GPT-2 模型综述与研究

院　　　系：统计与信息学院

课 程 名 称：（选）深度学习

任 课 教 师：胡光

学 生 姓 名：陶盛皿

学　　　号：19024075

专　　　业：大数据 1901

考试诚信承诺书

本人郑重承诺：在 2021-2022 学年第二学期课程期末考试中，严格遵守学校《学生考试规定》，独立完成考试（论文、报告、作业等），不违纪，不作弊，如有违反，按学校规定接受处理。

学生签名：陶[illegible]　　　　日期：2022 年 5 月 26 日

摘要

本文是对于自然语言处理领域GPT系列的深度学习模型的研究与综述，主要研究的是GPT-2，通过其原理、多任务学习、单向Transformer架构、注意力机制等方面进行研究。最后针对该模型的局限性进行评论并提出展望。附录部分为GPT-2使用预训练模型的文本生成的应用的实现代码以及对使用到关键技术的模块的从零实现。

关键词：单向Transformer模型，注意力机制，多任务学习，生成式语言模型

GPT-2 模型综述与研究

引言

GPT 系列模型是由 OpenAI 研发用于自然语言处理领域的模型，是基于单向 Transformer 模型。GPT-1 于 2018 年退出（早于 Google 提出的 Bert 模型），采用预训练的迁移学习和下游任务微调的方式处理自然语言处理任务，能够解决动态语义模型，模型架构是将词嵌入（word embedding）输入单向 Transformer 模型的监督学习。GPT-2 是该团队对 GPT 模型的升级版，是沿用了 GPT-1 的架构的多任务学习模型，刷新了 7 大数据集基准（GLUE benchmark），被认为是“最强通用 NLP 模型”。2020 年推出的具有 1750 亿巨大参数量的 GPT-3 是第三代生成式模型，即情景学习模型，用于语言预测任务，比如创造性写作（包括：诗歌、新闻、对联、文学作品、小说撰写等），目前闭源。本文是对 GPT-2 模型的综述与研究，以及使用 Pytorch 框架对模型关键模块的代码实现（见附录）。

一、原理综述

(一) 核心思想

GPT-2 核心思想是使用无监督的预训练模型去做有监督学习任务，沿用 GPT-1 的单向 Transformer 模型结构。

1. 预训练模型

预训练是只对无标注语料进行无监督学习得到语言模型，该语言模型即预训练模型，预训练模型通过微改、参数微调的方式移植到有监督的下游任务中。与训练的过程中使用最大似然函数对语料序列训练[1]。

预训练沿用单向 Transformer 结构， 即 GPT-1 中使用的 12 层 Transformer 解码器架构。结构如下所示：

$$h_0 = UW_e + W_p$$

$$h_l = Transformerblock\left(h_{l-1}\right), \forall i \in [1, n]$$

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

其中表示语料词向量，n 表示 Transformer 的层数，W_e^T表示词向量矩阵，W_p是位置嵌入（Position embedding）矩阵。

2. 语言模型

目的是完成下游有监督的语言预测任务，即基于条件概率的利用序列相关性语言预测（生成式）模型，如下：

$$P(S_n \mid S_1, S_2, S_3, \ldots.., S_{n-1})$$

本质上是通过输入序列估计文本输出：

$$\hat{P}(output \mid input) = P(output \mid input, task)$$

因为该语言模型不仅能学习到输入的语料信息，还能进行输出信息的预测，这是 GPT-2 区别于 Bert 模型之处（Bert 模型是 Google 推出的对标 GPT-1 的模型，GPT-2 是对 GPT-1 的升级）。研究人员认为：具有足够能力的语言模型不仅能学习还能推断，并且模型自身能够识别任务类型。因此，原文中作者把该语言模型称之为“无监督多任务学习”[2]。

(二) 相比 GPT-1 的升级之处

GPT-2 是多任务学习模型去除了微调层（fine-tuning layer），即在多任务学习中每一个任务都保证损失函数能收敛的前提下，使得不同任务是共享参数。并且通过训练令模型自动识别下游任务，相较于 GPT-1 更加通用。使得 GPT-2 泛化能力显著提升之处，在于研究人员爬取了 40GB 超大数据集 WebText 进行模型训练，并且将 Transformer 模块堆叠至 48 层，使得该模型的参数进一步暴增，扩大了网络容量具有更强捕获语义信息的能力。此外，GPT-2 增加词汇表数量，批量大小以及词向量的长度，并调整了对 Transformer 做了微改。简而言之，通过增加输入语料数据量和堆叠模型结构增加参数来增强模型性能，提高泛化能力。

二、使用到的关键技术

(一) 注意力机制

生物体（比如：人）在观察、学习、思考行为中的过程的生理机制。即把注意力放在不同的位置，我们就能关注到该位置的信息。注意力机制中三要素为：查询（Query）、键（Key）和值（Value）。即首先将嵌入向量（embedding）转换为 Q，K，V 矩阵，再将注意力结果重新转换为嵌入向量，作为下一层的输入[3]。

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

查询（Query）和 键（key）首先进行矩阵乘法计算相似度，得到相关性的分值后缩放点积输入归一化层（softmax）进行归一化后得到一个注意力的分布，最后和值（value）进行计算，得到一个融合注意力的得分,即找到子序列和全局的权重。

1. 自注意力机制

Transformer 架构中使用的是自注意力机制（self-attention），是对所有的输入计算加权平均，并使用W_k，W_{q}，W_{v}将参数可学习化，即能进行反向传播（back-propagation）的注意力权重参数[4]，具体改进如下：

$$W_{ij}{}' = {X_i}^T X_j$$

$$W_{ij} = \frac{exp(W_{ij}{}')}{\sum_j exp(W_{ij}{}')}$$

因此，自注意力更能关注到输入层的信息，捕获输入序列之间的语义信息。

2. 多头注意力机制

多头注意力（Multi-Head Attention）就是将上述的注意力进行并行计算，然后将并行计算得到的多个注意力头进行合并（concat）得到最终的注意力分数的输出，提高了算法的稳定性。

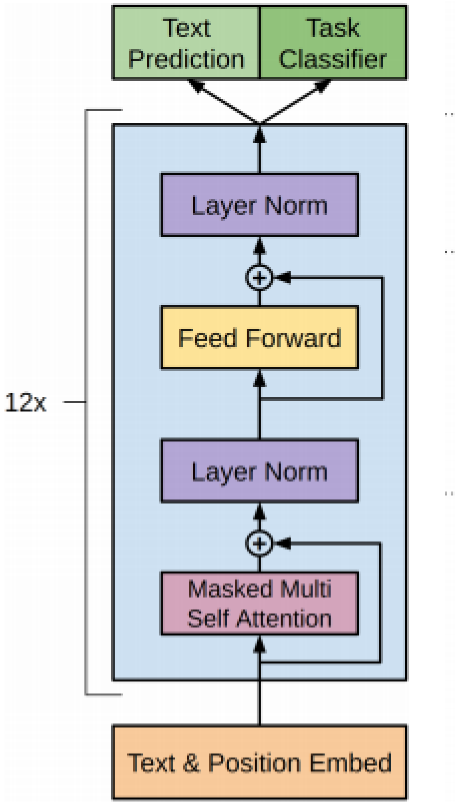
(二) 单向 Transformer 模型

1. 传统 Transformer 架构：解码器 – 编码器

基于 TF-IDF 的统计方法和词袋向量是基于传统机器学习的流程，该自然语言处理流程中每个步骤被认为划分成多个独立的模块组成，即分词、词性标注、句法分析、语义分析等多个独立模块，每个模块的效果会影响到后继模块，从而影响整个训练的结果，这是非端到端的学习。在深度学习时代使用的是得益于类似序列到序列模型(Sequence2Sequence, Seq2Seq)的端到端（End-to-end）的结构，从输入端到输出端会得到一个暂时预测结果，其与真实值的误差会在深度学习模型中的每一层反向传递，每一层根据这个误差调整权重参数，直到模型收敛，除去输入层和输出层的中间层均作为隐藏层[5]。因此，Transformer 整体架构是：输

入 – 编码器（encoder） – 解码器（decoder） – 输出。编码器本质的目的就是对输入生成中间表示，解码器是对中间表示解码，生成目标语言的输出[6]。

2. GPT-2 的 Transformer 架构



图表 1 单向 Transformer[1]

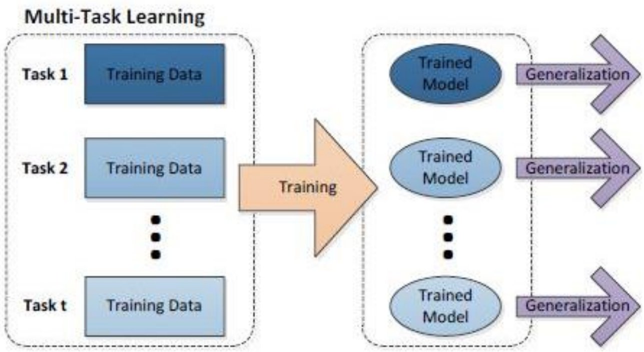
GPT-2 只使用了 Transformer 的解码器结构并进行 12 层堆叠，并且在解码器内部使用了掩蔽的自注意力机制（Masked Self-Attention），即序列向量每个词（token），都只能注意到对包括自己在内的前面所有词，因此是单向 Transformer。具体单向的定义在第一部分语言模型已论述，此处不赘述。

[1] 图源自 Bing 搜索。

(三)多任务学习（Multi-task）和零次学习（Zero-shot）

1. 多任务学习

多任务学习，在我的理解中本质是分治的算法思想，将大问题拆解问题分而治之，最后归并，即有把多个相关且独立子任务同时学习，在整个训练中共享参数。



图表 2 多任务学习[2]

GPT-2 中的多任务学习的具体应用和定义已在第一部分核心思想中论述，此处不赘述。

2. 零次学习

零次学习，本质上是使用无通过监督学习训练的预训练模型进行生成式任务。对于没有标注的数据，使用训练好有语义表征能力的语言模型直接应用到具体任务。

三、应用举例：文本生成

GPT-2 作者所谓的语言模型，即通过上文预测下一个单词，因此可以利用预训练已经学到的知识来生成文本，如：新闻、歌词等。也可以使用另一些数据

[2] 图源自 Bing 搜索。

进行微调，生成有特定格式的文本，如诗歌等，或有特定主题的文本，如：戏剧、小说等。具体应用实现的代码，见附录。

四、模型评价

在被人们称之为划时代的 Bert 模型之后，GPT-2 是巨大的进步，通过多任务学习和零次学习的方式来展开下游任务，是更通用的自然语言处理模型。并且在 2020 年 OpenAI 已推出 GPT-3，但是 OpenAI 和微软是基于商业价值的考虑并没有开源。GPT-3 比 GPT-2 更加通用对于任何输入文本都能生成文本来响应，并且公开了主题为“两个人工智能通过哲学讨论人类”的视频。

在 GPT 系列模型性能好的同时也具有局限性，GPT 系列升级的方式仅通过扩大模型（更大的模型、数据、计算量、参数）来升级。

(一) 文本重复性

该语言模型本质是计算序列生成的条件概率，并且使用极大似然函数拟合估计，即后验概率最大化的思想：

$$argmax_x(P^*(x))$$

既然模型是通过无标注文本进行预训练，那么最大后验概率依赖无标注语料的分布，会出现重复性预测[7]。

(二) 曝光误差

曝光误差是因为文本生成在训练和推断时的不一致，类比时间序列，时序之间具有相关性，通过最大似然函数训练得到的语言模型是一个自回归模型，GPT-2 语言模型算法与时间序列自回归预测类似，只对从目标语料分布中抽取的样本进行训练和评估，使用模型的输出进行回测，会产生曝光误差[8]。

(三) 展望

GPT-2 模型的思想类似于强化学习，但基于强化学习的方法对 GPT2 中的模型优化的研究较少，同时需要考虑文本重复性和曝光误差，因此利用强化学习优化 GPT 模型还有一定距离。

五、参考文献

[1] Radford, Alec and Karthik Narasimhan. Improving Language Understanding by Generative Pre-Training[J]. SEMANTIC SCHOLAR, 2018.

[2] Radford, Alec et al. Language Models are Unsupervised Multitask Learners[J]. SEMANTIC SCHOLAR, 2019.

[3] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30.

[4] Rumelhart D E, Hinton G E, Williams R J. Learning representations by back-propagating errors[J]. Nature, 1986, 323(6088): 533-536. https://doi.org/10.1038/323533a0.

[5] Felp Roza, End-to-end learning, the(almost) every purpose ML method[EB/OI]. (2019-05-31)[2022-05-23]. https://towardsdatascience.com/e2e-the-every-purpose-ml-method-5d4f20dafee4.

[6] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30.

[7] Holtzman, A., Buys, J., Forbes, M., & Choi, Y. The Curious Case of Neural Text Degeneration[J]. Arvix, 2019. abs/1904.09751: n.pag

[8] Xu, Yifan, Kening Zhang, Haoyu Dong, Yuezhou Sun, Wenlong Zhao and Zhuowen Tu. Rethinking Exposure Bias In Language Modeling[J]. ArXiv,2019. abs/1910.11235: n. pag.

六、附录：Pytorch 代码

```
import torch
import numpy as np
from transformers.models.gpt2 import GPT2Config, GPT2LMHeadModel,
```

GPT2Tokenizer

```
from transformers import BertTokenizer
#载入预训练模型
model = GPT2LMHeadModel.from_pretrained("gpt2 通用中文模型")
tokenizer = BertTokenizer(vocab_file="gpt2 通用中文模型/vocab.txt")
inputs_text = "成为人工智能好孤独"
max_length = 30
input_ids = []
input_ids.extend(tokenizer.encode(inputs_text))
input_ids = input_ids[:-1]
for i in range(max_length):
    inputs = {"input_ids": torch.tensor([input_ids])}
    outputs = model(**inputs)
    logits = outputs.logits
    last_token_id = int(np.argmax(logits[0][-1].detach().numpy()))
    last_token = tokenizer.convert_ids_to_tokens(last_token_id)
    inputs_text += last_token
    input_ids.append(last_token_id)
print(inputs_text)
#创建数据流
from torch.utils.data import Dataset, DataLoader
class MyDataset(Dataset):
    def __init__(self, data_list):
        self.data_list = data_list
    def __getitem__(self, index):
        input_ids = self.data_list[index]
        return input_ids
    def __len__(self):
        return len(self.data_list)
```

```
def collate_fn(batch):
    input_ids = []
    input_lens_list = [len(w) for w in batch]
    max_input_len = max(input_lens_list)
    for btc_idx in range(len(batch)):
        input_len = len(batch[btc_idx])
        input_ids.append(batch[btc_idx])
        input_ids[btc_idx].extend([tokenizer.pad_token_id] * (max_input_len -
input_len))
    return torch.tensor(input_ids, dtype=torch.long)
dataset = MyDataset(data_list)
dataloader = DataLoader(dataset=dataset,
                        batch_size=args.batch_size,
                        shuffle=True,
                        collate_fn=collate_fn)
#定义优化器和训练参数
optimizer = AdamW(model.parameters(), lr=args.lr)
lr_scheduler = get_scheduler(
        name="linear",
        optimizer=optimizer,
        num_warmup_steps=args.warmup_steps,
        num_training_steps=num_training_steps)
#构建验证指标
def rouge(not_ignore, shift_labels, preds):
    main_rouge = Rouge()
```

```
        true_length = [w.sum() for w in not_ignore.float()]
        rouge_labels = []
        rouge_predicts = []
        for idx, tmp_len in enumerate(true_length):
                tmp_labels = shift_labels[idx][:int(tmp_len)]
                rouge_labels.append(" ".join([str(w) for w in tmp_labels.tolist()]))
                tmp_pred = preds[idx][:int(tmp_len)]
                rouge_predicts.append(" ".join([str(w) for w in tmp_pred.tolist()]))
        rouge_score = main_rouge.get_scores(rouge_predicts, rouge_labels, avg=True)
return rouge_score


def calculate_loss_and_accuracy(outputs, labels, device):
logits = outputs.logits


        # Shift so that tokens < n predict n
        shift_logits = logits[..., :-1, :].contiguous()
shift_labels = labels[..., 1:].contiguous().to(device)


        # Flatten the tokens
        loss_fct = CrossEntropyLoss(ignore_index=tokenizer.pad_token_id,
reduction='sum')
        loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1))
_, preds = shift_logits.max(dim=-1)


        not_ignore = shift_labels.ne(tokenizer.pad_token_id)
```

```
        num_targets = not_ignore.long().sum().item()

        correct = (shift_labels == preds) & not_ignore
        correct = correct.float().sum()

        accuracy = correct / num_targets
        loss = loss / num_targets

        rouge_score = rouge(not_ignore, shift_labels, preds)
return loss, accuracy, rouge_score
#开始训练
batch_steps = 0
for epoch in range(args.epochs):
        for batch in dataloader:
                batch_steps += 1
                inputs = {"input_ids": batch.to(device)}
                outputs = model(**inputs, labels=batch.to(device))
                # loss = outputs.loss
                loss, acc, rouge_score = calculate_loss_and_accuracy(outputs,
batch.to(device), device)
                loss.backward()
                torch.nn.utils.clip_grad_norm_(model.parameters(), args.max_grad_norm)
                optimizer.step()
                lr_scheduler.step()
                optimizer.zero_grad()
```

```
#验证集评估
def evaluate(dataloader, args):
    device = torch.device("cuda") if torch.cuda.is_available() else
torch.device("cpu")
    model, _ = load_model(args.save_model_path, args.vocab_path)
    model.to(device)
    model.eval()
    loss_list, acc_list, rouge_1_list, rouge_2_list, rouge_l_list = [], [], [], [], []
    batch_steps = 0
    with torch.no_grad():
        for batch in dataloader:
            batch_steps += 1
            inputs = {"input_ids": batch.to(device)}
            outputs = model(**inputs, labels=batch.to(device))
            loss, acc, rouge_score = calculate_loss_and_accuracy(outputs,
batch.to(device), device)
#在预训练模型上继续预训练
model_config =
transformers.modeling_gpt2.GPT2Config.from_json_file(args.model_config)
model = GPT2LMHeadModel(config=model_config)

#英文版 gpt2 预训练模型使用
#!/usr/bin/env Python
# coding=utf-8
from transformers import GPT2LMHeadModel, GPT2Tokenizer
```

```
import torch

# 初始化 GPT2 模型的 Tokenizer 类.
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
# 初始化 GPT2 模型，此处以初始化 GPT2LMHeadModel()类的方式调用 GPT2 模型.
model = GPT2LMHeadModel.from_pretrained('gpt2')
model.config.use_return_dict = None
print(model.config.use_return_dict)

generated = tokenizer.encode("The Manhattan bridge")
context = torch.tensor([generated])
past_key_values = None
for i in range(30):
    CausalLMOutputWithPastAndCrossAttentions(
            loss=loss,
            logits=lm_logits,
            past_key_values=transformer_outputs.past_key_values,
            hidden_states=transformer_outputs.hidden_states,
            attentions=transformer_outputs.attentions,
            cross_attentions=transformer_outputs.cross_attentions,
        )
    output = model(context, past_key_values=past_key_values)
    past_key_values = output.past_key_values
    token = torch.argmax(output.logits[..., -1, :])
```

```
        context = token.unsqueeze(0)

        generated += [token.tolist()]

sequence = tokenizer.decode(generated)

sequence = sequence.split(".")[:-1]

print(sequence)


#各功能块的从零实现

class GPT2LMHeadModel(GPT2PreTrainedModel):

        _keys_to_ignore_on_load_missing = [r"h\.\d+\.attn\.masked_bias",
r"lm_head\.weight"]


        def __init__(self, config):

                super().__init__(config)

                # 初始化 GPT2Model(config)类.

                self.transformer = GPT2Model(config)

                self.init_weights()


        def get_output_embeddings(self):

                return self.lm_head


        def prepare_inputs_for_generation(self, input_ids, past=None, **kwargs):

                token_type_ids = kwargs.get("token_type_ids", None)

                # only last token for inputs_ids if past is defined in kwargs

                if past:

                        input_ids = input_ids[:, -1].unsqueeze(-1)
```

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

    attention_mask = kwargs.get("attention_mask", None)
    position_ids = kwargs.get("position_ids", None)

    if attention_mask is not None and position_ids is None:
        # create position_ids on the fly for batch generation
        position_ids = attention_mask.long().cumsum(-1) - 1
        position_ids.masked_fill_(attention_mask == 0, 1)
        if past:
            position_ids = position_ids[:, -1].unsqueeze(-1)
    else:
        position_ids = None
    return {
        "input_ids": input_ids,
        "past_key_values": past,
        "use_cache": kwargs.get("use_cache"),
        "position_ids": position_ids,
        "attention_mask": attention_mask,
        "token_type_ids": token_type_ids,
    }

@add_start_docstrings_to_model_forward(GPT2_INPUTS_DOCSTRING)
@add_code_sample_docstrings(
```

```
        tokenizer_class=_TOKENIZER_FOR_DOC,
        checkpoint="gpt2",
        output_type=CausalLMOutputWithPastAndCrossAttentions,
        config_class=_CONFIG_FOR_DOC,
    )
    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,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        labels=None,
        use_cache=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        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=encoder_hidden_states,
            encoder_attention_mask=encoder_attention_mask,
            use_cache=use_cache,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
        )
        hidden_states = transformer_outputs[0]
        loss = None
        if labels is not None:
            # Shift so that tokens < n predict n
            shift_logits = lm_logits[..., :-1, :].contiguous()
            shift_labels = labels[..., 1:].contiguous()
            # Flatten the tokens
            loss_fct = CrossEntropyLoss()
            loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)),
shift_labels.view(-1))
```

```
        if not return_dict:
            output = (lm_logits,) + transformer_outputs[1:]
            return ((loss,) + output) if loss is not None else output

        return CausalLMOutputWithPastAndCrossAttentions(
            loss=loss,
            logits=lm_logits,
            past_key_values=transformer_outputs.past_key_values,
            hidden_states=transformer_outputs.hidden_states,
            attentions=transformer_outputs.attentions,
            cross_attentions=transformer_outputs.cross_attentions,
        )

#定义 GPT2 模型
class GPT2Model(GPT2PreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.wte = nn.Embedding(config.vocab_size, config.n_embd)
        self.wpe = nn.Embedding(config.n_positions, config.n_embd)
        self.drop = nn.Dropout(config.embd_pdrop)
        self.h = nn.ModuleList([Block(config.n_ctx, config, scale=True) for _ in range(config.n_layer)])
        self.ln_f = nn.LayerNorm(config.n_embd, eps=config.layer_norm_epsilon)
        self.init_weights()
```

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

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

def _prune_heads(self, heads_to_prune):
    for layer, heads in heads_to_prune.items():
        self.h[layer].attn.prune_heads(heads)

@add_start_docstrings_to_model_forward(GPT2_INPUTS_DOCSTRING)
@add_code_sample_docstrings(
    tokenizer_class=_TOKENIZER_FOR_DOC,
    checkpoint="gpt2",
    output_type=BaseModelOutputWithPastAndCrossAttentions,
    config_class=_CONFIG_FOR_DOC,
)
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,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        use_cache=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        output_attentions = output_attentions if output_attentions is not None else
self.config.output_attentions
        output_hidden_states = (
            output_hidden_states if output_hidden_states is not None else
self.config.output_hidden_states
        )
        use_cache = use_cache if use_cache is not None else self.config.use_cache
        return_dict = return_dict if return_dict is not None else
self.config.use_return_dict
        if input_ids is not None and inputs_embeds is not None:
            raise ValueError("You cannot specify both input_ids and
inputs_embeds at the same time")
        elif input_ids is not None:
            input_shape = input_ids.size()
            input_ids = input_ids.view(-1, input_shape[-1])
            batch_size = input_ids.shape[0]
```

```
        elif inputs_embeds is not None:
            input_shape = inputs_embeds.size()[:-1]
            batch_size = inputs_embeds.shape[0]
        else:
            raise ValueError("You have to specify either input_ids or
inputs_embeds")

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

        if past_key_values is None:
            past_length = 0
            past_key_values = [None] * len(self.h)
        else:
            past_length = past_key_values[0][0].size(-2)
        if position_ids is None:
            device = input_ids.device if input_ids is not None else
inputs_embeds.device

            position_ids = torch.arange(past_length, input_shape[-1] + past_length,
dtype=torch.long, device=device)
            position_ids = position_ids.unsqueeze(0).view(-1, input_shape[-1])
```

```
# Attention mask.
if attention_mask is not None:
    assert batch_size > 0, "batch_size has to be defined and > 0"
    attention_mask = attention_mask.view(batch_size, -1)
    attention_mask = attention_mask[:, None, None, :]
    attention_mask = attention_mask.to(dtype=self.dtype)    # fp16
compatibility
    attention_mask = (1.0 - attention_mask) * -10000.0
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)
    if encoder_attention_mask is None:
        encoder_attention_mask = torch.ones(encoder_hidden_shape,
device=device)
    encoder_attention_mask =
self.invert_attention_mask(encoder_attention_mask)
else:
    encoder_attention_mask = None
head_mask = self.get_head_mask(head_mask, self.config.n_layer)
# inputs_embeds、position_embeds 与 token_type_embeds.
if inputs_embeds is None:
    inputs_embeds = self.wte(input_ids)
position_embeds = self.wpe(position_ids)
```

```
        hidden_states = inputs_embeds + position_embeds

        if token_type_ids is not None:
            token_type_embeds = self.wte(token_type_ids)
            hidden_states = hidden_states + token_type_embeds
        hidden_states = self.drop(hidden_states)
        output_shape = input_shape + (hidden_states.size(-1),)
        # config 对应的 GPT2Config()类中的 use_cache 默认为 True.
        presents = () if use_cache else None
        all_self_attentions = () if output_attentions else None
        all_cross_attentions = () if output_attentions and
self.config.add_cross_attention else None
        all_hidden_states = () if output_hidden_states else None

        for i, (block, layer_past) in enumerate(zip(self.h, past_key_values)):
            if output_hidden_states:
                all_hidden_states = all_hidden_states +
(hidden_states.view(*output_shape),)

            if getattr(self.config, "gradient_checkpointing", False):
                def create_custom_forward(module):
                    def custom_forward(*inputs):
                        # checkpointing only works with tuple returns, not with
lists
```

```
                    return tuple(output for output in module(*inputs,
use_cache, output_attentions))

                return custom_forward

            outputs = torch.utils.checkpoint.checkpoint(
                create_custom_forward(block),
                hidden_states,
                layer_past,
                attention_mask,
                head_mask[i],
                encoder_hidden_states,
                encoder_attention_mask,
            )
        else:
            outputs = block(
                hidden_states,
                layer_past=layer_past,
                attention_mask=attention_mask,
                head_mask=head_mask[i],
                encoder_hidden_states=encoder_hidden_states,
                encoder_attention_mask=encoder_attention_mask,
                use_cache=use_cache,
                output_attentions=output_attentions,
            )
```

```
        hidden_states, present = outputs[:2]
        if use_cache is True:
            presents = presents + (present,)

        if output_attentions:
            all_self_attentions = all_self_attentions + (outputs[2],)
            if self.config.add_cross_attention:
                all_cross_attentions = all_cross_attentions + (outputs[3],)


    hidden_states = self.ln_f(hidden_states)
    hidden_states = hidden_states.view(*output_shape)

    if output_hidden_states:
        all_hidden_states = all_hidden_states + (hidden_states,)

    if not return_dict:
        return tuple(v for v in [hidden_states, presents, all_hidden_states,
all_self_attentions] if v is not None)

    return BaseModelOutputWithPastAndCrossAttentions(
        last_hidden_state=hidden_states,
        past_key_values=presents,
        hidden_states=all_hidden_states,
```

```
                attentions=all_self_attentions,
                cross_attentions=all_cross_attentions,
            )
#transformerblock
class Block(nn.Module):
    def __init__(self, n_ctx, config, scale=False):
        super().__init__()
        hidden_size = config.n_embd
        inner_dim = config.n_inner if config.n_inner is not None else 4 *
hidden_size
        self.ln_1 = nn.LayerNorm(hidden_size, eps=config.layer_norm_epsilon)
        # 1024.
        self.attn = Attention(hidden_size, n_ctx, config, scale)
        self.ln_2 = nn.LayerNorm(hidden_size, eps=config.layer_norm_epsilon)

        if config.add_cross_attention:
            self.crossattention = Attention(hidden_size, n_ctx, config, scale,
is_cross_attention=True)
            self.ln_cross_attn = nn.LayerNorm(hidden_size,
eps=config.layer_norm_epsilon)
        self.mlp = MLP(inner_dim, config)

    def forward(
        self,
        hidden_states,
```

```
        layer_past=None,
        attention_mask=None,
        head_mask=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        use_cache=False,
        output_attentions=False,
    ):
        attn_outputs = self.attn(
            self.ln_1(hidden_states),
            layer_past=layer_past,
            attention_mask=attention_mask,
            head_mask=head_mask,
            use_cache=use_cache,
            output_attentions=output_attentions,
        )
        attn_output = attn_outputs[0]   # output_attn 列表: a, present, (attentions)
        outputs = attn_outputs[1:]
        # residual connection，进行残差连接.
        # hidden_states 的形状为(batch_size, 1, 768).
        hidden_states = attn_output + hidden_states
        if encoder_hidden_states is not None:

            cross_attn_outputs = self.crossattention(
                self.ln_cross_attn(hidden_states),
```

```
                attention_mask=attention_mask,
                head_mask=head_mask,
                encoder_hidden_states=encoder_hidden_states,
                encoder_attention_mask=encoder_attention_mask,
                output_attentions=output_attentions,
            )
            attn_output = cross_attn_outputs[0]
            # residual connection
            hidden_states = hidden_states + attn_output

            outputs = outputs + cross_attn_outputs[2:]



        feed_forward_hidden_states = self.mlp(self.ln_2(hidden_states))
        # residual connection
        hidden_states = hidden_states + feed_forward_hidden_states


        outputs = [hidden_states] + outputs
        return outputs    # hidden_states, present, (attentions, cross_attentions)
# 注意力机制太难了，用 numpy 写了个对象
class model:
    #将预训练好的整个权重字典输入进来
    def __init__(self, state_dict):
        self.num_attention_heads = 12
        self.hidden_size = 768
```

```
        self.num_layers = 12
        self.load_weights(state_dict)


    def load_weights(self, state_dict):
        #embedding 部分
        self.word_embeddings =
state_dict["embeddings.word_embeddings.weight"].numpy()
        self.position_embeddings =
state_dict["embeddings.position_embeddings.weight"].numpy()
        self.token_type_embeddings =
state_dict["embeddings.token_type_embeddings.weight"].numpy()
        self.embeddings_layer_norm_weight =
state_dict["embeddings.LayerNorm.weight"].numpy()
        self.embeddings_layer_norm_bias =
state_dict["embeddings.LayerNorm.bias"].numpy()
        self.transformer_weights = []
        #transformer 部分，有多层
        for i in range(self.num_layers):
            q_w = state_dict["encoder.layer.%d.attention.self.query.weight" %
i].numpy()
            q_b = state_dict["encoder.layer.%d.attention.self.query.bias" %
i].numpy()
            k_w = state_dict["encoder.layer.%d.attention.self.key.weight" %
i].numpy()
```

k_b = state_dict["encoder.layer.%d.attention.self.key.bias" % i].numpy()

v_w = state_dict["encoder.layer.%d.attention.self.value.weight" % i].numpy()

v_b = state_dict["encoder.layer.%d.attention.self.value.bias" % i].numpy()

attention_output_weight = state_dict["encoder.layer.%d.attention.output.dense.weight" % i].numpy()

attention_output_bias = state_dict["encoder.layer.%d.attention.output.dense.bias" % i].numpy()

attention_layer_norm_w = state_dict["encoder.layer.%d.attention.output.LayerNorm.weight" % i].numpy()

attention_layer_norm_b = state_dict["encoder.layer.%d.attention.output.LayerNorm.bias" % i].numpy()

intermediate_weight = state_dict["encoder.layer.%d.intermediate.dense.weight" % i].numpy()

intermediate_bias = state_dict["encoder.layer.%d.intermediate.dense.bias" % i].numpy()

output_weight = state_dict["encoder.layer.%d.output.dense.weight" % i].numpy()

output_bias = state_dict["encoder.layer.%d.output.dense.bias" % i].numpy()

ff_layer_norm_w = state_dict["encoder.layer.%d.output.LayerNorm.weight" % i].numpy()

```
                ff_layer_norm_b =
state_dict["encoder.layer.%d.output.LayerNorm.bias" % i].numpy()
                self.transformer_weights.append([q_w, q_b, k_w, k_b, v_w, v_b,
attention_output_weight, attention_output_bias,
                                                 attention_layer_norm_w,
attention_layer_norm_b, intermediate_weight, intermediate_bias,
                                                 output_weight, output_bias,
ff_layer_norm_w, ff_layer_norm_b])
            #pooler 层
            self.pooler_dense_weight = state_dict["pooler.dense.weight"].numpy()
            self.pooler_dense_bias = state_dict["pooler.dense.bias"].numpy()


        #embedding
        def embedding_forward(self, x):
            # x.shape = [max_len]
            we = self.get_embedding(self.word_embeddings, x)   # shpae: [max_len,
hidden_size]
            # position embeding 的输入  [0, 1, 2, 3]
            pe = self.get_embedding(self.position_embeddings,
np.array(list(range(len(x)))))   # shpae: [max_len, hidden_size]
            # token type embedding,单输入的情况下为[0, 0, 0, 0]
            te = self.get_embedding(self.token_type_embeddings, np.array([0] * len(x)))
# shpae: [max_len, hidden_size]
            embedding = we + pe + te
            #  加和后有一个归一化层
```

```
        embedding = self.layer_norm(embedding, self.embeddings_layer_norm_weight, self.embeddings_layer_norm_bias)    # shpae: [max_len, hidden_size]
        return embedding

    #embedding层实际上相当于按index索引，或理解为onehot输入乘以embedding矩阵
    def get_embedding(self, embedding_matrix, x):
        return np.array([embedding_matrix[index] for index in x])

    #执行全部的transformer层计算
    def all_transformer_layer_forward(self, x):
        for i in range(self.num_layers):
            x = self.single_transformer_layer_forward(x, i)
        return x

    #执行单层transformer层计算
    def single_transformer_layer_forward(self, x, layer_index):
        weights = self.transformer_weights[layer_index]
        #取出该层的参数，在实际中，这些参数都是随机初始化，之后进行预训练
        q_w, q_b, \
        k_w, k_b, \
        v_w, v_b, \
        attention_output_weight, attention_output_bias, \
```

```
        attention_layer_norm_w, attention_layer_norm_b, \
        intermediate_weight, intermediate_bias, \
        output_weight, output_bias, \
        ff_layer_norm_w, ff_layer_norm_b = weights
        #self attention 层
        attention_output = self.self_attention(x,
                                q_w, q_b,
                                k_w, k_b,
                                v_w, v_b,
                                attention_output_weight,
attention_output_bias,
                                self.num_attention_heads,
                                self.hidden_size)
        #bn 层，并使用了残差机制
        x = self.layer_norm(x + attention_output, attention_layer_norm_w,
attention_layer_norm_b)
        #feed forward 层
        feed_forward_x = self.feed_forward(x,
                              intermediate_weight, intermediate_bias,
                              output_weight, output_bias)
        #bn 层，并使用了残差机制
        x = self.layer_norm(x + feed_forward_x, ff_layer_norm_w,
ff_layer_norm_b)
        return x
```

```
# self attention 的计算
def self_attention(self,
                   x,
                   q_w,
                   q_b,
                   k_w,
                   k_b,
                   v_w,
                   v_b,
                   attention_output_weight,
                   attention_output_bias,
                   num_attention_heads,
                   hidden_size):
    # x.shape = max_len * hidden_size
    # q_w, k_w, v_w    shape = hidden_size * hidden_size
    # q_b, k_b, v_b    shape = hidden_size
    q = np.dot(x, q_w.T) + q_b    # shape: [max_len, hidden_size]          W * X
+ B lINER
    k = np.dot(x, k_w.T) + k_b    # shpae: [max_len, hidden_size]
    v = np.dot(x, v_w.T) + v_b    # shpae: [max_len, hidden_size]
    attention_head_size = int(hidden_size / num_attention_heads)
    # q.shape = num_attention_heads, max_len, attention_head_size
    q = self.transpose_for_scores(q, attention_head_size, num_attention_heads)
    # k.shape = num_attention_heads, max_len, attention_head_size
    k = self.transpose_for_scores(k, attention_head_size, num_attention_heads)
```

```
        # v.shape = num_attention_heads, max_len, attention_head_size
        v = self.transpose_for_scores(v, attention_head_size, num_attention_heads)
        # qk.shape = num_attention_heads, max_len, max_len
        # q = 8 * 10 * 96
        # k = 8 * 96 * 10
        qk = np.matmul(q, k.swapaxes(1, 2))    # 8 * 10 * 10
        qk /= np.sqrt(attention_head_size)
        qk = softmax(qk)
        # qkv.shape = num_attention_heads, max_len, attention_head_size
        # v = 8 * 10 * 96
        qkv = np.matmul(qk, v)    # 10 * 8 * 96 - > 10 * 768
        # qkv.shape = max_len, hidden_size
        qkv = qkv.swapaxes(0, 1).reshape(-1, hidden_size)
        # attention.shape = max_len, hidden_size
        attention = np.dot(qkv, attention_output_weight.T) + attention_output_bias
        return attention

    def transpose_for_scores(self, x, attention_head_size, num_attention_heads):
        # hidden_size = 768    num_attent_heads = 8 attention_head_size = 96
        max_len, hidden_size = x.shape
        x = x.reshape(max_len, num_attention_heads, attention_head_size)
        # 10 * 768    - >    Q: 8 * 10 * 96        k: 8 * 96 * 10    qk : 8 * 10 * 10
        x = x.swapaxes(1, 0)    # output shape = [num_attention_heads, max_len,
attention_head_size]
        return x
```

```
    #前馈网络的计算
    def feed_forward(self,
                     x,
                     intermediate_weight,    # intermediate_size, hidden_size
                     intermediate_bias,    # intermediate_size
                     output_weight,    # hidden_size, intermediate_size
                     output_bias,    # hidden_size
                     ):
        # output shpae: [max_len, intermediate_size]
        x = np.dot(x, intermediate_weight.T) + intermediate_bias
        x = gelu(x)
        # output shpae: [max_len, hidden_size]
        x = np.dot(x, output_weight.T) + output_bias
        return x

    #归一化层
    def layer_norm(self, x, w, b):
        x = (x - np.mean(x, axis=1, keepdims=True)) / np.std(x, axis=1,
keepdims=True)
        x = x * w + b
        return x

    #链接[cls] token 的输出层
    def pooler_output_layer(self, x):
```

```
        x = np.dot(x, self.pooler_dense_weight.T) + self.pooler_dense_bias
        x = np.tanh(x)
        return x

    #最终输出
    def forward(self, x):
        x = self.embedding_forward(x)
        sequence_output = self.all_transformer_layer_forward(x)
        pooler_output = self.pooler_output_layer(sequence_output[0])
        return sequence_output, pooler_output
```