part 1 学习的路线

训练一个 **Transformer** 模型进行 **自然语言处理（NLP）** 需要掌握几个关键知识点。下面是一个 **循序渐进的学习路径**，帮助你高效学习并训练自己的 Transformer 模型。

第一阶段：打好基础

在深入 Transformer 之前，建议先掌握以下基础知识：

1. 数学基础

- 线性代数（矩阵运算、特征向量）
- 概率统计（概率分布、最大似然估计）
- 信息论（熵、交叉熵损失）
- 微积分（梯度计算、链式法则）

📖 **推荐学习资源**：

- 《深度学习数学基础》(Michael Nielsen)
- 3Blue1Brown 线性代数课程 (YouTube)

2. 深度学习基础

- **神经网络**（MLP、CNN、RNN）
- **优化方法**（SGD、Adam）
- **损失函数**（交叉熵、MSE）

📖 **推荐学习资源**：

- 《Deep Learning》（Ian Goodfellow）
- Coursera 深度学习课程（Andrew Ng）

3. 自然语言处理（NLP）

- 词向量（Word2Vec、GloVe、FastText）
- 传统 NLP 技术（TF-IDF、n-gram、LSTM）
- 句子和文本表示（BERT、GPT）

📖 **推荐学习资源**：

- 《Speech and Language Processing》（Jurafsky & Martin）
- Stanford NLP 课程 (CS224N)

第二阶段：学习 Transformer 及其变体

Transformer 模型是 NLP 的核心，以下是关键概念：

1. Transformer 结构

- Self-Attention（自注意力机制）
- Multi-Head Attention（多头注意力）
- Position Encoding（位置编码）
- Feed Forward Network（前馈神经网络）

📖 **推荐学习资源**：

- **论文**：《Attention Is All You Need》 (Vaswani et al.)
- **课程**：Stanford CS25 (Transformer 课程)
- **博客**：Jay Alammar - The Illustrated Transformer

2. Transformer 变体

- **BERT（双向编码器）**：适合文本分类、问答、命名实体识别等任务
- **GPT（生成式预训练模型）**：适合文本生成、对话系统
- **T5 & BART（文本生成与翻译）**

📖 **推荐学习资源**：

- BERT 论文：《BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding》
- GPT-3 论文：《Language Models are Few-Shot Learners》

第三阶段：实践 Transformer 训练

🔥 **目标**：自己动手训练 Transformer 模型，掌握 PyTorch & TensorFlow。

1. 使用 Hugging Face

Hugging Face 提供了大量预训练的 Transformer 模型，可以直接用于 NLP 任务：

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer
import torch

model_name = "bert-base-uncased"
model = AutoModelForSequenceClassification.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)

text = "I love deep learning!"
inputs = tokenizer(text, return_tensors="pt")
outputs = model(**inputs)

print(outputs.logits)  # 分类输出
```

📖 **推荐学习资源**：

- Hugging Face 官方教程
- Hugging Face 文档

2. 从零训练 Transformer

如果你想自己训练一个 Transformer（如 GPT/BERT），需要：

- 数据集（如 IMDb、WikiText、GLUE 任务）
- 训练框架（PyTorch / TensorFlow）
- GPU 资源（如 Google Colab、Kaggle、AWS）

示例：**使用 PyTorch 训练 GPT**

```python
from transformers import GPT2LMHeadModel, GPT2Tokenizer, Trainer, TrainingArguments

model_name = "gpt2"
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)

# 准备数据集
train_texts = ["Hello, how are you?", "I love programming in Python!"]
train_encodings = tokenizer(train_texts, truncation=True, padding=True, return_tensors="pt")

# 训练参数
training_args = TrainingArguments(output_dir="./results", per_device_train_batch_size=2)
trainer = Trainer(model=model, args=training_args, train_dataset=train_encodings)

trainer.train()
```

📖 **推荐学习资源**：

- Hugging Face Trainer 训练指南
- PyTorch 官方教程

第四阶段：优化 Transformer 训练

当你掌握基础训练后，可以学习一些优化方法：

1. 高效训练技巧

- 混合精度训练（FP16 / BF16）
- 分布式训练（DDP / FSDP）
- 数据增强（EDA, Back Translation）

2. 超参数调优

- 学习率调度（Linear Warmup, Cosine Annealing）
- Batch Size & Optimizer 调整

3. 模型压缩

- 量化（Quantization）

- 剪枝（Pruning）
- 知识蒸馏（Distillation）

📖 **推荐学习资源**：

- DeepSpeed / ZeRO 训练优化：微软 DeepSpeed
- 模型压缩：Hugging Face Optimum

第五阶段：应用 Transformer

最后，你可以尝试在实际任务中应用 Transformer：

1. 文本分类

- 任务：情感分析、垃圾邮件检测
- 预训练模型：BERT、RoBERTa

2. 文本生成

- 任务：对话机器人、自动写作
- 预训练模型：GPT-2、GPT-3

3. 机器翻译

- 任务：英文→法文、英文→中文
- 预训练模型：T5、mBART

4. 问答系统

- 任务：知识问答、阅读理解
- 预训练模型：BERT、T5

📖 **推荐学习资源**：

- Kaggle NLP 竞赛
- Hugging Face Spaces

总结

✅ **第一阶段**：掌握数学 & 深度学习基础
✅ **第二阶段**：深入理解 Transformer 及其变体
✅ **第三阶段**：动手训练 Transformer（Hugging Face / PyTorch）
✅ **第四阶段**：优化 Transformer 训练
✅ **第五阶段**：应用 Transformer 解决实际问题

如果你想要针对某个任务训练 Transformer 模型，可以告诉我，我可以帮你写更具体的代码示例！ 🚀

part 2 完整的步骤

好的，我将更详细地解释每一个步骤，帮助你深入理解如何使用 Transformer 模型（例如 BERT）对 IMDb 数据集进行情感分类任务。

步骤 1：数据集下载与加载

首先，你需要下载 IMDb 数据集。IMDb 数据集包含 50,000 条影评，其中 25,000 条用于训练，25,000 条用于测试。每条影评都会被标记为正面或负面情感。

如果你使用 Hugging Face Datasets，可以非常方便地加载数据。你可以通过以下代码加载 IMDb 数据集：

```
from datasets import load_dataset

# 加载 IMDb 数据集，包含 'train' 和 'test' 两部分
dataset = load_dataset("imdb")
```

这样， `dataset` 变量将包含一个字典，包含两个数据集： `train` （用于训练）和 `test` （用于测试）。

```
# 查看数据集结构
print(dataset['train'][0])
```

输出示例：

```
{'text': "This movie was terrible. I hated it.", 'label': 0}
```

`text` 是影评文本， `label` 是标签， `0` 表示负面情感， `1` 表示正面情感。

步骤 2：数据预处理

在开始训练之前，我们需要对文本数据进行预处理，主要步骤包括：

1. **分词**：将文本拆分成一个个单独的词语或子词。BERT 使用的是子词级别的分词方法。
2. **填充和截断**：为了确保所有输入的序列长度一致，文本必须进行填充和截断。
3. **转换成模型的输入格式**：BERT 模型需要输入包含 `input_ids`（词的索引）、`attention_mask`（标记哪些词是填充）等信息的张量。

分词和预处理函数

我们将使用 Hugging Face `transformers` 库中的 BERT 分词器来分词。BERT 使用的分词器是 `BertTokenizer`，它可以将文本转换为模型所需的格式。

```
from transformers import BertTokenizer

# 加载预训练的 BERT 分词器
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# 定义数据预处理函数
def preprocess_function(examples):
    # 对每条评论进行分词，padding 和 truncation 操作
    return tokenizer(examples['text'], padding='max_length', truncation=True, max_length=512)

# 对训练集和测试集进行预处理
train_dataset = dataset['train'].map(preprocess_function, batched=True)
test_dataset = dataset['test'].map(preprocess_function, batched=True)

# 查看处理后的数据
print(train_dataset[0])
```

这时，每条数据将包含 `input_ids`、`attention_mask` 和 `label`。`input_ids` 是一个数值数组，表示每个词的索引；`attention_mask` 是一个 0 或 1 的数组，1 表示该词是有效的，0 表示该词是填充的。

转换为 PyTorch 数据集格式

在训练模型之前，我们需要将数据集转换为 PyTorch 格式。这里我们使用 `set_format()` 方法来实现：

```
train_dataset.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])
test_dataset.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])
```

步骤 3：模型训练

现在我们可以加载预训练的 BERT 模型，并将其用于情感分类任务。

加载预训练模型

我们将使用 `BertForSequenceClassification` 类，该类适用于文本分类任务。它会自动添加一个分类头部用于情感预测。

```
from transformers import BertForSequenceClassification

# 加载 BERT 模型用于序列分类
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
```

- `num_labels=2` 表示分类问题是二分类问题（正面/负面）。

设置训练参数

接下来，我们需要设置训练参数。这里我们使用 Hugging Face 的 `TrainingArguments` 类来设置参数，如训练轮数、批处理大小、学习率等。

```
from transformers import TrainingArguments

# 设置训练参数
training_args = TrainingArguments(
    output_dir='./results',          # 输出文件夹，保存模型和日志
    num_train_epochs=3,              # 训练轮数
    per_device_train_batch_size=16,  # 每设备训练批大小
    per_device_eval_batch_size=64,   # 每设备评估批大小
    warmup_steps=500,                # 预热步数
    weight_decay=0.01,               # 权重衰减
    logging_dir='./logs',            # 日志文件夹
    logging_steps=10,
)
```

- `output_dir`： 模型和日志的保存位置。
- `num_train_epochs`： 训练的轮数。
- `per_device_train_batch_size`： 每次训练的批处理大小。
- `warmup_steps`： 预热步数（通常用于避免学习率过高导致训练不稳定）。
- `logging_dir`： 日志文件夹，用于 TensorBoard 可视化。

使用 `Trainer` 进行训练

`Trainer` 是 Hugging Face 提供的一个高级接口，它封装了训练的全过程，包括前向传播、损失计算、梯度计算、参数更新等操作。

```
from transformers import Trainer

# 使用 Trainer 进行训练
trainer = Trainer(
    model=model,                         # 要训练的模型
    args=training_args,                  # 训练参数
    train_dataset=train_dataset,         # 训练数据集
    eval_dataset=test_dataset,           # 测试数据集
)

# 开始训练
trainer.train()
```

步骤 4：模型评估与监控

训练过程中，我们可以使用 `Trainer` 来评估模型在测试集上的表现。训练完成后，我们也可以查看模型的最终性能。

```
# 评估模型
results = trainer.evaluate()

# 打印评估结果
print(f"Test Accuracy: {results['eval_accuracy']}")
```

你将得到模型在测试集上的分类准确率。

步骤 5：可视化训练过程

Hugging Face 的 `Trainer` 会自动记录训练过程中的一些信息，包括损失（loss）和准确率。你可以使用 TensorBoard 来可视化这些信息。

首先，启动 TensorBoard 以查看训练过程的可视化效果：

```
tensorboard --logdir=./logs
```

然后，打开浏览器并访问 `http://localhost:6006` 查看训练过程中的损失和准确率曲线。

结束语

通过这些步骤，你就可以使用 Transformer 模型（如 BERT）对 IMDb 数据集进行情感分类任务。主要的过程包括：

1. 数据加载：加载 IMDb 数据集并分为训练集和测试集。
2. 数据预处理：使用分词器对文本进行处理，转换为模型需要的输入格式。
3. 模型训练：加载预训练的 BERT 模型，设置训练参数，并使用 `Trainer` 进行训练。
4. 模型评估：评估训练后的模型在测试集上的性能。
5. 可视化：使用 TensorBoard 可视化训练过程中的损失和准确率变化。

这些步骤涵盖了从数据准备到模型训练和评估的全过程。

part 3 注意力原理的理解

一个attention和maskedattention代码

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SelfAttention(nn.Module):
    def __init__(self, d_model=2, row_dim=0, col_dim=1):
        super().__init__()
        self.W_q= nn.Linear(in_features=d_model,
                            out_features=d_model,
                            bias=False)
        self.W_k = nn.Linear(in_features=d_model,
                             out_features=d_model,
                             bias=False)
        self.W_v = nn.Linear(in_features=d_model,
                             out_features=d_model,
                             bias=False)
        self.row_dim= row_dim
        self.col_dim= col_dim
    def forward(self, token_encodings):
        q = self.W_q(token_encodings)
        k = self.W_k(token_encodings)
        v = self.W_v(token_encodings)

        sims = torch.matmul(q,k.transpose(dim0= self.row_dim,
                                          dim1= self.col_dim))

        scaled_sims = sims / torch.tensor(k.size(self.col_dim)**0.5)
        attention_percents = F.softmax(scaled_sims, dim= self.col_dim)
        attention_scores = torch.matmul(attention_percents, v)
        return attention_scores

'''encodings_matrix = torch.tensor([[1.16, 0.23],
                                    [0.57, 1.36],
                                    [4.41, -2.16]])
torch.manual_seed(42)

selfAttention = SelfAttention(d_model=2,
                              row_dim=0,
```

```
                              col_dim=1)
print(selfAttention(encodings_matrix))'''

class MaskedSelfAttention(nn.Module):
    def __init__(self, d_model=2, row_dim=0, col_dim=1):
        super().__init__()
        self.W_q= nn.Linear(in_features=d_model,
                            out_features=d_model,
                            bias=False)
        self.W_k = nn.Linear(in_features=d_model,
                             out_features=d_model,
                             bias=False)
        self.W_v = nn.Linear(in_features=d_model,
                             out_features=d_model,
                             bias=False)
        self.row_dim = row_dim
        self.col_dim = col_dim
    def forward(self, token_encodings, mask=None):
        q = self.W_q(token_encodings)
        k = self.W_k(token_encodings)
        v = self.W_v(token_encodings)

        sims = torch.matmul(q,k.transpose(dim0= self.row_dim,
                                          dim1= self.col_dim))
        scaled_sims = sims / torch.tensor(k.size(self.col_dim)**0.5)
        if mask is not None:
            scaled_sims = scaled_sims.masked_fill(mask=mask,
                                                  value=-1e9)
        attention_percents = F.softmax(scaled_sims, dim=self.col_dim)
        attention_scores = torch.matmul(attention_percents, v)
        return attention_scores

encodings_matrix = torch.tensor([[1.16, 0.23],
                                 [0.57, 1.36],
                                 [4.41, -2.16]])
torch.manual_seed(42)

maskedSelfAttention = MaskedSelfAttention(d_model=2,
                              row_dim=0,
                              col_dim=1)

mask = torch.tril(torch.ones(3,3))
```

```
mask = mask==0
print(maskedSelfAttention(encodings_matrix, mask))
```

part 4 完整的步骤

你已经成功加载了IMDB影评数据集，接下来可以进行数据预处理和使用 **Transformers** 进行情感分析。下面是完整的流程：

第一步：数据预处理

主要包括：

- **文本清理**（可选）
- **Tokenization**（使用预训练的 `transformers` tokenizer）
- **转换为 PyTorch Dataset**

1. 安装必要库

确保你安装了 `transformers` 和 `datasets`：

```
pip install transformers datasets torch
```

2. 使用 Hugging Face Tokenizer

我们使用 `AutoTokenizer` 进行 tokenization：

```python
import os
import torch
from transformers import AutoTokenizer
from torch.utils.data import Dataset

# 选择预训练模型
MODEL_NAME = "distilbert-base-uncased"  # 轻量BERT模型

# 加载Tokenizer
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)

class IMDBDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=512):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

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

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]

        encoding = self.tokenizer(
            text,
            padding="max_length",
            truncation=True,
            max_length=self.max_length,
            return_tensors="pt"
        )

        return {
            "input_ids": encoding["input_ids"].squeeze(0),
            "attention_mask": encoding["attention_mask"].squeeze(0),
            "labels": torch.tensor(label, dtype=torch.long),
        }

# 创建Dataset对象
train_dataset = IMDBDataset(train_texts, train_labels, tokenizer)
test_dataset = IMDBDataset(test_texts, test_labels, tokenizer)
```

第二步：使用 Transformer 进行情感分析

我们使用 `AutoModelForSequenceClassification` 进行分类，并使用 `Trainer` 进行微调。

1. 加载模型

```
from transformers import AutoModelForSequenceClassification

# 加载模型（2分类任务 num_labels=2）
model = AutoModelForSequenceClassification.from_pretrained(MODEL_NAME, num_labels=2)
```

2. 训练数据加载

```
from torch.utils.data import DataLoader

BATCH_SIZE = 8  # 视显存情况调整
train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

第三步：微调模型

使用 `Trainer` 进行训练：

```
from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=BATCH_SIZE,
    num_train_epochs=3,  # 训练轮数
    weight_decay=0.01,
    logging_dir="./logs",
)

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

trainer.train()
```

第四步：进行情感分析

训练完成后，我们可以使用模型进行预测：

```
def predict_sentiment(texts, model, tokenizer):
    model.eval()
    inputs = tokenizer(texts, return_tensors="pt", padding=True, truncation=True, max_length=512
    with torch.no_grad():
        outputs = model(**inputs)
    predictions = torch.argmax(outputs.logits, dim=-1)
    return ["Positive" if p == 1 else "Negative" for p in predictions]

sample_texts = ["This movie was absolutely fantastic!", "The film was boring and too long."]
predictions = predict_sentiment(sample_texts, model, tokenizer)

for text, sentiment in zip(sample_texts, predictions):
    print(f"Review: {text} | Sentiment: {sentiment}")
```

总结

- 你成功加载了 IMDB 数据集
- 我们使用 `transformers` 进行文本 **Tokenization**
- 用 `AutoModelForSequenceClassification` 进行 **二分类情感分析**
- 通过 `Trainer` 微调模型
- 训练完成后进行 **预测**

你可以运行上面的代码来训练和测试模型！ 🚀

part 5 性能的优化

使用4090显卡，显存非常充足，所以你可以将批次大小（batch size）设置得更大，以及增加训练轮数（epochs），从而提高训练效率。以下是一些建议：

建议参数调整：

- **批次大小（Batch Size）**：
  - 4090显卡具有24GB显存，你可以将 `batch_size` 设置为 **16** 或 **32**，甚至 **64**，具体取决于你的模型大小（比如DistilBERT）。
  - 推荐从 **16** 开始，逐渐增加，直到显存达到极限。
- **训练轮数（Epochs）**：
  - 如果模型和数据集较大，可以设置为 **3** 到 **5** 轮（epochs）。
  - 可以适当增加到 **6** 轮，如果需要更高的精度，训练时间允许。
- **学习率（Learning Rate）**：
  - 可以将学习率设为 **5e-5** 或 **3e-5**，这些是大多数预训练模型的推荐值。
- **Gradient Accumulation**：
  - 如果你增加批次大小但遇到显存不足的情况，可以使用 **gradient accumulation**（梯度累积），通过减少一次训练中批次的计算量来节省显存。

具体代码调整：

```
# 假设我们选择 batch_size = 32，epochs = 3
BATCH_SIZE = 32  # 更大的batch size
EPOCHS = 3  # 训练轮数

# 加载训练参数
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",  # 每个epoch后评估
    save_strategy="epoch",  # 每个epoch后保存模型
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=BATCH_SIZE,
    num_train_epochs=EPOCHS,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=100,  # 每100步记录一次日志
    learning_rate=5e-5,  # 学习率
    fp16=True,  # 开启混合精度训练
)

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

trainer.train()
```

混合精度训练（FP16）

4090显卡支持 **FP16**（混合精度训练），可以显著提高训练速度并减少显存占用。在 `TrainingArguments` 中设置 `fp16=True` 来启用混合精度训练。

调试和监控

你可以通过监控显存使用情况（使用 `nvidia-smi` ）来调整批次大小和其他参数。如果显存已经满了，适当减少 `batch_size` 或调整其他参数，直到显存足够。

这种设置将让你充分利用4090的强大计算能力！

为了在训练过程中实时监控模型在测试集上的性能，并记录损失函数和预测准确率等指标，我们可以使用 `Trainer` 和 `TrainingArguments` 提供的一些功能来实现。

步骤：

1. **评估和记录：**
   - 使用 `evaluation_strategy="steps"` 或 `evaluation_strategy="epoch"` 来设置评估频率。
   - 配置 `metric_for_best_model` 来选择性能最好时保存的模型（例如选择准确率）。
2. **设置自定义评估指标：**
   - 通过 `compute_metrics` 函数来计算准确率（accuracy）等指标。
3. **可视化训练过程：**
   - 使用 `TensorBoard` 来可视化损失函数和准确率。

完整代码：

1. 自定义计算指标

我们可以使用 `compute_metrics` 来计算模型的准确率，并传递给 `Trainer` 进行实时计算。

```
from sklearn.metrics import accuracy_score
from transformers import Trainer, TrainingArguments

# 计算准确率
def compute_metrics(p):
    predictions, labels = p
    predictions = predictions.argmax(axis=-1)  # 获取预测标签
    accuracy = accuracy_score(labels, predictions)
    return {"accuracy": accuracy}

# 设置训练参数
training_args = TrainingArguments(
    output_dir="./results",  # 保存模型的路径
    evaluation_strategy="epoch",  # 每个epoch后评估
    save_strategy="epoch",  # 每个epoch后保存模型
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=BATCH_SIZE,
    num_train_epochs=EPOCHS,
    weight_decay=0.01,
    logging_dir="./logs",  # 保存日志
    logging_steps=100,  # 每100步记录一次日志
    learning_rate=5e-5,  # 学习率
    fp16=True,  # 开启混合精度训练
    load_best_model_at_end=True,  # 最好模型保存
    metric_for_best_model="accuracy",  # 选择评估指标为准确率
)

# 创建Trainer对象并传递compute_metrics
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    compute_metrics=compute_metrics,  # 添加评估函数
)
```

2. 启用 TensorBoard 监控

`logging_dir` 参数指定日志文件的存储位置，这样就可以使用 `TensorBoard` 来可视化训练过程中的损失和准确率。

```
# 在终端中启动 TensorBoard
tensorboard --logdir=./logs
```

然后你可以在浏览器中访问 `http://localhost:6006/` 来查看可视化结果。

3. 可视化损失函数和准确率

在训练过程中， `Trainer` 会自动将损失值和准确率记录到 TensorBoard。你只需要在训练时设置 `logging_dir` 和 `logging_steps` 参数。

4. 模型训练与监控

```
# 开始训练
trainer.train()
```

在训练过程中，TensorBoard 会显示损失曲线和准确率曲线。你可以通过调整 `logging_steps` 来调整记录频率。

5. 进阶：使用 `evaluate` 方法进行测试集评估

你还可以在训练完成后手动评估模型性能，例如获取损失和准确率：

```
# 手动评估模型在测试集上的性能
eval_results = trainer.evaluate(eval_dataset=test_dataset)
print(f"Test Accuracy: {eval_results['eval_accuracy']}")
```

总结：

- 使用 `compute_metrics` 来计算准确率等评估指标。
- 配置 `TrainingArguments` 中的 `evaluation_strategy` 和 `logging_dir` ，并启用 `TensorBoard` 来监控训练过程中的损失和准确率。
- 利用 `trainer.evaluate()` 来在训练过程中或训练后手动评估模型性能。

这样，你就能实时监控模型的性能，并在训练过程中进行必要的调整。

part 6 离线的解决

这个错误通常是由于以下几个原因之一导致的：

1. 本地目录冲突

- 你的本地环境可能有一个名为 `distilbert-base-uncased` 的文件夹，而 `transformers` 库会尝试从本地加载模型文件，而不是从 Hugging Face Hub 下载它。请确保没有与模型名称冲突的本地文件夹。

2. 缓存问题

- Hugging Face 模型可能未能成功下载，或者缓存中的文件可能损坏，导致无法加载 tokenizer。

3. 网络连接问题

- 如果你无法从 Hugging Face Hub 下载模型，可能是网络连接问题。

4. 安装版本问题

- 有时，如果你的 `transformers` 库不是最新版本，可能会出现兼容性问题。

解决方法：

1. 删除本地缓存和冲突的文件夹

检查你的本地文件系统，确保没有名为 `distilbert-base-uncased` 的文件夹。如果有，请删除它。

```
rm -rf distilbert-base-uncased
```

你也可以尝试清空 Hugging Face 的缓存目录：

```
import shutil
cache_dir = os.path.expanduser("~/.cache/huggingface")
shutil.rmtree(cache_dir, ignore_errors=True)
```

2. 更新 `transformers` 库

确保你正在使用 `transformers` 的最新版本，执行以下命令来更新它：

```
pip install --upgrade transformers
```

3. 手动下载模型文件

如果网络问题导致无法自动下载，你可以手动下载模型文件并指定路径：

- 访问 Hugging Face 上的 `distilbert-base-uncased` 页面：https://huggingface.co/distilbert-base-uncased
- 下载所需的模型文件，尤其是 `tokenizer_config.json` 和 `vocab.txt` 。
- 将它们存放在本地目录，并通过 `from_pretrained` 加载：

```
from transformers import AutoTokenizer

# 使用本地目录加载 tokenizer
tokenizer = AutoTokenizer.from_pretrained("/path/to/local/directory")
```

4. 检查网络连接

确保你的计算机能够连接到 Hugging Face Hub，检查是否有防火墙或代理设置干扰。

你可以通过 `ping` 命令检查是否能够连接到 Hugging Face 服务器：

```
ping huggingface.co
```

5. 更换模型名称或检查路径

确保你正在使用正确的模型名称。你可以尝试加载不同的模型来确认问题是否仅限于 `distilbert-base-uncased` 。

```
from transformers import AutoTokenizer

# 使用其他模型，例如 bert-base-uncased
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
```

通过这些步骤，你应该能够解决 `OSError: Can't load tokenizer for 'distilbert-base-uncased'` 的问题。如果问题仍然存在，欢迎提供更多信息，我可以进一步帮助你。