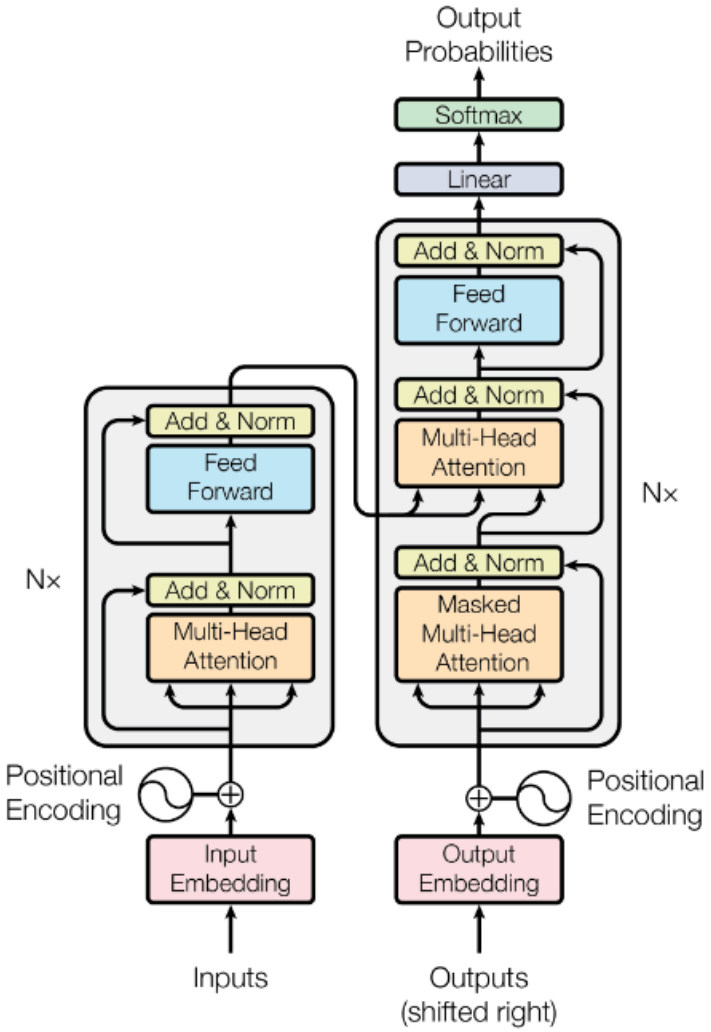
Transformer Language Model Architecture

故事开始于这副“世界名画”.....



但它并不是这次的主角。我们这次的主角是一个 Decoder-only Transformer

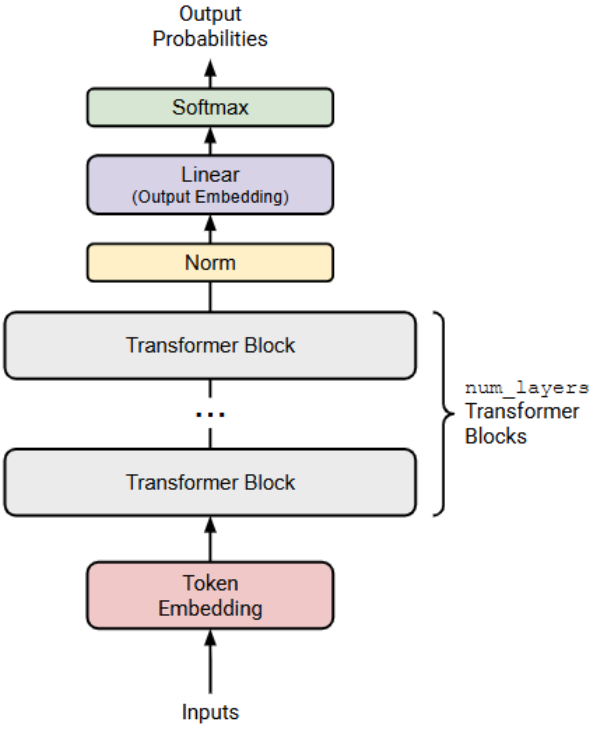


Figure 1: An overview of our Transformer language model.

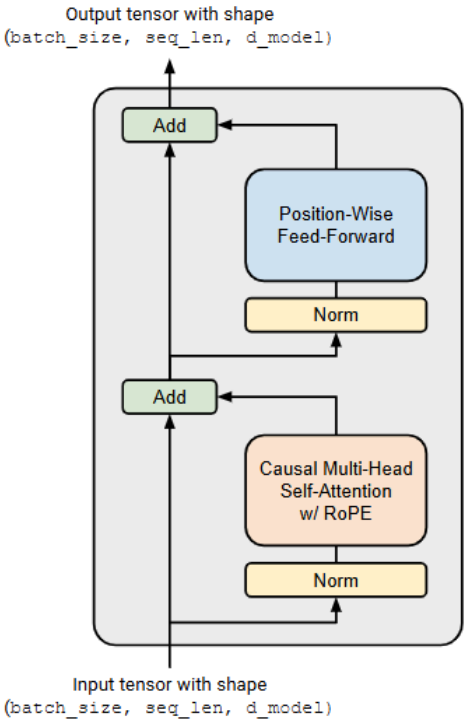


Figure 2: A pre-norm Transformer block.

可以先对比一下最原始的 transformer 中的 decoder 与我们将要实现的 Decoder-only Transformer 有何不同。

上面的第二张图便是我们需要搭建的框架，在实现它的具体细节前，先从较为宏观的层面了解一下它的工作原理。

0.0.1 输入与输出

Transformer 语言模型本质上是一个概率分布预测机。

- **输入 (Input):** 一个批次（Batch）的整数 Token ID 序列。
  - 数据格式：PyTorch Tensor。
  - 形状 (Shape)：`(batch_size, sequence_length)` 。
- **输出 (Output):** 针对词表中每个词的归一化概率分布。
  - **关键点：** 模型是对输入序列中的**每一个** Token，预测它的**下一个**词是什么 。
  - 形状 (Shape)：`(batch_size, sequence_length, vocab_size)` 。

0.0.2 两种模式：训练与生成

模型在训练（Training）和推理（Inference/Generation）时的不同用途：

- **训练时 (Training):**
  - 我们利用模型预测的“下一个词”分布，与真实的“下一个词”进行比较。
  - 使用 **交叉熵损失 (Cross-entropy loss)** 来计算误差并更新模型 。
- **生成时 (Inference):**
  - 这是一个循环过程。
  - 模型根据当前序列预测最后一个时间步（Time step）的下一个词分布 -> 采样选出一个词 -> 把这个新词加到输入序列末尾 -> 重复上述过程 。

1 Transformer LM

1.1 Pipeline

数据在模型中的流动过程如下：

1. **Input Embedding:** 将整数 Token ID 转换为稠密向量 (Dense Vectors)。
2. **Transformer Blocks:** 经过 `num_layers` 层堆叠的 Transformer 块（这是模型最厚重、计算量最大的部分）。
3. **Output Embedding (LM Head):** 一个学习到的线性投影层（Linear Projection），将向量重新映射回 Logits（即未归一化的概率分数），用于预测下一个 Token。

1.2 Token Embeddings

作用： 将离散的整数 Token IDs 转换为包含 Token 信息的连续向量 。

维度变化：

- 输入：`(batch_size, sequence_length)` (整数)
- 输出：`(batch_size, sequence_length, d_model)` (浮点数向量) 。
- *注：`d_model` 是模型的隐藏层维度，例如 GPT-2 Small 中是 768。*

1.3 Pre-norm Transformer Block

结构： 在我们要实现的 Transformer 模型中由 `num_layers` 个结构完全相同的“块”（Block）堆叠而成 。

输入输出同构： 每个块的输入和输出形状完全一致，都是 `(batch_size, sequence_length, d_model)` 。这使得我们可以像搭积木一样无限堆叠它们。

内部机制： 每个块主要做两件事：

1. **聚合信息 (Aggregate):** 通过 **Self-Attention**（自注意力机制）让序列中的 Token 互相交换信息。
2. **非线性变换 (Transform):** 通过 **Feed-Forward**（前馈网络）对每个位置的信息进行加工。

Pre-norm（预归一化）：，这是指 Layer Normalization 放在子层输入之前，而不是输出之后

> 可以对比原始的 transformer 和我们要实现的 transformer，原始的架构是先做 attention 和残差连接计算，再进行 layer norm。而我们要实现的架构是先进行 norm 再去进行 attention 与残差连接；这种结构就叫做 pre-norm。

1.4 Output Normalization and Embedding

在经过 `num_layers` 层的 transformer blocks 后我们还需要做一次额外的 normalization，再将其进行一次线性变换（具体原因后续介绍）。

维度变化：

- **操作：** 乘以形状为 `(d_model, vocab_size)` 的权重矩阵。
- **输出：** `(batch_size, sequence_length, vocab_size)`

产物 - Logits： 这一层的输出被称为 **Logits**（未归一化的对数概率）。

---

2 Batching, Einsum and Efficient Computation

这一节介绍一种高效的矩阵运算方法。

2.1 为什么要用 Einsum？

在深度学习（尤其是 Transformer）中，我们处理的不是二维矩阵，而是多维张量（Tensors）。

传统方法的痛苦：

假设你有一个数据 D 形状是 (batch_size, sequence_length, d_model)，你想和一个权重矩阵 A (d_model, d_out) 做乘法。

在老式的 PyTorch 写法中，你必须在大脑中通过空间想象力来“扭转”这些数据：

1. 先把 D 压扁成二维。
2. 做乘法。
3. 再把结果变回三维。

   代码写出来是这样的：D.view(-1, d_model).mm(A).view(batch_size, seq_len, d_out)。

   如果你过两个月再看这行代码，你很难直接看出它在做什么，也很容易写错维度。

Einsum 的解法：

`einsum` (爱因斯坦求和约定) 允许你直接**用字符串描述你想要的数学操作**，而不必关心底层怎么搬运数据。它是“自文档化”的（Self-documenting）。

2.2 具体案例

案例 1：批量矩阵乘法 (Batched Matrix Multiplication)

这是 Transformer 中最常见的操作（例如计算 Attention 里的 Q, K, V）。

- **场景**：
  - 输入 D：形状是 `(batch, sequence, d_in)`
  - 权重 A：形状是 `(d_in, d_out)`
- **目标**：让 D 的最后一个维度和 A 做乘法，同时保持 `batch` 和 `sequence` 不变。
- **代码**：

```
Y = einsum(D, A, "batch sequence d_in, d_in d_out -> batch sequence d_out")
```

- **解析**：
  - `batch` 和 `sequence`：在箭头左右都存在 -> 保留（不做改变，原样输出）。
  - `d_in`：在 D 和 A 中都存在，但在输出中消失了 -> 求和/消掉（这是矩阵乘法的连接轴）。

- **优势**：你不需要先把 `batch` 和 `sequence` 合并起来，代码直接告诉了读者维度的变化逻辑。

案例 2：使用 `einops.rearrange` 进行广播 (Broadcasting)

`einops` 是一个专门配合 `einsum` 的库，`rearrange` 用来替代 `view` 和 `reshape`。

- **场景**：有一批图片 `images` `(batch, height, width, channel)`，还有一个调节亮度的向量 `dim_by`（长度为 10）。你想把这 10 个亮度值分别应用到每一张图片上，生成 10 个变暗的版本。
- **难点**：需要把这两个张量的维度对齐才能相乘。
- **代码**：

```
# 先调整 dim_by 的形状，让它变成 (1, 10, 1, 1, 1) 这种形式以便广播
dim_value = rearrange(dim_by, "dim_value -> 1 dim_value 1 1 1")

# 或者直接用 einsum 一步到位：
dimmed_images = einsum(
    images, dim_by,
    "batch h w c, dim_val -> batch dim_val h w c"
)
```

- **解析**：
  - 输入1 (images): `batch h w c`
  - 输入2 (dim_by): `dim_val`
  - 输出: `batch dim_val h w c`
  - Einsum 自动理解为：把 `dim_val` 插入到 `batch` 后面，并对其他维度进行广播（复制）以匹配形状。

案例 3：像素混合 (Pixel Mixing)

- **场景**：对图像进行线性变换，但是**独立地对每个通道（Channel）**进行处理。
- **数据**：
  - `input`: `(batch, height, width, channel)`
  - 变换矩阵 B: `(pixel_in, pixel_out)`，其中 `pixel_in = height * width`。
- **传统做法**：需要先把 `height` 和 `width` 展平，然后把 `channel` 移到前面去，乘完再移回来。这非常容易出错。
- **Einsum/Einops 做法**：
  1. **Rearrange (准备数据)**：先把高和宽合并，把通道独立出来。

```
# "把 h 和 w 合并成一个维度，叫 pixel_in"
input_flat = rearrange(input, "b h w c -> b c (h w)")
```

  2. **Einsum (核心运算)**：

```
# 这里的 pixel_in 就是 (h w)
output_flat = einsum(
    input_flat, B,
    "batch channel pixel_in, pixel_out pixel_in -> batch channel pixel_out"
)
```

  - `batch`, `channel`: 保留。
  - `pixel_in`: 匹配并求和（矩阵乘法）。
  - `pixel_out`: 新的像素维度。

2.3 效率问题

我们为什么要用 `einsum` 方法而不是 `view/transpose` 呢?

首先，文档中指出 `einsum` 方法和传统方法比，在运算速度上的差距微乎其微。

真正的“快”体现在哪里? 传统的 `view` + `transpose` 写法是“笨重且容易出 Bug 的。

- **View 的风险**：使用 `.view()` 时，你需要在大脑里极其精准地计算维度。一旦算错或者内存顺序（Memory Ordering）不对，PyTorch 可能不会报错，但你的数据就乱了（比如把行变成了列），模型训练出来就是错的。
- **Einsum 的优势**：它是“自文档化”的 (self-documenting) 。代码本身就写明了维度的变化逻辑。