实验二：基于BERT的文本分类实验

PB22151796-莫环欣

实验目的

`BERT` 在自然语言处理中可以同时考虑句子上下文，支持 `MLM` 与 `NSP` 任务与微调，被广泛应用于文本分类、**情感分析**、机器翻译、问答系统等任务。本次实验目的是在情感分析任务上微调 `BERT` ，以学习 `端到端训练` 与 `先预训练后微调` 两种训练策略。

实验环境

```
Python 3.11.12
os          : posix
torch       : 2.6.0+cu124
pandas      : 2.2.2
matplotlib  : 3.10.0
sklearn     : 1.6.1
transformers: 4.51.3

+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |                    0 |
| N/A   53C    P8             10W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

数据集描述

本次实验中主要选择了`SST2`数据集，并将`IMDB`数据集的无监督数据用于第二部分实验微调前的预训练以熟悉领域特征

```
IMDB_PATH = "./data/imdb/plain_text/"
IMDB_TRAIN = IMDB_PATH + "train-00000-of-00001.parquet"
IMDB_TEST = IMDB_PATH + "test-00000-of-00001.parquet"
IMDB_UNSUPERVISED = IMDB_PATH + "unsupervised-00000-of-00001.parquet"    # 无标签数据，拿来做Task2的语料训练
```

首先是`IMDB`数据集，无监督数据集共有`50000`条，训练集与测试集各有`25000`条，但其中有`123`条重复数据

```
                                                text  label
0  I rented I AM CURIOUS-YELLOW from my video sto...      0
1  "I Am Curious: Yellow" is a risible and preten...      0
2  If only to avoid making this type of film in t...      0
3  This film was probably inspired by Godard's Ma...      0
4  Oh, brother...after hearing about this ridicul...      0
                                                text  label
0  This is just a precious little diamond. The pl...     -1
1  When I say this is my favourite film of all ti...     -1
2  I saw this movie because I am a huge fan of th...     -1
3  Being that the only foreign films I usually li...     -1
4  After seeing Point of No Return (a great movie...     -1
训练集长度: 25000; 测试集长度: 25000; 无标签数据长度: 50000
警告：训练集和测试集有 123 条重复样本!
label
0    0.5
1    0.5
Name: proportion, dtype: float64
```

上面是训练集头部，下面是无监督集头部，同时在最下方可以看到测试集里的两种标签分布非常平均（使用`print(df["label"].value_counts(normalize=True))`统计）

对于`SST2`数据集，其文本列的原名为`sentence`，这里在后续加载时会统一改为`text`；另外其还有一列索引，对于本次任务用处不大，加载时直接去掉了

```
SST2_PATH = "./data/sst2/data/"
SST2_TRAIN = SST2_PATH + "train-00000-of-00001.parquet"
SST2_TEST = SST2_PATH + "test-00000-of-00001.parquet"
SST2_VALID = SST2_PATH + "validation-00000-of-00001.parquet"
```

可以看到标签分布虽有些差别，但并不是很大。这里测试集标签全部被屏蔽（也就相当于`IMDB`的无监督数据集），后续使用时将训练集与验证集拼接后再按`8:1:1`划分出训练集、验证集、测试集

```
   idx                                           sentence  label
0    0      hide new secretions from the parental units       0
1    1              contains no wit , only labored gags       0
2    2  that loves its characters and communicates som...      1
3    3  remains utterly satisfied to remain the same t...      0
4    4  on the worst revenge-of-the-nerds clichés the ...      0
   idx                                           sentence  label
0    0            uneasy mishmash of styles and genres .     -1
1    1  this film 's relationship to actual tension is...     -1
2    2  by the end of no such thing the audience , lik...     -1
3    3  director rob marshall went out gunning to make...     -1
4    4  lathan and diggs have considerable personal ch...     -1
训练集长度: 67349; 验证集长度: 872; 测试集长度: 1821
训练集和验证集完全不同。
label
1    0.557826
0    0.442174
Name: proportion, dtype: float64
```

基础实验步骤

其余配置如下

```
MAX_LENGTH = 128 # 不超过512
BATCH_SIZE = 128 # 为128以上时会导致colab崩溃
LEARNING_RATE = 5e-5
EPOCHS = 20
NUM_LABELS = 2
```

数据预处理

从上面的数据头展示中可以看到文本并不干净，这里在加载数据集之前按文档里的要求首先去除`html`标签、标点符号和多余的空格，同时按照`BERT`的要求将输入全部改为小写（这里微调的是`bert-base-uncased`），同时针对三种情况（两个数据集的正常训练（统一包含`text`与`label`两列字典的列表）与`MLM`（仅需`text`））进行了处理

```
def clean_text(text):
    text = re.sub(r'<.*?>', '', text)                                  # 去除HTML标签
    text = text.translate(str.maketrans('', '', string.punctuation))  # 去除标点符号
    text = text.lower()                                                # 转为小写（uncased模型需要）
    text = re.sub(r'\s+', ' ', text).strip()                           # 去除多余空格
    return text
def preprocess_dataset(df, ds_type="SST2", is_MLM=False):
    row_name="sentence" if ds_type == "SST2" else "text"
    df["text"] = df[row_name].apply(clean_text)
    if not is_MLM:
        data_list = df[["text", "label"]].to_dict(orient="records")
        print(df["label"].value_counts(normalize=True))
    else:
        data_list = df["text"].to_list()
    return data_list
```

预处理后可以得到以下数据结构，列表中每条数据字典里都包含文本与其标签

```
[{'text': 'hide new secretions from the parental units', 'label': 0}, {'text': 'contains no wit only labored gags', 'label': 0}]
```

之前另一门课在做`Enron-Spam`的时候也进行过相似处理，当时我选用了`spaCy`模型进行分词，并且额外去除了停用词和单个字母，但这里感觉还是保留一下，可以包含更多上下文信息和情感倾向

数据装载

得到干净数据之后，参考文档中的伪代码搭建了数据集结构，同时对`MLM`与非`MLM`任务进行了区分

```python
# data_list是经过预处理的列表，每项包含 text 和 label
class SentimentDataset(Dataset):
    def __init__(self, data_list, tokenizer, max_length=MAX_LENGTH, is_MLM=False):
        self.data = data_list
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.is_MLM = is_MLM

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

    def __getitem__(self, idx):
        if not self.is_MLM:
            text = self.data[idx]["text"]
        else:
            text = self.data[idx]   # MLM 只有一列

        encoding = self.tokenizer(
            text,
            add_special_tokens=True,    # 加上 [CLS]（开头） 和 [SEP]（结尾），不够的用[PAD]填充
            max_length=self.max_length, # 输入序列的最大长度
            padding="max_length",
            truncation=True,            # 启用截断
            return_tensors="pt"
        )
        if not self.is_MLM:
            return {key: val.squeeze(0) for key, val in encoding.items()}, torch.tensor(self.data[idx]["label"])

        return {key: val.squeeze(0) for key, val in encoding.items()}
```

随后逐步装载即可，同时为`MLM`任务引入随机掩码，这里借助`DataCollatorForLanguageModeling`实现

```python
def load_data(tokenizer, is_MLM=False):
    train_data, val_data, test_data = init_data(is_MLM)
    train_dataset = SentimentDataset(train_data, tokenizer, is_MLM=is_MLM)
    val_dataset = SentimentDataset(val_data, tokenizer, is_MLM=is_MLM)
    test_dataset = SentimentDataset(test_data, tokenizer)
    test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE)

    # 自动处理掩码，随机掩码（默认15%的）token并使用原始input_ids作为labels（非掩码位置设为-100以忽略损失）
    collate_fn = None if not is_MLM else DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=True)

    train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn, num_workers=2, pin_memory=True)
    val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, collate_fn=collate_fn, num_workers=2, pin_memory=True)

    return train_loader, val_loader, test_loader
```

其中的数据初始化部分封装了文件读取功能，当为`MLM`任务时会读取`IMDB`的无监督数据集并覆盖掉训练集与验证集

```
'''
IMDB只有训练集和验证集，SST2的测试集也都没有标注
因此这里先统一处理，将训练集和验证集合并后再统一划分，并且确保划分后三个数据集中的标签分布与原数据集（这里指合并后的）
    需要注意的是IMDB的训练集和验证集有123条重复数据（这里因为没用到所以就先不管）
'''
def init_data(is_MLM=False):
    train_data = preprocess_dataset(pd.read_parquet(SST2_TRAIN), ds_type="SST2")
    val_data = preprocess_dataset(pd.read_parquet(SST2_VALID), ds_type="SST2")
    combined_data = (train_data + val_data)[:10000]
    labels = [item["label"] for item in combined_data]

    # 训练：测试+验证 = 8:2
    train_data, temp_data = train_test_split(
        combined_data, test_size=0.2, random_state=42, stratify=labels
    )

    temp_labels = [item["label"] for item in temp_data]
    # 验证：测试 = 1:1
    val_data, test_data = train_test_split(
        temp_data, test_size=0.5, random_state=42, stratify=temp_labels
    )

    # 无监督训练用不到测试集，把原有的训练集和验证集覆盖，保留正式数据划分好的测试集
    if is_MLM:
        combined_data = preprocess_dataset(pd.read_parquet(IMDB_UNSUPERVISED), ds_type="IMDB", is_MLM=True)
        train_data, val_data = train_test_split(
            combined_data[:10000], test_size=0.2, random_state=42
        )
    random.shuffle(train_data)
    random.shuffle(val_data)
    random.shuffle(test_data)

    print(f"训练集长度: {len(train_data)}; 验证集长度: {len(val_data)}; 测试集长度: {len(test_data)}")
    return train_data, val_data, test_data
```

模型结构

模型的结构比较简单，`BERT`模型本身已经比较复杂，因此直接将输入接到`BERT`，并将其输出经过分类器即可；另外这里使用`outputs.pooler_output`而非`outputs[0][:, 0, :]`切片，这样过渡到本次任务的全连接层（分类器）会更“平滑”一些，也可以理解为多出来的那些也是本次全连接层的一部分；又由于需要计算损失，模型最后直接返回`logits`，在训练时再套上`softmax`

```
class BertClassifier(nn.Module):
    def __init__(self, mlm=False):
        super(BertClassifier, self).__init__()
        model_path = MODEL_PATH if not mlm else MODEL_PRE_PATH
        self.bert = BertModel.from_pretrained(model_path)

        self.dropout = nn.Dropout(0.6)
        self.classifier = nn.Linear(self.bert.config.hidden_size, NUM_LABELS)  # 分类层
        #self.activation = nn.Sigmoid() # 二分类

    def forward(self, input_ids, attention_mask):
        # BERT 编码器
        outputs = self.bert(
            input_ids=input_ids,
            attention_mask=attention_mask # 区别有效区域与填充区域
        )
        # 基于 [CLS] token 的隐藏状态，经过一个额外的全连接层和 tanh 激活函数后的结果
        cls_output = outputs.pooler_output

        # Dropout + 分类层
        #cls_output = self.dropout(cls_output)
        logits = self.classifier(cls_output)
        return logits
```

对于`self.bert.config.hidden_size`，其值为`768`，表示`BERT`的输出维度，`NUM_LABELS`被我配置为`2`（情感二分类）

```
bert_out = BertModel.from_pretrained(MODEL_PATH).config.hidden_size
print(f"bert_out: {bert_out}")
```

```
bert_out: 768
```

当然也可以直接通过`BertForSequenceClassification.from_pretrained(MODEL_PATH, num_labels=NUM_LABELS)`调用（像伪代码里那样），但由于文档里要求直接将`BERT`作为编码器并接上分类层，这里就换用了`BertModel.from_pretrained(model_path)`然后手动接上全连接层

训练逻辑

这里的训练代码可以直接沿用其它模型的训练器，但由于`BERT`本身具有相当的深度（十二层编码器，十二个注意力头，约`1.1`亿参数），需要进行一些关键修改以降低过拟合风险：

- 优化器改用`AdamW`
  - 其在`Adam`的基础上加入了权重衰减，防止过拟合
- 学习率调度器改用`get_linear_schedule_with_warmup`
  - 主要包含`num_warmup_steps`和`num_training_steps`两个参数
    - 训练过程中会在前`num_warmup_steps`步中逐渐增大到优化器中预设的学习率，并在后续步骤中逐渐线性下降到`0`
    - `num_training_steps`直接设为总步数即可，会自动考虑预热步数
  - 这里感觉预热步骤可以设为`1-2`个`epoch`，让模型在初始化权重之后先不急着进行大面积更新，以防陷入负优化

```
# AdamW 是加入了权重衰减的Adam
self.optimizer = AdamW(self.model.parameters(), lr=lr)
# 学习率从0增加到设定的最大值然后逐渐线性下降
self.scheduler = get_linear_schedule_with_warmup(
    self.optimizer,
    num_warmup_steps=2,
    num_training_steps=len(train_loader)*epochs
)
```

同时由于文档中要求使用交叉熵损失函数，这里选用`nn.CrossEntropyLoss()`，不过感觉对这个二分类任务用`BCE`可能也很不错

另外，训练器中还包含了早停逻辑，我在这里将其设计为同时考虑验证集主要性能与损失，当性能有提升或损失有下降时重置容忍轮次计数器，否则将其自增，超过容忍上限时停止训练

```
if not self.is_MLM and val_acc > self.best_accuracy:
    self.best_accuracy = val_acc
    self.bear_cnt=0
    torch.save(self.model.state_dict(), self.save_name)
if val_loss < self.min_loss:
    self.min_loss = val_loss
    self.bear_cnt = 0
else:
    self.bear_cnt += 1
    if self.bear_cnt >= 3:
        print("[INFO] Early stop")
        break
```

同时还使用了`torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)`进行梯度裁剪；并且考虑到模型较深，还引入了混合精度训练，此处下文再谈

此外，`MLM`任务不需要使用标签，因此在前向传递以及获取输出的时候需要注意一下，例如

```
if not self.is_MLM:
    inputs, labels = batch
    labels = labels.to(self.device)
else:
    inputs = batch
    labels = inputs["input_ids"].to(self.device)
```

端到端训练

这里在初始化模型结构后，从预训练的`BERT`路径下获取分词器用于数据装载，之后就是很常规的训练步骤，不分阶段直接训练整个网络并更新所有参数。这里在`model = BertClassifier()`一步会自动加载`BERT`预训练权重，从头训练太慢了

```
# 初始化模型和数据
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = BertClassifier()
model.to(device)
tokenizer = BertTokenizer.from_pretrained(MODEL_PATH)
train_loader, val_loader, test_loader = load_data(tokenizer)

trainer = ModelTrainer(
    model=model,
    train_loader=train_loader,
    val_loader=val_loader,
    epochs=EPOCHS,
    lr=LEARNING_RATE
)

evaluator = ModelEvaluator(test_loader=test_loader)
print("端到端微调前的性能：")
initial_metrics = evaluator.evaluate_model(model=model)
trainer.train()
train_show(trainer)
print("端到端微调后的性能：")
final_metrics = evaluator.evaluate_model(model=model)
# 对比端到端训练前后性能变化
compare_eval(initial_metrics, final_metrics)
```

先预训练后微调

预训练时需要使用`BertForMaskedLM`而非`BertModel`来加载预训练权重，其它的区别在于接口内部的逻辑，上文已经谈过了

```
# 加载预训练模型
is_MLM = True
model = BertForMaskedLM.from_pretrained(MODEL_PATH)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

tokenizer = BertTokenizer.from_pretrained(MODEL_PATH)

train_loader, val_loader, test_loader = load_data(tokenizer, is_MLM)

mlm_trainer = ModelTrainer(model=model, train_loader=train_loader, val_loader=val_loader, epochs=2, lr=LEARNING_RATE, is_MLM=is_MLM)

# 损失在3.5以下（通用预训练MLM损失）比较正常
mlm_trainer.train()

model.save_pretrained(MODEL_PRE_PATH, safe_serialization=False)
tokenizer.save_pretrained(MODEL_PRE_PATH)
```

微调时又有一些新内容，此时需要在保持预训练权重大部分参数学习率较低的前提下对分类头进行相对较高的学习率微调以加快适应任务，因此引入了分层学习率

```python
def layerwise_lr_decay(model, lr=5e-5, decay=0.8):
    # 分类头参数（学习率较高）
    # classifier_params = list(model.classifier.parameters()) + list(model.dropout.parameters())
    classifier_params = list(model.classifier.parameters())

    # BERT本体参数（学习率较低）
    bert_params = []
    bert_lr = lr *0.5
    for i, layer in enumerate(model.bert.encoder.layer[::-1]):  # 从最后一层开始
        bert_params.append({"params": layer.parameters(), "lr": bert_lr})
        bert_lr *= decay  # 每层学习率衰减

    # 分类头学习率较高，放在前面确保优化器优先处理
    return [{"params": classifier_params, "lr": lr}] + bert_params
```

之后再将其覆盖到原有训练器中即可

```python
...
params = layerwise_lr_decay(model)
optimizer = AdamW(params, weight_decay=0.01)
scheduler = get_linear_schedule_with_warmup(
    optimizer, num_warmup_steps=2, num_training_steps=len(train_loader) * EPOCHS
)
...
trainer.optimizer = optimizer  # 覆盖默认优化器 <-- 引入分层学习率
trainer.scheduler = scheduler
trainer.save_name = MODEL_PRE_NAME
...
```

结果与分析

端到端训练

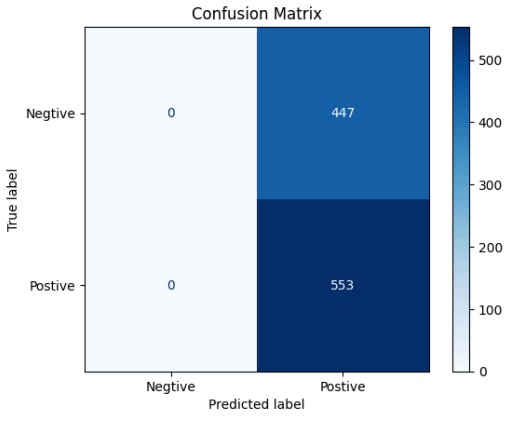
这里先裁剪出`SST2`数据集的前`10000`条数据进行实验

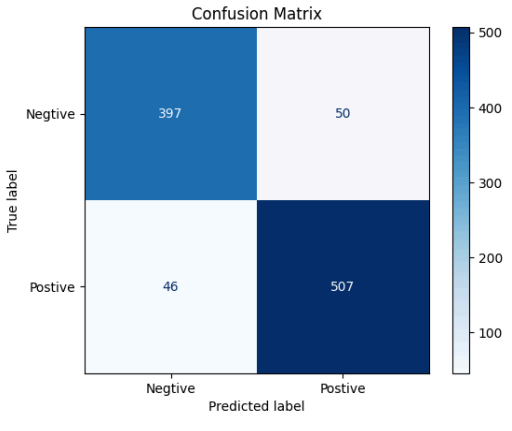
```
label
1    0.557826
0    0.442174
Name: proportion, dtype: float64
label
1    0.509174
0    0.490826
Name: proportion, dtype: float64
训练集长度: 8000; 验证集长度: 1000; 测试集长度: 1000
```

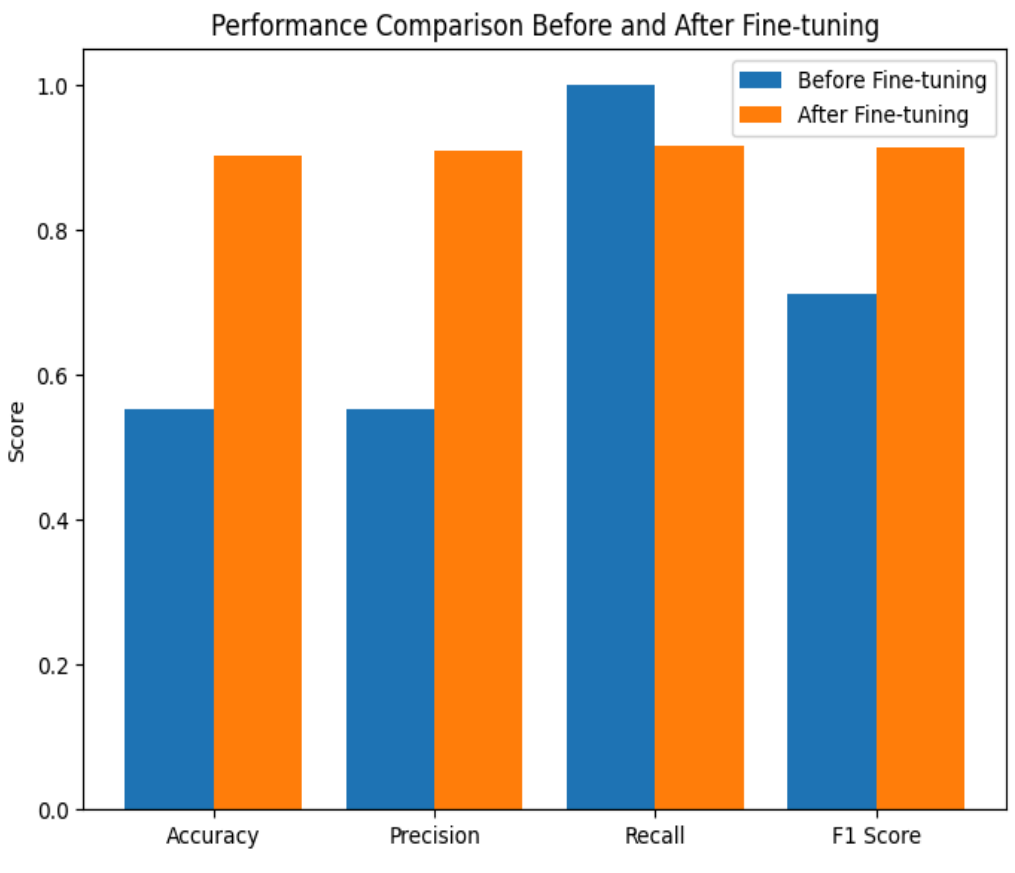
从这些定量结果上可以看到训练后模型性能有了显著提升（`Recall`显示出**模型对正类的倾向性太强**，结合其与`F1`可以看到模型在训练过后对两种类别的倾向性有了一些平衡）

> 这可能是BERT在预训练时正样本更多或正样本特征更显著

性能指标	微调前	微调后
Accuracy	0.5530	0.9040
Precision	0.5530	0.9102
Recall	1.0000	0.9168
F1 Score	0.7122	0.9135

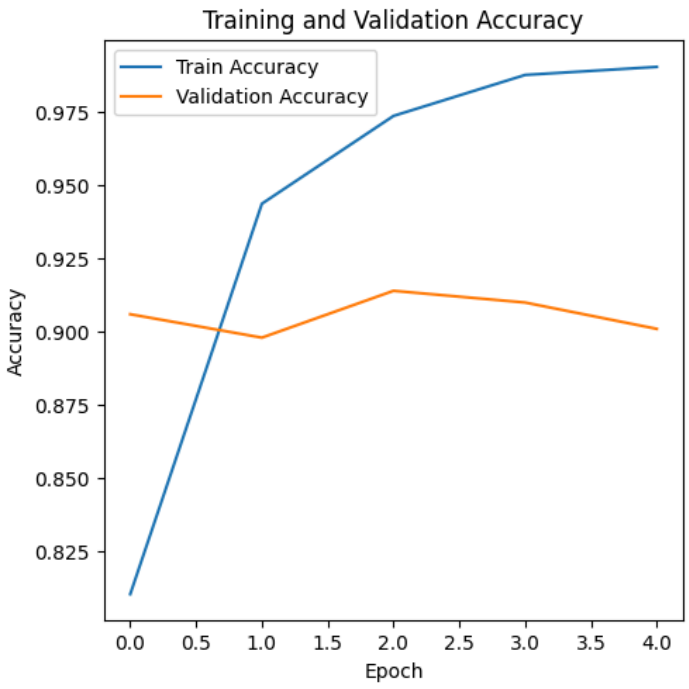


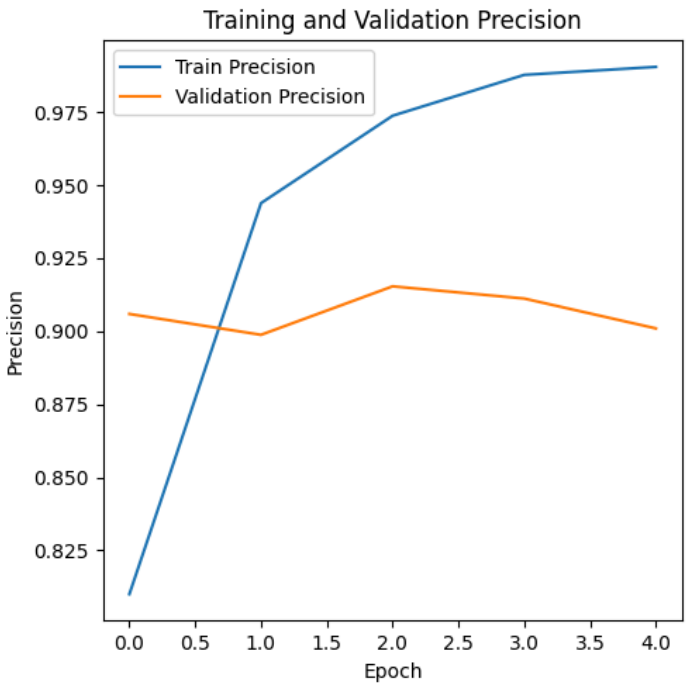


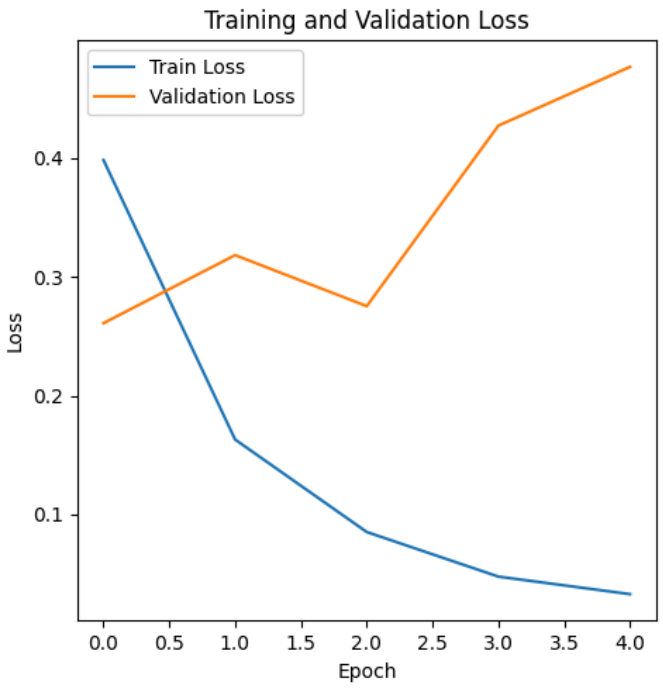


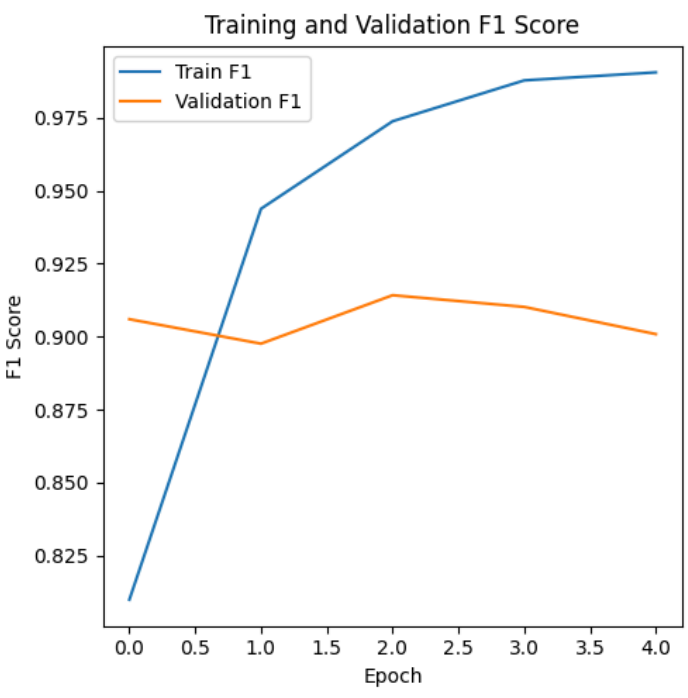
并且由于加载了预训练好的`BERT`权重，其中包含了大量的预训练通用参数，因此在第一轮权重更新之后模型就已经能够很好地专注于当前具体的分类任务

但也由于`BERT`模型的参数量非常大，我们本次的数据相对来说就比较少了，因此这之后模型就由于数据不足、模型太复杂而出现了一定程度的过拟合，并且验证集有不收敛的倾向





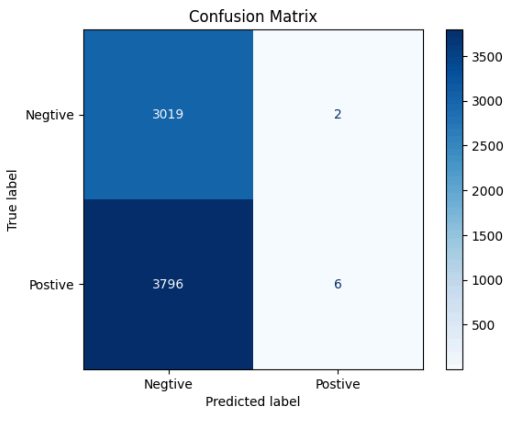


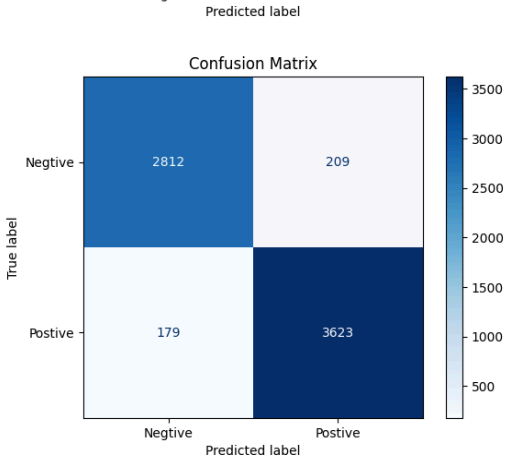


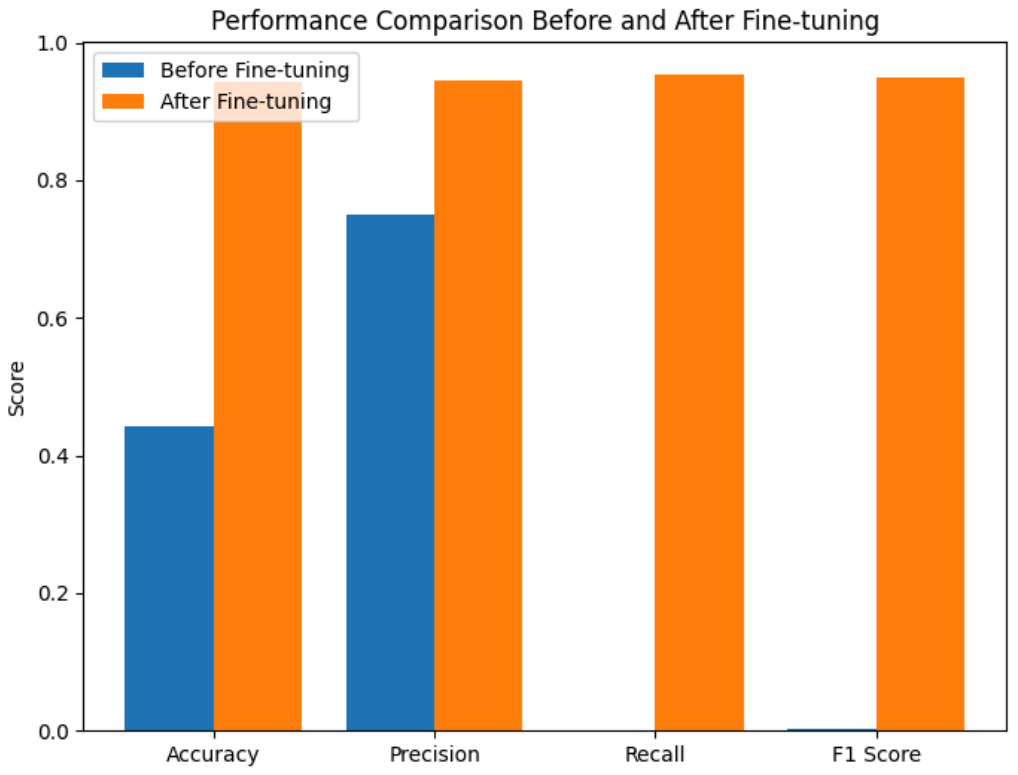
为了印证这一观点，我又换用全量数据集重新进行了一次实验

```
label
1    0.557826
0    0.442174
Name: proportion, dtype: float64
label
1    0.509174
0    0.490826
Name: proportion, dtype: float64
训练集长度：54576；验证集长度：6822；测试集长度：6823
```

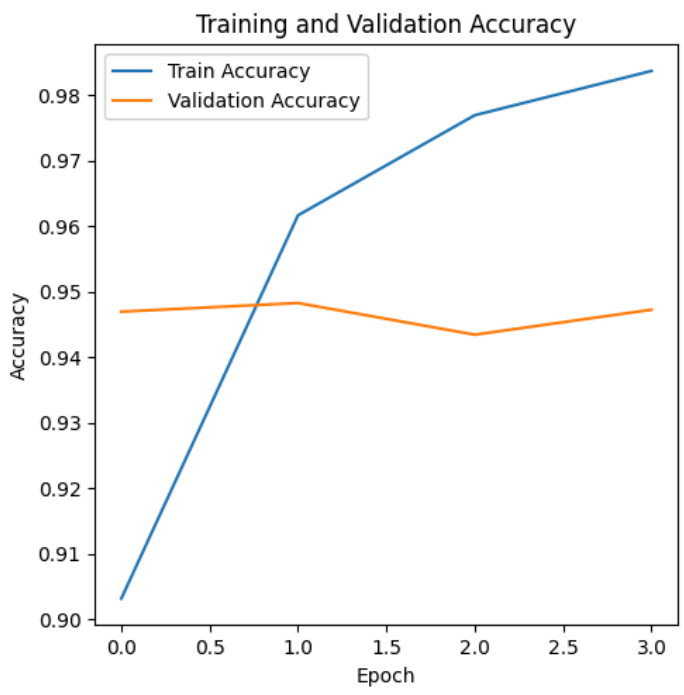
性能指标	微调前	微调后
Accuracy	0.5565	0.9377
Precision	0.5570	0.9401
Recall	0.9974	0.9487
F1 Score	0.7148	0.9444

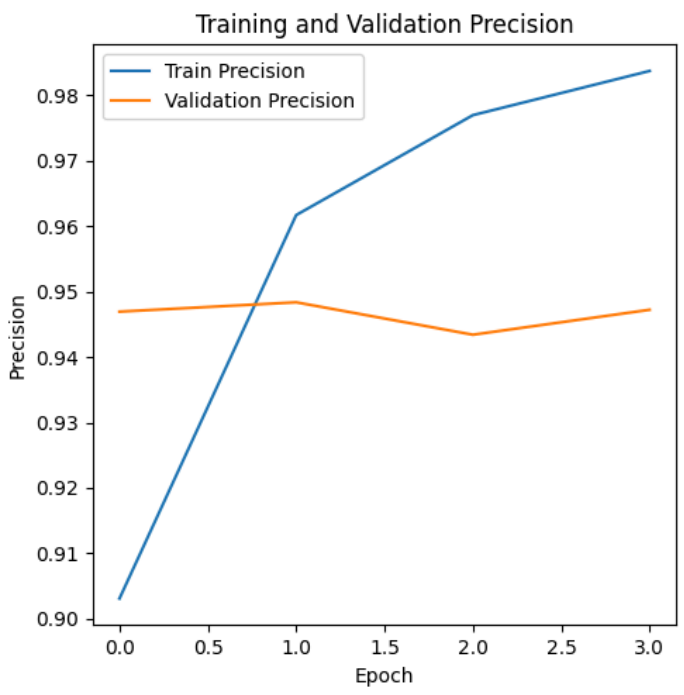


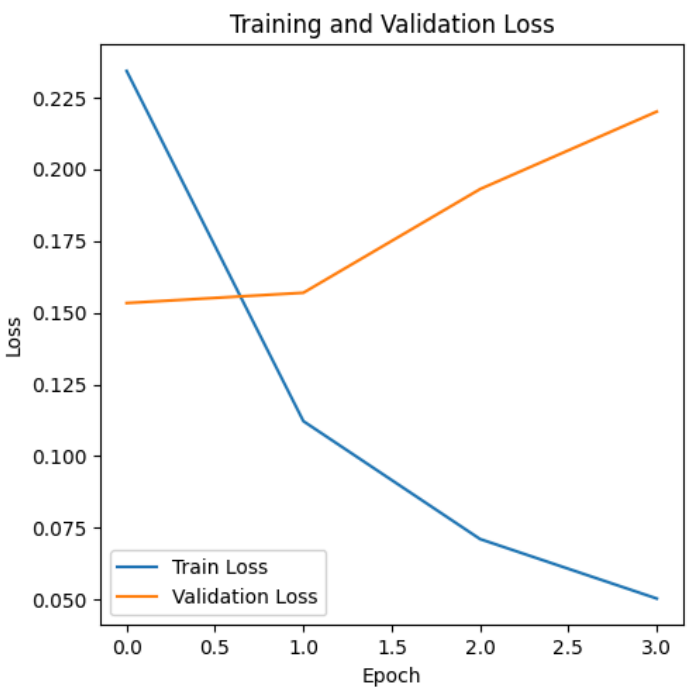


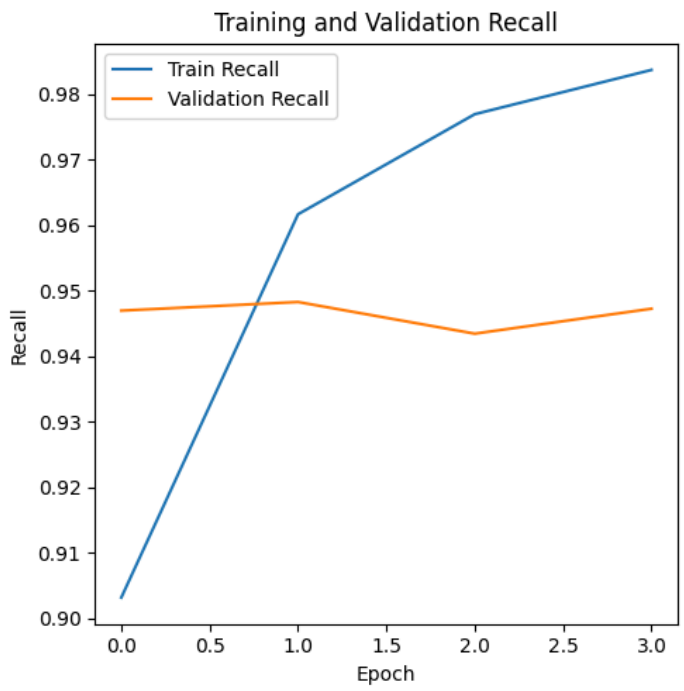


可以看到虽然`BERT`的参数量依然很大，各指标整体趋势不变，但我们增大数据量之后模型在验证集上的表现有显著上升。可以预见的是，如果有足量的数据、外加适当的数据增强策略，模型性能有望再提升两个百分点以上（但本次由于资源限制，实在无法继续进行）









那么对于全量的数据集，分别对正负样本分析两项指标：

- 正样本：
  - 精确率：$\frac{3623}{3623+179} \approx 95.29$
  - 召回率：$\frac{3623}{3623+209} \approx 94.55$
- 负样本：
  - 精确率：$\frac{2812}{2812+209} \approx 93.08$
  - 召回率：$\frac{2812}{2812+179} \approx 94.02$

可以看到模型**对于正样本的区分能力和覆盖能力相对负样本更强**，同时整体性能也挺好。前者在训练前的`Recall`指标中就初见端倪，训练后由于模型对于本任务的处理能力更强了，正负样本的分类效果就更相近了一些，但依然延续着原有权重对正样本的倾向性（也**有一部分原因是因为本任务的训练集的正样本比例（`56%`）也比负样本（`44%`）大**）

从测试集中抽取`1000`条数据进行测试，出现`43`条误分类文本，这里取前十条进行分析

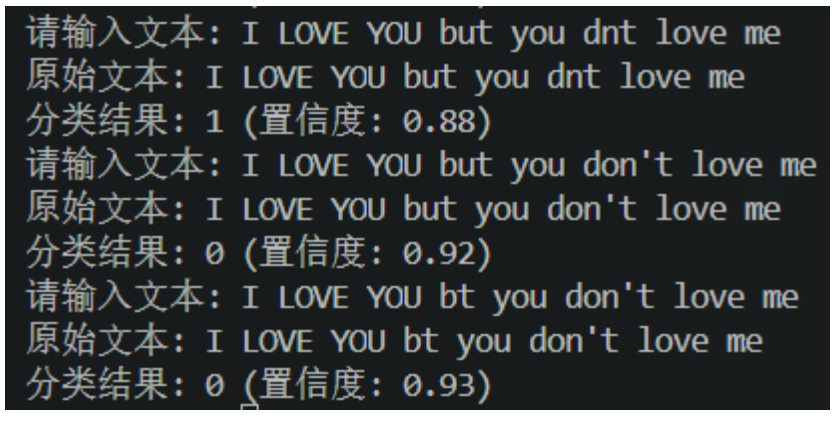
```
Text: streetsmart
True Label: 1, Pred: 0
--------------------------------------------------
Text: gimmicky
True Label: 1, Pred: 0
```

对于这两个以单个单词组成的句子，在词表中都没有出现过，一般情况下在向量化/序列化时这些词会被处理为`<unk>`；不过`BERT`会尝试将它们分开，`streetsmart`就变为词表中有的`street`和`smart`，理论上完全应该判为正例，但模型可能之前学习的数据中`street`带有更多的负例倾向，盖过了`smart`；第二条文本比较好理解一些，自身没有出现且拆开后也不在词表中，模型随机进行了判别

```
Text: i do nt mind having my heartstrings pulled but do nt treat me like a fool
True Label: 0, Pred: 1
```

这一句重点需要关注中间的`but`转折以及后面的负面态度，模型应该是过于关注前半部分（`i do not mind`）而忽视了转折在句子中的作用，从而产生误判

在做拓展内容时，针对以下句子的测试也说明了这一点，模型对于`but`的注意力非常有限，其产生的扰动远远不如`don't`等否定词



这可能是因为分类头在微调过程中接收到的内容主要是`BERT`传入的特征向量，并基于此进行情感的分类，而由于我们设计的分类头中只包含一个`Linear`，对这些特征向量的处理能力可能不太够，导致一些比较重要的转折没有成功获取到；针对这一点其实可以加入一些数据增强，比如手动将上面测试的几个数据放进训练集去

不过再测试发现`however`、`so`这些词的表现（和鲁棒性）都还不错，也可能刚好是`but`这个转折的权重太低了

```
Text: has made a film of intoxicating atmosphere and little else
True Label: 0, Pred: 1
```

对于这一句，`intoxicating atmosphere`比较积极，但后续就有些消极，模型可能无法很好地理解这种先扬后抑的语义组合，导致将整句话误判为积极

```
Text: sticks with its subjects a little longer
True Label: 1, Pred: 0
```

但是上面的猜测到了这一句似乎又有些说不通了，毕竟`stick`表述为坚持的场景应该比较多，是经典的正例词，而模型似乎更看重`little`的负面影响，从而误判为负

```
Text: this visual trickery
True Label: 0, Pred: 1
```

这句的误判就更难理解了，词表中都有，而且出现的基本都是负面词，感觉只能解释为模型没学到过这部分数据；或者在针对当前任务更新权重时这部分内容恰好被遗忘了（且训练集中相关内容较少）

```
Text: creepy stories
True Label: 1, Pred: 0
```

对于这一句，如果单看字面意思我个人也认为这是一个负例，但可能因为这只是一个客观表述而不涉及具体情感，因此不能说它具有负面评价？模型在分类时大概率就是检测并识别到了`creepy`，从而判为了负例

```
Text: on cable
True Label: 0, Pred: 1
```

这句跟上面的一样，表述内容很少且都非常的中性，感觉是属于噪声数据，模型误判是可以理解的

```
Text: lynch jeunet and von trier
True Label: 1, Pred: 0
```

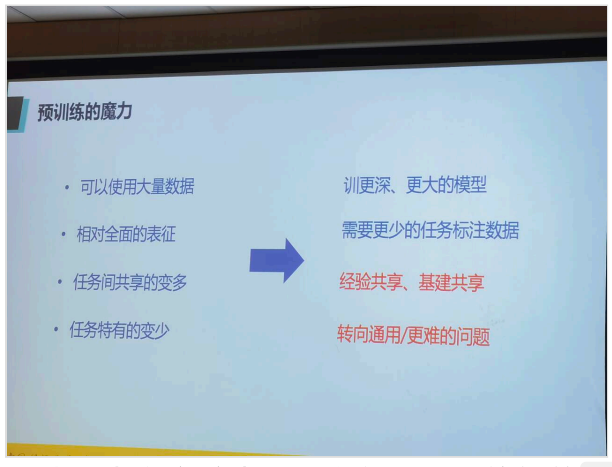
这些都是具体的人名，而模型大概是没有学习过相关知识，从而作为中性词随机（上面提到模型更倾向于判正例）判为了负例；当然也可能是作为中性词拆开后出现了某些我不认识的负面英文单词

```
Text: long after this film has ended
True Label: 1, Pred: 0
```

这句其实感觉也是中性的，模型可能放大了`long`和`ended`的负面影响

先预训练再微调

预训练



预训练可以使得我们使用更多的数据，获得相对更全面的表征，在训练新网络时需要更少的任务标注数据（比如预训练数据中包含多种交通工具的数据，现在需要进行一个车辆分类任务，那么其中很多车辆可能都在预训练数据中出现过了，在此基础上细化会比从零开始容易很多）

刚好敲到这里的时候老师在上面放这张`PPT`，真是缘分

具体到本次实验中，我们使用预训练好的`BERT`模型权重，其中包含大量通用数据的特征；再使用情感分类相关领域的数据过一遍短时微调，能够让模型对本任务领域数据特征更熟悉，理论上之后训练起来也会更为轻松、效果更好

这里取`IMDB`的无监督数据集对模型进行预训练，同样取`10000`条，取法可看上文（这里不需要使用测试集），`IMDB`是文本数据集，相比起来数据量会大很多

```
训练集长度：8000；验证集长度：2000；测试集长度：1000
```

为防止灾难性遗忘原有知识，这里预训练两轮，同时仅需考虑训练集与验证集的损失

```
Epoch 1/2
Train Loss: 0.2289, Acc: N/A, Precision: N/A, Recall: N/A, F1: N/A | Val Loss: 0.0008, Acc: N/A, Precision: N/A, Recall: N/A, F1: N/A
Epoch 2/2
Train Loss: 0.0050, Acc: N/A, Precision: N/A, Recall: N/A, F1: N/A | Val Loss: 0.0005, Acc: N/A, Precision: N/A, Recall: N/A, F1: N/A
```

可以看到训练过后模型对当前领域表现得更为熟悉

后续发现使用全量数据集进行预训练也是可以装得下的，损失也比较合理。这里训练两个`epoch`，不过由于数据量比较多，训练一个`epoch`可能也够了，毕竟第一轮之后损失就已经比较低

```
label
1    0.557826
0    0.442174
Name: proportion, dtype: float64
label
1    0.509174
0    0.490826
Name: proportion, dtype: float64
训练集长度: 40000; 验证集长度: 10000;
```

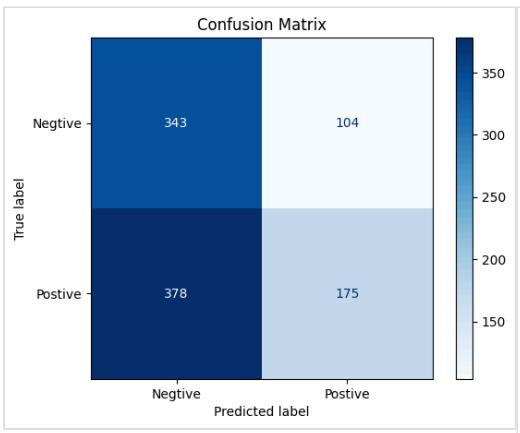
```
Epoch 1/2
Train Loss: 0.0557, Acc: N/A, Precision: N/A, Recall: N/A, F1: N/A | Val Loss: 0.0030, Acc: N/A, Precision: N/A, Recall: N/A, F1: N/A
Epoch 2/2
Train Loss: 0.0031, Acc: N/A, Precision: N/A, Recall: N/A, F1: N/A | Val Loss: 0.0026, Acc: N/A, Precision: N/A, Recall: N/A, F1: N/A
```

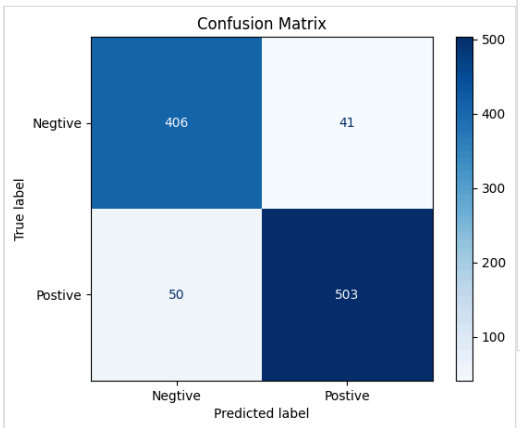
微调

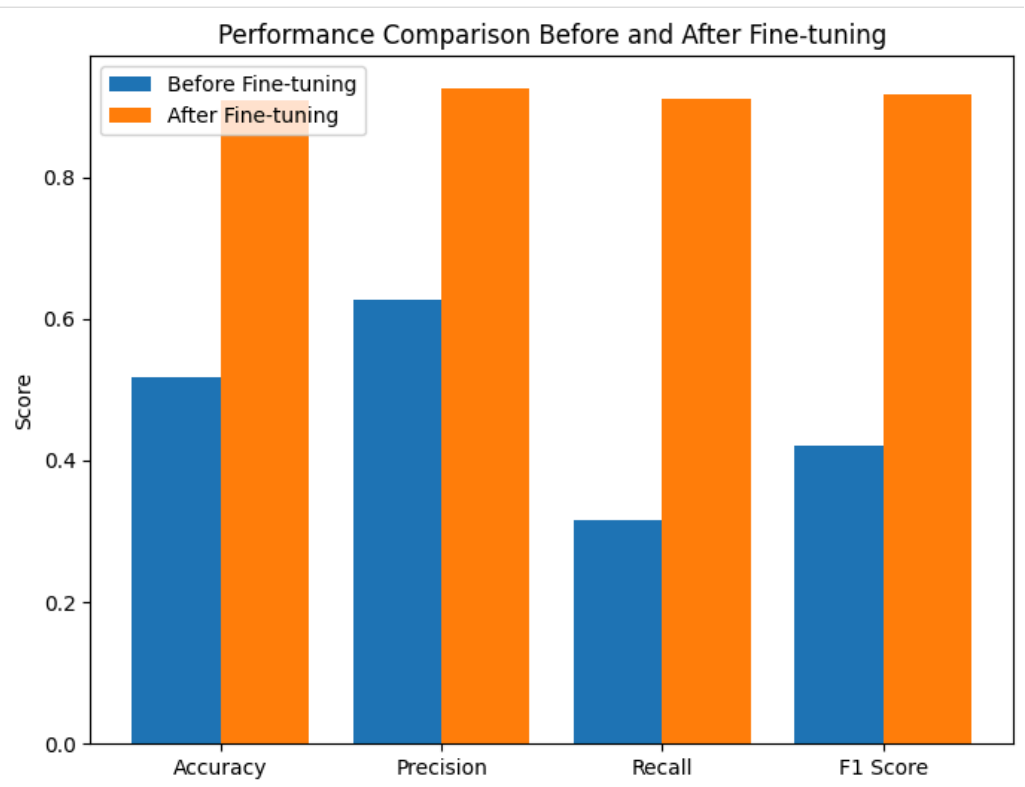
还是使用`SST2`数据集，重新裁剪出前`10000`条数据（注意这里与上面端到端训练使用的数据完全相同，裁剪时都是从前往后取，装载之前再打乱，从标签分布上也可以看出这一点（完全相同））

```
label
1    0.557826
0    0.442174
Name: proportion, dtype: float64
label
1    0.509174
0    0.490826
Name: proportion, dtype: float64
训练集长度: 8000; 验证集长度: 1000; 测试集长度: 1000
```

性能指标	微调前	微调后
Accuracy	0.5180	0.9090
Precision	0.6272	0.9246
Recall	0.3165	0.9096
F1 Score	0.4207	0.9170



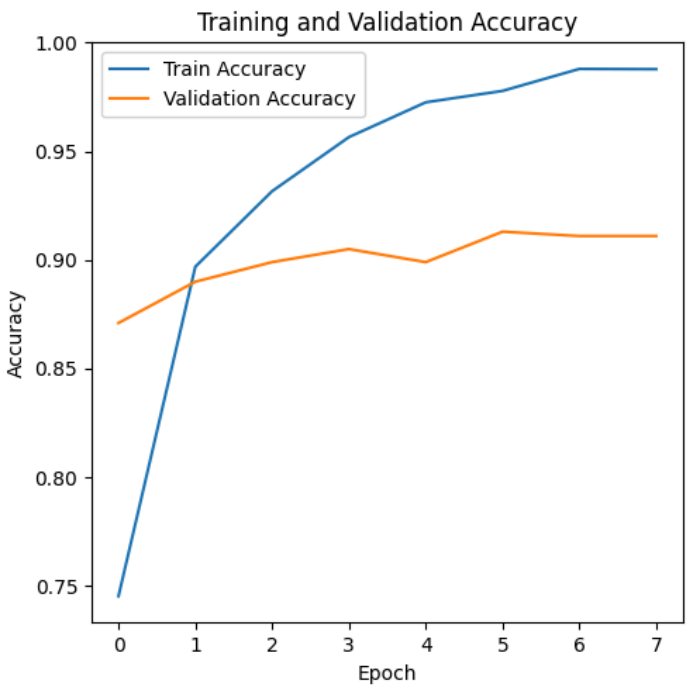


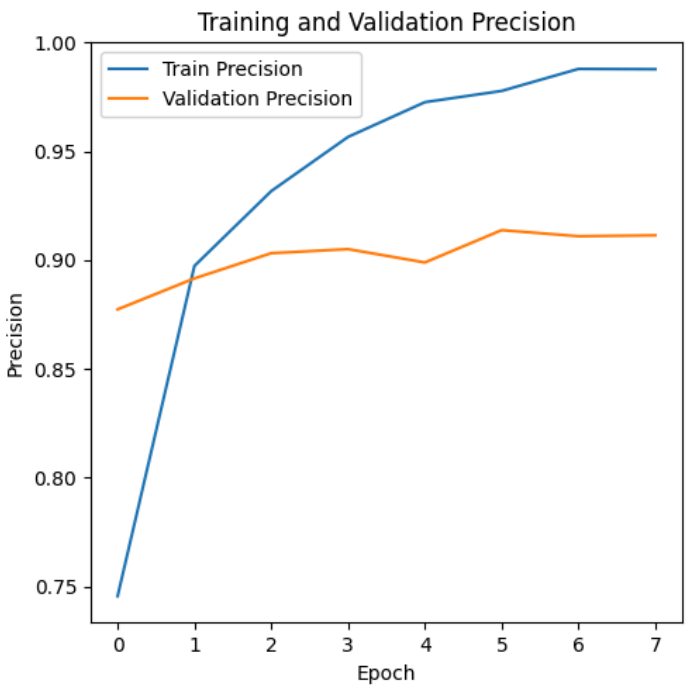


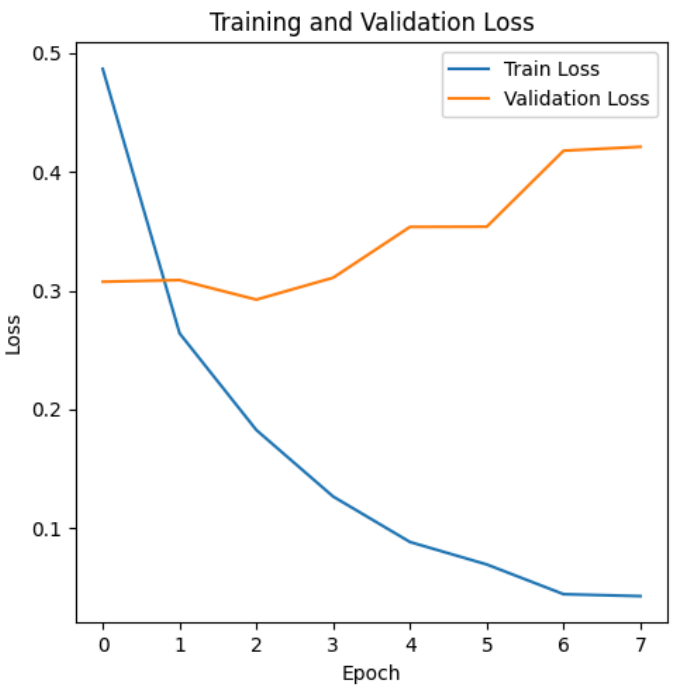
可以看到这里微调同样成功了（并且由于前置预训练数据集`IMDB`中的样本分布比较均匀，这里初始`Recall`的值下降到了比较正常的范围），而且模型各指标在第五个`epoch`之后就开始**呈收敛趋势**，不过由于验证集损失始终在增加触发了早停（这其中也有一些`ACC`值没有继续上升的缘故）

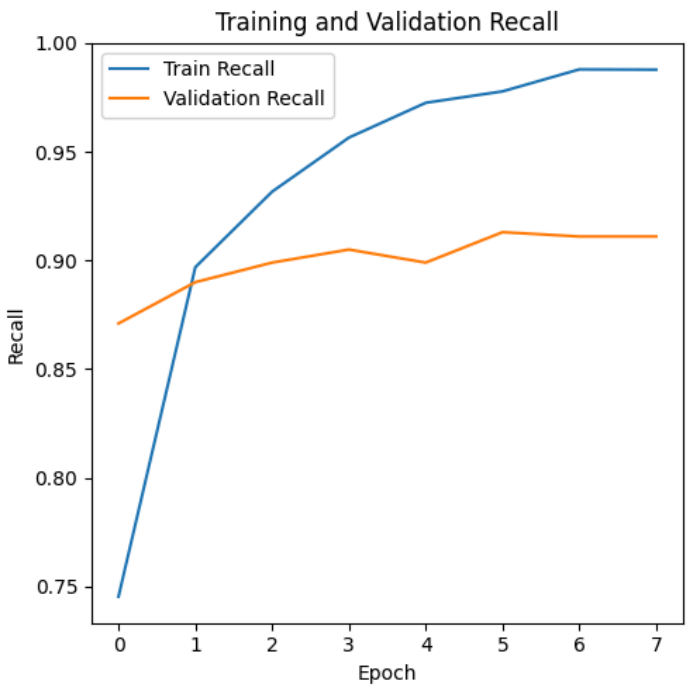
出现这种现象的原因在上面端到端训练的分析中已经提到过了，这里不再赘述

可以看到模型在验证集与测试集上泛化能力基本一致，在验证集上可能略高一些；预训练知识的促进作用在本节开头已经讨论过





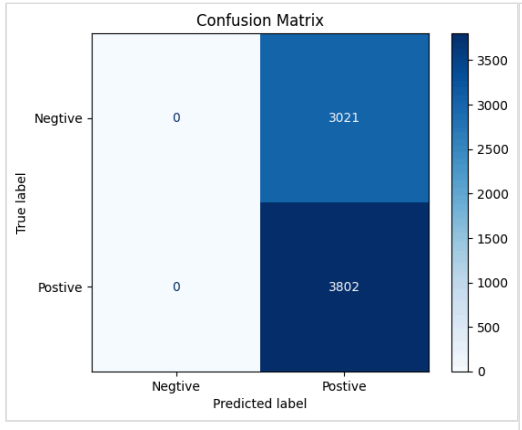


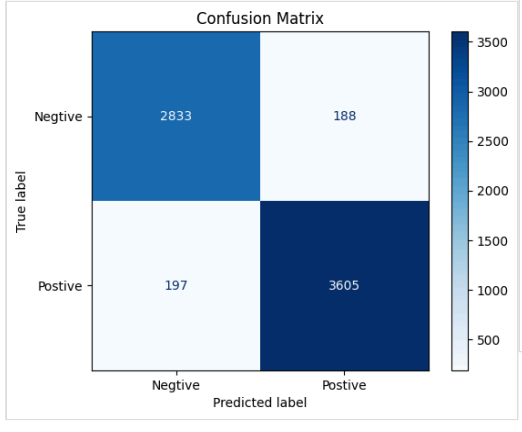


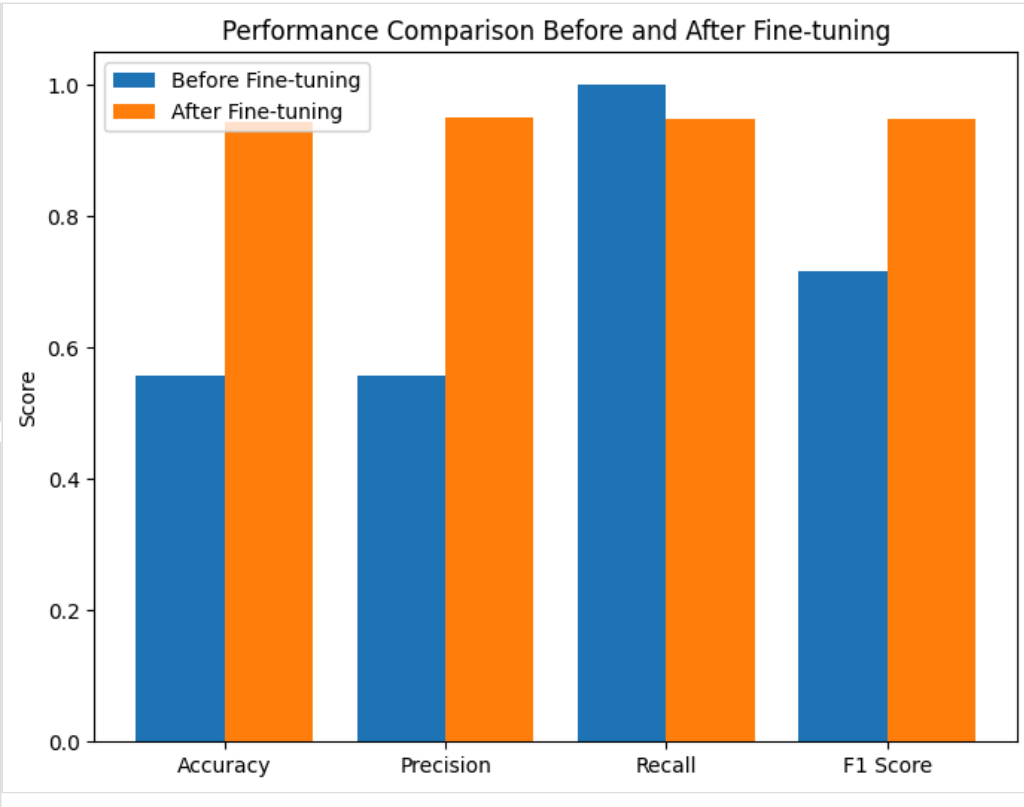
而如果增大数据集，训练几轮后各指标有显著增长，并且可以看到验证集曲线有上移，过拟合趋势减少。这里由于训练时`colab`免费时长达到限制，训练五轮后手动中断了训练，如果继续训练必然能达到更好的结果

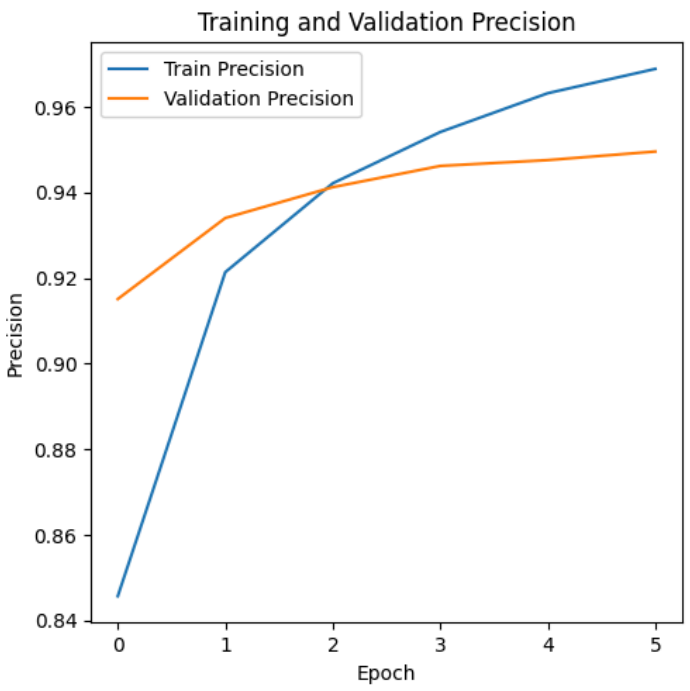
```
label
1    0.557826
0    0.442174
Name: proportion, dtype: float64
label
1    0.509174
0    0.490826
Name: proportion, dtype: float64
训练集长度: 54576; 验证集长度: 6822; 测试集长度: 6823
```

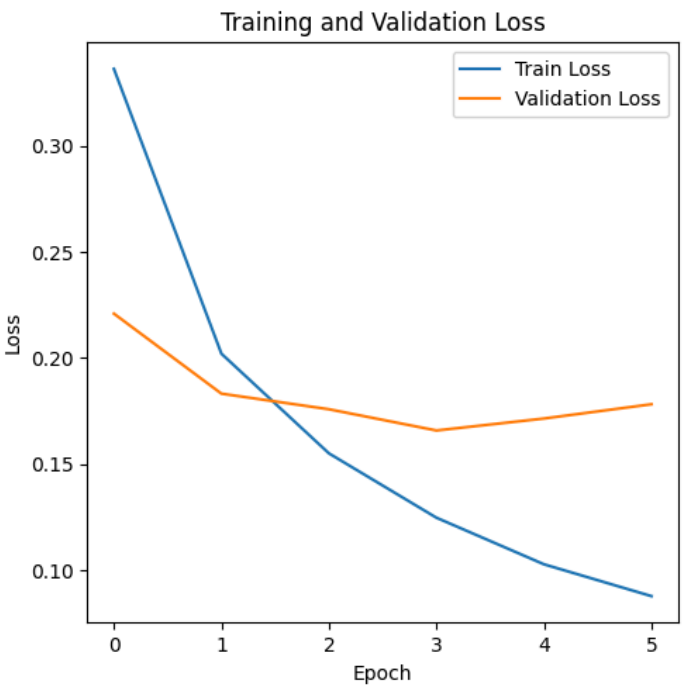
性能指标	微调前	微调后
Accuracy	0.5572	0.9436
Precision	0.5572	0.9504
Recall	1.0000	0.9482
F1 Score	0.7157	0.9493

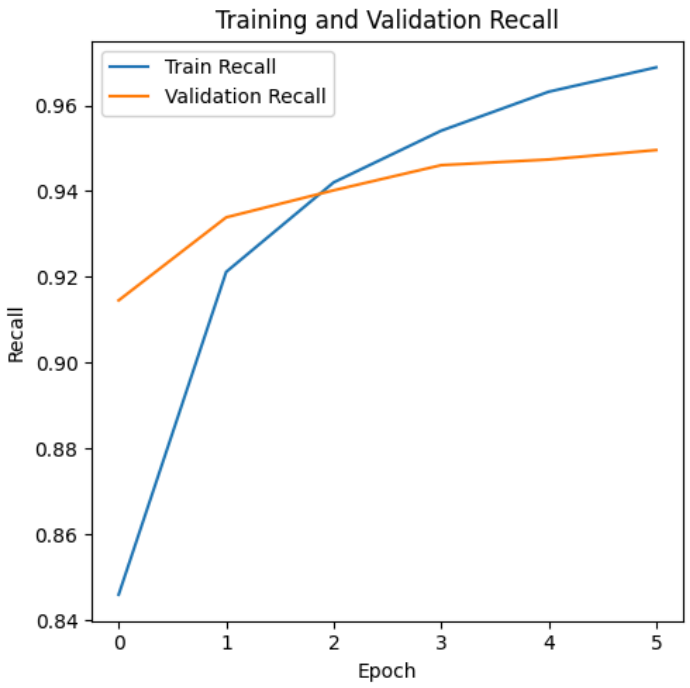


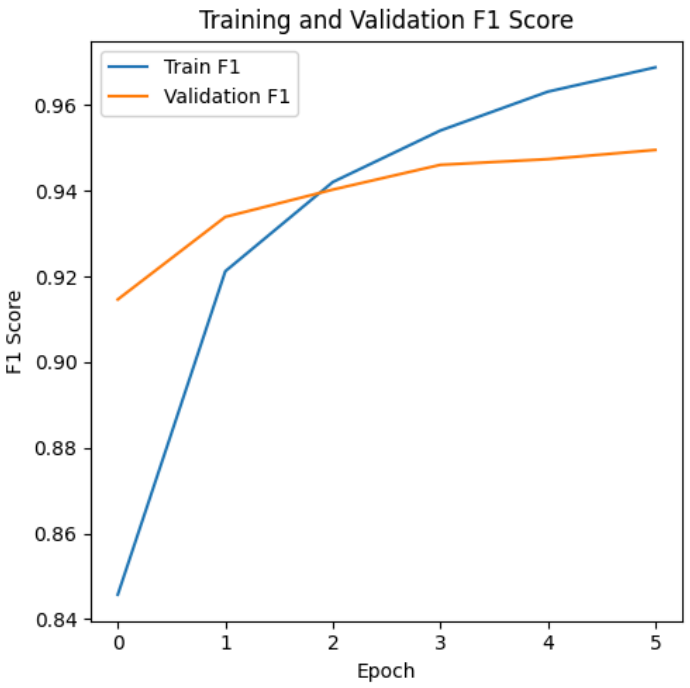












- 正样本：
  - 精确率：$\frac{3605}{3605+188} \approx 0.9504$
  - 召回率：$\frac{3605}{3605+197} \approx 0.9482$
- 负样本：
  - 精确率：$\frac{2833}{2833+197} \approx 0.9350$
  - 召回率：$\frac{2833}{2833+188} \approx 0.9378$

可以看到此处依然更倾向于正样本，理由与上面相同，并且上面对错误案例的分析也已经很详细，此处仅作简单展示

```
使用的测试集大小：6823 | 错误分类样本数量：420
Text: the young stars are too cute
True Label: 1, Pred: 0
--------------------------------------------------
Text: this is an exercise not in biography but in hero worship
True Label: 0, Pred: 1
--------------------------------------------------
Text: that it ca nt help but engage an audience
True Label: 1, Pred: 0
--------------------------------------------------
Text: pulls no punches in its depiction of the lives of the papin sister and the events that led to their notorious rise to infamy
True Label: 1, Pred: 0
--------------------------------------------------
Text: a simpler leaner treatment would have been preferable after all being about nothing is sometimes funnier than being about something
True Label: 0, Pred: 1
--------------------------------------------------
Text: somewhat defuses this provocative theme by submerging it in a hoary love triangle
True Label: 0, Pred: 1
--------------------------------------------------
Text: downbeat periodperfect biopic hammers home a heavyhanded moralistic message
True Label: 1, Pred: 0
--------------------------------------------------
```

```
Text: a perfect example of rancid wellintentioned but shamelessly manipulative movie making
True Label: 0, Pred: 1
--------------------------------------------------
Text: underrated professionals
True Label: 1, Pred: 0
--------------------------------------------------
Text: that breaks your heart
True Label: 0, Pred: 1
--------------------------------------------------
```

对比讨论

收敛与表现

对于预训练后微调的策略，其实也有做全量数据集训练，但是由于资源限制没有能够跑完。下面第一条是`E2E`的，可以看到第一轮就已经触及了效果上限，后续损失不断上升、性能不断下降，最终触发了早停；第二条是预训练后微调的，由于对特定领域数据训练了两轮，因此在初始时刻的性能相对直接在通用数据训练后的权重上更新时稍差一些，但由于其更为熟悉当前领域，损失下降和性能提升也是越来越好的，到第三轮时已经与直接训练的性能相当了，并且其仍能继续提升（观察损失变化可以看出）

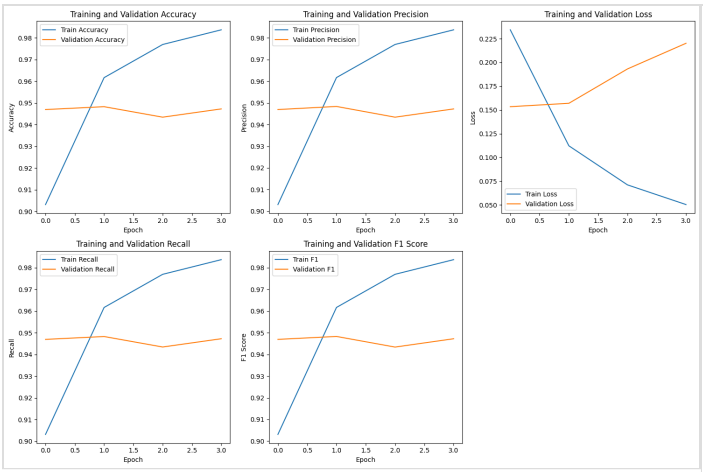
```
Epoch 1/20
Train Loss: 0.2343, Acc: 0.9031, Precision: 0.9031, Recall: 0.9031, F1: 0.9031 |
Val Loss: 0.1534, Acc: 0.9469, Precision: 0.9469, Recall: 0.9469, F1: 0.9469

Epoch 2/20
Train Loss: 0.1122, Acc: 0.9616, Precision: 0.9617, Recall: 0.9616, F1: 0.9617 |
Val Loss: 0.1570, Acc: 0.9483, Precision: 0.9484, Recall: 0.9483, F1: 0.9483

Epoch 3/20
Train Loss: 0.0711, Acc: 0.9769, Precision: 0.9770, Recall: 0.9769, F1: 0.9769 |
Val Loss: 0.1931, Acc: 0.9434, Precision: 0.9434, Recall: 0.9434, F1: 0.9434

[INFO] Early stop
```

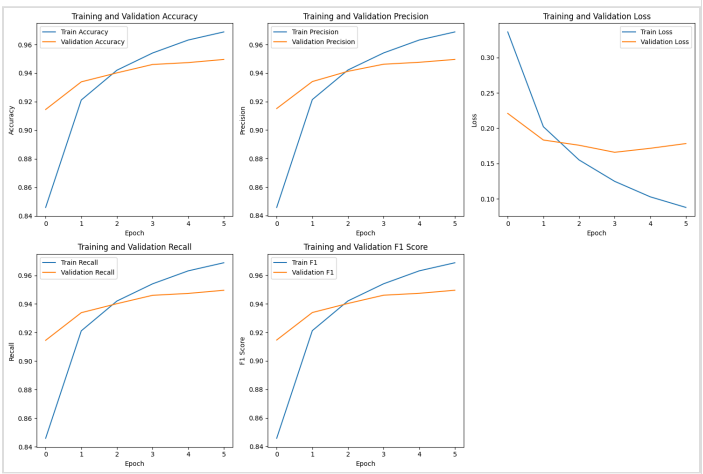
```
Epoch 1/20
Train Loss: 0.3357, Acc: 0.8488, Precision: 0.8487, Recall: 0.8488, F1: 0.8487 |
Val Loss: 0.2207, Acc: 0.9132, Precision: 0.9134, Recall: 0.9132, F1: 0.9133

Epoch 2/20
Train Loss: 0.2056, Acc: 0.9180, Precision: 0.9181, Recall: 0.9180, F1: 0.9180 |
Val Loss: 0.1867, Acc: 0.9330, Precision: 0.9330, Recall: 0.9330, F1: 0.9330

Epoch 3/20
Train Loss: 0.1581, Acc: 0.9403, Precision: 0.9404, Recall: 0.9403, F1: 0.9404 |
Val Loss: 0.1646, Acc: 0.9433, Precision: 0.9435, Recall: 0.9433, F1: 0.9433

KeyboardInterrupt
```

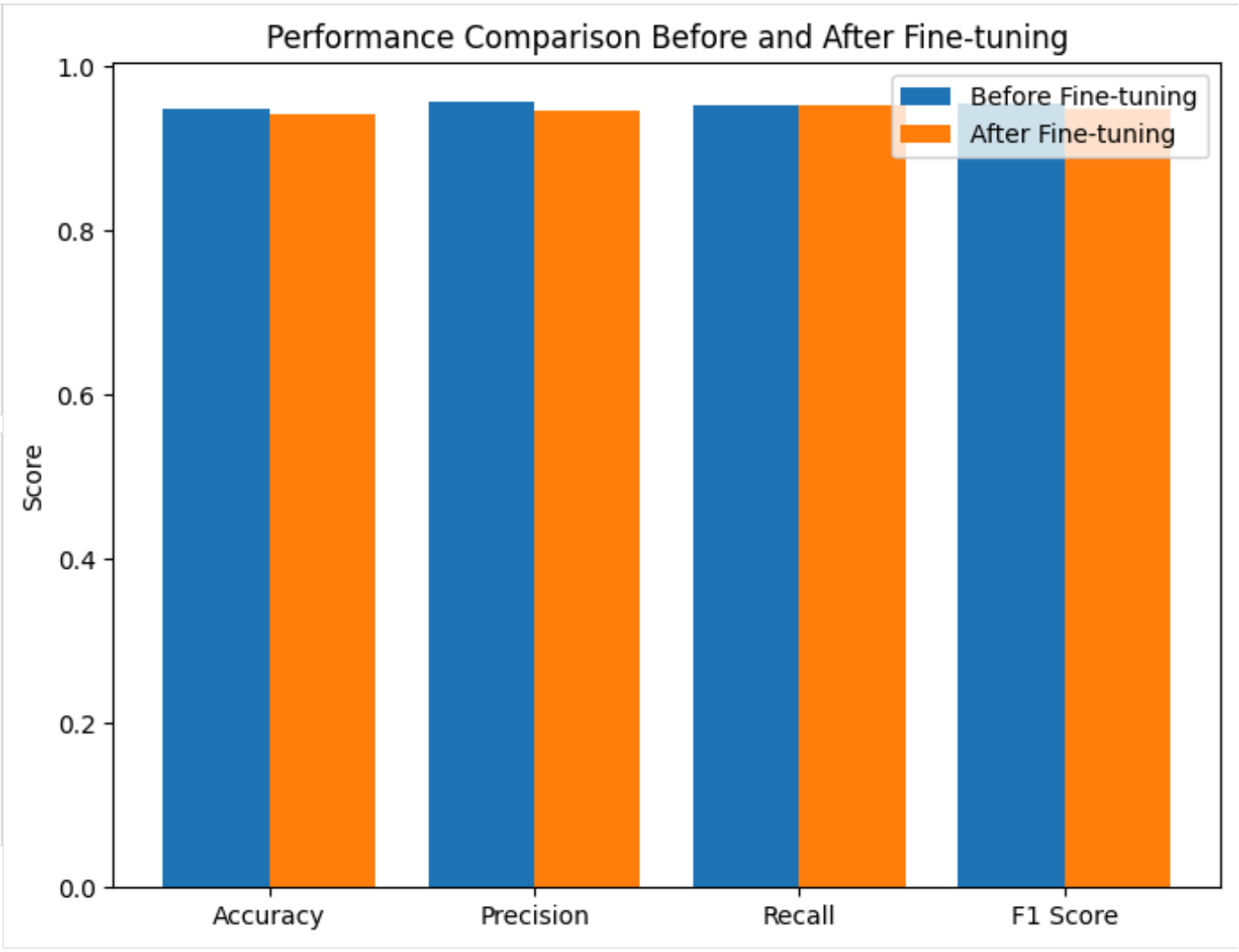
这之后又跑了一遍（不过由于免费资源没有完全恢复也没跑完），对于下面的指标对比图，**蓝色**为端到端策略全程训练指标，**橙色**为先预训练后微调策略指标

可以看到两种策略的指标效果十分接近，先**预训练后微调**策略（左侧下方）略低一些，这是因为`colab`时间限制而**手动中断**了训练过程，从指标曲线与损失曲线看模型此时还**并未达到早停条件**，继续训练是很有可能超过端到端训练的；而**端到端**训练策略（左侧上方）在一开始可能随机到了一个较好的权重更新方向，随后**所有指标都在不断下降，并且以最快的速度触发了早停**

另外这里可能是比较偶然的，因为上面刚训练完进行测试时得到的指标对比应该是预训练后微调的更高，感觉可以解释为两个模型的性能非常相近，测试时都在`94%`此起彼伏，如果预训练后微调的模型继续训练下去大概是能够超过端到端模型的





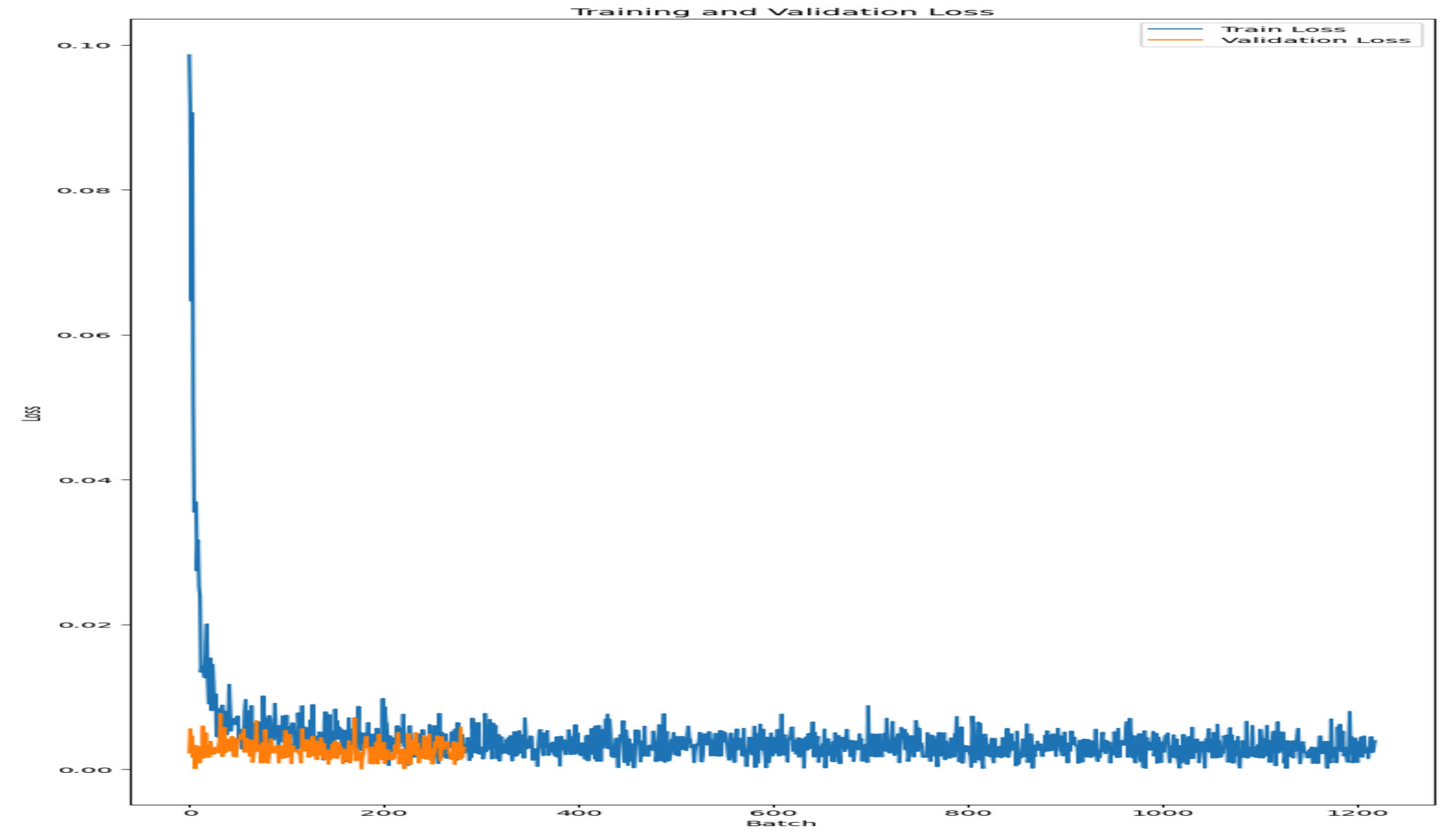


性能指标	端到端	预训练后微调
Accuracy	0.9040	0.9090
Precision	0.9102	0.9246
Recall	0.9168	0.9096
F1 Score	0.9135	0.9170

从图像上看，在相同结构与超参数情况下，**端到端训练策略不收敛，而预训练后微调策略能够收敛**；由于预训练后微调策略没有在全量数据集上跑完，**针对前 `10000` 条数据训练的模型，从上文以及左表展示中可以看到其在测试集上的表现是更优的的**，并且即使是**全量数据集预计也会更优**（其有指标收敛趋势而端到端策略没有），并且由于其微调时使用的是分层学习率，**参数更新起来也更轻松一些**（而且最上面几层似乎是可以直接冻结起来的）

额外预训练中损失曲线的变化与分析

对于 `MLM` 任务，由于训练的轮次较短，这里以 `batch` 为横轴绘图



预训练的`BERT`本身就已经具备了通用语言处理与理解能力，因此在`MLM`环节只需要稍微调整权重即可适应本领域语言特征，同时`MLM`任务相对比较简单，因此可以看到在训练初期损失值就快速下降，最后稳定在了较低值（而非持续下降，这意味着过拟合），成功捕捉到了本领域的特定语言特征，同时也保留了通用数据的特性

实验过程中的问题与对比

正如上文所述，由于`BERT`本身的参数量太大，**过拟合问题在本次实验中是比较明显的，不过也还在可以接受的范围内（`5%`以内）**。数据集样本分布在上文中也有描述：`IMDB`数据集非常均衡，但存在少量重复样本；`SST2`稍微有些差距（`56%`正、`44%`负），但没有重复样本

若需要进一步优化，可以考虑针对上面找到的错误案例手动加入一些数据增强样本（比如针对`but`转折的），也可以引入专门的数据增强措施：同义词/反义词替换、随机插入/删除/交换（在确保句意不变的前提下）、拼写错误模拟（比如`but -> bt`）、情感词替换、句式转换、噪声注入等

还可以在预训练阶段引入对比学习，对一个句子构造出一个近义样本与反义样本，由于我们这次是一个二分类任务，对比学习的效果应该会很好，不过也要注意训练的`epoch`，避免灾难性遗忘通用知识

对于`Transformer`上的文本分类，由于文本数据是离散的，这就需要我们手动将其向量化、序列化，而在序列化时由于文本长度不定还需要进行截断或填充，一般而言都有以下三步（清洗、向量化（构建词表、映射）、序列化（截断、填充）），例如下面是对于`Enron-Spam`数据集的处理：

```
                                              text  label
1  vastar resources , inc . gary , production fro...      0
2  calpine daily gas nomination - calpine daily g...      0
3  re : issue fyi - see note below - already done...      0
4  meter 7268 nov allocation fyi .\n- - - - - - -...      0
5  mcmullen gas for 11 / 99 jackie ,\nsince the i...      0

                                    processed_text  label
0  vastar resources inc gary production high isla...      0
1  calpine daily gas nomination calpine daily gas...      0
2  issue fyi note stella forwarded stella morris ...      0
3  meter 7268 nov allocation fyi forwarded lauri ...      0
4  mcmullen gas 11 99 jackie inlet river plant sh...      0

                                vectorization_text
0  [16387, 439, 43, 923, 319, 126, 3551, 2430, 28...
1  [1915, 383, 16, 1371, 1915, 383, 16, 1371, 688]
2  [309, 774, 247, 5118, 122, 5118, 2925, 6, 3, 3...
3  [257, 55136, 595, 1853, 774, 122, 4505, 1504, ...
4  [15903, 16, 24, 166, 1861, 10571, 2069, 531, 2...

                        padding_vectorization_text
0  [16387, 439, 43, 923, 319, 126, 3551, 2430, 28...
1  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
2  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
3  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
4  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
```

而在本次实验中，后续步骤可以直接通过`BERT`的分词器完成，我们只需进行数据清洗步骤即可

而对基于`CNN`的图片分类，输入的是连续数据（图片），仅需统一尺寸并进行归一化即可，不需要太繁琐的步骤

另外，`Transformer`模型的参数量很大，极其容易过拟合，好在可以使用的预训练模型比较多，找到通用模型拿来微调即可；`CNN`模型一般来说参数不会太大，不过也可能出现过拟合（比如网络太深数据太少），不过也有可以微调的预训练模型（如`ResNet`）

再有一点就是计算资源上的区别，正如上一段所述，两者的参数区别很大，后者（`CNN`）训练占用的资源会比少很多。不过前者也可以通过限制截断长度来减少计算量，也可以引入混合精度训练（当然这个对两者都有效）

总之，两者都需要对数据进行预处理，并且都需要注意过拟合情况。不同在于，`Transformer`更关注上下文等全局信息，参数量更大，依赖于注意力机制；`CNN`更倾向于捕捉局部特征，参数量更小，依赖于卷积层

拓展部分

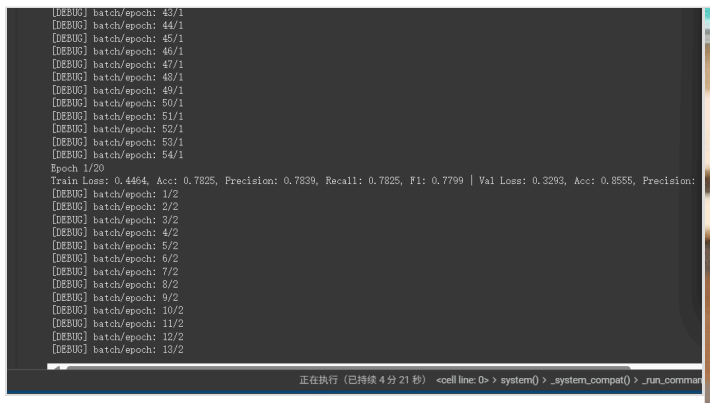
训练技巧-混合精度训练

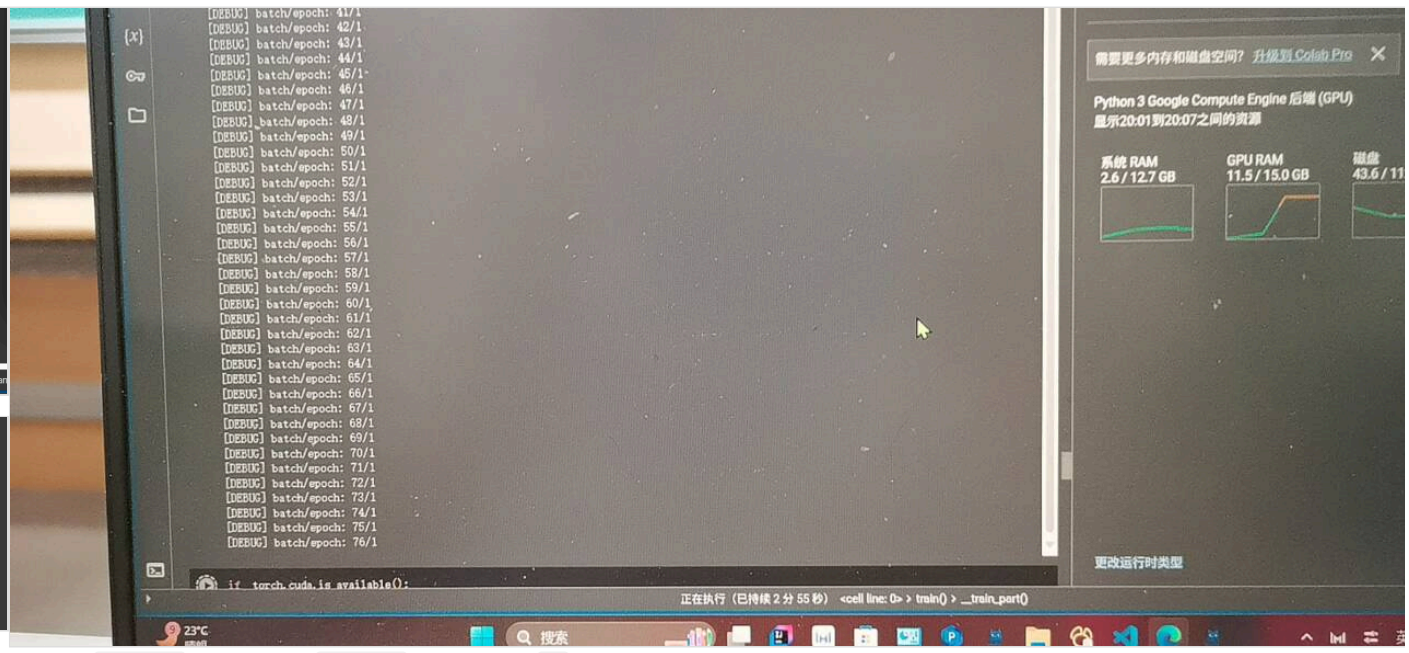
混合精度训练无需额外引入库或自己编写框架，`torch.amp`提供了`autocast`和`GradScaler`以简化实现

要引入混合精度训练，只需要在前向传播和损失计算外套上一层`with torch.amp.autocast(device_type=self.device.type):`即可，损失的反向传播与优化器的调整逻辑都移交给`scaler`自动进行，不过需要特别注意的是在训练部分需要对梯度进行特别处理（在裁剪梯度之前需要先将其反缩放回原始值）

```
self.optimizer.zero_grad()
#loss.backward()
self.scaler.scale(loss).backward()  # 反向传播前对loss乘一个缩放因子，避免梯度过小导致的下溢
self.scaler.unscale_(self.optimizer)  # 反缩放，梯度裁通常是基于原始梯度值进行的
# 梯度裁剪 <-- 防止爆炸、稳定训练、加速收敛
torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)
#self.optimizer.step()
self.scaler.step(self.optimizer)
self.scaler.update()  # 更新缩放因子，为下次迭代做准备
self.scheduler.step()
```

一开始做的时候没有想到记录训练消耗，不过好在拍了一张图（右），这个是在截断长度和`batch_size`都为`128`时的训练过程（预训练后微调的微调步骤），可以看到`76`个`batch`（每轮训练共`427`个`batch`）耗时约`3`分钟，一个`epoch`耗时大致为`20`分钟，且占用显存`11.5GB`



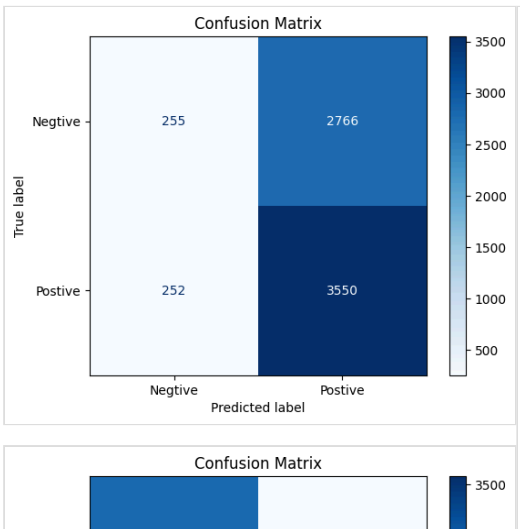


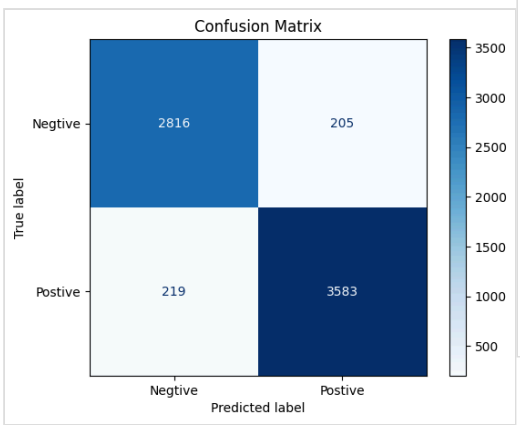
而在使用混合精度的训练策略后（左），显存占用降低到`8.2GB`，且一个`epoch`训练仅需`4`分钟，训练时间与训练显存占用大大降低

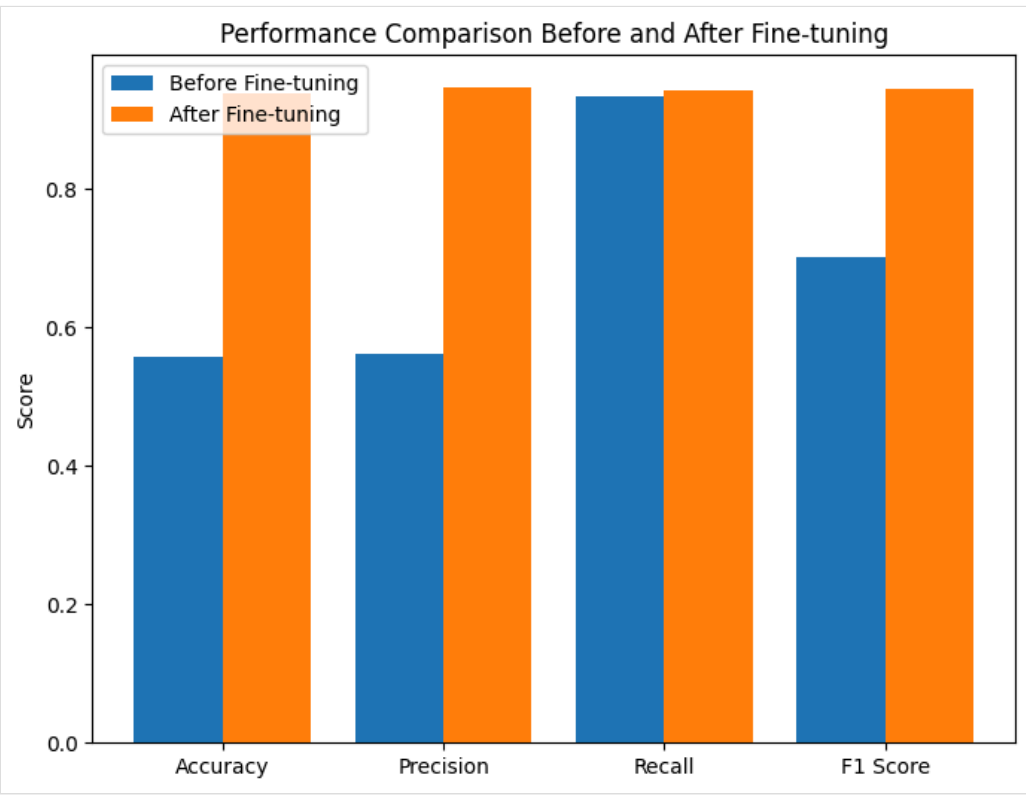
```
Elapsed Time: 5042.40 seconds
Memory Used: 7697.48 MB
```

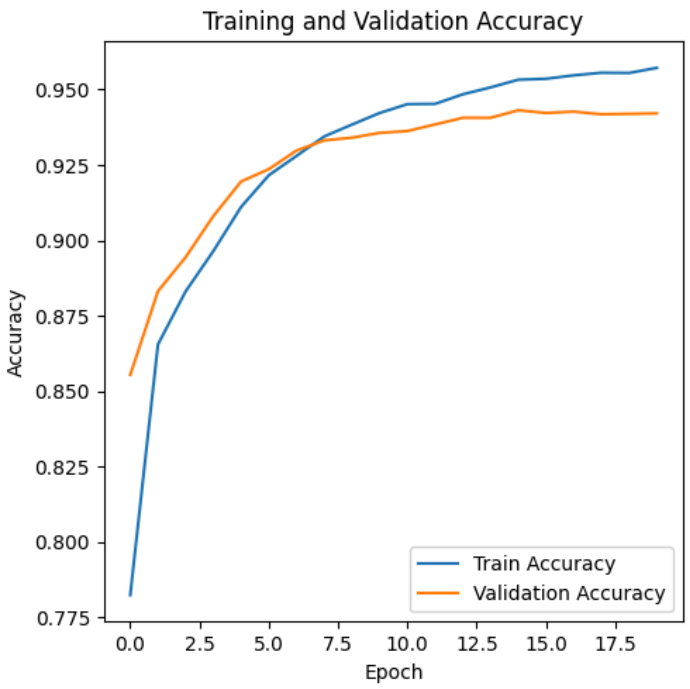
并且模型依然可以正常收敛，整体趋势及最后性能与修改前区别不大（甚至各指标变化趋势较之前更好），准确率与精确率都保持在了`94%`左右，并且训练速度和训练显存占用大大降低

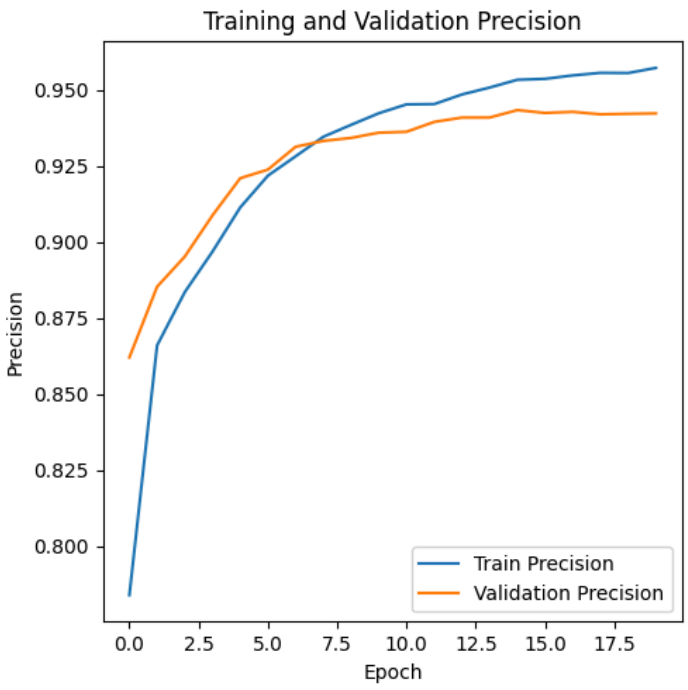
性能指标	微调前	微调后
Accuracy	0.5577	0.9379
Precision	0.5621	0.9459
Recall	0.9337	0.9424
F1 Score	0.7017	0.9441

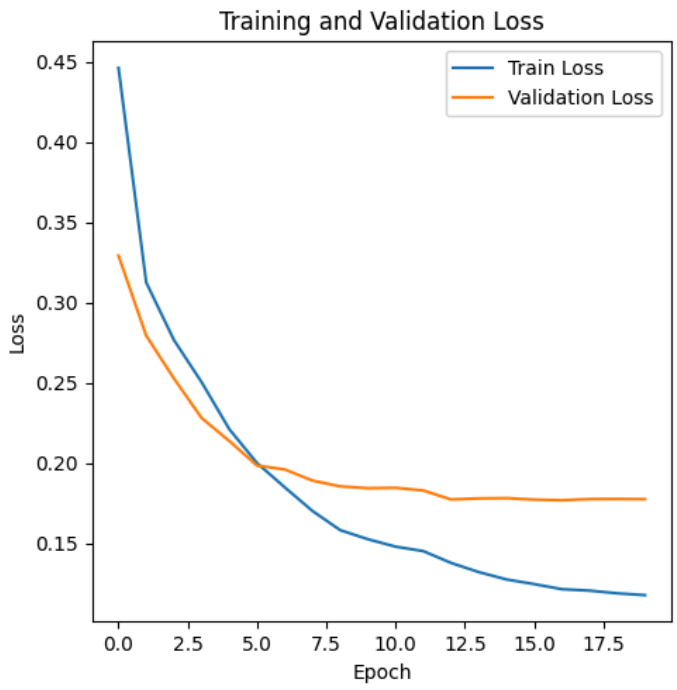


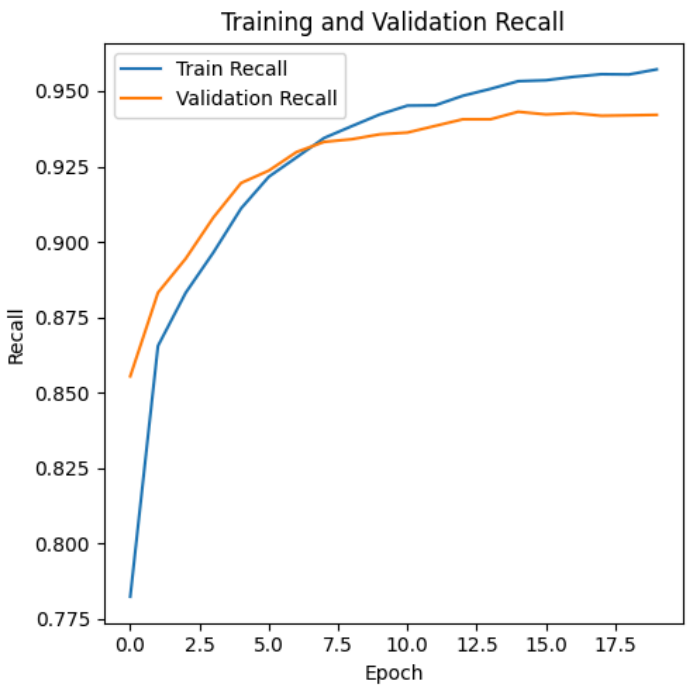


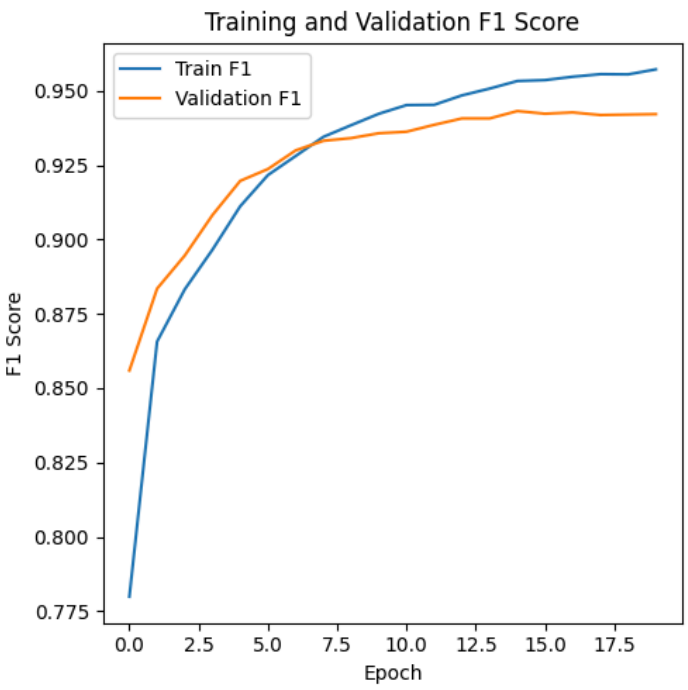












并且正负样本的分类效果也与上面引入之前相同

- 正样本：
  - 精确率：$\frac{3583}{3583+205} \approx 0.9459$
  - 召回率：$\frac{3583}{3583+219} \approx 0.9424$
- 负样本：
  - 精确率：$\frac{2816}{2816+219} \approx 0.9278$
  - 召回率：$\frac{2816}{2816+205} \approx 0.9321$

由于这里仅仅是引入了混合精度计算，错误案例/原因以及其它内容还是与上文相同的，这里不再赘述，并且再将本部分引入端到端训练的效果也是相同的

场景落地-API 封装与服务提供

这一部分实际上就是在模型与外界之间加上一个中间层，外界提供输入后，中间层对输入句子进行清洗并使用模型对应的分词器进行预处理，接着再把经过预处理的数据传给模型，得到输出后再返回给调用者即可

```python
class BertSentence:
    def __init__(self, model_path, tokenizer_path):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.tokenizer = BertTokenizer.from_pretrained(tokenizer_path)
        self.model = BertClassifier()
        self.model.load_state_dict(torch.load(model_path, map_location=self.device))
        self.model.to(self.device)
        self.model.eval()
        print("[SERVER] 模型初始化完成")

    def process(self, sentences):
        if isinstance(sentences, str):
            sentences = [sentences]  # 如果是单条文本，转为列表
        sentences = [self.__clean_text(sentence) for sentence in sentences]
        print(f"[SERVER] 队列长度: {len(sentences)}")
        return self.__classify(sentences)

    def __classify(self, sentences):
        inputs = self.tokenizer(
            sentences,
            return_tensors="pt",
            truncation=True,
            padding="max_length",
            max_length=128,
        )
        inputs.pop("token_type_ids", None)  # 去除 token_type_ids
        inputs = {key: val.to(self.device) for key, val in inputs.items()}

        # 把预处理数据传入模型
        with torch.no_grad():
            outputs = self.model(**inputs)
            labels = torch.argmax(outputs, dim=1)

        # 返回每条句子的分类结果
        return [label.item() for label in labels]

    def __clean_text(self, text):
        text = re.sub(r"<.*?>", "", text)  # 去除HTML标签
        text = text.translate(str.maketrans("", "", string.punctuation))  # 去除标点符号
        text = text.lower()  # 转为小写（uncased模型需要）
        text = re.sub(r"\s+", " ", text).strip()  # 去除多余空格
        return text
```

调用者在实例化之后可以一次传入一条或多条句子并获取分类结果

```python
classifier = BertSentence(MODEL_E2E_NAME, MODEL_PATH)
sentences = [
    "This is a great cource!",
    "I didn't like it.",
    "sdxxylysj is really fantastic!",
]
results = classifier.process(sentences)
for i, label in enumerate(results):
    print(f"{sentences[i]}: 分类结果: {label}")
```

而如果想在当前基础上对外提供服务，一个简单可行的方法是引入`flask`框架（`SpringBoot`有点大材小用），之后再套一个中间层函数用来提供`API`传入`JSON`数据与模型所需实际数据之间的转换即可，如果还有别的需要也可以再套中间件，之后只需要向本机地址的`query`路由发送`POST`请求（可以使用任何形式，`curl`、`python request`...）即可获取分类结果

```python
app = Flask(__name__)
print("加载模型和分词器...")
classifier = BertSentence(MODEL_E2E_NAME, MODEL_PATH)


@app.route("/query", methods=["POST"])
def query():
    try:
        data = request.json
        sentences = data.get("sentences", [])
        if not sentences:
            return jsonify({"error": "No sentences provided"}), 400

        results = classifier.process(sentences)
        return jsonify({"results": results})
    except Exception as e:
        return jsonify({"error": str(e)}), 500


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

例如下面是一次运行结果：

```
PS D:\Study_Work\Electronic_data\CS\AAAUniversity\sdxxylysj\Lab2\extra> python rq.py
{'results': [1, 0]}
```

```
加载模型和分词器...
[SERVER] 模型初始化完成
 * Serving Flask app 'deploy'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.43.166:5000
Press CTRL+C to quit
[SERVER] 队列长度: 2
127.0.0.1 - - [27/Apr/2025 17:08:57] "POST /query HTTP/1.1" 200 -
```

当然，如果想再实用一些，还可以为其编写浏览器/客户端界面，但这个就有点太前端而且有些偏离本课程了，这里省略

> 这一部分代码可以直接复制运行，记得加上库和模型结构，就不附带在代码文件中了

对于模型服务化，公网访问问题很好解决，借助`cloudflared`即可（稳定且免费，可以穿过学校`vlab`，支持国内外访问），延迟是肯定有的，不过由于本次是文本分类，传输几个字符串肯定比传输图片更快，因此**网络延迟这一部分基本可以忽略，主要限制就在模型部署端的处理能力（资源）上**：模型一次只能接受一个批次，那一个线程就只能处理一个请求，即使架构做得很好，几个线程作为请求池其它线程拿来跑，同时能处理的请求数目也是有限的。这种情况下服务端可以主动对请求进行毫秒级的延迟（~~也可以用秒级延迟然后请用户看几个广告~~），以便收集到足够多的单批次并行输入以降低线程爆满概率，但这样依旧有局限性，线程总是会满的，最好的解决办法就是增大可用线程数目（~~加钱~~）

总结

当前模型在准确率与精确率上的性能基本稳定在了`94%`，并且先预训练后微调的训练策略可以减小过拟合并使得验证集指标曲线收敛（与之相对的是端到端训练策略不收敛）。接下来可以在预训练阶段引入对比学习，并在微调时启用数据增强，同时继续微调各层学习率等超参数（包括预训练步骤的学习率、`epoch`等），相信准确率与精确率还能再提升一两个百分点以上

本实验有助于更深入地理解两种训练策略，并很好地介绍了文本分类的常用步骤与`BERT`通用预训练模型，要是有前置再深入进行一下数据预处理步骤就更好了（清洗 → 向量化 → 序列化）