Assignment 4

Datasets: In this assignment, I chose eight categories from 20 news groups in sklearn. They are: ['comp.os.ms-windows.misc', 'misc.forsale', 'rec.motorcycles', 'sci.med', 'sci.space', 'soc.religion.christian', 'talk.politics.misc', 'talk.religion.misc'] There are 4402 samples in the training dataset, and 2931 samples in the test dataset.

Data processing with fastnlp

In order to use the fastnlp framework, we have to process the data to follow the standard format of fastnlp. First, we have to understand the basic components of fastnlp dataset, namely, Instance. Every data record is treated as an Instance, which has different fields. In the text classification problem, there are two basic fields: **text**, the raw sentence; **target**, the class one sentence belongs to. However, in order to model the whole sentence, we have to first split it up into words, and remove all the unnecessary punctuations and whitespaces. After that, we can construct a vocabulary of the training dataset based on the Vocabulary module of fastnlp. In this assignment, I filtered all the words whose frequencies are below 5. Then every sentence is mapped to a sequence of integers, which is renamed to **word_seq** and treated as the input of our model.

```python
def get_fastnlp_dataset():
    text_train, text_test = get_text_classification_datasets()
    train_data = DataSet()
    test_data = DataSet()
    for i in range(len(text_train.data)):
        train_data.append(Instance(text=split_sent(text_train.data[i]), target=int(text_train.target
[i])))
    for i in range(len(text_test.data)):
        test_data.append(Instance(text=split_sent(text_test.data[i]), target=int(text_test.target
[i])))

    # 构建词表
    vocab = Vocabulary(min_freq=5, unknown='<unk>', padding='<pad>')
    train_data.apply(lambda x: [vocab.add(word) for word in x['text']])
    vocab.build_vocab()

    # 根据词表映射句子
    train_data.apply(lambda x: [vocab.to_index(word) for word in x['text']], new_field_name='word_se
q')
    test_data.apply(lambda x: [vocab.to_index(word) for word in x['text']], new_field_name='word_se
q')

    # 设定特征域和标签域
    train_data.set_input("word_seq")
    test_data.set_input("word_seq")
    train_data.set_target("target")
    test_data.set_target("target")

    return train_data, test_data, vocab
```

```python


def split_sent(line):
    line = line.lower()
    for c in string.punctuation:
        line = line.replace(c, "")
    for w in string.whitespace:
        line = line.replace(w, " ")
    line = line.split()
    return line
```

Word2Vec

We use Word2Vec to generate word embeddings rather than randomly initialize the embeddings. In the field of Natural Language Processing, traditional statistical language model often suffers from data sparsity and curse of dimensionality. However, Word2Vec, as a distributed representation of word, can mark the similarity between two words based on their contexts.

In this assignment, I trained a simple skip-gram model for Word2Vec using genism, and calculated the pre-trained weight for nn.Embedding.

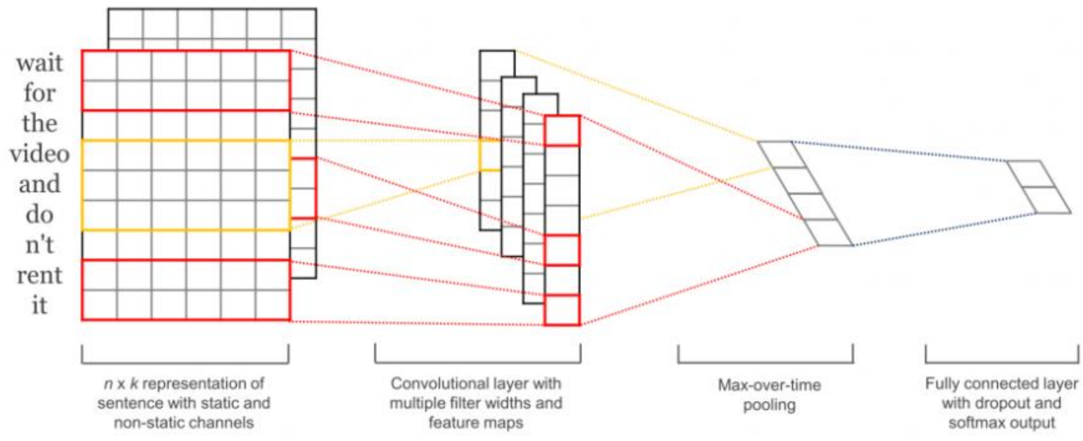
```python
def get_word2vec():
    categories = ['comp.os.ms-windows.misc', 'misc.forsale', 'rec.motorcycles', 'sci.med', 'sci.spac
e',
                  'soc.religion.christian', 'talk.politics.misc', 'talk.religion.misc']
    dataset_train = fetch_20newsgroups(subset='train', categories=categories, data_home='../../..')
    sentences = []
    for sent in dataset_train.data:
        sentences.append(split_sent(sent))
    model = Word2Vec(sentences, sg=1, size=50, window=3)
    model.save('word2vec')
```

```python
def get_pretrained_weight(vocab, embed_size=50):
    model = Word2Vec.load('word2vec')
    weight = torch.zeros(len(vocab.idx2word), embed_size)

    for i in range(len(model.wv.index2word)):
        try:
            index = vocab.word2idx[model.wv.index2word[i]]
        except:
            continue
        weight[index, :] = torch.from_numpy(model.wv.get_vector(vocab.idx2word[vocab.word2idx[model.w
v.index2word[i]]]))
    return weight
```

CNN model for text classification

Yoon Kim proposed the textCNN model for sentence classification. When CNN is applied for text classification tasks, multiple kernels of different sizes are used to extract features from the sentence, which is similar to ngram models with different window sizes.

The network looks roughly as follows:


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

The first layer embeds words into low-dimensional vectors. The next layer performs convolutions over the embedded word vectors using multiple filter sizes. Next, we max-pool the result of the convolutional layer into a long feature vector, add dropout regularization and classify the result using a softmax layer.

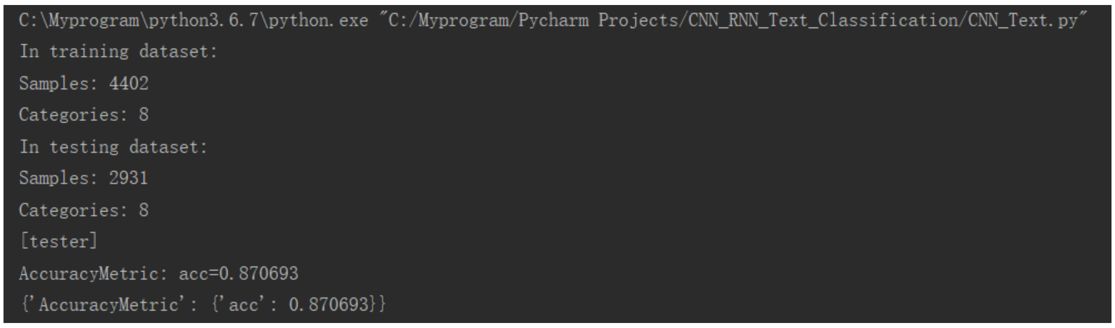
The hyperparameters are as below: filter windows of 3,4,5 with 100 feature maps each, dropout rate is 0.1.

```python
class myCNNText(nn.Module):
    """
    Text classification model based on CNN
    """
    def __init__(self, embed_num, embed_dim, num_classes, kernel_nums=(100, 100, 100), kernel_sizes=(3,4,5), padding=0, dropout=0.5, pre_weight=None):
        super(myCNNText, self).__init__()
        if pre_weight is None:
            self.embed = nn.Embedding(embed_num, embed_dim)
        else:
            self.embed = nn.Embedding.from_pretrained(pre_weight)
        self.convs1 = nn.ModuleList([nn.Conv2d(1, kernel_nums[i], (kernel_sizes[i], embed_dim)) for i in range(len(kernel_sizes))])
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(sum(kernel_nums), num_classes)

    def conv_and_pool(self, x, conv):
        x = F.relu(conv(x)).squeeze(3)
        x = F.max_pool1d(x, x.size(2)).squeeze(2)
        return x

    def forward(self, word_seq):
        x = self.embed(word_seq)  # [N, L] -> [N, L, C]
        x = x.unsqueeze(1)
        x = [self.conv_and_pool(x, conv) for conv in self.convs1]
        x = torch.cat(x, 1)
        x = self.dropout(x)
        x = self.fc(x)

        return {'pred': x}

    def predict(self, word_seq):
        output = self(word_seq)
        _, predict = output['pred'].max(dim=1)
        return {'pred': predict}
```

The trainer process supported by fastnlp:

```python
train_data, test_data, vocab = get_fastnlp_dataset()
weight = get_pretrained_weight(vocab)
cnn_text_model = myCNNText(embed_num=len(vocab), embed_dim=50, num_classes=8, padding=2, dropout=0.1, pre_weight=weight)
trainer = Trainer(train_data=train_data, model=cnn_text_model,
                  loss=CrossEntropyLoss(pred='pred', target='target'),
                  metrics=AccuracyMetric(),
                  n_epochs=10,
                  batch_size=32,
                  print_every=-1,
                  validate_every=-1,
                  dev_data=test_data,
                  save_path='./model',
                  optimizer=Adam(lr=1e-3, weight_decay=0),
                  check_code_level=-1,
                  metric_key='acc',
                  use_tqdm=False,
                  )
trainer.train()
```

Actually, the model has a quite good performance. The final accuracy of it is shown below(loaded the best model)



```
C:\Myprogram\python3.6.7\python.exe "C:/Myprogram/Pycharm Projects/CNN_RNN_Text_Classification/CNN_Text.py"
In training dataset:
Samples: 4402
Categories: 8
In testing dataset:
Samples: 2931
Categories: 8
[tester]
AccuracyMetric: acc=0.870693
{'AccuracyMetric': {'acc': 0.870693}}
```

RNN model for text classification

RNN are one of the most popular architectures used in NLP problems for their recurrent structure is very suitable to process the variable-length text. Among different types of RNN cells, LSTM is considered very efficient when there exists long-term dependencies. The output at the last moment h_T is regarded as the representation of whole sequence, which has a fully connected layer followed by a softmax layer that predicts the probability distribution over classes. Therefore, LSTM network can also be used as text classifiers.

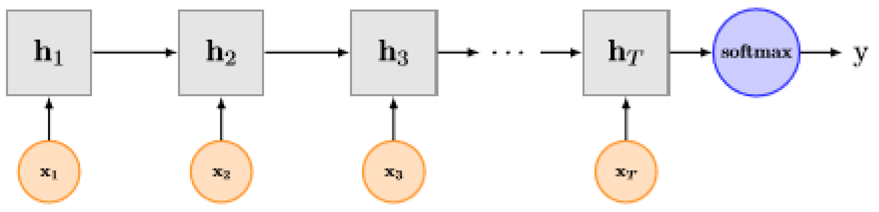
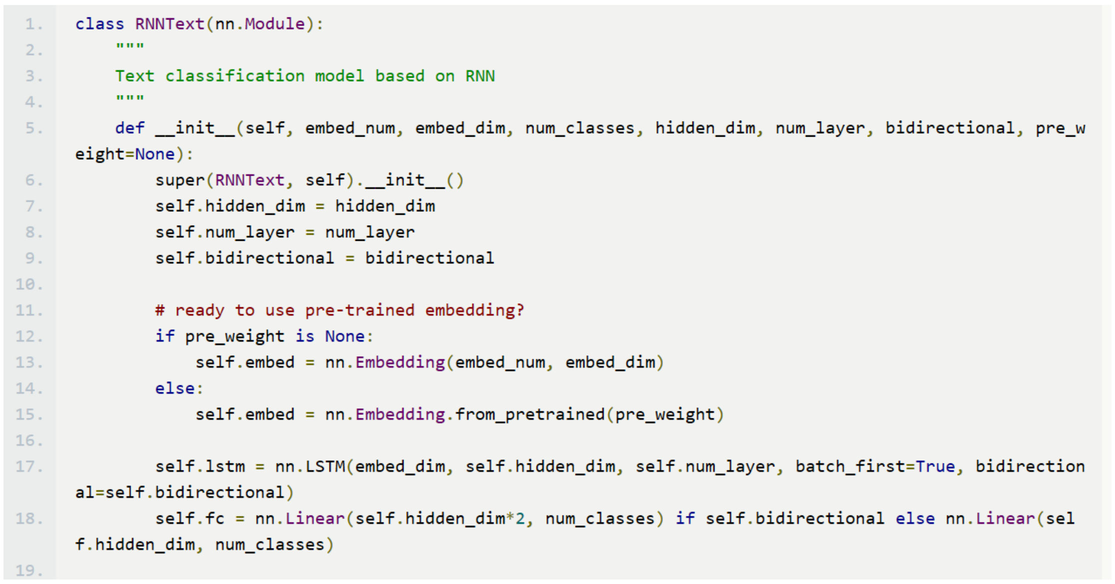
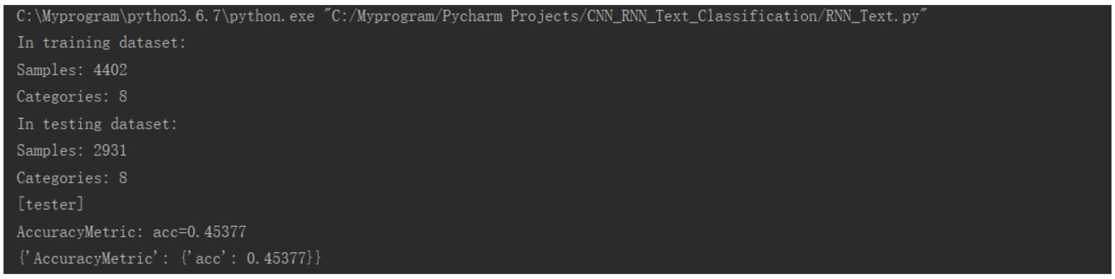


Figure 1: Recurrent Neural Network for Classification



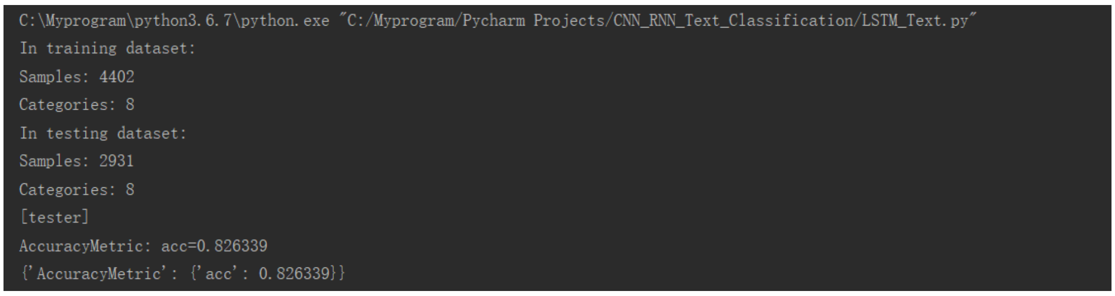
```python
class RNNText(nn.Module):
    """
    Text classification model based on RNN
    """
    def __init__(self, embed_num, embed_dim, num_classes, hidden_dim, num_layer, bidirectional, pre_weight=None):
        super(RNNText, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layer = num_layer
        self.bidirectional = bidirectional

        # ready to use pre-trained embedding?
        if pre_weight is None:
            self.embed = nn.Embedding(embed_num, embed_dim)
        else:
            self.embed = nn.Embedding.from_pretrained(pre_weight)

        self.lstm = nn.LSTM(embed_dim, self.hidden_dim, self.num_layer, batch_first=True, bidirectional=self.bidirectional)
        self.fc = nn.Linear(self.hidden_dim*2, num_classes) if self.bidirectional else nn.Linear(self.hidden_dim, num_classes)

    def forward(self, word_seq):
        x = self.embed(word_seq)
        h0 = torch.zeros(self.num_layer*2, x.size(0), self.hidden_dim) if self.bidirectional else torch.zeros(self.num_layer, x.size(0), self.hidden_dim)
        c0 = torch.zeros(self.num_layer*2, x.size(0), self.hidden_dim) if self.bidirectional else torch.zeros(self.num_layer, x.size(0), self.hidden_dim)
        out, (hn, cn) = self.lstm(x, (h0, c0))

        out = self.fc(out[:, -1, :])
        return {'pred': out}

    def predict(self, word_seq):
        output = self(word_seq)
        _, predict = output['pred'].max(dim=1)
        return {'pred': predict}
```

The training of a RNN model is really slow. What's worse, it cannot achieve the same performance as CNN. The final accuracy is only 0.45, even worse than logistic regression.



```
C:\Myprogram\python3.6.7\python.exe "C:/Myprogram/Pycharm Projects/CNN_RNN_Text_Classification/RNN_Text.py"
In training dataset:
Samples: 4402
Categories: 8
In testing dataset:
Samples: 2931
Categories: 8
[tester]
AccuracyMetric: acc=0.45377
{'AccuracyMetric': {'acc': 0.45377}}
```

However, later I discovered that, RNN-based model can achieve good results through little modification. In the original model, the final stage output is fed to the full-connect layer. However, if we want to utilize all the features of the outputs at different time steps, namely, feeding the average of the outputs at different time steps to the full-connect layer, the performance will be much better and convergence of the loss curve becomes faster. The final accuracy of the modified model is shown below.



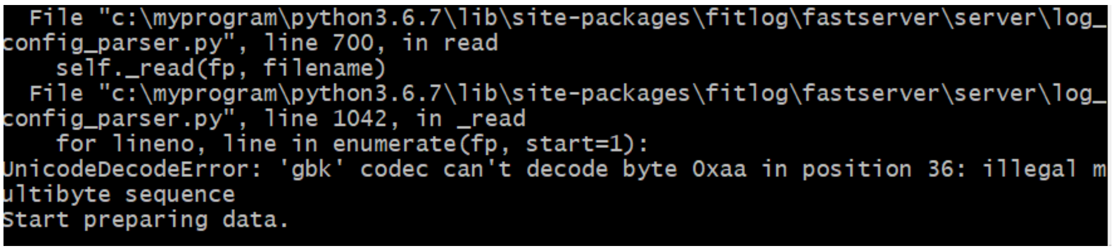
```
C:\Myprogram\python3.6.7\python.exe "C:/Myprogram/Pycharm Projects/CNN_RNN_Text_Classification/LSTM_Text.py"
In training dataset:
Samples: 4402
Categories: 8
In testing dataset:
Samples: 2931
Categories: 8
[tester]
AccuracyMetric: acc=0.826339
{'AccuracyMetric': {'acc': 0.826339}}
```

About fastnlp

Actually, I think fastnlp is a good framework for fast and tidy implementation, training and testing of neural network models in NLP. It does provide great convenience for us, especially in dataset preparation and training support. However, as we all know, high encapsulation means less freedom and harder debugging.

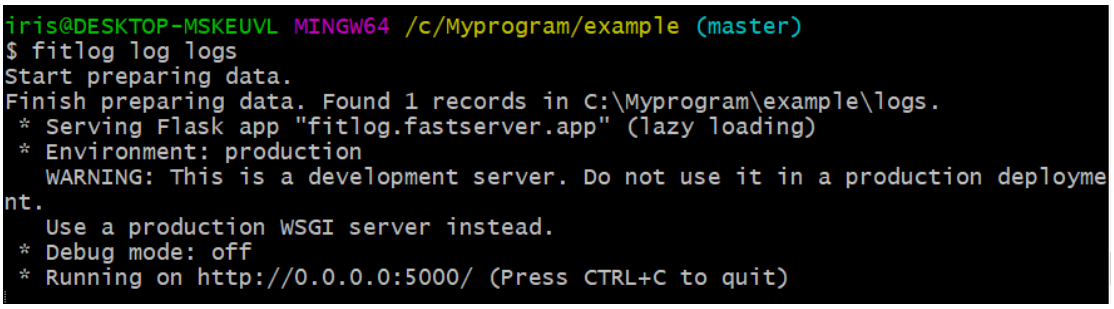
During my usage of it, first I read the advance tutorial thoroughly. The tutorial is in jupyter notebook form, which is quite clear and I have no difficulty learning how to use fastnlp through the examples it provides. But it was not until the new version of fastnlp became available that I found its API docs. I found that it can really save a lot of effort. Besides, some rules of it (eg. the input parameters of forward should be word_seq) are consistent with common practices in machine learning, so it's

quite easy for beginners to follow.
However, there are really something I need to complain about. The biggest challenge I found is to use fitlog. I don't know why the author of the docs doesn't specify that the fitlog command is only available in Linux command line. I had to run my code in gitbash, however it still gives error messages saying that I have decoding error(I found the default encoding of the read function is None, so I have to change the source code of fitlog to decoding the file as utf-8)



```
  File "c:\myprogram\python3.6.7\lib\site-packages\fitlog\fastserver\server\log_
config_parser.py", line 700, in read
    self._read(fp, filename)
  File "c:\myprogram\python3.6.7\lib\site-packages\fitlog\fastserver\server\log_
config_parser.py", line 1042, in _read
    for lineno, line in enumerate(fp, start=1):
UnicodeDecodeError: 'gbk' codec can't decode byte 0xaa in position 36: illegal m
ultibyte sequence
Start preparing data.
```

But it still doesn't work on Windows, as I can't open the url 0.0.0.0:5000.



```
iris@DESKTOP-MSKEUVL MINGW64 /c/Myprogram/example (master)
$ fitlog log logs
Start preparing data.
Finish preparing data. Found 1 records in C:\Myprogram\example\logs.
 * Serving Flask app "fitlog.fastserver.app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
nt.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

I'm still at a loss, and I don't know what to do, as I don't want to bother setting up the environment in my virtual machine.

As to other nlp frameworks, I have heard about or used some NLP frameworks before, such as nltk and genism. They are not built on top of deep learning, but I think there are something in them that's valuable for the improvement of fastnlp.

1. More easy-to-use modules for some basic NLP tasks, such as NER and POS.
2. Built-in support for some pre-trained embeddings, such as BERT or Word2Vec.
3. More freedom for the training and saving procedure of the model. For example, the filename of our saved model can be specified as we like, or save the model after several epochs.