

Requirements 1

实验中使用了4个模型来进行文本分类任务，分别是：RNN，双向LSTM，FastText，CNN。

RNN:

RNN的实现代码:

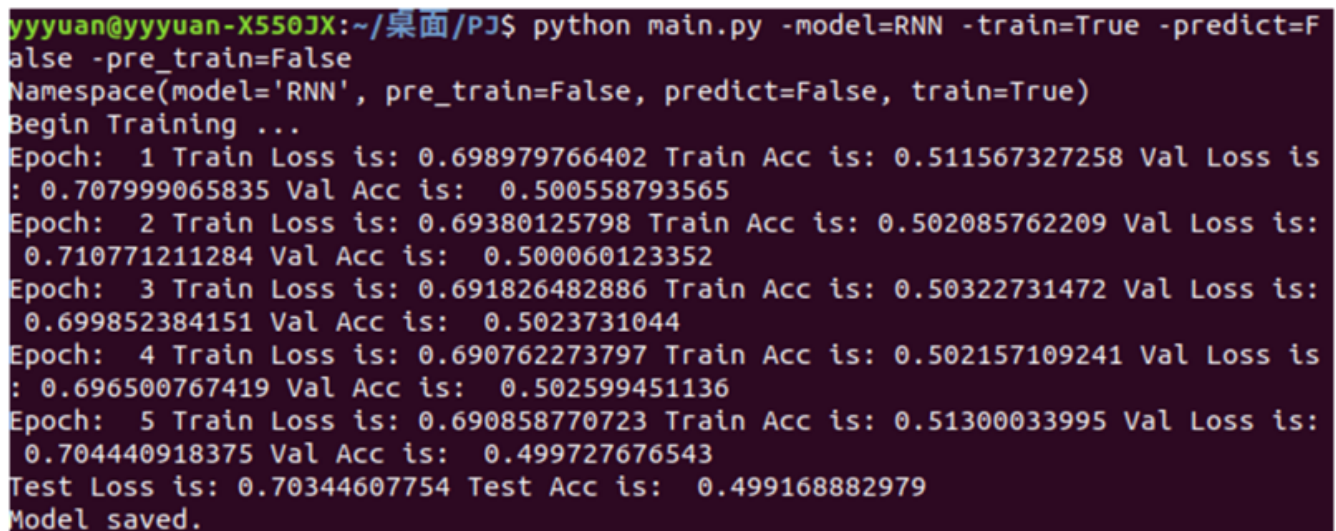
```
class RNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):
        super(RNN, self).__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        embedded = self.embedding(x)
        output, hidden = self.rnn(embedded)
        assert torch.equal(output[-1, :, :], hidden.squeeze(0))

        return self.fc(hidden.squeeze(0))
```

对输入的数据流进行embedding之后，通过RNN进行分类之后之后，再经过一个全连接层将结果映射到最终的分类结果的维度上。

实验结果:



```
yyyuan@yyyuan-X550JX:~/桌面/PJ$ python main.py -model=RNN -train=True -predict=F
alse -pre_train=False
Namespace(model='RNN', pre_train=False, predict=False, train=True)
Begin Training ...
Epoch: 1 Train Loss is: 0.698979766402 Train Acc is: 0.511567327258 Val Loss is
: 0.707999065835 Val Acc is: 0.500558793565
Epoch: 2 Train Loss is: 0.69380125798 Train Acc is: 0.502085762209 Val Loss is:
0.710771211284 Val Acc is: 0.500060123352
Epoch: 3 Train Loss is: 0.691826482886 Train Acc is: 0.50322731472 Val Loss is:
0.699852384151 Val Acc is: 0.5023731044
Epoch: 4 Train Loss is: 0.690762273797 Train Acc is: 0.502157109241 Val Loss is
: 0.696500767419 Val Acc is: 0.502599451136
Epoch: 5 Train Loss is: 0.690858770723 Train Acc is: 0.51300033995 Val Loss is:
0.704440918375 Val Acc is: 0.499727676543
Test Loss is: 0.70344607754 Test Acc is: 0.499168882979
Model saved.
```

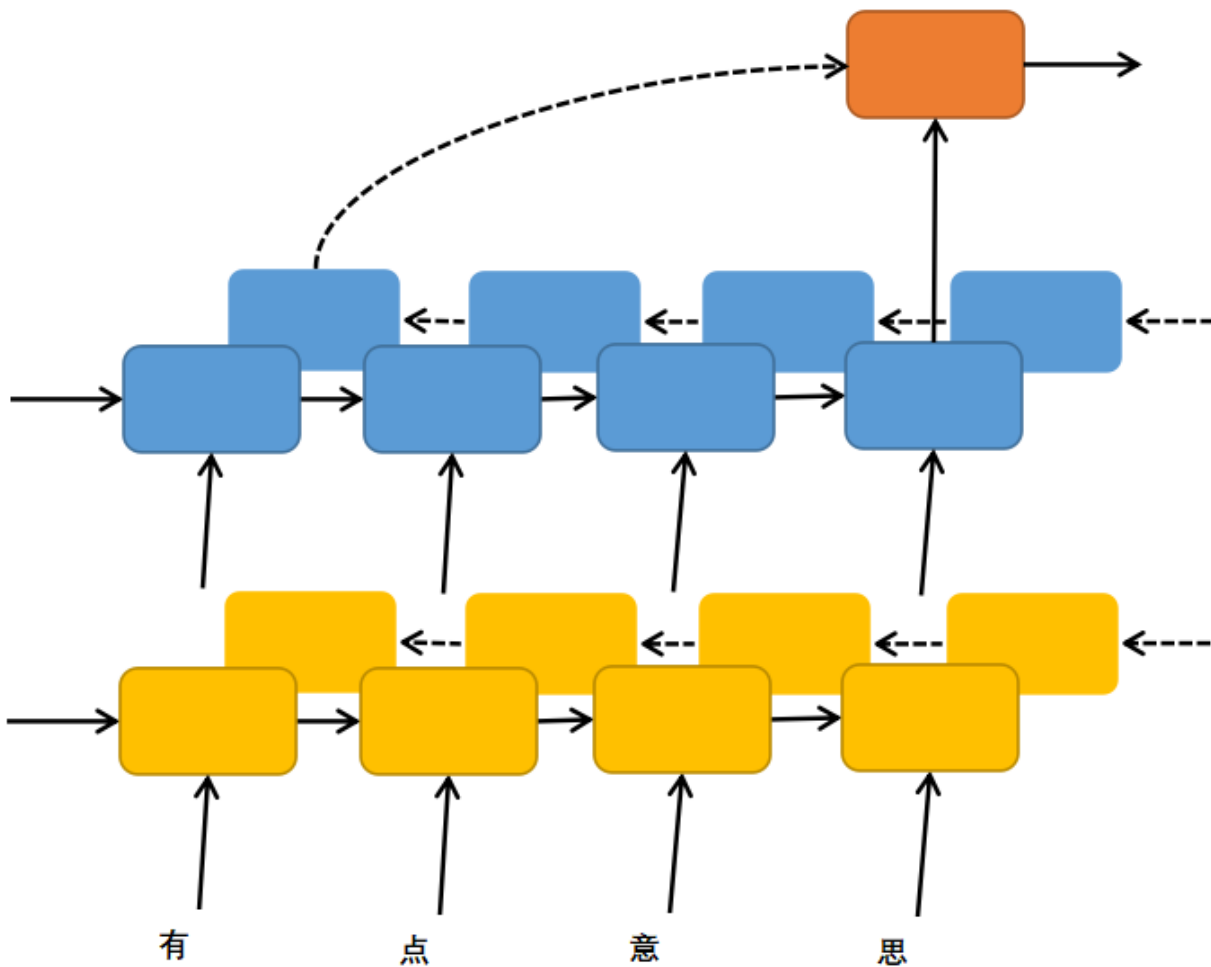
可以看到，实际上使用该模型分类结果并不是很好，原因有两点：

- 模型过于简单
- embedding的初始值是随机化的，后续的的训练中由于样本不足，导致模型过拟合。

可以通过增加模型复杂度和**加载预训练词向量模型**来解决上述问题。

双向LSTM:

结构图:



多层双向 LSTM 示意图 (黄色和蓝色为隐藏层, 橙色为输出层)

双向LSTM的实现代码:

```
class LSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout):
        super(LSTM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=n_layers,
                             bidirectional=bidirectional, dropout=dropout)
        self.fc = nn.Linear(hidden_dim*2, output_dim)
        self.dropout = nn.Dropout(dropout)

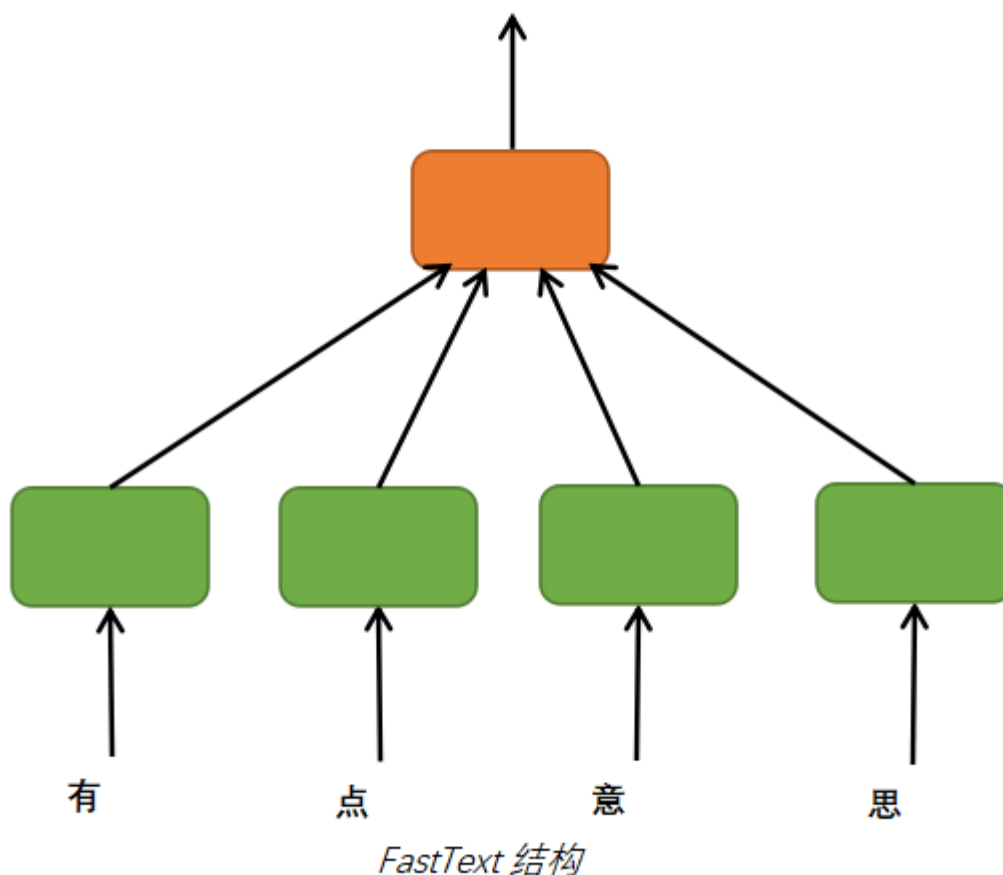
    def forward(self, x):
        embedded = self.dropout(self.embedding(x))
        output, (hidden, cell) = self.lstm(embedded)
        hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1))
        return self.fc(hidden.squeeze(0))
```

相比较于前一个RNN，LSTM通过增加gate解决了RNN对输入序列无法实现长期依赖的问题。此外，为了防止由于模型参数增加而产生的过拟合问题，增加了drop out，随机将网络中间层一些参数设为0，从而避免过拟合。

通过实验结果可以发现，模型的准确率明显提升，但是还存在一个很大的问题，模型的**训练时间明显增加**。

FastText:

结构图:



FastText的实现代码:

```
class FastText(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim):
        super(FastText, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.fc = nn.Linear(embedding_dim, output_dim)

    def forward(self, x):
        embedded = self.embedding(x)
        embedded = embedded.permute(1, 0, 2)
        pooled = F.avg_pool2d(embedded, (embedded.shape[1], 1)).squeeze(1)
        return self.fc(pooled)
```

FastText在可以**保证模型准确率的前提下，大幅度提升模型的训练时间**。关键点在于，FastText中将句子分为n-grams之后，将其添加在原句子的末尾作为新的输入。（简单起见，实验中使用2-grams）

实验结果：

```
yyyuan@yyyuan-X550JX:~/桌面/PJ$ python main.py -model=FastText -train=True -predict=False -pre_train=True
Namespace(model='FastText', pre_train=True, predict=False, train=True)
Loading pre-trained model ...
Finish loading.
Begin Training ...
Epoch: 1 Train Loss is: 0.678488118463 Train Acc is: 0.61993587888 Val Loss is: 0.648303730057 Val Acc is: 0.793522238731
Epoch: 2 Train Loss is: 0.576376693298 Train Acc is: 0.813337948224 Val Loss is: 0.518469081914 Val Acc is: 0.799538111433
Epoch: 3 Train Loss is: 0.438098236034 Train Acc is: 0.848381485025 Val Loss is: 0.430565342624 Val Acc is: 0.820450430221
Epoch: 4 Train Loss is: 0.356257302576 Train Acc is: 0.875156356319 Val Loss is: 0.39118170009 Val Acc is: 0.835230166608
Epoch: 5 Train Loss is: 0.300225675174 Train Acc is: 0.895969955344 Val Loss is: 0.368614913618 Val Acc is: 0.845051494051
Test Loss is: 0.36380973807 Test Acc is: 0.845751908231
Model saved.
```

loss and accuracy curves in training procedure



FastText模型分为3部分：

- 对每个词作word embedding
- 将每个词的word embedding求平均值
- 将求得的平均值经过全连接层之后作为输出

CNN：

CNN的实现代码：

```
class CNN(nn.Module):
```

```

def __init__(self, vocab_size, embedding_dim, n_filters, filter_sizes, output_dim,
dropout):
    super(CNN, self).__init__()

    self.embedding = nn.Embedding(vocab_size, embedding_dim)
    self.convs = nn.ModuleList([nn.Conv2d(in_channels=1, out_channels=n_filters,
kernel_size=(fs, embedding_dim)) for fs in filter_sizes])
    self.fc = nn.Linear(len(filter_sizes)*n_filters, output_dim)
    self.dropout = nn.Dropout(dropout)

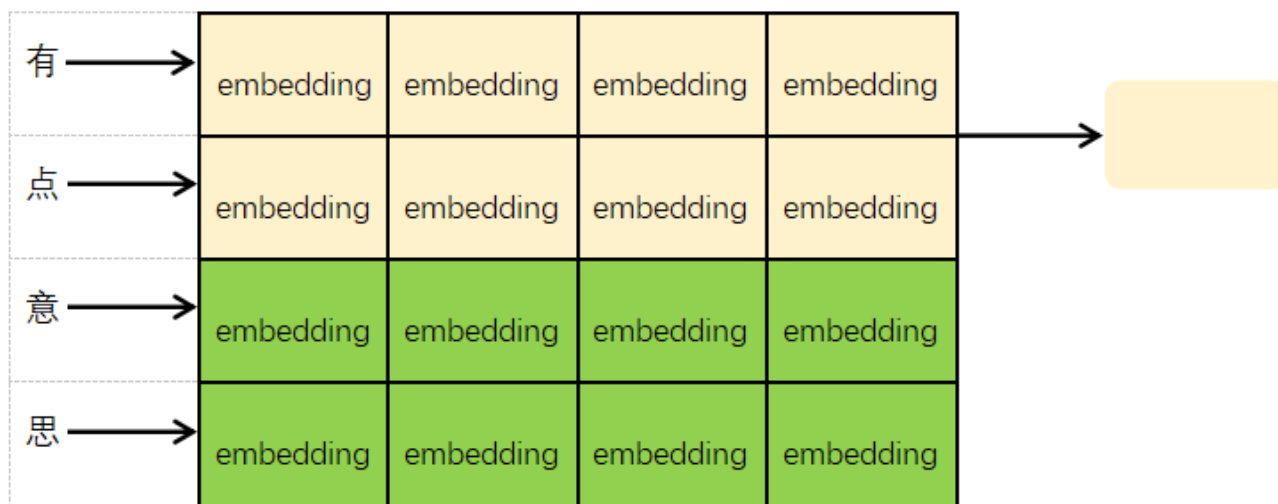
def forward(self, x):
    x = x.permute(1, 0)
    embedded = self.embedding(x)
    embedded = embedded.unsqueeze(1)
    conved = [F.relu(conv(embedded)).squeeze(3) for conv in self.convs]
    pooled = [F.max_pool1d(conv, conv.shape[2]).squeeze(2) for conv in conved]
    cat = self.dropout(torch.cat(pooled, dim=1))
    return self.fc(cat)

```

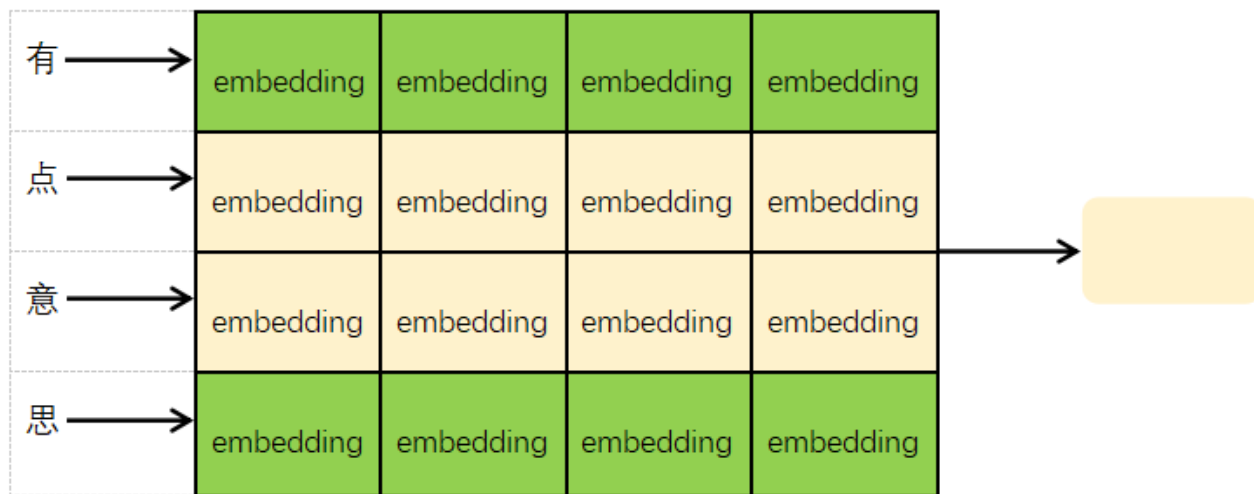
相比较于RNN类（LSTM，GRU等）的模型，CNN具有很好的特征提取能力。同时，由于不需要按序列处理数据，所以相对来说模型的训练速度更快一些。

CNN可以通过不同尺寸的filter，来处理不同的n-grams，从而达到文本特征提取的目的。

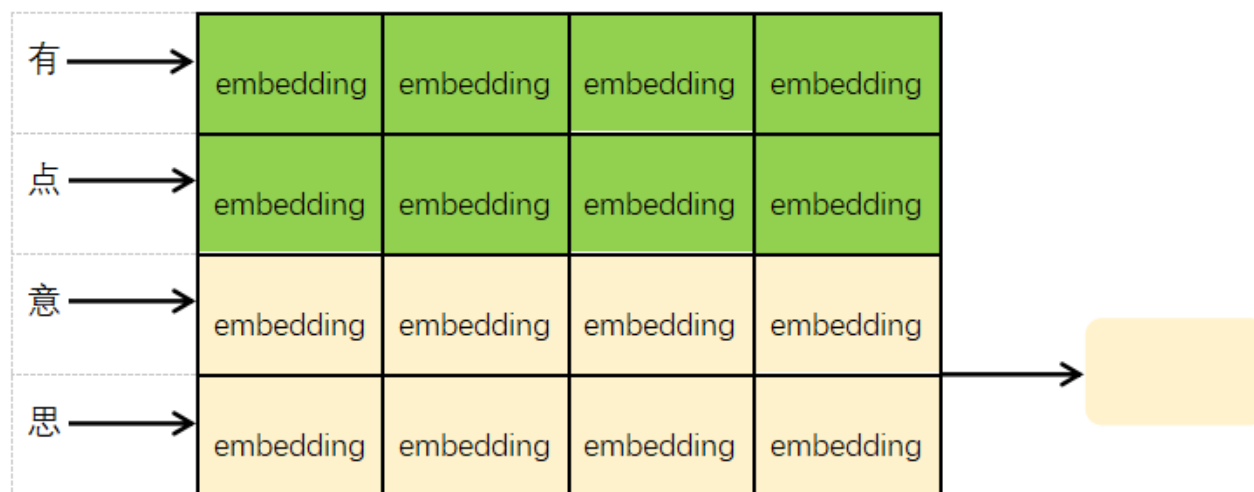
CNN对filter的处理过程如下：



filter 处理2-grams (浅色为 filter)



filter 处理2-grams (浅色为 filter)



filter 处理2-grams (浅色为 filter)

新数据集：

除了assignment中提供的数据之外，还额外使用了爬取的中文影评作为数据，采用上述模型，实现了**中文的文本情感分类**（二分类任务，判断是否为正面情绪/负面情绪）。

中文影评的数据中，训练集共有30000影评，测试集共有6000条影评。通过对得到的原始数据使用jieba进行分词之后，过滤掉停用词，再将其作为输入数据。

新数据集上训练完成后的实测结果（越接近1表示为正面情绪的概率越大）：

Please input a sentence:放假真好
0.99973231554

Please input a sentence:实验报告快要写不完了
1.06929292087e-05

```
Please input a sentence:完了，完蛋了，真的要完蛋了
0.10714238137
```

```
Please input a sentence:凉了，凉了，这下要凉了，凉透了
0.910346806049
```

Requirements 2

本次的实验之中，使用的较多的是fastNLP的数据预处理模块，在实现中文文本分类的时候，使用了torchtext中的数据预处理模块，所以主要对这两者进行比较。

fastNLP和torchtext的共同点在于，都是通过field来对加载的原始数据进行处理。fastNLP的优势在于，field_name为字典中的key，支持对field_name的增加、删除、修改和重命名等操作，使用起来更加**直观和便捷**。此外，fastNLP中独立封装的Vocabulary十分好用，而torchtext中vocabulary和field是混在一起的，使用的时候会显得比较混乱，所以还是fastNLP的设计更好一些。美中不足之处在于，对于json文件的处理，fastNLP的处理稍逊于torchtext，大部分情况下需要通过手动处理json文件后，再使用fastNLP的模块进行数据预处理。不过，torchtext虽然方便，但却对json文件中的数据格式要求比较严格，总的来说，各有所长。

fastNLP和torchtext都支持自定义数据预处理的函数，这点极大的减少了数据预处理的工作，且使得代码更加整洁，精简。区别在于，torchtext支持的函数更加“泛化”，形如：

```
def generate_bigrams(x):
    n_grams = set(zip(*[x[i:] for i in range(2)]))
    for n_gram in n_grams:
        x.append(' '.join(n_gram))
    return x
```

即自定义的函数还可以用来处理别的数据，而fastNLP由于Instance中field_name为字典的key，所以预处理的函数比较特殊，默认传入的参数为一个字典，形如：

```
def remove_empty(x):
    while '' in x['text']:
        x['text'].remove('')
    return x['text']
```

这点可以稍微改进一下。

另外，fastNLP中封装的trainer虽然很大程度上提高了便捷性，但是灵活性不是很高（如：模型forward的返回值必须为字典、train过程中查看中间变量不是很方便），所以此次实验中train的过程都没有使用trainer。感觉这是模块化的固有问题，提高便捷的同时自然会降低灵活性，主要看使用者如何权衡了。

在整个使用fastNLP的过程中，感受最深的其实还是文档的问题（个人感觉文档能做成现在这样子真的已经很不错了！），可以适当的增加一些用例，因为有时候仅仅看函数参数的说明，对函数功能的理解并不是很透彻，需要反复尝试或者看源码才能明白到底应该怎么使用（刚才又特意看了一下文档，发现突然变好用了，可能之前是因为刚刚接触新的框架，所以不太习惯吧.....）。

在使用过程中，发现fastNLP对**数据格式的兼容性**似乎不是很好，经常会出现由于数据格式而导致的问题，这些基本上最后都是通过看源码才解决掉的。这点可以后续改进一下。

总而言之，fastNLP确实是一个非常好用的工具，其中对于很多操作的模块化处理，使得最终的代码写起来更加简洁与便捷，极大的提高了编程的效率。同时，详尽的文档在帮助新手学习使用fastNLP时也起了很大的作用。十分感谢参与到这个开源项目中的老师和同学！

补充

由于json格式的文件无法提交，故补充config.json文件中的参数

```
{
  "data": {
    "dataset": "en",
    "dataset_path": "data",
    "pretrain_path": "'sgns.literature.bigram'",
    "batch_size": 32
  },
  "model": {
    "arch": "RNN",
    "optimizer": "Adam",
    "learning_rate": 1e-2,
    "weight_decay": 1e-5,
    "path": "ckpt/",
    "params": {
      "embedding_dim": 300,
      "hidden_dim": 256,
      "n_filters": 100,
      "filter_size": [3, 4, 5],
      "output_dim": 4,
      "n_layers": 2,
      "dropout": 0.5,
      "bidirectional": true
    }
  },
  "n_epoch": 20,
  "train": true,
  "continue": false,
  "predict": false,
  "pre_train": false,
  "name": "test"
}
```