CSCI 699: The Science of Large Language Models
Lecture Notes on Language Models and Transformers

Robin Jia

Fall 2024

Chapter 1

Language Models

In this class, we will focus on **autoregressive language models (LMs)**. This stands in contrast to models like BERT, which are *masked* language models. Informally, an autoregressive language model predicts the next token in a document given all previous tokens.

1.1 Formal Definition

Let $w_1, \ldots, w_T$ denote the first T tokens of a document.[1] We assume that all tokens in the world (including $w_1, \ldots, w_T$) come from a known, finite set of possible tokens V, often referred to as the **vocabulary** (you can think of this as the set of all English words).

An autoregressive language model with parameters θ defines a probability distribution over the next token in the document, w_{T+1}, given any prefix of tokens $w_1, \ldots, w_T$:

$$p_{LM}(w_{T+1} \mid w_1, \ldots, w_T; \theta).$$

Since all tokens come from the vocabulary V, p_{LM} outputs a distribution over the $|V|$ possible next tokens. We can view this as a vector in $\mathbb{R}^{|V|}$, or more specifically, a vector in the $|V|$-dimensional simplex $\Delta^{|V|}$. Recall that the n-dimensional (probability) simplex $\Delta^{|V|}$ is defined as the set of points $v \in \mathbb{R}^n$ that satisfy:

$$v_i \geq 0 \, \forall \, i = 1, \ldots, n$$
$$\sum_{i=1}^{n} v_i = 1.$$

This is the space of valid probability distributions over $|V|$ objects.

1.2 Probabilities over sequences

Since an autoregressive LM produces a probability distribution given any prefix of any length T, it also defines a valid probability distribution over $|V|^k$, the set of length-k strings whose

[1] If you are unfamiliar with tokenization, you can think of each "token" as a word for now. Later, we will discuss the difference between words and tokens.

elements come from V, for any positive integer k. For any string $s = w_1, \ldots, w_T$, we define the overall probability assigned to s by the language model with parameters θ as

$$p_{LM}(w_1, \ldots, w_T; \theta) = \prod_{t=1}^{T} p_{LM}(w_t \mid w_1, \ldots, w_{t-1}; \theta).$$

1.3 Autoregressive pre-training

Pre-training an autoregressive LM is straightforward. We obtain a large dataset D of documents, each of which is a sequence of tokens $s = [s_1, \ldots, s_T]$. On this dataset, we can write down the overall cross-entropy loss for the model parameters θ:

$$L(\theta) = \frac{1}{|D|} \sum_{s \in D} \sum_{t=1}^{T} \log p_{LM}(s_t \mid s_1, \ldots, s_{t-1}; \theta).$$

This is a standard loss function that sums of the log-likelihood of every token given its prefix, and averages this across the entire dataset. We will learn all model parameters θ to minimize this objective.

For pre-training large language models, it is common to optimize this objective using stochastic gradient descent with a relatively large batch size, and to do only a single pass over the training data (to a first approximation).

1.4 Usage and efficiency considerations

The most common way to use an autoregressive LM is to generate a *continuation* of an input prefix $w_1, \ldots, w_T$. By continuation, we just mean a sequence of k plausible tokens $w_{T+1}, \ldots, w_{T+k}$ that could be the next k tokens in the document.

Given this common use case, it would be highly desirable for the computation of

$$p_{LM}(w_{T+2} \mid w_1, \ldots, w_T, w_{t+1} \mid \theta)$$

to be able to reuse work done to compute

$$p_{LM}(w_{T+1} \mid w_1, \ldots, w_T \mid \theta),$$

and so on. This property is a key design consideration for the two most popular families of autoregressive LMs: recurrent neural networks (RNNs), which include models like LSTMs, and Transformers, the subject of this class.

Chapter 2

Transformers

A **Transformer** is a type of neural network architecture that can be trained as an autoregressive language model. In these notes, we will focus on the special case of Transformer-based autoregressive LMs, and refer to these as Transformers for short. Note that it is also possible to use Transformers in other ways. In addition, we will follow one particular common Transformer variant, but many slight variants are also possible and in use.

2.1 Architecture overview

We first given an overview of the Transformer architecture before diving into each component ind etail. Recall that the input to the model is a sequence of tokens $w_1, \ldots, w_T$, and the output is a vector in the simplex $\Delta^{|V|}$. A Transformer has two key hyperparameters, the hidden dimension d and number of layers L. It processes its input in three steps:

1. Each input token w_t is first mapped to a corresponding *token embedding* $x_t \in \mathbb{R}^d$. This is necessary because all other model components operate on vectors, so we must communicate the input sequence to them in vector format.

2. For each layer $\ell = 1, \ldots, L$, and each token $t = 1, \ldots, T$, a hidden state $h_t^\ell \in \mathbb{R}^d$ is computed based on the update rule:

$$h_t^\ell = h_t^{\ell-1} + a_t^\ell + m_t^\ell. \tag{2.1}$$

   The base case h_t^0 is defined to be x_t. a_t^ℓ and m_t^ℓ are the outputs of the ℓ-th multi-headed attention layer and the ℓ-th Feedforward Network/Multi-layer Perceptron layer, respectively, at position t. We will define those in subsequent sections. At a high level, the multi-headed attention layer is responsible for retrieving relevant information about previous tokens and incorporating that into the representation for the current token. The feedforward network does additional non-linear processing about the information we have accumulated so far about the current token.

3. The final hidden state for the last token, h_t^L, is mapped to the final output, a vector in $\Delta^{|V|}$, by computing

$$\text{Softmax}(W^{\text{unembed}} \cdot \text{LayerNorm}(h_t^L)).$$

W^{unembed} is a parameter matrix $\in \mathbb{R}^{d \times |V|}$, and LayerNorm is the layer normalization operation (to be described in detail later), which maps vectors in $\mathbb{R}^d$ to other vectors in $\mathbb{R}^d$. Softmax transforms a vector $v \in \mathbb{R}^n$ into a new vector of the same dimension as defined by the following:

$$\text{Softmax}(v) = \left[\frac{e^{v_1}}{\sum_{i=1}^{n} e^{v_i}}, \ldots, \frac{e^{v_n}}{\sum_{i=1}^{n} e^{v_i}} \right]$$

Softmax transforms the vector v into a new vector whose entries are all positive (since e^x is always positive) and sum to 1, so this is a valid probability distribution as desired. It also preserves the relative ordering of all the entries, e.g., if the largest entry of v is at index i, then the largest entry of Softmax(v) is also at index i, and so forth.

2.1.1 The Residual Stream

Note that we can unroll the hidden state recurrence and write the final hidden state h_t^L as

$$h_t^L = x_t + \sum_{\ell=1}^{L} a_t^\ell + m_t^\ell.$$

The final hidden state is simply the sum of many different modules of the Transformer model; it is incrementally updated over the layers. Since the practice of adding new components to the previous layer's hidden state is often referred to as using "residual connections," the incrementally updating hidden state of the Transformer is often referred to as the **residual stream**.

2.2 Layer Normalization

We've mentioned a mysterious operation called Layer Normalization. Before we go any further, let's address what that is. Layer Normalization (LN) is just a particular type of neural network layer. It takes in a vector and transforms it according to a couple learnable parameters. This transformation includes a "normalization" step that ensures that all the entries in the vector are on a reasonable "scale," i.e., they're not all really big numbers or really small numbers.

2.2.1 Layer Normalization definition

A Layer Normalization layer takes as input a vector $x \in \mathbb{R}^n$ and outputs a vector of the same dimension through the following steps:

1. Compute the mean of x:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

2. Compute the standard deviation of x:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

3. Renormalize x to have 0 mean and unit variance:

$$\tilde{x} = \frac{x - \mu}{\sigma}$$

4. Rescale and shift this normalized vector by a learned elementwise scale $a \in \mathbb{R}^n$ and shift $b \in \mathbb{R}^n$, and return this quantity:

$$\text{LayerNorm}(x) = a \odot \tilde{x} + b,$$

where $\odot$ denotes elementwise multiplication.

Let's walk through a simple example: say we have $x = [100, 200, 100, 0]$. We can compute the mean $\mu = 100$ and the standard deviation $\sigma = \sqrt{\frac{1}{4} \cdot (0^2 + 100^2 + 0^2 + 100^2)} = \sqrt{5000} \approx 71$. The normalized vector $\tilde{x}$ is thus

$$[0, 100, 0, -100]/71 \approx [0, 1.4, 0, -1.4].$$

The final output will be shifted by b and scaled by a, so it is

$$[b_1, 1.4 \cdot a_2 + b_2, b_3, -1.4 \cdot a_4 + b_4].$$

2.2.2 Motivation

Why is LN useful? Roughly speaking, you want everything in your network to have roughly zero mean and unit variance, because really large numbers mess with optimization (gradients can get very big, so you don't converge), as do really small numbers (gradients can get very small, so you make little progress on each gradient step). But you also want to be able to have some flexibility in the scale of your values, so you include the learned a and b parameters.

2.2.3 Post-LN or Pre-LN?

In the original Transformer architecture, Layer Normalization was applied *after* every residual connection, referred to as "Post-LN." However, current common practice is instead to apply Layer Normalization right before computing either the MHA or FFN layer, referred to as "Pre-LN." Refer to Figure 2.1 for an illustration.

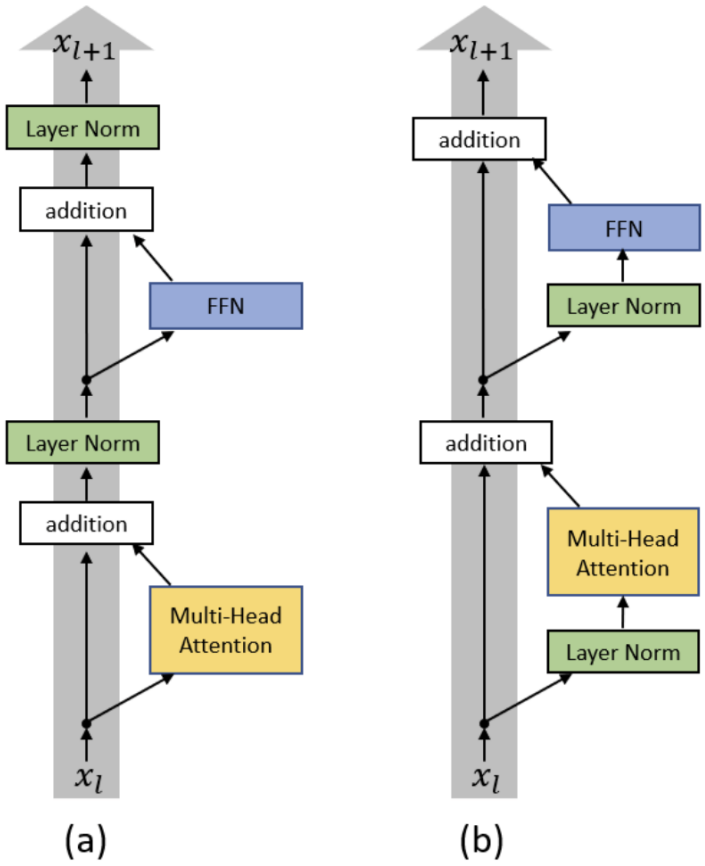

Figure 2.1: Illustration of the residual stream for a model that uses (a) post-Layer Normalization and (b) pre-Layer Normalization. The current standard is to use pre-Layer Normalization. Taken from https://arxiv.org/abs/2002.04745.

2.2.4 RMSNorm: A simpler Layer Normalization

While Layer Normalization was used in the original Transformer, Zhang and Sennrich later proposed RMSNorm as a simpler alternative.[1] Given a vector x, RMSNorm computes

$$\text{RMSNorm}(x) = a \odot \frac{x}{\text{RMS}(x)}$$

where $a \in \mathbb{R}^n$ is a learned elementwise scale parameter, and RMS denotes the root-mean-square operation:

$$\text{RMS}(x) \triangleq \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}.$$

Compared to standard Layer Normalization, RMSNorm requires fewer steps, as it only computes RMS as opposed to computing the mean and then standard deviation; thus, it offers a slight efficiency boost. The intuition is that the most important thing to normalize is the

[1] https://arxiv.org/abs/1910.07467

scale of the inputs, rather than their average value. Because it is slightly faster, RMSNorm is commonly used in modern Transformers.

To compare this with standard Layer Normalization, let's use the same example $x = [100, 200, 100, 0]$. We directly compute the RMS as $\sqrt{\frac{1}{4} \cdot (0^2 + 100^2 + 200^2 + 100^2)} \approx 122$, so we have

$$\tilde{x} = [100, 200, 100, 0]/122 \approx [0.8, 1.6, 0.8, 0],$$

and so the final output with the scale is

$$[0.8 \cdot a_1, 1.6 \cdot a_2, 0.8 \cdot a_3, 0].$$

2.3 Token embeddings

Let's move on to the first step of the Transformer: converting the sequence of input tokens $w_1, \ldots, w_T$ into a corresponding sequence of vectors $x_1, \ldots, x_T$, where each $x_t \in \mathbb{R}^d$.

At a high level, each x_t is the sum of two vectors: one that encodes the identity of the token w_t, and one that encodes the index t of that token. An alternative, called relative positional embeddings, omits the second vector, and instead uses different mechanisms to keep track of the position of each word.

2.3.1 Vocabulary embeddings

First, recall that every token comes from the finite set V. We can easily assign each unique token in the vocabulary a unique number from $1, \ldots, |V|$. So, we will now treat each token $w_1, \ldots, w_n$ as an integer between 1 and $|V|$.

The token embeddings of the model are represented by a single matrix W^{embed} of dimension $|V| \times d$. The token embedding layer simply maps each token w_t to the w_t-th row of W^{embed}, which we will denote as $W^{\text{embed}}[w_t]$. You can think of W^{embed} as a big dictionary that maps each possible token $v \in V$ to a corresponding vector.

2.3.2 Tokenization

We keep referring to "tokens" rather than words. The reason for this is that everything here depends on V being a finite set, whereas the set of possible "words" is virtually infinite. Imagine new fictional characters, or typos of words—the model should not break when these words are passed in as input. To handle this problem, most modern language models employ **subword tokenization**, meaning that they split (some) words into multiple tokens. In particular, one popular strategy for subword tokenization is called **byte pair encoding** (BPE). I will not go into the details of BPE here, but just think of it as a way to choose a vocabulary V of subword tokens such that any possible word can be broken down deterministically into one or more subword tokens. Common words will generally be a single subword token, whereas rare words will be split into many subword tokens.

Figure 2.2 shows how an example sentence with some rare words (Lord of the Rings entity names) is tokenized by the GPT-4 tokenizer. We can see that it splits rare words like

"Aragorn," "Frodo," and "Lothlorien" into multiple tokens. In contrast, a long but common word like "instructed" is represented as a single token.

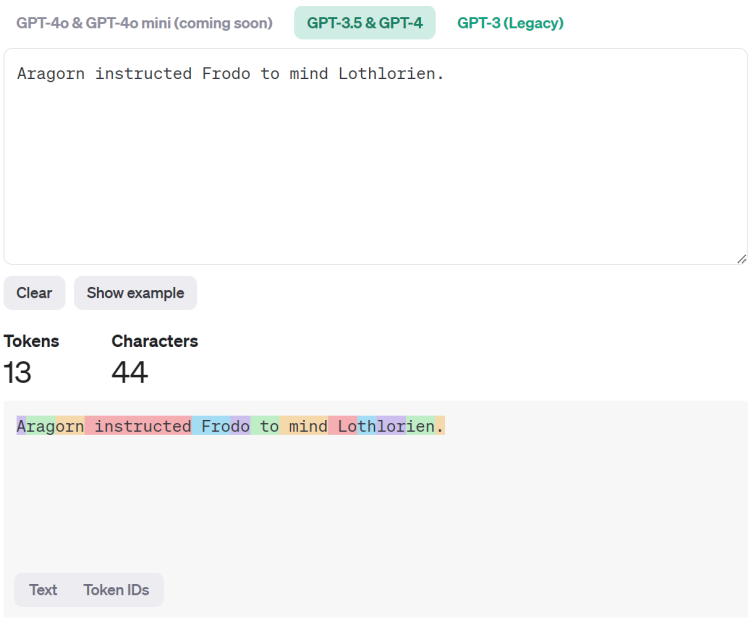

Figure 2.2: How the GPT-4 tokenizer processes the sentence "Aragorn instructed Frodo to mind Lothlorien." We can see that rare words are split into multiple tokens, while common words tend to be single tokens. Visual taken from https://platform.openai.com/tokenizer.

2.3.3 Absolute positional embeddings

We also need some way to tell the model which index in the sequence each token comes from. One standard way to do this is called **absolute positional embeddings**. We simply learn a separate positional embedding $p_t \in \mathbb{R}^d$ for each possible index t. The final token embedding for token t is the sum of the embedding for that token and the sum for its position:

$$x_t = W^{\text{embed}}[w_t] + p_t.$$

2.3.4 An alternative: Relative positional embeddings

Instead of absolute positional embeddings, modern Transformers tend to use one of several different **relative positional embedding** methods. These do away with the absolute positional embeddings and use a different mechanism to tell the model the order of the tokens. We will discuss one particular strategy called Rotary Positional Embeddings (RoPE), used by Llama-3, in the next class.

2.4 Multi-headed Self Attention Layers

The signature component of a Transformer is the **Multi-headed Self Attention (MHA)** layers. There is one MHA layer at each layer ℓ of the Transformer. An MHA layer takes in a sequence of vectors $u_1, \ldots, u_T \in \mathbb{R}^d$, of which u_T is the vector for the current timestep. It produces a vector of the same dimension.

MHA is a critical component of Transformers because it is the only component where information flows from one token to another. This is of course critical, as predicting the next token must involve incorporating information about all the past tokens in the sequence, not just the most recent one. Through the attention mechanism, the model learns which information from which tokens is relevant to the current prediction, and is able to access that information while ignoring other information.

As one simple example, consider a sentence like, *"Susan disliked Joe Biden because he. . . "* What comes next? Well, we have to understand that "he" must refer to "Joe Biden" and not to "Susan." Thus, what comes next must be a description of Biden. On the other hand, suppose that the sentence started, *"Susan disliked Joe Biden because* **she** *. . . "* Now, we understand that "she" refers to "Susan," so what comes next is likely a description of her views; we may know something about her views from the earlier context of the document, or we can infer from this snippet that her views are opposed to Biden's. In either case, it is important to understand the **coreference** relationship between the pronoun and its *antecedent*—the noun it refers to. These sorts of relationships between words are well-captured by attention, which will allow the model to "attend" back to either the word "Susan" or "Biden" when processing the pronoun. By gathering information about the antecedent, the model can make a more reasonable prediction of what comes next.

2.4.1 Single-headed self attention

To understand MHA, we must first understand "single-headed" self attention, and then generalize to the multi-headed case.

The attention head produces a vector of dimension d_{attn}, which is a hyperparameter (and is generally $< d$). A single self attention head is parameterized by three weight matrices W^Q, W^K, and W^V, short for query, key, and value, respectively. Each of these is a $d_{\text{attn}} \times d$ matrix This output is produced as follows:

1. The current input vector u_T is multiplied by W^Q to yield the query vector q_T
2. The input vectors $u_1, \ldots, u_T$ are multiplied by W^K and W^V to yield key and value vectors, respectively:
$$k_t = W^K \cdot u_t, \forall\, t = 1, \ldots, T;$$
$$v_t = W^V \cdot u_t, \forall\, t = 1, \ldots, T.$$
3. The query vector is dot producted with each key vector, computing a "matching score" s_t between the query and each key vector. This score is then normalized by $\sqrt{d_{\text{attn}}}$:
$$s_t = \frac{q_T^\top k_t}{d_{\text{attn}}} \forall\, t = 1, \ldots, T.$$

4. These matching scores are then transformed via softmax to yield probabilities $p_1, \ldots, p_T$:

$$[p_1, \ldots, p_T] = \text{Softmax}([s_1, \ldots, s_T]).$$

5. Finally, these resulting probabilities are treated as weights and a weighted sum of the value vectors is computed as the final output:

$$\sum_{t=1}^{T} p_t v_t.$$

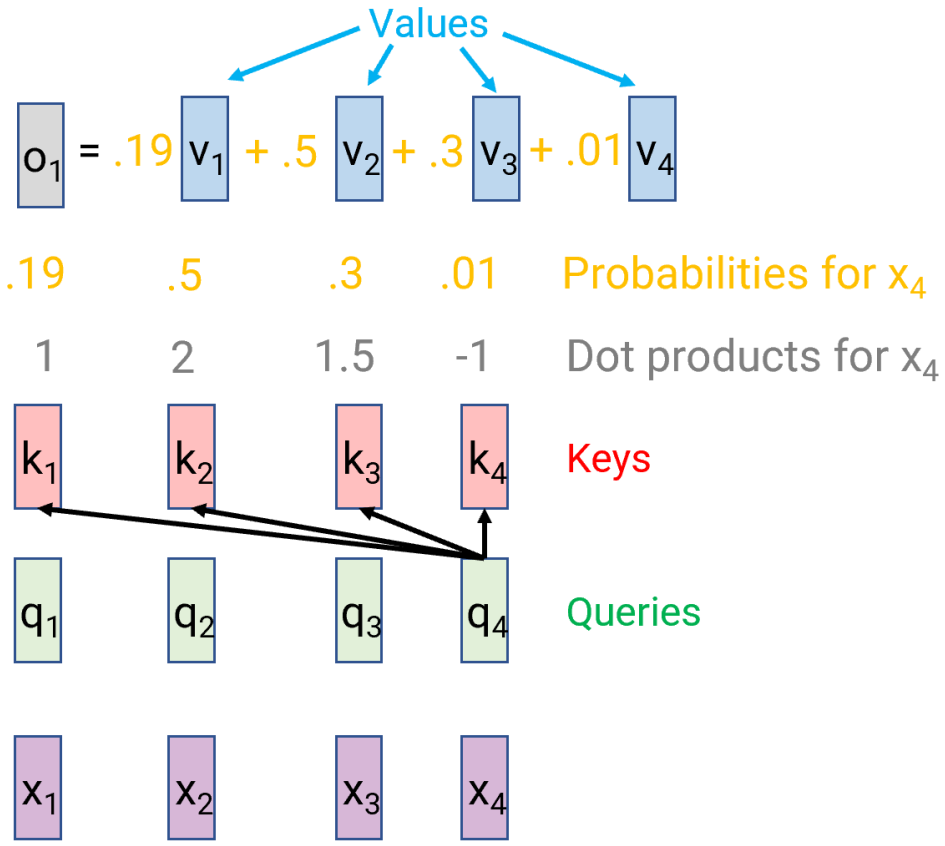

Figure 2.3: Illustration of one attention head, for an input sequence of length $T = 4$. We compute the dot product between the current query vector q_4 and each key vector k_1, k_2, k_3, k_4. The dot products are normalized to a probability distribution with softmax. Finally, we compute a weighted average of the value vectors using these probabilities.

Overall, what does this attention head do? It first tries to look for "relevant" previous tokens to the current token. Relevance is defined by the W^K and W^Q matrices: W^K projects the u_t's down to a lower dimension, preserving some information but erasing other information, and W^Q similar projects u_T down to a lower dimension to encapsulate what sorts of information this query is "looking for." Dot products are used to identify the most relevant indices, which get turned into the highest-weight indices after softmax. The weighted sum then grabs the corresponding information (in value vectors)

One helpful analogy could be the following:

- Query vector: A search query that a user types into a search engine (e.g., Google)
- Key vectors: A set of keywords from each webpage crawled by the search engine.
- Value vectors: The corresponding webpage itself, which is what the user actually wants to receive.

2.4.2 Multi-headed self attention

Now that we understand what a single head does, let's look at a full MHA layer. Recall that the MHA layer takes in vectors $u_1, \ldots, u_T$ of dimension d, and outputs a vector of the same dimension.

A full MHA layer is composed of n_{attn} attention heads (this is another hyperparameter). Each of them will usually have a head dimension of $d_{\text{attn}} = d/n_{\text{attn}}$. They each have their own separate parameter matrices, so in total we would have parameter matrices W_i^K, W_i^Q, and W_i^V for each $i = 1, \ldots, n_{\text{attn}}$. Finally, the MHA layer has one final parameter matrix W^O of dimension $d \times d$, which aggregates the outputs of all the heads into a final output.

The output of an MHA layer is defined as the following:

- Each of the n_{attn} attention heads is run on the input sequence in parallel, yielding head vectors $o_1, \ldots, o_{n_{\text{attn}}}$.
- These vectors are concatenated together and multiplied by W^O to yield the final output:

$$MHA(u_1, \ldots, u_T) = W^O \cdot [o_1; \cdots; o_{n_{\text{attn}}}]$$

Note that due to how we defined d_{attn}, the concatenation of all the n_{attn} vectors will exactly be of dimension d.

2.4.3 Use in Transformers

Now that we understand what an MHA layer does in isolation, let's return to how it is used in a Transformer.

A Transformer has one MHA layer, which we denote MHA^ℓ, for each layer $\ell = 1, \ldots, L$. MHA^ℓ is used at layer ℓ to compute a_t^ℓ from Equation 2.1, for all timesteps t. The input to MHA^ℓ at time t is the list of hidden states from the previous layer across all timesteps after applying Layer Normalization. In other words, we have

$$a_t^\ell = MHA([\text{LayerNorm}(h_1^{\ell-1}), \ldots, \text{LayerNorm}(h_t^{\ell-1})]).$$

2.4.4 Grouped Query Attention

A small optimization done in some modern Transformers, including the Llama-3 models, is Grouped Query Attention (GQA). The idea is simply that we will group some of the attention heads together, and within each group we will tie some of their weights together. In particular, attention heads in the same group have the same W^K and W^V matrices, and thus have the same key and value vectors. This saves us time, since we only have to compute

the key and value vectors once within each "group." However, they will still have different W^Q matrices, and so they will produce different query vectors and thus different attention distributions and final outputs. Refer to Figure 2.4 for an illustration.

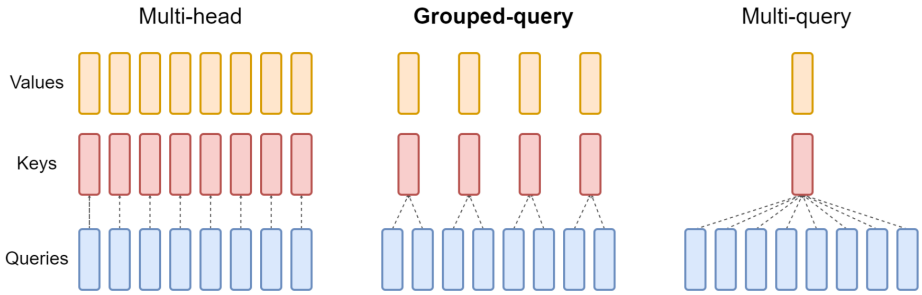

Figure 2.4: Illustration of grouped query attention. Each head has its own query vectors, but some heads share the same key and value vectors. Figure taken from https://arxiv.org/abs/2305.13245.

As an example, Llama-3 8B has 32 total attention heads per layer, but only 8 key/value heads. This means that the 32 attention heads are divided into 8 groups of 4 each; within each group, the key and value vectors are identical, but the query vectors are different.

2.5 Feedforward Network/Multi-Layer Perceptron

2.5.1 Feedforward networks

A **feedforward network** (FFN) layer, also known as a **multi-layer perceptron**, is a neural network layer that maps an input vector $x \in \mathbb{R}^m$ to an output vector $y \in \mathbb{R}^n$, for some fixed dimensions m and n. FFNs are the prototypical neural network, as even a simple two-layer FFN is powerful enough to approximate any function, given a large enough hidden dimension.

Formally, a two-layer FFN is given by the equation

$$y = W^{(2)} \cdot f\left(W^{(1)} \cdot x + b^{(1)}\right) + b^{(2)}$$

where $W^{(1)} \in \mathbb{R}^{h \times m}$, $b \in \mathbb{R}^{d_h}$, $W^{(2)} \in \mathbb{R}^{d_h \times n}$, $b^{(2)} \in \mathbb{R}^n$, and f is an elementwise non-linear function. d_h is a constant dimension often referred to as the hidden dimension. One can see how this could be easily extended to three or more layers.

The non-linearity f, also called the activation function, is crucial for ensuring that the overall output of the FFN y is a non-linear function of the input x. Common choices for f include the sigmoid function $f(z) = \frac{1}{1+e^{-z}}$ and the Rectified Linear Unit (ReLU) function $f(z) = \max(z, 0)$. Note that following common practice, we abuse notation and allow a function $f : \mathbb{R} \to \mathbb{R}$ to be applied to a vector by applying f elementwise (i.e., separately to each element of that vector).

2.5.2 Use in Transformers

A Transformer has one FFN, which we denote FFN^ℓ, for each layer $\ell = 1, \ldots, L$. FFN^ℓ is used at layer ℓ to compute m_t^ℓ from Equation 2.1, for all timesteps t. It is standard to use a two-layer FFN for this purpose. Each FFN has input dimension d and output dimension d. The hidden dimension d_h is a hyperparameter, and may be different from d (for example, in Llama-3 8B, $d = 4096$ and $d_h = 14336$) but is the same for all layers ℓ.

The input to the FFN at layer ℓ is the current value of the residual stream, which is the previous hidden state $h_t^{\ell-1}$ plus the current layer's attention a_t^ℓ. This quantity is first transformed by LayerNorm, then fed to the FFN. Formally, if FFN^ℓ denotes the FFN at layer ℓ, we have

$$m_t^\ell = FFN^\ell(\text{LayerNorm}(h_t^{\ell-1} + a_t^\ell)).$$

Use of biases. In some cases, the bias vectors are omitted from the FFN, as they are viewed as unnecessary. For example, Llama-3 omits bias terms, but BERT and GPT-2 have them.

GLU variants. Some modern Transformers, including Llama-3, use a different type of FFN that has a **gated linear unit** (GLU). GLU variants are different in that the first layer actually applies *two* different linear transformations to the input. These are then elementwise multiplied, with a non-linearity function applied to one of them. For example, Llama-3 uses a particular GLU function known as SwiGLU, which does the following

$$FFN^\ell(x) = W^{(3)} \cdot \left(\text{Swish}(W^{(1)} \cdot x) \odot (w^{(2)} \cdot x)\right).$$

Note that there are now three weight matrices and three matrix-vector multiplications instead of two. Swish is a particular elementwise non-linear function, similar to ReLU:

$$\text{Swish}(z) = z \cdot \sigma(x),$$

where σ is the familiar sigmoid function. Note that ReLU is exactly this but replacing sigmoid with the step function $\mathbb{1}[x > 0]$.

Here is what the SwiGLU paper says on why this works well: "We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence."[2]

2.6 Runtime and Memory Usage

2.6.1 Test time

Suppose we wish to generate a sequence of total length T using a Transformer. To do this, we must first choose the first token w_1, then the next token w_2, and so on until w_T. How expensive is each step? We claim that the t-th step requires $O(td + d^2)$ time and $O(td)$ memory (not counting the parameters themselves).

[2] https://arxiv.org/pdf/2002.05202

The key thing to note is that at each step t, we only have to compute the hidden states for the current timestep, as long as we have stored the hidden states from all previous timesteps.

The two expensive steps are the MHA and FFN layers. Let's analyze the FFN layer first, since it's simpler. The main cost of the FFN is the two matrix-vector multiplications. Each of these takes time $O(d \cdot d_h)$. For simplicity, let's assume that d_h is $O(d)$ (in practice, the FFN hidden dimension does generally scale linearly with the overall Transformer dimension), so this is $O(d^2)$. In terms of memory, we only need $O(d)$ memory (not counting the parameters) to store the intermediate results of the computation.

Now let's look at the MHA layer. It essentially involves two steps. First, the key, query, and value vectors must be computed. Luckily, most of them have been computed before, so we just store then in memory—this requires $O(td)$ memory since it's $O(t)$ vectors of dimension d. This is often referred to as the "KV cache" (for key-vector). Only the key, query, and value for the current token must be newly computed: each of those is a single matrix-vector multiply, which is $O(d^2)$ (as $d_{\text{attn}} \leq d$). Second, we compare the current query vector to every key vector, and then use the derived weights to aggregate across all the value vectors. This step is $O(td)$, since we are doing d-dimensional vector operations on t vectors.

Putting this all together, we see indeed that our runtime is $O(td+d^2)$ for the t-th step and we use $O(td)$ memory (for all the KV caches). Thus, overall the runtime is $\sum_{t=1}^{T} O(td+d^2) = O(T^2d + Td^2)$. The bad news is here is that Transformers have **quadratic runtime** in the length of the sequence. The good news is that d is actually often larger than T (though this can change depending on how long of a context your model was trained to handle), so your runtime might be dominated by the $O(td^2)$ portion anyways. The total memory needed at any given time is at most $O(T)$ (at the last step), i.e., Transformers have **linear memory usage** in the length of the sequence.

2.7 Parameter counting exercise

Let's see if we can re-produce Llama-3 405B's parameter count. They list:

- $L = 126$ layers
- $d = 16,384$ as the model dimension
- $d_h = 53,248$ as the FFN dimension
- $n_{\text{attn}} = 128$ attention heads
- From that, we can infer that $d_{\text{attn}} = 16,384/128 = 128$
- 8 key/value heads (i.e., heads are in 8 groups of $128/8 = 16$)
- Vocabulary size $|V| = 128,000$.

Let's count up the parameters needed.

- Token embeddings: W^{embed} is $128,000 \times 16,384 = 2,097,152,000 \approx \mathbf{2.1B}$

- MHA W^K matrices: Each one is $128 \times 16,384$, there are 8 per layer, and there are 126 layers, so the count is $126 \times 8 \times 128 \times 16384 = 2,113,929,216 \approx \mathbf{2.1B}$.
- MHA W^V matrices: Same computation, so another $\mathbf{2.1B}$ parameters.
- MHA W^Q matrices: Similar to the above, except there are 128 per layer instead of 8, so we have $33,822,867,456 \approx \mathbf{33.8B}$ parameters.
- MHA W^O matrices: We have one per layer, and they are $16,384 \times 16,384$, so 126 of them is $33,822,867,456 \approx \mathbf{33.8B}$ parameters.
- FFN weight matrices: We have three per layer, and they are $16,384 \times 53,248$, so 126×3 of them is $109,924,319,232 \times 3 = 329,772,957,696 \approx \mathbf{329.8B}$ parameters.
- FFN second layer weights: Same size as the first layer, so another $109.9B$ parameters.
- RMSNorm: We do RMSNorm twice per layer, and again at the end, so there's a total of $2 \times 126 + 1 = 253$ of them. Each one has $16,384$ parameters (since it's an elementwise scaling factor), so the total count is $253 \times 16,384 = \mathbf{4,145,152}$ (negligible)
- Unembedding matrix: W^{unembed} is $16,384 \times 128,000$, so the same number of parameters as W^{embed}, or $\approx \mathbf{2.1B}$.

If we count this all up, we have approximately

$$4 \times 2.1B + 2 \times 33.8B + 329.8B = \mathbf{405.8B}.$$

So yes, Llama-3 does have 405 billion parameters (rounding down)!

We notice that the majority of parameters are actually found in the FFN layers ($329.8B/405.8B \approx 81\%$), rather than the attention layers. By comparing the number of parameters for W^Q and W^K, we can see how grouped query attention reduces the number of parameters. The embedding and unembedding layers are also non-trivial in size, whereas RMSNorm includes only a neglible number of parameters.