Assignment 4

Assignment 4 includes the implementations of RNN and CNN with the help of fastNLP and PyTorch. Therefore, this report can be roughly divided into two parts: the first one will focus on the implementations of these two models, and the second part will focus on the topics of fastNLP.

Part 1: Models and Training

This section will include the implementation of basic RNN and CNN by PyTorch, and discuss about the features of these different networks.

The Data

The data is the same as what I have used in Assignment 2: *20 news group* dataset in sklearn, the sentences with more than 500 words are dropped. And a dictionary with about 4000 words is built according to their frequency.

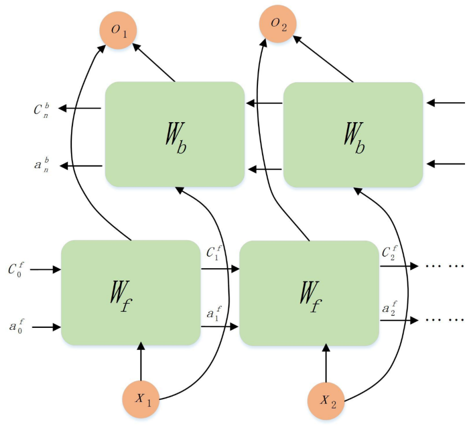
Implementation of RNN

The implementation of RNN is almost the same as the one in Assignment three for *Writing-A-Poem*. One thing special, the BiLSTM is used in this task, which we will dive deeper into the difference. Like the style in Assignment-3, the Cell is a manual version relies on autograd, without using developed layers and models in PyTorch or fastNLP.

Model

As I have mentioned before, the RNN differs the model in Assignment 3 is that the kernel of the model is a bidirectional LSTM. I have thought to use BiLSTM in Assignment 3, but the problem is that when we need to produce a poem, there is no future information which makes it impossible to predict the next character. However, in this task, we receive a whole sequence, and only need to get the output after all the sequence has be loaded, which makes it possible to have a BiLSTM.

So, since all the other part of the model is the same as the former one, I will only discuss about the BiLSTM layer. The structure of BiLSTM is simply a combination of two alike LSTM cells. The only difference is the input order of the sequence will be reversed for the second LSTM. Finally, we get two hidden state vectors. We can simply concatenate the two outputs to form a new hidden state as a input to the output layer. Therefore, the input size of the output layer will be doubled while the output keeps its size.



Because of the length of this task is much longer than that of Assignment 3, the time needed for training increases, too, so a single layer BiLSTM is used instead of a double layers one.

About other parameters, 128 for hidden size for both LSTM is chosen, and the embedding size is cut down to 128. Pre-train word2vector is not used in this task, although the model with pre-train has been shown holds better performance in the Assignment 3. And the output size of the output layer is the same as the number of classes, which is 4 in this task.

And we use the last hidden state only as the output to the output layer, to get the result of this network and to compute loss and BPTT.

Train

The training process is mainly driven by the trainer provided by fastNLP, so the implementation will be much easier. And it seems that the trainer of fastNLP has implemented BATCH naturally, so I need not worry about the batch mechanism.

Initialization. The initialization keeps the method which is proven to be useful in Assignment 3. To make it clear, the small-initialization. Since the use of 1-initialization for *forget gate* has not been proven, it will not be used here. So, all the parameter in LSTM is initialized with small random floats.

Optimizer. According to Assignment 3, *RMSprop* shows the best performance over all, so it will be applied without further tests on other kinds of optimizers.

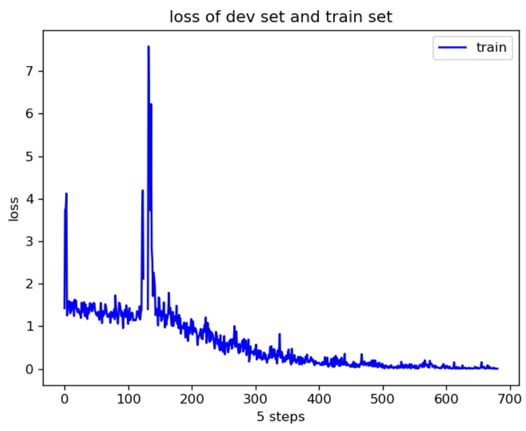
Overfitting. In this task, basically, a model with higher accuracy still remains relative higher accuracy in the test set, which means that overfitting in this task is not that terrible. And the problem can be solved easily by Early-Stop or Dropout. For convenience, Dropout is chosen here to prevent from overfitting.

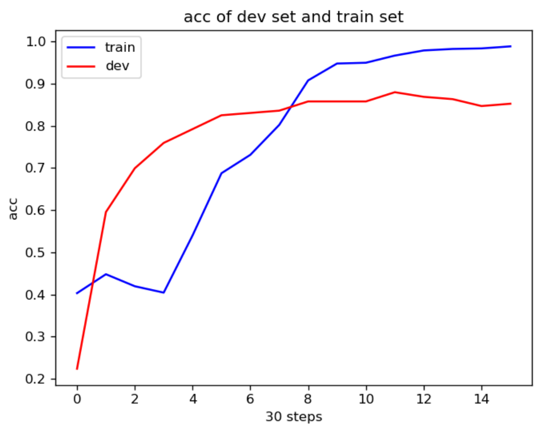
Loss Function. Needless to say, the loss function is the same as the choice in Assignment 2, the Cross Entropy Loss.

Result

The training of this model is extremely slow, and doesn't have a very satisfying performance.

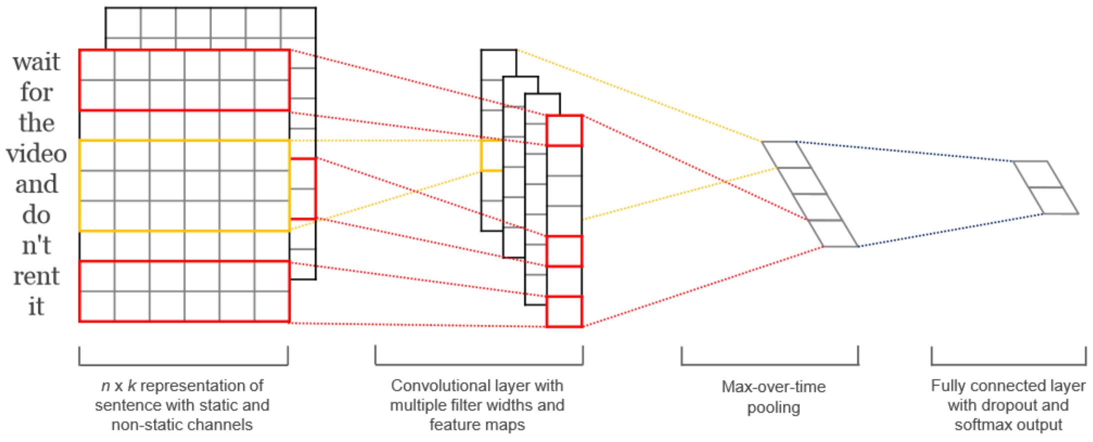
After q0 epochs of training on BiLSTM, the accuracy of the training set is 99.9%, and the accuracy on the test set is about 88.5%. And from the figures, we could figure out that there's some overfitting of this model after the 11th epochs. But in a word, the performance is not as good as the that of Assignment 2. And one thing strange is that although the accuracy of the training set outperforms the accuracy of the development set at last, the process is quite out of expectations.





Implementation of CNN

The implementation of CNN here is a recurrent model based on *Convolutional Neural Networks for Sentence Classification*, which includes Embedding layer, Conv1D layer, pooling layer and output layer. As the implementation in RNN, the implementation of CNN cell is a manual version relies on autograd. Also, fastNLP and PyTorch are used in this task.


wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Model

As I have mentioned before, CNN contains Embedding layer, Conv1D layer, pooling layer and output layer.

Conv1D

Conv1D is a necessary part of a CNN, which uses a window to visit every part of the whole sentence. Here the window will be a $ker * width$ long vector w. It's goal is to map the original part of the sentence into a scalar, which we can call it a feature.

$$c = wx + b$$

x is the input of a reshaped sequence of word vectors with a shape of $(1, ker * width)$, and b is the bias of this window. Conv1D is formed by many windows with different widths. Because their random initialization, the feature each window tries to get varies, which means that with more windows, we could get more information from the original data. In the paper, windows' widths are 3 to 5 , and there are 100 windows for each width. To make the training faster, 5 windows for every width are used instead of 100. So the output size of this layer will be $seq \times (3 \times 5) = seq \times 15$.

Pooling

This layer simply tries to get the most important feature of the sequence for each window. Max pooling strategy is used here. The largest number of vector produced by each window is selected as the output of this layer, so the final output of this layer will be 15.

Output layer

Output layer is a fully connected layer with map the output of pooling layer to the output size which is the target vector, a one-hot classifier.

Train

Most training details here are basically the same as the settings in RNN we have mentioned before.

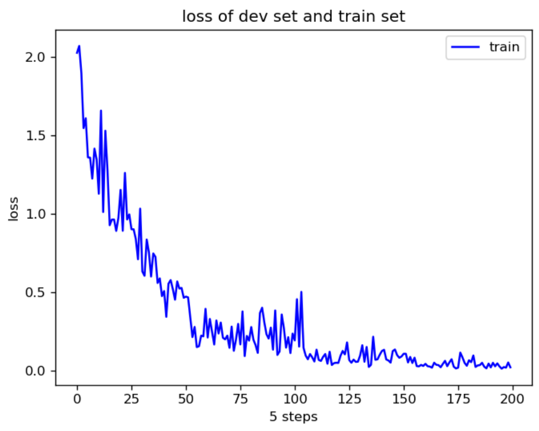
Initialization. Although a regularized initialization suites almost all the tasks we have met till now, I forgot to forget to initialize the parameters in this way. Therefore, I spent some time training a useless model. Finally, I choose to use small-initialization just like what I have done in LSTM, and it seems that it works well.

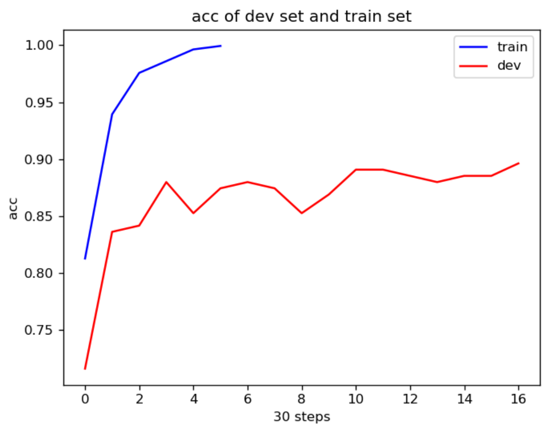
Optimizer. Because I didn't have special explorations of the optimizers in CNN, and to have a better view of the two models, I choose the same optimizer as RNN, which is *RMSprop*.

Overfitting. Also the same as RNN, only Dropout is introduced to solve the problem of overfitting.

Loss Function. The Cross Entropy.

Result

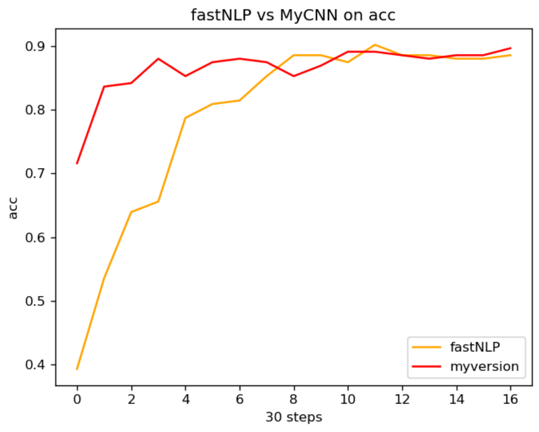


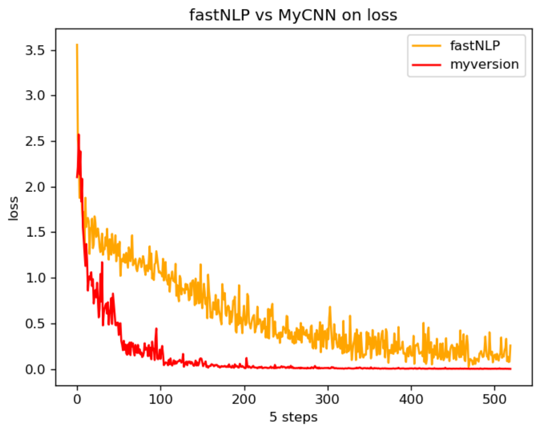


The fluctuations are almost the same during the training process of RNN and CNN. And the accuracy of the training set also goes up to 99.9%. And the if the model has grown overfitting is not very obvious.

Comparisons of fastNLP version with mine.

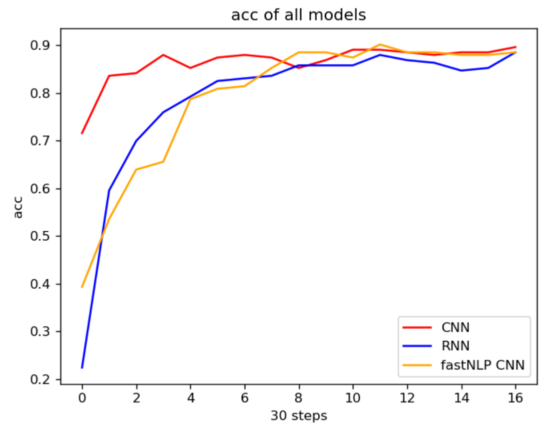
One thing weird is that I have tried two kinds of CNN, the one that I implemented myself and the version provided by fastNLP. Even the fastNLP version has more Conv1D layers, it runs much faster than my version. And the model which uses only components provided by `torch` also keeps a high training speed. They keep the same accuracy on the test set at last. But it's hard to explain why the PyTorch version is much faster. Some information shows that the real computation in Conv provided by torch is not in a normal way. In fact, it' s obvious that the **Conv** layer has a lot to be optimized.

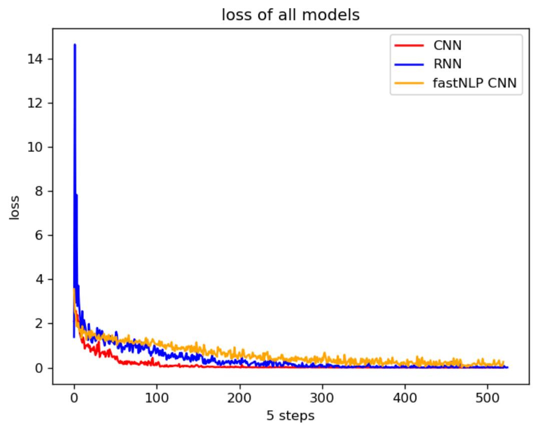




Comparisons of RNN and CNN

Here we will try to compare the different models, also the fastNLP version CNN.





So, it seems that a CNN with a relative small number of layers can do a better job in this task. It convergences more quickly and reaches a higher accuracy on the test set. And because the large number of parameters of fastNLP CNN, the training will be much slower and the performance isn't guaranteed on the little dataset.

Conclusions

The two methods seem not overperform the method in Assignment 2, and they cost much more time to get a final solution. One possible reason is that the dataset is too small to train the model, which is not enough for the model to consume the data well. And a CNN with a small number of layers may be more suitable for this task.

Part 2: About fastNLP

In this part, the pros and cons of fastNLP will be discussed. In this task, fastNLP is used to deal with the dataset including, build a dataset, building the vocabulary, and the whole training process.

Roughly speaking, fastNLP provides me with satisfying experiences, especially the training part. The trainer is quite easy to be used. The callback system is wonderful!

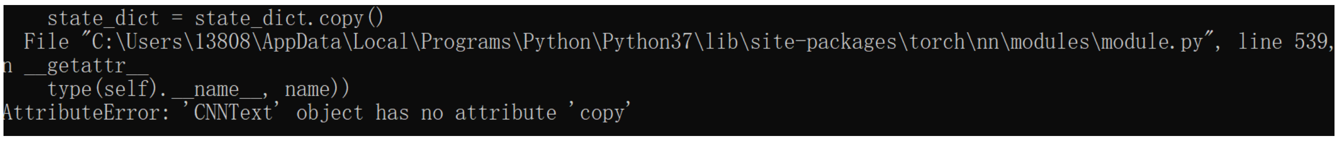
The Docs

The most important, the structure of the repo on GitHub is quite confusing. In a word, I think **a list for all the well-developed tools** in fastNLP is necessary. For a new hand to fastNLP like me, the first thing he will look for is the tutorials for certain problem when he has one. But he will soon need to dive deeper into this system, for there are always cases that special demands show up.

In fact, although almost all the key features of fastNLP are written in the front of source codes, and notes in codes are very clear, they are not systematic. One reason is that I don't keep a whole concept of the structure of this repo, and I will never know what kinds of methods and tools are provided until I read the source code myself, which is really inconvenient. In fact, when I need to use a component like trainer, it will usually not go smoothly, and the only thing I could do is to look up in the source codes which is really frustrating and time-consuming.

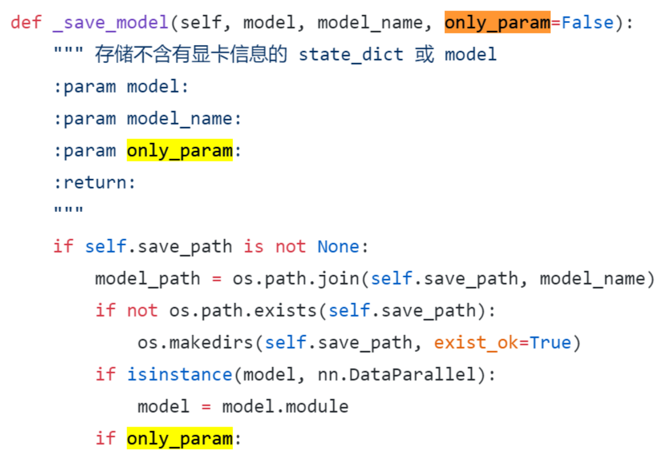
The save path problem

The last thing I'm not sure if there's something wrong is the `save_path` problem. I used the `save_path` argument to save my trained model, and it seemed that it really did this work and produced a record exactly where I had asked. But the problem lied in the load of the model. Although I used the same method to load the model parameters like when I did Assignment 3, but an error raises saying that I do not have a copy for the variables. And I have looked the source code of this part in fastNLP, and finds that the method I used to store the model is the same as the source code of fastNLP, but it just won't work when I want to load it. And the source code tells me to use `.state_dict()` to load the model, and it actually works. So, the problem is that I do not know why the old method just went wrong.



```
    state_dict = state_dict.copy()
  File "C:\Users\13808\AppData\Local\Programs\Python\Python37\lib\site-packages\torch\nn\modules\module.py", line 539,
n __getattr__
    type(self).__name__, name))
AttributeError: 'CNNText' object has no attribute 'copy'
```

Another related matter is that it seems there is an argument for users to control how to store the model, but I didn't find any method to change this argument. If this is just an unfinished function?

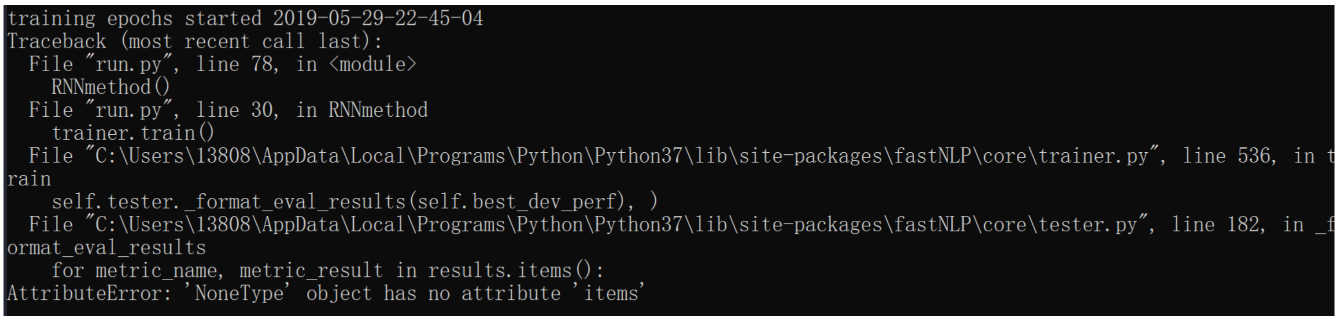


```
def _save_model(self, model, model_name, only_param=False):
    """ 存储不含有显卡信息的 state_dict 或 model
    :param model:
    :param model_name:
    :param only_param:
    :return:
    """
    if self.save_path is not None:
        model_path = os.path.join(self.save_path, model_name)
        if not os.path.exists(self.save_path):
            os.makedirs(self.save_path, exist_ok=True)
        if isinstance(model, nn.DataParallel):
            model = model.module
        if only_param:
```

One more thing about the `save_path` is that when I load the model I have saved through the suggested way, the accuracy for the model suddenly drops, but I don't know why. And the loaded model could go back to a normal status after few training epochs.

Others

And another bug when starting training the model. I happened to meet this problem at the beginning of this task, and I didn't figure out the reason. Oddly, it disappeared automatically. However, I finally meet this problem again, when I wrote this report...



```
training epochs started 2019-05-29-22-45-04
Traceback (most recent call last):
  File "run.py", line 78, in <module>
    RNNmethod()
  File "run.py", line 30, in RNNmethod
    trainer.train()
  File "C:\Users\13808\AppData\Local\Programs\Python\Python37\lib\site-packages\fastNLP\core\trainer.py", line 536, in t
rain
    self.tester._format_eval_results(self.best_dev_perf), )
  File "C:\Users\13808\AppData\Local\Programs\Python\Python37\lib\site-packages\fastNLP\core\tester.py", line 182, in _f
ormat_eval_results
    for metric_name, metric_result in results.items():
AttributeError: 'NoneType' object has no attribute 'items'
```

After tests, this error raises when I add a Callback class to it with the form:

```
class MyCallback(Callback):
    def on_backward_begin(self, loss):
        with open('.record/loss record rnn.txt','a') as f:
            f.write(loss)
            f.write(' ')
            f.close()

    def on_valid_end(self, eval_result, metric_key, optimizer, is_better_eval):
        with open('.record/acc record rnn.txt','a') as f:
            f.write(eval_result['AccuracyMetric']['acc'])
            f.write(' ')
            f.close()
```

Finally, I make sure that the problem here is that only string is accepted by the `write` function, but I have no idea why the raised problem has nothing with the real error? But surely, this is not the only reason raises this error, because when I fisrt met the error, no Callback was used.

PS:

I tried to use `fitlog`, but it seems that it does not support Windows very well. Unfortunately, the partition for my Ubuntu is almost full-filled. Although the functions `fitlog` provides are really attractive, I happen to do not have much time to take care of the reinstallation of Ubuntu or other kinds of jobs, so I'm sure I will have a taste in the near future.

Refereces

1. Christopher M.Bishop, *Recognizing and Machine Learning*.
2. 《神经网络与深度学习》
3. YoonKim, *Convolutional Neural Networks for Sentence Classification*