从头开始实现一个简化的版本的GPT模型

发表评论 / ChatGPT, GPT, OpenAI

如果你想从头开始实现一个简化的版本的GPT模型，而不依赖于现成的GPT-2模型库，你可以采用PyTorch这样的深度学习框架。下面是一个非常基础的例子，展示了如何实现一个简化的Transformer模型架构，这是构建GPT模型的基础。

这个例子将不会覆盖GPT-2的所有复杂性和特性，但可以提供一个起点，帮助你理解如何从头开始构建类似GPT的模型。

1. 基础Transformer块

首先，我们定义一个基础的Transformer块，它是构成GPT模型的基本单元。这个块将包括自注意力机制和前馈神经网络。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import re
import jieba

from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence


class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads):
        super(SelfAttention, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert self.head_dim * heads == embed_size, "Embedding size needs to be divisible by

        self.values = nn.Linear(embed_size, embed_size, bias=False)
        self.keys = nn.Linear(embed_size, embed_size, bias=False)
        self.queries = nn.Linear(embed_size, embed_size, bias=False)
        self.fc_out = nn.Linear(heads * self.head_dim, embed_size)

    def forward(self, value, key, query, mask):
        N = query.shape&#91;0]
        value_len, key_len, query_len = value.shape&#91;1], key.shape&#91;1], query.shape&#91

        # Split the embedding into `self.heads` pieces
        values = self.values(value).view(N, value_len, self.heads, self.head_dim)
        keys = self.keys(key).view(N, key_len, self.heads, self.head_dim)
        queries = self.queries(query).view(N, query_len, self.heads, self.head_dim)

        # Transpose for attention dot product: from &#91;N, value_len, self.heads, self.head_
        # to &#91;N, self.heads, value_len, self.head_dim] to match the shape for `torch.eins
        values = values.transpose(1, 2)
        keys = keys.transpose(1, 2)
        queries = queries.transpose(1, 2)

        # Attention mechanism
        energy = torch.einsum("nqhd,nkhd->nhqk", &#91;queries, keys])

        if mask is not None:
```

```
            energy = energy.masked_fill(mask == 0, float("-1e20"))

        attention = torch.softmax(energy / (self.embed_size ** (1 / 2)), dim=3)

        # 重塑前进行张量乘法，然后重塑回 &#91;batch_size, seq_len, heads * head_dim]
        out = torch.einsum("nhql,nlhd->nqhd", &#91;attention, values]).reshape(
            N, query_len, self.heads * self.head_dim
        )

        out = self.fc_out(out)
        return out

class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TransformerBlock, self).__init__()
        self.attention = SelfAttention(embed_size, heads)
        self.norm1 = nn.LayerNorm(embed_size)
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion * embed_size),
            nn.ReLU(),
            nn.Linear(forward_expansion * embed_size, embed_size),
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, value, key, query, mask):
        attention = self.attention(value, key, query, mask)

        # Add skip connection, followed by layer normalization
        x = self.norm1(attention + query)
        forward = self.feed_forward(x)
        out = self.norm2(forward + x)  # Add skip connection, followed by layer normalizatior
        return out

```

2. 简化版的GPT模型

接下来，我们定义一个简化版的`GPT`模型，它利用上面定义的`Transformer`块。

```
class GPT(nn.Module):
    def __init__(self, embed_size, num_layers, heads, forward_expansion, dropout, vocab_size,
        super(GPT, self).__init__()
        self.embed_size = embed_size
        self.transformer_blocks = nn.ModuleList(
            &#91;
                TransformerBlock(
                    embed_size,
                    heads,
                    dropout=dropout,
                    forward_expansion=forward_expansion,
                )
                for _ in range(num_layers)
            ]
        )

        self.word_embedding = nn.Embedding(vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

    def forward(self, x, mask):
        N, seq_length = x.shape
        print(f"Input shape: {x.shape}")  # 打印输入形状
```

```

        positions = torch.arange(0, seq_length).expand(N, seq_length).to(x.device)
        out = self.word_embedding(x) + self.position_embedding(positions)
        print(f"After embedding and position shape: {out.shape}")  # 打印嵌入和位置编码后的形状

        for layer in self.transformer_blocks:
            out = layer(out, out, out, mask)
            print(f"After transformer block shape: {out.shape}")  # 打印经过每个Transformer块后

        return out

```

继续前面的简化版GPT模型实现，下面提供一个基本的训练框架。这个例子将展示如何准备数据、定义损失函数、选择优化器，并执行训练循环。请注意，这是一个高度简化的例子，旨在演示基本概念。

3. 准备数据

假设你已经有了一个文本数据集，并且你已经进行了预处理（例如，分词和转换为词汇索引）。为了简单起见，这里不展示数据预处理的代码。我们将直接从创建数据加载器开始。

```
def clean_text_mixed_with_symbols(text):
    # 保留中文、英文字符、数字和常见的标点符号
    # 注意：根据需要，你可以在这里添加或删除特定的符号
    text = re.sub(r'&#91;^\u4e00-\u9fffA-Za-z0-9，。！？、；：“”‘’（）《》【】—…]+', ' ', text)
    return text.strip()

def preprocess_text_mixed_with_symbols(text):
    text = clean_text_mixed_with_symbols(text)
    tokens = &#91;]
    for token in jieba.cut(text, cut_all=False):
        token = token.strip()
        if token:
            tokens.append(token)
    return tokens

def load_and_preprocess_data(file_paths):
    # 这里简化处理，具体实现依据你的需求定
    texts = &#91;]
    for file_path in file_paths:
        with open(file_path, 'r', encoding='utf-8') as file:
            text = file.read()
            # 添加文本清洗和预处理逻辑
            processed_text = preprocess_text_mixed_with_symbols(text)
            texts.append(processed_text)
    return texts


class TextDataset(Dataset):
    def __init__(self, indexed_texts, vocab_size):
        self.texts = &#91;torch.tensor(text, dtype=torch.long) for text in indexed_texts]  #
        self.vocab_size = vocab_size

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        return self.texts&#91;idx]

    def collate_fn(self, batch):
        input_ids = &#91;item&#91;:-1] for item in batch]
        target_ids = &#91;item&#91;1:] for item in batch]
        input_ids_padded = pad_sequence(input_ids, batch_first=True, padding_value=0)
```

```
        target_ids_padded = pad_sequence(target_ids, batch_first=True, padding_value=0)
        return input_ids_padded, target_ids_padded


def build_vocab(texts):
    vocab = set(token for text in texts for token in text)
    vocab_to_index = {word: i for i, word in enumerate(vocab, start=1)}  # 从1开始编号
    return vocab_to_index


def index_text(text, vocab_to_index):
    return &#91;vocab_to_index&#91;token] for token in text if token in vocab_to_index]

```

4. 定义模型、损失函数和优化器

```
# 实例化模型
model = GPT(
    embed_size=embed_size,
    num_layers=num_layers,
    heads=heads,
    forward_expansion=forward_expansion,
    dropout=dropout,
    vocab_size=vocab_size,
    max_length=max_length
)

loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)

```

5. 训练循环

最后，我们执行训练循环，每个批次处理数据，计算损失，并更新模型的权重。

```
def train(model, dataloader, loss_fn, optimizer, device, epochs):
    model.train()
    model.to(device)

    for epoch in range(epochs):
        for batch_idx, (input_ids, target_ids) in enumerate(dataloader):
            input_ids = input_ids.to(device)
            target_ids = target_ids.to(device)

            # 前向传播
            predictions = model(input_ids, mask=None) # 这里简化处理，没有使用mask
            predictions = predictions.view(-1, predictions.size(-1))
            target_ids = target_ids.view(-1)

            # 计算损失
            loss = loss_fn(predictions, target_ids)

            # 反向传播和优化
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if batch_idx % 100 == 0:
                print(f"Epoch {epoch} Batch {batch_idx} Loss {loss.item()}")

```

这段代码展示了如何设置和执行模型的训练过程。请注意，这只是一个起点，真实世界的应用可能需要更复杂的数据处理、模型调参、正则化策略、以及训练过程监控。此外，为了处理大规模数据集和模型，可能还需要考虑分布式训练和模型并行化。

6. 完整的训练代码

代码如下：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import re
import jieba

from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence


class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads):
        super(SelfAttention, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert self.head_dim * heads == embed_size, "Embedding size needs to be divisible by

        self.values = nn.Linear(embed_size, embed_size, bias=False)
        self.keys = nn.Linear(embed_size, embed_size, bias=False)
        self.queries = nn.Linear(embed_size, embed_size, bias=False)
        self.fc_out = nn.Linear(heads * self.head_dim, embed_size)

    def forward(self, value, key, query, mask):
        N = query.shape&#91;0]
        value_len, key_len, query_len = value.shape&#91;1], key.shape&#91;1], query.shape&#9

        # Split the embedding into `self.heads` pieces
        values = self.values(value).view(N, value_len, self.heads, self.head_dim)
        keys = self.keys(key).view(N, key_len, self.heads, self.head_dim)
        queries = self.queries(query).view(N, query_len, self.heads, self.head_dim)

        # Transpose for attention dot product: from &#91;N, value_len, self.heads, self.head
        # to &#91;N, self.heads, value_len, self.head_dim] to match the shape for `torch.ein
        values = values.transpose(1, 2)
        keys = keys.transpose(1, 2)
        queries = queries.transpose(1, 2)

        # Attention mechanism
        energy = torch.einsum("nqhd,nkhd-&gt;nhqk", &#91;queries, keys])

        if mask is not None:
            energy = energy.masked_fill(mask == 0, float("-1e20"))

        attention = torch.softmax(energy / (self.embed_size ** (1 / 2)), dim=3)

        # 重塑前进行张量乘法，然后重塑回 &#91;batch_size, seq_len, heads * head_dim]
        out = torch.einsum("nhql,nlhd-&gt;nqhd", &#91;attention, values]).reshape(
            N, query_len, self.heads * self.head_dim
        )

        out = self.fc_out(out)
```

```
        return out

class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TransformerBlock, self).__init__()
        self.attention = SelfAttention(embed_size, heads)
        self.norm1 = nn.LayerNorm(embed_size)
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion * embed_size),
            nn.ReLU(),
            nn.Linear(forward_expansion * embed_size, embed_size),
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, value, key, query, mask):
        attention = self.attention(value, key, query, mask)

        # Add skip connection, followed by layer normalization
        x = self.norm1(attention + query)
        forward = self.feed_forward(x)
        out = self.norm2(forward + x)  # Add skip connection, followed by layer normalizatio
        return out

class GPT(nn.Module):
    def __init__(self, embed_size, num_layers, heads, forward_expansion, dropout, vocab_size
        super(GPT, self).__init__()
        self.embed_size = embed_size
        self.transformer_blocks = nn.ModuleList(
            &#91;
                TransformerBlock(
                    embed_size,
                    heads,
                    dropout=dropout,
                    forward_expansion=forward_expansion,
                )
                for _ in range(num_layers)
            ]
        )

        self.word_embedding = nn.Embedding(vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

    def forward(self, x, mask):
        N, seq_length = x.shape
        print(f"Input shape: {x.shape}")  # 打印输入形状

        positions = torch.arange(0, seq_length).expand(N, seq_length).to(x.device)
        out = self.word_embedding(x) + self.position_embedding(positions)
        print(f"After embedding and position shape: {out.shape}")  # 打印嵌入和位置编码后的形状

        for layer in self.transformer_blocks:
            out = layer(out, out, out, mask)
            print(f"After transformer block shape: {out.shape}")  # 打印经过每个Transformer块后

        return out

def clean_text_mixed_with_symbols(text):
    # 保留中文、英文字符、数字和常见的标点符号
    # 注意：根据需要，你可以在这里添加或删除特定的符号
    text = re.sub(r'&#91;^\u4e00-\u9fffA-Za-z0-9，。！？、；：“”‘’（）《》【】—…]+', ' ', text)
    return text.strip()

```

```
def preprocess_text_mixed_with_symbols(text):
    text = clean_text_mixed_with_symbols(text)
    tokens = &#91;]
    for token in jieba.cut(text, cut_all=False):
        token = token.strip()
        if token:
            tokens.append(token)
    return tokens

def load_and_preprocess_data(file_paths):
    # 这里简化处理，具体实现依据你的需求定
    texts = &#91;]
    for file_path in file_paths:
        with open(file_path, 'r', encoding='utf-8') as file:
            text = file.read()
            # 添加文本清洗和预处理逻辑
            processed_text = preprocess_text_mixed_with_symbols(text)
            texts.append(processed_text)
    return texts


class TextDataset(Dataset):
    def __init__(self, indexed_texts, vocab_size):
        self.texts = &#91;torch.tensor(text, dtype=torch.long) for text in indexed_texts]  #
        self.vocab_size = vocab_size

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        return self.texts&#91;idx]

    def collate_fn(self, batch):
        input_ids = &#91;item&#91;:-1] for item in batch]
        target_ids = &#91;item&#91;1:] for item in batch]
        input_ids_padded = pad_sequence(input_ids, batch_first=True, padding_value=0)
        target_ids_padded = pad_sequence(target_ids, batch_first=True, padding_value=0)
        return input_ids_padded, target_ids_padded


def build_vocab(texts):
    vocab = set(token for text in texts for token in text)
    vocab_to_index = {word: i for i, word in enumerate(vocab, start=1)}  # 从1开始编号
    return vocab_to_index


def index_text(text, vocab_to_index):
    return &#91;vocab_to_index&#91;token] for token in text if token in vocab_to_index]

def train(model, dataloader, loss_fn, optimizer, device, epochs):
    model.train()
    model.to(device)

    for epoch in range(epochs):
        for batch_idx, (input_ids, target_ids) in enumerate(dataloader):
            input_ids = input_ids.to(device)
            target_ids = target_ids.to(device)

            # 前向传播
            predictions = model(input_ids, mask=None) # 这里简化处理，没有使用mask
            predictions = predictions.view(-1, predictions.size(-1))
            target_ids = target_ids.view(-1)

            # 计算损失
            loss = loss_fn(predictions, target_ids)
```

```python

            # 反向传播和优化
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if batch_idx % 100 == 0:
                print(f"Epoch {epoch} Batch {batch_idx} Loss {loss.item()}")


# 模型参数
vocab_size = 10000  # 假设的词汇表大小
embed_size = 256
max_length = 100
num_layers = 6
heads = 8
forward_expansion = 4
dropout = 0.1

# 假设你的文本文件路径
#file_paths = &#91;'path/to/your/text1.txt', 'path/to/your/text2.txt']
#texts = load_and_preprocess_data(file_paths)

# 假设文本包含中文、英文和常用符号
text = "1977年，三位数学家Rivest、Shamir 和 Adleman 设计了一种算法，可以实现非对称加密。这种算法用他们三

# 预处理文本
texts = preprocess_text_mixed_with_symbols(text)
# 输出分词结果
print(texts)

# 假设`texts`是分词后的文本列表
vocab_to_index = build_vocab(texts)
indexed_texts = &#91;index_text(text, vocab_to_index) for text in texts]

vocab_size=len(vocab_to_index) + 1
# 现在`texts`应该是索引化后的文本列表
dataset = TextDataset(indexed_texts, vocab_size=len(vocab_to_index) + 1)  # +1因为从1开始编号
dataloader = DataLoader(dataset, batch_size=32, shuffle=True, collate_fn=dataset.collate_fn)

# 实例化模型
model = GPT(
    embed_size=embed_size,
    num_layers=num_layers,
    heads=heads,
    forward_expansion=forward_expansion,
    dropout=dropout,
    vocab_size=vocab_size,
    max_length=max_length
)

loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)

epochs = 1

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
train(model, dataloader, loss_fn, optimizer, device, epochs)

# 保存模型参数
# model_path = "gpt_simple_model.pth"
# torch.save(model.state_dict(), model_path)

# 如果要保存整个模型（包括模型结构），可以使用以下方式
model_path = "gpt_simple_model_full.pth"
```

```
torch.save(model, model_path)

```

代码提供了一个使用PyTorch实现类似GPT模型的全面示例，这个示例涵盖了多个关键方面，包括自注意力层的定义、变压器块、整体GPT模型、文本数据的预处理（包括混合语言内容的文本清理和使用Jieba进行分词），以及最后的模型训练、自定义数据集和数据加载器的使用。

以下是一些建议和澄清点，以确保代码按预期工作，并遵循最佳实践：

1. **自注意力和变压器块实现**：您的自注意力和变压器块实现看起来很好。它遵循了构建基于变压器模型的标准方法，包括将输入分割成多个头、应用自注意力，然后使用前馈网络。
2. **模型训练循环**：训练循环包括深度学习模型典型训练过程的基本步骤。它通过模型处理输入、计算损失、执行反向传播和更新模型的权重。您还包括了设备兼容性，以便在GPU上运行模型（如果可用），这对于训练效率至关重要。
3. **文本预处理和分词**：您包含了清理文本和分词的功能，这对于NLP任务至关重要。使用Jieba进行分词适用于处理中文文本，您的正则表达式清理混合语言文本涵盖了广泛的字符。
4. **数据处理和数据加载器**：您定义了一个自定义的Dataset类，并使用PyTorch的DataLoader进行批处理和填充。这是处理NLP任务中可变长度序列的好方法。
5. **潜在改进**：
    - **数据预处理中的错误处理**：确保您的文件读取和文本预处理能够优雅地处理错误，尤其是对于可能不存在或有编码问题的文件。
    - **模型中的掩码使用**：您的评论提到了为了简化而没有使用掩码。实际上，特别是对于长度不同的序列，掩码对于通知模型哪些输入部分是填充且不应该被关注是至关重要的。
    - **词汇表构建**：构建词汇表和索引文本的过程假设所有文本都被分词成一个平面列表。实际上，您可能有多个文档或句子，您可能希望分别处理它们或保持句子边界。
    - **保存模型**：您展示了两种保存模型的方式；仅保存模型参数（state_dict）更节省空间，是大多数用例推荐的方法。保存整个模型虽然方便，但如果需要在不同环境中加载模型，可能会导致问题。

在运行代码之前，请确保调整文件路径，并根据您的具体需求可能扩展预处理和数据集处理。此外，考虑尝试不同的模型超参数（如embed_size、num_layers、heads等）和训练配置，以找到适合您任务的最佳设置。