基于BiLSTM-CRF的命名实体识别

刘帅 2020212267

实验简介

本次实验基于paddle和paddlenlp框架，基于BiLSTM-CRF算法实现了命名实体的识别。

本次实验的收获及贡献点如下：

- 了解了BiLSTM-CRF的算法逻辑
- 进一步熟悉了paddle框架，并在此基础上尝试paddle的高层api
- 调试paddle高阶api及回调函数，实现了训练流程的优化以及训练过程的可视化记录。
- 利用paddle高阶训练流程实现了自动更改学习率、earlystop等训练优化机制

目录组织形式

```
|-- 2020212267.txt #测试集的命名实体推理结果
|-- ddp_skip-gram.py #gpu并行代码#更改训练流程
|-- log    #tensorboard的可视化代码
|-- test.py #测试集命名实体生成代码
|-- train.py #加入回调函数和训练优化的代码
|-- emb300 #embedding=300的训练log
|-- emb600 #embedding=600的训练log
|-- n2e300 #embedding=300，layer=2的训练log
|-- n2e600 #embedding=600，layer=2的训练log
```

一、实验环境

由于本次实验设计了多个消融实验及比较，且为了方便调试，从而对paddle框架更加熟悉，本次实验的完整训练过程在自己的集群上进行：

在aistudio中仅展示部分训练流程，以表示可以在官方的paddle环境中成功训练。

```
Sun Apr 30 14:08:44 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.129.06   Driver Version: 470.129.06   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A100-SXM...  On   | 00000000:65:01.0 Off |                    0 |
| N/A   39C    P0    95W / 400W |  80333MiB / 81251MiB |     25%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA A100-SXM...  On   | 00000000:67:01.0 Off |                    0 |
| N/A   58C    P0   161W / 400W |  52199MiB / 81251MiB |     85%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
```

paddle环境部署

根据paddle文档内容配置paddle环境

```
python -m pip install paddlepaddle-gpu==2.4.2.post112 -f
https://www.paddlepaddle.org.cn/whl/linux/mkl/avx/stable.html
```

二、代码分析

2.1 导入相关package

BiLSTM-CRF的命名实体识别任务共包含两个框架，其一是比较low-level的paddle框架，类比mmlab中mmcv框架，其主要目的是搭建paddle环境下的基本训练流程；paddlenlp框架则支持了一些常见的自然语言处理任务。并指定利用0号gpu进行训练。

```
import paddle
import paddle.nn as nn
import paddlenlp
from paddlenlp.datasets import MapDataset
from paddlenlp.data import Stack, Tuple, Pad
from paddlenlp.layers import LinearChainCrf, ViterbiDecoder, LinearChainCrfLoss
from paddlenlp.metrics import ChunkEvaluator
import paddle.nn.initializer as I
paddle.set_device("gpu:0")
```

2.2 构建标注集

2.2.1 定义从语料、命名实体到id的映射

首先建立一个字典，给输入的语料和命名实体独特的标识，以便于在网络中进行后续训练。为保证表示的唯一性，根据当前maps的长度来赋值。

```
def build_map(lists):
    maps = {}
    for list_ in lists:
        for e in list_:
            if e not in maps:
                maps[e] = len(maps)

    return maps
```

以训练集语料和命名实体作为标注集来源进行构建，并且在word_vocab的标注集中添加OOV，表示针对Out-Of-Vocabulary部分的数据用OOV进行替换，**该部分将在2.2.3节中进行使用。**

```
word_lists,tag_lists=[],[]
with open("/mnt/ve_share/liushuai/lstm/train.txt",encoding="utf-8")as f:
    with open("/mnt/ve_share/liushuai/lstm/train_TAG.txt",encoding='utf-8')as t:
        lines=f.readlines()
        labels=t.readlines()
        for line in lines:
            word_lists.append(line.strip().split(' '))
        for label in labels:
            tag_lists.append(label.strip().split(' '))
        word_vocab = build_map(word_lists)
        label_vocab = build_map(tag_lists)
word_vocab.update({'OOV':len(word_vocab)})
```

构建后的标注集如下：

```
→ word_vocab
> {'人': 0, '民': 1, '网': 2, '1': 3, '月': 4, '日': 5, '讯': 6, '据': 7, '《': 8, '组': 9, '约': 10, '时': 11, '报': 12, '》': 13, ...}
→ label_vocab
> {'O': 0, 'B_T': 1, 'I_T': 2, 'B_LOC': 3, 'I_LOC': 4, 'B_ORG': 5, 'I_ORG': 6, 'B_PER': 7, 'I_PER': 8}
```

2.2.2 构建dataset

从训练集和验证集及它们的tag文件中提取语料和标签，将其封装进mapdataset中。从训练角度来讲：由于目前每个语料长度不固定，因此无法封装成batch并在网络进行训练。从语料的角度来讲，出现在训练集中的语料可能不会在验证集和测试集中存在，无法为其分配对应的标注类别，因此需要对Out-Of-Vocabulary的语料数据进行处理。出现在用于后续convert将原始数据集映射为另一个数据集。

```
def load_dataset(datafiles):
    def read(data_path):
        with open(data_path,encoding="utf-8")as f:
            if data_path.split('/')[-1]!='test.txt':
                with open(f'{data_path.split(".txt")[0]}_TAG.txt',encoding='utf-
8')as t:
                    lines=f.readlines()
                    labels=t.readlines()
                assert len(lines)==len(labels)
                for num in range(len(lines)):
                    line=lines[num].strip().split(' ')
                    label=labels[num].strip().split(' ')
                    yield line, label

    if isinstance(datafiles, str):
        return MapDataset(list(read(datafiles)))
    elif isinstance(datafiles, list) or isinstance(datafiles, tuple):
        return [MapDataset(list(read(datafile))) for datafile in datafiles]
train_ds, dev_ds = load_dataset(datafiles=('/mnt/ve_share/liushuai/lstm/train.txt',
'/mnt/ve_share/liushuai/lstm/dev.txt'))
```

2.2.3 处理未知语料

convert_tokens_to_ids首先读取了tokens的id，将语料和label转化为它们各自的id，针对那些未在训练语料中出现的token，将他们的id换成OOV对应的id。

```
def convert_tokens_to_ids(tokens, vocab, oov_token=None):
    token_ids = []
    oov_id = vocab.get(oov_token) if oov_token else None
    for token in tokens:
        token_id = vocab.get(token, oov_id)
        token_ids.append(token_id)
    return token_ids
```

定义convert函数，将dataset的标签从他们的语料和label映射为对应的id

```
def convert_example(example):
        tokens, labels = example
        token_ids = convert_tokens_to_ids(tokens, word_vocab, 'OOV')
        label_ids = convert_tokens_to_ids(labels, label_vocab, 'O')
        return token_ids, len(token_ids), label_ids

train_ds.map(convert_example)
dev_ds.map(convert_example)
```

```
train_ds.map(convert_example)
dev_ds.map(convert_example)
```

可见，dataset在mapping前后的输入变化：

```
→ train_ds.__getitem__(1)
> (['《', '纽', '约', '时', '报', '》', '报', '道', '说', ...], ['O', 'B_LOC', 'I_LOC', 'O', 'O', 'O', 'O', 'O', 'O', ...])
→ train_ds.map(convert_example)
> <paddlenlp.datasets.dataset.MapDataset object at 0x7fcc89ec0050>
→ train_ds.__getitem__(1)
> ([8, 9, 10, 11, 12, 13, 12, 14, 55, ...], 84, [0, 3, 4, 0, 0, 0, 0, 0, 0, ...])
```

从而将数据处理成便于输入网络的格式。

2.3 构建dataloader

将数据全部转换为小写，并且以每个单词的形式进行切分

```
def data_preprocess(corpus):
    corpus = corpus.strip().lower()
    corpus = corpus.split(" ")
    return corpus
```

2.2.4 构造字典

对于词典中的词，根据其出现次序重新排列，并且构造三个映射，分别为word-id，word-freq，id-word的映射。

```
def build_dict(corpus):
    #首先统计每个不同词的频率（出现的次数），使用一个词典记录
    word_freq_dict = dict()
    for word in corpus:
        if word not in word_freq_dict:
            word_freq_dict[word] = 0
        word_freq_dict[word] += 1
    #将这个词典中的词，按照出现次数排序，出现次数越高，排序越靠前
    word_freq_dict = sorted(word_freq_dict.items(), key = lambda x:x[1], reverse =
True)
    word2id_dict = dict()
    word2id_freq = dict()
    id2word_dict = dict()
    for word, freq in word_freq_dict:
        curr_id = len(word2id_dict)
        word2id_dict[word] = curr_id
        word2id_freq[curr_id] = freq #以id为标识，把标识加到freq中
        id2word_dict[curr_id] = word #
    return word2id_freq, word2id_dict, id2word_dict
word2id_freq, word2id_dict, id2word_dict = build_dict(corpus)
```

2.2.5 转换为id序列

将每个位置的词用id来表示，作为这个词的token，便于后续送入网络训练。

```
def convert_corpus_to_id(corpus, word2id_dict):
  corpus = [word2id_dict[word] for word in corpus] #一个text中每个位置的词用id来表示
  return corpus
corpus = convert_corpus_to_id(corpus, word2id_dict)
```

2.3 数据集

2.3.1 构建数据集

利用batchify_fn函数，对训练语料进行填充，DataLoader调用collate_fn时会自动选择一个minibatch中长度最长的序列作为这个batch的截断长度，此时，我们将短于该截断长度的数据进行填充，可以保证一个minibatch的训练数据shape的统一。

```
batchify_fn = lambda samples, fn=Tuple(
        Pad(axis=0, pad_val=word_vocab.get('OOV')),  # token_ids
        Stack(),  # seq_len
        Pad(axis=0, pad_val=label_vocab.get('O'))  # label_ids
    ): fn(samples)

train_loader = paddle.io.DataLoader(
        dataset=train_ds,
```

```
        batch_size=32,
        shuffle=True,
        drop_last=True,
        return_list=True,
        collate_fn=batchify_fn)

dev_loader = paddle.io.DataLoader(
        dataset=dev_ds,
        batch_size=32,
        drop_last=True,
        return_list=True,
        collate_fn=batchify_fn)
```

2.4 构建模型

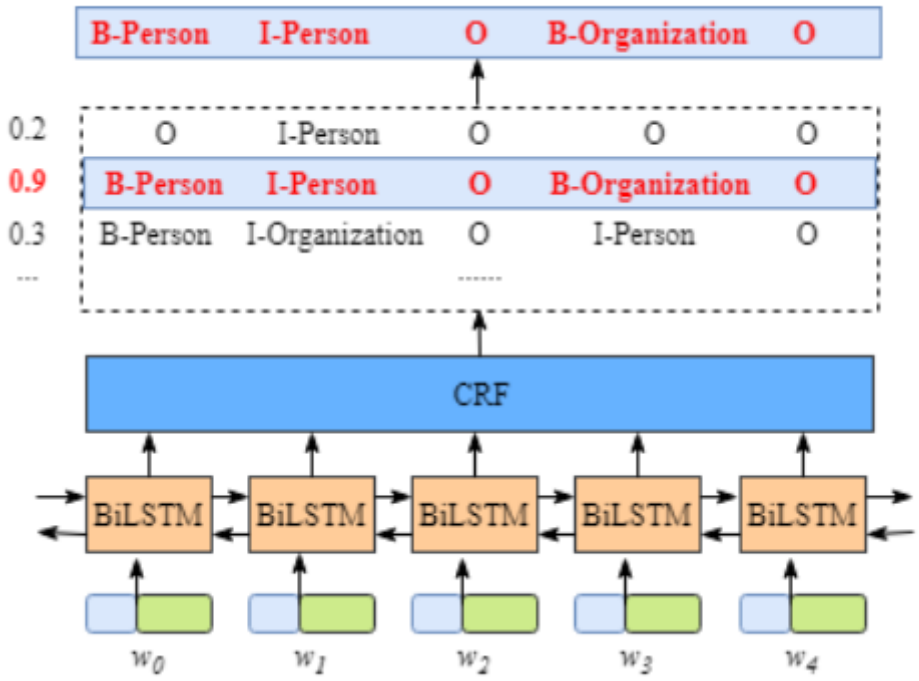
```
class BiLSTMWithCRF(nn.Layer):
    def __init__(
        self,
        embed_dim, #300
        hidden_size, #300
        word_num, #len
        label_num, #num_classes
        num_layers=1,
        dropout_prob=0.0,
        init_scale=0.1,#权重初始化
    ):
        super(BiLSTMWithCRF, self).__init__()
        # 定义词向量层，将输入的词语索引映射为词向量
        self.embedder = nn.Embedding(word_num, embed_dim)
        # 定义 LSTM 层，用于处理输入序列，并提取特征
        self.lstm = nn.LSTM(embed_dim, hidden_size, num_layers, "bidirectional",
dropout=dropout_prob)
        # 全连接层输入维度，即 LSTM 输出的维度乘以 2（因为是双向 LSTM）
        self.fc = nn.Linear(
            hidden_size * 2,
            hidden_size,
            weight_attr=paddle.ParamAttr(initializer=I.Uniform(low=-init_scale,
high=init_scale)),
        )
        self.output_layer = nn.Linear(
            hidden_size,
            label_num+2,
            weight_attr=paddle.ParamAttr(initializer=I.Uniform(low=-init_scale,
high=init_scale)),
        )
        # 定义 CRF 模型，用于对序列标注结果进行建模和预测
        self.crf = LinearChainCrf(label_num,with_start_stop_tag=True)
         # 定义解码器，用于利用 CRF 模型的转移矩阵进行标注结果的预测
        self.decoder = ViterbiDecoder(self.crf.transitions)

    def forward(self, x_1, seq_len_1):
```

```
        # 输入 x_1 是一个形状为 [batch_size, seq_len] 的整数张量，表示输入的词语索引序列
        # seq_len_1 是一个形状为 [batch_size] 的整数张量，表示每个输入序列的真实长度
        x_embed_1 = self.embedder(x_1)
        # 下面的代码将输入的词语索引映射为词向量，并输入到 LSTM 层中进行处理
        lstm_out_1, (_, _) = self.lstm(x_embed_1, sequence_length=seq_len_1)
        out = paddle.tanh(self.fc(lstm_out_1))
        logits = self.output_layer(out)
         # 利用 CRF 模型和解码器对标注结果进行预测，并返回预测结果、输入序列长度和标签概率分布
        _, pred = self.decoder(logits, seq_len_1)
        return logits, seq_len_1, pred
```

利用paddleNLP中的Embedding，LSTM，LinearChainCrf，ViterbiDecoder类进行网络构建，定义embedding的输入为单词长度，LSTM的层数为1，且为bidirectional，定义了lstm为双向，使得模型同时考虑前向和后向的上下文信息，从而更好地处理序列中的长期依赖关系。设计fc-activation-fc的结果，将结果映射到类别，由于是双向lstm，因此第一个fc的hidden_size长度要乘2。此处linear的最终输出维度为label_num+2，引入了两特殊标记两个特殊标记 "BOS" 和 "EOS"作为起始和结束位置，从而帮助模型更好的预测序列的边界和结构。

在超参设置方面，我们定义embed_dim的默认值为300，表示了将语料数据投影到embedding空间的大小，hidden_size也设为300，表示了LSTM 模型维护历史信息的能力。

forward中展示了Bi-lstm-crf的两个stage，第一个阶段将bilstm的输出送入到全连接层，给出一个预测结果，而后经过线性CRF实现序列依赖，并利用viterbidecoder进行解码。



其中，LinearChainCrf引入的目的是为了减少无效输出，例如以I中间字作为开头的无效序列。LinearChaincrf通过构造转移矩阵的方法学习两个命名实体间的关系，有

$$F = logZ(x) = log\Sigma exp(score(x, y))$$
$$score(x, y) = \Sigma Emit(x_i, y_i) + Trans(y_{i-1}, y_i)$$
$$p(y_i) = Emit(x_i, y_i), T(y_{i-1}, y_i) = Trans(y_{i-1}, y_i)$$

而后利用viterbidecoder对得分最高的序列进行解码。

2.5 模型训练

```
network = BiLSTMWithCRF(300, 300, len(word_vocab), len(label_vocab))
model = paddle.Model(network)

optimizer = paddle.optimizer.Adam(learning_rate=0.001,
parameters=model.parameters())
crf_loss = LinearChainCrfLoss(network.crf)
chunk_evaluator = ChunkEvaluator(label_list=label_vocab.keys(), suffix=True)

visualcallback =
paddle.callbacks.VisualDL(log_dir='/mnt/ve_share/liushuai/lstm/n2e300')
lrcallback = paddle.callbacks.LRScheduler(by_step=True, by_epoch=False)
stopcallback = paddle.callbacks.EarlyStopping(
    'loss',
    mode='min',
    patience=6,
    verbose=1,
    min_delta=0,
    baseline=None,
    save_best_model=True)
callbacks=[visualcallback,lrcallback,stopcallback]
model.prepare(optimizer, crf_loss, chunk_evaluator)
model.fit(train_data=train_loader,
              eval_data=dev_loader,
              epochs=3,
              save_dir='/mnt/ve_share/liushuai/lstm/n2e300',
              log_freq=1,
              callbacks=callbacks)
```

- LinearChainCrfLoss将计算线性CRF的负对数似然，令

$$Z(x) = sum_{y'} exp(score * x, y')$$

有 $loss = -logp(y|x) = -log(exp(score(x,y))/Z(x)) = -score(x,y) + logZ(x)$

通过查看文档，我们发现有三个较为适合的回调函数可以用于训练优化。

- **chunk_evaluator**

  构建chunkevaluator对训练的loss,accuracy,f1,recall等指标进行评估，model.fit会自动将指标的回显显示在训练流程中。

- **visualcallback**

  visualcallback提供了visualdl的输入文件，在训练时会将chunk_evaluator评估的结果

- **lrcallback**

  lrcallback的回调函数提供了LRScheduler，用于调整训练过程的学习率。具体来讲，通过设置by_step=True可以实现逐step的学习率调整

- **stopcallback**

  stopcallback引入了early stop机制。在这里通过检测loss的指标，设定patience为6，表示如果梯度在近六组中未出现下降的情况，我们则认为此时的训练已经收敛，则对模型进行保存，训练停止。

2.6 结果推理

由于我们在前面构建的label_vocab为label->id的字典，此时我们需要对label进行解码，因此首先需要构建id2label的逆向字典。

```
id2label=dict(enumerate(label_vocab))
```

而后我们构建并加载test_loader

```
def load_dataset(datafiles):
    def read(data_path):
        with open(data_path,encoding="utf-8")as f:
                    lines=f.readlines()
                    for num in range(len(lines)):
                        line=lines[num].strip().split(' ')
                        yield line,[]

    if isinstance(datafiles, str):
        return MapDataset(list(read(datafiles)))
    elif isinstance(datafiles, list) or isinstance(datafiles, tuple):
        return [MapDataset(list(read(datafile))) for datafile in datafiles]
test_ds = load_dataset(datafiles=('/mnt/ve_share/liushuai/lstm/test.txt'))
```

```
test_loader = paddle.io.DataLoader(
        dataset=test_ds,
        batch_size=32,
        drop_last=False,
        return_list=True,
        collate_fn=batchify_fn,
        shuffle=False)
```

将结果写入到txt文件中

```
import tqdm
for num in tqdm.tqdm(range(length)):
    for num2 in range(len(list(lens[num]))):
        index=lens[num][num2]
        templist=list(pred[num][num2][:index])
        with open("/mnt/ve_share/liushuai/lstm/2020212267.txt",'a+')as f:
            for ele in templist:
                    f.write(id2label[ele].replace("'","")+" ")
                f.write('\n')
```

3 结果分析及消融实验

由于时间原因，作者针对以下四种设置构建了消融实验

embedding_size	lstm_layer

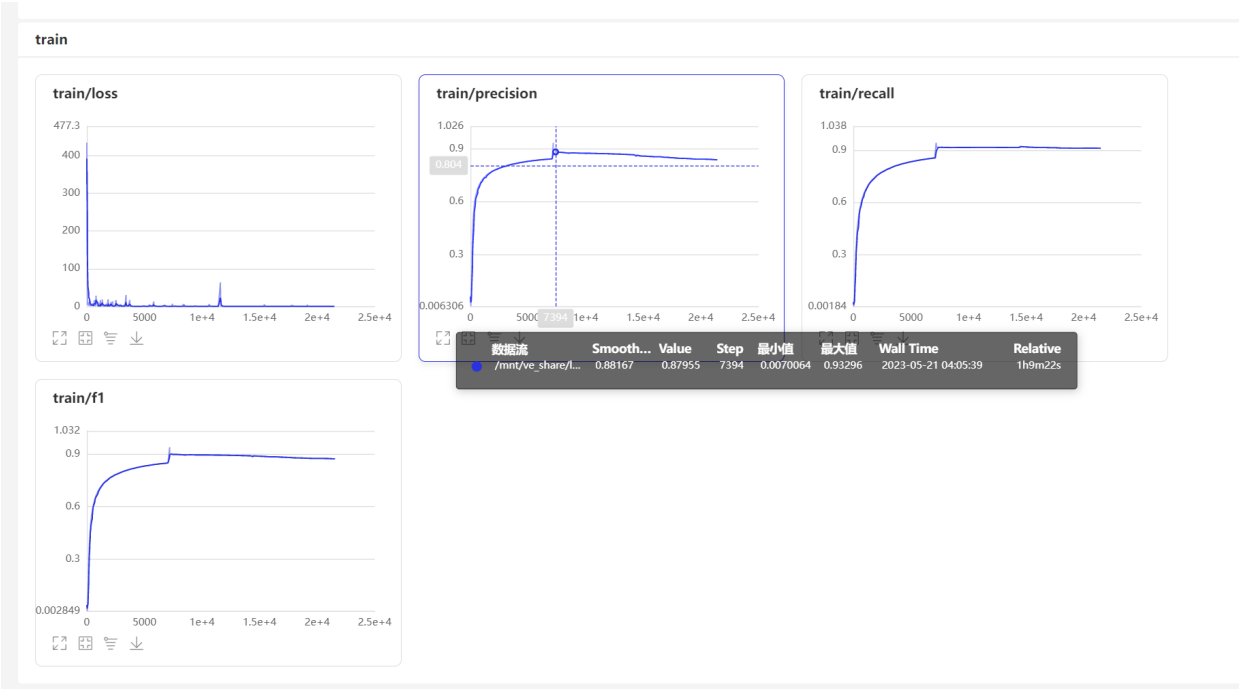
embedding_size	lstm_layer
300	1
600	1
300	2
600	2

后文将用E表示embedding大小，N表示lstm的层数

3.1.1 E300N1结果

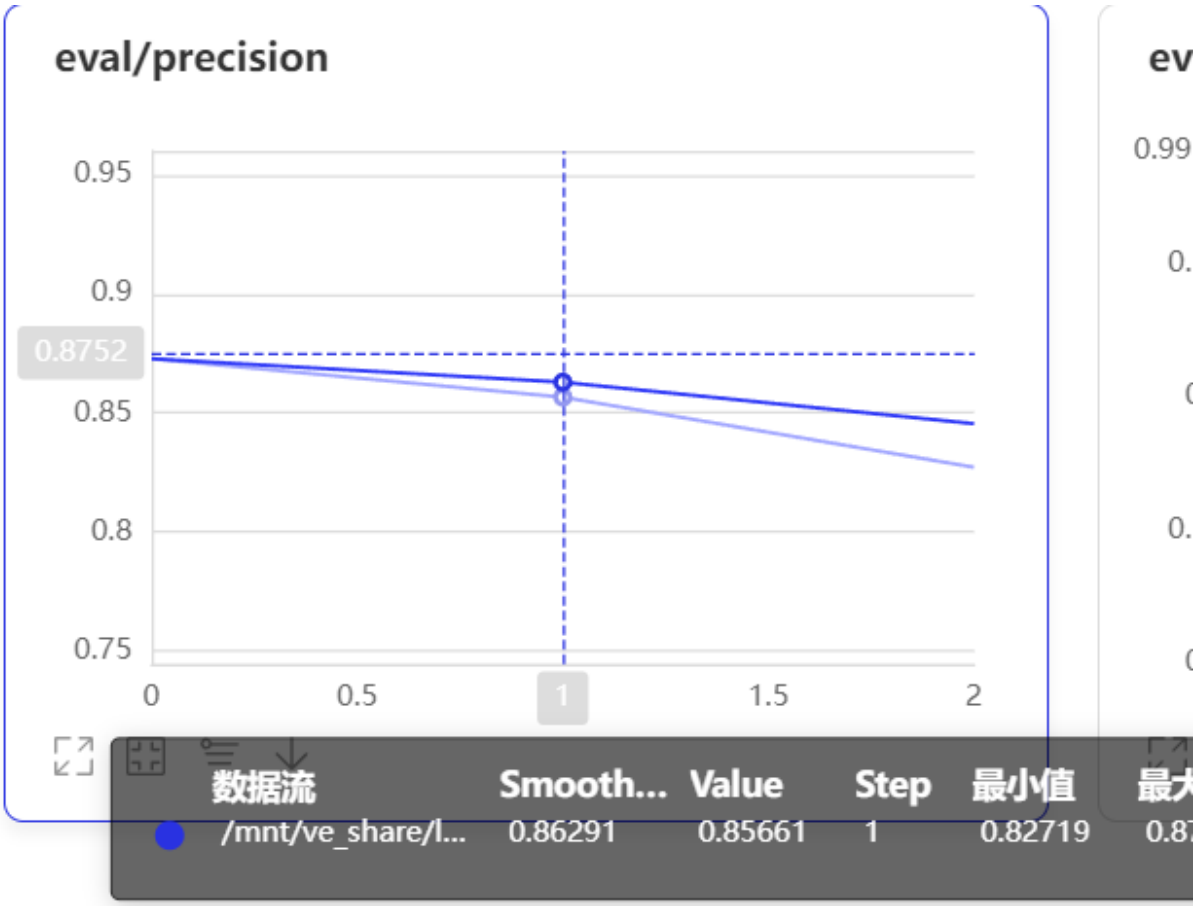
其中，E300N1的可视化结果为：

loss在一个epoch结束后基本已经趋于稳定。



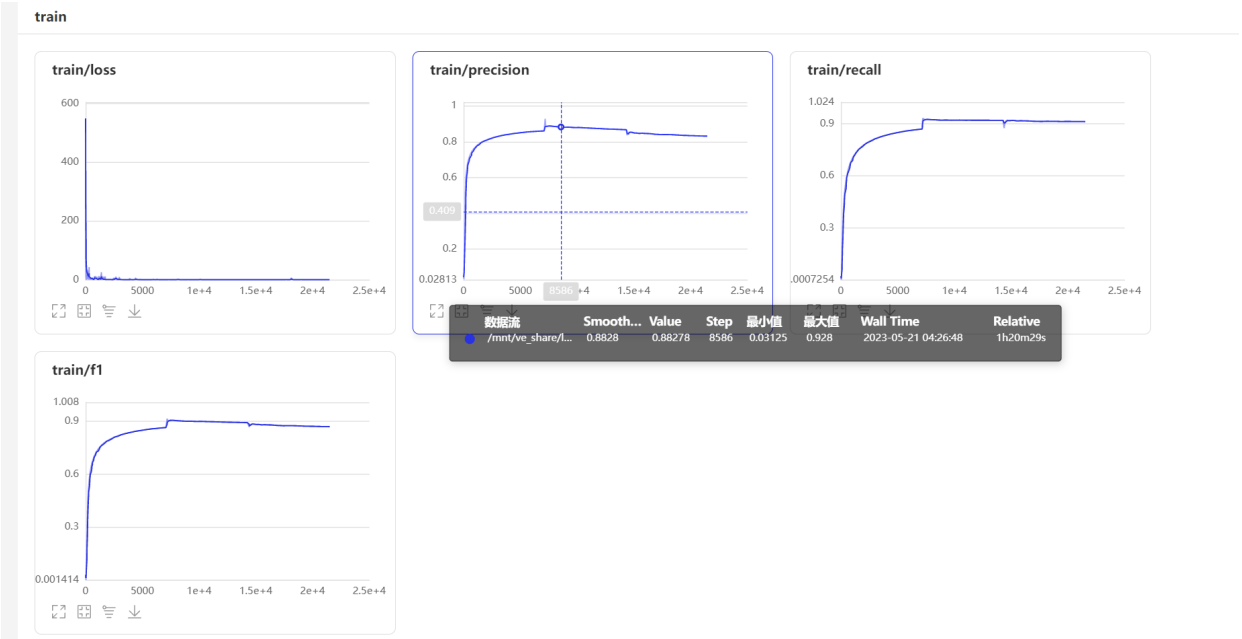
precision,recall,f1的指标在step=7000左右时发生了小范围的增长，可能因为此时第一个epoch训练结束，学习率和训练策略有了部分更改，同时引入了验证集进行训练，增强了模型的泛化能力。在第二个epoch接近结束时后三者的指标均有一定程度的下降，推测可能因为由于引入了验证集，导致在测试集上的过拟合效果有所缓解。在第三个epoch中loss不再下降，可能出现过拟合情况。

在验证集上，模型的准确率为0.87，且第三个epoch的效果差于第二个，说明模型训练出现了过拟合。



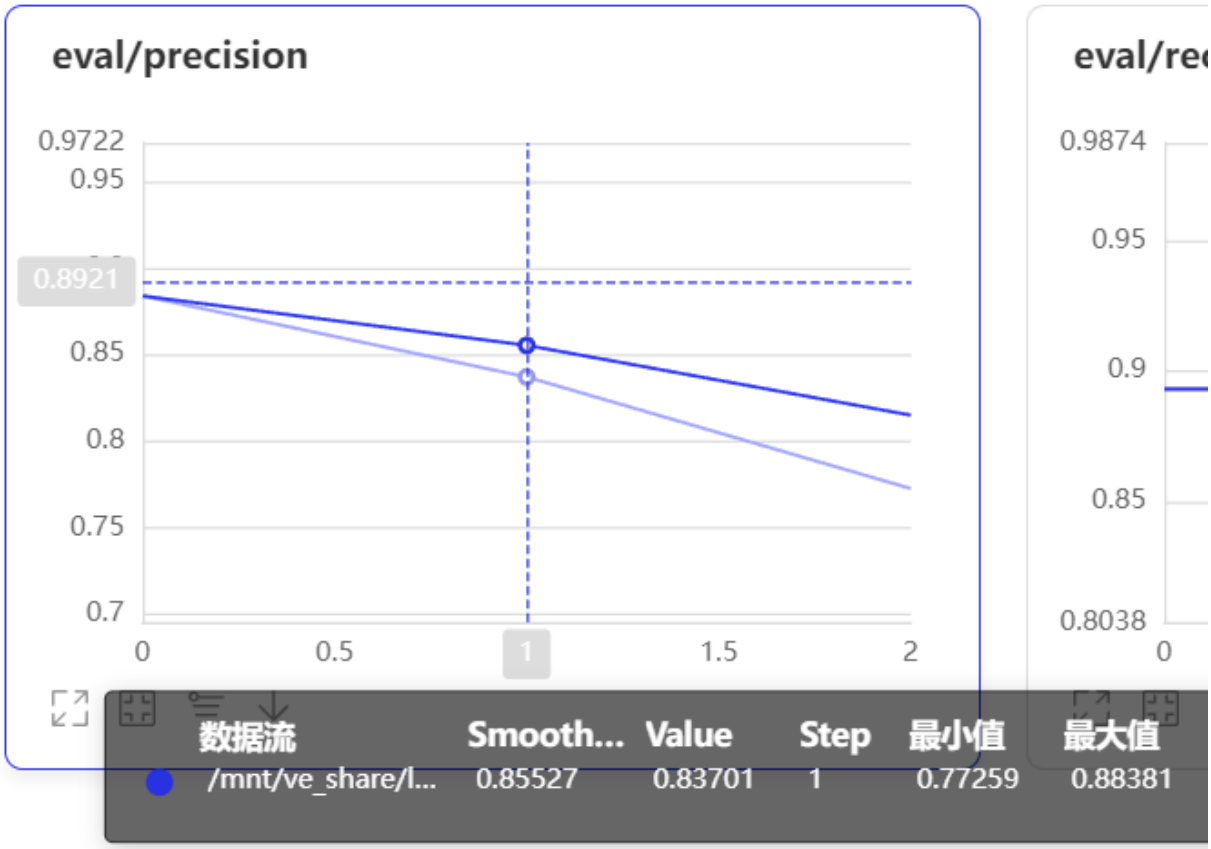
3.1.2 E600N1结果

在lstm层数不变的情况下改变embedding_size的训练结果如下



可以看出loss的变化更加平稳，后期几乎很少出现波动，同时准确率,f1,recall同样出现了类似的“先突增后缓慢下降”的现象，但在训练完一个epoch，即引入dev集之前，会发现train的准确率比e300更高些，说明在更高维的embedding空间，模型对于复杂数据的表征能力确实会变强。

在验证集上，模型的准确率从0.88降至0.77，说明出现了更为严重的过拟合。

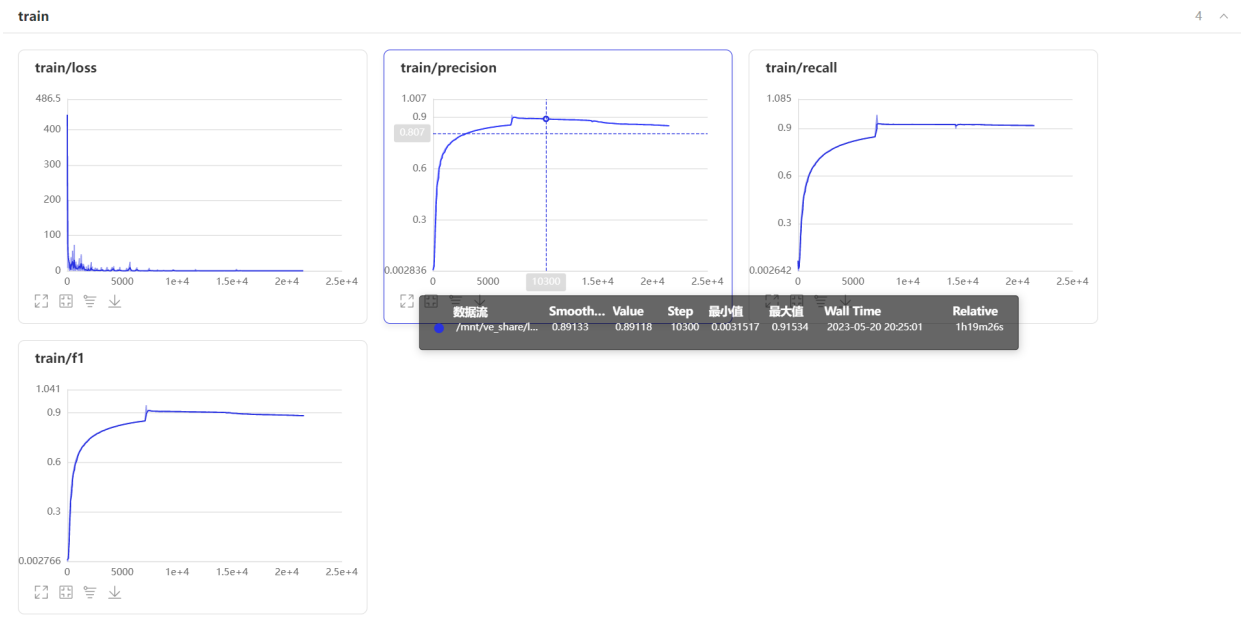


3.1.3 E300N2结果

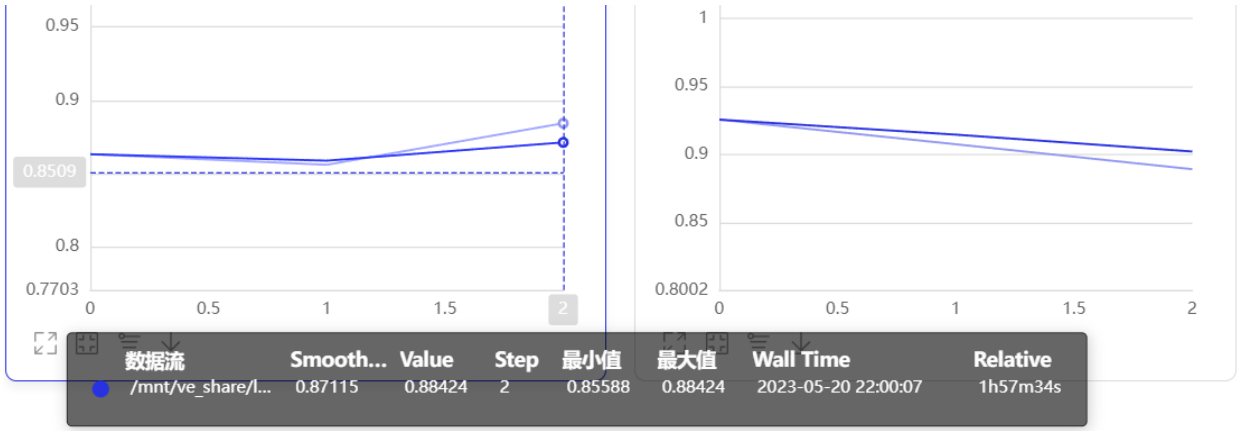
LSTM 模型可以由多个 LSTM 层组成，每个层都有一组隐藏单元和一组门控单元，用于捕捉序列中的长期依赖关系。每个 LSTM 层的输出都会成为下一层的输入，从而使得模型可以逐步地学习更复杂的序列模式。

增加 LSTM 层数可以增加模型的深度和能力，从而使其更好地捕捉数据中的复杂模式和长期依赖关系。

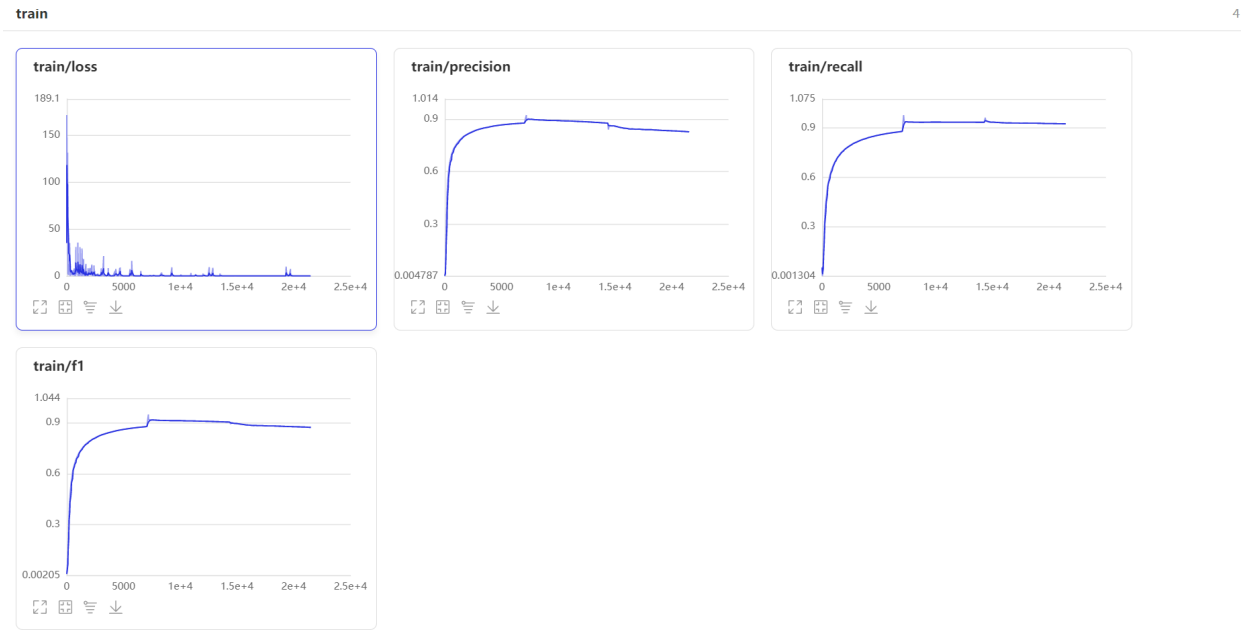
当我们把lstm的层数改为2后，会发现在更晚的地方(即step更大的地方)出现了模型准确率的提升，推测此时模型同样具有了更好的表征能力，模型具有更强的鲁棒性。



在验证集上，模型的准确率为0.88，且未出现1，2中的验证集准确率下降的问题，说明模型的鲁棒性有显著增强。

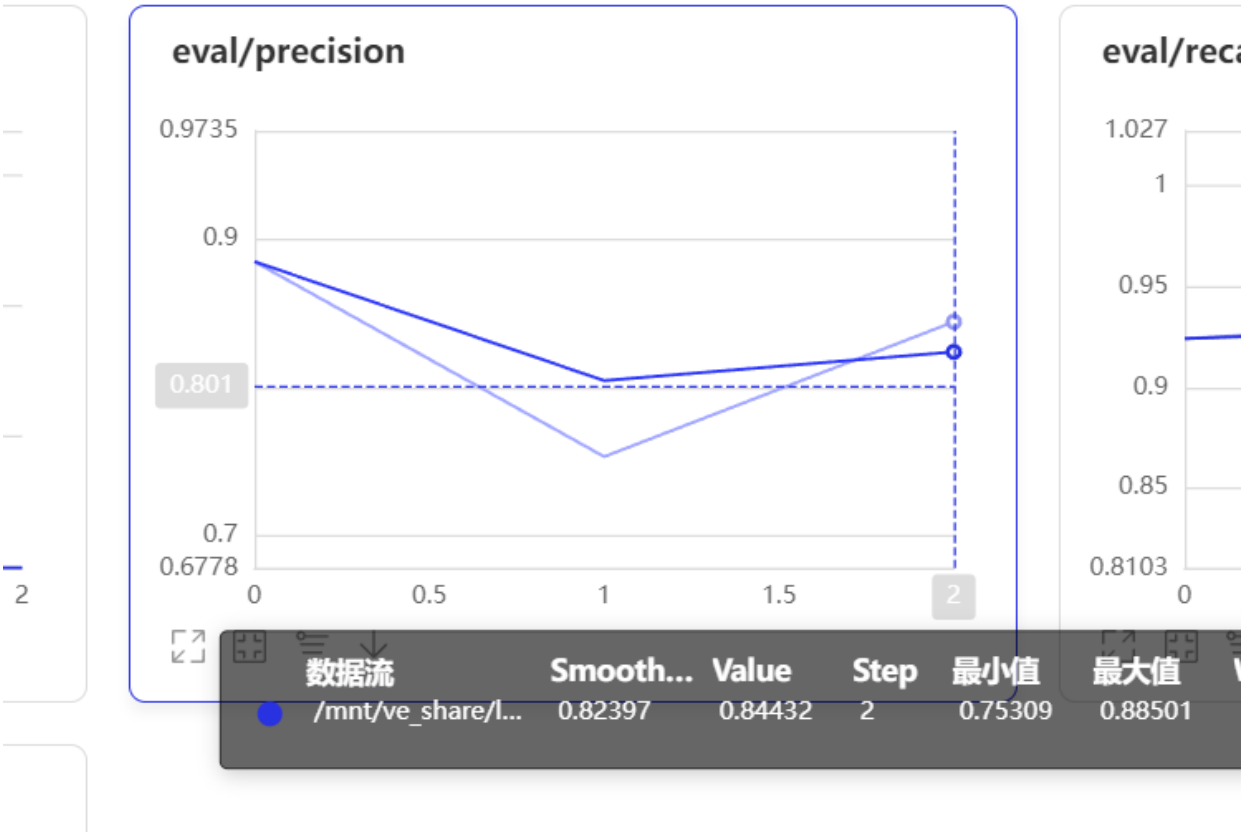


3.1.4 E600N2的结果



模型在稍微提早的位置出现了"阶跃",训练后期在训练集上的指标变差。

在验证集上，模型的准确率为0.82，测试集准确率先下降再上升，说明此时模型还没有完全拟合，由于E600N2模型的参数量更为庞大，因此需要更多轮训练。



4.实验总结

本次实验让我对paddle的训练流程及paddle的高阶训练框架更为熟悉，同时学习了Bilstm-crf算法的内在逻辑和模型训练的优化流程。在完成消融实验的过程中增强了作者对于实验结果的分析和推理能力。