CS5302-PA2 Instruction Manual

Muhammad Khaquan
Department of Computer Science
Lahore University of Management Sciences
25100095@lums.edu.pk

Omer Tafveez
Department of Economics
Lahore University of Management Sciences
25020254@lums.edu.pk

Abstract

This manual serves as a brief introduction to the SmolLM architecture [Allal et al., 2024] that you will be implementing.

Each section describes the components that your are required to implement. Please follow the instructions carefully and in case of any confusion, contact either Omer Tafveez or Muhammad Khaquan via slack.

1 Introduction

The SmolLM architecture [Allal et al., 2024] is based on the highly successful Llama3 architecture. The major building blocks of this architecture can be seen in figure 1.

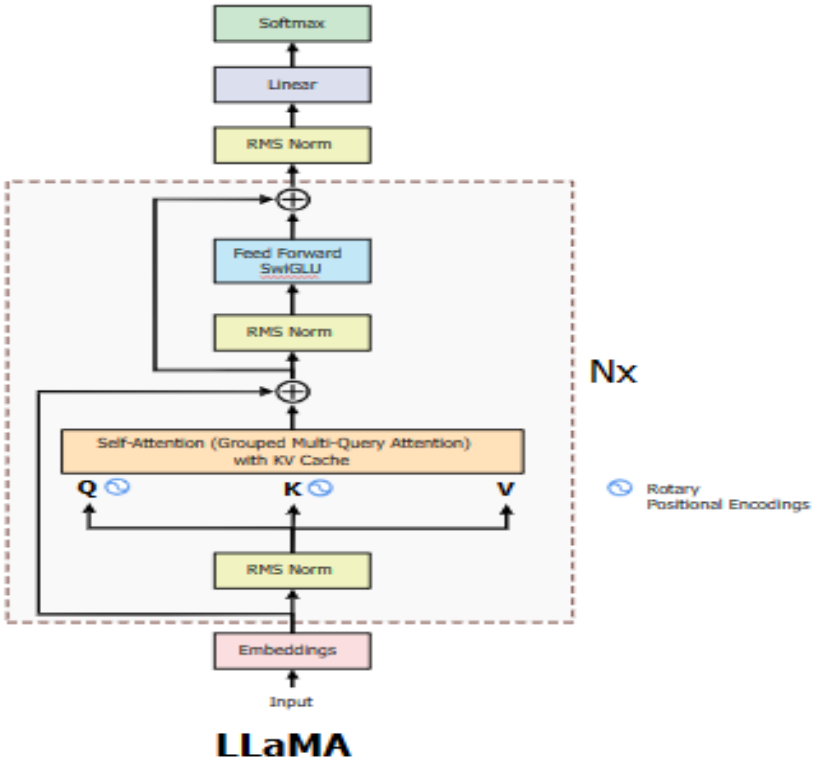

Figure 1: An overview of SmolLM/Llama3 architecture.

We have provided you with some boiler plate code. You are required to implement the following components from figure 1 (in no specific order):

1. **RMS Norm** [Zhang and Sennrich, 2019] in the *layers.py* file.
2. **Rotary Positional Embeddings** (RoPE) [Su et al., 2023] in the *attention.py* file.
3. **Grouped Query Attention** [Ainslie et al., 2023] in the *attention.py* file.
4. **Gated Feed Forward Network** (i.e., SwiGLU FFN) [Shazeer, 2020] in the *layers.py* file.
5. **Decoder block** in *layers.py* file.
6. **SmolLM backcbone** (class SmolModel) in *model.py* file.
7. **SmolLM** language modelling head (class SmolLM) in *model.py* file.

The subsequent sections contain a concise overview of each of these components. Read these sections carefully and then start your implementation!

2 Root Mean Squared Normalization (RMS Norm)

Recall that the normalization of any matrix or vector is given by

$$\text{y} = \frac{x - E[X]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

where γ and β are learnable parameters that allow the model to amplify the scale of each feature or apply a translation to the feature according to the needs of the loss function. In batch normalization, we normalize by columns (features), and in layer normalization, we normalize by rows (data items).

The paper [Zhang and Sennrich, 2019] on RMS Norm was proposed as an alternative to Layer Normalization. The author claimed that we do not need the mean (or to centralize the values around the mean) to get the same affect as Layer Normalization. Therefore, we use root-mean-squared to re-scale the vector instead of re-centering the vector as follows:

$$\text{x} = \frac{x}{RMS(x)} * \gamma$$

3 Rotary Positional Embeddings (RoPE)

Recall that for each word in the sequence, an embedding is computed with another embedding of the same dimension D_E added. We obtain this using a sinusoidal function. This is known as the **positional embedding**, which only determines the word's position. Since it is the word's position in the sequence, it is only computed once.
To understand RoPE, we need to understand **relative positional encoding**. Relative positional encoding deals with two tokens at a time and is *only* computed during attention. Since attention captures the *intensity* of how much two words are related to each other, relative positional encodings tell the attention mechanism the distance between the two words involved in it. So given two tokens, we create a vector that represents their distance given as follows

$$\text{absolute positional encoding}_i = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}}$$

$$\text{relative positional encoding}_{i,j} = \frac{(x_i W^Q)(x_j W^K + a^K_{i,j})^T}{\sqrt{d_z}}$$

where **a** is the distance between the two vectors.

Rotary Positional Encoding are somewhere between absolute and relative encoding since each token gets its own embedding, and an attention mechanism is applied to it. In RoPE, we aim to find an inner product of the query and key that depends on the embedding and the

relative distance between them. RoPE essentially gets away from using positional embeddings; instead, it encodes position into the query and key embeddings before computing the attention weights.

Rotary Positional Embedding (RoPE) conceptually treats each token's embedding, for instance $\left(W^Q x_m\right)$ for a query token or $\left(W^K x_n\right)$ for a key token, as a vector in a 2D plane, where m and n represent their respective position indices. The main steps are:

- **Coordinate Conversion:**
  Split the embedding dimension into pairs of coordinates (or interpret them as complex numbers). This effectively uses polar coordinates via Euler's Formula.
- **Rotation by Position:**
  Each pair is then multiplied by a rotation matrix M that depends on the position index (i.e., m or n). In practice, we do not literally multiply by a dense matrix—because M can be expressed with sines and cosines; it is more efficient to apply the rotation directly rather than building a large sparse matrix.

For example, let $\mathbf{p}$ be the embedding at position m in the query vector $\mathbf{Q}$, and let $\mathbf{r}$ be the embedding at position n in the key vector $\mathbf{K}$. Both $\mathbf{p}$ and $\mathbf{r}$ are rotated by some angle θ (which depends on their positions), producing $\mathbf{p}'$ and $\mathbf{r}'$. When the attention mechanism takes the dot product $\mathbf{p}' \cdot \mathbf{r}'$, it inherently captures the relative distance $\left(m - n\right)$.

By rotating each 2D "chunk" of the embedding, the model can learn *how far apart* two tokens are in the sequence. Although we do not derive every equation here, a solid grasp of the underlying math helps explain why RoPE is effective and how to implement it properly.
Hint: Since the rotation matrix M is essentially diagonal and sparse, it is much more efficient to apply the sines and cosines directly rather than performing a full matrix multiplication.

Read Huggingface-RoPE to understand the implementation and design behind RoPE.

4 Grouped Query Attention (GQA)

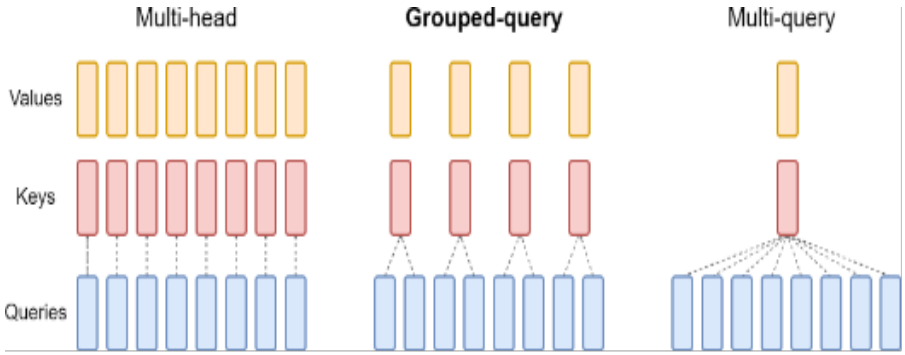

Figure 2: An overview of Grouped Query Attention.

Grouped Query Attention (GQA) Ainslie et al. [2023] is an efficient variant of Multi-Head Attention (MHA) designed to reduce computational and memory costs while maintaining model performance. In GQA, *query heads are grouped*, and *each group shares a single key/value (KV) head*. This reduces the number of unique KV heads compared to MHA, where each query head has its own KV head. GQA strikes a balance between the efficiency of Multi-Query Attention (MQA) Shazeer [2019], which uses a single KV head for all queries, and the expressiveness of MHA. You can see a comparitive overview of all three attention schemes in figure 2.

GQA enjoys several advantages compared to standard MHA:

- **Memory Efficiency:** Fewer KV heads reduce the size of the KV cache, which is critical for autoregressive decoding in large models.

- **Compute Efficiency:** Fewer projections for KV heads lower the computational cost of attention operations.
- **Performance Retention:** GQA maintains better model quality compared to MQA, which sacrifices performance for efficiency.

While implementing GQA, you need to consider the following factors:

- **Query Projection:** Given that your self-attention block has n_heads, then your queries will be projected to n_heads too. So the shape of the query tensor must be $(batch\_size, n\_heads, sequence\_len, emb\_dim)$.
- **Key & Value Projections:** Given that you have kv_heads key and value heads, then your keys and values will be projected to only $kv\_heads \leq n\_heads$. The final shape of key and value tensor must be $(batch\_size, kv\_heads, sequence\_len, emb\_dim)$.
- **Repetition of KV heads:** For us to be able to carry out the scaled dot product used in self-attention, the shape of the Key and Value tensors must match the shape of the Query tensor along the head dimension. This means we will need to repeat the same KV head for each group $int(\frac{n\_heads}{kv\_heads})$ times. For the example in figure 2, we have 4 groups each with 1 unique KV head and 2 Query heads. So, for this example, we will need to repeat each KV head twice for each group.

5 Gated Feed-Forward Network (SwiGLU FFN)

This is the feed-forward network of the transformers module. The original Transformers paper suggested the ReLU non-linearity for the MLP network given as follows

$$\text{FFN}(x) = max(0, xW_1 + b1)W_2 + b_2$$

SwiGLU is the current choice for most architectures. To understand the design of SwiGLU, here is the blog you can refer to SwiGLU Blog

To understand the implementation, let's look at the formula

$$\text{Swish}_\beta(x) = x\sigma(\beta x)$$

$$\text{FFN}_{\text{SwiGLU}}(x, W, V, W_2) = (\text{Swish}_1(xW) \otimes xV)W_2$$

Thus, instead of applying ReLU on the linear transformation xW, we have two linear transformation - xW and xV, and we then apply swish on the first one.

6 Decoder block

This is the part of the architecture where we combine all the components above into one class. Specifically, the outlined stack in figure 1 represents the decoder block.

You will token embeddings as input into this decoder block. You will pass these embeddings successively into Nx decoder blocks until you finally recieve highly informative, semantically rich embeddings at the output. These embeddings are then handed over to the language modelling head which projects these embeddings into the vocabulary space.

7 SmolLM backcbone

This is the part of the model where you store your decoder layers. You run the input token embeddings into the decoder layer successively and return the enriched embeddings at the ouput.

8 SmolLM Language Modelling Head

This is the language modelling head of the SmolLM architecture. One interesting caveat here is this:

> In SmolLM, we use a weight sharing scheme. The embedding projection (the first block which the input is passed to in figure 1) and the language modelling projection use the same weight matrix.

The embedding projection is of shape $(vocab\_size, emb\_dim)$ and it maps your input tokens (i.e., raw integers) into embedding vectors. On the other hand, the language modelling head does the opposite. It takes as input the enriched embedding vectors returned by the successive decoder blocks and maps them onto the vocabulary space (i.e., the token space). It has shape $(emb\_dim, vocab\_size)$. So, as you can see, the shape of the weight matrix of the language modelling head is tranpose of the shape of the embedding weight matrix. We can use this transpose relation to "tie" the two weights into one during training.

This weight sharing scheme is a nifty trick that enables small language models to save on parameters. Usually, the embedding projection and the language modelling projection add a large number of parameters to the model (for example, if we have a $vocab\_size = 48,000$ and $emb\_dim = 500$, these projections would introduce $24M$ parameters each. This is $48M$ parameters in total i.e., $\frac{1}{3}$ the number of parameters in the SmolLM model! By using weight sharing we halve this parameter count to $24M$).

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL https://arxiv.org/abs/2305.13245.

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Leandro von Werra, and Thomas Wolf. Smollm - blazingly fast and remarkably powerful, 2024.

Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019. URL https://arxiv.org/abs/1911.02150.

Noam Shazeer. Glu variants improve transformer, 2020. URL https://arxiv.org/abs/2002.05202.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Biao Zhang and Rico Sennrich. Root mean square layer normalization, 2019. URL https://arxiv.org/abs/1910.07467.