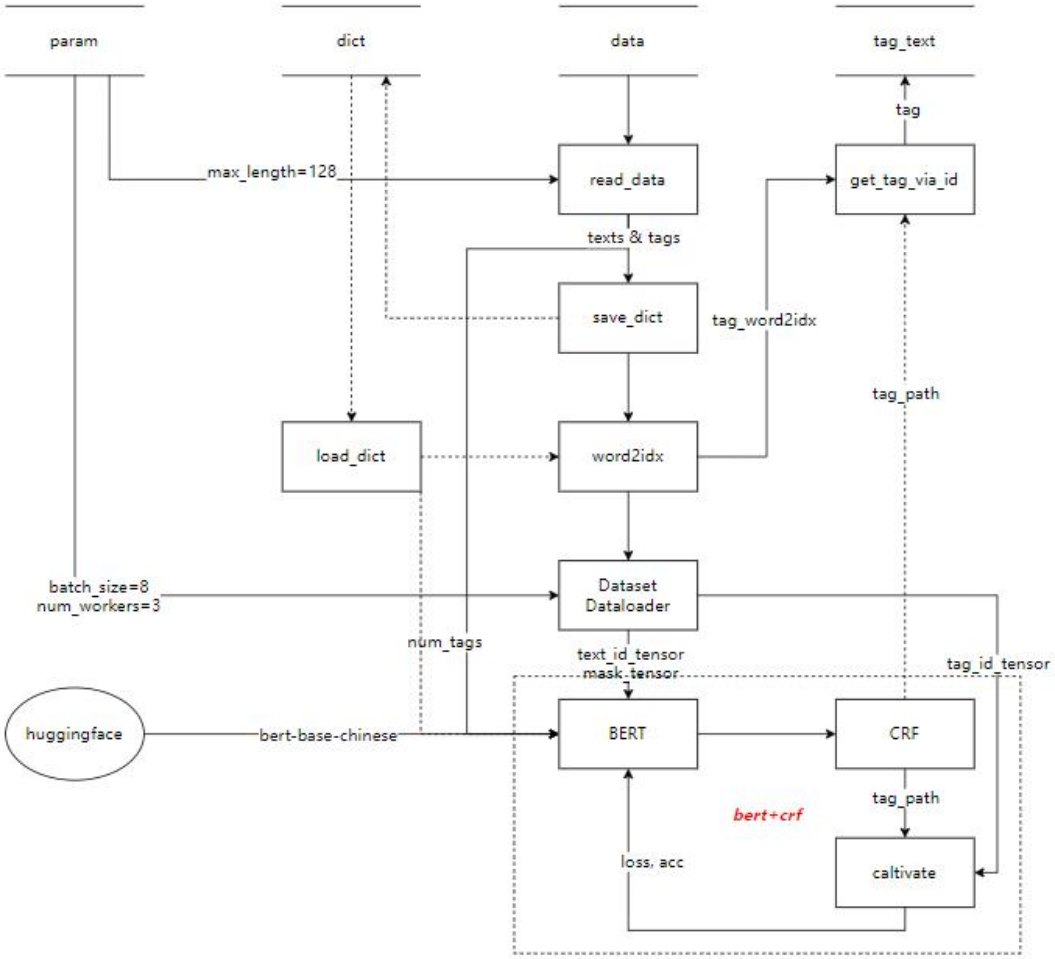
1、实验背景与目标

本实验旨在利用自然语言处理技术中的序列标注方法，针对特定任务进行模型构建与性能优化。实验的核心目标是利用 BERT-Base 模型与条件随机场（CRF）结合的框架，实现对中文文本的有效标注，并探索不同超参数配置、模型架构选择对任务性能的影响。

2、算法数据流图



3、数据标注集

从数据中提取所有的标注标签，创建一个不重复的词汇表（tag_word2idx），存在 tag_dict_v2.txt 中，方便使用：

```
# 定义一个函数来构建词汇表（for tag）
def build_vocab_v2(data):
    '''
    从给定的标注数据中构建词汇表，并生成字典形式的词汇表及其索引。

    :param data: list 包含标注数据的列表

    :return: 词汇表到索引的字典、索引到词汇表的字典、词汇表大小
    '''
```

```python
# 保存词表
def write_dict2file(path, word2idx):
    '''
    将词汇表写入文件。

    :param path: str 文件路径
    :param word2idx: dict 词汇表到索引的映射字典
    '''
```

```python
# 加载词表
def load_dict(path):
    '''
    从文件中加载词汇表。

    :param path: str 文件路径

    :return: 词汇表到索引的字典、词汇表大小
    '''
```

```
B_LOC 0
I_PER 1
I_ORG 2
B_ORG 3
B_PER 4
O 5
I_LOC 6
B_T 7
I_T 8
```

4、模型与执行细节

（1）模型结构及其参数

使用 BERT-Base 作为基本架构，包含 12 层 Transformer 块，隐藏层尺寸为 768。

输入序列最大长度为 512 个 tokens，这里综合内存压力设定预处理最大长度为 128 个 Tokens。

读取全部数据并按空格切分输入，每 128 维作为一个 tensor，未在训练集中见过的字记为<O>，最后一个用<PAD>补全，对应 mask 用 0 填充，对应 tag 用 O 补全。

```python
# 读取训练验证文本数据
def read_data_v2(file_path, length=128, padding_value='<PAD>'):
    '''
    从给定文件路径中读取文本数据，并按指定长度进行分割和填充。

    :param file_path: str 文件路径
    :param length: int, optional 分割长度，默认为128
    :param padding_value: str, optional 填充值，默认为'<PAD>'

    :return: 分割后的文本数据列表、注意力掩码列表
    '''
```

```python
# 读取训练验证tag数据
def read_tag_data_v2(file_path, length=128, padding_value='O'):
    '''
    从给定文件路径中读取标签数据，并按指定长度进行分割和填充。

    :param file_path: str 文件路径
    :param length: int, optional 分割长度，默认为128
    :param padding_value: str, optional 填充值，默认为'O'

    :return: 分割后的标签数据列表
    '''
```

在 BERT 的输出之上添加条件随机场层（CRF），用于序列标注任务，优化边界处理（单独使用 BERT 时，模型在序列的起始和结束处的标签预测可能不够准确，没有足够的上下文信息来确定一个序列的准确开始或结束）和标签依赖性（尽管 BERT 通过 Transformer 架构捕获了文本的上下文信息，生成了每个单词的高维语义表示，但它本质上是一个基于独立决策的模型，每个位置的标签预测是基于当前位置的特征向量独立做出的，不直接考虑相邻标签之间的相互依赖关系）。

理论上来说，添加 CRF 层可以带来以下好处：

①性能提升

通过结合 BERT 和 CRF，模型能够在保留深度学习模型强大特征提取能力的同时，增强序列标注的全局视角，解决序列内部的标签依赖性和边界问题，提升精确度、F1 分数等性能指标。

②训练与推断差异

BERT 模型的训练是端到端的，而 CRF 层的加入涉及 Viterbi 解码算法来找到最佳路径，在训练阶段增加了一定的复杂度，但在推断时通过动态规划可以高效完成标注任务。

③灵活性与复杂性

虽然 CRF 增加了模型的复杂性，但也提供了更高的灵活性，允许对序列结构和标签间的关系进行细粒度控制，尤其适合于具有复杂标注规则的语言处理任务。

由于时间和语料关系，并没有实际对比验证。

打印模型结构如下：

```
model ok to cpu BERT_NER(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(21128, 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)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.2, inplace=False)
  (classifier): Linear(in_features=768, out_features=9, bias=True)
  (crf): CRF()
)
```

具体参数可见下表：

名称	值	说明
batch_size	设置为 8。	训练批次大小，内存综合考虑的结果。
num_workers	设置为 0。	Dataloader 加载线程，同样是内存综合考虑的结果。
drop	默认值为 0.1。 设置为 0.2。	Dropout，在训练中随机丢弃节点的比例。如果不丢，在小数据上极容易过拟合。
lr	设置为 1e-6。	学习率决定了在每次迭代中参数更新的幅度。学习率过高可能会导致训练不稳定，对于 BERT 大模型来讲，问题尤为突出；而学习率过低则可能导致训练过程缓慢。
weight_decay	设置为 1e-6。	学习率衰减，实际上并没有使用。
Epochs	设置为 10。	依据验证集性能提前停止（这里设计为连续 2 个 epoch 验证准确率无提升则停止，并没有实现通常意义上基于损失和性能的早停策略）。

（2）执行细节

①内存管理

对不再使用的变量，及时释放，保证内存高效利用。如训练之前，当读入数据转化为索引之后，便释放内存，保证后续 load data 时不被 killed：

```
# 将字或词转换为索引
train_data_idx = get_idx_v2(train_data) # , data_word2idx)
train_tag_idx = get_idx(train_tags, tag_word2idx)
del train_data, train_tags

dev_data, dev_mask = read_data_v2(dev_file_path_data, max_length)
dev_tags = read_tag_data_v2(dev_file_path_tags, max_length)
dev_data_idx = get_idx_v2(dev_data) # , data_word2idx)
dev_tag_idx = get_idx(dev_tags, tag_word2idx)
del dev_data, dev_tags

del data_word2idx, tag_word2idx
```

②初始字向量

统计训练文本数据中的字，我们可以建立一个词表：

```
# 定义一个函数来构建词汇表（for data，增加未识别词'<O>'）
def build_vocab(data):
    '''
    从给定的文本数据中构建词汇表，并生成字典形式的词汇表及其索引。

    :param data: list 包含文本数据的列表

    :return: 词汇表到索引的字典、索引到词汇表的字典、词汇表大小
    '''
```

因此，初始字向量可以考虑两种方法：自定义（上）/利用 BERT 预训练模型提供的嵌入（下），分别对应（不）使用 bert-base-chinese 预训练模型，即下面展示的 radom 和 pre_trained。

```
# without pretrain and use own vocab
def get_idx(data, word2idx):
    '''
    将给定的数据转换为索引形式，使用自定义的词汇表进行索引映射。

    :param data: list 包含数据的列表
    :param word2idx: dict 词汇表到索引的映射字典

    :return: 包含数据索引的列表
    '''
```

```
# use pretrained
def get_idx_v2(data):
    '''
    使用BERT预训练模型对应的Tokenizer将给定的数据转换为索引形式。

    :param data: list 包含数据的列表
    :param word2idx: dict 词汇表到索引的映射字典

    :return: 包含数据索引的列表
    '''
```

这时候注意到关于预训练模型的获取问题：

如果直接下载没有外网资源是没有办法访问 huggingface 的，因而我们开启 VPN，从 https://huggingface.co/google-bert/bert-base-chinese/tree/main[1] 中下载模型（包括 pytorch_model.bin 等）到默认.cache 中，之后打包放在训练机器的.cache 中即可。

```
u2021213513@n1:~/.cache$ ls
conda  fontconfig  huggingface  matplotlib  pip  torch
u2021213513@n1:~/.cache$ cp /home/u2021213513/jupyterlab/homework2/huggingface ./
cp: -r not specified; omitting directory '/home/u2021213513/jupyterlab/homework2/huggingface'
u2021213513@n1:~/.cache$ cp -r /home/u2021213513/jupyterlab/homework2/huggingface ./
u2021213513@n1:~/.cache$
```

这里注意，若我们默认本地运行，只会下载如下文件：

则后期加载还需要联网访问.bin、.h5 等模型文件，因而必须从上述链接下载。

目录下有以下文件即可在未来离线状态下正常加载分词器和预训练模型。

③学习率

使用 AdamW 优化器，初始学习率设为 1e-5，根据验证集性能动态调整（如使用学习率衰减）。

注意到，学习率若太大，则预训练的模型不会继续往下训练。如我们在小数据上的训练（早期验证模型正确性时使用）：

```
input_ids = torch.tensor([[1, 2, 3, 4, 1, 2, 3, 4, 4]])
attention_mask = torch.tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1]])
input_labels = torch.tensor([[0, 1, 2, 3, 0, 1, 2, 3, 3]])
```

当 lr=1e-3：

```
Dev Accuracy:  0.3333
Epoch 92/100, Loss:  11.4937
Epoch 93/100, Loss:  11.4407
Epoch 94/100, Loss:  11.6149
Epoch 95/100, Loss:  10.4548
Epoch 96/100, Loss:  12.1212
Epoch 97/100, Loss:  11.1877
Epoch 98/100, Loss:  12.3488
Epoch 99/100, Loss:  12.0470
Epoch 100/100, Loss:  10.8371
```

```
Dev Accuracy:  0.3333
Epoch 92/100, Loss:  10.8351
Epoch 93/100, Loss:  10.5498
Epoch 94/100, Loss:  10.8829
Epoch 95/100, Loss:  11.3872
Epoch 96/100, Loss:  10.7803
Epoch 97/100, Loss:  11.1146
Epoch 98/100, Loss:  11.0301
Epoch 99/100, Loss:  11.0314
Epoch 100/100, Loss:  10.9566
```

pre_trained　　radom

即使 epoch 增大，Loss 也几乎不变，训练准确率也极低。

当减小至 1e-5：

```
Dev Accuracy:  0.8889
Epoch 62/100, Loss:  8.1671
Epoch 63/100, Loss:  6.9661
Epoch 64/100, Loss:  7.0453
Epoch 65/100, Loss:  6.9780
Epoch 66/100, Loss:  5.5116
Epoch 67/100, Loss:  5.9154
Epoch 68/100, Loss:  5.5013
Epoch 69/100, Loss:  7.6641
Epoch 70/100, Loss:  4.7123
Epoch 71/100, Loss:  4.9387
Dev Accuracy:  1.0000
```

```
Dev Accuracy:  0.4444
Epoch 2/100, Loss:  9.7862
Epoch 3/100, Loss:  7.0631
Epoch 4/100, Loss:  5.9138
Epoch 5/100, Loss:  4.0974
Epoch 6/100, Loss:  2.5380
Epoch 7/100, Loss:  1.7397
Epoch 8/100, Loss:  1.0472
Epoch 9/100, Loss:  0.8288
Epoch 10/100, Loss:  0.3530
Epoch 11/100, Loss:  0.3823
Dev Accuracy:  1.0000
Epoch 12/100, Loss:  0.2478
```

pre_trained　　radom

能够达到训练效果。这里数据直接给的编号，没有什么特殊意义，所以相比预训练的微调来讲，直接随机初始化会更快，但训练中我们可以发现，如果给定大数据集从头开始训练大模型，无疑是巨大的挑战。

比如我们最开始训练时，无论如何调整学习率等参数，模型标注结果几乎全为 O，导致测试准确率保持在 90%上下不变。

对于优化器，Adam 更适合小数据，而在训练我们给的数据的时候，使用 AdamW。

在训练中，初始学习率 1e-5 输出：

pre_trained　　radom

考虑可能还是学习率过大。

使用 pretrain，选择 1e-6：

Loss 下降，2 个 epoch 内准确率上升至 99%，在测试集上能够标注，在 test 下也能基本实现标注：

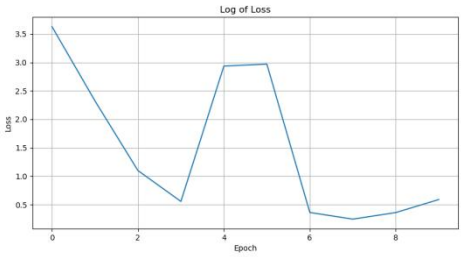
```
 1, 1, 5, 5, 5, 5, 5, 5, 5, 5, 0, 6, 6, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 7, 8,
 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
 5, 5, 5, 5, 5]]
Epoch 1/1, Loss: 9.4775, Dev Accuracy: 0.9897
```

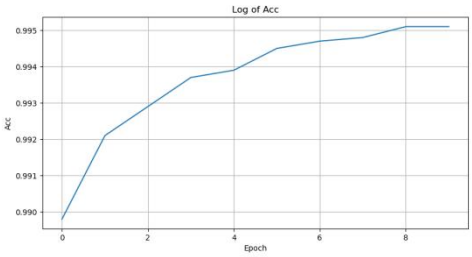
这里不再进一步考虑降低学习率，因为 1e-6 的学习率已经带来很大时间开销。

5、损失与准确率曲线

训练过程中，每一轮记录训练损失，每轮在发展集上评估模型，记录标注准确率。选择在验证集性能（这里用准确率来评估）最佳的那一轮作为最终模型。

```
model ok to cuda
evaling
Epoch 1/10, Loss: 3.6303, Dev Accuracy: 0.9898
evaling
Epoch 2/10, Loss: 2.3248, Dev Accuracy: 0.9921
evaling
Epoch 3/10, Loss: 1.0987, Dev Accuracy: 0.9929
evaling
Epoch 4/10, Loss: 0.5552, Dev Accuracy: 0.9937
evaling
Epoch 5/10, Loss: 2.9363, Dev Accuracy: 0.9939
evaling
Epoch 6/10, Loss: 2.9710, Dev Accuracy: 0.9945
evaling
Epoch 7/10, Loss: 0.3606, Dev Accuracy: 0.9947
evaling
Epoch 8/10, Loss: 0.2436, Dev Accuracy: 0.9948
evaling
Epoch 9/10, Loss: 0.3589, Dev Accuracy: 0.9951
evaling
Epoch 10/10, Loss: 0.5882, Dev Accuracy: 0.9951
```





可以看到 loss 相对来说是不太稳定的，主要原因在于这里 log 的 loss 只是每次最后一个 batch 的，正确的做法应该记录每个 batch 的平均。

最后选择保存的最早最优性能（ACC= 0.9951）的模型做测试，测试时，数据按行输入

并标注。

6、反思和改进

（1）学习率的进一步探究

调整学习率并观察，引入衰减并尝试。

（2）模型性能评估

引入更丰富的评估指标，如 F1 分数、召回率和精确度，结合交叉验证，优化模型评估应用。

（3）指定模型层进行对应微调

在实验中，我们对整个 bert+crf 进行微调，未来可以考虑指定模型层，让训练更加高效。

（4）gpt、T5 的对比研究

本次实验由于时间有限，前期数据处理、内存优化到后期模型训练均耗时严重，未能对比编码器、解码器、编码器-解码器三种不同架构下 transformer 的具体表现和优劣。理论上来讲：

		BERT	GPT（2）	T5
结构	Transformer	12	24	11
	Hidden_size	768	768	512
	Tokens	512	1024	512
中文支持		bert-base-chinese	gpt2-chinese-cluecorpussmall	t5-small-chinese-cluecorpussmall
优		双向理解	单向处理 文本生成 模型简洁	灵活通用
劣		生成能力有限 内存占用大	缺乏双向理解	资源需求高 训练复杂

对于序列标注任务，这三类模型各有怎样的特点和不足，还需要进一步实验验证，加深理解。

7、参考文献

本项目直接利用了预训练的 BERT 大模型进行序列标注任务实现，参考了课程 PPT 与 huggingface 教程，未额外引入其他大模型辅助。

这里列出参考链接：

[1]https://huggingface.co/google-bert/bert-base-chinese/tree/main

[2]https://gitcode.com/qukequke/bert-crf-token_classification_ner/overview?utm_source=artical_gitcode&isLogin=1