GPT

Generative Pretrained Transformer

我觉得要深入了解一个架构，应该去探索其进化历程

这个文档将探讨 GPT 系列的论文

以及 GPT2 源码架构的分析

GPT1

Improving Language Understanding by Generative Pre-Training

Abstract

> Although large unlabeled text corpora are abundant, labeled data for learning these specific tasks is scarce, making it challenging for discriminatively trained models to perform adequately.

有标签的的数据少，以往的判别式模型效果差作为背景

> We demonstrate that large gains on these tasks can be realized by generative pre-training of a language model on a diverse corpus of unlabeled text, followed by discriminative fine-tuning on each specific task.

无监督预训练 + 有监督任务微调

有监督微调是当年 NLP 任务的主流模式

而无监督预训练是 OpenAI 的创新点，即本文的卖点

切合本文标题 *"Improving Language Understanding by Generative Pre-Training"*

Introduction

> Leveraging more than word-level information from unlabeled text, however, is challenging for two main reasons.

想从无标注文本中提取出超越**词级别**的信息有两个主要难点：

1. **目标不明确**：很难判断哪种优化目标对学习到的文本表示最有利于迁移（如语言建模、机器翻译、话语连贯性等，适用于不同任务）。
2. **迁移方法不确定**：如何将学习到的表示高效迁移到目标任务也是一个挑战。现有方法通常需要在模型架构上进行特定任务的调整，或者使用复杂的学习方案和额外的学习目标（说白了就是不同任务设计不同架构），这些都增加了难度。

> In this paper, we explore a semi-supervised approach for language understanding tasks using a combination of unsupervised pre-training and supervised fine-tuning.

本文提出了一种**半监督学习**的方法，结合了无监督的预训练和有监督的微调。

其目标是学习一种**通用表示**，并能轻松地迁移到多个任务中。

现在看来没什么，在当时这是开创式尝试。

> For our model architecture, we use the **Transformer** [62], ... provides us with a more structured memory for handling **long-term dependencies in text**, compared to alternatives like recurrent networks, resulting in **robust transfer performance** across diverse tasks.

Transformer 模型通过自注意力机制更好地处理文本中的**长程依赖**，使其在不同任务上的**迁移表现**更加稳健。

长距离语义理解，这个点确实是 GPT 成功的核心要素，也是我在学习 Transformer 的时候着重探索的点；

迁移表现，其稳健性是我没想到的优势，或者说在读论文了解背景前没往这个方向思考，

不过确实，有了更大的模型更多的数据，所学到的底座的通用的知识更多

像人一样，对于某个任务也许很快就能上手，这也是因为人对于处事有基本的素质

> During transfer, we utilize task-specific input adaptations derived from traversal-style approaches [52], which process structured text input as a single contiguous sequence of tokens. As we demonstrate in our experiments, these adaptations enable us to fine-tune effectively with minimal changes to the architecture of the pre-trained model.

在微调阶段，本文使用了基于**遍历式处理**的任务特定输入调整，这将结构化文本输入处理为连续的token序列。

这种方法使得微调过程无需对预训练模型的架构做大幅度改动。

本文方法在四种语言理解任务上进行了评估，分别是：

1. **自然语言推理**（如文本蕴涵任务）
2. **问答**（如RACE数据集）

3. **语义相似性**（如句子间的语义比较）
4. **文本分类**

本文提出的通用任务无关模型**超越了针对每个任务设计的传统架构**，在12个任务中有9个达到了最新的技术水平。

常识推理任务（Stories Cloze Test）上取得了**8.9%的提升**

在问答任务（RACE）上取得了**5.7%的提升**

在文本蕴涵（MultiNLI）任务上提升了**1.5%的提升**

以及预训练模型在四种不同环境下的**零样本（zero-shot）行为**，展示了其获取了对下游任务有用的语言知识。

Related Work

相关工作部分的逻辑是提出当前技术瓶颈和难点，阐述本文合理性和必要性。

背景和难点：

1. **词嵌入**（word2vec、GloVe）和即便是**句子级别嵌入**（phrase-level or sentence-level embeddings），都难以捕捉更高层次的语义，挑战如何有效转移这些表示用于下游任务。
2. **LSTM 模型**的结构限制，难以处理长程依赖。
3. 将预训练模型的**隐藏表示**作为辅助特征，这种方法通常需要为每个任务增加大量新参数，增加模型复杂度和开发难度。

因此，本文通过采用 **Transformer 架构**，利用其多头自注意力机制有效捕捉文本中**深层次语义关系**的**长程依赖**，而无需像LSTM那样受限于短程依赖；同时，本文提出的**目标任务有监督微调**，避免了大量任务特定架构调整，有效简化了开发复杂度，证明了这种新方法论的必要性和合理性。

Framework

Unsupervised pre-training

Loss Function

> Given an unsupervised corpus of tokens $U = \{u_1, \ldots, u_n\}$, we use a standard language modeling objective to maximize the following likelihood:
>
> $$L_1(U) = \sum_i \log P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta)$$

> where k is the size of the context window, and the conditional probability P is modeled using a neural network with parameters Θ. These parameters are trained using Stochastic Gradient Descent .

Θ 是神经网络/模型参数；k 是一个窗口值，若没有则计算复杂度会几何式增长

在**无监督预训练**阶段，模型根据前 k 个 token 预测下一个 token

通过交叉熵 CrossEntropy 计算损失并求和，得到预训练模型的损失函数 L_1

其中，参数通过随机梯度下降法 SGD 进行优化

Structure

> In our experiments, we use a multi-layer Transformer decoder for the language model, which is a variant of the transformer. This model applies a multi-headed self-attention operation over the input context tokens followed by position-wise feedforward layers to produce an output distribution over target tokens:

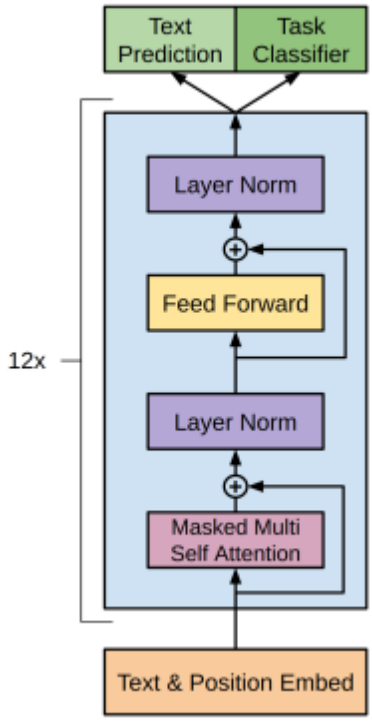
$$h_0 = UW_e + W_p$$

$$h_l = transformer\_block(h_{l-1}),\ \forall l \in [1, n]$$

$$P(u) = softmax(h_n W_e^T)$$

> where $U = (u_{-k}, \ldots, u_{-1})$ is the context vector of tokens, n is the number of layers, W_e is the token embedding matrix, and W_p is the position embedding matrix.

这三行公式很清晰地展示了 GPT 的架构，



1. 嵌入表示层：将 tokens 集合 U 进行语义嵌入的映射 W_e，再加上位置信息 W_p，输出 h_0
2. 解码器：将输出的 h_0 迭代入 n 层解码器中，输出 h_1
3. 输出层：将 h_1 反向映射回词汇表，得到每个 token 置信度的未归一化表示；$softmax$ 后输出下一个 token 对应的词概率

这里多说一句，在训练的时候，将训练批和目标批错位处理，最后输出的 $P(u)$ 是目标 token 对应的概率大小

Supervised fine-tuning

> After training the model with the objective in Eq. 1, we adapt the parameters to the supervised target task. We assume a labeled dataset C, where each instance consists of a sequence of input tokens $x_1, \ldots, x_m$, along with a label y. The inputs are passed through our pre-trained model to obtain the final transformer block' s activation h_m^l, which is then fed into an added linear output layer with parameters W_y to predict y
>
> $$P(y|x_1, \ldots, x_m) = softmax(h_m^l W_y)$$
>
> This gives us the following objective to maximize:
>
> $$L_2(C) = \sum_{(x,y)} \log P(y|x_1, \ldots, x_m)$$

在**有监督微调**阶段，利用的是带标签的数据集 C，即序列 $x_1, \ldots, x_m$ 有标签 y

假设一个批次将一个序列输入到预训练好的模型中，最后的激活层输出 h_m^l，这里还是一串置信度

相比预训练阶段，得到标签 y 的概率前还需要进行一次线性变换，这里是想让模型再学到一些特定任务的特征

最后经过 $softmax$ 得到不同的输出概率

> We additionally found that including language modeling as an auxiliary objective to fine-tuning helped learning by (a) improving generalization of the supervised model, and (b) accelerating convergence. This is in line with prior work [50, 43], who also observed improved performance with such an auxiliary objective. Specifically, we optimize the following objective (with weight λ):
>
> $$L_3(C) = L_2(C) + \lambda \cdot L_1(C)$$
>
> Overall, the only extra parameters we require during fine-tuning are W_y, and embeddings for delimiter tokens (described below in Section 3.3).

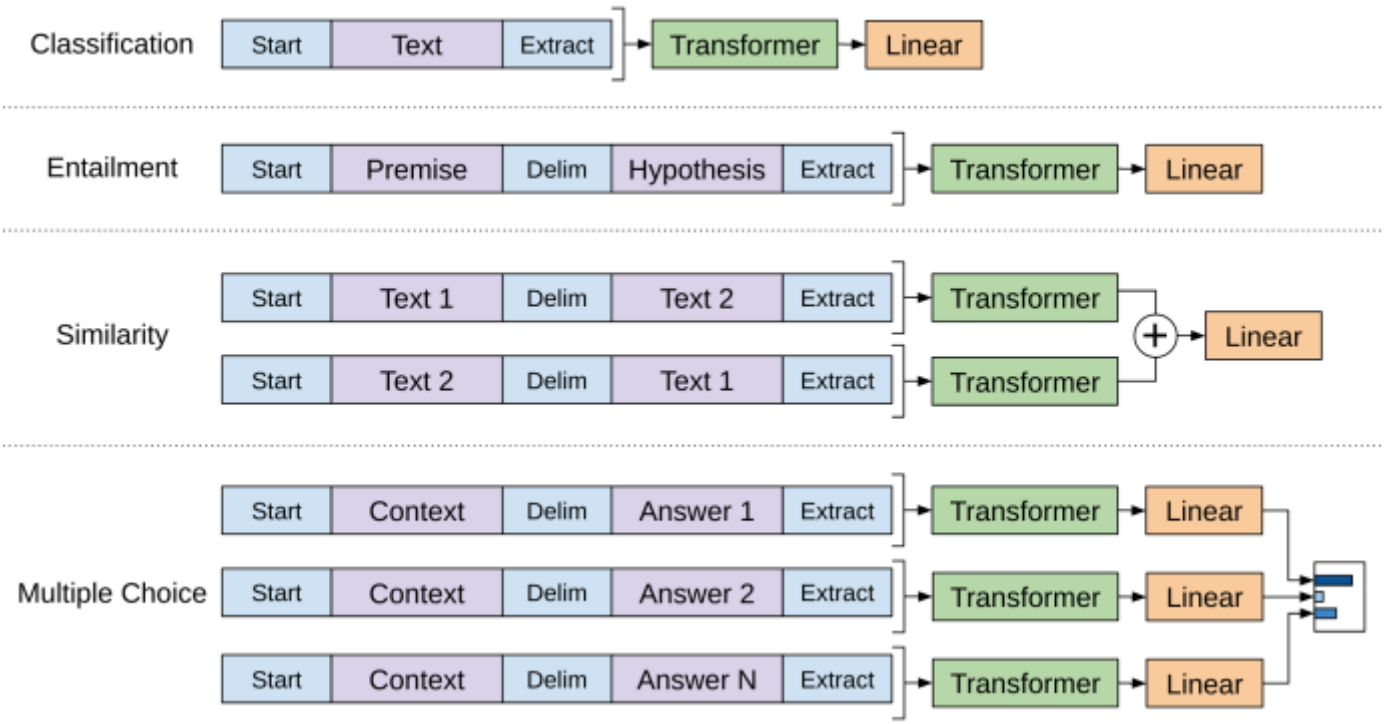
这是微调中的损失函数，为了获取特定任务的表现能力，采用 L_2 损失

同时为了维持模型原有的泛化能力，加上了 L_1 并赋予可调超参数

Task-specific input transformations

> We provide a brief description of these input transformations below and Figure 1 provides a visual illustration. All transformations include adding randomly initialized start and end tokens (hsi, hei).
>
> - **Textual entailment**: For entailment tasks, we concatenate the premise p and hypothesis h token sequences, with a delimiter token ($) in between.
> - **Similarity**: For similarity tasks, there is no inherent ordering of the two sentences being compared. To reflect this, we modify the input sequence to contain both possible sentence orderings (with a delimiter in between) and process each independently to produce two sequence representations h_m^l, which are added element-wise before being fed into the linear output layer.

- **Question Answering and Commonsense Reasoning**: For these tasks, we are given a context document z , a question qqq, and a set of possible answers a_k . We concatenate the document context and question with each possible answer, adding a delimiter token in between to get $[z; q; \$; a_k]$. Each of these sequences is processed independently with our model and then normalized via a softmax layer to produce an output distribution over possible answers.



- **文本蕴涵任务**：将前提（premise）和假设（hypothesis）的token序列拼接在一起，并在中间加入一个分隔符（$）。
- **相似性任务**：由于两个句子之间没有固有的顺序，我们将输入序列修改为包含两种句子顺序的形式（中间使用分隔符），分别处理每个顺序，生成两个序列表示，然后将它们逐元素相加，最后输入线性输出层。
- **问答与常识推理任务**：给定上下文文档 z 、问题 q 和一组可能的答案 a_k ，我们将上下文和问题与每个可能的答案拼接在一起，并加入分隔符以形成输入序列 $[z; q; \$; a_k]$ 。每个序列独立通过模型处理，最后通过softmax层归一化，生成可能答案的概率分布。

Setups

Model specifications:

Our model largely follows the original transformer work [62]. We trained a 12-layer decoder-only transformer with masked self-attention heads (768 dimensional states and 12 attention heads). For the position-wise feed-forward networks, we used 3072-dimensional inner states. We used the *Adam* optimization scheme [27] with a max learning rate of 2.5e-4. The learning rate was increased linearly from zero over the first 2000 updates and annealed to 0 using a

> cosine schedule. We train for 100 epochs on mini-batches of 64 randomly sampled, contiguous sequences of 512 tokens. Since layernorm [2] is used extensively throughout the model, a simple weight initialization of *N(0, 0.02)* was sufficient. We used a bytepair encoding (BPE) vocabulary with 40,000 merges [53] and residual, embedding, and attention dropouts with a rate of 0.1 for regularization. We also employed a modified version of L2 regularization proposed in [37], with w = 0.01 on all non-bias or gain weights. For the activation function, we used the *Gaussian Error Linear Unit (GELU)* [18]. We used learned position embeddings instead of the sinusoidal version proposed in the original work. We use the ftfy library to clean the raw text in *BooksCorpus*, standardize some punctuation and whitespace, and use the spaCy tokenizer.

Dataset

BooksCorpus *Token-level perplexity: 18.4*

- 7,000+ unique books
- Long contiguous text stretches
- *1B Word Benchmark (alternative)

Model

1. **Type**: Transformer - Decoder
2. **Layers**: `n_layer=12`
3. **Attention Heads**: `h=12`
4. **Hidden Dimensionality**:
   - State Dimensionality: `n_embd=768`
   - Position-wise Feed-forward Inner States: `n_inner=4*n_embd`，即 `n_inner=3072`
5. **Embedding**:
   - Vocabulary Size (Byte Pair Encoding - BPE): `vocab_size=40,000`
6. **Dropout**: `dropout=0.1`
   - Residual: `resid_pdrop=0.1`
   - Embedding: `embd_pdrop=0.1`
   - Attention: `attn_pdrop=0.1`
7. **Activation Function**: Gaussian Error Linear Unit
   - `activation_function='GELU_new'`
8. **Layer Normalization**: LayerNorm (with simple weight initialization $N(0, 0.02)$)
9. **Regularization**:

- Modified L2 Regularization: $w = 0.01$ on non-bias or gain weights

Training

1. **Epochs**: 100 epochs
2. **Batch Size**: 64 sequences
3. **Sequence Length**: 512 tokens
4. **Weight Initialization**: $N(0, 0.02)$
5. **Optimizer**:
   - Adam
   - Learning Rate
     - Max Learning Rate: 2.5e-4
     - Warmup (first 2000 updates)
     - Cosine annealing to zero

> **Fine-tuning details:**
> Unless specified, we reuse the hyperparameter settings from unsupervised pre-training. We add dropout to the classifier with a rate of 0.1. For most tasks, we use a learning rate of 6.25e-5 and a batch size of 32. Our model fine-tunes quickly, and 3 epochs of training were sufficient for most cases. We use a linear learning rate decay schedule with warmup over 0.2% of training. λ was set to 0.5.

Fine-tuning

1. **Learning Rate**: 6.25e-5
2. **Batch Size**: 32
3. **Epochs**: 3 (sufficient for most tasks)
4. **Learning Rate Schedule**: Linear decay, 0.2% warmup
5. **Dropout**: 0.1 (applied to the classifier)
6. **Auxiliary Objective**: Language modeling (weight $\lambda = 0.5$)

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	**Average** -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

在 OpenAI 公布 GPT1 后的三个月，Google AI 发布他们的模型 BERT

相反，他们用的是 Transformer 的编码器；而且，数据更大，模型更大，效果更好

当然，OpenAI 为了反击不能仅仅是“大力出奇迹”，更不能反水换阵营

所以在维持原本的技术路线上，在 GPT2 中做了一些创新

GPT2

Language Models are Unsupervised Multitask Learners

> *"Our largest model, GPT-2, is a **1.5B parameter** Transformer that achieves state of the art results on 7 out of 8 tested language modeling datasets in a **zero-shot** setting but still underfits WebText."*
>
> *"We demonstrate **language models can perform down-stream tasks in a zero-shot setting** – without any parameter or architecture modification. "*

GPT2 在模型架构上没有改动，核心目标和思想是 *“预训练模型在下游任务上具备一定表现力”*

换言之，其核心概念和卖点是 *"Unsupervised Multitask Learning"* 能在 *"Zero-shot Setting"* *具备竞争力和前景*

原文略长稍杂，GPT2 有一定表现能力的原因，根据我的理解可以总结为以下两个方面：

1. **更大的数据，更大的模型**
    a. GPT-2 构建了一个全新的数据集 **WebText**，它包含了超过 **800万** 网页文档，总计 **40GB** 的文本数据
    b. **GPT-2** 是在 **GPT** 基础上大幅度扩大了模型的参数规模，达到了 **1.5 billion（15亿）参数**，远超 GPT 的 **110 million（1.1亿）参数**

Parameters	Layers	d_{model}
117M	12	768
345M	24	1024
762M	36	1280
1542M	48	1600

2. **优质的数据结构，优质的数据源**

> If listened carefully at 29:55, a conversation can be heard between two guys in French: “- **Comment on fait pour aller de l’ autre cote? -Quel autre cot ´ e?´ ”** , which means “- **How do you get to the other side? - What side?”**

这个还是很有意思的，首先举例论证了无监督学习下游任务的可行性

原因是数据集中存在大量的特定任务的数据

可能某一个法语电影的影评中，就会有类似的台词分析，便涉及了英法语的翻译

> ... That is, it should model $p(output|input, task)$ ...
>
> For example, a translation training example can be written as the sequence **(translate to french, english text, french text)**. Likewise, a reading comprehension training example can be written as **(answer the question, document, question, answer)**.

由于 zero-shot 的设定，在下游任务处理的过程中，不能像 GPT1 一样有分隔符来参与微调；换句话说，模型不认识那些符号

所以在做预训练阶段，要使得训练数据更接近自然问答：

~~(...premise, $, hypothesis...)~~
~~(...question, $, answer...)~~

(translate to french, english text, french text)
(answer the question, document, question, answer)

这便是所谓的 **Prompt**，结合上面的例子，GPT2 可能得到的训练数据是：

(translate to french, “- How do you get to the other side? - What side?”，“-Comment on fait pour aller de l’ autre cote? -Quel autre cot ´ e?´ ”)

> Manually filtering a full web scrape would be exceptionally expensive so as a starting point, we scraped all outbound links from **Reddit**, a social media platform, which received at least 3 **karma**. This can be thought of as a heuristic indicator of whether other users found the link interesting, educational, or just funny.

由于 Common Crawl 的数据信噪比比较低，本次训练数据集 **WebText** 爬的是 Reddit 上的数据

同时，该数据包含一种“标签” Karma，代表了关注度

筛选 Karma >= 3 的数据，在保持数量的前提下，拥有较高的质量

“一篇论文的价值取决于其**新意度×有效性×解决问题大小**”

GPT 的技术路线不变，解决的仍然是极有挑战性的生成式任务；zero-shot 在新意度上是拉满的，也逐渐有了在未来大火的 ChatGPT 的雏形

唯一不足点就是效果不尽人意，这也是在情理之中

那么 GPT3 便改进了这一问题

GPT3

Language Models are Few-Shot Learners

> However, a major limitation to this approach is that while the architecture is task-agnostic, there is still a need for task-specific datasets and task-specific fine-tuning: to achieve strong performance on a desired task typically requires fine-tuning on a dataset of thousands to hundreds of thousands of examples specific to that task.

虽然不需要改变架构和参数的微调，GPT2 还是需要特定任务的数据集为模型提供特定任务的处理能力，

"Removing this limitation would be desirable, for several reasons"

以下几点说明了消除这个限制的必要性：

> First, from a practical perspective, the need for a large dataset of labeled examples for every new task limits the applicability of language models.

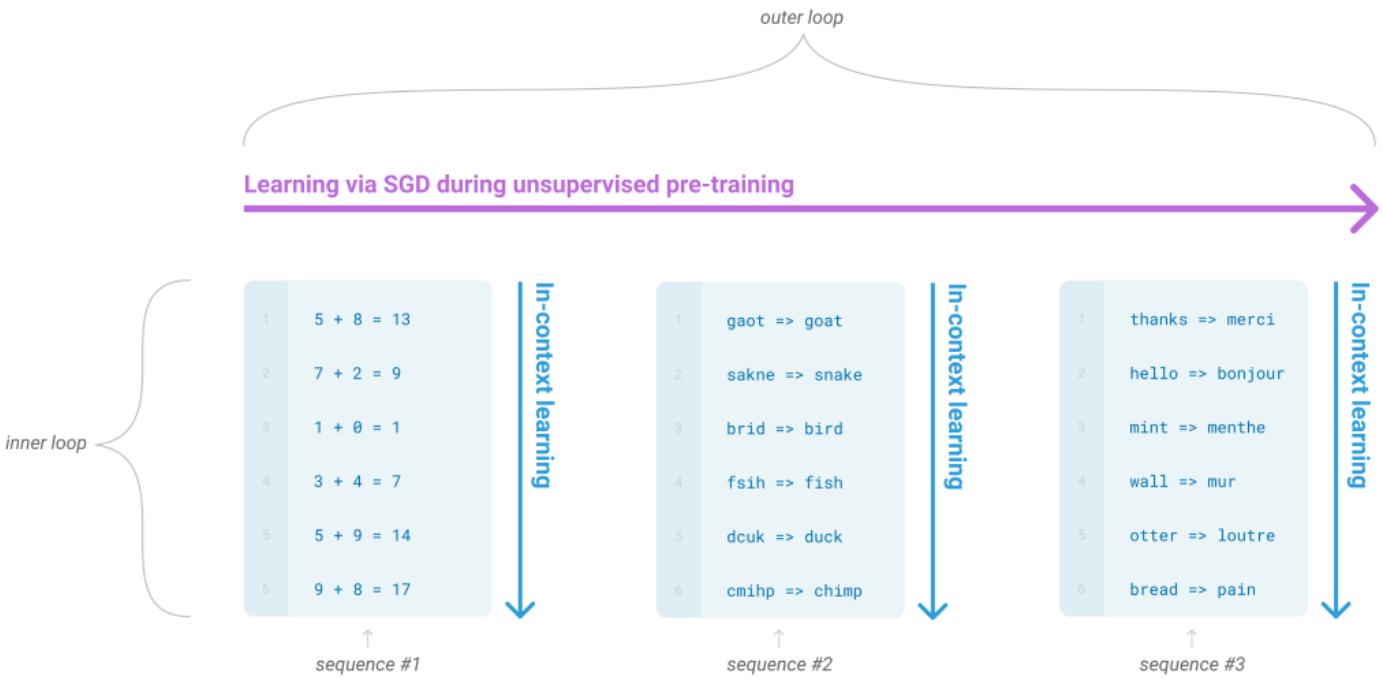
1. **实际应用受限：** 对每个新任务都需要大量标注数据，导致非主流语言任务（如语法纠正、抽象概念生成、短篇小说批评）难以收集大量的监督训练数据，尤其当每个新任务都需要重复这一过程。

> Second, the potential to exploit spurious correlations in training data fundamentally grows with the expressiveness of the model and the narrowness of the training distribution.

2. **泛化能力问题：** 模型的表达能力越强，利用训练数据中虚假关联的可能性就越大；这可能导致模型过度拟合训练分布，无法很好地泛化到未见过的数据。

> Third, humans do not require large supervised datasets to learn most language tasks – a brief directive in natural language (e.g. “please tell me if this sentence describes something happy or something sad”) or at most a tiny number of demonstrations (e.g. “here are two examples of people acting brave; please give a third example of bravery”) is often sufficient to enable a human to perform a new task to at least a reasonable degree of competence.

3. **与人类学习的差距：** 人类在学习大多数语言任务时，不需要大量的监督数据。简短的自然语言指令或少量示例通常足以让人类在新任务上达到合理的熟练程度。



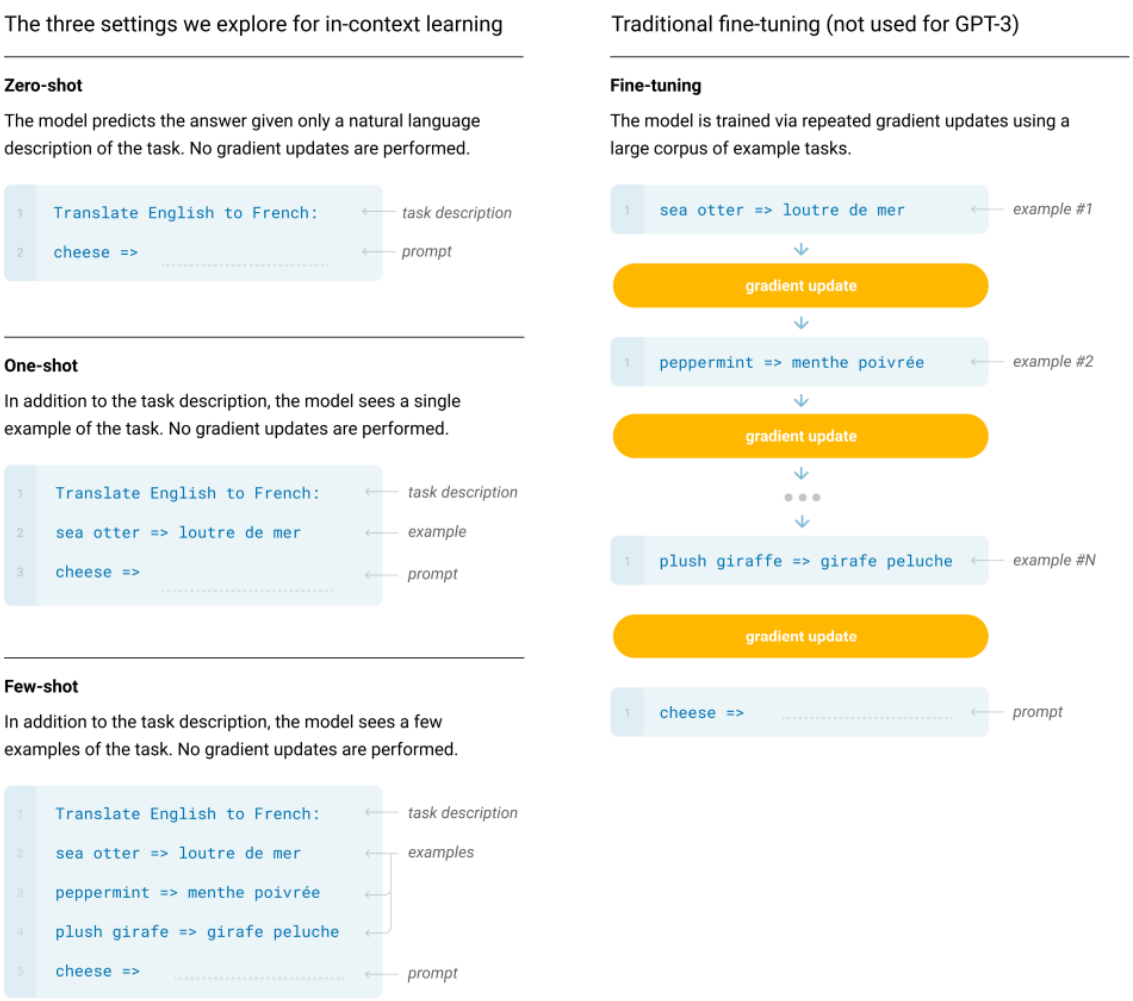
> One potential route towards addressing these issues is **meta-learning**1 – which in the context of language models means the model develops a broad set of skills and pattern recognition abilities at training time, and then uses those abilities at inference time to rapidly adapt to or recognize the desired task (illustrated in Figure 1.1).

所以 OpenAI 提出个方法叫 Meta-Learning，其灵魂是所谓的 In-Context Learning 让模型理解不同的任务

我觉得还蛮扯淡的，说好听点叫元学习，说难听点就是把一坨更大的数据炼出一坨更大的模型

> For each task, we evaluate GPT-3 under 3 conditions: (a) “few-shot learning”, or in-context learning where we allow as many demonstrations as will fit into the model’s context window (typically 10 to 100), (b) “one-shot learning”, where we allow only one demonstration, and (c) “zero-shot” learning, where no demonstrations are allowed and only an instruction in natural language is given to the model. GPT-3 could also, in principle, be evaluated in the traditional fine-tuning setting, but we leave this to future work.

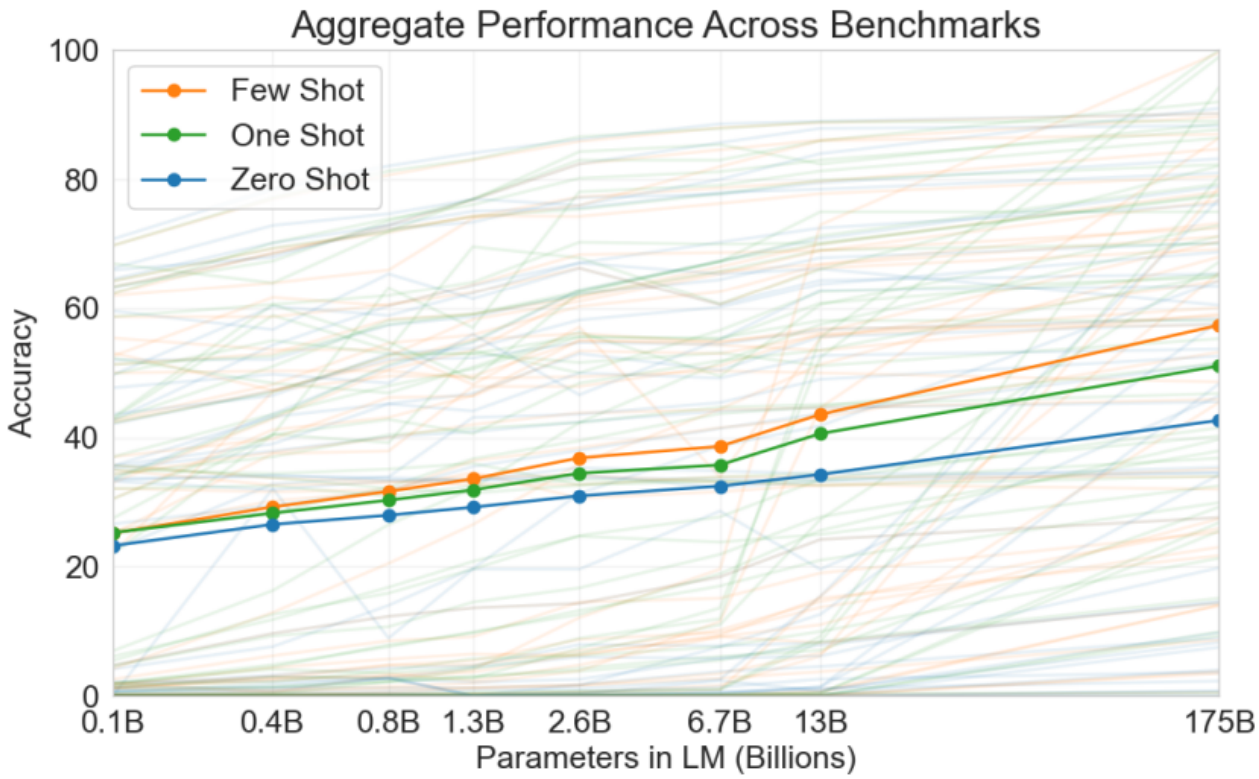
最有价值的是其评估利用了：



- **Few-shot学习：** 在模型的上下文窗口中提供尽可能多的示例。
- **One-shot学习：** 仅提供一个示例。
- **Zero-shot学习：** 不提供任何示例，只给出自然语言的指令。

相比以往的微调，不需要梯度更新，那么大的模型也很难更新

这更符合人类的思维范式，当然大力出奇迹确实带来了更好的效果：

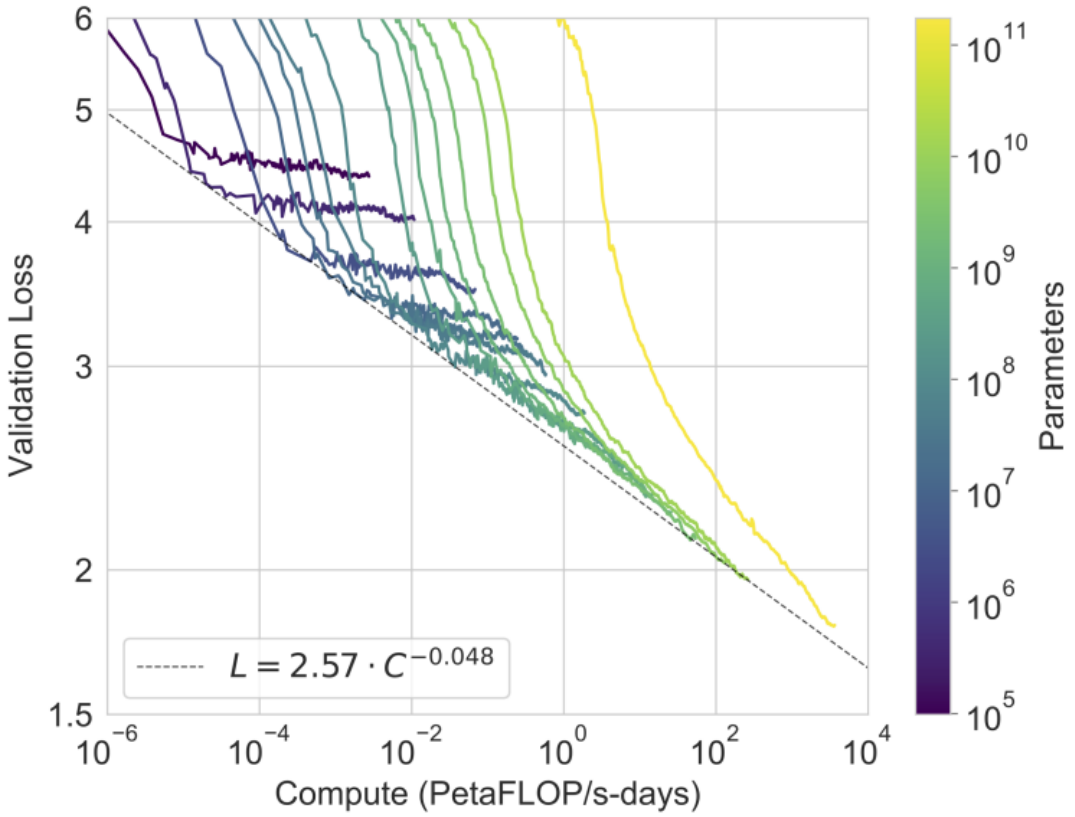


此外，更大数据集的前提是，信噪比不能大幅下降；换句话说，没有用的垃圾数据再多也没用

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

可以看出，虽然 Common Crawl 的数据量非常大，但是采样数并不多

侧面反应作者对该数据集的认可度不高，反而是偏爱 WebText2 的，即 Reddit 上的数据



这个图很有意思，可以称之为近些年大模型进化趋势的可视化

随着 Compute 次数的上升，Validation Loss 随之下降

其中，由于模型的架构相同，梯度下降几乎平行

且越小的模型越早到达 Critical Point；换句话说，模型越大，Loss 越低，效果越好

最重要的是，Loss 的瓶颈/最低点遵循参数和算力指数级上升而线性下降

> Smooth scaling of performance with compute. Performance (measured in terms of cross-entropy validation loss) follows a power-law trend with the amount of compute used for training. The power-law behavior observed in [KMH+20] continues for an additional two orders of magnitude with only small deviations from the predicted curve.

这种**幂律**趋势意味着性能的改善速度逐渐变缓：当计算资源加倍时，性能会持续提升，但提升的幅度会越来越小。

这种现象反映了深度学习模型的一个常见现象——即在资源充足时，性能可以持续提升，但边际收益递减。

从前三篇能看到一个大致的 GPT 演化过程：

特定任务模型 → 无监督预训练 + 有监督微调 → 无监督预训练（更优数据源和数据结构）→ 无监督预训练（更大数据信噪比和参数）

诚然，各种实验证明这样的演化取得更好的表现和分数，

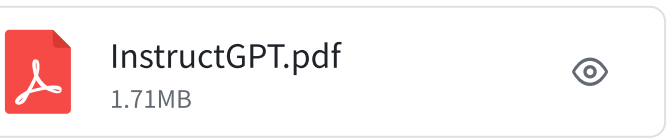
但此时的 GPT 只是一个模型，还算不上一个产品或服务，

因为它缺少了一个关键要素——“人性”

InstructGPT 和后来的 ChatGPT 将落脚点放在了人身上

InstructGPT

Training language models to follow instructions with human feedback



> Making language models bigger does not inherently make them better at **following a user’s intent**. For example, large language models can generate outputs that are **untruthful, toxic, or simply not helpful to the user**. In other words, these models **are not aligned with their users**.

“Alignment “，是本文的核心要素

”*Aligning with Users*”，是本文的核心目标

这里的 Align 不仅指人机更好地交互，而且代表了大模型的安全性

> In this paper, we show an avenue for aligning language models with user intent on a wide range of tasks by **fine-tuning with human feedback**.

要知道本次 OpenAI 的目的，就不会觉得又回到微调策略是一种打脸

以往的模型是为了变得更大更强，但如果性能的增长是以指数级增长的计算资源为代价，那么很容易到达瓶颈

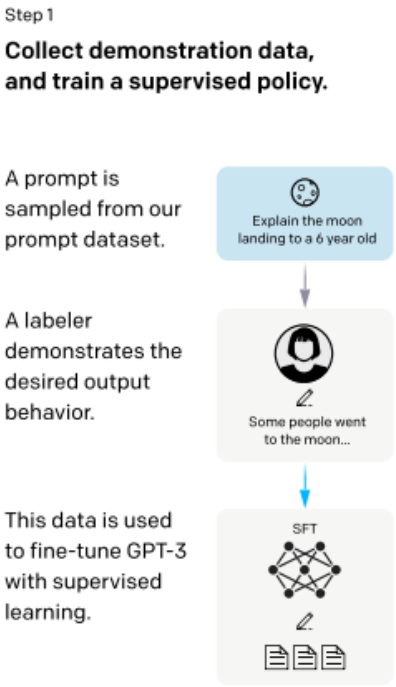
更符合人类的需求 + 更具备安全性 是模型的纵向发展

Step 1

Supervised Fine-Tuning (SFT)

> Starting with a set of labeler-written prompts and prompts submitted through the OpenAI API, we collect a dataset of labeler demonstrations of the desired model behavior, which we use to fine-tune GPT-3 using supervised learning.

提示词取样 → 人工答案 → **有监督微调 GPT3**



优点：

- 让模型更符合人类预期的答案
- 使模型规范化，避免不实的有害的回答

缺点：

- 成本高昂
- 数据量少

> Supervised fine-tuning (SFT). We fine-tune GPT-3 on our labeler demonstrations using supervised learning. We trained for **16 epochs**, using a **cosine learning rate decay**, and residual dropout of 0.2. We do our final SFT model selection based on the RM score on the validation set. Similarly to Wu et al. (2021), we find that our SFT models overfit on validation loss after 1 epoch; however, we find that training for more epochs helps both the RM score and human preference ratings, despite this overfitting.

- **Epochs:** 16
- LR**:** Cosine Learning Rate Decay
- **Residual Dropout:** 0.2

SFT 的最终模型选择是基于验证集上的 RM（Reward Model）得分。

虽然验证损失在第1轮后即过拟合，但多轮训练对提升 RM 得分和人工偏好评级都有帮助。

Step 2

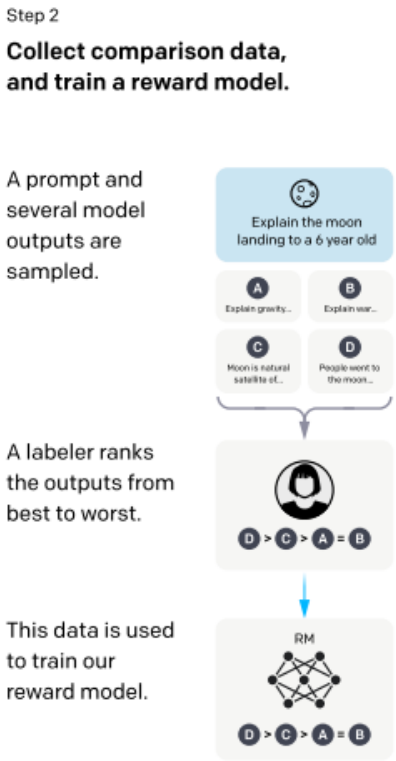
Reward Model (RM) Training

> We then collect a dataset of rankings of model outputs, which we use to further fine-tune this supervised model using reinforcement learning from human feedback.

取样提示词和多个回答 → 人工排序/排名 → 数据用于训练奖励模型

优点：

- 简单易用



- 相对更大的数据集

缺点：

- 人工成本

> In order to speed up comparison collection, we present labelers with anywhere between K = 4 and K = 9 responses to rank. This produces K 2 comparisons for each prompt shown to a labeler. Since comparisons are very correlated within each labeling task, we found that if we simply shuffle the comparisons into one dataset, a single pass over the dataset caused the reward model to overfit.5 Instead, we train on all K 2 comparisons from each prompt as a single batch element. This is much more computationally efficient because it only requires a single forward pass of the RM for each completion (rather than K 2 forward passes for K completions) and, because it no longer overfits, it achieves much improved validation accuracy and log lo

从 SFT 模型开始，去掉最终的嵌入层，训练一个模型输入提示和响应，输出一个标量奖励。

这里应该是移除了 Softmax(Linear(n_embd, vocab_size)) 换成了 Linear(n_embd, 1)

在研究中，主要使用 6B 参数的 RM，因为 175B 的 RM 训练可能会不稳定。

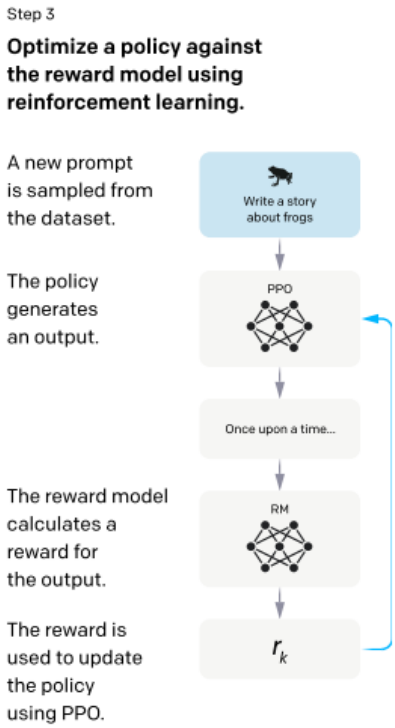
训练 RM 的方法基于比较两个模型输出的方式来训练，使用交叉熵损失函数，并基于人类标注者的偏好进行优化。

$$\text{loss}(\theta) = -\frac{1}{K^2}\mathbb{E}_{(x,y_w,y_l)\sim D}\left[\log(\sigma(r_\theta(x,y_w) - r_\theta(x,y_l)))\right]$$

其中，$r_\theta(x,y)$ 是奖励模型的输出，y_w 是标注者偏好的响应，y_l 是另一响应。

为了加速比较数据的收集，标注者会为每个提示展示 4 到 9 个响应，并进行排序。

每次标注任务内的比较是相关的，因此训练时将这些比较作为单个批处理元素，可以减少模型的过拟合并提高验证准确率。



Step 3

Reinforcement Learning via Proximal Policy Optimization (PPO) on this Reward Model

> We call the resulting models InstructGPT

取样新的提示词 → GPT3 生成输出 → RM 计算奖励 → 通过 PPO 更新模型

> Once again following Stiennon et al. (2020), we fine-tuned the SFT model on our environment using PPO (Schulman et al., 2017). The environment is a bandit environment which presents a random customer prompt and expects a response to the prompt. Given the prompt and response, it produces a reward determined by the reward model and ends the episode. In addition, we add a per-token KL penalty from the SFT model at each token to mitigate overoptimization of the reward model. The value function is initialized from the RM. We call these models “PPO.”

继 SFT 之后，使用强化学习的 PPO（Proximal Policy Optimization）算法对模型进行微调。

在这个环境中，模型会根据随机生成的用户提示提供响应，奖励由奖励模型决定，并在响应后结束当前回合。

PPO 的训练目标函数为：

$$\text{objective}(\phi) = \mathbb{E}_{(x,y)\sim D_{\pi_{RL}}}[r_\theta(x,y) - \beta \log(\pi_{RL}(y|x)/\pi_{SFT}(y|x))] + \gamma \mathbb{E}_{x\sim D_{\text{pretrain}}}[\log(\pi_{RL}(x))]$$

其中，π_{RL} 是强化学习策略，π_{SFT} 是监督微调后的模型，β 和 γ 是控制 KL 惩罚和预训练梯度的系数。

为了避免过度优化，添加了一个基于 SFT 模型的 KL 惩罚。

此外，还引入了 "PPO-ptx" 模型，将预训练梯度与 PPO 梯度混合，以修复公共 NLP 数据集上的性能回退。

这里涉及 RL 的相关知识，后面补一下再来学习这段。

读了 GPT 的四篇论文，也只是了解了一点皮毛

真正讲架构和技术的部分并不多，尤其是越往后工程味道越浓。

不过还是了解到了很多策略和方法论，

拓宽了我学术和工业的的眼界。

只能说，这个领域需要不断地学习，

正如强化学习中的“在线学习”，不断地给反馈才能持续优化和进步！

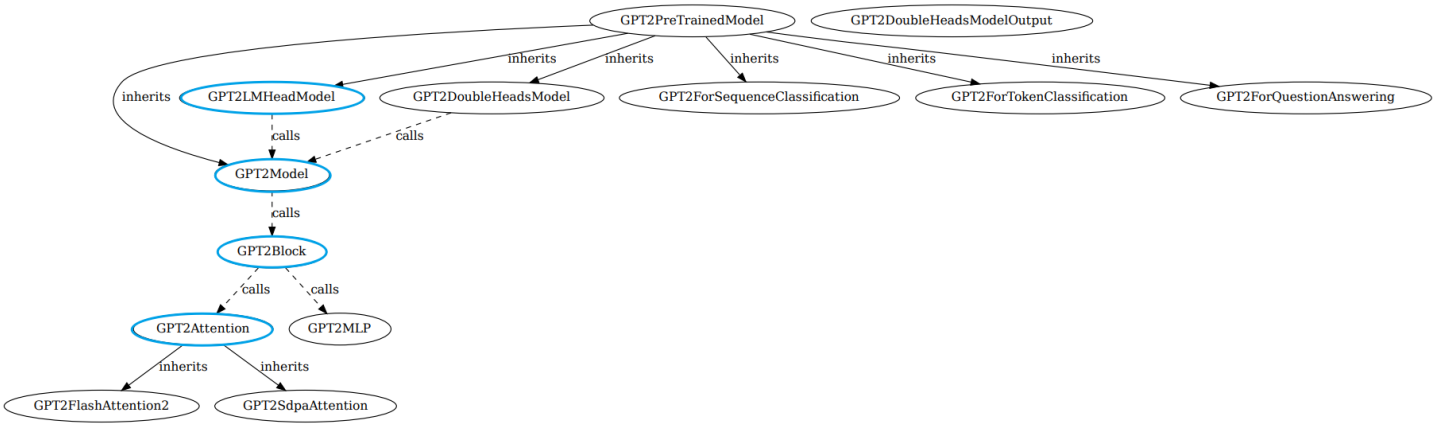
GPT Architecture

在学完 Transformer 之后，需要研究 GPT 是怎么实现的。

在实现上，和自己的构想有什么区别。

以下来解读 HuggingFace Transformer 中 GPT2 模块的源码，

窥探 OpenAI 设计师的智慧。



1. `GPT2LMHeadModel`：负责语言模型的最后一步，将隐藏状态投影到词汇表的维度空间中，生成词预测。
2. `GPT2Model`：基础模型，负责从输入中提取出隐藏状态。
3. `GPT2Block`：组成 GPT2Model 的主要单元，包含注意力机制和前馈网络。
4. `GPT2Attention`：实现自注意力机制，计算每个 token 与其他 token 之间的依赖关系。
5. `*GPT2Config`：用于配置 GPT2 模型的参数，包括模型的结构、维度、注意力机制等重要配置。

---

1. GPT2LMHeadModel

`GPT2LMHeadModel` 是 GPT2 模型的语言建模头部（language modeling head），负责将来自 `GPT2Model` 的隐藏状态投影到词汇表的维度，用于生成每个位置上词汇的预测分布。

它在自回归生成任务（如文本生成）中尤为重要。这个类还可以计算语言建模的损失函数，通常用于训练任务。

- 它的主要任务是接收 `GPT2Model` 的输出，并通过一个线性投影层将隐藏状态转化为词汇表大小的 logits。
- 如果提供了标签，它还能计算交叉熵损失，用于模型训练。

```
class GPT2LMHeadModel(GPT2PreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.transformer = GPT2Model(config)

        # 将 self.transformer 中最后一个 Block 层输出的隐藏状态的
        # 最后一个维度由 config.n_embd 投影为 config.vocab_size
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)

    def forward(self, input_ids, past_key_values=None, attention_mask=None,
token_type_ids=None,
                position_ids=None, head_mask=None, inputs_embeds=None,
labels=None,
                use_cache=None, output_attentions=None,
output_hidden_states=None,
                return_dict=None):

        # 是否以字典形式返回结果，默认为 True
        return_dict = return_dict if return_dict is not None else
self.config.use_return_dict

        transformer_outputs = self.transformer(
            input_ids,
            past_key_values=past_key_values,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids,
            position_ids=position_ids,
            head_mask=head_mask,
            inputs_embeds=inputs_embeds,
            encoder_hidden_states=None, # 只在 Decoder 中使用
            encoder_attention_mask=None, # 只在 Decoder 中使用
            use_cache=use_cache,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
        )
```

```
33
34          # [batch_size, sequence_length, hidden_size]
35          hidden_states = transformer_outputs[0]
36
37          # 将 hidden_states 投影到词汇表大小，得到 logits
38          # [batch_size, sequence_length, vocab_size]
39          lm_logits = self.lm_head(hidden_states)
40
41          # 计算损失（如果提供了 labels）
42          loss = None
43          if labels is not None:
44              # shift_logits 和 shift_labels 都是为了对齐预测和标签
45              # [batch_size, sequence_length - 1, vocab_size]
46              shift_logits = lm_logits[..., :-1, :].contiguous()
47
48              # [batch_size, sequence_length - 1]
49              shift_labels = labels[..., 1:].contiguous()
50
51              # 使用 CrossEntropyLoss 计算损失
52              loss_fct = CrossEntropyLoss()
53              loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)),
    shift_labels.view(-1))
54
55          return CausalLMOutputWithCrossAttentions(
56              loss=loss,
57              logits=lm_logits,
58              past_key_values=transformer_outputs.past_key_values,
59              hidden_states=transformer_outputs.hidden_states,
60              attentions=transformer_outputs.attentions,
61          )
```

---

2. GPT2Model

`GPT2Model` 是 GPT2 的核心基础模型，负责从输入的 token 序列中提取隐藏状态。

它由一系列 `GPT2Block` 组成，每个 block 包含注意力机制和前馈网络。

- `GPT2Model` 负责将输入 token 的嵌入和位置编码相加后，通过多个 `GPT2Block` 处理输入序列，逐步提取出深层次的特征表示。
- 最终，`GPT2Model` 输出的隐藏状态可以用于语言模型头、分类任务等。

```python
class GPT2Model(GPT2PreTrainedModel):
    def __init__(self, config):
        super().__init__(config)

        # token 的编码维度，默认 768
        self.embed_dim = config.hidden_size

        # 词向量编码
        self.wte = nn.Embedding(config.vocab_size, self.embed_dim)

        # token 的位置编码
        self.wpe = nn.Embedding(config.max_position_embeddings, self.embed_dim)

        self.drop = nn.Dropout(config.embd_pdrop)

        # Block 列表，默认 12 个 Block 层
        self.h = nn.ModuleList([GPT2Block(config, layer_idx=i) for i in range(config.num_hidden_layers)])

        # 层归一化层
        self.ln_f = nn.LayerNorm(self.embed_dim, eps=config.layer_norm_epsilon)

    def forward(self, input_ids=None, past_key_values=None, attention_mask=None, token_type_ids=None,
                position_ids=None, head_mask=None, inputs_embeds=None, use_cache=None,
                output_attentions=None, output_hidden_states=None, return_dict=None):

        input_shape = input_ids.size() if input_ids is not None else inputs_embeds.size()[:-1]

        # 默认 False，是否需要输出 GPT2Model 处理过程中每个 block 的 attentions
        output_attentions = output_attentions if output_attentions is not None else self.config.output_attentions

        # 默认 False，是否需要输出 GPT2Model 中每个 GPT2Block 输出的 hidden states
        output_hidden_states = (
            output_hidden_states if output_hidden_states is not None else self.config.output_hidden_states
        )

        # 默认 True，输出 past_key_values
        use_cache = use_cache if use_cache is not None else self.config.use_cache

        if token_type_ids is not None:
```

```
            # [batch_size, sequence_len]
            token_type_ids = token_type_ids.view(-1, input_shape[-1])

         # 默认为 None，表示没有 position_ids 传入
        if position_ids is not None:
            position_ids = position_ids.view(-1, input_shape[-1])

        # 如果没有 past_key_values，则设置为 None 列表，用于后续计算
        if past_key_values is None:
            past_length = 0
            past_key_values = tuple([None] * len(self.h))
        else:
            past_length = past_key_values[0][0].size(-2)

        # 如果 position_ids 为 None，则生成绝对位置编码
        if position_ids is None:
            position_ids = torch.arange(past_length, input_shape[-1] +
past_length, dtype=torch.long, device=input_ids.device)
            position_ids = position_ids.unsqueeze(0).view(-1, input_shape[-1])

        # 词向量编码
        if inputs_embeds is None:
            inputs_embeds = self.wte(input_ids)

        # 位置编码
        position_embeds = self.wpe(position_ids)

        # 将词向量和位置编码相加
        hidden_states = inputs_embeds + position_embeds

        # 添加 dropout 防止过拟合
        hidden_states = self.drop(hidden_states)

        # 存储输出形状
        output_shape = input_shape + (hidden_states.size(-1),)

        # 存储 past_key_values
        presents = () if use_cache else None

        # 遍历所有 Block 层，逐层计算
        for i, (block, layer_past) in enumerate(zip(self.h, past_key_values)):
            outputs = block(
                hidden_states,
                layer_past=layer_past,
                attention_mask=attention_mask,
                head_mask=head_mask[i] if head_mask is not None else None,
                use_cache=use_cache,
```

```
                output_attentions=output_attentions,
            )

            hidden_states = outputs[0]

            if use_cache:
                presents = presents + (outputs[1],)

        # 经过所有 Block 层后进行最后的层归一化
        hidden_states = self.ln_f(hidden_states)

        hidden_states = hidden_states.view(output_shape)

        return BaseModelOutputWithPastAndCrossAttentions(
            last_hidden_state=hidden_states,
            past_key_values=presents,
            hidden_states=None,  # 如果 output_hidden_states 为 True，此处可返回所有 hidden states
            attentions=None  # 如果 output_attentions 为 True，此处可返回所有 attentions
        )
```

这是 GPT-2 模型的核心结构，继承自 `GPT2PreTrainedModel`。

除了核心结构，还有一些灵活的可选参数，尤其是关乎模型优化策略，很有意思。

2.1 初始化方法 `init`

值得注意其中的**模型优化**参数

```
# Model parallel
self.model_parallel = False
self.device_map = None
self.gradient_checkpointing = False
self._attn_implementation = config._attn_implementation
```

`self.model_parallel = False`

- **模型并行**是将模型的不同部分分布在多个计算设备（如GPU）上。
  这个选项主要在模型非常大，无法放入单个设备的情况下使用。
- 当 `self.model_parallel` 为 `False` 时，表示模型的所有部分都将在单一设备上运行。
  这有助于简化代码逻辑，并在设备内存足够的情况下提高效率。
- **如果需要在多个设备上并行处理，通常会将其设置为 `True`，并在后续代码中指定如何分配层。**

`self.device_map = None`

- `device_map` 是一个字典，用于映射模型的不同层或模块到特定的设备（如不同的GPU）。
- 初始值为 `None`，意味着没有设备分配。
  在实现模型并行 `self.model_parallel = True` 时，可以将某些层放在特定的设备上进行计算。
- **不同的设备可能会有不同的计算能力和内存限制，合理的映射可以提高计算效率。**

`self.gradient_checkpointing = False`

- **梯度检查点**用于在训练大型模型时节省内存。
  它通过在前向传播时不存储所有中间激活值来减少内存使用，而是在需要时重新计算它们。
- 当 `self.gradient_checkpointing` 设置为 `True` 时，模型会在前向传播中进行适当的修改，以便在反向传播时重新计算激活。
- **减少内存使用，但增加计算时间，因为需要多次计算部分前向传递。**
  这对于需要大内存的深度学习模型尤其重要，可以帮助在较小的GPU上训练更大的模型。

`self._attn_implementation = config._attn_implementation`

- 该行代码保存了注意力机制的实现方式，可能包括不同的优化或算法选择。
  `_attn_implementation` 可能指向不同的注意力算法，如标准的缩放点积注意力，或者一些变体（如稀疏注意力）。
- 例如，某些实现可能在内存或计算效率上有优势，具体选择依赖于模型需求和硬件资源。
- **使用这种配置的方式，可以在运行时灵活选择不同的实现，从而方便地进行实验或优化特定任务的性能。**

模型优化从以下三个方面理解：

- **算力**：有效利用多设备的计算能力，提高训练效率。
- **内存**：通过合理分配模型层，可以减轻单一设备的内存压力。
- **调整**：设备映射和注意力实现的选择使得模型在不同硬件条件下能够灵活调整，适应不同的训练需求。

2.2 并行化方法 `parallelize`

上述提到 `device_map` 是一个字典，用于映射模型的不同层或模块到特定的设备

这个模块细致地展示了如何通过设备映射实现并行计算，优化多GPU的使用效率

```
    @add_start_docstrings(PARALLELIZE_DOCSTRING)
    def parallelize(self, device_map=None):
        # Check validity of device_map
```

```
        warnings.warn(
            "`GPT2Model.parallelize` is deprecated and will be removed in v5 of Transformers, you should load your"
            " model with `device_map='balanced'` in the call to `from_pretrained`. You can also provide your own"
            " `device_map` but it needs to be a dictionary module_name to device, so for instance {'h.0': 0, 'h.1': 1,"
            " ...}",
            FutureWarning,
        )
        self.device_map = (
            get_device_map(len(self.h), range(torch.cuda.device_count())) if device_map is None else device_map
        )
        assert_device_map(self.device_map, len(self.h))
        self.model_parallel = True
        self.first_device = "cpu" if "cpu" in self.device_map.keys() else "cuda:" + str(min(self.device_map.keys()))
        self.last_device = "cuda:" + str(max(self.device_map.keys()))
        self.wte = self.wte.to(self.first_device)
        self.wpe = self.wpe.to(self.first_device)
        # Load onto devices
        for k, v in self.device_map.items():
            for block in v:
                cuda_device = "cuda:" + str(k)
                self.h[block] = self.h[block].to(cuda_device)
        # ln_f to last
        self.ln_f = self.ln_f.to(self.last_device)
```

1. **设备映射 (`device_map`)**:
   - 自动生成设备映射，映射模型的不同模块到可用的计算设备。

```
self.device_map = (
    get_device_map(len(self.h), range(torch.cuda.device_count())) if device_map is None else device_map
)
```

通过动态映射，确保计算资源得到最优分配。

2. **验证设备映射**:
   - 确保生成的设备映射有效，避免潜在错误。

```
1 assert_device_map(self.device_map, len(self.h))
```

3. **模型并行标志**:
   - 设置 `self.model_parallel` 为 `True`，表示模型支持并行计算。

```
1 self.model_parallel = True
```

4. **智能设备选择**:
   - 根据设备映射确定第一个和最后一个设备，优先使用CPU。

```
1 self.first_device = "cpu" if "cpu" in self.device_map.keys() else "cuda:" +
  str(min(self.device_map.keys()))
2 self.last_device = "cuda:" + str(max(self.device_map.keys()))
```

在必要时利用CPU，最大限度地利用所有计算资源。

5. **模块加载到设备**:
   - 遍历设备映射，将模型模块分配到相应的设备。

```
1 for k, v in self.device_map.items():
2     for block in v:
3         cuda_device = "cuda:" + str(k)
4         self.h[block] = self.h[block].to(cuda_device)
```

确保每个模块在正确的设备上，避免计算瓶颈。

6. **确保输出计算的有效性**:
   - 最后，将最终层分配到最后一个设备。

```
1 self.ln_f = self.ln_f.to(self.last_device)
```

确保输出计算在最佳设备上进行，提升整体性能。

7. **补充说明：**

* self.h 表征的是多层的叠加，直接将每一层编码/解码器封装到 ModuleList 列表中。

```
self.h = nn.ModuleList([GPT2Block(config, layer_idx=i) for i in
range(config.num_hidden_layers)])
```

`@add_start_docstrings` 是一个装饰器，它会将入参添加到函数的文档中。

```
@add_start_docstrings()
```

该模块通过智能的设备分配和资源管理，实现了GPT-2模型的高效并行化，

展现了在深度学习中对计算资源的合理利用和优化的重要性

虽然该方法在后续的版本已经弃用

但管中窥豹，能摸到并行计算的一个初步门槛

2.3 去并行化方法 `deparallelize`

与并行化相反，这个方法的核心功能是将 GPT-2 模型的所有参数和计算模块从 GPU（或其他加速硬件）迁移回 CPU，通常用于在不需要并行计算或者想在单设备上继续调试或训练时使用。

```
    @add_start_docstrings(DEPARALLELIZE_DOCSTRING)
    def deparallelize(self):
        warnings.warn(
            "Like `parallelize`, `deparallelize` is deprecated and will be
removed in v5 of Transformers.",
            FutureWarning,
        )
        self.model_parallel = False
        self.device_map = None
        self.first_device = "cpu"
        self.last_device = "cpu"
        self.wte = self.wte.to("cpu")
        self.wpe = self.wpe.to("cpu")
        for index in range(len(self.h)):
            self.h[index] = self.h[index].to("cpu")
        self.ln_f = self.ln_f.to("cpu")
        torch.cuda.empty_cache()
```

`torch.cuda.empty_cache()`：清理 GPU 上未使用的缓存，以释放显存资源。这在将模型从 GPU 移回 CPU 时非常有用，确保显存资源不被浪费。

并行化的方式在未来的版本中会被更新或替代，因此该功能已被弃用。

2.4 输入嵌入方法 `get_input_embeddings` 和 `set_input_embeddings`

提供获取和设置输入嵌入的方法。

很简洁但功能非常重要的方法，它们允许用户获取和设置 GPT-2 模型的输入嵌入层（即词嵌入层，`wte`）

```
    def get_input_embeddings(self):
        return self.wte
```

- **模型微调**：当你需要对 GPT-2 进行微调时，可以通过获取嵌入层的权重来了解当前模型的词嵌入状态。
  如果需要冻结或调整嵌入层，`get_input_embeddings` 会是很好的起点。
- **自定义操作**：在嵌入层上进行一些裁剪或插入新的词汇，可以通过 `get_input_embeddings` 获取嵌入层然后进行处理。

```
    def set_input_embeddings(self, new_embeddings):
        self.wte = new_embeddings
```

- **使用领域特定的预训练嵌入**：有特定领域（如医学或法律）的词嵌入，用这些嵌入层替换 GPT-2 原有的嵌入层，以提升该领域任务的效果。
- **更改词汇表**：在原始词汇表上添加了新的词汇或符号时，通常需要调整词嵌入层的尺寸，此时你可以通过 `set_input_embeddings` 方法来更新模型。

2.5 剪枝方法 `_prune_heads`

剪枝是一种模型优化方法，用于减少计算量和内存占用。

针对 GPT-2 模型中的多头注意力机制，剪去不必要的注意力头，可以提高推理和训练效率，同时保持模型性能。

```
def _prune_heads(self, heads_to_prune):
    """
    Prunes heads of the model. heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
    """
    for layer, heads in heads_to_prune.items():
        self.h[layer].attn.prune_heads(heads)
```

- **`heads_to_prune`：是一个字典，键是层的索引，值是该层要剪枝的注意力头列表。**

- `self.h[layer].attn.prune_heads(heads)`：
  - `self.h[layer]` 对应 GPT-2 模型中的每一层 `GPT2Block`
  - `attn` 是每个 `GPT2Block` 有的模块（即 `GPT2Attention`）
  - `prune_heads()` 方法实现剪枝操作

这里就涉及到，怎么剪枝？剪哪些枝？即去掉哪些注意力头？实现在GPT2Attention中的prune_heads方法

```
 def prune_heads(self, heads):
        if len(heads) == 0:
            return
        heads, index = find_pruneable_heads_and_indices(heads, self.num_heads, self.head_dim, self.pruned_heads)
        index_attn = torch.cat([index, index + self.split_size, index + (2 * self.split_size)])

        # Prune conv1d layers
        self.c_attn = prune_conv1d_layer(self.c_attn, index_attn, dim=1)
        self.c_proj = prune_conv1d_layer(self.c_proj, index, dim=0)

        # Update hyper params
        self.split_size = (self.split_size // self.num_heads) * (self.num_heads - len(heads))
        self.num_heads = self.num_heads - len(heads)
        self.pruned_heads = self.pruned_heads.union(heads)
```

核心逻辑

剪枝本质上是移除与特定注意力头相关的张量部分。在 GPT-2 中，注意力机制采用 `Q`、`K`、`V` 三个矩阵，这些矩阵通过卷积操作生成。因此，剪枝操作需要从这些**卷积层**中剪去对应的部分。

具体步骤：

1. **定位要剪去的注意力头**：根据传入的 `heads` 列表，找到这些头对应的矩阵索引。
2. **移除卷积层中的对应部分**：利用 `prune_conv1d_layer` 方法，从 `c_attn` 和 `c_proj` 卷积层中剪去与这些头相关的部分。
3. **调整超参数**：更新 `num_heads` 和 `split_size`，确保模型结构与剩余的注意力头数匹配。

这里有个非常重要的的点，也是我乃至之前学习 Transformer 有误解的点：

1. 在 GPT2Attention 中，有 `self.c_attn` 和 `self.c_proj` 这两个成员

- `self.c_attn` 可以理解为在注意力分数匹配时的隐藏层 Q，K，V

与之前手搓不同的是，这里用了 1D 卷积层实现，而非全连接层：

```
if self.is_cross_attention:
    self.c_attn = Conv1D(2 * self.embed_dim, self.embed_dim)
    self.q_attn = Conv1D(self.embed_dim, self.embed_dim)
else:
    self.c_attn = Conv1D(3 * self.embed_dim, self.embed_dim)
self.c_proj = Conv1D(self.embed_dim, self.embed_dim)
```

1. **效率上的优化**

GPT-2 中使用 1D 卷积层的一个重要原因是提升计算效率。相比传统的全连接层，1D 卷积层在处理长序列数据时更为高效，尤其在批量处理和并行计算的场景中具有显著优势。

- **并行计算**：1D 卷积层能够更好地利用现代硬件（如 GPU）的并行计算特性，在处理大规模序列时表现尤为突出。这是因为卷积操作可以在同一时间内处理多个输入位置的数据，实现计算加速。
- **矩阵分块与内存优化**：卷积操作的底层实现本质上是矩阵乘法。在 1D 卷积中，输入序列被视为一块整体，允许 GPU 等硬件在处理时进行优化，如矩阵分块计算、减少内存访问开销等。这种方式使得 GPT-2 在处理长序列时表现优异。

---

2. **共享权重的灵活性**

1D 卷积层的一个显著特点是 **权重共享**。在传统全连接层中，每个输入和输出的连接都有独立的权重，而 1D 卷积允许在输入序列的不同位置共享相同的权重。

- **权重共享机制**：尽管 GPT-2 的卷积核大小为 1，但这种设计仍然意味着模型在序列的不同位置应用了相同的线性变换。权重共享不仅减少了参数量，还增强了模型对输入序列的平移不变性（即在序列不同位置识别相同模式的能力）。
- **全局表达与局部特征捕捉**：这种权重共享方式让 GPT-2 在处理不同长度的输入时更加灵活，同时保持全局的表达能力。在自然语言处理任务中，这种机制可以帮助模型有效处理变长输入的序列数据。

---

3. **方便的扩展**

使用 1D 卷积层还使 GPT-2 的设计更加简洁灵活，特别是在 **多头自注意力机制** 中投影矩阵（Q、K、V）的处理上。

- **一次卷积操作处理多任务**：GPT-2 中的 1D 卷积不仅可以同时处理多个投影操作（如 Query、Key 和 Value 的计算），而且可以通过一次卷积操作生成所有所需的投影矩阵。这种设计比单独定义多个全连接层更加简洁高效，减少了实现的复杂性。

- **参数调整的灵活性**：由于卷积层本质上可以适应不同长度的输入序列，GPT-2 中的卷积操作能够根据任务需求灵活调整参数，适应不同的输入条件。

---

4. **减少参数量**

1D 卷积层的另一个重要优点是能够 **减少模型的参数量**，特别是在处理大规模序列输入时，卷积层的权重共享机制减少了冗余计算和参数存储。

- **权重共享减少参数**：传统全连接层的每一个连接都有独立的权重，而 1D 卷积层通过共享权重机制有效地减少了参数数量，尤其在需要大规模序列处理时（如长文本生成）具有显著的优势。
- **冗余计算的减少**：在多头注意力机制中，1D 卷积减少了多个投影矩阵间的冗余计算。这不仅提升了模型的计算效率，还在一定程度上降低了模型的存储和计算复杂度。

这是我在手搓时没考虑到的，极具智慧的处理，但效果不好说

当然在 `forward()` 的时候需要进行 `split`，这就是后话了

```
query = self.q_attn(hidden_states)
key, value = self.c_attn(encoder_hidden_states).split(self.split_size, dim=2)
```

- `self.c_proj` 实则是注意力匹配后的输出，对该层进行剪枝是很有必要的
往往最后一层线性层会有冗余的权重，不论以什么策略剪枝，都有相应的优化

拓展一下剪枝策略：

> **剪枝策略概述**
>
> 剪枝（Pruning）是一种模型压缩技术，主要通过移除不重要或冗余的部分模型参数来减少计算和存储开销。在深度学习模型中，常见的剪枝策略有以下几种：
>
> 1. **不重要权重剪枝（Unstructured Pruning）**：
>   - 基于权重的重要性来选择要剪去的参数，不考虑参数在张量中的位置。通常会将权重较小的参数视为不重要，从而剪掉它们。
>   - 这种方式灵活，但在硬件加速器上难以直接优化，因为剪掉的权重可能会导致不规则的存储和计算。
>   - **Reference:** *Learning both Weights and Connections for Efficient Neural Networks*（Han et al., 2015）

2. **结构化剪枝（Structured Pruning）**：
    - 针对特定结构进行剪枝，比如剪掉某些神经元、通道或整个注意力头。在多头注意力机制中，可以剪去一些注意力头（head），减少计算量。
    - **结构化剪枝能够在硬件加速器上高效执行，因为其剪枝后的张量仍保持规整的结构。**
    - **Reference:** *Structured Pruning of a BERT-based Question Answering Model*（Michel et al., 2019）
3. **层级剪枝（Layer-wise Pruning）**：
    - 在模型的不同层级上选择性地进行剪枝。可以是逐层剪枝（对每一层选择部分参数或注意力头），也可以是某些层整体剪枝或跳过某些层的剪枝操作。
    - **Reference:** *Layer-wise Adaptive Rate Scaling for Structured Pruning of Transformers*（Lagunas et al., 2021）
4. **基于重要性的剪枝（Importance-based Pruning）**：
    - 通过对注意力头或神经元的重要性评估（例如基于梯度、注意力得分等）来决定哪些部分可以被剪掉。常用的策略包括梯度敏感性分析、L1/L2正则化、熵等度量方法。
    - **Reference:** *Are Sixteen Heads Really Better than One?*（Michel et al., 2019）

在 GPT-2 中的剪枝策略

GPT-2 的剪枝主要应用在 **多头注意力机制**（Multi-head Attention）上，采用的是结构化剪枝（Structured Pruning）。该方法通过选择性地移除一些注意力头，减少计算量而尽量不损害模型的性能。

在 GPT-2 的剪枝实现中，通过 `prune_heads` 方法直接对注意力头进行操作，移除那些被标记为不重要的头，并相应更新权重矩阵的维度。

2.6 前向传播方法 `forward`

- `input_ids`：输入 token 的索引序列，通常是从词汇表（vocab）中得到的。
- `past_key_values`：前面计算的 key 和 value 缓存（用于加速推理）。
- `attention_mask`：注意力掩码，用于避免模型关注某些无效的位置（如填充 token）。
- `token_type_ids`：用于区分输入中的不同句子（在双句子任务中使用）。
- `position_ids`：用于指定 token 在序列中的位置信息。
- `head_mask`：用于屏蔽特定的注意力头。
- `inputs_embeds`：直接传入嵌入后的输入，而不是 token ID。
- `encoder_hidden_states`：在跨注意力机制（如 Transformer 编解码器架构）中，来自编码器的隐藏状态。

- `encoder_attention_mask`：编码器中的注意力掩码，用于跨注意力机制。
- `use_cache`：指示是否缓存 key 和 value，以加速后续的推理步骤。
- `output_attentions`：指示是否输出注意力权重。
- `output_hidden_states`：指示是否输出每一层的隐藏状态。
- `return_dict`：指定是否以字典形式返回结果

标记的是 Transformer 核心参数，即在手搓代码中的入参
划线的是很有意思的几个参数，扒后面代码的时候注意一下

`output_attentions`

- **作用**：控制是否返回每一层的注意力权重（attention weights）。
- **用处**：
  - **调试与解释**：查看模型在每个输入 token 上的关注点，了解模型关注的输入部分，有助于调试和解释模型的行为。
  - **可视化**：生成注意力热图，分析模型在 NLP 任务中的注意力分布，特别在机器翻译和文本生成中很有用。

`output_hidden_states`

- **作用**：控制是否返回每一层的隐藏状态（hidden states）。
- **用处**：
  - **特征提取**：获取中间层的隐藏状态用于特征提取，例如用于下游任务的迁移学习。
  - **模型分析**：分析每层隐藏状态的演变，了解模型在不同层次如何处理输入。

`use_cache`

- **作用**：控制是否缓存 `past_key_values`（包含每层的 key 和 value 值）。
- **用处**：
  - **加速推理**：特别在自回归生成任务中（如文本生成），缓存机制可以避免重复计算先前的 key 和 value，从而加快推理速度。
  - **实时生成**：逐步生成文本时，缓存前面步骤的计算结果，减少重复计算，显著提升效率。

`return_dict`

- **作用**：控制返回格式，选择返回字典形式还是元组形式的输出。
- **用处**：
  - **灵活性**：使用字典格式时，可以通过键直接访问模型输出（如 `last_hidden_state`、`attentions` 等），更加直观和便捷。
  - **兼容性**：某些代码或框架可能需要特定的输出格式（如元组格式），可根据需求选择合适的返回形式。

这些参数使得 GPT-2 增加了灵活性，可解释性，以及可视性，能够更好地适应不同的任务需求

`input_ids` & `inputs_embeds`

1. `input_ids`：

`input_ids` 是文本通过 `Tokenizer` 转换为的整数序列，每个整数代表词汇表中的一个词汇 ID。它在进入模型时需要通过 `nn.Embedding` 层将这些整数 ID 映射为词嵌入向量。

优点：

- **简便性**：使用 `input_ids` 时，模型会自动处理词嵌入生成的过程，用户不需要额外准备词嵌入向量，减少了手动预处理步骤。
- **嵌入可训练**：模型的嵌入层在训练过程中会自动更新，可以随着任务的训练动态调整词嵌入，使其适应特定任务。
- **内嵌优化**：使用 `input_ids` 能让词嵌入和模型的其他部分一起被训练和优化，适用于大多数任务场景。

缺点：

- **灵活性受限**：用户无法指定外部词嵌入（如 GloVe、FastText 等），只能使用模型内部的嵌入层。如果想要使用特定的预训练词嵌入，需要对模型做一些额外的调整。
- **依赖内置的嵌入层**：无法直接控制词嵌入的具体数值和初始化方式，受限于模型的内置机制。

---

2. `inputs_embeds`：

`inputs_embeds` 是直接传入的词嵌入向量，这些向量可以来自外部的预训练嵌入或自定义的嵌入表示，因此可以跳过模型的嵌入层。

优点：

- **灵活性高**：允许使用外部预训练的词嵌入或自己计算的词嵌入，适用于某些特殊场景，如需要使用 GloVe、Word2Vec 等预训练的词嵌入，或者自定义嵌入方式。
- **自定义控制**：用户可以完全控制嵌入的初始化及其数值，这在需要使用特定的嵌入技术（如领域特定嵌入）时非常有用。

- **跨模型迁移**：能够使用预先在其他模型或任务中学到的词嵌入，直接在当前任务中进行迁移和微调。

缺点：

- **复杂性增加**：需要提前计算好词嵌入，在数据处理和准备阶段增加了复杂度。
- **词嵌入固定**：通常情况下，`inputs_embeds` 一旦传入后，词嵌入是不可训练的。这意味着这些嵌入向量在训练过程中不会动态更新，除非在自定义逻辑中手动允许更新。
- **维护成本高**：自定义词嵌入需要用户手动管理词嵌入的质量和适用性，尤其是在涉及大量外部数据或复杂任务时，管理这些嵌入可能会较为复杂。

特性	input_ids	inputs_embeds
嵌入生成方式	通过 nn.Embedding 层自动生成词嵌入	直接传入预先计算好的词嵌入
灵活性	灵活性有限，依赖模型内部嵌入层	高度灵活，允许使用外部或自定义词嵌入
嵌入更新	嵌入层可在训练时更新	通常固定，不会在训练中更新
简便性	更加简便，用户无需额外处理词嵌入	需要预先准备词嵌入，增加数据处理复杂性
适用场景	适合常规任务，内部嵌入自动优化	适合有外部词嵌入或特殊需求的任务

`token_type_ids`

```
if token_type_ids is not None:
    token_type_ids = token_type_ids.view(-1, input_shape[-1])
```

- **含义**：`token_type_ids` 是用于处理双句子输入（如句子对任务）的标识。它是一种辅助输入，用于区分不同句子中的 token（词汇）属于哪个句子，尤其是在句子对任务中（如自然语言推理、问答系统等）。
  - 例如，在句子对任务中，输入通常是由两个句子拼接而成的，在这种情况下，`token_type_ids` 可以指示句子 A 的 token 被标记为 0，句子 B 的 token 被标记为 1，从而帮助模型区分两个句子。
  - 在 `forward` 函数中，通过 `view` 函数调整 `token_type_ids` 的形状，以确保它与 `input_ids` 或 `inputs_embeds` 的形状一致，方便后续的计算。

`past_key_values`

```
if past_key_values is None:
    past_length = 0
    past_key_values = tuple([None] * len(self.h))
else:
    past_length = past_key_values[0][0].size(-2)
```

- **含义：** `past_key_values` 存储的是上一轮计算时生成的 key 和 value 的缓存。这在自回归生成任务（如文本生成）中非常重要，因为每一时刻的输出依赖于前一时刻的输出。通过缓存 `past_key_values`，模型可以跳过重复计算先前的部分，从而加速推理过程。
  - 如果 **`past_key_values`** 为空，则初始化为包含 `None` 的元组，长度与模型的层数相同。这表明模型没有历史状态。
  - 如果 **`past_key_values`** 已经存在，则获取其中第一个 key 的长度（`past_length`），它代表了模型在前一轮计算中已经处理过的序列长度。这个长度会影响生成 `position_ids` 的过程，确保位置信息能够正确连接上之前的状态。

> 在文本生成任务中（如 GPT-2 生成文本时），`past_key_values` 可以极大加速生成过程，因为它避免了每次重新计算所有的注意力，而是只计算新增的部分。

这是很好的优化策略，在推理过程中，生成每一个 token 需要经过 N 层的计算；而由于每层的词汇是固定的，即已经生成的词汇，所以其 Key 和 Value 可以是固定的

只有当新的 token 生成后，需要重新计算 Key 和 Value

`position_ids`

```
if position_ids is None:
    position_ids = torch.arange(past_length, input_shape[-1] + past_length, dtype=torch.long, device=device)
    position_ids = position_ids.unsqueeze(0)
```

- **含义：** `position_ids` 是指示每个 token 在序列中的位置。Transformer 模型本身没有位置信息的感知能力，因此需要通过显式的位置编码来告诉模型每个 token 在输入序列中的具体位置。这就是 `position_ids` 的作用。
  - 如果没有提供 `position_ids`，则根据输入序列的长度自动生成一组位置信息，从 `past_length` 开始，确保模型能够感知 token 在整个序列中的位置。
  - `torch.arange` 生成从 `past_length` 到 `input_shape[-1] + past_length` 的整数序列，代表每个 token 的位置索引。`unsqueeze(0)` 将它扩展为 batch 维度。

> 例如，在生成式任务中（如 GPT-2 生成文本时），`position_ids` 可以确保模型为每个新生成的 token 分配正确的位置编码，即使这个 token 是在先前生成的序列后追加的。

在处理掩码的过程中，提到了一个关键的优化策略：

`Flash_Attention_2`

Flash_Attention_2 是一种用于 Transformer 模型中的自注意力机制的优化算法，

旨在最大化利用 GPU 硬件（尤其是张量核心）的并行计算能力，降低内存占用，并提高计算效率。

特别适合长序列任务，因为传统的自注意力机制在长序列情况下会导致显存占用过大和计算速度瓶颈。

核心概念

1. **流式计算（Streaming Computation）**：
   - **传统自注意力**会在计算 `QK^T` 点积时生成完整的注意力矩阵，并将其存储在显存中，接下来再通过 softmax 加权计算出结果。这会导致显存大量占用，尤其是在处理长序列时，因为每个序列的长度成平方关系地影响存储需求。
   - **Flash_Attention_2**则通过流式计算，将 `QK^T` 的计算与 softmax 计算一步步流式完成，不需要将整个注意力矩阵存储下来。每次只存储当前需要的部分，计算出中间结果后立即进行下一个操作。这种方式有效减少了内存占用，同时加快了计算速度。

> 假设我们有以下设置：
> - 序列长度（`seq_length`）：1024
> - 每个注意力头的维度（`head_dim`）：64
> - 注意力头数（`num_heads`）：8
> - 批量大小（`batch_size`）：2
>
> 最终的 `Q`（Query），`K`（Key），`V`（Value）矩阵的形状为：
> - `Q`、`K`、`V`：`(batch_size, num_heads, seq_length, head_dim)`，即 `(2, 8, 1024, 64)`
>
> `flash_attention_2` 采用流式处理（分块计算），我们假设将序列长度 `1024` 分成 8 个块，每个块的长度为 `128`。
>
> **1. 第一块的计算（`128` tokens）**

- 加载第一块的 `Q` 和 `K` 矩阵，它们的维度分别是：
  - `Q_block1`：`(batch_size, num_heads, 128, head_dim)`，即 `(2, 8, 128, 64)`
  - `K_block1`：`(batch_size, num_heads, 128, head_dim)`，即 `(2, 8, 128, 64)`
- 计算 `Q_block1` 和 `K_block1` 的点积（注意力得分）：

```
attention_scores_block1 = torch.matmul(Q_block1, K_block1.transpose(-1, -2))
```

  结果的维度为：
  - `attention_scores_block1`：`(batch_size, num_heads, 128, 128)`，即 `(2, 8, 128, 128)`
- 对 `attention_scores_block1` 进行归一化（如 softmax），并与第一块的 `V` 矩阵进行点积：

```
attention_probs_block1 = softmax(attention_scores_block1)
output_block1 = torch.matmul(attention_probs_block1, V_block1)
```

  其中 `V_block1` 的维度为：
  - `V_block1`：`(batch_size, num_heads, 128, head_dim)`，即 `(2, 8, 128, 64)`

经过矩阵乘法后，得到的输出 `output_block1` 的维度为：

- `output_block1`：`(batch_size, num_heads, 128, head_dim)`，即 `(2, 8, 128, 64)`

丢弃的中间数据：

在这个过程中，注意力得分 `attention_scores_block1` 在计算完后就不再需要，可以直接丢弃，不会保存在全局内存中。这有效减少了存储空间需求。

2. 第二块的计算（接下来 `128` tokens）

- 加载第二块的 `Q` 和 `K` 矩阵，形状相同：
  - `Q_block2`：`(batch_size, num_heads, 128, head_dim)`，即 `(2, 8, 128, 64)`

- `K_block2`：`(batch_size, num_heads, 128, head_dim)`，即 `(2, 8, 128, 64)`

- 计算 `Q_block2` 和前两块的 `K` 矩阵（`K_block1` 和 `K_block2`）的点积。为了保持因果性，必须考虑前面的所有块，因此计算是：

```
attention_scores_block2 = torch.matmul(Q_block2, torch.cat([K_block1, K_block2], dim=-2).transpose(-1, -2))
```

这时 `K_block1 + K_block2` 的维度为：

- `K_total`：`(batch_size, num_heads, 256, head_dim)`，即 `(2, 8, 256, 64)`

这会产生新的 `attention_scores_block2`：

- `attention_scores_block2`：`(batch_size, num_heads, 128, 256)`，即 `(2, 8, 128, 256)`
- 经过归一化后，`attention_probs_block2` 也会与前两块的 `V` 矩阵（`V_block1` 和 `V_block2`）进行点积：

```
output_block2 = torch.matmul(attention_probs_block2, torch.cat([V_block1, V_block2], dim=-2))
```

输出 `output_block2` 的维度仍为：

- `output_block2`：`(batch_size, num_heads, 128, head_dim)`，即 `(2, 8, 128, 64)`

中间数据的丢弃：

在计算完 `attention_scores_block2` 后，`attention_scores_block2` 和前两块的得分矩阵都可以丢弃。而且，`K_block1` 和 `V_block1` 只需要在第二块计算时保留，之后也可以释放内存。

3. **后续块的计算（逐块处理）**

接下来，`flash_attention_2` 按照类似的方式处理每一块。每块的 `Q` 都需要和前面所有块的 `K` 矩阵进行点积，注意力分数计算完成后丢弃，最终输出的结果与所有之前的 `V` 矩阵进行点积，计算出每个块的输出。

4. **最终输出：**

所有块的 `output_block` 最终会被拼接起来，构成完整的输出序列：

```
output = torch.cat([output_block1, output_block2, ..., output_block8], dim=2)
```

最终输出 `output` 的形状为：

- `output`：`(batch_size, num_heads, seq_length, head_dim)`，即 `(2, 8, 1024, 64)`

关键点总结：

1. **块状计算**：每次只计算一部分序列的 `Q`、`K`、`V` 矩阵，避免全局存储巨大的注意力矩阵。
2. **中间结果的丢弃**：每一块计算完后，立即丢弃不再需要的注意力得分矩阵，减少内存占用。
3. **分块拼接输出**：每块的输出会最终拼接成完整的输出序列，保持模型的正常运行。

通过这种方式，`flash_attention_2` 减少了不必要的内存使用，并最大化利用寄存器和片上缓存来进行高效的并行计算。这种分块计算的思想特别适合长序列的任务，能够显著提高计算效率并降低硬件资源消耗。

2. **张量核心的利用（Tensor Core Optimization）**：
   - 在 GPU 中，张量核心专门用于处理矩阵运算，而 Flash_Attention_2 通过深度优化这些矩阵运算，将注意力机制中的大规模矩阵点积和 softmax 操作转化为最适合张量核心的运算形式。通过批量矩阵乘法（`matmul`），Flash_Attention_2 可以在一个操作中完成多头注意力机制的计算，而不是像传统的方式那样逐个头独立计算。
   - 这意味着 GPU 能够更高效地并行计算，充分利用其算力。

传统多头注意力 vs FlashAttention 2

假设我们有 4 个注意力头，每个头对应的查询矩阵 `Q` 和键矩阵 `K` 维度是 `(batch_size, seq_length, head_dim)`，我们将分别看看传统计算方式和 FlashAttention 2 的区别。

1. 传统多头注意力计算

在传统多头注意力机制中，我们通常会为每个头独立计算注意力权重：

```
import torch

# 每个头的查询和键矩阵
Q_head1 = torch.randn(batch_size, seq_length, head_dim)
K_head1 = torch.randn(batch_size, seq_length, head_dim)

# 逐头计算 Q 和 K 的点积，得到注意力权重
attention_scores_head1 = torch.matmul(Q_head1, K_head1.transpose(-1, -2))
```

```
9
10 # 其他头的点积计算
11 # attention_scores_head2 = torch.matmul(Q_head2, K_head2.transpose(-1, -2))
12 # ...
```

这意味着我们需要为每个头独立调用 `matmul`，计算成本较高。

2. FlashAttention 2 的优化计算

使用 FlashAttention 2，通过批量矩阵乘法和并行计算，我们可以同时处理多个注意力头的计算：

```
1 # 假设 Q 和 K 是合并后的多头查询和键矩阵，维度是 (batch_size, num_heads,
  seq_length, head_dim)
2 Q = torch.randn(batch_size, num_heads, seq_length, head_dim)
3 K = torch.randn(batch_size, num_heads, seq_length, head_dim)
4
5 # 同时计算多个头的 Q 和 K 点积
6 attention_scores = torch.matmul(Q, K.transpose(-1, -2))
```

在这个例子中，`Q` 和 `K` 的维度是 `(batch_size, num_heads, seq_length, head_dim)`，通过批量矩阵乘法（`matmul`）一次性计算多个头的点积，大大减少了时间复杂度。同时，FlashAttention 2 在内部优化了中间结果的存储与应用，减少了内存占用，使得计算更加高效。

3. **混合精度计算（Mixed Precision Computation）**：
   - Flash_Attention_2 还利用了混合精度计算（FP16 和 FP32 结合使用），即在不影响最终精度的情况下，部分计算可以使用半精度（FP16），从而减少显存占用并提高计算速度。通常，`QK^T` 点积和 softmax 的中间计算步骤可以使用 FP16，而关键计算如反向传播则使用 FP32 以保证数值稳定性。

`attention_mask`

在 GPT-2 模型的实现中，`attention_mask` 用于指定哪些 token 需要被模型关注，哪些 token 需要被忽略。

在不同的注意力机制（如 `flash_attention_2` 和 `sdpa`）下，对 `attention_mask` 的处理方式不同。

```python
# 调整 attention_mask 的形状
attention_mask = attention_mask.view(batch_size, -1) if attention_mask is not None else None

# 处理 flash_attention_2
if self._attn_implementation == "flash_attention_2":
    attention_mask = attention_mask if (attention_mask is not None and 0 in attention_mask) else None

# 处理 sdpa 机制
elif _use_sdpa:
    attention_mask = _prepare_4d_causal_attention_mask_for_sdpa(
        attention_mask=attention_mask,
        input_shape=(batch_size, input_shape[-1]),
        inputs_embeds=inputs_embeds,
        past_key_values_length=past_length,
    )

# 处理默认自注意力机制
else:
    if attention_mask is not None:
        attention_mask = attention_mask[:, None, None, :]
        attention_mask = attention_mask.to(dtype=self.dtype)
        attention_mask = (1.0 - attention_mask) * torch.finfo(self.dtype).min
```

1. 调整 `attention_mask` **的形状**

```python
attention_mask = attention_mask.view(batch_size, -1) if attention_mask is not None else None
```

- 如果 `attention_mask` 不是 `None`，将其调整为 `[batch_size, sequence_length]` 形状。
- 如果没有提供 `attention_mask`，则设为 `None`，表示没有需要特别处理的掩码。

2. 处理 `flash_attention_2` **注意力机制**

```python
if self._attn_implementation == "flash_attention_2":
    attention_mask = attention_mask if (attention_mask is not None and 0 in attention_mask) else None
```

- 如果当前模型使用 `flash_attention_2`，需要确保 `attention_mask` 中存在 0（表示填充位置）。否则，将 `attention_mask` 设为 `None`。
- `flash_attention_2` 的高效计算需要明确掩码的存在，若无填充 token，则不使用 `attention_mask`。

3. 处理 `sdpa` **（缩放点积注意力）机制**

```
elif _use_sdpa:
    attention_mask = _prepare_4d_causal_attention_mask_for_sdpa(
        attention_mask=attention_mask,
        input_shape=(batch_size, input_shape[-1]),
        inputs_embeds=inputs_embeds,
        past_key_values_length=past_length,
    )
```

- 如果使用 `sdpa`，则将 `attention_mask` 转换为四维的因果掩码。
- `attention_mask` 被转换为 `[batch_size, num_heads, seq_length, seq_length]` 形状，以适应因果自注意力的计算方式。在这种注意力机制下，未来的 token 不应该关注到过去的 token。

4. 标准自注意力处理（非 `flash_attention_2` 和 `sdpa`）

```
else:
    if attention_mask is not None:
        attention_mask = attention_mask[:, None, None, :]
        attention_mask = attention_mask.to(dtype=self.dtype)
        attention_mask = (1.0 - attention_mask) * torch.finfo(self.dtype).min
```

- 这是默认情况下的处理方式，适用于标准的自注意力机制。
- 将 `attention_mask` 扩展为四维 `[batch_size, 1, 1, seq_length]`，以确保它可以广播到 `[batch_size, num_heads, from_seq_length, to_seq_length]` 的形状。
- 然后将掩码转换为与模型数据类型一致的格式，以确保计算精度。
- 最后，通过 `(1.0 - attention_mask) * torch.finfo(self.dtype).min` 将无效位置设置为非常小的负数，在 softmax 计算时，这些位置的权重接近 0，从而忽略这些 token。

`cross-attentionmask`

在 GPT-2 模型中，跨注意力用于处理多模态输入或将编码器输出作为解码器的输入。

在这种情况下，除了处理自注意力掩码，还需要对跨注意力掩码进行特殊处理，确保在注意力机制中，解码器仅关注有效的编码器输出部分。

```
# 跨注意力掩码处理
if self.config.add_cross_attention and encoder_hidden_states is not None:
    encoder_batch_size, encoder_sequence_length, _ = encoder_hidden_states.size()
    encoder_hidden_shape = (encoder_batch_size, encoder_sequence_length)

    # 如果没有提供 encoder_attention_mask，则创建全为1的掩码
    if encoder_attention_mask is None:
        encoder_attention_mask = torch.ones(encoder_hidden_shape, device=device)

    # 处理 sdpa 注意力机制
    if _use_sdpa:
        encoder_attention_mask = _prepare_4d_attention_mask_for_sdpa(
            mask=encoder_attention_mask, dtype=inputs_embeds.dtype, tgt_len=input_shape[-1]
        )

    # 处理标准的跨注意力机制
    elif not self._attn_implementation == "flash_attention_2":
        encoder_attention_mask = self.invert_attention_mask(encoder_attention_mask)
else:
    encoder_attention_mask = None
```

1. 检查是否需要跨注意力

```
if self.config.add_cross_attention and encoder_hidden_states is not None:
```

- 该条件判断是否启用了跨注意力机制，即模型配置中 `add_cross_attention` 为 `True` 并且提供了 `encoder_hidden_states`。
- 如果满足条件，模型将对编码器输出进行关注，并对 `encoder_attention_mask` 进行处理。

2. 获取编码器输入的尺寸信息

```
encoder_batch_size, encoder_sequence_length, _ = encoder_hidden_states.size()
encoder_hidden_shape = (encoder_batch_size, encoder_sequence_length)
```

- 计算出 `encoder_hidden_states` 的批量大小和序列长度，用于生成或调整 `encoder_attention_mask`。

3. 默认生成全为1的 `encoder_attention_mask`

```
if encoder_attention_mask is None:
    encoder_attention_mask = torch.ones(encoder_hidden_shape, device=device)
```

- 如果没有提供 `encoder_attention_mask`，则默认生成一个全为 1 的掩码，表示模型应完全关注编码器的所有输出。
- 掩码的形状为 `[batch_size, seq_length]`。

4. 处理 `sdpa` 注意力机制

```
if _use_sdpa:
    encoder_attention_mask = _prepare_4d_attention_mask_for_sdpa(
        mask=encoder_attention_mask, dtype=inputs_embeds.dtype,
tgt_len=input_shape[-1]
    )
```

- 如果启用了 `sdpa`（缩放点积注意力），需要将 `encoder_attention_mask` 转换为四维的掩码，适配 `sdpa` 机制的计算方式。
- 该掩码的形状为 `[batch_size, num_heads, seq_length, seq_length]`，以适应多头注意力机制的需求。

5. 处理非 `flash_attention_2` 的注意力机制

```
elif not self._attn_implementation == "flash_attention_2":
    encoder_attention_mask = self.invert_attention_mask(encoder_attention_mask)
```

- 如果使用的是标准的跨注意力机制或非 `flash_attention_2` 的实现方式，调用 `invert_attention_mask` 方法将掩码进行反转处理。

- 反转处理的结果是将有效位置（1）变为 0，忽略位置（0）变为负无穷，这样在 softmax 计算时可以忽略无效位置。

6. 如果不需要跨注意力，则设为 `None`

```
else:
    encoder_attention_mask = None
```

- 如果没有启用跨注意力或者没有提供 `encoder_hidden_states`，那么 `encoder_attention_mask` 设为 `None`。

---

3. GPT2Block

`GPT2Block` 是 GPT2 模型中的一个基本组成单元。

每个 `GPT2Block` 包含自注意力机制和前馈神经网络，并通过残差连接和层归一化确保信息流畅。

- 它的作用是通过自注意力机制捕捉 token 之间的依赖关系，并通过前馈神经网络进行特征转换。
- 这种模块化设计使得 GPT2 模型能够处理更长的序列，并逐层捕捉上下文关系。

```
class GPT2Block(nn.Module):
    def __init__(self, config, layer_idx=None):

        # 默认为 n_embd: 768
        hidden_size = config.hidden_size

        # 层归一化
        self.ln_1 = nn.LayerNorm(hidden_size, eps=config.layer_norm_epsilon)

        # 自注意力
        self.attn = GPT2Attention(config, layer_idx=layer_idx)

        # 层归一化
        self.ln_2 = nn.LayerNorm(hidden_size, eps=config.layer_norm_epsilon)

        # 前馈网络，包含两层线性变换和一个 ReLU 激活函数
        self.mlp = nn.Sequential(
            nn.Linear(hidden_size, config.intermediate_size),
```

```
            nn.ReLU(),
            nn.Linear(config.intermediate_size, hidden_size),
        )

    def forward(self, hidden_states, layer_past=None, attention_mask=None, head_mask=None,
                use_cache=None, output_attentions=None):

        # 残差连接
        residual = hidden_states

        # 第一个层归一化
        hidden_states = self.ln_

1(hidden_states)

        # 计算自注意力
        attn_outputs = self.attn(
            hidden_states,
            layer_past=layer_past,
            attention_mask=attention_mask,
            head_mask=head_mask,
            use_cache=use_cache,
            output_attentions=output_attentions,
        )

        # 提取注意力输出
        attn_output = attn_outputs[0]

        # 添加残差
        hidden_states = attn_output + residual

        # 第二次残差连接
        residual = hidden_states
        hidden_states = self.ln_2(hidden_states)

        # 前馈网络
        feed_forward_hidden_states = self.mlp(hidden_states)

        # 添加残差
        hidden_states = residual + feed_forward_hidden_states

        return (hidden_states,) + attn_outputs[1:]
```

4. GPT2Attention

`GPT2Attention` 实现了自注意力机制，用于计算每个 token 与序列中其他 token 之间的相关性。通过 Query、Key 和 Value 矩阵的计算，注意力机制能够在序列中捕捉远距离 token 之间的依赖关系。

- 它的主要功能是计算注意力权重并生成注意力输出，用于更新 token 的表示。
- `GPT2Attention` 使用多头注意力机制，使模型能够并行处理不同的子空间，增强表示能力。

```
class GPT2Attention(nn.Module):
    def __init__(self, config, is_cross_attention=False, layer_idx=None):
        super().__init__()

        # 模型曾使用过的最大的 sequence 长度，默认 1024
        max_positions = config.max_position_embeddings

        # 解码时，每个位置的 token 只能跟自己以及之前位置的 token 计算注意力
        self.register_buffer(
            "bias",
            torch.tril(torch.ones((max_positions, max_positions),
dtype=torch.bool)).view(
                1, 1, max_positions, max_positions
            ),
        )
        self.register_buffer("masked_bias", torch.tensor(-1e4))

        # 单个 token 的编码纬度，默认 768
        self.embed_dim = config.hidden_size

        # 多头注意力，默认 12 个头
        self.num_heads = config.num_attention_heads

        # 每个头分到的纬度
        self.head_dim = self.embed_dim // self.num_heads
        self.split_size = self.embed_dim

        # 是否需要将注意力除以 sqrt(n_embd) ，默认 True
        self.scale_attn_weights = config.scale_attn_weights

        if self.is_cross_attention:
            self.c_attn = Conv1D(2 * self.embed_dim, self.embed_dim)
            self.q_attn = Conv1D(self.embed_dim, self.embed_dim)
        else:
```

```
34              self.c_attn = Conv1D(3 * self.embed_dim, self.embed_dim)
35
36          self.c_proj = Conv1D(self.embed_dim, self.embed_dim)
37
38          self.attn_dropout = nn.Dropout(config.attn_pdrop)
39          self.resid_dropout = nn.Dropout(config.resid_pdrop)
40
41      def _attn(self, query, key, value, attention_mask=None, head_mask=None):
42
43          # Q, K矩阵相乘，计算每个 token 与其他 token 的注意力权重
44          # [batch_size, num_heads, seq_length, head_dim] * [batch_size,
    num_heads, head_dim, seq_length] -> [batch_size, num_heads, seq_length,
    seq_length]
45          attn_weights = torch.matmul(query, key.transpose(-1, -2))
46
47          if self.scale_attn_weights:
48              # 缩放，除以 sqrt(n_embd)
49              attn_weights = attn_weights / self.head_dim ** 0.5
50
51          # 掩去 mask 位置的注意力
52          if not self.is_cross_attention:
53              query_length, key_length = query.size(-2), key.size(-2)
54              causal_mask = self.bias[:, :, key_length - query_length :
    key_length, :key_length]
55              mask_value = torch.finfo(attn_weights.dtype).min
56              attn_weights = torch.where(causal_mask, attn_weights, mask_value)
57
58          if attention_mask is not None:
59              attn_weights += attention_mask
60
61          attn_weights = nn.functional.softmax(attn_weights, dim=-1)
62
63          # 计算注意力输出
64          attn_output = torch.matmul(attn_weights, value)
65
66          return attn_output, attn_weights
67
68      def forward(self, hidden_states, layer_past=None, attention_mask=None,
    head_mask=None, use_cache=None, output_attentions=None):
69
70          query, key, value = self.c_attn(hidden_states).split(self.embed_dim,
    dim=2)
71
72          if layer_past is not None:
73              past_key, past_value = layer_past
74              key = torch.cat([past_key, key], dim=-2)
75              value = torch.cat([past_value, value], dim=-2)
```

```
76
77          present = (key, value) if use_cache else None
78
79          attn_output, attn_weights = self._attn(query, key, value,
    attention_mask, head_mask)
80
81          attn_output = self.c_proj(attn_output)
82
83          attn_output = self.resid_dropout(attn_output)
84
85          outputs = (attn_output, present)
86          if output_attentions:
87              outputs += (attn_weights,)
88
89          return outputs  # (attn_output, present, (attn_weights))
```

---

5. GPT2Config

GPT2Config 是 GPT2 模型的配置类，定义了模型的各种参数，例如隐藏层的维度、注意力头的数量、层的数量等。这个配置类提供了灵活性，可以根据需要调整模型的结构。

参数	说明
`vocab_size`	词汇表大小，默认为 50257。
`n_positions`	最大序列长度，即 `seq_len` ，默认为 1024。
`n_embd`	嵌入和隐藏状态的维度，即 `d_model` ，默认为 768。
`n_layer`	隐藏层数量，即 `N` ，默认为 12。
`n_head`	多头注意力的头数量，默认为 12。
`n_inner`	内部前馈层的维度（默认为 `4 * n_embd` ），即 4 * 768。
`activation_function`	激活函数，默认为 `gelu_new` 。
`resid_pdrop`	全连接层的 dropout 比例，默认为 0.1。
`embd_pdrop`	嵌入层的 dropout 比例，默认为 0.1。

`attn_pdrop`	注意力层的 dropout 比例，默认为 0.1。
`layer_norm_epsilon`	层归一化的 epsilon 值，默认为 1e-5。
`initializer_range`	权重初始化的标准差，默认为 0.02。
`summary_type`	序列汇总方式，默认为 `"cls_index"`。
`summary_use_proj`	是否使用投影层，默认为 `True`。
`summary_activation`	汇总激活函数，可选。
`summary_proj_to_labels`	投影输出类别设置，默认为 `True`。
`summary_first_dropout`	投影和激活后的 dropout 比例，默认为 0.1。
`scale_attn_weights`	是否缩放注意力权重，默认为 `True`。
`use_cache`	是否返回最后的 key/value 注意力，默认为 `True`。
`bos_token_id`	句子开始标记的 ID，默认为 50256。
`eos_token_id`	句子结束标记的 ID，默认为 50256。
`scale_attn_by_inverse_layer_idx`	是否按层级缩放注意力权重，默认为 `False`。
`reorder_and_upcast_attn`	是否在计算注意力前进行缩放，默认为 `False`。

> *GELU (Gaussian Error Linear Unit) 激活函数公式为：
>
> $$GELU(x) = x \cdot \Phi(x)$$
>
> 其中，$\Phi(x)$ 是标准正态分布的累积分布函数。可以理解为，输入值 x 的大部分影响会通过高斯分布的“门”来调整。与其他激活函数相比，GELU 更适合于处理神经网络中的非线性转换。

```python
class GPT2Config(PretrainedConfig):
    model_type = "gpt2"

    def __init__(
        self,
        vocab_size=50257,
```

```
        n_positions=1024,
        n_ctx=1024,
        n_embd=768,
        n_layer=12,
        n_head=12,
        n_inner=None,
        activation_function="gelu_new",
        resid_pdrop=0.1,
        embd_pdrop=0.1,
        attn_pdrop=0.1,
        layer_norm_epsilon=1e-5,
        initializer_range=0.02,
        summary_type="cls_index",
        summary_use_proj=True,
        summary_activation=None,
        summary_proj_to_labels=True,
        summary_first_dropout=0.1,
        scale_attn_weights=True,
        use_cache=True,
        bos_token_id=50256,
        eos_token_id=50256,
        scale_attn_by_inverse_layer_idx=False,
        reorder_and_upcast_attn=False,
        **kwargs,
    ):
        super().__init__(**kwargs)
        self.vocab_size = vocab_size
        self.n_positions = n_positions
        self.n_ctx = n_ctx
        self.n_embd = n_embd
        self.n_layer = n_layer
        self.n_head = n_head
        self.n_inner = n_inner
        self.activation_function = activation_function
        self.resid_pdrop = resid_pdrop
        self.embd_pdrop = embd_pdrop
        self.attn_pdrop = attn_pdrop
        self.layer_norm_epsilon = layer_norm_epsilon
        self.initializer_range = initializer_range
        self.summary_type = summary_type
        self.summary_use_proj = summary_use_proj
        self.summary_activation = summary_activation
        self.summary_proj_to_labels = summary_proj_to_labels
        self.summary_first_dropout = summary_first_dropout
        self.scale_attn_weights = scale_attn_weights
        self.use_cache = use_cache
        self.bos_token_id = bos_token_id
```

```
        self.eos_token_id = eos_token_id
        self.scale_attn_by_inverse_layer_idx = scale_attn_by_inverse_layer_idx
        self.reorder_and_upcast_attn = reorder_and_upcast_attn
```