lab3 BERT

袁雨 PB20151804

一、实验要求

使用 pytorch 或者 tensorflow 的相关神经网络库编写基于 BERT 的预训练语言模型，利用少量的训练数据，微调模型用于文本情感分类。并和直接用 RNN/Transformer 训练的文本分类器进行对比，研究训练数据量变化对性能带来的影响。

二、实验原理

（一）RNN的问题

- RNN难以并行。
- 简单RNN网络存在长程依赖问题， LSTM引入近似线性依赖的记忆单元来存储远距离信息，但都应对阅读理解等长序列任务。
- 由于优化算法和计算能力的限制，不希望在过度增加模型复杂度的情况下来提升模型表达能力，RNN在实践中很难达到通用近似的能力。

（二）注意力机制

Attention机制：它迫使模型在解码时学会将注意力集中在输入序列的特定部分，而不是仅仅依赖于解码器的LSTM的隐藏向量C。

- **定义**
  - 从大量输入信息里面选择小部分的有用信息来重点处理，并忽略其他信息，这种能力就叫做注意力（Attention）。
- **分类**
  - 聚焦式注意力（Focus Attention）：自上而下的、有意识的注意力。指有预定目的、依赖任务的、主动有意识地聚焦于某一对象的注意力。
  - 基于显著性的注意力（Saliency-Based Attention）：自下而上的、无意识的。不需要主动干预，和任务无关，由外界刺激驱动的注意。举例：赢者通吃（最大汇聚）或者门控机制。
- **计算方式**
  - 在所有输入信息上计算注意力分布。
  - 根据注意力分布计算输入信息的加权平均。
- **变体**
  - 键值对注意力(key-value Attention)

    键用来计算注意力分布，值用来计算聚合信息。用 $(K,V)=[(k_1,v_1),\ldots,(k_N,v_N)]$ 表示 N 组输入信息，给定任务相关的查询向量 q 时，注意力函数为

$$\begin{aligned}\text{att}((K,V),q) &= \sum_{n=1}^{N}\alpha_n v_n \\ &= \sum_{n=1}^{N}\text{softmax}(s\,(k_n,q))v_n \\ &= \sum_{n=1}^{N}\frac{\exp(s\,(k_n,q))}{\sum_j \exp(s\,(k_j,q))}v_n\end{aligned}$$

键值对模式的示例如下图右边所示。

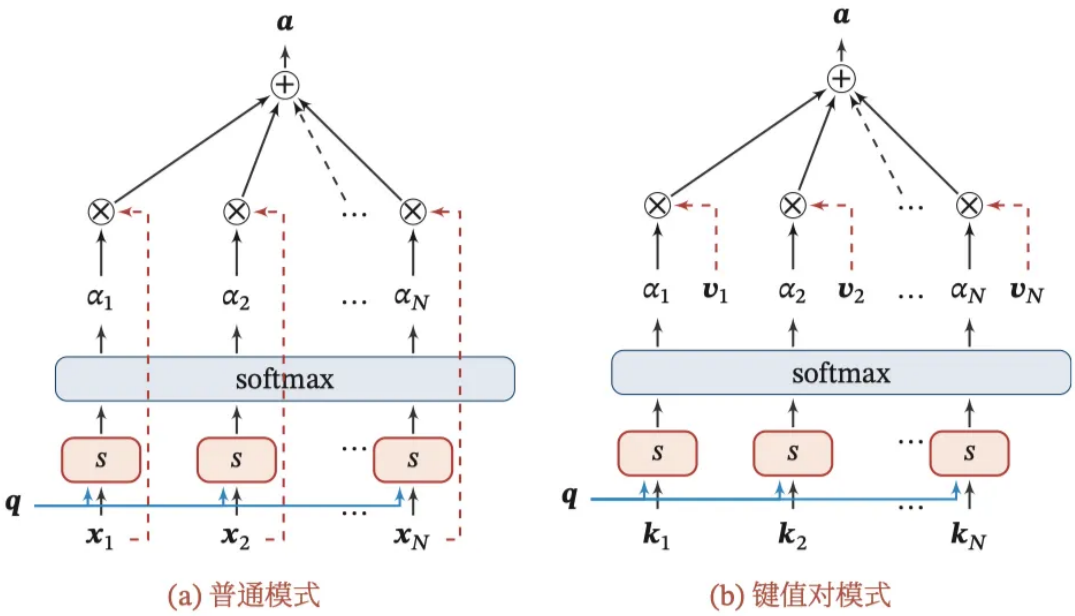


图 8.1　注意力机制

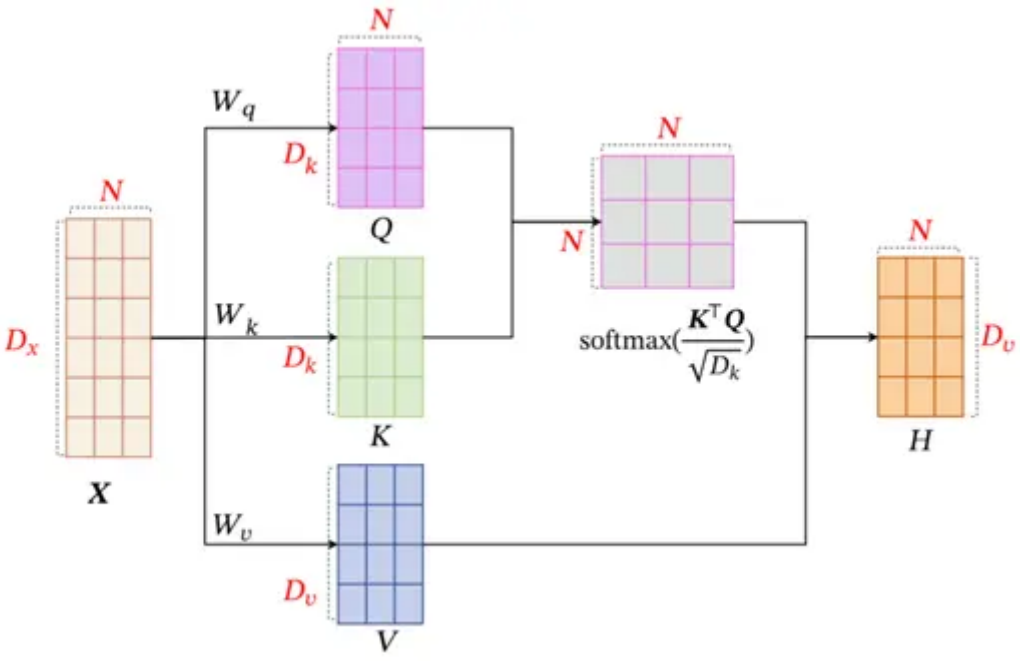
- 多头注意力(Multi-head Attention)

  利用多个查询 $Q=[q_1,\ldots,q_M]$ 来并行地从输入信息中选取多组信息，每个注意力关注输入信息的不同部分。

  $$\text{att}((K,V),Q)=\text{att}((K,V),q_1)\oplus\ldots\oplus\text{att}((K,V),q_M)$$

  其中 $\oplus$ 表示拼接。

（三）自注意力模型



假设输入序列为 $X = [x_1, \ldots, x_N] \in R^{D_x \times N}$, 输出序列为 $H = [h_1, \ldots, h_N] \in R^{D_v \times N}$, 则具体计算过程如下:

1. 对于每个输入，首先将其映射到三个不同的空间，得到三个向量: 查询向量 $q_i \in R^{D_k}$ 、键向量 $k_i \in R^{D_k}$ 和值向量 $v_i \in R^{D_v}$

$$
\begin{aligned}
Q &= W_q X \in R^{D_k \times N} \\
K &= W_k X \in R^{D_k \times N} \\
V &= W_v X \in R^{D_v \times N}
\end{aligned}
$$

其中 $W_q \in R^{D_k \times D_x}, W_k \in R^{D_k \times D_x}, W_v \in R^{D_v \times D_x}$ 分别外线性映射的参数矩阵，$Q = [q_1, \ldots, q_N], K = [k_1, \ldots, k_N], V = [v_1, \ldots, v_N]$ ，分别是查询向量、键向量和值向 量构成的矩阵

2. 对于每一个查询向量 $q_n \in Q$ ，利用公式5的键值对注意力机制，可以得到输出向量 h_n

$$
\begin{aligned}
h_n &= \text{att}((K, V), q_n) \\
&= \sum_{j=1}^{N} \alpha_{nj} v_j \\
&= \sum_{j=1}^{N} \text{softmax}(s\left(k_j, q_n\right)) v_j
\end{aligned}
$$

其中， $n, j \in [1, N]$ 为输出和输入向量序列的位置， α_{nj} 表示第 n 个输出关注到第 j 个输入的权重。

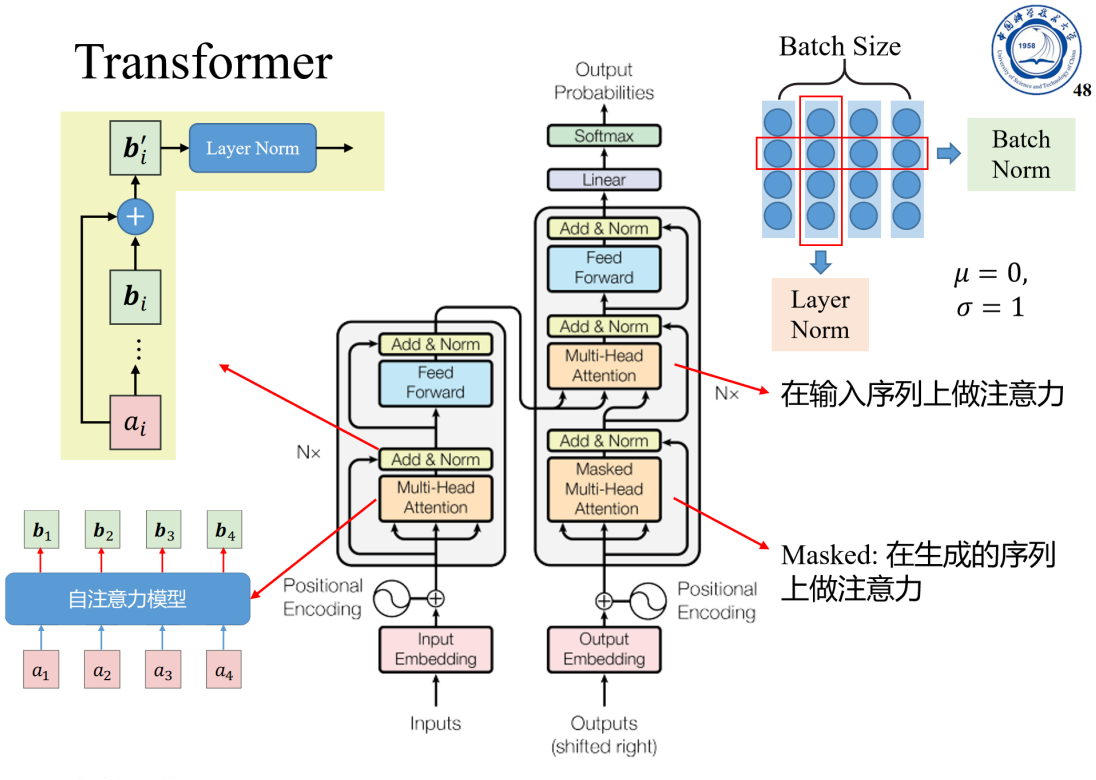
如果使用缩放点积来作为注意力打分函数，输出向量序列可以简写为

$$
H = V \,\text{softmax}\left(\frac{K^T Q}{\sqrt{D_k}}\right)
$$

其中 softmax $(\cdot)$ 为按列进行归一化的函数。

(四) Transformer

- **模型结构**



- **Encoder**
  将输入信息在输入encode部分之前，首先需要和每个位置token的position embedding进行拼接，作为mult-head-self-attention机制的输入，经过mult-head-self-attention机制输入输出的维度一样，这时需要经过一个add&norm层。
  - Add

    add指残差连接。假如我们的输入为X，要得到的输出为H (X)。 那么我们可以通过 $H = F(X) + X$，转换为学习F。 等得到F的输出后，在此基础上加上X即可得H的输出。 在Transformer中，此时的F即是Multi-Head Attention和Feed Forward，再加上X即可得到需要的输出。
  - Norm

    norm指归一化，transformer的归一化用的是层归一化。
  - FFN
    前馈神经网络，用在transformer中，相当于将每个位置的Attention结果映射到一个更大维度的特征空间，然后使用ReLU引入非线性进行筛选，最后恢复回原始维度。需要说明的是，在抛弃了 LSTM 结构后，FFN 中的 ReLU成为了一个主要的能提供非线性变换的单元。

至此，encode部分的三个模块我们都理清了， 在经过mult-head-self-attention以及残差连接和层归一化以及FFN之后我们得到的是每个词经过encode之后的表示，这个表示在影藏层中的维度可能不是初始输入维度，transformer在最中间的隐藏层的embedding size是2048，然后最后一层输出的embedding和输入的维度一样，最终我们得到的是[seq_len,embed_size]大小的矩阵。

- **Decoder**

  Decoder 结构与 Encoder 相似，但是存在一些区别：
  - 包含两个 Multi-Head Attention 层
  - 第一个 Multi-Head Attention 层采用了 Masked 操作

翻译的过程是顺序的，即翻译完第 i 个单词，才可以翻译第 i+1 个单词。通过 Masked 操作可以防止第 i 个单词知道 i+1 个单词之后的信息。

- 第二个 Multi-Head Attention 层的K, V矩阵使用 Encoder 的编码信息矩阵C进行计算，而Q使用上一个 Decoder block 的输出计算

  Decoder block 第二个 Multi-Head Attention 变化不大， 主要的区别在于其中 Self-Attention 的 **K, V**矩阵不是使用 上一个 Decoder block 的输出计算的，而是使用 **Encoder 的编码信息矩阵 C** 计算的。

  根据 Encoder 的输出 **C**计算得到 **K, V**，根据上一个 Decoder block 的输出 **Z** 计算 **Q** (如果是第一个 Decoder block 则使用输入矩阵 **X** 进行计算)，后续的计算方法与之前描述的一致。

  这样做的好处是在 Decoder 的时候，每一位单词都可以利用到 Encoder 所有单词的信息（这些信息无需 **Mask**）。

- 最后有一个 Softmax 层计算下一个翻译单词的概率

（五）BERT

Bert是基于Transformer实现的，BERT中包含很多Transformer模块，其取得成功的一个关键因素是Transformer的强大作用。

- **预训练模型**

  所谓预训练模型，举个例子，假设我们有大量的维基百科数据，那么我们可以用这部分巨大的数据来训练一个泛化能力很强的模型，当我们需要在特定场景使用时，例如做医学命名实体识别，那么，只需要简单的修改一些输出层，再用我们自己的数据进行一个增量训练，对权重进行一个轻微的调整即可。预训练语言模型有很多，典型的如ELMO、GPT、BERT等。

- **BERT的预训练**

  BERT的预训练阶段包括两个任务：Masked LM 和下句预测（Next Sentence Prediction，NSP）。

  - Task 1：Masked LM

    Masked LM 可以形象地称为完形填空问题，随机掩盖掉每一个句子中15%的词，用其上下文来去判断被盖住的词原本应该是什么。举例来说，有这样一个未标注句子 my dog is hairy ，我们可能随机选择了hairy进行遮掩，就变成 my dog is [mask] ，训练模型去预测 [mask] 位置的词，使预测出 hairy的可能性最大，在这个过程中就将上下文的语义信息学习并体现到模型参数中去了。

    这里需要说明，GPT使用统计语言模型，这限制了它只能是单向的，而BERT通过Masked LM能够提取上下文信息。

    论文中，作者做了如下的处理：

    - 80%的时间采用[mask]，my dog is hairy → my dog is [MASK]
    - 10%的时间随机取一个词来代替mask的词，my dog is hairy -> my dog is apple
    - 10%的时间保持不变，my dog is hairy -> my dog is hairy

  - Task 2：Next Sentence Prediction

    很多下游任务（QA和natural language inference）都是基于两个句子之间关系的理解，基于此项任务，为了增强模型对句子之间关系的理解能力。训练数据选择两个句子（50%情况下是真正相连的两个句子，50%是随机拼接的两个句子），判断第二个句子是不是真正的第一个句子的下文。

    其输入形式是，开头是一个特殊符号[CLS]，然后两个句子之间用[SEP]隔断：

```
Input = [CLS] the man went to [MASK] store [SEP]he bought a gallon
[MASK] milk [SEP]
Label = IsNext
Input = [CLS] the man [MASK] to the store [SEP]penguin [MASK] are flight
##less birds[SEP]
Label = NotNext
```

实际构建预训练任务时，是首选设计好 “下句预测” 任务，生成该任务的标注信息，在此基础上构建 “Masked LM” 任务，生成掩码语言模型的标注信息。考虑到预训练涉及两个句子，BERT 采用如下的输入信息表征方案：

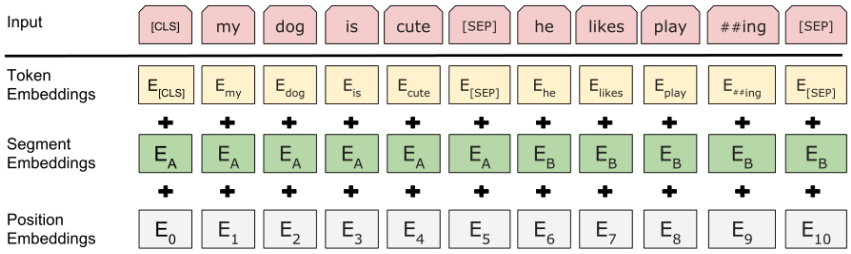


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

- token embedding ： 将各个词转换成固定维度的向量。在BERT中，每个词会被转换成768维的向量表示。在实际代码实现中，输入文本在送入token embeddings 层之前要先进行tokenization处理。此外，两个特殊的token会被插入到tokenization的结果的开头 ([CLS])和结尾 ([SEP])
- segment embedding： 用于区分一个token属于句子对中的哪个句子。Segment Embeddings 层只有两种向量表示。前一个向量是把0赋给第一个句子中的各个token, 后一个向量是把1赋给第二个句子中的各个token。如果输入仅仅只有一个句子，那么它的segment embedding就是全0 。
- position embedding：Transformers无法编码输入的序列的顺序性，所以要在各个位置上学习一个向量表示来将序列顺序的信息编码进来。加入position embeddings会让BERT理解下面下面这种情况，“ I think, therefore I am ”，第一个 “I” 和第二个 “I”应该有着不同的向量表示。

这3种embedding都是768维的，最后要将其按元素相加，得到每一个token最终的768维的向量表示。

- **BERT的使用**

  BERT最终输出的就是句子中每个token的768维的向量，第一个位置是[CLS]，它的向量表示蕴含了这个句子整体的信息，用于做文本分类等句子级任务；对于序列标注等token级任务，就需要使用到每一个token的向量表示。只要将768维向量通过一个线性层映射到最终的分类空间中即可。

三、实验步骤

1.数据预处理

文本数据的处理包括下面几步：

1. 分词，即把一个句子分成一个个的单词
2. tokenizer, 给每个单词一个唯一的 ID，方便后续计算
3. embedding, 对于每个单词（也就是 ID），都需要有一个对应的词向量。Bert 这种预训练的模型自身是有 Embedding Matrix 的。但是对于自己训练 LSTM 这种模型，先需要使用 word2vec 的方法

自己训练得到 Embedding Matrix。

文本在训练的时候，还需要进行填补，即通过填充字符，使得一个 Batch 中每个 sentence 数据是等长的。填补的时候，需要记录原来 sentence 的长度，这样 LSTM 才知道中间停下来的位置。

（1）处理原始数据

```
class RawData:
    """原始数据静态类"""
    __type = None
    __data = []

    __vocab = None

    @staticmethod
    def read_imdb(type_):  # 读取原始数据，可以读取 训练集或者测试集
        if RawData.__type != type_:
            RawData.__type = type_

            if not os.path.exists(os.path.join(root_data_path, "aclImdb")):  # 解
压数据
                print(">>> start decompression")
                with tarfile.open(compress_data_file, 'r') as f:
                    f.extractall(root_data_path)
                print(">>> end decompression")

            RawData.__data = []
            print(">>> start read data")
            for label in ['pos', 'neg']:
                folder_name = os.path.join(root_data_path, 'aclImdb', type_,
label)
                file_list = os.listdir(folder_name)
                if type_ == 'train':
                    file_list = file_list[:int(len(file_list) * train_rate)]  #
训练集使用部分数据
                for file in tqdm(file_list):
                    with open(os.path.join(folder_name, file), 'rb') as f:
                        review = f.read().decode('utf-8').replace('\n',
'').lower()
                        RawData.__data.append([review, 1 if label == 'pos' else
0])
            print(">>> end read data")
            random.seed(2023)
            random.shuffle(RawData.__data)  # 这里就把数据打乱，这样后面读取数据就不打乱
了

        return RawData.__data

    @staticmethod
    def get_vocab():  # 利用 word2vec 的方法训练 token embedding
        # 只使用训练集去建立 vocab ，这样保证训练集和测试集 token 对应的 id 是一样的，还防
止数据泄露
        if RawData.__vocab is None:
            tokenizer = get_tokenizer('basic_english')
            data = RawData.read_imdb(type_="train")
            counter = Counter()
            sen_tokens = []
```

```
            for line, label in data:  # 把 sentence 变成 work list
                tokens = tokenizer(line)
                counter.update(tokens)
                sen_tokens.append(tokens)

            # 读取词向量，如果没有则训练得到
            if os.path.exists(embedding_file_path):
                word_vector = KeyedVectors.load(embedding_file_path)
            else:
                print(">>> start train word embedding")
                model = Word2Vec(sentences=sen_tokens, vector_size=vector_size,
window=window, min_count=1, workers=4)
                print(">>> end train word embedding")
                word_vector = model.wv
                # 增加 <unk> 和 <pad> 词向量，用于填充
                zeros = np.zeros(vector_size)
                word_vector.add_vector('<unk>', zeros)
                word_vector.add_vector('<pad>', zeros)
                word_vector.save(embedding_file_path)

            # 创建词到下标的转换
            sorted_by_freq_tuples = sorted(counter.items(), key=lambda x: x[1],
reverse=True)
            ordered_dict = OrderedDict(sorted_by_freq_tuples)
            # min_freq，去掉出现次数小于 min_freq 的词
            # specials，特殊字符 '<unk>', '<pad>'
            _vocab = vocab(ordered_dict=ordered_dict, min_freq=min_freq,
specials=['<unk>', '<pad>'], special_first=True)
            _vocab.set_default_index(_vocab['<unk>'])  # 这个使得对于没有出现的词，会
输出 '<unk>' 的下标
            keys = _vocab.get_itos()  # all the word sorted by index
            embeddings = word_vector[keys]
            _vocab.embeddings = embeddings
            RawData.__vocab = _vocab
        return RawData.__vocab
```

定义类 `RawData`，包含两个函数 `read_imdb` 和 `get_vocab`。其中 `read_imdb` 读取原始数据，分为读取训练集和测试集；`get_vocab` 利用 word2vec 的方法训练 token embedding 并创建词到下标的转换。

- `read_imdb` 先检查是否已经解压数据，如果没有则解压。然后将数据整理成形如 `[[review1, label1], [review2, label2], ...]` 的列表。对于每个 review，首先使用 utf-8 解码，去掉换行符，并将其转化为小写。最后将正负样本合并，打乱顺序，并返回整理好的数据列表。
- `get_vocab` 先调用 `read_imdb` 获取训练集数据。然后使用 `get_tokenizer('basic_english')` 得到一个 tokenizer，tokenizer 会将输入的句子分词为单词的列表。接下来，对于每个 review，循环将其转化为单词列表，把所有单词计数，同时把所有单词列表存储到列表 `sen_tokens` 中。在此基础上，使用 word2vec 的方法训练 token embedding，得到词向量。如果以前已经训练得到词向量，则直接从文件中读取。接下来，将单词按词频排序，并去掉词频小于 `min_freq` 的单词，增加特殊字符 `<unk>` 和 `<pad>` 的词向量，用于填充。最后创建词到下标的转换，返回该转换。

(2) 构建 Bert 模型的数据集和 DataLoader

```
def create_dataset_bert(type_):
    # 利用 Bert 自带的 tokenizer 对文本进行编码
    tokenizer = BertTokenizer.from_pretrained(pretrained_bert_model_name,
do_lower_case=True)

    input_ids = []
    attention_masks = []
    labels = []

    all_text = RawData.read_imdb(type_=type_)
    print(f">>> start process bert tokenizer")
    for element in tqdm(all_text):
        sentence, label = element
        encoded_dict = tokenizer.encode_plus(sentence, add_special_tokens=True,
max_length=sentence_max_length,
                                             padding="max_length",
truncation=True, return_attention_mask=True,
                                             return_tensors="pt")
        input_ids.append(encoded_dict['input_ids'])
        attention_masks.append(encoded_dict['attention_mask'])
        labels.append(label)
    print(f">>> end process bert tokenizer")
    input_ids = torch.cat(input_ids, dim=0)
    attention_masks = torch.cat(attention_masks, dim=0)
    labels = torch.tensor(labels)

    dataset = TensorDataset(input_ids, attention_masks, labels)
    return dataset


def create_loader_bert(type_):
    dataset = create_dataset_bert(type_=type_)

    if type_ == "train":
        val_size = int(split_rate * len(dataset))
        train_size = len(dataset) - val_size

        train_dataset, val_dataset = random_split(dataset, [train_size,
val_size])

        train_loader = DataLoader(train_dataset,
sampler=RandomSampler(train_dataset), batch_size=batch_size,
                                  num_workers=3)
        val_loader = DataLoader(val_dataset,
sampler=SequentialSampler(val_dataset), batch_size=batch_size,
                                num_workers=3)

        return train_loader, val_loader
    else:  # test
        test_loader = DataLoader(dataset, sampler=SequentialSampler(dataset),
batch_size=batch_size, num_workers=3)

        return test_loader
```

定义两个函数 `create_dataset_bert` 和 `create_loader_bert`。

`create_dataset_bert` 首先使用 `BertTokenizer` 的 `from_pretrained` 方法得到一个 tokenizer，用于将句子转化为 token。接下来，遍历传入的数据类型所代表的全部数据（即训练数据或测试数据），把句子和标签分别加到 all_text 列表中。在此基础上，循环对每个元素进行处理，即使用 tokenizer 对句子进行编码，并填充到统一的长度。其中 attention_masks 表示哪些 token 是 padding symbol，哪些是有效的 token。最后，将所有 input_ids, attention_masks, labels 放到三个列表中并返回。

`create_loader_bert` 函数调用 `create_dataset_bert` 函数创建 dataset。如果数据类型是训练数据，则把 dataset 分割为训练集和验证集，分别创建 DataLoader 并返回；否则直接创建 test_loader 并返回。其中，在训练集 DataLoader 的 Sampler 中，使用 `RandomSampler` 随机采样一个 batch 的数据。在验证集 DataLoader 的 Sampler 中，使用 `SequentialSampler` 按序排列。

(3) 构建 RNN 模型的数据集和 DataLoader

```
class DatasetClassRNN(Dataset):
    def __init__(self, type_):
        if (type_ == "train") or (type_ == "val"):
            all_text = RawData.read_imdb(type_="train")
            train_len = int(len(all_text) * (1 - split_rate))
            if type_ == "train":
                self.raw_text = all_text[:train_len]
            else:
                self.raw_text = all_text[train_len:]
        else:
            self.raw_text = RawData.read_imdb(type_="test")
        self.vocab = RawData.get_vocab()  # 保证使用的是同一个 embedding matrix
        self.tokenizer = get_tokenizer('basic_english')
        self.vocab_size = len(self.vocab)

    def text_pipeline(self, text):
        return [self.vocab[token] for token in self.tokenizer(text)]

    def __getitem__(self, item):
        text, label = self.raw_text[item]
        text_processed = self.text_pipeline(text)
        return text_processed, label

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


class MySampler(Sampler):
    """RNN训练时，相似长度的句子放在一个 Batch 中
    """
    def __init__(self, dataset, batch_size):
        super(Sampler, self).__init__()
        self.dataset = dataset
        self.batch_size = batch_size
        # 这里按照长度降序排序，这样符合后面 rnn 的 pack_padded_sequence 输入
        self.indices = np.argsort([len(sentence[0]) for sentence in dataset])
[::-1]
        self.count = int(len(dataset) / self.batch_size)

    def __iter__(self):
        for i in range(self.count):
            yield self.indices[i * self.batch_size: (i + 1) * self.batch_size]
```

```
    def __len__(self):
        return self.count


def collate_fn(batch_data, pad=0):
    # label_list, text_list, offsets = [], [], []
    # 把每个句子填补成一样长度
    offsets = [len(sentence[0]) for sentence in batch_data]
    max_len = max(offsets)
    label_list = [sentence[1] for sentence in batch_data]
    text_list = [sentence[0] + [pad] * (max_len - len(sentence[0])) for sentence
in batch_data]
    text_tensor = torch.LongTensor(text_list)
    label_tensor = torch.LongTensor(label_list)
    offsets_tensor = torch.LongTensor(offsets)
    return text_tensor, label_tensor, offsets_tensor


def data_loader_RNN(dataset, sort_length=True):
    if sort_length:
        my_sampler = MySampler(dataset=dataset, batch_size=batch_size)
        loader = DataLoader(dataset=dataset, batch_sampler=my_sampler,
collate_fn=collate_fn)
    else:
        loader = DataLoader(dataset=dataset,
                            batch_size=batch_size,
                            shuffle=True,
                            collate_fn=collate_fn)

    return loader
```

定义 `DatasetClassRNN` 类，先根据传入的数据类型（train、val 或 test）读取原始文本，然后调用 `RawData.get_vocab` 方法获取词汇表，再使用 `get_tokenizer('basic_english')` 方法创建一个单词分割器 tokenizer。在 `text_pipeline` 方法中，将原始文本通过 tokenizer 分割成单词，再将每个单词转换成在词汇表中对应的索引，并返回索引组成的列表。在 `__getitem__` 函数中，从原始文本列表中取出一个样本，把样本的文本进行处理得到文本序列，再把文本序列和标签返回。在 `__len__` 函数中，返回原始文本长度。

定义 `MySampler` 类，首先，对文本进行排序，按照长度降序排序，这样符合后面 rnn 的 pack_padded_sequence 输入。然后采用迭代器的方式，把相似长度的句子放在一个 Batch 中。

定义 `collate_fn` 函数，对文本进行填充，使得每个批次中的文本长度一致，同时记录下每个文本的长度信息。

定义 `data_loader_RNN` 函数，如果 sort_length 为 True 表示需要按照文本长度排序，则调用 `MySampler` 类进行采样并返回对应的 DataLoader 对象。如果 sort_length 为 False，则使用 PyTorch 中的 `DataLoader` 类并指定 shuffle=True 进行数据随机采样。最后返回对应的 DataLoader。

2.定义网络结构

（1）BERT

```
def get_bert():
    # 直接获取带有分类器的 Bert 预训练模型
    net = BertForSequenceClassification.from_pretrained(
        pretrained_bert_model_name,
        num_labels=2,
        output_attentions=False,
        output_hidden_states=False
    )

    return net
```

定义 `get_bert` 函数，在函数内部，使用 `BertForSequenceClassification.from_pretrained` 方法直接从预训练模型库中获取指定的预训练 Bert 模型并初始化，其中需要传入预训练 Bert 模型的名称 `pretrained_bert_model_name`，标签数 `num_labels`=2（因为这里是二分类任务，在 IMDB 数据集中只有正面和负面两种标签），同时设置输出中是否包含 attention 和 hidden state 的信息。然后返回初始化后的 Bert 模型。

（2）Transformer

```
class Transformer(nn.Module):
    def __init__(self, embedding_dim, num_classes, num_layers, num_heads,
hidden_size, dropout):
        super(Transformer, self).__init__()
        self.pos_encoding = PositionalEncoding(d_model = embedding_dim,
dropout=dropout)
        encoder_layer = nn.TransformerEncoderLayer(d_model = embedding_dim,
                                                   nhead = num_heads,
                                                   dim_feedforward=hidden_size,
                                                   dropout = dropout,
                                                   batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
num_layers)
        self.fc = nn.Linear(embedding_dim, num_classes)

    def forward(self, x):
        x = self.pos_encoding(x)
        x = self.transformer_encoder(x)
        x = torch.mean(x, dim=1)
        x = self.fc(x)
        return x

# 位置编码
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-
math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
```

```
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)
```

- 定义 `Transformer` 类。
  - 在 `__init__` 函数中，定义了 Transformer 模型的各层结构。调用 `PositionalEncoding` 类，创建位置编码器 `pos_encoding`，用于将输入序列加上相对位置的权值；定义了 `nn.TransformerEncoderLayer`，用于构建多层的 `TransformerEncoder`；以及一个全连接层。
  - 在 `forward` 函数中，首先将输入序列 x 加上位置编码，然后通过多层 `TransformerEncoder` 进行计算。之后使用 `torch.mean` 取所有时刻的输出的平均值，最后通过全连接层进行分类。
- 定义 `PositionalEncoding` 类。
  - 首先定义了一个 dropout 层。然后根据论文中的公式，采用 sin 和 cos 函数定义相对位置的权重。具体来说，pe 矩阵记录了不同位置的权重，其中第 i 行第 j 列表示的是第 i 个位置、第 j 维的权重。position 矩阵表示序列中各个位置的索引，div_term 矩阵记录了不同维度下的除数。最后将计算出的权重 pe 添加到输入序列 x 上，并对结果应用 dropout。使用 register_buffer 方法对 pe 进行注册，可以将其添加到模型的缓存中，并在模型进行保存或加载时一并处理。

(3) RNN

```
class RNN(nn.Module):
    def __init__(self, model_type, vocab_size, embed_dim, num_class,
hidden_size, num_layers, num_heads, dropout):
        super(RNN, self).__init__()
        self.num_layers = num_layers
        self.model_type = model_type

        self.embedding =
nn.Embedding(num_embeddings=vocab_size,embedding_dim=embed_dim)
        # 得到训练好的词向量，利用训练好的词向量初始化 embedding 层
        vocab = RawData.get_vocab()
        self.embedding.weight.data.copy_(torch.tensor(vocab.embeddings,
dtype=torch.float32))

        if model_type == "RNN":
            self.rnn = nn.RNN(input_size=embed_dim,
                              hidden_size=hidden_size,
                              num_layers=num_layers,
                              nonlinearity='tanh',
                              batch_first=True)
            self.fc = nn.Linear(hidden_size, num_class)  # 线性模型分类

        elif model_type == 'LSTM':
            self.rnn = nn.LSTM(input_size=embed_dim,
                               hidden_size=hidden_size,
                               num_layers=num_layers,
                               batch_first=True)
            self.fc = nn.Linear(hidden_size, num_class)  # 线性模型分类
```

```
        elif model_type == 'Transformer':
            self.rnn = Transformer(embedding_dim = embed_dim,
                                   num_classes=num_class,
                                   num_layers=num_layers,
                                   num_heads=num_heads,
                                   hidden_size=hidden_size,
                                   dropout=dropout)

    def forward(self, inputs, offsets):
        embeds = self.embedding(inputs)

        if self.model_type == "Transformer":
            x = self.rnn(embeds)

        else :
            offsets = offsets.to("cpu")
            packed_x = pack_padded_sequence(input=embeds, lengths=offsets,
batch_first=True)
            x = self.rnn(packed_x)
            x = x[1]
            if self.model_type == "LSTM":
                x = x[0]
            x = x.view(-1, x.shape[1], x.shape[2])
            x = x[-1]
            x = self.fc(x)
        return x
```

定义 `RNN` 类。

- 在 `__init__` 函数中，定义了 RNN 模型的各层结构，包括 Embedding 层、RNN/LSTM/Transformer 层和全连接层（即线性模型分类器）。
- 在 `forward` 函数中，先将输入序列通过 Embedding 层转换为对应的嵌入表示 `embeds`。然后根据输入文本的偏移量 `offsets`，使用 `pack_padded_sequence` 进行填充操作。接着将填充后的数据送入 RNN/LSTM/Tranformer 层，并获取其状态输出。若 `model_type` 为 "LSTM"，在 `forward` 方法中需要对状态输出分成 hidden state 和 cell state 两部分，并只取 hidden state 用于后面的计算。最后将输出结果送入全连接层进行分类，并返回最终结果。当 `model_type` 为 "Transformer" 时，直接调用 `Transformer` 类。

3.定义训练流程

（1）一些功能函数

```
def criterion(output, target):
    cri = nn.CrossEntropyLoss()
    loss = cri(output, target)
    return loss


def calc_acc(output, labels):
    output = output.cpu().numpy()
    output = np.argmax(output, axis=1)
    labels = labels.cpu().numpy()
    labels = labels.reshape(len(labels))
```

```
    acc = np.sum(output == labels) / len(labels)
    return acc


def plot_loss(loss_list, model_type):
    plt.figure()
    train_loss = loss_list['train']
    val_loss = loss_list['val']
    plt.plot(train_loss, c="red", label="train_loss")
    plt.plot(val_loss, c="blue", label="val_loss")
    plt.legend()
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title(f"Training and Validation Loss of {model_type} in Each Epoch")
    plt.savefig(f"./fig/{model_type}_loss.png")
```

定义了三个函数：`criterion`、`calc_acc` 和 `plot_loss`。

- `criterion` 使用交叉熵损失函数计算模型输出 output 和目标值 target 之间的损失。
- `calc_acc` 计算模型预测结果和真实标签之间的准确率。使用 np.argmax 将 output 中概率值最大的类别作为预测结果。
- `plot_loss` 绘制训练和验证集上的 loss 变化曲线。

(2) Lab3_RNN

```
class Lab3_RNN(object):

    def __init__(self, lr=0.001, epochs=20, device='cuda',
                 hidden_size=100, sort_length=True, patience=5,
                 fix_embedding=False,
                 num_layers=1,
                 model_type="RNN",
                 num_heads=8,
                 dropout=0.1):
        self.lr = lr
        self.epochs = epochs
        self.device = torch.device(device)
        self.hidden_size = hidden_size
        self.sort_length = sort_length
        self.patience = patience
        self.fix_embedding = fix_embedding
        self.num_layers = num_layers
        self.model_type = model_type
        self.num_heads = num_heads
        self.dropout = dropout

        self.net = None
        self.optimizer = None
        self.vocab_size = None
        self.embed_size = None
        self.num_class = None
        self.loss_list = {"train": [], "val": []}


    def train(self):
```

```
        save_path = os.path.join(best_model_dir, f"
{self.model_type}_best_model.pth")
        # 加载数据
        train_dataset = DatasetClassRNN(type_="train")
        train_loader = data_loader_RNN(dataset=train_dataset,
sort_length=self.sort_length)

        val_dataset = DatasetClassRNN(type_="val")
        val_loader = data_loader_RNN(dataset=val_dataset,
sort_length=self.sort_length)

        self.vocab_size = train_dataset.vocab_size
        self.embed_size = embed_size
        self.num_class = num_class
        # 定义网络
        self.net = RNN(self.model_type,
                       self.vocab_size,
                       self.embed_size,
                       self.num_class,
                       self.hidden_size,
                       self.num_layers,
                       self.num_heads,
                       self.dropout).to(self.device)

        if self.fix_embedding:
            for k, v in self.net.named_parameters():
                if k == "embedding.weight":
                    v.requires_grad = False

        self.optimizer = optim.Adam(filter(lambda p: p.requires_grad,
self.net.parameters()),
                                    lr=self.lr)

        total_params = sum([param.nelement() for param in self.net.parameters()
if param.requires_grad])
        print(f">>> model type: {self.model_type}, train rate: {train_rate}")
        print(f">>> total parameters: {total_params}")
        patience = 0
        best_val_acc = 0.
        best_train_acc = 0.
        for epoch in range(self.epochs):
            t1 = datetime.now()
            for data in tqdm(train_loader):
                text_tensor, label_tensor, offsets_tensor = data
                text_tensor = text_tensor.to(self.device)
                label_tensor = label_tensor.to(self.device)
                offsets_tensor = offsets_tensor.to(self.device)
                predict_label = self.net(text_tensor, offsets_tensor)
                self.optimizer.zero_grad()
                loss = criterion(predict_label, label_tensor)
                loss.backward()
                self.optimizer.step()

            train_loss = 0.0
            train_acc = 0.0
            val_loss = 0.0
            val_acc = 0.0
```

```
            with torch.no_grad():
                for data in tqdm(train_loader):
                    # for data in train_loader:
                    text_tensor, label_tensor, offsets_tensor = data
                    text_tensor = text_tensor.to(self.device)
                    label_tensor = label_tensor.to(self.device)
                    offsets_tensor = offsets_tensor.to(self.device)
                    predict_label = self.net(text_tensor, offsets_tensor)
                    loss = criterion(predict_label, label_tensor)
                    acc_ = calc_acc(predict_label, label_tensor)

                    train_acc += acc_
                    train_loss += loss.item()

                train_loss = train_loss / len(train_loader)
                train_acc = train_acc / len(train_loader)

                for data in tqdm(val_loader):
                    # for data in val_loader:
                    text_tensor, label_tensor, offsets_tensor = data
                    text_tensor = text_tensor.to(self.device)
                    label_tensor = label_tensor.to(self.device)
                    offsets_tensor = offsets_tensor.to(self.device)
                    predict_label = self.net(text_tensor, offsets_tensor)
                    loss = criterion(predict_label, label_tensor)
                    acc_ = calc_acc(predict_label, label_tensor)

                    val_acc += acc_
                    val_loss += loss.item()

                val_loss = val_loss / len(val_loader)
                val_acc = val_acc / len(val_loader)

            self.loss_list['train'].append(train_loss)
            self.loss_list['val'].append(val_loss)

            print(f"epoch: {epoch}, train loss: {train_loss:.6f}, train acc:
{train_acc:.6f}, "
                  f"val loss: {val_loss:.6f}, val acc: {val_acc:.6f},  time:
{datetime.now() - t1}")
            t1 = datetime.now()
            if val_acc < best_val_acc:
                patience = patience + 1
                if patience > self.patience:
                    break
            else:
                patience = 0
                best_val_acc = val_acc
                best_train_acc = train_acc
                if not os.path.exists(best_model_dir):
                    os.mkdir(best_model_dir)
                print(f"save best weights to {save_path}")
                torch.save(self.net.state_dict(), save_path)

        print(">>> Finished training")
        plot_loss(self.loss_list,self.model_type)
        print(">>> Finished plot loss")
```

```
        return best_train_acc, best_val_acc

    def test(self, model_path=None):
        if model_path is None:
            model_path = os.path.join(best_model_dir, f"
{self.model_type}_best_model.pth")
        test_dataset = DatasetClassRNN(type_="test")
        test_loader = data_loader_RNN(dataset=test_dataset,
sort_length=self.sort_length)

        test_loss = 0.0
        test_acc = 0.0
        # init model
        # 定义网络
        if self.net is None:
            self.vocab_size = test_dataset.vocab_size
            self.embed_size = embed_size
            self.num_class = num_class
            self.net = RNN(self.model_type,
                           self.vocab_size,
                           self.embed_size,
                           self.num_class,
                           self.hidden_size,
                           self.num_layers).to(self.device)
        # load best weights
        self.net.load_state_dict(torch.load(model_path))
        with torch.no_grad():
            for data in tqdm(test_loader):
                text_tensor, label_tensor, offsets_tensor = data
                text_tensor = text_tensor.to(self.device)
                label_tensor = label_tensor.to(self.device)
                offsets_tensor = offsets_tensor.to(self.device)
                predict_label = self.net(text_tensor, offsets_tensor)
                loss = criterion(predict_label, label_tensor)
                acc_ = calc_acc(predict_label, label_tensor)

                test_loss += loss.item()
                test_acc += acc_

            test_loss = test_loss / len(test_loader)
            test_acc = test_acc / (len(test_loader))
            print(f"test loss: {test_loss}, test acc: {test_acc}")
        return test_acc
```

定义`Lab3_RNN`类，实现 RNN 模型的训练和测试。

- 初始化参数：模型结构包括 RNN、LSTM、Transformer，可以通过传入参数 model_type 来选择。
- 训练函数：训练 RNN 模型，采用了交叉熵损失函数和 Adam 优化器，同时还使用了 Early Stopping 策略。在每一轮训练后输出每个 epoch 的训练集和验证集上的 loss 和 accuracy，以及所花费的时间。并保存最优的权重文件以及打印输出训练过程。
- 测试函数：加载保存的最佳模型权重，并输出测试集上的 loss 和 accuracy。
- 可视化：将训练和验证损失绘制成图像。

(3) Lab3_Bert

```
class Lab3_Bert(object):

    def __init__(self, device="cuda:0"):
        self.net = None
        self.device = device
        self.loss_list = {"train": [], "val": []}

    def train(self, patience=3, epochs=10, lr=2e-5):

        self.net = get_bert()
        self.net.cuda(self.device)
        device = self.device
        optimizer = Adam(self.net.parameters(), lr=lr)

        save_path = os.path.join(best_model_dir, 'bert_best_model.pth')
        train_loader, val_loader = create_loader_bert(type_="train")
        total_params = sum([param.nelement() for param in
self.net.parameters()])
        print(f">>> model type: Bert, train rate: {train_rate}")
        print(f">>> total params: {total_params}")
        delay = 0
        best_val_loss = 1.0
        for epoch in range(epochs):
            t1 = datetime.now()
            self.net.train()

            for batch in tqdm(train_loader):
                b_input_ids = batch[0].to(device)
                b_input_mask = batch[1].to(device)
                b_labels = batch[2].to(device)

                self.net.zero_grad()

                tmp = self.net(b_input_ids, token_type_ids=None,
attention_mask=b_input_mask, labels=b_labels)
                loss = tmp[0]
                logits = tmp[1].detach()
                acc = calc_acc(logits, b_labels)
                loss.backward()

                optimizer.step()

            train_loss, train_acc = 0.0, 0.0
            val_loss, val_acc = 0.0, 0.0
            with torch.no_grad():
                for batch in tqdm(train_loader):
                    b_input_ids = batch[0].to(device)
                    b_input_mask = batch[1].to(device)
                    b_labels = batch[2].to(device)

                    tmp = self.net(b_input_ids, token_type_ids=None,
attention_mask=b_input_mask, labels=b_labels)

                    loss = tmp[0]
                    logits = tmp[1]
```

```
                    acc = calc_acc(logits, b_labels)
                    train_loss += loss.item()
                    train_acc += acc

                for batch in tqdm(val_loader):
                    b_input_ids = batch[0].to(device)
                    b_input_mask = batch[1].to(device)
                    b_labels = batch[2].to(device)

                    tmp = self.net(b_input_ids, token_type_ids=None,
attention_mask=b_input_mask, labels=b_labels)

                    loss = tmp[0]
                    logits = tmp[1]

                    acc = calc_acc(logits, b_labels)
                    val_loss += loss.item()
                    val_acc += acc

            train_loss, train_acc = train_loss / len(train_loader), train_acc /
len(train_loader)
            self.loss_list['train'].append(train_loss)
            val_loss, val_acc = val_loss / len(val_loader), val_acc /
len(val_loader)
            self.loss_list['val'].append(val_loss)
            print(f"epoch {epoch}, train loss is: {train_loss:8.6f}, acc is
{train_acc:7.6f}, "
                  f"val_loss is: {val_loss:8.6f}, acc is {val_acc:7.6f}, time:
{datetime.now() - t1}")
            t1 = datetime.now()
            if val_loss < best_val_loss:
                best_val_loss = val_loss
                delay = 0
                if not os.path.exists(best_model_dir):
                    os.mkdir(best_model_dir)
                print(f">>> save best weights to f{save_path}")
                torch.save(self.net, save_path)
            else:
                delay = delay + 1
                if delay > patience:
                    break
        print(">>> Finished training")
        plot_loss(self.loss_list, 'BERT')
        print(">>> Finished plot loss")

    def test(self):
        best_model_path = os.path.join(best_model_dir, 'bert_best_model.pth')
        test_loader = create_loader_bert(type_="test")
        device = self.device
        self.net = torch.load(best_model_path)
        self.net.cuda(self.device)
        test_loss, test_acc = 0.0, 0.0

        with torch.no_grad():
            for batch in tqdm(test_loader):
                b_input_ids = batch[0].to(device)
                b_input_mask = batch[1].to(device)
```

```
                b_labels = batch[2].to(device)

                tmp = self.net(b_input_ids, token_type_ids=None,
attention_mask=b_input_mask, labels=b_labels)

                loss = tmp[0]
                logits = tmp[1]

                acc = calc_acc(logits, b_labels)
                test_loss += loss.item()
                test_acc += acc

        test_loss, test_acc = test_loss / len(test_loader), test_acc /
len(test_loader)
        print(f"test loss is: {test_loss:8.6f}, acc is: {test_acc:7.6f}")
```

定义 `Lab3_Bert` 类，实现 BERT 模型的训练和测试。

- 初始化参数。
- 训练：训练 Bert 模型，采用了交叉熵损失函数和 Adam 优化器，同时还使用了 Early Stopping 策略。在每一轮训练后输出每个 epoch 的训练集和验证集上的 loss 和 accuracy，以及所花费的时间。并保存最优的权重文件以及打印输出训练过程。
- 测试：加载保存的最佳模型权重，并输出测试集上的 loss 和 accuracy。
- 可视化：将训练和验证损失绘制成图像。

四、实验过程与分析

- train rate

  train rate 表示 train data 中使用的比例。

train rate	train acc	val acc	time per epoch(min:s)
0.25（2）	0.965455	0.906646	03:47
0.5（1）	0.967200	0.915207	07:34
0.75（3）	0.936900	0.907270	11:17
1（4）	0.951000	0.924521	15:00

实验可知，增大训练样本比例对准确率的提升不大，而且会很大程度地增加训练时长。

选择 train rate = 0.5较合适，其表现相对较好，且时间开销适中。

- model

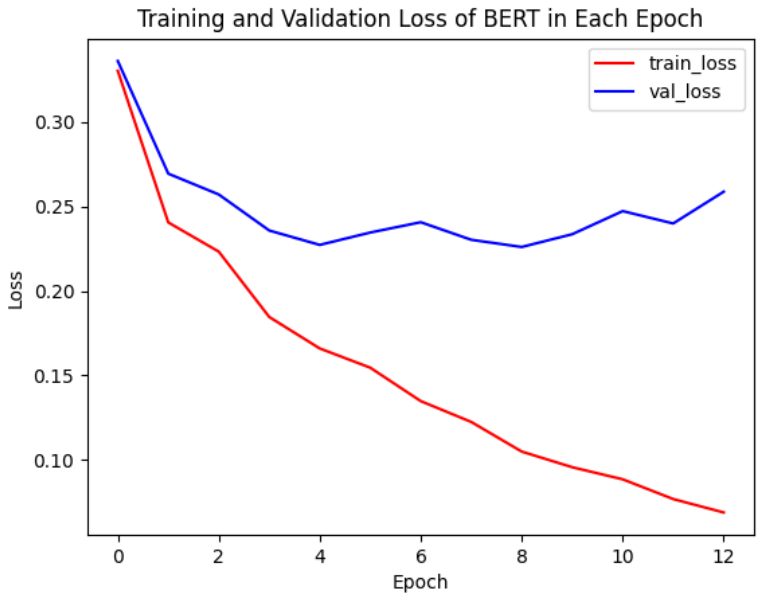
  不同的语言模型，其中 tr 表示 train rate 。

model	train acc	val acc	time per epoch(min:s)
RNN（tr=0.5）	0.714800	0.708800	02:03
RNN（tr=1）	0.762750	0.767400	04:46
LSTM（tr=0.5）	0.869100	0.854400	02:31
LSTM（tr=1）	0.950900	0.906800	05:12
Transformer（tr=0.5）	0.848900	0.844400	01:34
Transformer（tr=1）	0.897850	0.881000	03:59
BERT（tr=0.5）	0.967200	0.915207	07:34
BERT（tr=1）	0.951000	0.924521	15:00

实验可知，BERT 的准确率比 RNN/LSTM 和 Transformer 高，且只需要少量训练样本进行微调就可以得到很好的效果，而其他模型在训练样本较少的时候准确率不高。

五、实验结果

- BERT（tr=0.5）训练过程的 loss 曲线



由图可知训练较少的轮数就可以达到较好的效果，轮数过多会发生过拟合。

- BERT（tr=0.5）在测试集上的结果

  test loss is: 0.224821, acc is: 0.918506