【手搓大模型】从零手写GPT2 — Embedding

系列前言

> What I cannot create, I do not understand.
>
> —— *Richard Feynman*

- 理解大模型最好的方式，应该是亲自动手、从零开始实现。大模型之大在于参数（动辄几十B），而不在于代码量（即便很强大的模型也不过几百行代码）。这样我们便可以在动手写代码中，去思考问题、发现问题、解决问题。
- 本文不深究背后原理，提供尽可能简单的实现，以便整体理解大模型。
- 参考Sebastian Raschka和Andrej Karpathy的教程，并进行重新组织，并对核心代码做了优化，使之更简单更清晰。
- 零基础，具备基本的Python技能，了解Pytorch和Tensor的基本操作。
- 资源：所有代码均运行在个人电脑上，无需GPU。使用数据均为公开数据集。
- 系列文章：将会分为以下5篇
  - **【手搓大模型】从零手写GPT2 — Embedding**：介绍如何从text到token，再到vector；理解BPE的思想；会用滑动窗口取样；理解Embedding的本质是查表操作；理解位置编码。
  - **【手搓大模型】从零手写GPT2 — Attention**：理解注意力机制，Mask遮蔽未来词，Dropout随机丢弃，实现单一与多头注意力机制。
  - **【手搓大模型】从零手写GPT2 — Model**：构建GPT2的完整骨架，理解LayerNorm和Relu激活，实现Transformer Block；使用未训练的GPT2补全文本。
  - **【手搓大模型】从零训练GPT2:** 理解Cross-Entropy，实现在数据集和批量上计算Loss；实现训练代码，并在超小数据集上训练；实现decode控制随机性的方式，包括temperature和top k；尝试在更大数据集上训练，并学会save和load模型参数。
  - **【手搓大模型】从零微调GPT2:** 实现手动load公开模型权重；利用超小数据集微调GPT2，让GPT2学会响应指令，而不是补全文本；利用本地运行llama3评估训练效果。

大模型的本质是空间映射（Mapping Between Spaces）与空间优化（Optimization in Latent Space）。大模型的代码，本质是把输入空间映射到输出空间的函数近似器（function approximator）；而大模型不是直接编程实现规则，而是通过大规模训练过程，在参数空间（weights）中搜索最优解，使得映射函数拟合输入输出的真实分布。

Embedding是把原始输入空间，如文本、语音、图片、视频等，映射到中间空间latent space的过程。

Tokenize: from text to words/tokens

对于文本而言，最先进行的是分词；比如下面代码用最简单的字符分割，把长短的text拆分中单词或token。

代码块

```
import re

def tokenize(text):
    # Split by punctuation and whitespace
    tokens = re.split(r'([,.:;?_!"()\']|--|\s)', text)
    # Remove empty strings and strip whitespace
    tokens = [t.strip() for t in tokens if t.strip()]
    return tokens
```

我们拿Gutenberg中不到1000字的文本为例，分词结果如下：

代码块

```
with open("Peter_Rabbit.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

tokens = tokenize(raw_text)
print(tokens[:10])
```

> ['Once', 'upon', 'a', 'time', 'there', 'were', 'four', 'little', 'Rabbits', ',']

Encode: from token to ID

假设大模型还只是婴儿，看到的所有知识仅是上面的文本。我们可以对文本分词后，从0开始编号。

代码块

```
def build_vocab(whole_text):
    tokens = tokenize(whole_text)
    vocab = {token:id for id,token in enumerate(sorted(set(tokens)))}
    return vocab

vocab = build_vocab(raw_text)
print(len(vocab))
print(list(vocab.items())[:20])
```

> 405
>
> [('!', 0), ('"', 1), (',', 2), ('--', 3), ('.', 4), (':', 5), (';', 6), ('A', 7), ('After', 8), ('Also', 9), ('An', 10), ('And', 11), ('Benjamin', 12), ('Bunny', 13), ('But', 14), ('Cotton-tail', 15), ('Cottontail', 16), ('END', 17), ('Father', 18), ('First', 19)]

可见上文仅有405个不同的token。

而编码的过程，就是把不同的token，映射到从0开始的编号中，这个过程就是小学生的“查字典”过程，看看某个单词在字典里的序号是多少。

代码块

```
def encode(vocab, text):
    return [vocab[token] for token in tokenize(text)]

print(encode(vocab, "Once upon a time there were four little Rabbits"))
```

> [33, 373, 46, 354, 346, 386, 155, 210, 38]

如上示例，encode返回了每个token的编号。

Decode: from ID to token

而Decode的过程，刚好相反，是从编号还原到原始文本的过程。

代码块

```
def decode(vocab, ids):
    vocab_inverse = {id:token for token,id in vocab.items()}
    text= " ".join([vocab_inverse[id] for id in ids])
    return text

print(decode(vocab,[33, 373, 46, 354, 346, 386, 155, 210, 38]))
```

> Once upon a time there were four little Rabbits

如上所示，我们成功根据ID还原了原始文本。

Tokenizer: vocab, encode, decode

代码块

```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.vocab = vocab
        self.vocab_inverse = {id:token for token,id in vocab.items()}
```

```
5
6       def encode(self, text):
7           return [self.vocab[token] for token in tokenize(text)]
8
9       def decode(self, ids):
10          return " ".join([self.vocab_inverse[id] for id in ids])
11
12  tokenizer = SimpleTokenizerV1(vocab)
13  print(tokenizer.decode(tokenizer.encode("Once upon a time there were four
    little Rabbits")))
```

> Once upon a time there were four little Rabbits

把上面的代码放在一起，可以验证先把文本encode，再decode还原。

注：有时候会不成功；因为此处故意使用了特别简单的分词；你可以想办法完善。

Special token: UNKnown/EndOfSentence

上面的字典vocab明显太小，如果遇到新的单词，就会报错，如下：

代码块

```
1   print(tokenizer.decode(tokenizer.encode("Once upon a time there were four
    little Rabbits, and they were all very happy.")))
```

> KeyError Traceback (most recent call last)
>
> Cell In[24], line 1
>
> ----> 1 print(tokenizer.decode(tokenizer.encode("Once upon a time there were four little Rabbits, and they were all very happy.")))
>
> Cell In[15], line 7, in SimpleTokenizerV1.encode(self, text)
>
> 6 def encode(self, text):
>
> ----> 7 return [self.vocab[token] for token in tokenize(text)]
>
> Cell In[15], line 7, in <listcomp>(.0)
>
> 6 def encode(self, text):
>
> ----> 7 return [self.vocab[token] for token in tokenize(text)]

> KeyError: 'they'

回想幼儿园的学生，遇到不认识的字，会画个圆圈。同理，我们可以为vocab添加未知token。

代码块

```
vocab['<unk>'] = len(vocab)

print(list(vocab.items())[-5:])
```

> [('wriggled', 401), ('you', 402), ('young', 403), ('your', 404), ('<unk>', 405)]

如上，我们在字段最后加上了<unk>，以代表所有未知单词。

改进下上述代码，再次运行。

代码块

```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.vocab = vocab
        self.vocab_inverse = {id:token for token,id in vocab.items()}

    def encode(self, text):
        unk_id = self.vocab.get("<unk>")
        return [self.vocab.get(token,unk_id) for token in tokenize(text)]

    def decode(self, ids):
        return " ".join([self.vocab_inverse[id] for id in ids])

tokenizer = SimpleTokenizerV2(vocab)
print(tokenizer.decode(tokenizer.encode("Once upon a time there were four little Rabbits, and they were all very happy.")))
```

> Once upon a time there were four little Rabbits , and <unk> were all very <unk> .

可见至少没有报错了。当然，这依然不完美，因为我们无法完全还原最初的文本，所有不认识的token都成了unknown，当然会造成信息的损失。

BytePair Encoding: break words into chunks/subwords

现在回想幼儿园学生学习新单词的另一个方法：拆分。

以现在大模型最常用的tokenizer为例，一个单词可能会被拆分成如下4个token：

代码块

```
import tiktoken
```

```
2   tokenizer = tiktoken.get_encoding("gpt2")
3   print(tokenizer.encode("unbelievability"))
```

> [403, 6667, 11203, 1799]

代码块

```
1   print(tokenizer.decode([403,12,6667,12,11203,12,1799]))
```

> un-bel-iev-ability

tiktoken是OpenAI发布的高性能分词器，上述过程也可以在线可视化看到：

可以查看下gpt2使用的词表大小是50257。

代码块

```
1   print("vocab size of gpt2: ",tokenizer.n_vocab)
```

> vocab size of gpt2: 50257

通过这种拆分的方式，gpt2可以应对任何未知单词，因为最坏情况下可以拆成26个字母和标点。而Byte Pair的过程，就是反复合并频率最高的token对，构建出固定大小的词表。

注：BPE的原理和详细过程这里不赘述。可以从上面直观理解，unbelievability之所以拆分出un和ability，显然是因为他们是英文里面常见的前缀和后缀。

词表一点都不神秘，跟小学生用的字典没有本质上的区别，请看gpt2的vocab，刚好有50257个，最后一个是"<|endoftext|>": 50256

BPE 就像把一个完整的词语打碎成字符拼图，再根据频率统计一步步把常见组合粘回去，形成一个适合机器处理的 token 库。而合并的过程，需要根据merges的指引，直到不能再合并，最终每个 token 必须存在于 `vocab.json` 中才能作为模型输入。

在本文中，我们会始终使用该分词器：tokenizer = tiktoken.get_encoding("gpt2")。

Data Sampling with Sliding Window

用gpt2分词器把上述短文转成token IDs，如下：

代码块

```
with open("Peter_Rabbit.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

enc_text = tokenizer.encode(raw_text)
print("tokens: ",len(enc_text))
print("first 15 token IDs: ", enc_text[:15])
print("first 15 tokens: ","|".join(tokenizer.decode([token]) for token in enc_text[:15]))
```

> tokens: 1547
>
> first 15 token IDs: [7454, 2402, 257, 640, 612, 547, 1440, 1310, 22502, 896, 11, 290, 511, 3891, 198]
>
> first 15 tokens: Once| upon| a| time| there| were| four| little| Rabb|its|,| and| their| names|

从此我们的关注点都在token ID上，而不在关注原始的token；也就是不再看原文、只记单词编号。

大模型在训练和推理过程中，**本质上就是一个自回归（autoregressive）过程**。即我们在训练大模型的过程中，需要依次把单词逐一输入，并把当前的预测输出，作为下次的输入。

而模型一次能够“看到”的最大上下文长度，叫context_size。在gpt2中是1024，表示模型支持的输入序列最多是1024个token。

假设context_size是5，那么这一过程如下：

> Once --> upon
>
> Once upon --> a
>
> Once upon a --> time
>
> Once upon a time --> there
>
> Once upon a time there --> were

我们用token ID来表示：

代码块

```
context_size = 5
for i in range(1,context_size+1):
    context = enc_text[:i]
    desired = enc_text[i]
    print(context, "-->", desired)
```

> [7454] --> 2402
>
> [7454, 2402] --> 257
>
> [7454, 2402, 257] --> 640
>
> [7454, 2402, 257, 640] --> 612
>
> [7454, 2402, 257, 640, 612] --> 547

而上述就是滑动窗口的过程，每次target相比input偏移1。

而在训练的过程中，我们需要把输入分batch，并进行shuffle，完整代码示例如下：

代码块

```
from torch.utils.data import Dataset
import torch

class GPTDatasetV1(Dataset):
    def __init__(self, txt,tokenizer, context_size, stride):
        token_ids = tokenizer.encode(txt)
        assert len(token_ids) > context_size, "Text is too short"

        self.input_ids = [torch.tensor(token_ids[i:i+context_size])
                          for i in range(0, len(token_ids)-context_size, stride)]
        self.target_ids = [torch.tensor(token_ids[i+1:i+context_size+1])
                          for i in range(0, len(token_ids)-context_size, stride)]
    def __len__(self):
        return len(self.input_ids)
    def __getitem__(self, idx):
        return self.input_ids[idx], self.target_ids[idx]

def dataloader_v1(txt,batch_size=3,context_size=5,stride=2,shuffle=False,drop_last=True,num_workers=0):
    tokenizer = tiktoken.get_encoding("gpt2")
    dataset = GPTDatasetV1(txt,tokenizer,context_size,stride)
    return DataLoader(dataset, batch_size, shuffle=shuffle, drop_last=drop_last, num_workers=num_workers)
```

依然读取上述短文，通过dataloader和构建迭代器读取inputs和targets，均是token id，如下：

代码块

```
with open("Peter_Rabbit.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
dataloader = dataloader_v1(raw_text)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("shape of input: ",inputs.shape)
print("first batch, input: \n", inputs,"\n targets: \n", targets)
```

结果如下：

```
shape of inputs:  torch.Size([3, 5])
first batch, input:
 tensor([[ 7454,  2402,   257,   640,   612],
        [  257,   640,   612,   547,  1440],
        [  612,   547,  1440,  1310, 22502]])
 targets:
tensor([[ 2402,   257,   640,   612,   547],
        [  640,   612,   547,  1440,  1310],
        [  547,  1440,  1310, 22502,   896]])
```

在上述示例中，batch=3，context_size=5，所以inputs和targets的维度都是[3, 5]，也就分为3个batch，每个batch里面最多5个token id；而targets相比inputs总是偏移1；stride=2表示每次采样的偏移是2，也就是inputs的第二行相比第一行偏移2。因此batch对应二维张量的行数，context size对应列数，stride对应行间偏移。而targets与inputs的偏移总是1，这是由上述大模型自回归训练的本质决定的。

Token Embedding: From Words to Vectors

✏️ Vectors are

- high-dimensional
- dense
- learnable

Embedding is

- looking up vectors from a big table

- usually a matrix with shape (vocab_size, embed_dim)
- initialized with random values
- updated during training

上面我们把单词转换成了离散的编号数字，其实已经很接近大模型所能理解的空间了。只不过前面是连续的整数，如0,1,2。而大模型所希望的空间是高纬度的、连续的浮点数张量。为什么呢？最重要的原因是embedding空间的张量是可学习的参数，因此需要是“可微”的，便于做微分计算。而可学习参数的另一个隐含意思是，初始值没有那么重要，模型训练的过程中会不断地调整优化参数，直到达到最终较为完美的状态。

注：训练过程是根据梯度指明的方向优化参数的过程，可结合Pytorch理解计算图和自动微分过程；此处不赘述。

此处我们简单示例下Embedding空间，假设字典的总单词数是10，而Embedding的维度是4，如下：

代码块

```
from torch import nn

vocab_size = 10
embed_dim = 4
torch.manual_seed(123)
token_embedding_layer = nn.Embedding(vocab_size, embed_dim)
print("token_embedding_layer shape: ", token_embedding_layer.weight.shape)
print("token_embedding_layer weight: ", token_embedding_layer.weight)
```

```
token_embedding_layer shape:  torch.Size([10, 4])
token_embedding_layer weight:  Parameter containing:
tensor([[ 0.3374, -0.1778, -0.3035, -0.5880],
        [ 0.3486,  0.6603, -0.2196, -0.3792],
        [ 0.7671, -1.1925,  0.6984, -1.4097],
        [ 0.1794,  1.8951,  0.4954,  0.2692],
        [-0.0770, -1.0205, -0.1690,  0.9178],
        [ 1.5810,  1.3010,  1.2753, -0.2010],
        [ 0.9624,  0.2492, -0.4845, -2.0929],
        [-0.8199, -0.4210, -0.9620,  1.2825],
        [-0.3430, -0.6821, -0.9887, -1.7018],
        [-0.7498, -1.1285,  0.4135,  0.2892]], requires_grad=True)
```

可见，最终生成的Embedding空间的维度是(vocab_size, embed_dim)；而我们随机初始化了该空间，得到的是10行4列的二维张量。相当于构建了另一本“字典”，只不过这个字典后续是要被训练优化的。

而Embedding的过程，就是查询“新字典”的过程。

代码块

```
input_ids = torch.tensor([2,3,5])
token_embeddings = token_embedding_layer(input_ids)
print("token_embeddings: \n", token_embeddings) # return row 2,3,5 of weights
```

```
token_embeddings:
tensor([[ 0.7671, -1.1925,  0.6984, -1.4097],
        [ 0.1794,  1.8951,  0.4954,  0.2692],
        [ 1.5810,  1.3010,  1.2753, -0.2010]], grad_fn=<EmbeddingBackward0>)
```

如上，假设对token id分别为2、3、5的token做Embedding，那么返回的是上面二维张量的第2、3、5行（从0开始）。

上面仅是示例，在真实的GPT2中，使用的是(50257 tokens × 768 dimensions)。随机初始化如下：

代码块

```
from torch import nn

vocab_size = 50527
embed_dim = 768
torch.manual_seed(123)
token_embedding_layer_gpt2 = nn.Embedding(vocab_size, embed_dim)
print("token_embedding_layer_gpt2 shape: ",
token_embedding_layer_gpt2.weight.shape)
print("token_embedding_layer_gpt2 weight: ", token_embedding_layer_gpt2.weight)
```

```
token_embedding_layer_gpt2 shape:  torch.Size([50527, 768])
token_embedding_layer_gpt2 weight:  Parameter containing:
tensor([[ 0.3374, -0.1778, -0.3035,  ..., -0.3181, -1.3936,  0.5226],
        [ 0.2579,  0.3420, -0.8168,  ..., -0.4098,  0.4978, -0.3721],
        [ 0.7957,  0.5350,  0.9427,  ..., -1.0749,  0.0955, -1.4138],
        ...,
        [-1.8239,  0.0192,  0.9472,  ..., -0.2287,  1.0394,  0.1882],
```

```
       [-0.8952, -1.3001,  1.4985,  ..., -0.5879, -0.0340, -0.0092],
       [-1.3114, -2.2304, -0.4247,  ...,  0.8176,  1.3480, -0.5107]],
      requires_grad=True)
```

可以把上面巨大的二维张量，想象成如下的表格：

```
代码块
1   Token ID     |     Embedding vector (768 dims)
2   ---------------------------------------------
3   0            |    [0.12, -0.03, ...,  0.88]
4   1            |    [0.54,  0.21, ..., -0.77]
5   ...          |    ...
6   50526        |    [...]
```

同样的，Embedding的过程依然是查询这个巨大的表格的过程。

```
代码块
1   input_ids = torch.tensor([2,3,5])
2   print(token_embedding_layer_gpt2(input_ids))
```

```
tensor([[ 0.7957,  0.5350,  0.9427,  ..., -1.0749,  0.0955, -1.4138],
        [-0.0312,  1.6913, -2.2380,  ...,  0.2379, -1.1839, -0.3179],
        [-0.4334, -0.5095, -0.7118,  ...,  0.8329,  0.2992,  0.2496]],
       grad_fn=<EmbeddingBackward0>)
```

如上示例，对token id=2进行Embedding，取的是表格的第2行。

综上可知，Embedding的过程，就是把token转换为tensor的过程，就是把一维离散的token id映射到高纬、连续的稠密空间的过程。而这个过程，就是平淡无奇的lookup操作。

Position Embedding: From Position to Vectors

✏️ position embeddin is

- a matrix with shape (context_size, embed_dim)
- initialized with random values
- a learnable parameter, updated during training

前面讲述了token的嵌入过程，其实就是在大表中根据token id查询的过程。然而，我们需要注意到，在上面表格里的不同行之间，似乎是无关的。

如“You eat fish”和“Fish eat you”在token embeding的层面是相似的（Transformer本身是顺序无感的），但表达的语义却是截然不同的。因此，需要引入位置信息，从0开始进行位置编号。

如下例子，第0的position是you还是fish，一目了然：

代码块

```
"You eat fish"
   ↓        ↓       ↓
[you] + P0 [eat] + P1 [fish] + P2

"Fish eat you"
   ↓         ↓         ↓
[fish] + P0 [eat] + P1 [you] + P2

→ 即使 Token 一样，只要位置不同，最终向量就不同。
→ Transformer 能区分主语、宾语等结构含义。
```

显然，位置编码应该从0开始，直到context_size-1。

回想如前所述，使用离散整数会导致空间表达能力削弱、无法进行自动求导优化，因此同样地，我们依然需要把位置编号转化为高维稠密的张量。

如下，假设context_size是5，Embedding维度是4：

代码块

```
from torch import nn

context_size = 5
embed_dim = 4
torch.manual_seed(123)
position_embedding_layer = nn.Embedding(context_size, embed_dim)
print("position_embedding_layer shape: ",
position_embedding_layer.weight.shape)
print("position_embedding_layer weight: ", position_embedding_layer.weight)
```

```
position_embedding_layer shape:  torch.Size([5, 4])
position_embedding_layer weight:  Parameter containing:
tensor([[ 0.3374, -0.1778, -0.3035, -0.5880],
        [ 1.5810,  1.3010,  1.2753, -0.2010],
        [-0.1606, -0.4015,  0.6957, -1.8061],
```

> [-1.1589, 0.3255, -0.6315, -2.8400],
>
> [-0.7849, -1.4096, -0.4076, 0.7953]], requires_grad=True)

我们就会得到5*4的二维张量。

请注意位置张量的维度是(context_size, embed_dim)，也就是说跟token张量的列数是一样的，但是行数不一样。

Position Tensor本质上是另一本“位置字典”，embedding的过程是根据位置编号查询的过程。

如下示例，

代码块

```
input_ids = torch.tensor([2,3,5])
# use Position of input_ids, NOT values of it
position_embeddings = position_embedding_layer(torch.arange(len(input_ids)))
print("position_embeddings: \n", position_embeddings) # return row 0,1,2 of weights
```

> position_embeddings:
>
> tensor([[0.3374, -0.1778, -0.3035, -0.5880],
>
> [1.5810, 1.3010, 1.2753, -0.2010],
>
> [-0.1606, -0.4015, 0.6957, -1.8061]], grad_fn=<EmbeddingBackward0>)

返回的是上述Embedding的前3行，因为输入是3个token。请特别注意，做position Embedding的时候，使用的是token id的位置，而不是token id的值。

注：此处PE使用的是Learnable Absolute Positional Embeddings，除此之外，还有不可训练的、固定位置编码，如Sinusoidal PE（固定位置、使用sin/cos）、RoPE（相对旋转）等，性能更强。不过从初学角度上，Learnable PE是最简单、最容易理解的位置编码思想。

Input Embedding: token_embedding + position_embedding

综上，有了token embeding和pos embedding之后，只需要简单相加，就可以得到最终的input embedding。

代码块

```
input_embeddings = token_embeddings + position_embeddings
print("shape of input_embeddings : ",input_embeddings.shape)
print("input_embeddings: ", input_embeddings)
```

```
shape of input_embeddings :  torch.Size([3, 4])
input_embeddings:  tensor([[ 1.1045, -1.3703,  0.3948, -1.9977],
        [ 1.7603,  3.1962,  1.7707,  0.0682],
        [ 1.4204,  0.8996,  1.9710, -2.0070]], grad_fn=<AddBackward0>)
```

你可以手动计算验证，0.7671+0.3374=1.1045

GPT2使用的位置编号是(1024 positions × 768 dimensions)，1024代表模型最大能处理的输入长度，768是与token Embedding维度相同，选择了一个相对高纬稠密的空间。

我们通过如下代码复现：

代码块

```
from torch import nn

context_size = 1024
embed_dim = 768
torch.manual_seed(123)
position_embedding_layer_gpt2 = nn.Embedding(context_size, embed_dim)
print("position_embedding_layer_gpt2 shape: ",
position_embedding_layer_gpt2.weight.shape)
print("position_embedding_layer_gpt2 weight: ",
position_embedding_layer_gpt2.weight)
```

```
position_embedding_layer_gpt2 shape:  torch.Size([1024, 768])
position_embedding_layer_gpt2 weight:  Parameter containing:
tensor([[ 0.3374, -0.1778, -0.3035,  ..., -0.3181, -1.3936,  0.5226],
        [ 0.2579,  0.3420, -0.8168,  ..., -0.4098,  0.4978, -0.3721],
        [ 0.7957,  0.5350,  0.9427,  ..., -1.0749,  0.0955, -1.4138],
        ...,
        [-1.2094,  0.6397,  0.6342,  ..., -0.4582,  1.4911,  1.2406],
        [-0.2253, -0.1078,  0.0479,  ...,  0.2521, -0.2893, -0.5639],
        [-0.5375, -1.1562,  2.2554,  ...,  1.4322,  1.2488,  0.1897]],
       requires_grad=True)
```

我们依然用上面短文为例：

```
with open("Peter_Rabbit.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
dataloader = dataloader_v1(raw_text,batch_size=3, context_size=1024,stride=2)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("shape of input: ",inputs.shape)
print("first batch, input: \n", inputs,"\n targets: \n", targets)
```

```
shape of input:  torch.Size([3, 1024])
first batch, input:
 tensor([[ 7454,  2402,   257,  ...,   480,   517,   290],
        [  257,   640,   612,  ...,   290,   517, 36907],
        [  612,   547,  1440,  ..., 36907,    13,  1763]])
 targets:
 tensor([[ 2402,   257,   640,  ...,   517,   290,   517],
        [  640,   612,   547,  ...,   517, 36907,    13],
        [  547,  1440,  1310,  ...,    13,  1763,  1473]])
```

输入input的维度是：3批 * 1024的context长度。

我们可以对batch0的input进行Embedding，如下：

代码块

```
token_embeddings = token_embedding_layer_gpt2(inputs)
print("shape of token_embeddings: ",token_embeddings.shape)

position_embeddings = position_embedding_layer_gpt2(torch.arange(context_size))
print("shape of position_embeddings: ",position_embeddings.shape)

# token_embeddings shape: [batch_size, seq_len, embedding_dim]
# position_embeddings shape: [seq_len, embedding_dim]
# PyTorch automatically broadcasts position_embeddings across batch dimension
input_embeddings = token_embeddings + position_embeddings
print("shape of input_embeddings : ",input_embeddings.shape)
```

```
shape of token_embeddings:  torch.Size([3, 1024, 768])
shape of position_embeddings:  torch.Size([1024, 768])
shape of input_embeddings :  torch.Size([3, 1024, 768])
```

请特别注意tensor shape的变化；其中position embedding是没有batch维度的，但有赖于pytorch的广播机制，依然可以自动地相加。

注：tensor是大模型的“语言”，所有大模型的操作本质上都是对tensor的操作。建议熟悉tensor常用的概念与操作，如broadcast/view/reshape/squeeze/transpose/einsum/mul/matmul/dot等。Tensor是描述集合与高维空间最简单优雅的语言。初学者不要把tensor想象成复杂的数学难题，可以当成简单的“外语”，只是一种简洁的约定和表达手段。

至此，我们得到了大模型真正能够处理的输入空间，即inputs embedding，包含了token embeding的token相关信息，以及token位置的position embedding信息。