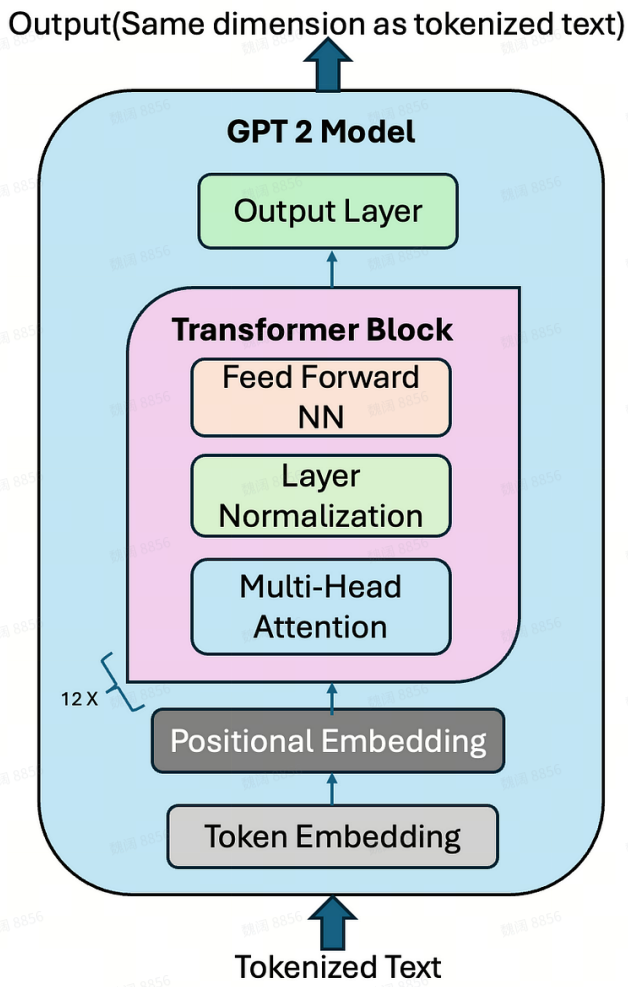
【手搓大模型】从零手写GPT2 — Model

GPT2 Model Architecture

我们已经实现了Embedding和Multi-Head Attention，现在开始实现完整的gpt2。gpt2的整体架构如下图所示：



其中：

1）最重要的Transformer Block重复堆叠了12次。

2）Transformer Block包含了MHA、Layer Norm和FeedFroward Neutral Network.

3）MHA内含12个多头。

如前所述，我们还知道GPT2的Vocab Size是50257，最大seql length是1024，嵌入的维度是768。

综上，给出GPT2的model配置如下：

代码块

```
GPT_CONFIG_124M = {
```

```
    "vocab_size": 50257,    # Vocabulary size
    "context_length": 1024, # Context length
    "emb_dim": 768,         # Embedding dimension
    "n_heads": 12,          # Number of attention heads
    "n_layers": 12,         # Number of layers
    "drop_rate": 0.1,       # Dropout rate
    "qkv_bias": False       # Query-Key-Value bias
}
```

Dummy model

现在我们开始按照GPT2的上述模型架构，先搭建出骨架，即先给出整体的dummy实现，如下：

代码块

```
import torch
import torch.nn as nn

class DummyGPT2(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.token_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop = nn.Dropout(cfg["drop_rate"])

        self.blocks = nn.Sequential(*[DummyTransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = DummyLayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(cfg["emb_dim"], cfg["vocab_size"], bias=False)

    def forward(self, token_ids):
        batch_size, seq_len = token_ids.shape
        token_emb = self.token_emb(token_ids)
        pos_emb = self.pos_emb(torch.arange(seq_len, device=token_ids.device))
        x = token_emb + pos_emb
        x = self.drop(x)
        x = self.blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits

class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
```

```
30
31      def forward(self, x):
32          return x
33
34  class DummyLayerNorm(nn.Module):
35      def __init__(self, cfg):
36          super().__init__()
37
38      def forward(self, x):
39          return x
```

在上述代码中，我们用DummyTransformerBlock作为假实现，里面是空操作。不过整体的架子都是有的，后续只需要继续实现dummy部分即可。

其实，上面的dummy代码，依然是可以运行的，只是结果意义不大。不过我们依然可以运行下，看到最终输出的结构。

运行示例如下，我们先手动构建2个batch、context为5的input token IDs：

代码块

```
1  import torch
2  import tiktoken
3
4  tokenizer = tiktoken.get_encoding("gpt2")
5
6  texts = ["Once upon a time there", "were four little Rabbits"]
7  batch = torch.stack([torch.tensor(tokenizer.encode(t)) for t in texts])
8  print(batch)
```

```
tensor([[ 7454,  2402,   257,   640,   612],
        [22474,  1440,  1310, 22502,   896]])
```

我们再把上述token id输入模型：

代码块

```
1  torch.manual_seed(123)
2  model = DummyGPT2(GPT_CONFIG_124M)
3
4  logits = model(batch)
5  print("Output shape:", logits.shape)
6  print(logits)
```

```
Output shape: torch.Size([2, 5, 50257])
```

```
tensor([[[-0.1453, -0.5939,  0.3767,  ...,  0.4361,  0.3913,  1.1740],
         [ 0.2646,  0.5527, -1.0897,  ...,  0.3165,  0.7068,  1.9168],
         [-0.2009, -0.7217,  0.7162,  ...,  0.6297,  0.6221, -0.1177],
         [ 0.1959,  0.4116,  1.1859,  ...,  2.2309,  0.2540,  0.7609],
         [-0.4772, -0.7713,  0.6711,  ...,  0.9593, -1.1426, -1.0256]],

        [[-0.7387,  0.2473, -2.2699,  ..., -0.9243, -1.1297,  0.1037],
         [-0.5791,  1.0997, -0.4741,  ..., -0.7711,  0.9321,  1.0572],
         [ 0.7911,  1.0512,  0.4935,  ...,  0.8441, -0.2399, -0.5090],
         [ 1.1721,  0.9144, -0.7984,  ...,  1.6035,  0.5685,  1.0169],
         [-1.0692, -1.7418,  0.1271,  ...,  0.1854, -0.5162, -0.7783]]],
       grad_fn=<UnsafeViewBackward0>)
```

得到的是(batch, seq_len, vocab_size)维度的向量。输出后续经过处理后，代表的是结果在50257个字典中每个词的可能性，因此输出的维度的最后一维必须是vocab_size。

LayerNorm

在上面的dummy model中，我们还预留了DummyLayerNorm，现在开始实现。其实LayerNorm的思想非常简单，就是把样本特征归一化为均值0、方差1。

实现如下：

代码块

```python
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift
```

非常简单，根据统计常识，就是输入X减去均值再除以标准差；只是引入了超小常数eps防止被除数为0。

运行如下示例：

```
torch.manual_seed(123)

batch_example = torch.randn(2, 5)
ln = LayerNorm(emb_dim=5)
out = ln(batch_example)
print(out)
mean = out.mean(dim=-1, keepdim=True)
var = out.var(dim=-1, unbiased=False, keepdim=True)
print("mean:\n", mean)
print("var:\n", var)
```

```
tensor([[ 0.5528,  1.0693, -0.0223,  0.2656, -1.8654],
        [ 0.9087, -1.3767, -0.9564,  1.1304,  0.2940]], grad_fn=<AddBackward0>)
mean:
 tensor([[-2.9802e-08],
        [ 0.0000e+00]], grad_fn=<MeanBackward1>)
var:
 tensor([[1.0000],
        [1.0000]], grad_fn=<VarBackward0>)
```

可见layernorm之后，得到的mean是0，var是1。

其实LayerNorm在LLM中非常常见，实现也非常简单，我们后续可以直接使用pytorch自带的，如下：

```
torch.manual_seed(123)

batch_example = torch.randn(2, 5)

layer = nn.LayerNorm(5)
out = layer(batch_example)
print(out)

mean = out.mean(dim=-1, keepdim=True)
var = out.var(dim=-1, unbiased=False, keepdim=True)
print("mean:\n", mean)
print("var:\n", var)
```

```
tensor([[ 0.5528,  1.0693, -0.0223,  0.2656, -1.8654],
        [ 0.9087, -1.3767, -0.9564,  1.1304,  0.2940]],
       grad_fn=<NativeLayerNormBackward0>)
mean:
 tensor([[-3.5763e-08],
        [ 2.3842e-08]], grad_fn=<MeanBackward1>)
var:
 tensor([[1.0000],
        [1.0000]], grad_fn=<VarBackward0>)
```

结果跟上面相同。后续可直接使用pytorch自带的nn.LayerNorm()类；需要注意的是这并不是函数，而是pytorch中的层/类，很方便利用nn.Sequential进行层间堆叠。

Activations: Relu, GELU, SwiGLU

在神经网络中，除了前述线性变换（如矩阵投影、MHA、归一化等）之外，还需要引入非线性激活，以增强网络的表达能力。

可以直观地理解：

1）所有的线性变换，本质上都是矩阵操作，保持的都是线性结构。

2）线性变换的目的是进行不同空间之间的映射。

3）线性变换可以写成矩阵乘法形式y=W⊠x+b，具有加法封闭性和缩放封闭性。

4）如旋转、缩放、投影、剪切等，都属于线性变换。线性叠加后依然是线性的。

但是线性无法引入弯曲、拐点和门控等机制，从理论上无法拟合所有函数。

而为了能表达任意复杂函数，只需要引入看似非常简单的非线性激活函数。

这背后有比较严格的数学理论（Universal approximation theorem），其简化思想大意是：

> **一个包含至少一层非线性激活函数的前馈神经网络，只要隐藏层的神经元足够多，就可以逼近任意连续函数（在紧致区间上），误差可以小到任意程度。**

也就是说只要给模型在线性变换的基础上，来一点点非线性，从理论上来讲，只要模型足够深，可以表达任意复杂函数。

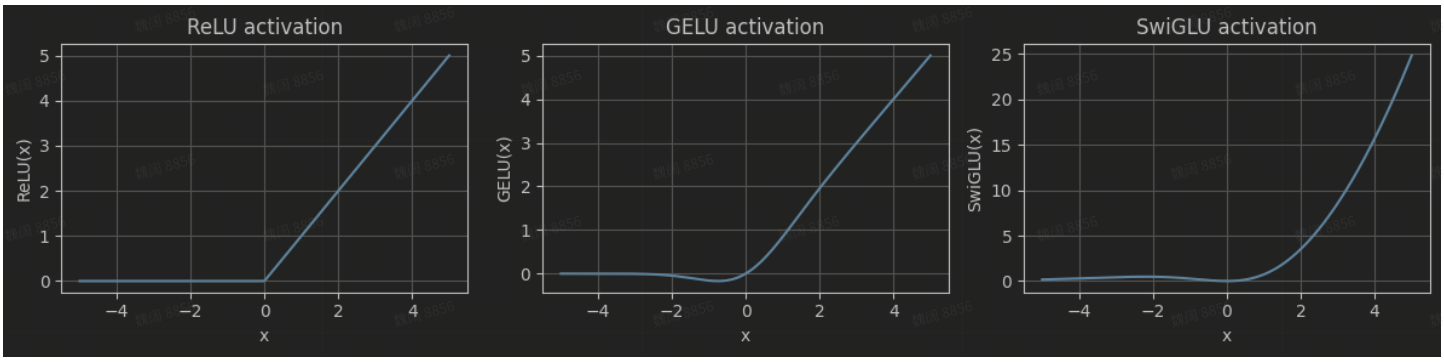
然而，非线性激活函数，其实非常简单，最简单的哪怕只是一段折线，其实就可以作为非线性激活函数。

我们可以直接给出代码，看图说话：

代码块

```
1  import torch
```

```python
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt

class SwiGLU_Simple(nn.Module):
    def forward(self, x):
        return x * F.silu(x)

gelu = nn.GELU()
relu = nn.ReLU()
swiglu = SwiGLU_Simple()

x = torch.linspace(-5, 5, 200)

y_gelu = gelu(x)
y_relu = relu(x)
y_swiglu = swiglu(x)

plt.figure(figsize=(12, 3))
for i, (y, label) in enumerate(zip(
    [y_relu, y_gelu, y_swiglu], ["ReLU", "GELU", "SwiGLU"]), 1):
    plt.subplot(1, 3, i)
    plt.plot(x.numpy(), y.detach().numpy(), label=label)
    plt.title(f"{label} activation")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)

plt.tight_layout()
plt.show()
```



可见，最简单地就是ReLU，其实就是初中学的分段函数。而现在更常用的GELU和SwiGLU，只不过是增加了一些曲度，让其更平滑，方便计算梯度。具体的函数定义很boring，这里不赘述。后续我们会使用GELU。

FeedForward Network

GPT的模型架构中，还有非常重要的前馈神经网络FFN层，如下。

$$\mathrm{FFN}(x) = \mathrm{Linear}_2(\ \mathrm{Activation}(\ \mathrm{Linear}_1(x)\ )\ )$$

也就是对于输入 x → 线性变换 → 非线性激活（ReLU/GELU）→ 再次线性变换 → 输出。

说白了，就是在两个线性层中间，夹带一个非线性层，以增强模型的表达能力。通常来说，需要先在第一个线性层升维，做非线性激活，然后再降维，回到最初的维度。

我们直接看代码示例：

代码块

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(nn.Linear(cfg["emb_dim"], 4*cfg["emb_dim"]), nn.GELU(), nn.Linear(4*cfg["emb_dim"], cfg["emb_dim"]))

    def forward(self, x):
        return self.layers(x)

print("model structure: \n",FeedForward(GPT_CONFIG_124M))
```

```
model structure:
 FeedForward(
  (layers): Sequential(
    (0): Linear(in_features=768, out_features=3072, bias=True)
    (1): GELU(approximate='none')
    (2): Linear(in_features=3072, out_features=768, bias=True)
  )
)
```

可见，在上面例子中，我们先从768维，扩大4倍到3072维；然后做GeLU激活操作；再降回到768维。

ShortCut Connections

神经网络的优化依赖梯度计算，可以说梯度是神经网络训练的发动机。当现代神经网络层数越来越惊人的时候，比较大的现实工程难题是梯度爆炸与梯度消失；相信大家并不陌生。

反向传播Backpropagation是神经网络中计算梯度的核心算法，其思想也非常简单，依然是利用求导的链式法则，逐层传递，计算梯度。有兴趣可以参考karpathy关于autograd的介绍，这里不再赘述。

自动微分也可以说是pytorch的灵魂，我们可以直接通过示例，感受下计算过程中的梯度变化，代码如下：

代码块

```
import torch
import torch.nn as nn

class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList(nn.Sequential(nn.Linear(layer_sizes[i], layer_sizes[i+1]), nn.GELU()) for i in range(len(layer_sizes)-1))
    def forward(self, x):
        for i, layer in enumerate(self.layers):
            out = layer(x)
            if self.use_shortcut and x.shape[-1] == out.shape[-1]:
                x = x + out
            else:
                x = out
        return x

def print_gradients(model,x):
    output = model(x)
    target = torch.zeros_like(output)
    loss = nn.MSELoss()(output, target)
    loss.backward()

    for name, param in model.named_parameters():
        if param.grad is not None and 'weight' in name:
            print(f"{name} has gradient mean of {param.grad.abs().mean().item()}")
```

在上述代码中，我们定义了多个Linear+Gelu的堆叠，而在forward中，最简单的是直接返回layer(x)。我们给模型一个模拟输入，生成1 x 4的张量作为输入，看下每层的梯度如何，如下：

代码块

```
layer_sizes = [4] * 6

x = torch.randn(1, 4)

torch.manual_seed(123)
model = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)
print(model)
print_gradients(model, x)
```

```
ExampleDeepNeuralNetwork(
  (layers): ModuleList(
    (0-4): 5 x Sequential(
      (0): Linear(in_features=4, out_features=4, bias=True)
      (1): GELU(approximate='none')
    )
  )
)
layers.0.0.weight has gradient mean of 3.108993041678332e-05
layers.1.0.weight has gradient mean of 7.357167487498373e-05
layers.2.0.weight has gradient mean of 0.0006941530737094581
layers.3.0.weight has gradient mean of 0.005131533369421959
layers.4.0.weight has gradient mean of 0.014868268743157387
```

可见，示例网络总共有5层；我们注意到layers0的梯度特别小3e-5，已经接近消失。我们重点关注前几层，因为依据反向传播的原理，梯度是从最后一层往前开始推算的，所以最容易出问题的是模型的前面的层。

现在我们打开use_shortcut=True，再次运行，结果如下：

```
layers.0.0.weight has gradient mean of 0.06876879185438156
layers.1.0.weight has gradient mean of 0.15942829847335815
layers.2.0.weight has gradient mean of 0.12936799228191376
layers.3.0.weight has gradient mean of 0.13758598268032074
layers.4.0.weight has gradient mean of 0.173927441239357
```

可见，layers0的梯度神奇地增大到了6e-2；而唯一的变化是上述forward中的返回值从layer(x)变成了x+layer(x)。

这便是shortcut的神奇之处，看起来只是在原输出F(x)的基础上，又加上了输入x，如下：

$$y = F(x) + x$$

隐含的意思是，此时F(x)代表了真正的输出y与输入x之间的差异，因此又被称为残差网络ResNet。

现在回头看确实非常简单，但是站在当时的时间节点，能够率先想到，并意识到其背后的意义，其实并不简单。

直觉上理解，shortcut相当于给非常深的神经网络的不同层之间，增加了新的通路，允许信息跨层流动；这个“跨层通道”，让信息和梯度都可以跳跃式传播，从而提升训练稳定性与效率。

Transformer Code

有了上述的实现，我们可以直接结合起来，给出Transformer的真实代码，如下：

代码块

```
import torch
import torch.nn as nn

class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        assert d_out % num_heads == 0, "d_out must be divisible by num_heads"
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads

        self.W_Q = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_K = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_V = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_O = nn.Linear(d_out, d_out)
        self.dropout = nn.Dropout(dropout)

        mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
        self.register_buffer("mask", mask.bool())

    def forward(self, x):
        # shape (batch_size, seq_len, d_in)
        batch_size, seq_len, _ = x.size()

        # Split Q, K, V into multiple heads
        # (batch_size, seq_len, d_in) -> (batch_size, seq_len, d_out) ->
        # -> (batch_size, seq_len, num_heads, head_dim) -> (batch_size, num_heads, seq_len, head_dim)
        Q = self.W_Q(x).view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
        K = self.W_K(x).view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
        V = self.W_V(x).view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)

        # Compute attention scores
        scores = Q @ K.transpose(-2, -1) / (self.d_out ** 0.5)  # (batch_size, num_heads, seq_len, seq_len)

```

```
        # Apply causal mask
        scores = scores.masked_fill(self.mask[:seq_len, :seq_len], -torch.inf)

        # Compute softmax weights and apply dropout
        weights = torch.softmax(scores, dim=-1)
        weights = self.dropout(weights)

        # Compute output
        output = weights @ V  # (batch_size, num_heads, seq_len, head_dim)
        # Concatenate heads and project to output dimension
        # (batch_size, num_heads, seq_len, head_dim) -> (batch_size, seq_len, num_heads, head_dim)
        # ->   (batch_size, seq_len, d_out)
        output = output.transpose(1, 2).contiguous().view(batch_size, seq_len, -1)
        # Should be helpful, but not strictly necessary.
        output = self.W_O(output)
        return output
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(nn.Linear(cfg["emb_dim"], 4*cfg["emb_dim"]), nn.GELU(), nn.Linear(4*cfg["emb_dim"], cfg["emb_dim"]))

    def forward(self, x):
        return self.layers(x)
class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.attn = MultiHeadAttention(cfg["emb_dim"], cfg["emb_dim"], cfg["context_length"], cfg["drop_rate"], cfg["n_heads"], cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.ln1 = nn.LayerNorm(cfg["emb_dim"])
        self.ln2 = nn.LayerNorm(cfg["emb_dim"])
        self.dropout = nn.Dropout(cfg["drop_rate"])
    def forward(self, x):
        x = x + self.dropout(self.attn(self.ln1(x)))
        x = x + self.dropout(self.ff(self.ln2(x)))
        return x
```

其中MHA代码来自Attention模块。

我们可以直接调用，检查下输出，如下：

代码块

```
torch.manual_seed(123)
```

```
x = torch.rand(2, 4, 768)  # Shape: [batch_size, num_tokens, emb_dim]
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)

print("Input shape:", x.shape)
print("Output shape:", output.shape)
```

> Input shape: torch.Size([2, 4, 768])
>
> Output shape: torch.Size([2, 4, 768])

可见，经过Transformer的一系列操作，最终输出的维度和输入是完全相同的。

GPT-2 code

我们再把Transformer堆叠12次，就得到了完整的GPT-2代码。

代码块

```
import torch
import torch.nn as nn

class GPT2Model(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.token_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop = nn.Dropout(cfg["drop_rate"])

        self.blocks = nn.Sequential(*[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = nn.LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(cfg["emb_dim"], cfg["vocab_size"], bias=False)

    def forward(self, token_ids):
        batch_size, seq_len = token_ids.shape
        token_emb = self.token_emb(token_ids)
        pos_emb = self.pos_emb(torch.arange(seq_len, device=token_ids.device))
        x = token_emb + pos_emb
        x = self.drop(x)
        x = self.blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

我们依然可以给模型模拟输入，看下结果，如下

```
torch.manual_seed(123)
model = GPT2Model(GPT_CONFIG_124M)

out = model(batch)
print("Input batch:\n", batch)
print("Output shape:", out.shape)
```

```
Input batch:
 tensor([[ 7454,  2402,   257,   640,   612],
        [22474,  1440,  1310, 22502,   896]])
Output shape: torch.Size([2, 5, 50257])
```

可见，输出的维度的最后一维，依然是等于vocab size。这里我们输出的是原始的logits，尚未经过softmax变换。softmax后续会详细讲，只是相当于做了概率归一化。

Model Overview

至此，GPT-2的代码已经构建完成。让我们审视一下模型的细节，这里我们引入torchinfo包，只需summary一下，就能看到模型架构和参数，如下：

```
from torchinfo import summary

summary(model)
```

```
==========================================================================
Layer (type:depth-idx)                   Param #
==========================================================================
GPT2Model                                --
├─Embedding: 1-1                         38,597,376
├─Embedding: 1-2                         786,432
├─Dropout: 1-3                           --
├─Sequential: 1-4                        --
│    └─TransformerBlock: 2-1             --
│    │    └─MultiHeadAttention: 3-1      2,360,064
```

```
│    │    └─FeedForward: 3-2          4,722,432
│    │    └─LayerNorm: 3-3            1,536
│    │    └─LayerNorm: 3-4            1,536
│    │    └─Dropout: 3-5              --
│    └─TransformerBlock: 2-2          --
│    │    └─MultiHeadAttention: 3-6   2,360,064
│    │    └─FeedForward: 3-7          4,722,432
│    │    └─LayerNorm: 3-8            1,536
│    │    └─LayerNorm: 3-9            1,536
│    │    └─Dropout: 3-10             --
.....(省略TransformerBlock2-3到2-11以节省篇幅)
│    └─TransformerBlock: 2-12         --
│    │    └─MultiHeadAttention: 3-56  2,360,064
│    │    └─FeedForward: 3-57         4,722,432
│    │    └─LayerNorm: 3-58           1,536
│    │    └─LayerNorm: 3-59           1,536
│    │    └─Dropout: 3-60             --
├─LayerNorm: 1-5                      1,536
├─Linear: 1-6                         38,597,376
==========================================================================
Total params: 163,009,536
Trainable params: 163,009,536
Non-trainable params: 0
==========================================================================
```

可见，模型的层次是：

1）总体结构：Token Embedding -> Position Embedding -> Dropout -> Transformer * 12 -> LayerNorm -> Linear；

2）Transformer结构： LayerNorm -> MHA -> Dropout-> LayerNorm -> FeedForward -> Dropout；这里的顺序summary的不太准确，可以代码为准。其中Dropout是可选的。

并且也可以看到模型的总参数是163M。

其实，我们也可以用pytorch自带函数方便地计算模型参数，如下：

代码块

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Total number of parameters: {total_params:,}")

print("Token embedding layer shape:", model.token_emb.weight.shape)
print("Output layer shape:", model.out_head.weight.shape)

```

```
total_params_gpt2 =  total_params - sum(p.numel() for p in model.out_head.parameters())
print(f"Number of trainable parameters considering weight tying: {total_params_gpt2:,}")

total_size_mb = total_params * 4 / (1024 ** 2)
print(f"Total size of the model: {total_size_mb:.2f} MB")
```

> Total number of parameters: 163,009,536
>
> Token embedding layer shape: torch.Size([50257, 768])
>
> Output layer shape: torch.Size([50257, 768])
>
> Number of trainable parameters considering weight tying: 124,412,160
>
> Total size of the model: 621.83 MB

总参数同上，其中除了out_head之外的参数是124M；通常会去除out_head，原因是在gpt2中采用了共享参数，最后一层使用的out_head的权重tensor，其实就是直接用的token_embedding的tensor；所以可以去重这部分重复参数。因此，模型的可训练总参数是124M；总模型大小是621M；但其实核心代码仅有100行。

Generate Text

上面的gpt2代码虽然未经训练，其实结构是完整的，我们可以直接拿来测试下文本生成。当然，我们预期会生成词不达意的乱码水平。不过，测试输出，可以帮助我们检查代码问题，如下：

代码块

```
def generate_text_simple(model, idx, max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]

        # Get logits from model
        with torch.no_grad():
            logits = model(idx_cond)

        # Take logits for the last time step
        # (batch, n_tokens, vocab_size) -> (batch, vocab_size)
        logits = logits[:, -1, :]

        # Apply softmax to get probabilities
        probas = torch.softmax(logits, dim=-1)  # (batch, vocab_size)

        # Get the idx of the vocab entry with the highest probability value
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)  # (batch, 1)

```

```
        # Append sampled index to the running sequence
        idx = torch.cat((idx, idx_next), dim=1)  # (batch, n_tokens+1)

    return idx
```

在上面代码中，我们首先torch.no_grad()把torch的模式设置为evaluation阶段，而非训练阶段，避免自动计算梯度带来的额外开销。

并且，每次只生成1个单词，因此我们每次只从生成的(batch, n_tokens, vocab_size)中提取最后一个token对应的50257维，即变成(batch, vocab_size)；我们把logits做softmax变换，然后从中挑出概率最大的值对应的id；这个id其实就是生成单词对应的token_id；其实这里softmax是多余的，即便只看原始的logits，从50257的维度中挑出最大的值对应的编号，也就是生成的单词对应的tokenId。因为softmax只是做了概率归一化，其实我们不关心值的大小，我们只想找出值最大的id即可。

我们再实现下token_id张量到text的转换，就是Embedding章节提到的tokenizer encode与decode，如下：

代码块

```
def text_to_tensor(text,tokenizer):
    return torch.tensor(tokenizer.encode(text)).unsqueeze(0)

def tensor_to_text(tensor,tokenizer):
    return tokenizer.decode(tensor.squeeze(0).tolist())
```

我们直接运行示例，让模型生成看看：

代码块

```
start_context = "Once upon a time there"

encoded_tensor = text_to_tensor(start_context,tokenizer)
print("encoded_tensor.shape:", encoded_tensor.shape)
print("encoded_tensor:", encoded_tensor)

model.eval() # disable dropout

out = generate_text_simple(
    model=model,
    idx=encoded_tensor,
    max_new_tokens=6,
    context_size=GPT_CONFIG_124M["context_length"]
)

print("Output:", out)
decoded_text = tensor_to_text(out,tokenizer)
```

```
18    print(decoded_text)
```

我们输入了仅有1个batch、长度为5的的文本，先转换为token ids；并把模型改为eval模式，输入到模型中。我们设置了最多生成6个new token。

结果如下：

> encoded_tensor.shape: torch.Size([1, 5])
>
> encoded_tensor: tensor([[7454, 2402, 257, 640, 612]])
>
> Output: tensor([[7454, 2402,   257,   640,   612, 41117, 4683, 36413, 33205, 35780,
>
> 22580]])
>
> Once upon a time there discriminated existing REALLY JehovahQUEST valve

可见，model确实生成了6个新的token，不过看起来在胡言乱语。毕竟，这是未经过训练的模型，所有的权重还都是初始值。在这里，我们依然可以看到，虽然大家都知道模型训练成本极高，但其实eval推理模式的模型生成，成本还是比较低的。

至此，我们已经完成了gpt2模型的完整构建，只是目前尚不具备智能，但是已经具备了智慧体的必要连接和通路。相当于虽然只是一个婴儿，但是脑神经通路是完好的，后续经过训练，打通神经连接，就可以具备智能。