

其中 μ 和 σ 分别表示均值和方差，用于将数据平移缩放到均值为 0、方差为 1 的标准分布， α 和 b 是可学习的参数。层归一化技术可以有效地缓解优化过程中潜在的不稳定、收敛速度慢等问题。

使用 PyTorch 实现的层归一化参考代码如下：

```
class Norm(nn.Module):

    def __init__(self, d_model, eps = 1e-6):
        super().__init__()

        self.size = d_model

        # 层归一化包含两个可以学习的参数
        self.alpha = nn.Parameter(torch.ones(self.size))
        self.bias = nn.Parameter(torch.zeros(self.size))

        self.eps = eps

    def forward(self, x):
        norm = self.alpha * (x - x.mean(dim=-1, keepdim=True)) \
            / (x.std(dim=-1, keepdim=True) + self.eps) + self.bias
        return norm
```

2.1.5 编码器和解码器结构

基于上述模块，根据图2.1 给出的网络架构，编码器端较容易实现。相比于编码器端，解码器端更复杂。具体来说，解码器的每个 Transformer 块的第一个自注意力子层额外增加了注意力掩码，对应图中的掩码多头注意力（Masked Multi-Head Attention）部分。这主要是因为翻译的过程中，编码器端主要用于编码源语言序列的信息，而这个序列是完全已知的，因而编码器仅需要考虑如何融合上下文语义信息。解码器端则负责生成目标语言序列，这一生成过程是自回归的，即对于每一个单词的生成过程，仅有当前单词之前的目标语言序列是可以被观测的，因此这一额外增加的掩码是用来掩盖后续的文本信息的，以防模型在训练阶段直接看到后续的文本序列，进而无法得到有效的训练。

此外，解码器端额外增加了一个多头交叉注意力（Multi-Head Cross-Attention）模块，使用交叉注意力（Cross-Attention）方法，同时接收来自编码器端的输出和当前 Transformer 块的前一个掩码注意力层的输出。查询是通过解码器前一层的输出进行投影的，而键和值是使用编码器的输出进行投影的。它的作用是在翻译的过程中，为了生成合理的目标语言序列，观测待翻译的源语言序列是什么。基于上述编码器和解码器结构，待翻译的源语言文本经过编码器端的每个 Transformer 块对其上下文语义进行层层抽象，最终输出每一个源语言单词上下文相关的表示。解码器端以自回归的方式生成目标语言文本，即在每个时间步 t ，根据编码器端输出的源语言文本表示，以及前 $t - 1$ 个时刻生成的目标语言文本，生成当前时刻的目标语言单词。

使用 PyTorch 实现的编码器参考代码如下：

```

class EncoderLayer(nn.Module):

    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.attn = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.ff = FeedForward(d_model, dropout=dropout)
        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)

    def forward(self, x, mask):
        attn_output = self.attn(x, x, x, mask)
        attn_output = self.dropout_1(attn_output)
        x = x + attn_output
        x = self.norm_1(x)
        ff_output = self.ff(x)
        ff_output = self.dropout_2(ff_output)
        x = x + ff_output
        x = self.norm_2(x)
        return x


class Encoder(nn.Module):

    def __init__(self, vocab_size, d_model, N, heads, dropout):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model, dropout=dropout)
        self.layers = get_clones(EncoderLayer(d_model, heads, dropout), N)
        self.norm = Norm(d_model)

    def forward(self, src, mask):
        x = self.embed(src)
        x = self.pe(x)
        for i in range(self.N):
            x = self.layers[i](x, mask)
        return self.norm(x)

```

使用 PyTorch 实现的解码器参考代码如下：

```

class DecoderLayer(nn.Module):

    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.norm_3 = Norm(d_model)

        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)
        self.dropout_3 = nn.Dropout(dropout)

        self.attn_1 = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.attn_2 = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.ff = FeedForward(d_model, dropout=dropout)

    def forward(self, x, e_outputs, src_mask, trg_mask):
        attn_output_1 = self.attn_1(x, x, x, trg_mask)
        attn_output_1 = self.dropout_1(attn_output_1)
        x = x + attn_output_1
        x = self.norm_1(x)
        attn_output_2 = self.attn_2(x, e_outputs, e_outputs, src_mask)
        attn_output_2 = self.dropout_2(attn_output_2)
        x = x + attn_output_2
        x = self.norm_2(x)

        ff_output = self.ff(x)
        ff_output = self.dropout_3(ff_output)
        x = x + ff_output
        x = self.norm_3(x)

        return x


class Decoder(nn.Module):

    def __init__(self, vocab_size, d_model, N, heads, dropout):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model, dropout=dropout)
        self.layers = get_clones(DecoderLayer(d_model, heads, dropout), N)
        self.norm = Norm(d_model)

    def forward(self, trg, e_outputs, src_mask, trg_mask):
        x = self.embed(trg)
        x = self.pe(x)

```

基于 Transformer 的编码器和解码器结构整体实现的参考代码如下：

```
class Transformer(nn.Module):

    def __init__(self, src_vocab, trg_vocab, d_model, N, heads, dropout):
        super().__init__()
        self.encoder = Encoder(src_vocab, d_model, N, heads, dropout)
        self.decoder = Decoder(trg_vocab, d_model, N, heads, dropout)
        self.out = nn.Linear(d_model, trg_vocab)

    def forward(self, src, trg, src_mask, trg_mask):
        e_outputs = self.encoder(src, src_mask)
        d_output = self.decoder(trg, e_outputs, src_mask, trg_mask)
        output = self.out(d_output)
        return output
```

可以使用如下代码对上述模型结构进行训练和测试：

```

# 模型参数定义
d_model = 512
heads = 8
N = 6
src_vocab = len(EN_TEXT.vocab)
trg_vocab = len(FR_TEXT.vocab)
model = Transformer(src_vocab, trg_vocab, d_model, N, heads)
for p in model.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)

optim = torch.optim.Adam(model.parameters(), lr=0.0001, betas=(0.9, 0.98), eps=1e-9)

# 模型训练
def train_model(epochs, print_every=100):

    model.train()

    start = time.time()
    temp = start

    total_loss = 0

    for epoch in range(epochs):

        for i, batch in enumerate(train_iter):
            src = batch.English.transpose(0,1)
            trg = batch.French.transpose(0,1)
            # 将我们输入的英语句子中的所有单词翻译成法语
            # 除了最后一个单词，因为它为结束符，不需要进行下一个单词的预测

            trg_input = trg[:, :-1]

            # 试图预测单词
            targets = trg[:, 1:].contiguous().view(-1)

            # 使用掩码代码创建函数来制作掩码
            src_mask, trg_mask = create_masks(src, trg_input)

            preds = model(src, trg_input, src_mask, trg_mask)

            optim.zero_grad()

            loss = F.cross_entropy(preds.view(-1, preds.size(-1)),
                                   results, ignore_index=target_pad)
            loss.backward()
            optim.step()

```

2.2 生成式预训练语言模型 GPT

受到计算机视觉领域采用 ImageNet^[9] 对模型进行一次预训练，使得模型可以通过海量图像充分学习如何提取特征，再根据任务目标进行模型微调的范式影响，自然语言处理领域基于预训练语言模型的方法也逐渐成为主流。以 ELMo^[10] 为代表的动态词向量模型开启了语言模型预训练的大门，此后，以 GPT^[11] 和 BERT^[1] 为代表的基于 Transformer 的大规模预训练语言模型的出现，使得自然语言处理全面进入了预训练微调范式新时代。利用丰富的训练数据、自监督的预训练任务及 Transformer 等深度神经网络结构，预训练语言模型具备了通用且强大的自然语言表示能力，能够有效地学习到词汇、语法和语义信息。将预训练模型应用于下游任务时，不需要了解太多的任务细节，不需要设计特定的神经网络结构，只需要“微调”预训练模型，即使用具体任务的标注数据在预训练语言模型上进行监督训练，就可以取得显著的性能提升。

OpenAI 公司在 2018 年提出的生成式预训练语言模型（Generative Pre-Training, GPT）^[11] 是典型的生成式预训练语言模型之一。GPT 的模型结构如图 2.3 所示，它是由多层 Transformer 组成的单向语言模型，主要分为输入层、编码层和输出层三部分。

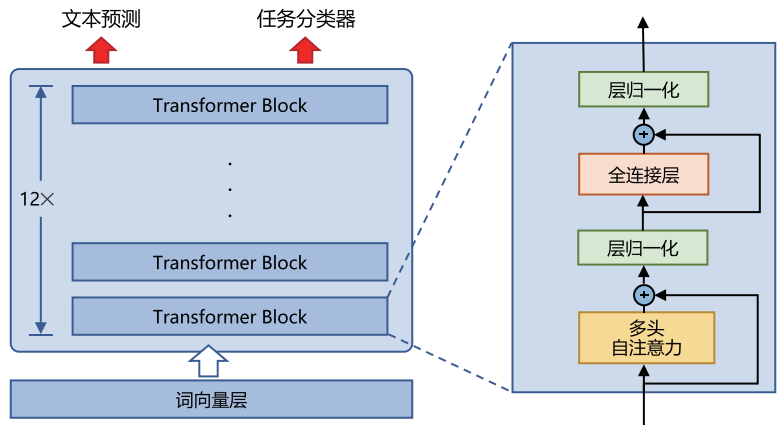


图 2.3 GPT 的模型结构

本节将重点介绍 GPT 自监督预训练、有监督下游任务微调及基于 HuggingFace 的预训练语言模型实践。

2.2.1 自监督预训练

GPT 采用生成式预训练方法，单向意味着模型只能从左到右或从右到左对文本序列建模，所采用的 Transformer 结构和解码策略保证了输入文本每个位置只能依赖过去时刻的信息。

给定文本序列 $w = w_1, w_2, \dots, w_n$, GPT 首先在输入层中将其映射为稠密的向量:

$$\mathbf{v}_i = \mathbf{v}_i^t + \mathbf{v}_i^p \quad (2.12)$$

其中, \mathbf{v}_i^t 是词 w_i 的词向量, \mathbf{v}_i^p 是词 w_i 的位置向量, \mathbf{v}_i 为第 i 个位置的单词经过模型输入层 (第 0 层) 后的输出。GPT 模型的输入层与前文中介绍的神经网络语言模型的不同之处在于其需要添加位置向量, 这是 Transformer 结构自身无法感知位置导致的, 因此需要来自输入层的额外位置信息。

经过输入层编码, 模型得到表示向量序列 $\mathbf{v} = \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$, 随后将 \mathbf{v} 送入模型编码层。编码层由 L 个 Transformer 模块组成, 在自注意力机制的作用下, 每一层的每个表示向量都会包含之前位置表示向量的信息, 使每个表示向量都具备丰富的上下文信息, 而且, 经过多层编码, GPT 能得到每个单词层次化的组合式表示, 其计算过程表示为:

$$\mathbf{h}^{(l)} = \text{Transformer-Block}^{(l)}(\mathbf{h}^{(0)}) \quad (2.13)$$

其中 $\mathbf{h}^{(l)} \in \mathbb{R}^{d \times n}$ 表示第 l 层的表示向量序列, n 为序列长度, d 为模型隐藏层维度, L 为模型总层数。

GPT 模型的输出层基于最后一层的表示 $\mathbf{h}^{(L)}$, 预测每个位置上的条件概率, 其计算过程可以表示为

$$P(w_i | w_1, w_2, \dots, w_{i-1}) = \text{Softmax}(\mathbf{W}^e \mathbf{h}_i^{(L)} + \mathbf{b}^{\text{out}}) \quad (2.14)$$

其中, $\mathbf{W}^e \in \mathbb{R}^{|\mathcal{V}| \times d}$ 为词向量矩阵, $|\mathcal{V}|$ 为词表大小。

单向语言模型按照阅读顺序输入文本序列 w , 用常规语言模型目标优化 w 的最大似然估计, 使之能根据输入历史序列对当前词做出准确的预测:

$$\mathcal{L}^{\text{PT}}(w) = - \sum_{i=1}^n \log P(w_i | w_0, w_1, \dots, w_{i-1}; \boldsymbol{\theta}) \quad (2.15)$$

其中 $\boldsymbol{\theta}$ 代表模型参数。也可以基于马尔可夫假设, 只使用部分过去词进行训练。预训练时通常使用随机梯度下降法进行反向传播, 优化该负对数似然函数。

2.2.2 有监督下游任务微调

通过自监督语言模型预训练, 使得 GPT 模型具备了一定的通用语义表示能力。下游任务微调 (Downstream Task Fine-tuning) 的目的是在通用语义表示的基础上, 根据下游任务的特性进行适配。下游任务通常需要利用有标注数据集进行训练, 数据集使用 \mathbb{D} 进行表示, 每个样例由输入长度为 n 的文本序列 $x = x_1, x_2, \dots, x_n$ 和对应的标签 y 构成。

先将文本序列 x 输入 GPT 模型, 获得最后一层的最后一个词所对应的隐藏层输出 $\mathbf{h}_n^{(L)}$, 在此

基础上，通过全连接层变换结合 Softmax 函数，得到标签预测结果。

$$P(y|x_1, x_2, \dots, x_n) = \text{Softmax}(\mathbf{h}_n^{(L)} \mathbf{W}^y) \quad (2.16)$$

其中 $\mathbf{W}^y \in \mathbb{R}^{d \times k}$ 为全连接层参数， k 为标签个数。通过对整个标注数据集 \mathbb{D} 优化如下目标函数精调下游任务：

$$\mathcal{L}^{\text{FT}}(\mathbb{D}) = - \sum_{(x,y)} \log P(y|x_1, x_2, \dots, x_n) \quad (2.17)$$

在微调过程中，下游任务针对任务目标进行优化，很容易使得模型遗忘预训练阶段所学习的通用语义知识表示，从而损失模型的通用性和泛化能力，导致出现灾难性遗忘 (Catastrophic Forgetting) 问题。因此，通常采用混合预训练任务损失和下游微调损失的方法来缓解上述问题。在实际应用中，通常采用式 (2.13) 进行下游任务微调：

$$\mathcal{L} = \mathcal{L}^{\text{FT}}(\mathbb{D}) + \lambda \mathcal{L}^{\text{PT}}(\mathbb{D}) \quad (2.18)$$

其中 λ 的取值为 $[0, 1]$ ，用于调节预训练任务的损失占比。

2.2.3 预训练语言模型实践

HuggingFace 是一个开源自然语言处理软件库，其目标是通过提供一套全面的工具、库和模型，使自然语言处理技术对开发人员和研究人员更易于使用。HuggingFace 最著名的贡献之一是 transformers 库，基于此，研究人员可以快速部署训练好的模型，以及实现新的网络结构。除此之外，HuggingFace 提供了 Dataset 库，可以非常方便地下载自然语言处理研究中经常使用的基准数据集。本节将以构建 BERT 模型为例，介绍基于 HuggingFace 的 BERT 模型的构建和使用方法。

1. 数据集准备

常见的用于预训练语言模型的大规模数据集都可以在 Dataset 库中直接下载并加载。例如，如果使用维基百科的英文数据集，可以直接通过如下代码完成数据获取：

```

from datasets import concatenate_datasets, load_dataset

bookcorpus = load_dataset("bookcorpus", split="train")
wiki = load_dataset("wikipedia", "20220301.en", split="train")
# 仅保留'text'列
wiki = wiki.remove_columns([col for col in wiki.column_names if col != "text"])

dataset = concatenate_datasets([bookcorpus, wiki])

# 将数据集切分为90%用于训练, 10%用于测试
d = dataset.train_test_split(test_size=0.1)

```

接下来，将训练和测试数据分别保存在本地文件中，代码如下所示：

```

def dataset_to_text(dataset, output_filename="data.txt"):
    """ 将数据集文本保存到磁盘的通用函数中 """
    with open(output_filename, "w") as f:
        for t in dataset["text"]:
            print(t, file=f)

# 将训练集保存为train.txt
dataset_to_text(d["train"], "train.txt")
# 将测试集保存为test.txt
dataset_to_text(d["test"], "test.txt")

```

2. 训练词元分析器

BERT 采用 WordPiece 分词算法，根据训练数据中的词频决定是否将一个完整的词切分为多个词元。因此，需要先训练词元分析器（Tokenizer）。可以使用 transformers 库中的 BertWordPieceTokenizer 类来完成，代码如下所示：

```

special_tokens = [
    "[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]", "<S>", "<T>"
]
# 如果根据训练和测试两个集合训练词元分析器，则需要修改files
# files = ["train.txt", "test.txt"]
# 仅根据训练集合训练词元分析器
files = ["train.txt"]
# BERT中采用的默认词表大小为30522，可以随意修改
vocab_size = 30_522
# 最大序列长度，该值越小，训练速度越快
max_length = 512
# 是否将长样本截断
truncate_longer_samples = False

# 初始化WordPiece词元分析器
tokenizer = BertWordPieceTokenizer()
# 训练词元分析器
tokenizer.train(files=files, vocab_size=vocab_size, special_tokens=special_tokens)
# 允许截断达到最大512个词元
tokenizer.enable_truncation(max_length=max_length)

model_path = "pretrained-bert"

# 如果文件夹不存在，则先创建文件夹
if not os.path.isdir(model_path):
    os.mkdir(model_path)
# 保存词元分析器模型
tokenizer.save_model(model_path)
# 将一些词元分析器中的配置保存到配置文件，包括特殊词元、转换为小写、最大序列长度等
with open(os.path.join(model_path, "config.json"), "w") as f:
    tokenizer_cfg = {
        "do_lower_case": True,
        "unk_token": "[UNK]",
        "sep_token": "[SEP]",
        "pad_token": "[PAD]",
        "cls_token": "[CLS]",
        "mask_token": "[MASK]",
        "model_max_length": max_length,
        "max_len": max_length,
    }
    json.dump(tokenizer_cfg, f)

# 当词元分析器进行训练和配置时，将其装载到BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained(model_path)

```

3. 预处理数据集

在启动整个模型训练之前，还需要将预训练数据根据训练好的词元分析器进行处理。如果文档长度超过 512 个词元，就直接截断。数据处理代码如下所示：

```

def encode_with_truncation(examples):
    """ 使用词元分析对句子进行处理并截断的映射函数 (Mapping function) """
    return tokenizer(examples["text"], truncation=True, padding="max_length",
                     max_length=max_length, return_special_tokens_mask=True)

def encode_without_truncation(examples):
    """ 使用词元分析对句子进行处理且不截断的映射函数 (Mapping function) """
    return tokenizer(examples["text"], return_special_tokens_mask=True)

# 编码函数将依赖于truncate_longer_samples变量
encode = encode_with_truncation if truncate_longer_samples else encode_without_truncation
# 对训练数据集进行分词处理
train_dataset = d["train"].map(encode, batched=True)
# 对测试数据集进行分词处理
test_dataset = d["test"].map(encode, batched=True)
if truncate_longer_samples:
    # 移除其他列，将input_ids和attention_mask设置为PyTorch张量
    train_dataset.set_format(type="torch", columns=["input_ids", "attention_mask"])
    test_dataset.set_format(type="torch", columns=["input_ids", "attention_mask"])
else:
    # 移除其他列，将它们保留为Python列表
    test_dataset.set_format(columns=["input_ids", "attention_mask", "special_tokens_mask"])
    train_dataset.set_format(columns=["input_ids", "attention_mask", "special_tokens_mask"])

```

`truncate_longer_samples` 布尔变量控制用于对数据集进行词元处理的 `encode()` 回调函数。如果该变量设置为 `True`，则会截断超过最大序列长度 (`max_length`) 的句子。如果该变量设置为 `False`，则需要将没有截断的样本连接起来，并组合成固定长度的向量。

```

from itertools import chain
# 主要数据处理函数，拼接数据集中的所有文本并生成最大序列长度的块

def group_texts(examples):
    # 拼接所有文本
    concatenated_examples = {k: list(chain(*examples[k])) for k in examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    # 舍弃了剩余部分，如果模型支持填充而不是舍弃，则可以根据需要自定义这部分
    if total_length >= max_length:
        total_length = (total_length // max_length) * max_length
    # 按照最大长度分割成块
    result = {
        k: [t[i : i + max_length] for i in range(0, total_length, max_length)]
        for k, t in concatenated_examples.items()
    }
    return result

# 请注意，使用batched=True，此映射一次处理1000个文本
# 因此，group_texts会为这1000个文本组抛弃不足的部分
# 可以在这里调整batch_size，但较高的值可能会使预处理速度变慢
#
# 为了加速这一部分，使用了多进程处理
if not truncate_longer_samples:
    train_dataset = train_dataset.map(group_texts, batched=True,
                                      desc=f"Grouping texts in chunks of {max_length}")
    test_dataset = test_dataset.map(group_texts, batched=True,
                                    desc=f"Grouping texts in chunks of {max_length}")

# 将它们从列表转换为PyTorch张量
train_dataset.set_format("torch")
test_dataset.set_format("torch")

```

4. 模型训练

在构建处理好的预训练数据之后，就可以开始模型训练。代码如下所示：

```

# 使用配置文件初始化模型
model_config = BertConfig(vocab_size=vocab_size, max_position_embeddings=max_length)
model = BertForMaskedLM(config=model_config)

# 初始化数据整理器，随机屏蔽20%（默认为15%）的标记
# 用于掩盖语言建模（MLM）任务
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer, mlm=True, mlm_probability=0.2
)

training_args = TrainingArguments(
    output_dir=model_path,          # 输出目录，用于保存模型检查点
    evaluation_strategy="steps",    # 每隔`logging_steps`步进行一次评估
    overwrite_output_dir=True,
    num_train_epochs=10,            # 训练时的轮数，可以根据需要调整
    per_device_train_batch_size=10, # 训练批量大小，可以根据GPU内存容量将其设置得尽可能大
    gradient_accumulation_steps=8,  # 在更新权重之前累积梯度

    per_device_eval_batch_size=64,  # 评估批量大小
    logging_steps=1000,             # 每隔1000步进行一次评估，记录并保存模型检查点
    save_steps=1000,
    # load_best_model_at_end=True,   # 是否在训练结束时加载最佳模型（根据损失）
    # save_total_limit=3,           # 如果磁盘空间有限，则可以限制只保存3个模型权重
)

trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
)

# 训练模型
trainer.train()

```

训练完成后，可以得到如下输出结果：

```
[10135/79670 18:53:08 < 129:35:53, 0.15 it/s, Epoch 1.27/10]
```

Step	Training Loss	Validation Loss
1000	6.904000	6.558231
2000	6.498800	6.401168
3000	6.362600	6.277831
4000	6.251000	6.172856
5000	6.155800	6.071129
6000	6.052800	5.942584
7000	5.834900	5.546123
8000	5.537200	5.248503
9000	5.272700	4.934949
10000	4.915900	4.549236

5. 模型使用

可以针对不同应用需求使用训练好的模型，以句子补全为例的代码如下所示：

```
# 加载模型检查点
model = BertForMaskedLM.from_pretrained(os.path.join(model_path, "checkpoint-10000"))
# 加载词元分析器
tokenizer = BertTokenizerFast.from_pretrained(model_path)

fill_mask = pipeline("fill-mask", model=model, tokenizer=tokenizer)

# 进行预测
examples = [
    "Today's most trending hashtags on [MASK] is Donald Trump",
    "The [MASK] was cloudy yesterday, but today it's rainy.",
]
for example in examples:
    for prediction in fill_mask(example):
        print(f"{prediction['sequence']}, confidence: {prediction['score']}")
    print("="*50)
```

通过上述代码可以得到如下输出：

```
today's most trending hashtags on twitter is donald trump, confidence: 0.1027069091796875
today's most trending hashtags on monday is donald trump, confidence: 0.09271949529647827
today's most trending hashtags on tuesday is donald trump, confidence: 0.08099588006734848
today's most trending hashtags on facebook is donald trump, confidence: 0.04266013577580452
today's most trending hashtags on wednesday is donald trump, confidence: 0.04120611026883125
=====
the weather was cloudy yesterday, but today it's rainy., confidence: 0.04445931687951088
the day was cloudy yesterday, but today it's rainy., confidence: 0.037249673157930374
the morning was cloudy yesterday, but today it's rainy., confidence: 0.023775646463036537
the weekend was cloudy yesterday, but today it's rainy., confidence: 0.022554103285074234
the storm was cloudy yesterday, but today it's rainy., confidence: 0.019406016916036606
=====
```

2.3 大语言模型的结构

当前, 绝大多数大语言模型都采用类似 GPT 的架构, 使用基于 Transformer 结构构建的仅由解码器组成的网络结构, 采用自回归的方式构建语言模型, 但是在位置编码、层归一化位置、激活函数等细节上各有不同。文献^[13]介绍了 GPT-3 模型的训练过程, 包括模型架构、训练数据组成、训练过程及评估方法。由于 GPT-3 并没有开放源代码, 根据论文直接重现整个训练过程并不容易, 因此文献^[29]介绍了根据 GPT-3 的描述复现的过程, 构造并开源了系统 OPT (Open Pre-trained Transformer Language Models)。MetaAI 也仿照 GPT-3 的架构开源了 LLaMA 模型^[34], 公开评测结果及利用该模型进行有监督微调后的模型都有非常好的表现。GPT-3 模型之

后，OpenAI 就不再开源（也没有开源模型），因此并不清楚 ChatGPT 和 GPT-4 采用的模型架构。

本节将以 LLaMA 模型为例，介绍大语言模型架构在 Transformer 原始结构上的改进，并介绍 Transformer 结构中空间和时间占比最大的注意力机制的优化方法。

2.3.1 LLaMA 的模型结构

文献 [34] 介绍了 LLaMA 采用的 Transformer 结构和细节，与 2.1 节介绍的 Transformer 结构的不同之处为采用了前置层归一化 (Pre-normalization) 方法并使用 RMSNorm 归一化函数 (Root Mean Square Normalizing Function)，激活函数更换为 SwiGLU，使用了旋转位置嵌入 (Rotary Positional Embeddings, RoPE)，使用的 Transformer 结构与 GPT-2 类似，如图 2.4 所示。

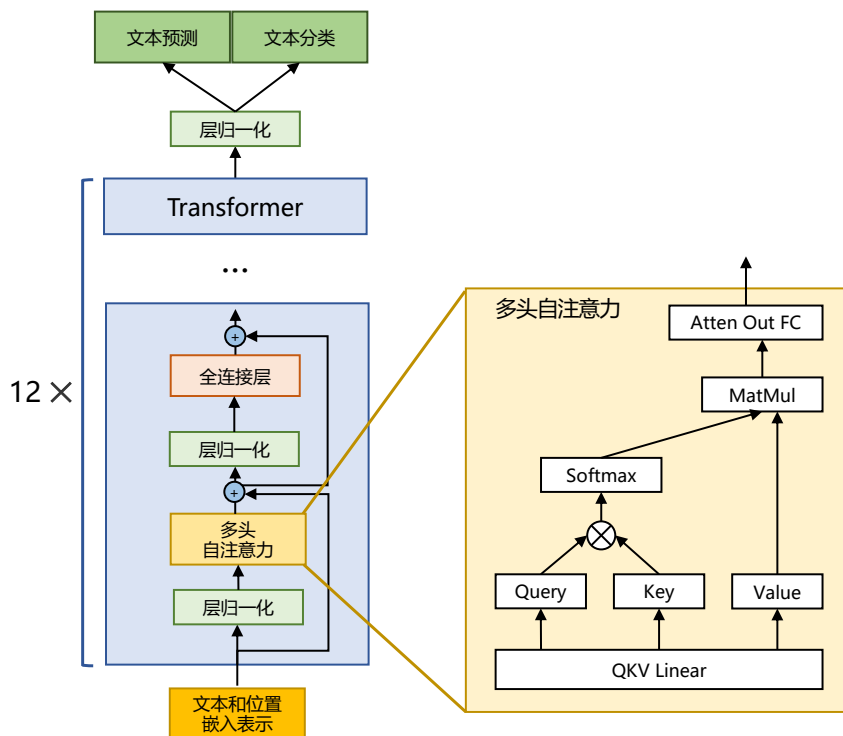


图 2.4 GPT-2 的模型结构

接下来，分别介绍 RMSNorm 归一化函数、SwiGLU 激活函数和 RoPE 的具体内容和实现。

1. RMSNorm 归一化函数

为了使模型训练过程更加稳定，GPT-2 相较于 GPT 引入了前置层归一化方法，将第一个层归一化移动到多头自注意力层之前，将第二个层归一化移动到全连接层之前。同时，残差连接的位

置调整到多头自注意力层与全连接层之后。层归一化中也采用了 RMSNorm 归一化函数^[45]。针对输入向量 \mathbf{a} ，RMSNorm 函数的计算公式如下：

$$\text{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n} \sum_{i=1}^n a_i^2} \quad (2.19)$$

$$\bar{a}_i = \frac{a_i}{\text{RMS}(\mathbf{a})} \quad (2.20)$$

此外，RMSNorm 还可以引入可学习的缩放因子 g_i 和偏移参数 b_i ，从而得到 $\bar{a}_i = \frac{a_i}{\text{RMS}(\mathbf{a})} g_i + b_i$ 。RMSNorm 在 HuggingFace transformers 库中的代码实现如下所示：

```
class LlamaRMSNorm(nn.Module):
    def __init__(self, hidden_size, eps=1e-6):
        """
        LlamaRMSNorm 等同于 T5LayerNorm
        """
        super().__init__()
        self.weight = nn.Parameter(torch.ones(hidden_size))
        self.variance_epsilon = eps # eps防止取倒数之后分母为0

    def forward(self, hidden_states):
        input_dtype = hidden_states.dtype
        variance = hidden_states.to(torch.float32).pow(2).mean(-1, keepdim=True)
        hidden_states = hidden_states * torch.rsqrt(variance + self.variance_epsilon)
        # weight是末尾乘的可训练参数，即g_i
        return (self.weight * hidden_states).to(input_dtype)
```

2. SwiGLU 激活函数

SwiGLU^[46] 激活函数是 Shazeer 在文献 [46] 中提出的，在 PaLM^[14] 等模型中进行了广泛应用，并且取得了不错的效果，相较于 ReLU 函数在大部分评测中都有不少提升。在 LLaMA 中，全连接层使用带有 SwiGLU 激活函数的位置感知前馈网络的计算公式如下：

$$\text{FFN}_{\text{SwiGLU}}(\mathbf{x}, \mathbf{W}, \mathbf{V}, \mathbf{W}_2) = \text{SwiGLU}(\mathbf{x}, \mathbf{W}, \mathbf{V}) \mathbf{W}_2 \quad (2.21)$$

$$\text{SwiGLU}(\mathbf{x}, \mathbf{W}, \mathbf{V}) = \text{Swish}_\beta(\mathbf{x}\mathbf{W}) \otimes \mathbf{x}\mathbf{V} \quad (2.22)$$

$$\text{Swish}_\beta(\mathbf{x}) = \mathbf{x}\sigma(\beta\mathbf{x}) \quad (2.23)$$

其中， $\sigma(x)$ 是 Sigmoid 函数。图2.5 给出了 Swish 激活函数在参数 β 取不同值时的形状。可以看到，当 β 趋近于 0 时，Swish 函数趋近于线性函数 $y = x$ ；当 β 趋近于无穷大时，Swish 函数趋近

于 ReLU 函数；当 β 取值为 1 时，Swish 函数是光滑且非单调的。在 HuggingFace 的 transformers 库中 Swish 函数被 SiLU 函数^[47] 代替。

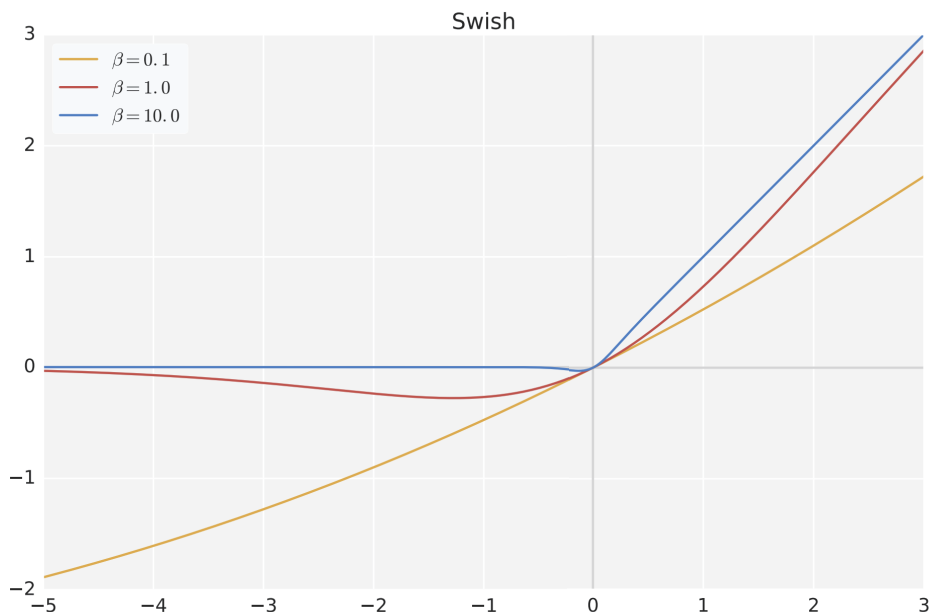


图 2.5 Swish 激活函数在参数 β 取不同值时的形状

3. RoPE

在位置编码上，使用旋转位置嵌入^[48] 代替原有的绝对位置编码。RoPE 借助复数的思想，出发点是通过绝对位置编码的方式实现相对位置编码。其目标是通过下述运算给 \mathbf{q}, \mathbf{k} 添加绝对位置信息：

$$\tilde{\mathbf{q}}_m = f(\mathbf{q}, m), \tilde{\mathbf{k}}_n = f(\mathbf{k}, n) \quad (2.24)$$

详细的证明和求解过程可以参考文献 [48]，最终可以得到二维情况下用复数表示的 RoPE：

$$f(\mathbf{q}, m) = R_f(\mathbf{q}, m)e^{i\Theta_f(\mathbf{q}, m)} = \|\mathbf{q}\|e^{i(\Theta(\mathbf{q})+m\theta)} = \mathbf{q}e^{im\theta} \quad (2.25)$$

根据复数乘法的几何意义，上述变换实际上是对应向量旋转，所以位置向量称为“旋转式位置编

码”。还可以使用矩阵形式表示：

$$f(\mathbf{q}, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \end{pmatrix} \quad (2.26)$$

根据内积满足线性叠加的性质，任意偶数维的 RoPE 都可以表示为二维情形的拼接，即

$$f(\mathbf{q}, m) = \underbrace{\begin{pmatrix} \cos m\theta_0 & -\sin m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_0 & \cos m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_1 & -\sin m\theta_1 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_1 & \cos m\theta_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2-1} & -\sin m\theta_{d/2-1} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2-1} & \cos m\theta_{d/2-1} \end{pmatrix}}_{\mathbf{R}_d} \begin{pmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \\ \mathbf{q}_2 \\ \mathbf{q}_3 \\ \vdots \\ \mathbf{q}_{d-2} \\ \mathbf{q}_{d-1} \end{pmatrix} \quad (2.27)$$

由于上述矩阵 \mathbf{R}_d 具有稀疏性，因此可以使用逐位相乘 \otimes 操作进一步提高计算速度。RoPE 在 HuggingFace transformers 库中的代码实现如下所示：

```

class LlamaRotaryEmbedding(torch.nn.Module):
    def __init__(self, dim, max_position_embeddings=2048, base=10000, device=None):
        super().__init__()
        inv_freq = 1.0 / (base ** (torch.arange(0, dim, 2).float().to(device) / dim))
        self.register_buffer("inv_freq", inv_freq)

        # 在这里构建，以便使`torch.jit.trace`正常工作
        self.max_seq_len_cached = max_position_embeddings
        t = torch.arange(self.max_seq_len_cached, device=self.inv_freq.device,
                        dtype=self.inv_freq.dtype)
        freqs = torch.einsum("i,j->ij", t, self.inv_freq)
        # 这里使用了与论文不同的排列，以便获得相同的计算结果
        emb = torch.cat((freqs, freqs), dim=-1)
        dtype = torch.get_default_dtype()
        self.register_buffer("cos_cached", emb.cos()[None, None, :, :].to(dtype), persistent=False)
        self.register_buffer("sin_cached", emb.sin()[None, None, :, :].to(dtype), persistent=False)

    def forward(self, x, seq_len=None):
        # x: [bs, num_attention_heads, seq_len, head_size]
        # 在`__init__`中构建了sin/cos, 这个`if`块不太可能被执行
        # 保留这里的逻辑
        if seq_len > self.max_seq_len_cached:
            self.max_seq_len_cached = seq_len
            t = torch.arange(self.max_seq_len_cached, device=x.device, dtype=self.inv_freq.dtype)
            freqs = torch.einsum("i,j->ij", t, self.inv_freq)
            # 这里使用了与论文不同的排列，以便获得相同的计算结果
            emb = torch.cat((freqs, freqs), dim=-1).to(x.device)
            self.register_buffer("cos_cached", emb.cos()[None, None, :, :].to(x.dtype),
                                persistent=False)
            self.register_buffer("sin_cached", emb.sin()[None, None, :, :].to(x.dtype),
                                persistent=False)

        return (
            self.cos_cached[:, :, :seq_len, ...].to(dtype=x.dtype),
            self.sin_cached[:, :, :seq_len, ...].to(dtype=x.dtype),
        )

    def rotate_half(x):
        """ 将输入的一半隐藏维度进行旋转 """
        x1 = x[..., : x.shape[-1] // 2]
        x2 = x[..., x.shape[-1] // 2 :]
        return torch.cat((-x2, x1), dim=-1)

    def apply_rotary_pos_emb(q, k, cos, sin, position_ids):
        # cos和sin的前两个维度始终为1, 因此可以对它们进行`squeeze`操作
        cos = cos.squeeze(1).squeeze(0) # [seq_len, dim]
        sin = sin.squeeze(1).squeeze(0) # [seq_len, dim]
        cos = cos[position_ids].unsqueeze(1) # [bs, 1, seq_len, dim]
        sin = sin[position_ids].unsqueeze(1) # [bs, 1, seq_len, dim]
        q_embed = (q * cos) + (rotate_half(q) * sin)
        k_embed = (k * cos) + (rotate_half(k) * sin)
        return q_embed, k_embed

```

4. 模型整体框架

基于上述模型和网络结构可以实现解码器层，根据自回归方式利用训练数据进行模型训练的过程与 2.2.3 节介绍的过程基本一致。不同规模的 LLaMA 模型使用的超参数如表2.1 所示。由于大语言模型的参数量非常大，并且需要大量的数据进行训练，因此仅利用单个 GPU 很难完成训练，需要依赖分布式模型训练框架（第 4 章将详细介绍相关内容）。

表 2.1 不同规模的 LLaMA 模型使用的超参数^[34]

参数规模	层数	自注意力头数	嵌入表示维度	学习率	全局批次大小	训练词元数量（个）
6.7B ^①	32	32	4096	3.0e-4	400 万	1.0 万亿
13.0B	40	40	5120	3.0e-4	400 万	1.0 万亿
32.5B	60	52	6656	1.5e-4	400 万	1.4 万亿
65.2B	80	64	8192	1.5e-4	400 万	1.4 万亿

HuggingFace transformers 库中 LLaMA 解码器的整体代码实现如下所示：

^① B，即 Billion，表示十亿个。