`) exists, the new key (k) and value (v) projections are concatenated with the cached ones:

$$k = \text{torch.cat}([\texttt{past\_kv}[0], k], \text{dim} = 1)$$

$$v = \text{torch.cat}([\texttt{past\_kv}[1], v], \text{dim} = 1)$$

  This ensures that the model reuses previously computed information rather than recomputing attention over the entire sequence.

- In the `DecoderLayer.forward` method, the `use_cache` flag is passed along to the attention module. The layer then returns the new KV pairs along with the output, which are accumulated for use in subsequent layers or iterations.

- In both the `DecoderOnlyTransformer.forward` and `generate` functions, the `use_cache` flag is managed to enable or disable KV caching. In the `generate` function, if caching is enabled, only the last token (rather than the entire generated sequence) is processed in each iteration, while the KV cache is updated accordingly.

- Additionally, the code ensures that when `use_cache` is enabled, the concatenated KV pairs are correctly passed through all layers, preserving the context from previous iterations.

Command

To execute this test, we run the file with mode `--kv_evaluate` using the following command:

```
python transformer_skeleton.py --mode kv_evaluate
```

Results

When executed in `--kv_evaluate` mode, the console output confirms that all test cases pass, indicating that the logits generated with and without KV caching match.

```
Successfully loaded model state dictionary from model_state_dict.pt
Test cases loaded from test_cases.json
Evaluation Results:
  Num test cases: 10
  All tests passed: True
  Pass rate: 100.00% (10/10)
```

Figure 5: Console output for Q2c showing 10/10 tests passed at 100% accuracy.

Pass rate: 100.00%

This confirms that the KV caching implementation is correct and that the logits remain consistent regardless of whether caching is used.

2(d) Vary the sequence length from [10, 50, 100, 500, 1000] and compare the performance (latency to generate 'n' tokens given a prompt) with and without KV caching of prior tokens. Plot a graph that shows latency with and without KV as a function of sequence length. Run the file with mode - - benchmark

In this part, we benchmark how quickly the model generates new tokens for prompts of varying lengths. By default, we measure the time taken to generate 20 new tokens (configurable in the code). We repeat the benchmark with and without KV caching and compute the speedup as

$$\text{speedup} = \frac{\text{time\_without\_cache}}{\text{time\_with\_cache}}$$

Command

To run the benchmark, we execute:

```
python transformer_skeleton.py --mode benchmark
```

This command generates random inputs of lengths 10, 50, 100, 500, and 1000, then measures the average latency for generating new tokens. It plots the latency versus sequence length and prints out the measured speedups.

Code Changes for Benchmarking

- A `benchmark_performance` function is used to measure the time taken to generate new tokens. It calls the `model.generate` method multiple times and averages the measured wall-clock durations.
- In the `main` function, branch `elif args.mode == 'benchmark':` handles the benchmark logic:
  - It creates random input sequences of specified lengths.
  - It runs `model.generate` with `use_cache=False` and `use_cache=True`.
  - It records the timing results, computes speedups, and plots the final latency vs. sequence length.
- The code checks whether a given sequence length exceeds `max_seq_len`; if so, that length is skipped to avoid errors. Output is saved in benchamark_results.json file
- Since this 'benchmark' was below "DO NOT modify code below this line" in the skeleton code provided, the changes made by me have been commented from line 448 - 482 to avoid modifying the existing code. These code changes are not required as we only wanted to see the plot of output.
- plot.py file reads benchamark_results.json and plots the graph

Assumptions and Parameters

- **Model dimensions:** $d_{\text{model}} = 50$, $d_{\text{ff}} = 100$, num_layers = 2, and vocab_size = 1000.
- **Max sequence length:** max_seq_len = 128. Hence, sequence lengths 500 and 1000 exceed the model's maximum sequence length and are skipped.
- **Latency measurement:** The script performs three timed runs (by default) for each setting and averages the results.

```
Successfully loaded model state dictionary from model_state_dict.pt
Benchmarking...
Results for default:
  Without KV cache: 0.0334 seconds
  With KV cache: 0.0262 seconds
  Speedup: 1.27x
Sequence Length: 10
  Without KV cache: 0.0339 seconds
  With KV cache: 0.0274 seconds
  Speedup: 1.24x
Sequence Length: 50
  Without KV cache: 0.0406 seconds
  With KV cache: 0.0315 seconds
  Speedup: 1.29x
Sequence Length: 100
  Without KV cache: 0.0661 seconds
  With KV cache: 0.0365 seconds
  Speedup: 1.81x
Skipping seq_len 500 as it exceeds max_seq_len (128)
Skipping seq_len 1000 as it exceeds max_seq_len (128)
```

Figure 6: Console output for various sequence lengths showing the speedups with KV caching.

Results and Graph

Below is a sample console output and corresponding latency plot:

- **Overhead at small lengths:** For shorter sequences (e.g., 10, 50), the overhead of caching is not large, yielding modest speedups around 1.2–1.3×.
- **Gains at larger lengths:** As sequence length grows (up to the maximum of 128 in our setup), KV caching avoids repeated computation over the entire sequence each step, achieving speedups of $\sim 1.8\times$.
- **Skipping 500 and 1000:** Since the model is constrained by max_seq_len = 128, prompts of length 500 and 1000 are not processed.

Analysis of the Graph

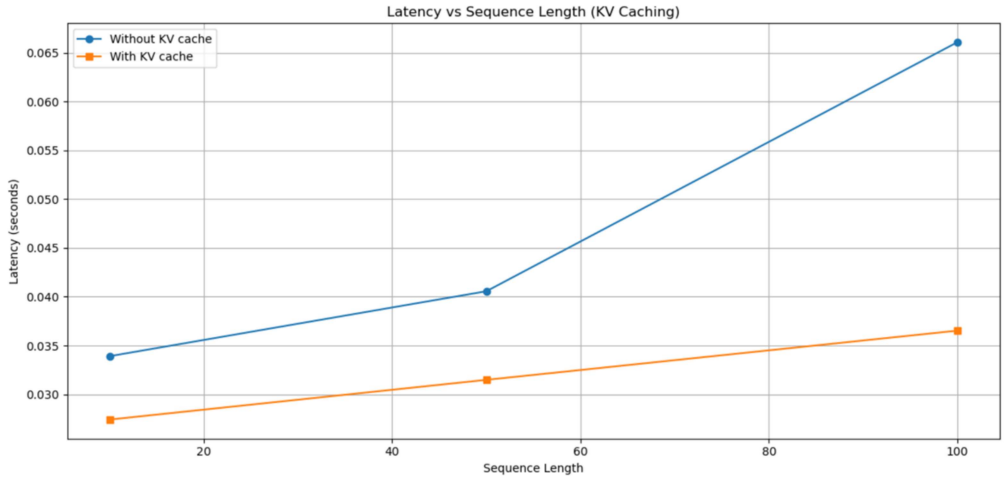

Figure 7: Latency vs. Sequence Length with and without KV caching.

The latency curves clearly show that without KV caching (blue line), the time required grows more steeply as the sequence length increases. By contrast, with KV caching (orange line), the growth in latency is more gradual, reflecting the efficiency gained by reusing previously computed keys and values

rather than reprocessing the entire history. This demonstrates that as sequence length grows, KV caching can substantially improve generation speed.

Overall, these benchmarks confirm that KV caching becomes increasingly beneficial as sequence length grows, reducing the time per generation step by leveraging previously computed attention keys and values.