

# Auto-GPT: Autonomous AI Experiment.

July 22, 2023

<https://github.com/Significant-Gravitas/Auto-GPT>

Used Link Reader

The link you provided leads to a GitHub repository for a project called "Auto-GPT: An Autonomous GPT-4 Experiment". Here's a summary of the project:

- Auto-GPT is an experimental open-source application that showcases the capabilities of the GPT-4 language model. It chains together LLM "thoughts" to autonomously achieve whatever goal you set.
- It's one of the first examples of GPT-4 running fully autonomously, pushing the boundaries of what is possible with AI.
- The project is under heavy development and the master branch may often be in a broken state. It's recommended to use the latest stable release.

Key Features:

- Internet access for searches and information gathering
- Long-term and short-term memory management
- GPT-4 instances for text generation
- Access to popular websites and platforms
- File storage and summarization with GPT-3.5
- Extensibility with Plugins

Limitations:

- It's an experiment, not a polished application or product.
- May not perform well in complex, real-world business scenarios.
- Quite expensive to run, so users need to set and monitor their API key limits with OpenAI.

Disclaimer:

- The project is provided "as-is" without any warranty. Users assume all risks associated with its use, including data loss, system failure, or any other issues that may arise.
- The use of the GPT-4 language model can be expensive due to its token usage. Users are responsible for monitoring and managing their own token usage and the associated costs.

For more detailed information, you can check out the [documentation](#) on the GitHub page.



<https://github.com/reworkd/AgentGPT>

Used Link Reader

The link you provided leads to a GitHub repository for a project called "AgentGPT". Here's a summary of the project:

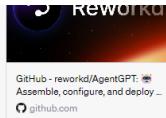
- AgentGPT allows you to assemble, configure, and deploy autonomous AI agents in your browser. You can name your own custom AI and set it on any goal imaginable. The AI will attempt to reach the goal by thinking of tasks to do, executing them, and learning from the results.
- The easiest way to get started with AgentGPT is the automatic setup CLI bundled with the project. The CLI sets up environment variables (and API Keys), a database (MySQL), a backend (FastAPI), and a frontend (Next.js).
- Prerequisites for getting started include an editor of your choice (e.g., Visual Studio Code), Node.js, Git, Docker, and an OpenAI API key.
- The platform is currently in beta, and a full list of completed and planned features can be found on their public roadmap.

Tech Stack:

- Bootstrapping: create-t3-app + FastAPI-template.
- Framework: Next.js 13 + TypeScript + FastAPI
- Auth: Next-Auth.js
- ORM: Prisma & SQLModel.
- Database: Planetscale.
- Styling: TailwindCSS + HeadlessUI.
- Schema Validation: Zod + Pydantic.
- LLM Tooling: Langchain.

For more detailed information, you can check out the [documentation](#) on the GitHub page.





https://github.com/hwchase17/langchain

Used Link Reader

The link you provided leads to a GitHub repository for a project called "LangChain". Here's a summary of the project:

- LangChain is a library that aims to assist in the development of applications using large language models (LLMs). It recognizes that the real power of LLMs comes when you can combine them with other sources of computation or knowledge.
- The library is designed to help with six main areas: LLMs and Prompts, Chains, Data Augmented Generation, Agents, Memory, and Evaluation.
- LLMs and Prompts include prompt management, prompt optimization, a generic interface for all LLMs, and common utilities for working with LLMs.
- Chains involve sequences of calls (whether to an LLM or a different utility). LangChain provides a standard interface for chains, lots of integrations with other tools, and end-to-end chains for common applications.
- Data Augmented Generation involves specific types of chains that first interact with an external data source to fetch data for use in the generation step.
- Agents involve an LLM making decisions about which Actions to take, taking that Action, seeing an Observation, and repeating that until done. LangChain provides a standard interface for agents, a selection of agents to choose from, and examples of end-to-end agents.
- Memory refers to persisting state between calls of a chain/agent. LangChain provides a standard interface for memory, a collection of memory implementations, and examples of chains/agents that use memory.
- Evaluation is a new way of evaluating generative models using language models themselves to do the evaluation. LangChain provides some prompts/chains for assisting in this.

For more detailed information, you can check out the [documentation](#) on the GitHub page.



LangChain/AutoGPT/AgentGPT 他们三者之前的技术差异是什么？我怎么感觉他们都非常相似呢

这三个项目 (LangChain, AutoGPT, AgentGPT) 都是基于大型语言模型 (LLM) 的应用，但他们的目标和实现方式有所不同。以下是对每个项目的简单概述和他们之间的主要区别：

- LangChain**: 这是一个库，旨在帮助开发者使用大型语言模型 (LLM) 构建应用。它提供了一种方式来组合LLM和其他计算或知识源。LangChain设计用于帮助处理LLM和提示，链，数据增强生成，代理，内存和评估等六个主要领域。这是一个更为基础的工具，可以用于构建各种基于LLM的应用。
- AutoGPT**: 这是一个实验性的开源应用，展示了GPT-4语言模型的能力。它通过GPT-4驱动，将LLM的“思考”链接在一起，以自主地实现你设定的任何目标。它是GPT-4完全自主运行的第一个例子之一，推动了AI的可能性边界。
- AgentGPT**: 这个项目允许你在浏览器中组装、配置和部署自主AI代理。你可以命名你自己的自定义AI，并让它开始实现任何你可以想象的目标。AI将尝试通过思考要做的任务，执行它们，并从结果中学以达到目标。

总的来说，这三个项目都是为了利用大型语言模型 (LLM) 的能力来构建应用，但他们的关注点和实现方式有所不同。LangChain更偏向于一个基础工具库，可以帮助开发者更好地利用LLM。而AutoGPT和AgentGPT则是具体的应用实例，展示了如何利用LLM来实现特定的目标，例如自主完成任务或者达到设定的目标。

https://github.com/liaokongVFX/LangChain-Chinese-Getting-Started-Guide

Used Link Reader

你提供的链接是一个名为"LangChain 中文入门教程"的GitHub仓库。这个教程是为了帮助中文用户更好地理解和使用LangChain库。以下是该教程的主要内容：

- 教程已经生成了gitbook，方便阅读：<https://liaokong.gitbook.io/lim-kai-fa-jiao-cheng/>
- LangChain库一直在飞速更新迭代，但该文档写于4月初，因此colab里面的代码可能有些已经过时。如果有运行失败的可以先搜索一下当前文档是否有更新，如文档也没更新欢迎提issue，或者修复后直接提pr。

教程主要介绍了以下几个方面的内容：

- LangChain的基本介绍**: LangChain是一个用于开发由语言模型驱动的应用程序的框架。他主要拥有2个能力：可以将LLM模型与外部数据源进行连接，允许与LLM模型进行交互。
- 基础知识**: 包括LLM调用，Prompt管理，文档加载器，文档分割器，向量化，对接向量存储与搜索，Chains等。
- 必知概念**: 包括Loader加载器，Document文档，Text Splitters文本分割，Vectorstores向量

数据链, Chain, Agent代理, Enduring等。

4. 完成: 包括完成一次问答, 通过Google搜索并返回答案, 对超长文本进行总结等。

这个教程是一个非常全面的LangChain使用指南, 对于想要使用LangChain的开发者来说是一个很好的参考资源。

https://python.langchain.com/docs/use\_cases/question\_answering/

Used Link Reader

你提供的链接是LangChain的官方文档, 详细介绍了如何使用LangChain进行聊天和问题回答(QA)的应用。以下是该文档的主要内容:

- LangChain支持在各种数据类型上进行聊天和QA, 包括非结构化数据(例如, PDFs)、结构化数据(例如, SQL)和代码(例如, Python)。
- 对于非结构化数据, LangChain提供了以下一般步骤:
  1. 分割: 文本分割器将文档分割成指定大小的片段。
  2. 存储: 存储(例如, 通常是向量存储)将存储并常嵌入这些片段。
  3. 检索: 应用程序从存储中检索片段(例如, 通常是与输入问题具有相似嵌入的片段)。
  4. 输出: LLM使用包含问题和检索到的片段的提示生成答案。
- 文档详细介绍了如何使用LangChain加载文档, 分割文档, 将片段嵌入并存储在向量数据库中, 以及如何检索相关片段以回答任何问题。
- 文档还详细介绍了如何使用LLM(例如, gpt-3.5-turbo)和RetrievalQA链将检索到的文档提炼成答案。
- 此外, 文档还介绍了如何使用内存缓冲区来跟踪对话输入/输出, 以及如何使用ConversationalRetrievalChain进行聊天。

这个文档是一个非常全面的LangChain使用指南, 对于想要使用LangChain进行聊天和问题回答的开发者来说是一个很好的参考资源。你可以查看原始文档以获取更详细的信息。

在LangChain中提到了使用了Embedding技术, 将知识文本内容通过Embedding转成词向量数据。然后通过将问题也转成词向量数据后, 对比他们之前的相似度, 将相关的信息提取出来再发送给LLM, 然后让LLM组织语言返回数据。

这里的Embedding是对整段话转成词向量吧, 那这样的话, 问题与文本的长度不就会不相同了吗? 这样的情况下他们如何计算相似度的呢?

你的理解是正确的。在LangChain中, 确实使用了嵌入(Embedding)技术, 将知识文本内容转化为词向量数据。然后, 将问题也转化为词向量数据, 通过比较他们之间的相似度, 将相关的信息提取出来再发送给LLM, 然后让LLM组织语言返回数据。

至于你的问题, 即使问题和文本的长度不同, 也可以计算他们的词向量的相似度。这是因为在词嵌入(Word Embedding)中, 无论输入的文本长度如何, 每个词都会被转化为一个固定长度的向量。因此, 我们可以通过比较这些向量来计算文本之间的相似度。

具体来说, 当我们说计算两个文本的相似度时, 通常是指计算他们的词向量的相似度。这可以通过各种方式完成, 例如计算余弦相似度(Cosine Similarity), 这是一种常用的度量向量之间相似度的方法。

另外, 对于整段话的嵌入, 通常使用的是句子嵌入(Sentence Embedding)或者段落嵌入(Paragraph Embedding), 这种嵌入方式会将整个句子或者段落映射到一个固定长度的向量, 这样也可以用来计算不同长度文本之间的相似度。

总的来说, 无论文本的长度如何, 我们都可以通过词嵌入或句子嵌入将它们转化为固定长度的向量, 然后通过计算这些向量的相似度来计算文本之间的相似度。

哦, 我看到代码中处理时用的OpenAIEmbeddings类, OpenAIEmbeddings类是不是用了OpenAI训练的LLM模型中的Embedding层作为模型的输出, 你所谓的给予不同长度的输入, 返回的长度相同, 是不是可以理解为LLM的Embeddings层之后的输出层是大小固定的所以都是相同的?

你的理解基本正确。OpenAIEmbeddings类确实使用了OpenAI训练的LLM模型的嵌入层(Embedding layer)。这个嵌入层可以将不同长度的输入文本转化为固定长度的向量。

在许多深度学习模型中, 嵌入层通常是模型的第一层, 它将输入的词(或者句子、段落等)转化为向量。这些向量在模型的后续层中被进一步处理。在某些模型中, 例如transformer模型(包括GPT系列模型), 最后一层(输出层)通常会输出一个固定长度的向量, 这个向量是模型对输入的整体理解或者表示。

然而, 需要注意的是, 虽然这个向量的长度是固定的, 但它并不直接等于输入的长度。例如, 如果我们输入一个包含10个词的句子和一个包含20个词的句子, 模型的输出向量的长度是相同的, 但这并不意味着这两个向量就可以直接比较。因为这两个向量是模型对两个不同长度的输入的理解, 所以它们可能在语义上有很大的差异。

在计算相似度时, 我们通常会使用一些度量方法, 例如余弦相似度。这种方法可以度量两个向量在语义上的相似度, 而不仅仅是他们的长度。

其实是采用OpenAI的Embedding还是自己训练的LLM中的Embedding对于距离计算的差距也不大吗? 当然我这里说的自己训练的LLM, 也是能够做到一定能力的模型, 而非训练不充分的模型

你的理解是正确的。无论是使用OpenAI的嵌入(Embedding)还是自己训练的大型语言模型(LLM)中的嵌入, 只要这些模型都被充分训练并且具有一定的性能, 那么他们在计算距离(或者说相似度)时的结果应该都是相似的。

这是因为无论是OpenAI的模型还是自己训练的模型, 他们的目标都是将文本转化为向量, 这些向量能够捕捉到文本的语义信息。因此, 如果两个模型都被充分训练, 那么他们转化出的向量应该都能够很好地表示原始文本的语义, 从而在计算相似度时得到相似的结果。

然而, 需要注意的是, 虽然这两种模型在计算相似度时的结果可能相似, 但他们在处理不同类型文本时可能会有所不同。例如, 如果一个模型在训练时使用了大量的科技文本, 那么

它可能在处理科技相关的问题时表现得更好。因此，选择哪种模型取决于你的具体需求和可用的资源。

那人这样的话，其实这里的关键技巧应该不是采用了什么样的Embedding，而是如何处理文字，然后Embedding得到结果吧。就像你前面说的10个词和20个词的句子的差异一样

你的理解是正确的。在这种情况下，关键的技术不仅仅是使用了哪种嵌入（Embedding）技术，更重要的是如何处理和理解文本，以及如何使用嵌入得到的结果。

首先，处理文本的方式对结果有很大的影响。例如，文本的预处理（如分词、去除停用词等）、文本的表示（如词袋模型、TF-IDF、词嵌入等）都会影响到最后的结果。

其次，如何使用嵌入得到的结果也非常重要。例如，如何从嵌入的向量中提取有用的信息、如何比较不同向量的相似度、如何根据这些相似度来检索相关信息等。

最后，你提到的10个词和20个词的句子的差异，这其实是一个很好的例子。虽然这两个句子的长度不同，但我们可以将它们转化为相同长度的向量。然后，我们可以通过比较这些向量的相似度来找出这两个句子的相似之处。这就是嵌入技术的强大之处，它可以将不同长度的文本转化为相同长度的向量，从而使我们能够比较和理解不同的文本。

那人还有一个问题，在LangChain知识库问答中，如果提问的问题的内容中能够体现出一些知识库的内容，相对来说更能找到准确答案。但是如果我训练的问题，就不曾在知识库中出现过，那这样的话就很难匹配了吧。

举例来说：  
如果我的知识库是三国演义的电子书，若我的问题是赤壁之战中诸葛亮做了那些事情？这样的问题或许就不太好解答  
如果我的问题是：刘备三顾茅庐时与诸葛亮的对话内容，这个问题或许会好回答一些。  
我的底层逻辑是，问题内容本身并没有出现在知识库中过，那么这样的匹配可能就不会太好

你的理解是正确的。在使用基于检索的问答系统（如LangChain）时，问题的内容和知识库的内容有直接关联是非常重要的。如果问题的内容在知识库中有直接的对应或者相关内容，那么系统更有可能找到准确的答案。

例如，你提到的问题“刘备三顾茅庐时与诸葛亮的对话内容”，如果这个对话在知识库（三国演义的电子书）中有明确的记录，那么系统就可以直接找到这个对话并返回作为答案。

然而，对于问题“赤壁之战中诸葛亮做了那些事情”，如果知识库中没有明确记录诸葛亮在赤壁之战中做了什么，那么系统可能就无法给出准确的答案。这时，系统可能需要从知识库中找到与赤壁之战和诸葛亮相关的所有信息，然后尝试从这些信息中推断出诸葛亮在赤壁之战中可能做了什么。这显然比直接找到对应的对话要困难得多。

因此，对于基于检索的问答系统，问题的内容和知识库的内容有直接关联是非常重要的。如果问题的内容在知识库中没有直接的对应或者相关内容，那么系统可能就无法给出准确的答案。

这是GPT2的模型：  
GPT2LMHeadModel(  
(transformer): GPT2Model(  
(wte): Embedding(50257, 768)  
(wpe): Embedding(1024, 768)  
(drop): Dropout(p=0.1, inplace=False)  
(h): ModuleList(  
(0-11): 12 x GPT2Block(  
(in\_1): LayerNorm((768,), eps=1e-05, elementwise\_affine=True)  
(attn): GPT2Attention(  
(c\_attn): Conv1D  
(c\_proj): Conv1D  
(attn\_dropout): Dropout(p=0.1, inplace=False)  
(resid\_dropout): Dropout(p=0.1, inplace=False)  
)  
(in\_2): LayerNorm((768,), eps=1e-05, elementwise\_affine=True)  
(mlp): GPT2MLP(  
(c\_fc): Conv1D  
(c\_proj): Conv1D  
(act): NewGELUActivation()  
(dropout): Dropout(p=0.1, inplace=False)  
)  
)  
(in\_f): LayerNorm((768,), eps=1e-05, elementwise\_affine=True)  
(lm\_head): Linear(in\_features=768, out\_features=50257, bias=False)  
)

这个模型如果只用Embedding层的话，输出的大小是768吗？

在GPT-2模型中，“wte”代表的是词嵌入（Word Embedding）层，“wpe”代表的是位置嵌入（Position Embedding）层。这两个嵌入层都是模型的输入部分。

对于“wte”，它是一个嵌入层，将每个词转换为一个768维的向量。所以，如果你只使用这个嵌入层，每个词的输出将是一个768维的向量。

然而，需要注意的是，这个输出并不代表整个句子或段落的输出。在GPT-2模型中，整个句子或段落的输出是通过模型的所有层（包括自注意力层和前馈神经网络层）计算得到的。这个输出通常也是一个768维的向量，但它包含了整个句子或段落的信息，而不仅仅是单个词的信息。

所以，如果你想得到整个句子或段落的嵌入，你需要将句子或段落输入到模型中，并取出最后一层的输出。这个输出将是一个包含了整个句子或段落信息的768维向量。

假如句子长度是20个字，那这里的输出是20x768？

是的，你的理解是正确的。在GPT-2模型中，每个输入的词都会被转换为一个768维的向量。

所以，如果你的句子有20个词，那么模型的输出将是一个 $20 \times 768$ 的矩阵，每一行对应于句子中的一个词的嵌入向量。

然而，需要注意的是，这个输出矩阵并不代表整个句子的嵌入。它只是句子中每个词的嵌入向量。如果你想得到整个句子的嵌入，你需要对这个矩阵进行某种形式的聚合，例如取平均值或者使用某种注意力机制。

另外，这个输出矩阵是模型的最后一层的输出，它包含了模型对句子中每个词的理解。这个理解不仅包括每个词本身的信息，还包括每个词在句子中的上下文信息。所以，即使两个词的词嵌入（即模型的输入）是相同的，他们在模型的输出中也可能是不同的，因为他们的上下文可能是不同的。

在前面你提到：

在LangChain中，确实使用了嵌入（Embedding）技术，将知识文本内容转化为词向量数据。然后，将问题也转化为词向量数据，通过比较他们之间的相似度，将相关的信息提取出来再发送给LLM，然后让LLM组织语言返回回答。

至于你的问题，即使问题和文本的长度不同，也可以计算他们的词向量的相似度。这是因为，在词嵌入（Word Embedding）中，无论输入的文本长度如何，每个词都会被转化为一个固定长度的向量。因此，我们可以通过比较这些向量来计算文本之间的相似度。

这里很显然与我用GPT2的Embedding的举例是不一致的。



对不起，我之前的解释可能造成了一些混淆。你是对的，当我们谈论GPT-2或者类似的transformer模型的嵌入层时，我们通常是指每个单词或者标记被转换为一个固定大小的向量。这意味着，如果你有一个包含20个词的句子，那么你会得到一个 $20 \times 768$ 的矩阵，每行对应一个词的嵌入。

然而，当我们在LangChain或者其他基于检索的问答系统中讨论嵌入时，我们通常是指整个文本或者文档的嵌入，而不仅仅是单个词的嵌入。这种情况下，无论文本的长度如何，我们都会得到一个固定大小的向量。这通常是通过将单词嵌入进行某种形式的聚合（例如，取平均值或者使用某种注意力机制）来实现的。

所以，你是对的。这两种情况下的嵌入是不一样的。我之前的解释可能没有清楚地区分这两种情况，对此我深感抱歉。希望这个解释能够帮助你理解这两种情况下的嵌入的区别。



我现在有一个训练好的GPT2网络，我如何将这个LLM的Embedding层抽取出来，设计成类似与LangChain一样的固定输出大小的Embedding神经网络呢？



如果你想从训练好的GPT-2模型中抽取嵌入层，并设计一个具有固定输出大小的嵌入神经网络，你可以考虑以下步骤：

1. **抽取嵌入层：**在GPT-2模型中，嵌入层可以通过模型的`wte`属性访问。例如，如果你的模型对象名为`model`，那么你可以通过`model.transformer.wte`来访问嵌入层。
2. **将句子转化为嵌入向量：**你可以将句子中的每个词通过嵌入层转化为一个向量，然后得到一个向量的序列。这个序列的大小是`[句子长度, 嵌入维度]`。
3. **聚合向量序列：**为了得到一个固定大小的向量，你需要将向量序列进行聚合。这可以通过多种方式实现，例如，你可以取向量的平均值，或者使用某种注意力机制。这样，无论句子的长度如何，你都可以得到一个固定大小的向量。

以下是一个简单的例子，展示了如何实现这个过程：

```
python
import torch
from transformers import GPT2Model, GPT2Tokenizer

# 加载模型和分词器
model = GPT2Model.from_pretrained('gpt2')
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

# 输入句子
sentence = "This is an example sentence."

# 分词并转化为张量
inputs = tokenizer(sentence, return_tensors='pt')

# 通过嵌入层得到向量序列
embeddings = model.transformer.wte(inputs['input_ids'])

# 聚合向量序列得到固定大小的向量
sentence_embedding = torch.mean(embeddings, dim=1)

print(sentence_embedding.size()) # 输出: torch.Size([1, 768])
```

在这个例子中，我们首先加载了模型和分词器，然后输入了一个句子。我们将句子分词并转化为张量，然后通过嵌入层得到了一个向量序列。最后，我们通过取向量序列的平均值得到了一个固定大小的向量。这个向量的大小是`[1, 768]`，无论输入句子的长度如何，这个大小都是固定的。



如果采用多头注意力机制来提取词向量的话，该怎么做？这里的多头注意力机制，是不是也应该用GPT2现有的模型中的层来做呢？



在GPT-2模型中，多头注意力机制是在模型的自注意力层中实现的。这个层级可以通过模型的`h`属性访问，它是一个包含多个`GPT2Block`的`ModuleList`。每个`GPT2Block`都包含一个`GPT2Attention`对象，这个对象实现了多头注意力机制。

如果你想使用GPT-2模型中的多头注意力机制来提取词向量，你可以考虑以下步骤：

1. **将句子转化为嵌入向量：**这一步与之前的步骤相同，你可以将句子中的每个词通过嵌入层转化为一个向量，然后得到一个向量的序列。
2. **通过自注意力层处理向量序列：**你可以将向量序列输入到模型的自注意力层，然后得到一个新的向量序列。这个新的向量序列包含了每个词在句子中的上下文信息。

③ **聚类向量序列**: 为了得到一个固定大小的向量, 你需要将向量序列进行聚类。这可以通过多种方式实现, 例如, 你可以取向量的平均值, 或者使用某种注意力机制。

以下是一个简单的例子，展示了如何实现这个过程：

```
python

import torch
from transformers import GPT2Model, GPT2Tokenizer

# 加载模型和分词器
model = GPT2Model.from_pretrained('gpt2')
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

# 输入句子
sentence = "This is an example sentence."

# 分词并转化为张量
inputs = tokenizer(sentence, return_tensors='pt')

# 通过嵌入层得到向量序列
embeddings = model.transformer.wte(inputs['input_ids'])

# 通过自注意力层处理向量序列
# 注意：这里我们只使用了模型的第一个自注意力层
# 如果你想使用其他的自注意力层，你可以修改这里的索引
attention_outputs = model.transformer.h[0].attn(embeddings)

# 聚合向量序列得到固定大小的向量
sentence_embedding = torch.mean(attention_outputs[0], dim=1)

print(sentence_embedding.size()) # 输出: torch.Size([1, 768])
```

在这个例子中，我们首先加载了模型和分词器，然后输入了一个句子。我们将句子分词并转化为张量，然后通过嵌入层得到了一个向量序列。然后，我们将向量序列输入到模型的注意力层，得到了一个新的向量序列。最后，我们通过取向量序列的平均值得到了一个固定大小的向量。这个向量的大小是“**[1, 768]**”，无论输入句子的长度如何，这个大小都是固定的。

而我给你打印的GPT2的模型中， $h$ 下一共有12层，而你上述的代码实际上只是使用了第一层的attn层？另外GPT2默认的神经网络中，在执行attn之后还做了一步LayerNorm，而你却忽略了，是什么呢？

 对不起，我在上述代码中确实忽略了一些步骤。在实际使用中，你应该遵循GPT-2模型的完整架构，包括所有的层和操作。这包括所有的自注意力层，以及每个自注意力层之前的层归一化（Layer Normalization）操作。

在GPT-2模型中，每个`**GPT2Block**`都包含一个自注意力层和一个前馈神经网络层，这两个层之间以及之后都有一个层归一化操作。这些操作都是模型的一部分，对于模型的性能和结果都有重要的影响。

如果你想使用GPT-2模型的所有自注意力层来提取词向量，你可以直接使用模型的前向传播函数（`forward`）。这个函数会自动处理所有的层和操作，包括所有的自注意力层和层归一化操作。

以下是一个简单的例子，展示了如何实现这个过程：

```
python

import torch
from transformers import GPT2Model, GPT2Tokenizer

# 加载模型和分词器
model = GPT2Model.from_pretrained('gpt2')
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

# 输入句子
sentence = "This is an example sentence."

# 分词并转化为张量
inputs = tokenizer(sentence, return_tensors='pt')

# 通过模型的前向传播函数处理输入
outputs = model(**inputs)

# 聚合向量序列得到固定大小的向量
# 注意: outputs.last_hidden_state 包含了模型最后一层的输出
sentence_embedding = torch.mean(outputs.last_hidden_state, dim=1)

print(sentence_embedding.size()) # 输出: torch.Size([1, 768])
```

在这个例子中，我们首先加载了模型和分词器，然后输入了一个句子。我们将句子分词并转化为张量，然后通过模型的前向传播函数处理了输入。然后，我们通过取向量序列的平均值得到了一个固定大小的向量。这个向量的大小是**[1, 768]**，无论输入句子的长度如何，这几个大小都是固定的。

这个last\_hidden\_state，其实是对应GPT2预测下一个字的输入信息吧？  
last\_hidden\_state的下一步计算是不是就是 (In�): LayerNorm((768), eps=1e-05  
elementwise\_affine=True)?

是的，你的理解是正确的。在GPT-2模型中，`last\_hidden\_state`是模型最后一层的输出，它包含了模型对输入句子的理解。这个输出可以被用来预测下一个词，也可以被用来提取句子的语义。

在预测下一个词的情况下，`last\_hidden\_state`会被输入到模型的输出头（`lm\_head`）。这个输出头是一个线性层，它将`last\_hidden\_state`转化为一个词汇表大小的向量。

然后通过softmax函数得到每个词的概率。

在提取句子嵌入的情况下，我们通常会对`last\_hidden\_state`进行某种形式的聚合，例如取平均值或者使用某种注意力机制，得到一个固定大小的向量。

至于`(`ln\_f): LayerNorm((768,), eps=1e-05, elementwise\_affine=True)`，这是模型最后一层的层归一化（Layer Normalization）操作。在模型的前向传播过程中，这个操作会被应用到`last\_hidden\_state`上，然后得到的结果会被输入到输出头中。但是，如果你直接从模型中取出`last\_hidden\_state`，这个操作并不会被应用。如果你想应用这个操作，你需要手动调用它，例如`model.transformer.ln\_f(last\_hidden\_state)`。

[Continue this conversation](#)

[Report content](#) | [Terms of use](#) | [Privacy policy](#)