

7.1 BERT

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin Ming-Wei Chang Kenton Lee Kristina Toutanova
Google AI Language

{jacobdevlin, mingweichang, kentonl, kristout}@google.com

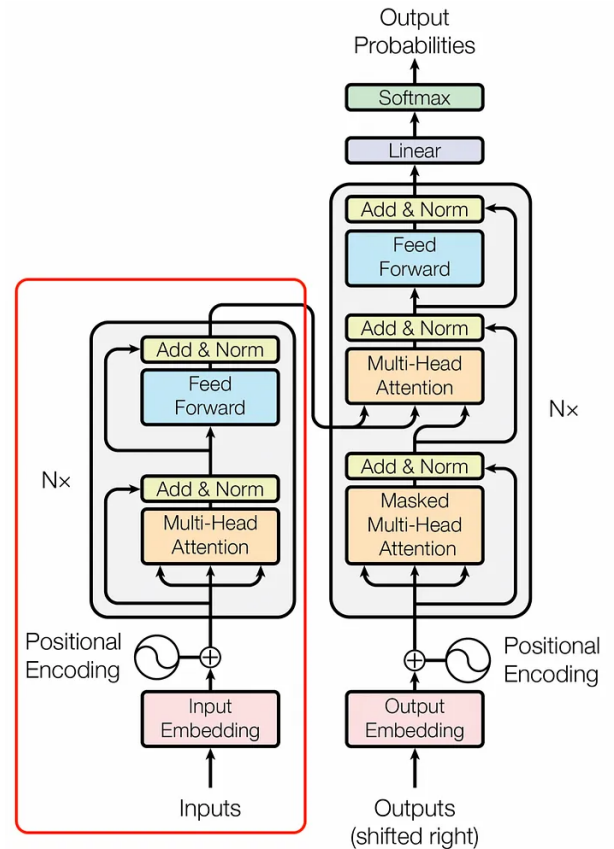
Abstract

We introduce a new language representation model called **BERT**, which stands for **Bidirectional Encoder Representations from Transformers**. Unlike recent language representation models (Peters et al., 2018a; Radford et al., 2018), BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

BERT is conceptually simple and empirically powerful. It obtains new state-of-the-art results on eleven natural language processing tasks, including pushing the GLUE score to 80.5% (7.7% point absolute improvement), MultiNLI accuracy to 86.7% (4.6% absolute improvement), SQuAD v1.1 question answering Test F1 to 93.2 (1.5 point absolute improvement) and SQuAD v2.0 Test F1 to 83.1 (5.1 point absolute improvement).

There are two existing strategies for applying pre-trained language representations to downstream tasks: *feature-based* and *fine-tuning*. The feature-based approach, such as ELMo (Peters et al., 2018a), uses task-specific architectures that include the pre-trained representations as additional features. The fine-tuning approach, such as the Generative Pre-trained Transformer (OpenAI GPT) (Radford et al., 2018), introduces minimal task-specific parameters, and is trained on the downstream tasks by simply fine-tuning *all* pre-trained parameters. The two approaches share the same objective function during pre-training, where they use unidirectional language models to learn general language representations.

We argue that current techniques restrict the power of the pre-trained representations, especially for the fine-tuning approaches. The major limitation is that standard language models are unidirectional, and this limits the choice of architectures that can be used during pre-training. For example, in OpenAI GPT, the authors use a left-to-right architecture, where every token can only attend to previous tokens in the self-attention layers of the Transformer (Vaswani et al., 2017). Such restrictions are sub-optimal for sentence-level tasks,



1 Introduction



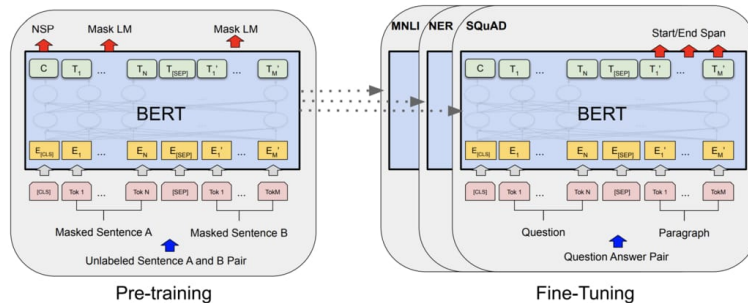
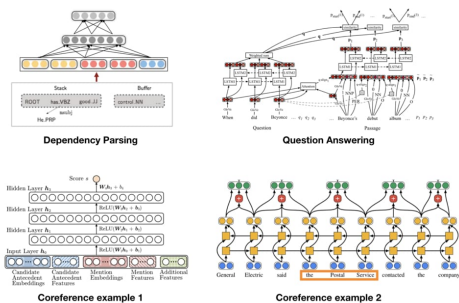
BERT: NLP领域的巨人

在BERT出现之前，很多NLP任务（如情感分析、问答系统等）需要分别设计任务特定的模型，导致开发成本高、迁移能力差。下图是一些例子

如果能有一个模型在大规模的无监督语料上做预训练，先捕获通用的语言知识（听懂人话），再通过微调适配具体任务，从而显著降低开发的复杂性就好了。

人类在理解语言时是双向的，会结合前后文语义来理解当前词的含义。BERT的核心创新是采用了双向编码器，通过Transformer架构实现了对上下文的双向理解，因此人们就想将文本理解和表征工作大一统于BERT。

Various Model Architectures for Different NLP Tasks



BERT介绍

- BERT(Bidirectional Encoder Representation from Transformers)是2018年10月由 Google AI研究院提出的一种预训练模型，该模型在机器阅读理解顶级水平测试SQuAD1.1中表现出惊人的成绩: 全部两个衡量指标上全面超越人类，并且在11种不同NLP测试中创出SOTA表现，包括将GLUE基准推高至80.4% (绝对改进7.6%)，MultiNLI准确度达到86.7% (绝对改进5.6%)，成为NLP发展史上的里程碑式的模型成就。
- BERT的网络架构使用的是《Attention is all you need》中提出的多层Transformer结构。其最大的特点是抛弃了传统的RNN和CNN，通过Attention机制将任意位置的两个单词的距离转换成1，有效的解决了NLP中棘手的长期依赖问题。Transformer的结构在NLP领域中已经得到了广泛应用。

Rank	Model	EM	F1
	Human Performance Stanford University (Rajpurkar & Jia et al. '18)	86.831	89.452
1 Mar 20, 2019	BERT + DAE + AoA (ensemble) Joint Laboratory of HIT and iFLYTEK Research	87.147	89.474
2 Mar 15, 2019	BERT + ConvLSTM + MTL + Verifier (ensemble) Layer 6 AI	86.730	89.286
3 Mar 05, 2019	BERT + N-Gram Masking + Synthetic Self-Training (ensemble) Google AI Language https://github.com/google-research/bert	86.673	89.147
4 May 21, 2019	XLNet (single model) Google Brain & CMU	86.346	89.133
5 Apr 13, 2019	Sem BERT (ensemble) Shanghai Jiao Tong University	86.166	88.886

SQuAD 2.0排行榜的前5名有4个使用BERT



BERT地位，不用多说了吧

📌 BERT框架

- 一个（巨大的）变换器模型编码器（没有解码器）
- 两种模型大小：
 - 基础版：# blocks = 12，隐含大小= 768，# heads = 12，# 参数 = 110M
 - 增强版：# blocks = 24，隐含大小= 1024，# heads = 16，# 参数= 340M
- 使用超过30亿单词的大型语料库（书籍和维基百科）训练

• 模型输入

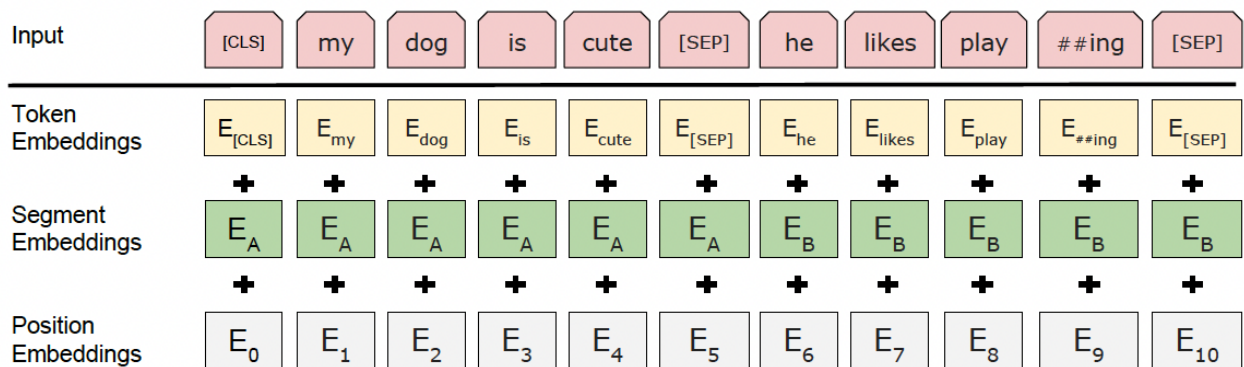
📌 Embedding

Embedding由三种Embedding求和而成：

- Token Embeddings是词向量，第一个单词是CLS标志，可以用于之后的分类任务
- Segment Embeddings用来区别两种句子，因为预训练不光做LM还要做以两个句子为输入的分类任务
- Position Embeddings和之前文章中的Transformer不一样，不是三角函数而是学习出来的

其中 [CLS] 表示该特征用于分类模型，对非分类模型，该符号可以省去。 [SEP] 表示分句符号，用于断开输入语料中的两个句子。

BERT在第一句前会加一个 [CLS] 标志，最后一层该位对应向量可以作为整句话的语义表示，从而用于下游的分类任务等。因为与文本中已有的其它词相比，这个无明显语义信息的符号会更“公平”地融合文本中各个词的语义信息，从而更好的表示整句话的语义。具体来说，self-attention是用文本中的其它词来增强目标词的语义表示，但是目标词本身的语义还是会占主要部分的，因此，经过BERT的12层（BERT-base为例），每次词的embedding融合了所有词的信息，可以去更好的表示自己的语义。而 [CLS] 位本身没有语义，经过12层，句子级别的向量，相比其他正常词，可以更好的表征句子语义。

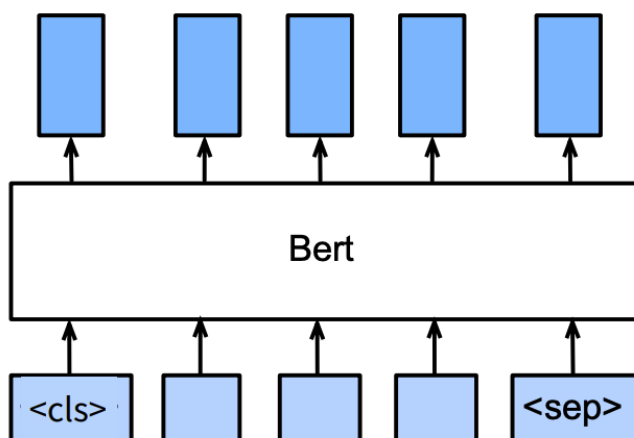


📌 Transformer Encoder 🏠

BERT是用了Transformer的encoder侧的网络。

在Transformer中，模型的输入会被转换成512维的向量，然后分为8个head，每个head的维度是64维，但是BERT的维度是768维度，然后分成12个head，每个head的维度是64维，这是一个微小的差别。Transformer中position Embedding是用的三角函数，BERT中也有一个Position Embedding是随机初始化，然后从数据中学出来的。

BERT模型分为24层和12层两种，其差别就是使用transformer encoder的层数的差异，BERT-base使用的是12层的Transformer Encoder结构，BERT-Large使用的是24层的Transformer Encoder结构。



BERT分词

句子A: I went to the store.句子B: At the store, I bought fresh strawberries.

BERT 用 WordPiece工具来进行分词，并插入特殊的分隔符（ [CLS] ，用来分隔样本）和分隔符（ [SEP] ，用来分隔样本内的不同句子）。

因此实际输入序列为：

[CLS] i went to the store . [SEP] at the store , i bought fresh straw ##berries . [SEP]

GELU作为激活函数替代ReLU

ReLU 的优点是简单、高效，但存在一些问题：

- 导数为 0 的负值区域会导致神经元死亡（Dead Neurons）。
- 在一些任务中，可能不如更复杂的非线性函数表现好。

GELU 的公式

GELU 激活函数的原始定义为：

$$GELU(x) = x \cdot \Phi(x)$$

其中 $\Phi(x)$ 是标准正态分布的累积分布函数，公式为

$$\Phi(x) = \int_{-\infty}^x \frac{e^{-t^2/2}}{\sqrt{2\pi}} dt = \frac{1}{2} \left[1 + \operatorname{erf} \left(\frac{x}{\sqrt{2}} \right) \right]$$

为了提升计算效率，[《A logistic approximation to the cumulative normal distribution》](#)论文中提出了一个近似公式：

$$GELU(x) = \sigma(1.7017449256323682x)$$

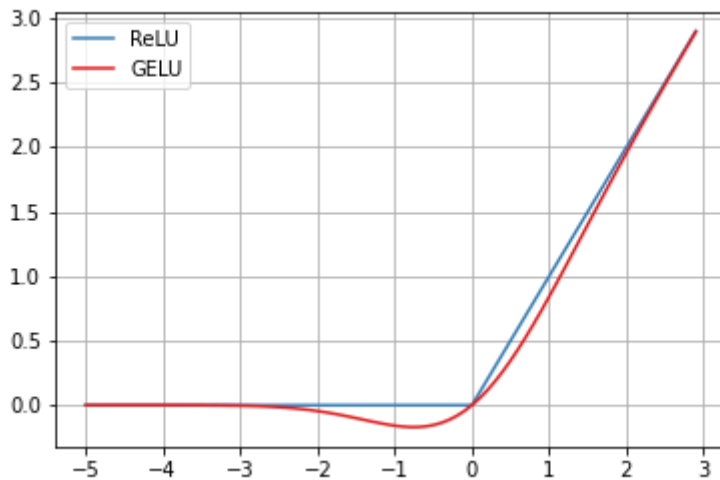
- 近似公式与精确公式的曲线非常接近，大多数情况下近似公式已经足够。

这是一种更平滑的激活函数，相较于 ReLU，其优点在于：

1. **平滑非线性**：GELU 平滑地将输入值映射到 0 和非线性激活区域，更接近自然的神经元激活方式。
2. **对负值的容忍性**：不像 ReLU 会将负值直接截断，GELU 在负值区域也有一定的激活作用。

BERT 论文中提到选择 GELU 是因为在预训练语言模型中表现优于 ReLU 和其他常用激活函数（如 Swish 和 ELU）。

GELU 在处理小输入值时更加灵活，可以更有效地捕获语言数据中复杂的特征关系。



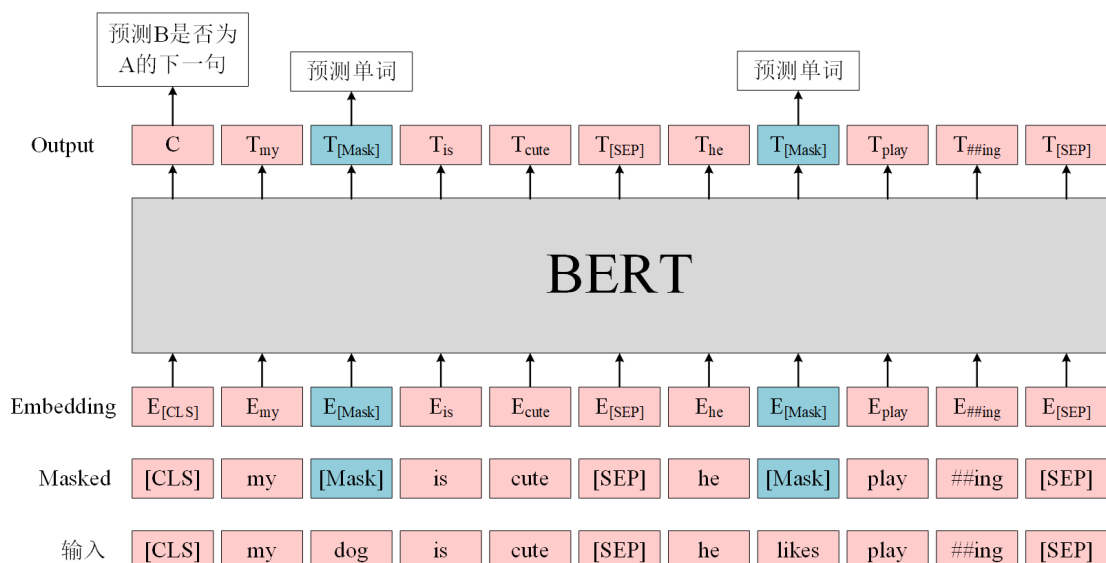
```

1  # transformers/models/bert/modeling_bert.py
2  class BertIntermediate(nn.Module):
3      def __init__(self, config):
4          super().__init__()
5          self.dense = nn.Linear(config.hidden_size, config.intermediate_size)
6          self.intermediate_act_fn = nn.GELU()
7
8      def forward(self, hidden_states):
9          hidden_states = self.dense(hidden_states)
10         hidden_states = self.intermediate_act_fn(hidden_states)
11         return hidden_states

```

BERT的预训练任务

BERT是一个多任务模型，它的预训练（Pre-training）任务是由两个自监督任务组成，即MLM和NSP，如下图所示。



MLM

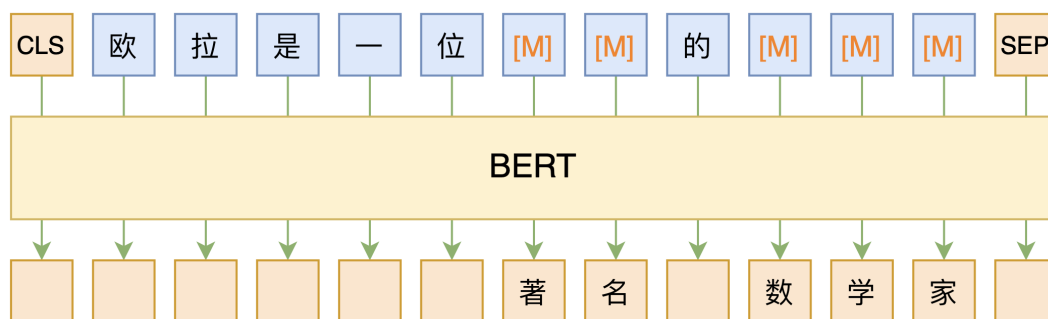
- MLM是指在训练的时候随即从输入语料上mask掉一些单词，然后通过上下文预测该单词，该任务非常像我们在中学时期经常做的完形填空。正如传统的语言模型算法和RNN匹配那样，MLM的这个性质和Transformer的结构是非常匹配的。在BERT的实验中，15%的WordPiece Token会被随机Mask掉。在训练模型时，一个句子会被多次喂到模型中用于参数学习，但是Google并没有在每次都mask掉这些单词，而是在确定要Mask掉的单词之后，做以下处理。
 - 80%的时候会直接替换为 [Mask]，将句子 my dog is cute 转换为句子 my dog is [Mask]。
 - 10%的时候将其替换为其它任意单词，将单词 cute 替换成另一个随机词，例如 apple。将句子 my dog is cute 转换为句子 my dog is apple。
 - 10%的时候会保留原始Token，例如保持句子为 my dog is cute 不变。
- 不是100%Mask掉的原因

这么做的原因是如果句子中的某个Token 100%都会被mask掉，在实际的下游任务中（如问答、分类等），输入数据通常不会包含 [Mask]。如果模型没有在非 [Mask] 的上下文中学会预测单词，就可能表现不佳。

- 加入随机token得原因
 - 随机替换会将目标词替换为一个与上下文语义可能完全无关的词。例如，将句子 My dog is cute 替换为 My dog is apple。
 - 这会让模型面对矛盾或异常的信息，迫使它更加依赖上下文中的其他线索（而不是简单模式匹配）来预测被遮蔽的词汇。
 - 这样增强了模型对上下文的整体理解能力，有助于其在复杂语境中进行语义推理。
- 保持不变token的原因
 - 在预训练阶段，如果目标词总是被遮蔽或替换，而下游任务中目标词通常是完整的（没有 [Mask] 标记），会导致模型在迁移到下游任务时表现不佳。
 - 通过保持10%的目标词不变，模型能够在预训练时学会直接从上下文中预测单词（即使没有显式的 [Mask] 标记），从而更贴合微调阶段的分布。

优点

- 1) 被随机选择15%的词当中以10%的概率用任意词替换去预测正确的词，相当于文本纠错任务，为BERT模型赋予了一定的文本纠错能力；
- 2) 被随机选择15%的词当中以10%的概率保持不变，缓解了finetune时候与预训练时候输入不匹配的问题（预训练时候输入句子当中有mask，而finetune时候输入是完整无缺的句子，即为输入不匹配问题）。



NSP

- Next Sentence Prediction (NSP) 的任务是判断句子B是否是句子A的下文。如果是的话输出 `IsNext`，否则输出 `NotNext`。训练数据的生成方式是从平行语料中随机抽取的连续两句话，其中50%保留抽取的两句话，它们符合`IsNext`关系，另外50%的第二句话是随机从预料中提取的，它们的关系是 `NotNext` 的。这个关系保存在图4中的 `[CLS]` 符号中。。

输入 = [CLS] 我喜欢玩 [Mask] 联盟 [SEP] 我最擅长的 [Mask] 是 亚索 [SEP]

类别 = IsNext

输入 = [CLS] 我喜欢玩 [Mask] 联盟 [SEP] 今天 天气 很 [Mask] [SEP]

类别 = NotNext

在此后的研究（论文《Crosslingual language model pretraining》等）中发现，NSP任务可能并不是必要的，消除NSP损失在下游任务的性能上能够与原始BERT持平或略有提高。这可能是由于Bert以单句子为单位输入，模型无法学习到词之间的远程依赖关系。针对这一点，后续的RoBERTa、ALBERT、spanBERT都移去了NSP任务。

BERT预训练模型最多只能输入512个词，这是因为在BERT中，Token，Position，Segment Embeddings 都是通过学习来得到的。在直接使用Google的BERT预训练模型时，输入最多512个词（还要除掉[CLS]和[SEP]），最多两个句子合成一句。这之外的词和句子会没有对应的embedding。

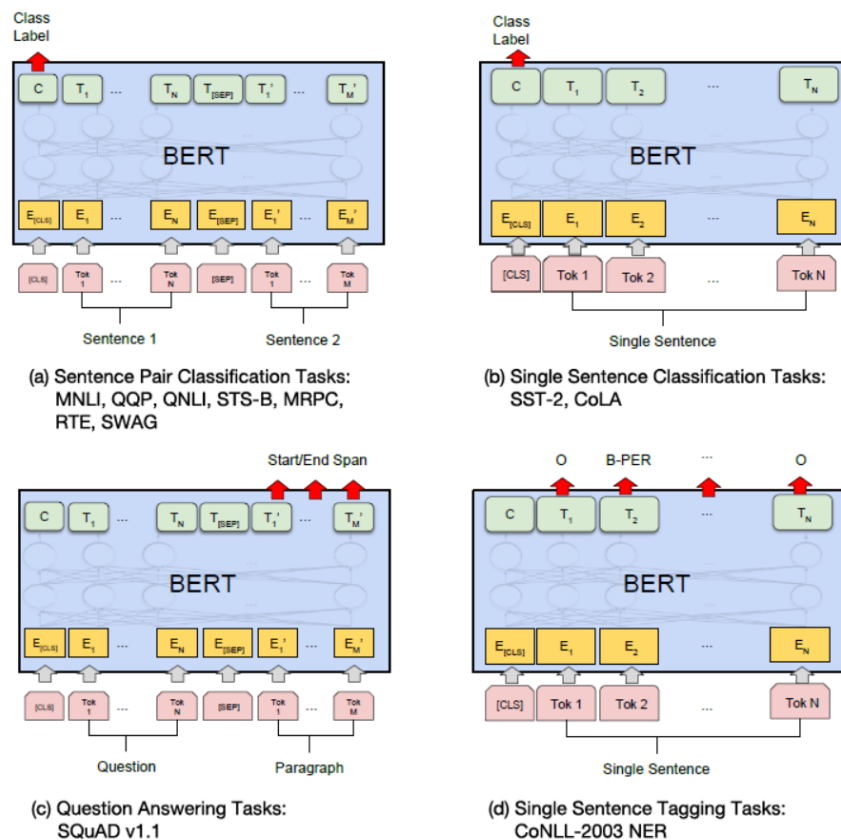
如果有足够的硬件资源自己重新训练BERT，可以更改 BERT config，设置更大 `max_position_embeddings` 和 `type_vocab_size` 值去满足自己的需求

BERT微调

在海量的语料上训练完BERT之后，便可以将其应用到NLP的各个任务中了。微调(Fine-Tuning)的任务包括：基于句子对的分类任务，基于单个句子的分类任务，问答任务，命名实体识别等。

- 基于句子对的分类任务：

- MNLI: 给定一个前提 (Premise)，根据这个前提去推断假设 (Hypothesis) 与前提的关系。该任务的关系分为三种，蕴含关系 (Entailment)、矛盾关系 (Contradiction) 以及中立关系 (Neutral)。所以这个问题本质上是一个分类问题，我们需要做的是去发掘前提和假设这两个句子对之间的交互信息。
- QQP: 基于Quora，判断 Quora 上的两个问题句是否表示的是同样的意思。
- QNLI: 用于判断文本是否包含问题的答案，类似于我们做阅读理解定位问题所在的段落。
- STS-B: 预测两个句子的相似性，包括5个级别。
- MRPC: 也是判断两个句子是否是等价的。
- RTE: 类似于MNLI，但是只是对蕴含关系的二分类判断，而且数据集更小。
- SWAG: 从四个句子中选择为可能为前句下文的那个。
- 基于单个句子的分类任务
 - SST-2: 电影评价的情感分析。
 - CoLA: 句子语义判断，是否是可接受的 (Acceptable) 。
- 问答任务
 - SQuAD v1.1: 给定一个句子（通常是一个问题）和一段描述文本，输出这个问题的答案，类似于做阅读理解的简答题。
- 命名实体识别
 - CoNLL-2003 NER: 判断一个句子中的单词是不是Person, Organization, Location, Miscellaneous或者other（无命名实体）。



BERT代码实践

从0到1

```

1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4
5  # BERT输入嵌入层, 包括Token、Position和Segment嵌入
6  class BertEmbedding(nn.Module):
7      def __init__(self, vocab_size, embed_size, max_len, type_vocab_size):
8          """
9          初始化BERT嵌入层
10         :param vocab_size: 词汇表大小
11         :param embed_size: 嵌入维度
12         :param max_len: 最大序列长度
13         :param type_vocab_size: 类型词汇表大小 (如区分句子A和B)
14         """
15         super(BertEmbedding, self).__init__()
16         self.token_embeddings = nn.Embedding(vocab_size, embed_size) # Token
17         self.position_embeddings = nn.Embedding(max_len, embed_size) # 位置嵌
18         self.segment_embeddings = nn.Embedding(type_vocab_size, embed_size) # Segment嵌入
19
20     def forward(self, tokens, positions, segment_ids):
21         token_embeddings = self.token_embeddings(tokens)
22         position_embeddings = self.position_embeddings(positions)
23         segment_embeddings = self.segment_embeddings(segment_ids)
24         embeddings = token_embeddings + position_embeddings + segment_embeddings
25         return embeddings

```

```

18         self.segment_embeddings = nn.Embedding(type_vocab_size, embed_size)
19         # Segment嵌入
20         self.layer_norm = nn.LayerNorm(embed_size) # 层归一化
21         self.dropout = nn.Dropout(0.1) # Dropout防止过拟合
22
23     def forward(self, input_ids, segment_ids):
24         """
25         前向传播
26         :param input_ids: 输入的词ID张量, 形状为(batch_size, seq_len)
27         :param segment_ids: Segment ID张量, 区分句子A和句子B
28         """
29         seq_len = input_ids.size(1) # 获取序列长度
30         position_ids = torch.arange(seq_len, dtype=torch.long,
31 device=input_ids.device) # 生成位置ID
32         position_ids = position_ids.unsqueeze(0).expand_as(input_ids) # 扩展到
33         # 和输入相同的维度
34
35         token_embeds = self.token_embeddings(input_ids) # Token嵌入
36         position_embeds = self.position_embeddings(position_ids) # 位置嵌入
37         segment_embeds = self.segment_embeddings(segment_ids) # Segment嵌入
38
39         embeddings = token_embeds + position_embeds + segment_embeds # 嵌入相
40         # 加
41
42         embeddings = self.layer_norm(embeddings) # 层归一化
43         embeddings = self.dropout(embeddings) # Dropout
44         return embeddings
45
46 # 多头自注意力机制
47 class MultiHeadSelfAttention(nn.Module):
48     def __init__(self, embed_size, num_heads):
49         """
50         初始化多头自注意力机制
51         :param embed_size: 嵌入维度
52         :param num_heads: 注意力头的数量
53         """
54         super(MultiHeadSelfAttention, self).__init__()
55         assert embed_size % num_heads == 0, "嵌入维度必须能被头数整除。"
56
57         self.num_heads = num_heads # 注意力头数
58         self.head_dim = embed_size // num_heads # 每个头的维度
59         self.query = nn.Linear(embed_size, embed_size) # Query线性变换
60         self.key = nn.Linear(embed_size, embed_size) # Key线性变换
61         self.value = nn.Linear(embed_size, embed_size) # Value线性变换
62         self.fc_out = nn.Linear(embed_size, embed_size) # 输出层
63
64     def forward(self, value, key, query, mask):
65         """

```

```

61         前向传播
62         :param value: Value矩阵
63         :param key: Key矩阵
64         :param query: Query矩阵
65         :param mask: 掩码矩阵, 用于屏蔽某些位置
66         """
67         N, seq_len, embed_size = query.size() # 获取输入的形状
68
69         # 计算Query, Key, Value并分头
70         Q = self.query(query).view(N, seq_len, self.num_heads,
self.head_dim).transpose(1, 2)
71         K = self.key(key).view(N, seq_len, self.num_heads,
self.head_dim).transpose(1, 2)
72         V = self.value(value).view(N, seq_len, self.num_heads,
self.head_dim).transpose(1, 2)
73
74         # 计算注意力分数
75         energy = torch.matmul(Q, K.transpose(-1, -2)) / (self.head_dim ** 0.5)
76         if mask is not None:
77             energy = energy.masked_fill(mask == 0, float("-1e20")) # 对掩码位
置填充极小值
78
79         attention = torch.softmax(energy, dim=-1) # 归一化得到注意力权重
80         out = torch.matmul(attention, V).transpose(1, 2).contiguous().view(N,
seq_len, embed_size) # 计算输出
81         out = self.fc_out(out) # 线性变换输出
82         return out
83
84     # Transformer块
85     class TransformerBlock(nn.Module):
86         def __init__(self, embed_size, num_heads, forward_expansion, dropout):
87             """
88             初始化Transformer块
89             :param embed_size: 嵌入维度
90             :param num_heads: 注意力头的数量
91             :param forward_expansion: 前馈网络扩展倍数
92             :param dropout: Dropout概率
93             """
94             super(TransformerBlock, self).__init__()
95             self.attention = MultiHeadSelfAttention(embed_size, num_heads) # 多头
自注意力
96             self.norm1 = nn.LayerNorm(embed_size) # 层归一化
97             self.norm2 = nn.LayerNorm(embed_size) # 层归一化
98
99             # 前馈神经网络 (使用GELU激活函数)
100             self.feed_forward = nn.Sequential(
101                 nn.Linear(embed_size, forward_expansion * embed_size),

```

```

102         nn.GELU(),
103         nn.Linear(forward_expansion * embed_size, embed_size)
104     )
105
106     self.dropout = nn.Dropout(dropout) # Dropout层
107
108     def forward(self, value, key, query, mask):
109         """
110         前向传播
111         """
112         attention = self.attention(value, key, query, mask) # 自注意力输出
113         x = self.dropout(self.norm1(attention + query)) # 残差连接 + 层归一化
114         forward = self.feed_forward(x) # 前馈网络
115         out = self.dropout(self.norm2(forward + x)) # 残差连接 + 层归一化
116         return out
117
118 # BERT模型
119 class BERT(nn.Module):
120     def __init__(self, vocab_size, embed_size, num_layers, num_heads,
121 forward_expansion, max_len, type_vocab_size, dropout):
122         """
123         初始化BERT模型
124         :param vocab_size: 词汇表大小
125         :param embed_size: 嵌入维度
126         :param num_layers: Transformer层数
127         :param num_heads: 注意力头的数量
128         :param forward_expansion: 前馈网络扩展倍数
129         :param max_len: 最大序列长度
130         :param type_vocab_size: Segment嵌入词表大小
131         :param dropout: Dropout概率
132         """
133         super(BERT, self).__init__()
134         self.embedding = BertEmbedding(vocab_size, embed_size, max_len,
type_vocab_size) # 嵌入层
135         self.layers = nn.ModuleList(
136             [
137                 TransformerBlock(
138                     embed_size,
139                     num_heads,
140                     forward_expansion,
141                     dropout
142                 ) for _ in range(num_layers) # 堆叠多个Transformer块
143             ]
144         )
145
146     def forward(self, input_ids, segment_ids, mask):
147         """

```



```

147         前向传播
148         """
149         out = self.embedding(input_ids, segment_ids) # 嵌入层输出
150         for layer in self.layers:
151             out = layer(out, out, out, mask) # 通过每一层Transformer块
152         return out
153
154     # 示例使用
155     vocab_size = 1000
156     embed_size = 768
157     num_layers = 12
158     num_heads = 12
159     forward_expansion = 4
160     max_len = 512
161     type_vocab_size = 2
162     dropout = 0.1
163
164     model = BERT(vocab_size, embed_size, num_layers, num_heads,
165                  forward_expansion, max_len, type_vocab_size, dropout)
166     input_ids = torch.randint(0, vocab_size, (2, 512)) # 批大小2, 序列长度512
167     segment_ids = torch.zeros_like(input_ids)
168     mask = None
169
170     output = model(input_ids, segment_ids, mask)
171     print(output.shape) # 期望输出形状: (2, 512, 768)

```

对于一般模型，都可以在Huggingface找到，关于模型参数，可以看 `config.json` 文件

```

1  {
2      "architectures": [
3          "BertForMaskedLM"
4      ],
5      "attention_probs_dropout_prob": 0.1,
6      "gradient_checkpointing": false,
7      "hidden_act": "gelu",
8      "hidden_dropout_prob": 0.1,
9      "hidden_size": 768,
10     "initializer_range": 0.02,
11     "intermediate_size": 3072,
12     "layer_norm_eps": 1e-12,
13     "max_position_embeddings": 512,
14     "model_type": "bert",
15     "num_attention_heads": 12,
16     "num_hidden_layers": 12,
17     "pad_token_id": 0,

```

```
18     "position_embedding_type": "absolute",
19     "transformers_version": "4.6.0.dev0",
20     "type_vocab_size": 2,
21     "use_cache": true,
22     "vocab_size": 30522
23 }
```

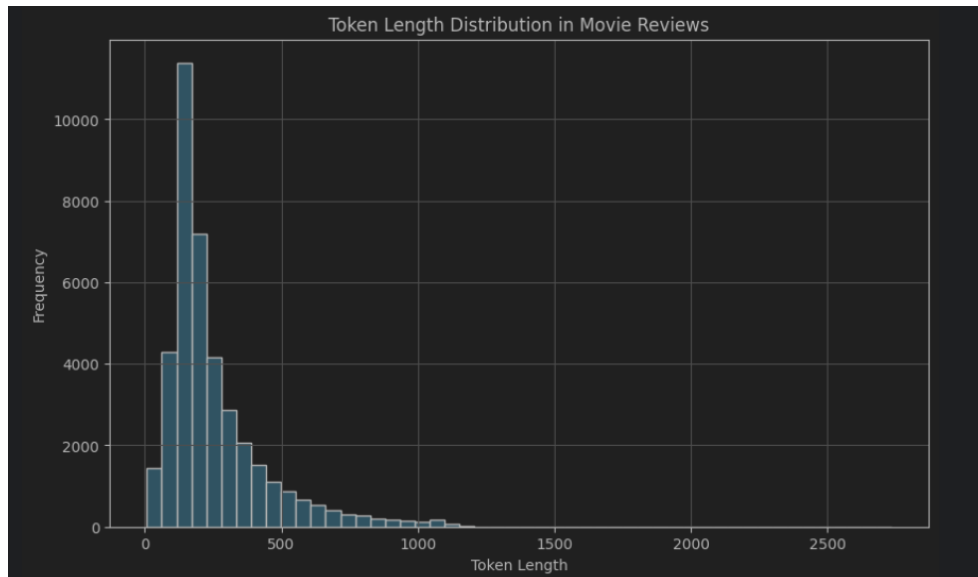
使用预训练BERT完成Kaggle任务：

完成相应包的导入

```
1  import pandas as pd
2  import torch
3  import torch.nn as nn
4  import torch.optim as optim
5  from sklearn.metrics import accuracy_score, classification_report
6  from torch.utils.data import DataLoader
7  from datasets import Dataset as HFDataset # 引入 Hugging Face 的 Dataset
8  from transformers import BertTokenizer
9  from tqdm import tqdm
10 from transformers import BertTokenizer, BertForSequenceClassification
```

```
1  import matplotlib.pyplot as plt
2  # 读取数据集
3  data = pd.read_csv("dsaa-6100-movie-review-sentiment-
classification/movie_reviews/movie_reviews.csv") # 替换为你的数据集路径
4
5  # 加载BERT分词器
6  tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
1  # 对每一条评论进行token化并统计token的数量
2  data['token_length'] = data['text'].apply(lambda x: len(tokenizer.encode(x)))
3
4  # 绘制token长度的分布图
5  plt.figure(figsize=(10, 6))
6  plt.hist(data['token_length'], bins=50, color='skyblue', edgecolor='black')
7  plt.title('Token Length Distribution in Movie Reviews')
8  plt.xlabel('Token Length')
9  plt.ylabel('Frequency')
10 plt.grid(True)
11 plt.show()
```



```
1
2 # 创建 Hugging Face 数据集
3 hf_dataset = HFDataset.from_pandas(data)
4 # 定义最大序列长度
5 max_len = 512 # 将max_len调整为512
6 # 定义分词函数
7 def tokenize_function(example):
8     return tokenizer(
9         example["text"],
10        max_length=max_len,
11        truncation=True,
12        padding="max_length"
13    )
```

```
1 # 对文本进行分词，并显示进度条
2 print("编码文本中...")
3 tokenized_dataset = hf_dataset.map(tokenize_function, batched=True,
4                                     desc="Tokenizing dataset")
```

编码文本中...

Tokenizing dataset: 100% 40000/40000 [01:41<00:00, 398.47 examples/s]

```
1 print(tokenized_dataset)
2 print(tokenized_dataset['text'][0])
3 print(tokenized_dataset['input_ids'][0])
```

```
Dataset({
    features: ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
    num_rows: 40000
})
```

If you havent seen this movie than you need to It rocks and you have to watch it It is so funny and will make you laugh your guts out so you have to watch it and i saw it about a billion and a half times and still think it is funny so you have to yes i have memorized the whole movie and could quote it to you from start to finish you must see this move it is also cute because it is half a chick flick if you dont watch it then you are really missing outthis movie even has cute guys in it and that is always a bonus so in summary watch the movie now and trust me you will not be making a mistake did i mention the music is good too So you should like it if you enjoy music This is a movie that they rated correctly and it will work for anyone

```
[101, 2065, 2017, 4033, 2102, 2464, 2023, 3185, 2084, 2017, 2342, 2000, 2009, 5749, 1998, 2017, 2031, 2000, 3422,
2009, 2009, 2003, 2061, 6057, 1998, 2097, 2191, 2017, 4756, 2115, 18453, 2041, 2061, 2017, 2031, 2000, 3422, 2009,
1998, 1045, 2387, 2009, 2055, 1037, 4551, 1998, 1037, 2431, 2335, 1998, 2145, 2228, 2009, 2003, 6057, 2061, 2017,
2031, 2000, 2748, 1045, 2031, 24443, 18425, 1996, 2878, 3185, 1998, 2071, 14686, 2009, 2000, 2017, 2013, 2707, 2000,
3926, 2017, 2442, 2156, 2023, 2693, 2009, 2003, 2036, 10140, 2138, 2009, 2003, 2431, 1037, 14556, 17312, 2065,
2017, 2123, 2102, 3422, 2009, 2059, 2017, 2024, 2428, 4394, 2041, 15222, 2015, 3185, 2130, 2038, 10140, 4364, 1999,
2009, 1998, 2008, 2003, 2467, 1037, 6781, 2061, 1999, 12654, 3422, 1996, 3185, 2085, 1998, 3404, 2033, 2017, 2097,
2025, 2022, 2437, 1037, 6707, 2106, 1045, 5254, 1996, 2189, 2003, 2204, 2205, 2061, 2017, 2323, 2066, 2009, 2065,
```

```
1  # 转换为 PyTorch 张量
2  tokenized_dataset.set_format(type="torch", columns=["input_ids",
    "attention_mask", "label"])
3  tokenized_dataset
```

```
Out 7  Dataset({
    features: ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
    num_rows: 40000
})
```

```
1  # 划分训练集和测试集
2  print("划分数据集中...")
3  train_test_split = tokenized_dataset.train_test_split(test_size=0.2, seed=42)
4  train_dataset = train_test_split["train"]
5  test_dataset = train_test_split["test"]
6  train_dataset
```

```
Dataset({
    features: ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
    num_rows: 32000
})
```

```
1  # 使用 DataLoader 加载数据
2  train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
3  test_loader = DataLoader(test_dataset, batch_size=16)
```

```
1  # 定义 BERT 分类模型
2  class BertClassifier(nn.Module):
3      def __init__(self, model_name, output_dim):
4          super(BertClassifier, self).__init__()
5          self.bert = BertForSequenceClassification.from_pretrained(model_name,
num_labels=output_dim)
6
7      def forward(self, input_ids, attention_mask):
8          outputs = self.bert(input_ids, attention_mask=attention_mask)
9          return outputs.logits  # 输出分类得分
```

```
1  # 初始化模型
2  print("初始化模型中...")
3  output_dim = 2  # 二分类任务
4  model = BertClassifier('bert-base-uncased', output_dim)  # 使用BERT作为预训练模
型
5
6  # 损失函数和优化器
7  criterion = nn.CrossEntropyLoss()  # 使用交叉熵损失
8  optimizer = optim.Adam(model.parameters(), lr=2e-5)  # BERT的学习率一般较小
```

```
1  model
```



```

BertClassifier(
  (bert): BertForSequenceClassification(
    (bert): BertModel(
      (embeddings): BertEmbeddings(
        (word_embeddings): Embedding(30522, 768, padding_idx=0)
        (position_embeddings): Embedding(512, 768)
        (token_type_embeddings): Embedding(2, 768)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (encoder): BertEncoder(
        (layer): ModuleList(
          (0-11): 12 x BertLayer(
            (attention): BertAttention(
              (self): BertSelfAttention(
                (query): Linear(in_features=768, out_features=768, bias=True)
                (key): Linear(in_features=768, out_features=768, bias=True)
                (value): Linear(in_features=768, out_features=768, bias=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
          )
        )
      )
    )
  )

```

```

1  # 评估模型
2  def evaluate_model(model, test_loader):
3      print("评估模型中...")
4      model.eval()  # 设置模型为评估模式
5      all_preds = []
6      all_labels = []
7      with torch.no_grad():  # 禁用梯度计算以提高评估效率
8          for batch in tqdm(test_loader, desc="评估中"):
9              input_ids = batch['input_ids'].to(device)  # 获取输入ID
10             attention_mask = batch['attention_mask'].to(device)  # 获取
11             labels = batch['label'].to(device)  # 获取标签
12             outputs = model(input_ids, attention_mask).argmax(dim=-1)  # 获取预
13             all_preds.extend(outputs.cpu().tolist())  # 收集预测结果
14             all_labels.extend(labels.cpu().tolist())  # 收集真实标签
15             accuracy = accuracy_score(all_labels, all_preds)  # 计算准确率
16             print("准确率:", accuracy)
17             print(classification_report(all_labels, all_preds))  # 打印分类报告
18             return accuracy

```

```

1  # 设置设备
2  device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
3  model.to(device)

```

```

1  # 训练模型
2  epochs = 5  # 训练轮数
3  def train_and_evaluate(model, train_loader, test_loader):
4      print("开始训练模型...")
5      global_step = 0  # 用于记录当前的训练步骤 (batch)
6
7      for epoch in range(epochs):
8          model.train()  # 设置模型为训练模式
9          total_loss = 0  # 累计当前 epoch 的损失
10
11         for batch in tqdm(train_loader, desc=f"训练第 {epoch+1} 轮"):
12             optimizer.zero_grad()  # 清零梯度
13             input_ids = batch['input_ids'].to(device)  # 获取输入ID
14             attention_mask = batch['attention_mask'].to(device)  # 获取
attention_mask
15             labels = batch['label'].to(device)  # 获取标签
16
17             outputs = model(input_ids, attention_mask)  # 前向传播
18             loss = criterion(outputs, labels)  # 计算损失
19             loss.backward()  # 反向传播
20             optimizer.step()  # 更新参数
21
22             total_loss += loss.item()  # 累加损失
23
24         # 打印每个 epoch 的总损失
25         print(f"第 {epoch+1}/{epochs} 轮, 损失: {total_loss:.4f}")
26
27         # 每轮训练后进行评估
28         accuracy = evaluate_model(model, test_loader)
29
30         # 每4轮保存一次模型
31
32         torch.save(model.state_dict(),
f"bert_text_classifier_epoch_{epoch+1}.pth")  # 保存模型权重
33         print(f"模型已保存: bert_text_classifier_epoch_{epoch+1}.pth")

```

```

1  train_and_evaluate(model, train_loader, test_loader)

```

开始训练模型...

训练第 1 轮: 100%|██████████| 2000/2000 [25:57<00:00, 1.28it/s]

第 1/5 轮, 损失: 465.3554

评估模型中...

评估中: 100%|██████████| 500/500 [02:12<00:00, 3.77it/s]

准确率: 0.933375

	precision	recall	f1-score	support
0	0.92	0.95	0.93	4019
1	0.95	0.92	0.93	3981
accuracy			0.93	8000
macro avg	0.93	0.93	0.93	8000
weighted avg	0.93	0.93	0.93	8000

模型已保存: bert_text_classifier_epoch_1.pth

训练第 2 轮: 100%|██████████| 2000/2000 [25:56<00:00, 1.29it/s]

第 2/5 轮, 损失: 237.5907

评估模型中...

评估中: 100%|██████████| 500/500 [02:11<00:00, 3.80it/s]

准确率: 0.93125

	precision	recall	f1-score	support
0	0.96	0.90	0.93	4019
1	0.91	0.96	0.93	3981
accuracy			0.93	8000
macro avg	0.93	0.93	0.93	8000
weighted avg	0.93	0.93	0.93	8000

模型已保存: bert_text_classifier_epoch_2.pth

训练第 3 轮: 8%|██████ | 166/2000 [02:08<23:41, 1.29it/s]

BERT.ipynb

bert_text_classifier_epoch_1.pth

bert_text_classifier_epoch_2.pth

```
1 # 加载测试数据
2 test_data = pd.read_csv("dsaa-6100-movie-review-sentiment-
classification/test_data.csv")
3 # 创建 Hugging Face 数据集
4 hf_test_dataset = HFDataset.from_pandas(test_data)
5
6 # 对文本进行分词, 并显示进度条
7 print("编码测试数据中...")
8 tokenized_test_dataset = hf_test_dataset.map(tokenize_function, batched=True,
desc="Tokenizing test_dataset")
9
10 # 转换为 PyTorch 张量
11 tokenized_test_dataset.set_format(type="torch", columns=["input_ids",
"attention_mask", "Id"])
12
13 test_loader = DataLoader(tokenized_test_dataset, batch_size=32)
14
15 # 推理并生成结果
16 print("开始推理...")
17 results = []
18 model.eval() # 设置为评估模式
19 with torch.no_grad(): # 禁用梯度计算, 提升推理效率
20     for batch in tqdm(test_loader, desc="推理中"):
21         input_ids = batch['input_ids'].to(device) # 获取输入ID
22         attention_mask = batch['attention_mask'].to(device) # 获取attention
mask
23         ids = batch['Id'] # 获取样本ID
24         outputs = model(input_ids, attention_mask) # 模型预测输出
25         preds = outputs.argmax(dim=-1).cpu().tolist() # 获取类别预测结果
26         results.extend(zip(ids.tolist(), preds)) # 收集预测结果
```

编码测试数据中...

Tokenizing test_dataset: 100% 10000/10000 [00:25<00:00, 394.45 examples/s]

开始推理...

推理中: 100% 313/313 [02:44<00:00, 1.91it/s]

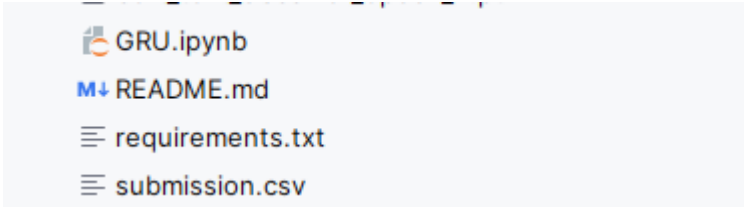
```
1 #%%
2 results[:10]
```

```
[(0, 0),  
(1, 1),  
(2, 0),  
(3, 1),  
(4, 1),  
(5, 1),  
(6, 1),  
(7, 1),  
(8, 1),  
(9, 1)]
```

```
1 # 保存结果  
2 print("保存结果中...")  
3 output_df = pd.DataFrame(results, columns=["Id", "Category"]) # 创建结果数据框  
4 output_df.to_csv("submission.csv", index=False) # 保存结果为CSV文件  
5 print("结果已保存到 submission.csv")
```



保存结果中...

结果已保存到 `submission.csv`










GRU.ipynb
README.md
requirements.txt
submission.csv

在Kaggle竞赛页面中提交

Submission and Description	Private Score ^①	Public Score ^①	Selected
 submission.csv Complete (after deadline) · 1m ago	0.94600	0.93988	<input type="checkbox"/>
 submission.csv Complete (after deadline) · 11d ago · v1	0.83300	0.82533	<input type="checkbox"/>

相比于GRU，获得了显著的性能提升，公榜和私榜分数都在**0.94**左右。

虽然竞赛已经结束，但我们在未经细致超参数调整和特征工程的情况下就能取得接近SoTA的分数，可见BERT模型预训练—微调模式的卓越能力。

#	△	Team	Members	Score	Entries	Last	Solution
1	—	50022319_Weitao Li		0.95800	6	2mo	
2	—	50019269_Xue XIA		0.95600	8	2mo	
3	▲ 1	50023507 yixiao zhang		0.95300	8	1mo	
4	▲ 4	50029771_Junyu_Ye		0.94800	5	1mo	
5	▲ 2	50028387 Wenyi LIAO		0.94400	12	2mo	
6	▼ 1	50023741_Jindong Xiao		0.94400	16	2mo	
7	▲ 4	50023789_Zhuoyu Yu		0.93900	7	1mo	