Large language model Architectures Explained With Llama Implemented from Scratch

Wenhan Gao
Department of Applied Mathematics and Statistics, Stony Brook University, Stony Brook, NY 11794, USA
wenhan.gao@stonybrook.edu

1 Overview

In this write-up, we dive into the foundations of large language models (LLMs), such as Llama and GPT. Specifically, we will:

- Break down each component of a modern LLM architecture.
- Explain concepts related to architectures such as **KV caching**, **quantization**, **mixture of experts**, etc..

We will implement Llama 3 (Grattafiori et al., 2024), specifically Llama 3.2-1B as a representative example, though the implementation applies to all Llama 3 models. We present the LLM concepts in the order in which they appear in the Llama 3 architecture. The code is a modified and simplified version of the Llama Original Implementation for demonstration purposes. The code for the simplified version can be accessed Here. Follow instructions below to download the weights.

Llama Initialization

Firstly, the model weights and configuration files should be downloaded. The files can be downloaded through Hugging Face or Llama Official Website.

We will be using the original Llama files, not the Hugging Face Transformers format. For Hugging Face, make sure that the original files are downloaded.

```
huggingface-cli download meta-llama/Llama-3.2-1B --include "original/*" --local-dir Llama-3.2-1B
```

Llama Implementation

```python
from pathlib import Path
import tiktoken
from tiktoken.load import load_tiktoken_bpe
import torch
import json

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

with open("original/params.json", "r") as f:
    config = json.load(f)

dim = config["dim"]
n_layers = config["n_layers"]
n_heads = config["n_heads"]
n_kv_heads = config["n_kv_heads"]
vocab_size = config["vocab_size"]
multiple_of = config["multiple_of"]
ffn_dim_multiplier = config["ffn_dim_multiplier"]
norm_eps = config["norm_eps"]
rope_theta = torch.tensor(config["rope_theta"])

print(config) # Check Config

model = torch.load("original/consolidated.00.pth") # Load the model weights
```

Python Output

```
{"dim": 2048,
 "ffn_dim_multiplier": 1.5,
 "multiple_of": 256,
 "n_heads": 32,
 "n_kv_heads": 8,
 "n_layers": 16,
 "norm_eps": 1e-05,
 "rope_theta": 500000.0,
 "use_scaled_rope": True,
 "vocab_size": 128256}
```

For clarity, we will use the following symbols consistently throughout the write-up. Despite the Python convention, we assume that indexing starts at 1.

Table 1: Table of Notation

Symbol	Meaning
d	Model dimension (embedding and hidden state feature dimensions)
$\mathbb{V}$ and V	Vocabulary set and vocabulary size, respectively
N	Number of input tokens
L	Number of layers
Superscript ^{l} and subscript $_i$	Denotes the l-th layer and the i-th token or feature vector, respectively
T or $T_{1:N} := \{t_1, t_2, \dots t_N\} \in \mathbb{V}^N$	Input tokens of length N
E or $E_{1:N} := \{e_1, e_2, \dots e_N\} \in \mathbb{R}^{N \times d}$	Input token embedding of length N
X^l or $X^l_{1:N} := \{x_1, x_2, \dots x_N\} \in \mathbb{R}^{N \times d}$	Hidden state feature of length N after the l-th Transformer layer

2 LLM Architectures

2.1 Input Text

The input text is the user's input raw text (prompt). Oftentimes, it is enhanced with system prompts, including prompt engineering strategies (e.g. "You are a helpful assistant"), persistent user information, chat histories, etc.

Llama Implementation

```
prompt = "The capital of France is"  # Hopefully, it will predict Paris
```

2.2 Tokenizer

The input text (including spaces, punctuations, emojis, etc.) is tokenized into discrete units using a predefined *tokenizer*. These units, known as tokens, can be as small as individual characters or as large as entire words or sub-words, depending on the tokenizer's design.

For example, the sentence

"The capital of France is"

is tokenized by the GPT-4o tokenizer as the following (tokens can include spaces):

The	capital	of	France	is
791	6864	315	9822	374

There are various choices of tokenizer, but the most common choice is a variant of Byte Pair Encoding (BPE). For example, given a training corpus, BPE initializes the tokens as individual Unicode characters, and then repeatedly merges the most frequent adjacent token pairs to build a vocabulary of tokens or sub-word units until a certain *vocabulary size*, V, is reached. There are also special tokens such as the beginning of sequence (<BOS>) and end of turn (<EOT>). The readers are encouraged to perform a Google search or just ask an LLM about the details and to examine some examples. Each token is assigned a unique integer ID in the vocabulary and is represented as such in the pipeline. We will denote the input tokens by $T_{1:N} := \{t_1, t_2, \dots t_N\}$, where t_i is the i-th token, and N is the length.

Llama Implementation

```
# We will not implement the tokenizer. Directly load the Llama tokenizer.
tokenizer_path = "original/tokenizer.model"
special_tokens = [
            "<|begin_of_text|>",
            "<|end_of_text|>",
            "<|reserved_special_token_0|>",
            "<|reserved_special_token_1|>",
            "<|reserved_special_token_2|>",
            "<|reserved_special_token_3|>",
            "<|start_header_id|>",
            "<|end_header_id|>",
            "<|reserved_special_token_4|>",
            "<|eot_id|>",  # end of turn
        ] + [f"<|reserved_special_token_{i}|>" for i in range(5, 256 - 5)]
mergeable_ranks = load_tiktoken_bpe(tokenizer_path)
tokenizer = tiktoken.Encoding(
    name=Path(tokenizer_path).name,
    pat_str=r"(?i:’s|’t|’re|’ve|’m|’ll|’d)|[^\r\n\p{L}\p{N}]?\p{L}+|\p{N}{1,3}| ?[^\s\p{L}\p{N}]+[\r\n]*|\
        ↪ s*[\r\n]+|\s+(?!\S)|\s+",
    mergeable_ranks=mergeable_ranks,
    special_tokens={token: len(mergeable_ranks) + i for i, token in enumerate(special_tokens)},
)
assert tokenizer.decode(tokenizer.encode(prompt)) == prompt # check if tokenizer works

tokens = [128000] + tokenizer.encode(prompt) # 128000 is the <|begin_of_text|> token
tokens = torch.tensor(tokens).to(device)

# Check the  tokens of the input prompt
print([tokenizer.decode([token.item()]) for token in tokens])
print(tokens)
print("Token Shape:", tokens.shape)
```

Python Output

```
["<|begin_of_text|>", "The", " capital", " of", " France", " is"]

tensor([128000,    791,   6864,    315,   9822,    374], device=’cuda:0’)

Token Shape: torch.Size([6])
```

2.3 Embedding

Given a tokenized input sequence $T_{1:N}$, discrete tokens t_i are mapped to a continuous vector space by a *token embedding* matrix TokenEmbed $\in \mathbb{R}^{V \times d}$, where V is the vocabulary size and d is the embedding dimension. Usually, there is a fixed model dimension for embedding and hidden state features in modern LLMs.

We denote the embedding of the sequence by $E_{1:N} := \{e_1, e_2, \ldots, e_N\} \in \mathbb{R}^{N \times d}$ with each token embedding $e_i =$ TokenEmbed (t_i). These embeddings will be learned during pretraining. They capture both the syntactic and semantic properties of tokens.

Llama Implementation

```
# nn.Embedding() is similar to nn.Parameters(), but  with efficient look up.
embedding_layer = torch.nn.Embedding(vocab_size, dim).to(device) # Initialize embedding layer
embedding_layer.weight.data.copy_(model["tok_embeddings.weight"]) # Load weights
token_embeddings = embedding_layer(tokens).to(dtype=torch.bfloat16) # Look up the table for embeddings
print("Token Embedding Shape:", token_embeddings.shape)
```

Python Output

```
Token Embedding Shape: torch.Size([6, 2048]) # 6 is the token length and 2048 is the model dimension
```

In some architectures, positional embeddings may be added to incorporate information about the position of each token within the sequence:

$$e_i = \text{ TokenEmbed }(t_i) + \text{ PosEmbed }(i),$$

where the positional embedding depends only on the position in the sequence. It can be implemented as a deterministic function, such as sinusoidal embedding in the original transformer Vaswani et al. (2017) or as a learned lookup table similar to token embedding. **However, modern architectures typically do not apply positional embeddings directly to the input embeddings. Instead, they incorporate positional encoding within the self-attention mechanism.**

2.4 Transformer Layers

We give an overview of the transformer layers first and then explain each component. Each transformer layer consists mainly of normalization, residual connection, attention mechanism, positional embeddings, and feedforward networks.

Specifically, LLaMA follows a Pre-Norm residual structure with RMS normalization. The computations for the l-th layer are given by:

$$\begin{aligned} H^l &= X^{l-1} + \text{MultiHeadAttention}\big(\text{RMSNorm}(X^{l-1})\big) \\ X^l &= H^l + \text{FeedForward}\big(\text{RMSNorm}(H^l)\big), \end{aligned} \tag{2.1}$$

with $X^0 = E$. The inputs and outputs are of the same shape $\mathbb{R}^{N\times d}$.

2.4.1 Normalization

It is common practice in modern Large Language Models (LLMs) to apply some form of normalization. Most LLMs typically adopt either layer normalization (LayerNorm) or root mean square normalization (RMSNorm).

LayerNorm. Given an input sequence of N features $X = \{x_1, x_2, \ldots, x_N\} \in \mathbb{R}^{N\times d}$, the LayerNorm operator normalizes each input vector across its feature dimension by subtracting its mean and dividing by its standard deviation, then applies learned scale and shift parameters:

$$X := \text{LayerNorm}(X_{1:N}), \text{ with } x_i := \gamma \cdot \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta, \quad \mu_i = \frac{1}{d}\sum_{j=1}^{d} x_{ij}, \quad \sigma_i^2 = \frac{1}{d}\sum_{j=1}^{d}(x_{ij} - \mu_i)^2, \forall i,$$

where $\gamma, \beta \in \mathbb{R}^d$ are learnable parameters shared for all features in the sequence, μ_i, σ_i are the mean and variance calculated for the i-th feature, and ϵ is a small constant for numerical stability. The number of parameters in each LayerNorm is twice the feature dimension.

RMSNorm. RMSNorm normalizes only by the root mean square of the input feature, omitting the mean subtraction step:

$$X := \text{RMSNorm}(X), \text{ with } x_i = \frac{x_i}{\text{RMS}(f_i) + \epsilon} \cdot \gamma, \quad \text{RMS}(f_i) = \sqrt{\frac{1}{d}\sum_{j=1}^{d} x_{ij}^2},$$

where $\gamma \in \mathbb{R}^d$ are learnable parameters shared for all features in the sequence, $\text{RMS}(x_i)$ is the root mean square calculated for the i-th feature vector, and ϵ is a small constant for numerical stability. The number of parameters in each RMSNorm is the same as the feature dimension.

Both LayerNorm and RMSNorm operate independently on each token's feature vector in the sequence, making them well-suited for language tasks where input lengths can vary. RMSNorm is computationally cheaper than LayerNorm and has been empirically shown to be effective for training very deep LLMs.

Llama Implementation

```python
# We will first implement a single Transformer layer for demonstration purposes.
x = token_embeddings # H^0 = E, input to the first layer is embedding
def rms_norm(x, weight, eps=1e-5):
    norm = torch.rsqrt(x.pow(2).mean(dim=-1, keepdim=True) + eps)
    return x * norm * weight

x_norm = rms_norm(x, model["layers.0.attention_norm.weight"], norm_eps) # (N, dim)
```

2.4.2 Attention

Given an input sequence of N features $X = \{x_1, x_2, \ldots, x_N\} \in \mathbb{R}^{N \times d}$, the self-attention mechanism first creates queries (Q), keys (K), and values (V) by multiplying X with learned weight matrices:

$$Q = XW^Q \in \mathbb{R}^{N \times d}, \quad K = XW^K \in \mathbb{R}^{N \times d}, \quad V = XW^V \in \mathbb{R}^{N \times d},$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d \times d}$. Technically, the weight matrices W^Q, W^K, W^V can project the input dimension d to a different dimension d_k. In most LLMs, $d_k = d$ so that the hidden state features will always have the same dimension. We will omit k in what follows.

Remark 2.1 (Simple Linear Algebra)**.** *The i-th row of the resulting Q, K, and V matrices depends only on the i-th input feature. Therefore, when a new token T_{i+1} is added, we only need to compute for Q_{i+1}, K_{i+1}, and V_{i+1}.*

The attention weights are then computed by:

$$\mathrm{A} = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) \in \mathbb{R}^{N \times N},$$

where $QK^\top$ is called the attention scores. As an autoregressive model, the attention weights are made *causal*, meaning that each token should not "see the future." Therefore, the attention weights are masked to be a lower triangular matrix, so each position can only attend to itself and the tokens that come before it: $A_{ij} = 0 \quad \text{for all } i < j$.

Remark 2.2 (Simple Linear Algebra)**.** *The i-th row of the attention weights only depends on Q_i and $K_{1:i}$.*

In implementation, this masking is done efficiently by adding a strictly upper triangular matrix $M \in \mathbb{R}^{N \times N}$ whose strictly upper triangular elements are set to $-\infty$:

$$M_{ij} = \begin{cases} -\infty & i < j \\ 0 & i \geq j. \end{cases}$$

The final output of attention is give by:

$$\mathrm{Attention}(X) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right) V \in \mathbb{R}^{N \times d}. \tag{2.2}$$

Remark 2.3 (Simple Linear Algebra)**.** *The i-th final output feature depends on Q_i, $K_{1:i}$, and $V_{1:i}$. Therefore, the i-th output feature of attention encodes information about the entire prefix $X_{1:i}$. In practice, $K_{1:i}$ and $V_{1:i}$ should be cached (**KV cache**) during generation to avoid recomputing them for each new token.*

After the attention weights are applied to the values V, the result is oftentimes passed through an additional learned linear projection:

$$\mathrm{Attention}(X) = \underbrace{\mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right) V}_{\in \mathbb{R}^{N \times d}} W^O, \quad W^O \in \mathbb{R}^{d \times d}. \tag{2.3}$$

This output projection layer W^O mixes the attended features to produce the final representation. Remark 2.4 holds regardless.

Complexity Analysis. First, the projection layers with W^Q, W^K, W^V, W^O: $(n, d) \times (d, d) \Rightarrow (n, d)$; the complexity is $O(Nd^2)$. Second, calculating attention scores: $(n, d) \times (d, n) \Rightarrow (n, n)$; the complexity is $O(N^2 d)$. Third, multiplying by the value matrix V: $(n, n) \Rightarrow (n, d) = (n, d)$; the complexity is $O(N^2 d)$.

Remark 2.4. *The number of parameters of attention is $4d^2$. The computational complexity of attention is $O(Nd^2 + N^2 d)$, with additional costs from softmax and overheads.*

Multi-head Attention. Instead of applying a single attention operation, multi-head attention uses multiple independent attention heads in parallel. Each head learns its own set of projections:

$$Q^{(h)} = XW^{Q(h)} \in \mathbb{R}^{N \times d_h}, \quad K^{(h)} = XW^{K(h)} \in \mathbb{R}^{N \times d_h}, \quad V^{(h)} = XW^{V(h)} \in \mathbb{R}^{N \times d_h}, \quad h = 1, \ldots, H,$$

where $W^{Q(h)}, W^{K(h)}, W^{V(h)} \in \mathbb{R}^{d \times d_h}$, and the total number of heads H satisfies $d = H \cdot d_h$. Each attention head computes its own output:

$$\mathrm{Attention}^{(h)}(X) = \mathrm{softmax}\left(\frac{Q^{(h)} K^{(h)\top}}{\sqrt{d_h}} + M\right) V^{(h)} \in \mathbb{R}^{N \times d_h}.$$

All head outputs are then concatenated along the feature dimension and projected back to the original dimension using a final learned projection $W^O \in \mathbb{R}^{d \times d}$:

$$\mathrm{MultiHead}(X) = \underbrace{\left[\mathrm{Attention}^{(1)}(X), \ldots, \mathrm{Attention}^{(H)}(X)\right]}_{\in \mathbb{R}^{N \times d}} W^O.$$

Remark 2.5. *In multi-head attention,*

$$\left[W^{Q(1)}, \dots, W^{Q(H)}\right] \in \mathbb{R}^{d \times d}, \quad \left[W^{K(1)}, \dots, W^{K(H)}\right] \in \mathbb{R}^{d \times d}, \quad \left[W^{V(1)}, \dots, W^{V(H)}\right] \in \mathbb{R}^{d \times d}.$$

Grouped query multi-head attention has the same number of parameters and same computational complexity as single-head attention.

The motivation for multi-head attention is to allow the model to jointly attend to information from different representation subspaces at different positions. Empirically, using multiple heads has been shown to consistently improve performance in practice.

Grouped Query Attention. In LLaMA 3 and similar models, *Grouped Query Attention* (GQA) reduces the number of unique key-value pairs by sharing them across multiple heads while keeping independent query projections per head.

Formally, we partition the H attention heads into G groups with $G < H$. Each group shares the same keys and values, but each head still has its own query. Mathematically, $W^{K(h)} = W^{K(h')} = W^{K(g)}$ and $W^{V(h)} = W^{V(h')} = W^{V(g)}$, $\forall h, h'$ in the same group g, $g = 1, \dots, G$. Since the weights are identical within each group, the resulting key and value matrices are also the same for all heads in that group, so they only need to be computed once. In the Llama configurations that we have shown above, $H = 32$ and $G = 8$.

Remark 2.6. *In GQA, each head still produces queries, keys, and values of the same dimension as standard multi-head attention — the only difference is that keys and values are shared within groups while queries remain unique per head.*

Remark 2.7. *In grouped query multi-head attention,*

$$\left[W^{Q(1)}, \dots, W^{Q(H)}\right] \in \mathbb{R}^{d \times d}, \quad \left[W^{K(1)}, \dots, W^{K(G)}\right] \in \mathbb{R}^{d \times \frac{d}{H/G}}, \quad \left[W^{V(1)}, \dots, W^{V(G)}\right] \in \mathbb{R}^{d \times \frac{d}{H/G}},$$

where H/G is the number of keys and values per group. Grouped query multi-head attention has less parameters and less computations.

2.4.3 Rotary Positional Embedding

Given the projected queries and keys $Q, K \in \mathbb{R}^{N \times d}$, RoPE divides the feature dimension d into $\frac{d}{2}$ 2D planes, assuming d is even. So, each pair of adjacent dimensions forms a 2D vector. For each pair $(Q_{i,2k-1}, Q_{i,2k})$ or $(K_{i,2k-1}, K_{i,2k})$, $k = 1, 2, \dots, \frac{d}{2}$,the rotation is:

$$\begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix}, \quad \theta_i = i \cdot \Theta_k,$$

where each dimension pair has its own frequency Θ_k:

$$\Theta_k = \theta_{\text{rope}}^{-2(k-1)/d},$$

where θ_{rope} is a constant hyperparameter. Overall, RoPE applies a position-dependent rotation to each row vector Q_i and K_i independently:

$$\tilde{Q}_i = R(i)Q_i, \quad \tilde{K}_i = R(i)K_i, \quad i = 1, \dots, N.$$

The rotation matrix $R(i)$ is given by

$$R(i) = \bigoplus_{k=1}^{d/2} \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix}, \quad \theta_i = i \cdot \Theta_k.$$

Remark 2.8. *RoPE is applied independently to each position, so adding a new token T_{i+1} only requires computing $R(i+1)Q_{i+1}$ and $R(i+1)K_{i+1}$.*

Remark 2.9 (Relative and Absolute Position Encoding)**.** *RoPE encodes **absolute** position information as it rotates by absolute index. Moreover, the queries and keys have the property:*

$$\left\langle \tilde{Q}_i, \tilde{K}_j \right\rangle = \langle R(i)Q_i, R(j)K_j \rangle = \langle R(j-i)Q_i, K_j \rangle,$$

*which shows that the dot product depends on the relative distance $j - i$. Therefore, the attention score, $\tilde{Q}\tilde{K}^\top$, depends on **relative** position difference.*

For multi-head attention, RoPE is applied to each head individually.

2.4.4 Llama Implementation

RoPE. First, we compute the rotations given the model dimension d, the sequence length N (since the rotations are fixed, they can also be precomputed up to a maximum sequence length), and the RoPE hyperparameter θ_{rope} .

```python
def precompute_freqs_cis(dim: int, end: int, theta: float):
```

We compute all the θ_i:

```python
    freqs = 1.0 / (theta ** (torch.arange(0, dim, 2)[: (dim // 2)].float() / dim)) # all \Theta_k
    t = torch.arange(end, device=freqs.device, dtype=torch.float32) # i in our notation
    freqs = torch.outer(t, freqs)  # all \theta_i
```

In actual implementation, we work with complex numbers for rotations as it is memory efficient and computationally faster; GPUs handle element-wise complex multiplication efficiently. A 2D rotation of $[x, y]$ by angle θ can be done either in matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix},$$

or in complex form:

$$z' = z \cdot e^{i\theta}, \quad z = x + iy.$$

These are exactly the same mathematically. We convert all the frequencies to complex frequencies:

```python
    freqs_cis = torch.polar(torch.ones_like(freqs), freqs)  # complex numbers using Eulers Formula
    return freqs_cis # cis means cosine + i sine; the shape is (N, d_head//2)
```

Now, we apply the rotations as a simple multiplication of complex numbers:

```python
def apply_rotary_emb(
  xq: torch.Tensor, # (N, n_heads, head_dim)
  xk: torch.Tensor, # (N, n_kv_heads, head_dim)
  freqs_cis: torch.Tensor # (N, head_dim//2)
  ) -> Tuple[torch.Tensor, torch.Tensor]:
    N, _, head_dim = xq.shape
    xq_ = torch.view_as_complex(xq.float().reshape(*xq.shape[:-1], -1, 2)) # convert to complex numbers
    xk_ = torch.view_as_complex(xk.float().reshape(*xk.shape[:-1], -1, 2))
    freqs_cis = freqs_cis.reshape(N, 1, head_dim//2) # reshape for broadcasting
    xq_out = torch.view_as_real(xq_ * freqs_cis).flatten(2) # convert back to real numbers
    xk_out = torch.view_as_real(xk_ * freqs_cis).flatten(2)
    return xq_out.type_as(xq), xk_out.type_as(xk) # (N, n_heads, head_dim), (N, n_kv_heads, head_dim)
```

Grouped Query Multi-head Attention. In Llama implementation, the weight matrices are stored in stacked format. A single large matrix multiplication is then performed, followed by reshaping the result into multiple heads. This is **equivalent** to first reshaping the weights into multiple heads and then performing the matrix multiplication separately for each head (can be parallelized with einsum.

Recall from Sec. 2.4.1 that the input is now the normalized hidden feature $x_{\text{norm}} \in \mathbb{R}^{N\times d}$. We first get N, d, and the projection weight matrices $\left[W^{Q(1)}, \ldots, W^{Q(H)}\right] \in \mathbb{R}^{d\times d}$, $\left[W^{K(1)}, \ldots, W^{K(G)}\right] \in \mathbb{R}^{d\times \frac{d}{H/G}}$, and $\left[W^{V(1)}, \ldots, W^{V(G)}\right] \in \mathbb{R}^{d\times \frac{d}{H/G}}$. In the configuration, $d = 2048, H = 32, G = 8, H/G = 4$.

```python
seqlen, _ = x_norm.shape
head_dim = dim//n_heads
wq = model["layers.0.attention.wq.weight"]  # (dim, dim) = (2048, 2048)
wk = model["layers.0.attention.wk.weight"]  # (dim/(H/G), dim) = (512, 2048)
wv = model["layers.0.attention.wv.weight"]  # (dim/(H/G), dim) = (512, 2048)
```

Now, we project the hidden features to keys, queries, and values, and reshape them to multiple heads:

```python
xq = torch.matmul(x_norm, wq.T) # (N, dim) x (dim, dim) -> (N, dim) = (6, 2048)
xk = torch.matmul(x_norm, wk.T) # (N, dim) x (dim, dim/group_size) -> (N, dim/group_size) =  (6, 512)
xv = torch.matmul(x_norm, wv.T) # (N, dim) x (dim, dim/group_size) -> (N, dim/group_size) =  (6, 512)

xq = xq.view(seqlen, n_heads, head_dim) # (N, n_heads, head_dim) = (6, 32, 64)
xk = xk.view(seqlen, n_kv_heads, head_dim) # (N, n_kv_heads, head_dim) = (6, 8, 64)
xv = xv.view(seqlen, n_kv_heads, head_dim) # (N, n_kv_heads, head_dim) = (6, 8, 64)
print("The shapes of xq, xk, xv are:", xq.shape, xk.shape, xv.shape)
```

Python Output

```
The shapes of xq, xk, xv are: torch.Size([6, 32, 64]) torch.Size([6, 8, 64]) torch.Size([6, 8, 64])
```

Additionally, we provide code for you to verify that this is equivalent to reshaping the weights to multiple heads and performing the projection within each head independently:

```
# Computing heads individually is equivalent to a single matrix multiplciation then reshape
def verify_xq(x_norm, wq):
  for i in range(n_heads):
    wq = wq.view(n_heads, head_dim, dim)
    wq_i = wq[i]
    xq_i = torch.matmul(x_norm, wq_i.T)
    if not torch.allclose(xq_i, xq[:,i,:], rtol=1e-2, atol=1e-2): # bfloat16 has low precision
      return False
    return True
print(verify_xq(x_norm,wq))
```

Python Output

```
True
```

We apply RoPE to queries and keys:

```
freqs_cis = precompute_freqs_cis(head_dim, seqlen, rope_theta).to(device)
xq, xk = apply_rotary_emb(xq, xk, freqs_cis=freqs_cis) # no change in dimension
```

Now that we have all queries, keys, and values ready, we can compute the final attention output as in Eq. 2.3:

```
### Equation 2.3: ###
# repeat k,v to the shape of q for parallel computing
n_rep = n_heads//n_kv_heads
xk, xv = torch.repeat_interleave(xk, repeats=n_rep, dim=1), torch.repeat_interleave(xv, repeats=n_rep, dim
    ↪ =1) # (6, 32, 64), (6, 32, 64)
# transpose for torch.matmul on the length and feature dimensions
xq, xk, xv = xq.transpose(0,1), xk.transpose(0,1), xv.transpose(0,1) # (32, 6, 64), (32, 6, 64), (32, 6,
    ↪ 64)
scores = torch.matmul(xq, xk.transpose(1, 2)) / math.sqrt(head_dim)  # (32, 6, 6)
mask = torch.full((seqlen, seqlen), float("-inf"), device=device)    # (6, 6)
mask = torch.triu(mask, diagonal=1) # making it strictly upper triangular for causality
scores = scores + mask  # (32, 6, 6)
scores = F.softmax(scores.float(), dim=-1).type_as(x) # (32, 6, 6)
attention = torch.matmul(scores, xv) # (32, 6, 6) x (32, 6, 64) -> (32, 6, 64)
attention = attention.transpose(0, 1).contiguous().view(seqlen, -1) # (6, 2048)
wo = model["layers.0.attention.wo.weight"] # (2048, 2048)
attention = torch.matmul(attention, wo.T) # (6, 2048)
```

2.4.5 Residual Connection and SwiGLU FFN

As shown in Equation 2.1, after attention, a residually connection is applied: $H = X + \text{MultiHeadAttention}\big(\text{RMSNorm}(X)\big) \in \mathbb{R}^{N\times d}$. Given H block, the transformer applies a position-wise feedforward network (FFN) to each feature of normalized H independently: $X' = H + \text{FeedForward}\big(\text{RMSNorm}(H)\big) \in \mathbb{R}^{N\times d}$. Specifically, Llama uses SwiGLU FFN with a gated activation mechanism with a Swish nonlinearity. Formally, let the normalized H be $H' \in \mathbb{R}^{N\times d}$, SwiGLU FFN is given by:

$$\text{SwiGLU}(H') = (H'W_1) \odot \sigma\left(H'W_3\right), \text{then project: } \text{FFN}(H') = \text{SwiGLU}(H')W_2,$$

where $W_1, W_3 \in \mathbb{R}^{d\times d_{ff}}, W_2 \in \mathbb{R}^{d_{ff}\times d}$ are learnable weights, d_{ff} is typically $4d$, and $\sigma(\cdot)$ is the SiLU(Swish) non-linearity.

Remark 2.10. *The SwiGLU FFN applies the same feedforward transformation independently to each row of X. Therefore, when a new token T_{i+1} is generated, only the $i+1$-th row of the FFN output needs to be computed.*

Llama Implementation

```
h = x + attention # (6, 2048)
h_norm = rms_norm(h, model["layers.0.ffn_norm.weight"], norm_eps) # (6, 2048)
w1 = model["layers.0.feed_forward.w1.weight"] # (8192, 2048)
w3 = model["layers.0.feed_forward.w3.weight"] # (8192, 2048)
w2 = model["layers.0.feed_forward.w2.weight"] # (2048, 8192)
h1 = torch.matmul(h_norm, w1.T) # (6, 8192)
h2 = torch.matmul(h_norm, w3.T) # (6, 8192)
activated = torch.functional.F.silu(h1) * h2 # (6, 8192)
# The final output after the first transformer layer, same shape as the input. Features are casual
x = torch.matmul(activated, w2.T) + h # (6, 2048)
```

We repeat this $L-1$ more times to get the final hidden features $H^L \in \mathbb{R}^{\mathbb{N}\times}$:

Llama Implementation

```
def transformer_block(x, layer):
    ...
for layer in range(1, n_layers):
  x = transformer_block(x, layer)
print(x.shape)
```

Python Output

```
torch.Size([6, 2048])
```

2.5 LM Head and Sampling

After passing through L stacked Transformer layers, **the final hidden features $H^L \in \mathbb{R}^{N\times d}$ contain contextualized casual representations: the i-th feature H_i^L models the context of the first i tokens.** To transform these features into a probability distribution over the vocabulary, we apply a final linear projection called the *language modeling (LM) head*. The LM head will project each row of H^L onto the vocabulary space.

Formally, let $W_{LM} \in \mathbb{R}^{V\times d}$ be the learned word embedding matrix, where V is the vocabulary size and d is the model dimension. The LM head is given by

$$\text{Logits} = LM(H^L) = \text{RMSNorm}(H^L)E^\top \in \mathbb{R}^{N\times V},$$

where Logits_i is the logits for predicting the $i+1$-th token.

Remark 2.11. *The LM head operates independently for each token position. Therefore, when generating a new token, only the last row H_N^L needs to be projected and normalized.*

The final next-token probability distribution for each position is then obtained by applying the softmax function to the logits:

$$P\left(T_{i+1}\right) = \text{softmax}\left(\text{Logits}_i\right) \in \mathbb{R}^V.$$

Sampling. Once the next-token probabilities are available, text generation proceeds by sampling from this distribution. Several decoding strategies exist:

- *Greedy decoding*: Select the token with the highest probability.
- *Top-k sampling*: Keep only the k most probable tokens and renormalize the distribution before sampling.
- *Top-p (nucleus) sampling*: Keep the smallest set of tokens whose cumulative probability exceeds p and renormalize before sampling.
- *Temperature scaling*: The logits can be divided by a temperature hyperparameter $\tau > 0$ before applying softmax. A lower τ makes the distribution sharper (more deterministic); a higher τ makes it flatter (more random).

The newly sampled token is appended to the input sequence, and the process repeats autoregressively until a stopping criterion is met, e.g., an end-of-sequence token is generated or a maximum length is reached.

Llama Implementation

```
x_norm = rms_norm(x, model["norm.weight"], norm_eps)
logits = torch.matmul(x_norm[-1], model["output.weight"].T) # Only need the last row
next_token = torch.argmax(logits, dim=-1) # Greedy prediction
print("The next token is:", tokenizer.decode([next_token.item()]))
```

Python Output

```
The next token is:  Paris
```

The input prompt is “The capital of France is". Try it yourself by adding a space to the end of this prompt: “The capital of France is ".

3 Advanced Concepts

3.1 KV Caching

To be added.

3.2 Quantization

To be added.

3.3 Mixture of Experts

To be added.

References

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Wyatt, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Guzmán, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Thattai, G., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Zhang, J., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Prasad, K., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Lakhotia, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Tsimpoukelli, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Zhang, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Maheswari, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Albiero, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Wang, X., Tan, X. E., Xia, X., Xie, X., Jia, X., Wang, X., Goldschlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang, Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Srivastava, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Teo, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Dong, A., Franco, A., Goyal, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Liu, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Gao, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Le, E.-T., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Kokkinos, F., Ozgenel, F., Caggioni, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Inan, H., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Zhan, H., Damlaj, I., Molybog, I., Tufanov, I., Leontiadis, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, J., Lam, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Jagadeesh, K., Huang, K., Chawla, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Liu, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Mehta, N., Laptev, N. P., Dong, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O., Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P., Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Parthasarathy, R., Li, R., Hogan, R., Battey, R., Wang, R., Howes, R., Rinott, R., Mehta, S., Siby, S., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Mahajan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Patil, S., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Deng, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Koehler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wu, X., Wang, X., Wu, X., Gao, X., Kleinman, Y., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Zhao, Y., Hao, Y., Qian, Y., Li, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., Zhao, Z., and Ma, Z. (2024). The llama 3 herd of models.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.