医学命名实体识别模型的改进

张哲昊 5190309103835　曾智霖 5190309103835

日期：2022 年 6 月 16 日

摘　要

本次项目综合运用了各种方法提升模型的命名实体的性能，其中曾智霖负责实现了加入 flat 模型，张哲昊负责对抗训练，加入预训练词向量，学习率逐层衰减，预训练语言模型。最优实验结果相对于 baseline 有明显的提升。

1　加入 FLAT 模型

整体思路为在原有 BERT 模型的基础上，将 BERT 模型得到的字的向量与预训练词向量进行拼接，从而加入词的知识，然后输入 FLAT 模型。模型整体流程如图 1 所示：

1.1　分词

本次实验用到的分词工具库为 **pkuseg**。这是一个由北京大学语言计算与机器学习研究组进行开发并开源的项目，它比其他的流行的分词工具比如 jieba 要好的一个优点在于它还匹配了专业领域的分词模型，比如我们这次要用到的医药领域。

```
import pkuseg
 pku_seg = pkuseg.pkuseg(model_name='medicine')
```

在参数中指定 model_name='medicine' 即能够用专门的医药领域的分词模型进行分词，实验发现得到的分词效果确实比 jieba 要好。我们利用 pkuseg 工具库进行分词，并得到该词的起始和终止的位置，从而构建出 lattice 的形式。代码如下所示：

```
def get_lattice_word(text):
    lattice_word = pkuseg_cut(text)
    # 按 词长度 升序    按 结束位置 升序
    lattice_word.sort(key=lambda x: len(x[0]))
    lattice_word.sort(key=lambda x: x[2])
    return lattice_word

def is_all_chinese(word_str):
    for c in word_str:
        if not '\u4e00' <= c <= '\u9fa5':
            return False
    return True
```

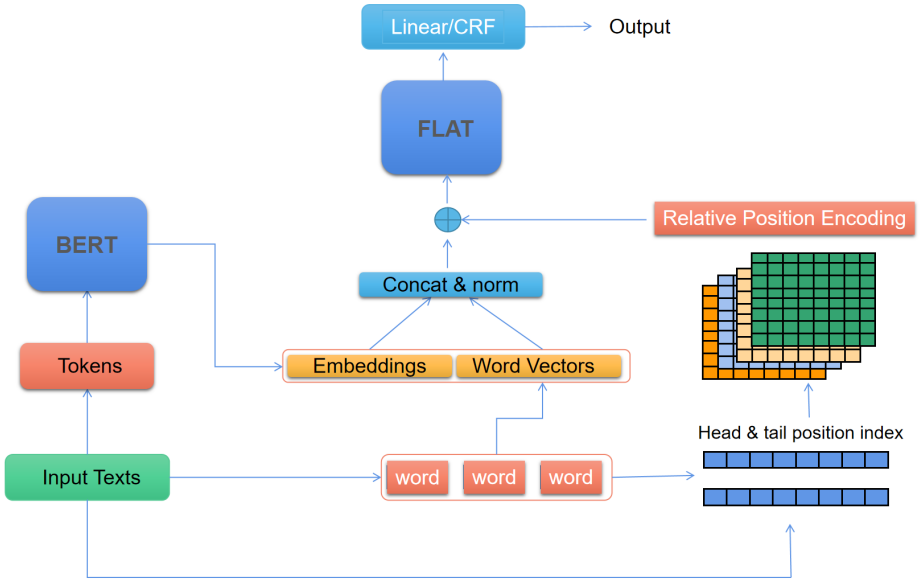

图 1: 加入 flat 模型的改进思路

```
def pkuseg_cut(text):
    #获取 lattice 所需的数据:  [(word, start idx, end idx), ...]
    index = 0
    word_list = []
    for word in pku_seg.cut(text):
        word_len = len(word)
        if word_len > 1 and is_all_chinese(word):
            word_list.append((word, index, index + word_len - 1))
        index += word_len
    return word_list
```

处理过程的一个例子如下:

```
text: 2.肾盂、膀胱及输尿管结石。
tokens: ['[CLS]', '2', '.', '肾', '盂', '、', '膀', '胱', '及', '输', '尿', '管', '
    结', '石', '。', '[SEP]']
token_ids: [101, 123, 119, 5513, 4655, 510, 5598, 5537, 1350, 6783, 2228, 5052,
    5310, 4767, 511, 102]
lattice: [('肾盂', 2, 3), ('膀胱', 5, 6), ('输尿管', 8, 10), ('结石', 11, 12)]
lattice_idx: [[3, 4, 91227], [6, 7, 19326], [9, 11, 44793], [12, 13, 25642]]
```

1.2 获取词向量

本次实验直接使用预训练 word2vec 获得词向量。我们选择了 https://github.com/Embedding/Chinese-Word-Vectors 里的 Mixed-large 综合。下载后得到‘sgns.merge.word’文件，我们从中提取出 vocab 词表和 vec 词向量, 并分别存入 w2v_vocab.pkl 和 w2v_vector.pkl 中。

```
def conv2pkl():
    raw_file = ROOT_LOCAL_DATA + 'sgns.merge.word'
```

```
    vocab_file = ROOT_LOCAL_DATA + 'w2v_vocab.pkl'
    vec_file = ROOT_LOCAL_DATA + 'w2v_vector.pkl'
    raw_fp = open(raw_file, 'r', encoding='utf-8')
    vocab_fp = open(vocab_file, 'wb+')
    vec_fp = open(vec_file, 'wb+')
    raw_data = raw_fp.readlines()[1:]
    vocab_list = {'PAD': 0}
    vec_list = [[0.0] * 300]
    for index, d in enumerate(raw_data):
        d = d.split()
        vocab_list[d[0]] = index + 1
        vec = [float(s) for s in d[1:]]
        vec_list.append(vec)

    pickle.dump(vocab_list, vocab_fp)
    pickle.dump(vec_list, vec_fp)
```

首先根据之前分词得到的结果，去查找得到的词是否在词表中，如果存在则获取该词在词表中对应的 index。

```
word_dict = pickle.load(open('w2v_vocab.pkl', 'rb'))
word_dict = {word: idx for idx, word in enumerate(word_dict)}
lattice=[]
lattice = get_lattice_word(text)
lattice_idx = []  # [start idx, end idx, 词在 word2vec中的index]
for lword in lattice:
    if (int(lword[2])+2)>=self.max_length: break #在最大长度之后的词舍去
    if lword[0] in word_dict:
        lword_index = word_dict[lword[0]]
        lattice_idx.append([lword[1]+1, lword[2]+1, lword_index])#生成的token开头加
            了CLS token，所以词对应的位置要+1。
```

最终在原来的 ee_data.py 修改输入为：

```
 inputs = {
            "input_ids": torch.tensor(input_ids, dtype=torch.long),
            "attention_mask": attention_mask,
            "labels": torch.tensor(labels, dtype=torch.long) if labels is not None
                else None,
            "no_decode": no_decode_flag,
            "char_lens": char_len,
            "lattice": lattice
            }
```

添加了两项，一个是”char_lens” 表示字的长度，在之后 embedding 拼接时要用到。“lattice” 存的则是每个词对应在 tokens 中的起始和终止位置以及在 word2vec 中的 index。

```
self.w2v_array = pickle.load(open('w2v_vector.pkl', 'rb'))
 vec = torch.tensor(self.w2v_array[int(word_idx)]).float().to(device)
```

最后根据词的 index 找到对应的预训练词向量。

1.3 构建 FLAT 模型输入

Flat 模型的输入需要 5 个部分，分别是字词拼接后的 embedding 向量，对应的 mask，head position index，tail position index 以及字的长度。

```
char_vec = sequence_output     # 从BERT的输出得到字的embbeding向量

char_word_vec = []   # 用于暂时存放字向量与词向量拼接的结果
max_word_len=max(map(len, lattice))
max_len=char_vec.shape[1]+max_word_len  #得到字加词总的最大长度，用于之后的padding

char_word_mask = torch.zeros((8, max_len), dtype=torch.long) #生成字与词拼接向量对
    应的mask，第一个参数8对应batch size

pos = torch.arange(0, char_vec.size(1)).long().unsqueeze(dim=0).to(device)
pos = pos * attention_mask.long()
pad = torch.tensor([0 for _ in range(max_len - attention_mask.size(1))]).unsqueeze
    (0).repeat(attention_mask.size(0), 1).to(device)
pos = torch.cat((pos, pad), dim=1)   #得到字对应的position index，并且pad到同一长度

char_word_head = pos.clone()   #Head position index
char_word_tail = pos.clone()   #Tail position index 此时两者相等因为只包含字的index
    。
for i, bchar_vec in enumerate(char_vec):
    bert_vec = []
    word_vec = []
    pad_vec = []
    for idx, vec in enumerate(bchar_vec):
        if idx < char_lens[i]:
            bert_vec.append(vec)  #得到所有的字向量
    lattice_per_batch=lattice[i]

    for idx, word in enumerate(lattice_per_batch):
        word_idx=word[2]
        vec = torch.tensor(self.w2v_array[int(word_idx)]).float().to(device) #根据
            word index得到对应的词向量
        word_vec.append(vec)  #得到所有的词向量
        char_word_head[i][len(bert_vec)+idx]=word[0] #词的start index对应到Head
            position index
        char_word_tail[i][len(bert_vec)+idx]=word[1] #词的end index对应到Tail
            position index,接在字的index之后
    bert_vec = torch.stack(bert_vec, dim=0).to(device) #2维

    if len(word_vec) > 0:
        word_vec = torch.stack(word_vec, dim=0).to(device)  # 2维
```

```
            word_vec = self.w2v_linear(word_vec)   #通过一个全连接层让词向量维度等于字
                向量维度

            new_vec = torch.cat((bert_vec, word_vec), dim=0)  # 2维, 字向量和词向量拼接
        else:
            new_vec = bert_vec  # 2维
        new_vec = self.layer_norm(new_vec)
        new_vec = self.dropout(new_vec)  #拼接后的向量做一个norm和dropout操作
        char_word_mask[i][:new_vec.shape[0]] = 1  #构建字词向量对应的mask
        pad_len=max_len-new_vec.shape[0]

        for pad in range(pad_len):
            pad_vec.append(torch.zeros(768).to(device))
        if len(pad_vec) > 1:
            pad_vec = torch.stack(pad_vec, dim=0).to(device) # 2维
            new_vec = torch.cat((new_vec, pad_vec), dim=0)
        elif len(pad_vec) == 1:
            pad_vec = pad_vec[0].unsqueeze(0)
            new_vec = torch.cat((new_vec, pad_vec), dim=0) #将所有字词向量pad到同一长度

        char_word_vec.append(new_vec)
    char_word_vec = torch.stack(char_word_vec, dim=0).float().to(device) #最后得到整个
        batch对应的字词向量

    #构建出FLAT模型的输入形式
    encoder_outputs={'char_word_vec': char_word_vec,   #字词拼接向量
            'char_word_mask': char_word_mask.to(device).bool(), #字词拼接向量对应得mask
            'char_word_s': char_word_head, #Head position index
            'char_word_e': char_word_tail, #Tail position index
            'char_len': attention_mask.size(1)} #字的长度
    fusion_outputs = self.flat(encoder_outputs)
```

1.4 FLAT 模型

1.4.1 相对位置编码

在 FLAT 原文中定义了四种相对距离

$$\begin{aligned} d_{ij}^{(hh)} &= head[i] - head[j] \\ d_{ij}^{(ht)} &= head[i] - tail[j] \\ d_{ij}^{(th)} &= tail[i] - head[j] \\ d_{tt}^{(hh)} &= tail[i] - tail[j] \end{aligned} \tag{1}$$

基于此构建相对位置编码:

$$R_{ij} = ReLU(W_r(P_{d_{ij}^{(hh)}} \oplus P_{d_{ij}^{(ht)}} \oplus P_{d_{ij}^{(th)}} \oplus P_{d_{ij}^{(tt)}}) \tag{2}$$

其中 P_d 的计算方式为:

$$P_d^{2k} = sin(d/10000^{2k/d_{model}})$$
$$P_d^{2k+1} = cos(d/10000^{2k/d_{model}}) \tag{3}$$

代码实现如下:

```
def get_pos_embedding(max_seq_len,embedding_dim,padding_idx=None,  ):
    '''
    根据公式  pd^(2k)=sin(d/10000^{2k/d_model})
             pd^(2k+1)=cos(d/10000^{2k/d_model})
    计算position embedding
    '''
    num_embeddings = 2 * max_seq_len + 1
    half_dim = embedding_dim // 2
    emb = math.log(10000) / (half_dim - 1)
    emb = torch.exp(torch.arange(half_dim, dtype=torch.float) * -emb)
    emb = torch.arange(num_embeddings,dtype=torch.float).unsqueeze(1) * emb.
        unsqueeze(0)
    emb = torch.cat([torch.sin(emb), torch.cos(emb)],
                    dim=1).view(num_embeddings, -1)
    if embedding_dim % 2 == 1:
        # zero pad
        emb = torch.cat([emb, torch.zeros(num_embeddings, 1)], dim=1)
    if padding_idx is not None:
        emb[padding_idx, :] = 0
    return emb
```

```
class FourPosFusionEmbedding(nn.Module):
    "FLAT 位置编码"
    def __init__(self, fusion_method, pe_ss, pe_se, pe_es, pe_ee, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.pe_ss = pe_ss
        self.pe_se = pe_se
        self.pe_es = pe_es
        self.pe_ee = pe_ee  #四种位置关系

        self.fusion_method = fusion_method
        self.w_r = nn.Linear(self.hidden_size, self.hidden_size)
        self.pos_attn_score = nn.Sequential(
            nn.Linear(self.hidden_size * 4, self.hidden_size * 4),
            nn.ReLU(),
            nn.Linear(self.hidden_size * 4, 4),
            nn.Softmax(dim=-1)
        )         #计算注意力score
```

```
    def forward(self, pos_s, pos_e):
        batch = pos_s.size(0)
        max_seq_len = pos_s.size(1)

        pos_ss = pos_s.unsqueeze(-1) - pos_s.unsqueeze(-2)
        pos_se = pos_s.unsqueeze(-1) - pos_e.unsqueeze(-2)
        pos_es = pos_e.unsqueeze(-1) - pos_s.unsqueeze(-2)
        pos_ee = pos_e.unsqueeze(-1) - pos_e.unsqueeze(-2)

        # 调整相对位置编码的shape为[batch_size,max_seq_len,max_seq_len,hidden_dim]
        pe_ss = self.pe_ss[(pos_ss).view(-1) + max_seq_len].view(
            size=[batch, max_seq_len, max_seq_len, -1])
        pe_se = self.pe_se[(pos_se).view(-1) + max_seq_len].view(
            size=[batch, max_seq_len, max_seq_len, -1])
        pe_es = self.pe_es[(pos_es).view(-1) + max_seq_len].view(
            size=[batch, max_seq_len, max_seq_len, -1])
        pe_ee = self.pe_ee[(pos_ee).view(-1) + max_seq_len].view(
            size=[batch, max_seq_len, max_seq_len, -1])

        pe_4 = torch.cat([pe_ss, pe_se, pe_es, pe_ee], dim=-1)#四种位置关系拼接在一
            起
        attn_score = self.pos_attn_score(pe_4)
        pe_4_unflat = self.w_r(pe_4.view(
            batch, max_seq_len, max_seq_len, 4, self.hidden_size))
        pe_4_fusion = (attn_score.unsqueeze(-1) * pe_4_unflat).sum(dim=-2)
        rel_pos_embedding = pe_4_fusion

        return rel_pos_embedding
```

1.5 Flat 模型实现

```
class FLAT(nn.Module):
    def __init__(self):
        super(FLAT, self).__init__()
        self.params = {'other': []}

        pe = get_pos_embedding(Config.max_len, Config.dim_pos) #获取position
            embedding, max_len要确保大于字和词加起来的总长度

        self.pe = nn.Parameter(pe, requires_grad=Config.learnable_position)#设置
            position embedding 是否可学习

        if Config.four_pos_shared: #是否共享每种 position embedding 的参数
            self.pe_ss = self.pe
            self.pe_se = self.pe
            self.pe_es = self.pe
```

```
            self.pe_ee = self.pe
        else:
            self.pe_ss = nn.Parameter(copy.deepcopy(pe), requires_grad=Config.
                learnable_position)
            self.pe_se = nn.Parameter(copy.deepcopy(pe), requires_grad=Config.
                learnable_position)
            self.pe_es = nn.Parameter(copy.deepcopy(pe), requires_grad=Config.
                learnable_position)
            self.pe_ee = nn.Parameter(copy.deepcopy(pe), requires_grad=Config.
                learnable_position)

        self.pos_layer = FourPosFusionEmbedding(self.pe_ss, self.pe_se, self.pe_es,
             self.pe_ee,Config.hidden_size)

        # flat layers
        # 就是一层tranformer的encoder, 利用现成的实现
        self.encoder_layers = []
        for _ in range(Config.num_flat_layers):
            encoder_layer = TransformerEncoderLayer(Config)
            self.encoder_layers.append(encoder_layer)
            self.params['other'].extend([p for p in encoder_layer.parameters()])
        self.encoder_layers = nn.ModuleList(self.encoder_layers)

    def get_params(self):
        return self.params

    def forward(self, inputs):
        char_word_vec = inputs['char_word_vec']
        char_word_mask = inputs['char_word_mask']
        char_word_s = inputs['char_word_s']
        char_word_e = inputs['char_word_e']
        char_len = inputs['char_len']

        pos_embedding = self.pos_layer(char_word_s, char_word_e)
        hidden = char_word_vec

        for layer in self.encoder_layers:
            hidden = layer(hidden, pos_embedding, char_word_mask)

        # 只取char token的输出
        char_vec = hidden[:, : char_len, :]

        return char_vec
```

1.6 测试结果

flat 模型是在之前 linear nested 基础上加的，其他参数配置，训练条件都一样，可以看到效果提升了 0.7 个点。需要注意的是加 flat 模型之后会占用更多显存，容易出现 cuda out of memory（计算 attention score 导致的），因此需要减小 max_length 参数，实验时将 max_length 参数从 512 减小到了 128，和 linear_nested 的对比都是在参数为 128 时的对比，并且由于资源有限每次只训练 5 个 epoch。实验还发现四种位置编码是否共享参数得到的效果基本相同，将位置编码设为可学习得到的效果要更差一些。

模型	状态	Score	CMeEE-F1	CMeEE-P	CMeEE-R	CMeIE-F1	CMeIE-P	CMeIE-
bert+flat	已完成	4.528	63.392	62.722	64.076	0.0	0.0	0.0
bert+linear_nested	已完成	4.477	62.684	62.168	63.208	0.0	0.0	

图 2: 加入 flat 模型后的测试结果

2 对抗训练

本次项目中尝试了两种不同的对抗训练方法：FGM 与 PGD。对抗训练（adversarial training）是增强神经网络鲁棒性的重要方式。在对抗训练的过程中，样本会被混合一些微小的扰动（改变很小，但是很可能造成误分类），然后使神经网络适应这种改变，从而对对抗样本具有鲁棒性。从优化层面来说，我们希望找到数据的扰动在一定范围内使损失函数最大，即是添加的扰动要尽量让神经网络迷惑。同时希望当扰动固定的情况下，我们训练神经网络模型使得在训练数据上的损失最小，即使模型具有一定的鲁棒性能够适应这种扰动。对抗训练能够作为一种正则化，提高模型的泛化能力

2.1 FGM (Fast Gradient Method)

FGM 的主要思想为在输入数据 x 上加上一个扰动 r_{adv}，得到对抗样本之后用于训练。将输入样本向着损失上升的方向再进一步，得到的对抗样本就能造成更大的损失，可以被抽象为以下形式：

$$\min_{\theta} -\log P(y|x + r_{adv}; \theta) \tag{4}$$

其中 y 为 label，θ 为模型参数。FGM 采用的是 L_2 归一化，即将梯度的每个维度的值除以梯度的 L_2 范数。具体实现代码如下：

```
class FGM:
    def __init__(self, model):
        self.model = model
        self.backup = {}

    def attack(self, epsilon=1, emb_name='emb.'):
```

```
        for name, param in self.model.named_parameters():
            if param.requires_grad and emb_name in name:
                self.backup[name] = param.data.clone()
                norm = torch.norm(param.grad)
                if norm != 0 and not torch.isnan(norm):
                    r_adv = epsilon * param.grad / norm
                    param.data.add_(r_adv)

    def restore(self, emb_name='emb.'):
        for name, param in self.model.named_parameters():
            if param.requires_grad and emb_name in name:
                assert name in self.backup
                param.data = self.backup[name]
        self.backup = {}
class CustomTrainer(Trainer):
    def training_step(self, model: torch.nn.Module, inputs: Dict[str, Union[torch.
        Tensor, Any]]) -> torch.Tensor:
        model.train()
        fgm = FGM(model)
        inputs = self._prepare_inputs(inputs)
        loss = self.compute_loss(model, inputs)
        loss.backward()
        fgm.attack()
        loss_adv = self.compute_loss(model, inputs)
        loss_adv.backward()
        fgm.restore()
        return loss_adv.detach()
```

具体实现过程中需要继承 Trainer 类然后重写 training_step 函数，然后实现每一步的对抗训练过程。

2.2 PGD (Projected Gradient descent)

PGD 是一种迭代攻击，相比于普通的 FGSM 和 FGM 仅做一次迭代，PGD 是做多次迭代（本次实验中设定为 3 步），每次走一小步，每次迭代都会将扰动投射到规定范围内。如果走出了扰动半径为 ϵ 的空间，就映射回“球面”上，以保证扰动不要过大，可以抽象为以下形式。

$$g_t = \nabla_{X_t} L(f_\theta(X_t), y) \tag{5}$$

$$X_{t+1} = \prod_{X+S} (X_t + \epsilon \frac{g_t}{||g_t||}) \tag{6}$$

其中 $\prod_{X+S}$ 的意思是，如果扰动超过一定的范围，就要映射回规定的范围 S 内。PGD 算法得到的攻击样本，是一阶对抗样本中最强的。如果模型对 PGD 产生的样本鲁棒，那几乎就对所有的一阶对抗样本都鲁棒。代码实现如下

```
class PGD:
```

```
    def __init__(self, model):
        self.model = model
        self.emb_backup = {}
        self.grad_backup = {}
    def attack(self,
               epsilon=1.,
               alpha=0.3,
               emb_name='emb.',
               is_first_attack=False):
        for name, param in self.model.named_parameters():
            if param.requires_grad and emb_name in name:
                if is_first_attack:
                    self.emb_backup[name] = param.data.clone()
                norm = torch.norm(param.grad)
                if norm != 0 and not torch.isnan(norm):
                    r_at = alpha * param.grad / norm
                    param.data.add_(r_at)
                    param.data = self.project(name, param.data, epsilon)

    def restore(self, emb_name='emb.'):
        for name, param in self.model.named_parameters():
            if param.requires_grad and emb_name in name:
                assert name in self.emb_backup
                param.data = self.emb_backup[name]
        self.emb_backup = {}

    def project(self, param_name, param_data, epsilon):
        r = param_data - self.emb_backup[param_name]
        if torch.norm(r) > epsilon:
            r = epsilon * r / torch.norm(r)
        return self.emb_backup[param_name] + r

    def backup_grad(self):
        for name, param in self.model.named_parameters():
            if param.requires_grad:
                self.grad_backup[name] = param.grad

    def restore_grad(self):
        for name, param in self.model.named_parameters():
            if param.requires_grad:
                param.grad = self.grad_backup[name]
        self.grad_backup = {}
class CustomTrainer(Trainer):
    def training_step(self, model: torch.nn.Module, inputs: Dict[str, Union[torch.
        Tensor, Any]]) -> torch.Tensor:
        model.train()
        pgd = PGD(model)
```

```
        K = 3
        inputs = self._prepare_inputs(inputs)
        loss = self.compute_loss(model, inputs)
        loss.backward()
        pgd.backup_grad()
        for t in range(K):
            pgd.attack(is_first_attack=(t==0))
            if t != K-1:
                model.zero_grad()
            else:
                pgd.restore_grad()
            loss_adv = self.compute_loss(model, inputs)
            loss_adv.backward()
        pgd.restore() # 恢复embedding参数
        return loss_adv.detach()
```

2.3 实验结果

PGD 和 FGM 对抗训练在测试集上的实验结果如下截图所示 从实验结果中看出 FGM 能够

2	2022-06-11 21:41:41	roberta-flat-medical-PGD	已完成	4.606	64.482	63.317	65.690	0.0	0.0
3	2022-06-11 12:16:31	roberta-flat-medical-FGM	已完成	4.596	64.349	63.046	65.707	0.0	0.0
4	2022-06-11 09:59:35	roberta-medical	已完成	4.583	64.155	62.645	65.740	0.0	0.0

图 3: PGD 和 FGM 对抗训练测试集上的实验结果

提升 0.194，PGD 能够提升 0.327，证明两种对抗训练方式均能提升模型的泛化能力从而提升模型的性能。

3 预训练词向量

之前使用的'sgns.merge.word' 虽然很大，但是发现包含的医学专业词汇并不是很多，发现有部分分词的结果在词表中找不到，因此为了更好的使用医学词汇信息，我们重新使用了一个专门的预训练医学词汇的词向量。预训练词向量来自于https://github.com/WENGSYX/Chinese-Word2vec-Medicine，将预训练词向量下载之后通过 gensim 进行加载，代码如下：

```
model = KeyedVectors.load_word2vec_format('Medical.txt', binary=False) #模型加载
tensor = torch.tensor(model.get_vector('大脑')) #得到词的词向量并转换成tensor
```

得到词向量之后经过全连接层，然后将输出与 BERT 的输出进行拼接，从而融合词汇信息。融合了预训练词向量之后的模型的实验结果如图4所示。实验结果表明更换了预训练词向量之后 F1 能有 0.246 的提升。

2	2022-06-13 09:15:59	roberta-flat-PGD-w2v	已完成	4.623	64.728	63.143	66.394	0.0
3	2022-06-11 21:41:41	roberta-flat-medical-PGD	已完成	4.606	64.482	63.317	65.690	0.0

图 4: 加入预训练词向量的实验结果

4 逐层学习率衰减

本次项目尝试了对训练过程中的学习率进行逐层衰减，实现思路为遍历模型的所有层的可调参数，建立一个参数与学习率对应的字典，在逐渐增加参数层数的过程中逐渐乘以一个衰减系数，但是同一层的不同参数的学习率保持不变（例如同一层的 weights 和 bias）。然后创建一个优化器（如 AdamW），将字典传入优化器，然后将优化器作为参数传入到 Trainer 的 optimizer 参数中。代码实现如下：

```
layer_names = []
for idx, (name, param) in enumerate(model.named_parameters()):
    layer_names.append(name)
lr = 3e-5
lr_mult = 0.70 # decay ratio
parameters      = []
prev_group_name = layer_names[5].split('.')[3]
for idx, name in enumerate(layer_names):

    # parameter group name
    if name.split('.')[1] == 'encoder':
        cur_group_name = name.split('.')[3]

        # update learning rate
        if cur_group_name != prev_group_name:
            lr *= lr_mult
        prev_group_name = cur_group_name
    print(f'{idx}: lr = {lr:.6f}, {name}')

    # append layer parameters
    parameters += [{'params': [p for n, p in model.named_parameters() if n ==
        name and p.requires_grad],
                    'lr':     lr}]
optimizer = AdamW(parameters)
```

学习率衰减之后的实验结果如图 6所示。实验结果均相对 baseline 有一定的下降。说明该方法并不适合本次任务。

序号	提交时间	模型	状态	Score	CMeEE-F1	CMeEE-P	CMeEE-R	CMeIE-F1	CMeIE-P
11	2022-06-08 20:40:50	decay-lr-3e-5-0.8	已完成	4.493	62.905	61.569	64.300	0.0	0.0
12	2022-06-08 19:58:04	decay-lr-3e-5-0.95	已完成	4.493	62.905	61.569	64.300	0.0	0.0
13	2022-06-08 17:16:31	decay-lr-3e-5-0.9	已完成	4.501	63.010	61.454	64.647	0.0	0.0

图 5: 学习率逐层衰减的实验结果

5 预训练语言模型

本次项目中使用了三种预训练语言模型：bert-base-chinese，chinese-roberta-wwm-ext，chinese-roberta-wwm-ext-large。三种模型中 chinese-roberta-wwm-ext-large 的效果最好，加上 FLAT 模型，对抗训练啊，预训练词向量之后得到效果最好的实验结果如下：

序号	提交时间	模型	状态	Score	CMeEE-F1	CMeEE-P	CMeEE-R	CMeIE-F
1	2022-06-14 09:55:27	roberta-large-flat-PGD-w2v	已完成	4.717	66.032	64.384	67.767	0.0

图 6: 最优实验结果

验证集上的 score 为 0.6300323530127223。