NER using CRF In tensorflow

Anand Namdev - 163050068
Kevin macwan - 163050053
Saurav Shrivastava - 163050028

February 16, 2018

Disclaimer: The complete code is written by ourselves. Only a tutorial publicly available on crf is followed to understand crf.
Link: [Tensorflow Git Repository]

Note: We have made separate directory for each steps named, Step1, Step2 & Step3. In each directory there is file named rollnum.sh which runs the main training file named main.py(for training) and java file named Main.java(for converting in test.txt format).

Dataset Description:

Corpus constituted of many documents where each document consist of many sentences separated by a empty line.Each document start with keyword "-DOCSTART-" The data files contain four columns separated by a single space. Each word has been put on a separate line where description of these four column are as follows:
First field : word
Second field : part-of-speech (POS) tag
Third field : Syntactic chunk tag
Fourth Field : Named entity tag(NER) - 8 possible values of tag
1) Inter-Location('I-LOC')
2) Begin Location('B-ORG')
3) Other 'O'
4) Inter-Person('I-PER')
5) Inter-Miscellaneous('MISC')
6) Before Miscellaneous('B-MISC')
7) Inter-Organization('I-ORG')
8) Before Location('B-LOC')

The chunk tags and the named entity tags have the format I-TYPE which means that the word is inside a phrase of type TYPE. Only if two phrases of the same type immediately follow each other, the first word of the second phrase will have tag B-TYPE to show that it starts a new phrase for example word "New York" here "New" is 'B-LOC' and York with 'I-LOC' NER tag. A word with tag O is not part of a phrase.

Task 1 : Simple linear chain CRF using Tensorflow without word embeddings.

What we did?

1. We first parsed train,dev and test file to create sequences as sentence wise since we can directly feed these sentences in CRF. In original data each word of a sentence is in new line which are separated by a blank line "\n".
2. We then parsed train.txt to collect all the words. There were around 21k distinct words. In order to produce the vocabulary, we picked the top 10,000 words by reverse sorting with their frequency. (the top k words is a parameter and can be run multiple times with trying different variants, it depends on memory availability of machine. We ran with 10,000).
3. We assigned those 10k words to a distinct index by maintaining a data structure wordToIndex = dict() which assigns separate index to each word from top 10k.
4. We also maintained a word unknown as 'unk' with index 0 for all those words which are not in top 10k, since those words would also need to be represented by some index.
5. After parsing training file, we parsed dev file and test file where is a word is in vocabulary it will be represented by that index otherwise by word "unk".
6. We also parsed all the NER tags, there are about 8 diff tags in train file. We gave a separate index to do the classification.
7. Now the data is fed in tensorflow code which can be understood by following the function do_train() in file main.py

About all the files present in folder step 1 :

- 163050068.sh: main bash file which runs all other modules.It calls main.py which does the training and runs on dev,test files on regular intervals.

- Once training is over, main.py creates a file named ‘pred.test.txt’ which is our output file. Basically it contains output NER tags of test file separated by new line. Each line contains tags for each sentence.
- Java code Main.java is compiled.
- Since the output format has to be in the same format as given test.txt, we created the same output format with the help of test.txt file. Main.java takes 3 args as input:
    - Our output file
    - text.txt (original test file)
    - The fully qualified file name which you want as output. Its format will be same as test.txt
- Data: contains train.txt, dev.txt, test.txt

Training details:

1. Code is trained for 50 iterations. During each iteration, code is tested on train data to see performance after every iteration on training file. While testing on dev and test file is done at an interval of 5 and 10 epochs respectively.
2. Once code is trained for 50 iterations, a final testing on dev and test file is done. During final testing, a file is written named ‘pred.test.txt’ which is our output file. This file contains numbers separated by new line.
3. The file ‘pred.test.txt’ is executed with Java code to produce test file in your specified format.

Here since each word is represented by vector of 10,000 it took a lot more time to train in comparison to training using GloVe word vectors.

Format: (epoch_num, #correct tags predicted, total num of tags, acc)
('Running Epoch on Training: ', 0)
(0, 167358, 204563, 81.8124489765989)
****
('Testing On Dev data: ', 0)
(0, 42964, 51574, 83.30554155194478)
****
('Testing On Test data: ', 0)
(0, 38411, 46666, 82.31046157802254)

****
Final Testing On Dev data:
(42875, 51574, 83.13297397913678)
****
Final Testing On Test data:
(38204, 46666, 81.8668838126259)
****
Final Accuracy: Format: (epoch_num, Acc)
Train: [(0, 81.8124489765989)]
****
DEv: [(0, 83.30554155194478), (1, 83.13297397913678)]
****
Test: [(0, 82.31046157802254), (1, 81.8668838126259)]

We got final accuracy after running 50 epochs as

- Training Acc: 83.23
- Dev Acc: 82.9
- Test Acc: 82.4

Task 2 : Using Word Representation: We used GloVe (Global Vectors for Words) by Christopher Manning.

What we did?

1. We first parsed train,dev and test file to create sequences as sentence wise since we can directly feed these sentences in CRF. In original data each word of a sentence is in new line which are separated by a blank line "\n".
2. We first parsed train,dev and test file to create sequences as sentence wise since we can directly feed these sentences in CRF. In original data each word of a sentence is in new line which are separated by a blank line "\n".
3. In this step, words have be represented by their embedding. So we used GloVe word embedding. There are various files under GloVe trained on different corpus size and with different size. We used the file trained on news corpus of 6B tokens and has 200-d vectors for about 400,000 words.
4. We also maintained a 'unk' word token and a random word vector corresponding to that word. Since not every word from train,dev and test file can be found in glove file.
5. We parsed train,dev and test file sentence wise. If a word is present in glove file, we replaced it with its word vector. Otherwise by the 'unk' word vector.
8. We also parsed all the NER tags, there are about 8 diff tags in train file. We gave a separate index to do the classification.
9. Now the data is fed in tensorflow code which can be understood by following the function do_train() in file main.py

About all the files present in folder step2.

- 163050068.sh: main bash file which runs all other modules.It calls main.py which does the training and runs on dev,test files on regular intervals.
- Once training is over, main.py creates a file named 'pred.test.txt' which is our output file. Basically it contains output NER tags of test file separated by new line. Each line contains tags for each sentence.
- Java code Main.java is compiled. I

- Since the output format has to be in the same format as given test.txt, we created the same output format with the help of test.txt file. Main.java takes 3 args as input:
    - Our output file
    - text.txt (original test file)
    - The file name which you want as output. Its format will be same as test.txt

  Java code converts output which is in the form of tags(0 to 7) to expected format which was given by test.txt.
- Data: contains train.txt, dev.txt, test.txt

Intermediate files that Java code processes :

Sample sentence from Test.txt :

SOCCER NN I-NP O
- : O O
JAPAN NNP I-NP I-LOC
GET VB I-VP O
LUCKY NNP I-NP O
WIN NNP I-NP O
, , O O
CHINA NNP I-NP I-PER
IN IN I-PP O
SURPRISE DT I-NP O
DEFEAT NN I-NP O
. . O O

Corresponding Mapping to Gold Labels : *1 1 4 1 1 1 1 3 1 1 1 1*
Predicted Labels : *2 1 4 1 1 1 1 1 1 1 1 1*
Where the number correspond to tags as follows:

```
map.put(0,"O");
map.put(1,"I-ORG");
map.put(2,"I-MISC");
map.put(3,"I-PER");
map.put(4,"I-LOC");
map.put(5,"B-LOC");
map.put(6,"B-MISC");
```

```
map.put(7,"B-ORG");
```

Our java code converts this numbered file named as: ‘pred.test.txt’ to the file in test.txt format.

Training details:

1. Code is trained for 50 iterations. During each iteration, code is tested on train data to see performance after every iteration on training file. While testing on dev and test file is done at an interval of 5 and 10 epochs respectively.
2. Once code is trained for 50 iterations, a final testing on dev and test file is done. During final testing, a file is written named ‘pred.test.txt’ which is our output file. This file contains numbers separated by new line.
3. The file ‘pred.test.txt’ is executed with Java code to produce test file in your specified format.

4. The below table mentions a table of the execution.
   Format: (epoch number, NER tag accuracy)

Train Accuracy: [(0, 85.7251800178918), (1, 90.53983369426534), (2, 91.16506895186323), (3, 91.39678240933111), (4, 91.53610379198584), (5, 91.63876165288933), (6, 91.72577641117896), (7, 91.77857188250074), (8, 91.7888376685911), (9, 91.82452349642897), (10, 91.86265355905027), (11, 91.92767020428914), (12, 91.93989137820623), (13, 91.97606605300079), (14, 91.98975376778792), (15, 92.00295263561837), (16, 92.05623695389684), (17, 92.04939309650328), (18, 92.05917003563694), (19, 92.0748131382508), (20, 92.09338932260478), (21, 92.16231674349712), (22, 92.18284831567782), (23, 92.18235946872113), (24, 92.18382600959117), (25, 92.1916475608981), (26, 92.2620415226605), (27, 92.24737611396), (28, 92.23515494004292), (29, 92.24493187917659), (30, 92.25079804265678), (31, 92.30701544267536), (32, 92.29528311571497), (33, 92.29479426875828), (34, 92.30506005484862), (35, 92.31288160615556), (36, 92.3436789644266), (37, 92.33390202529294), (38, 92.335368566163), (39, 92.3265693209427), (40, 92.32070315746249), (41, 92.35932206704047), (42, 92.36861015921745), (43, 92.35296705660359), (44, 92.35981091399715), (45, 92.35296705660359), (46, 92.38229787400458), (47, 92.36909900617414), (48, 92.36274399573726), (49, 92.36812131226077)]

****

Dev Accuracy: [(0, 89.61492224764416), (5, 91.59266296971342), (10, 91.86993446310156), (15, 92.13169426455191), (20, 92.23833714662427), (25, 92.3507969131733), (30, 92.41672160390894), (35, 92.55826579284135), (40, 92.55632683134912), (45, 92.57959436925583), (50, 92.56796060030248)]

****

Test Accuracy: [(0, 89.55556507950114), (10, 91.25487506964386), (20, 91.4863069472421), (30, 91.59987999828569), (40, 91.69630994728496), (50, 91.76059657995114)]
****

So after training for 50 epochs, we got

- Training Acc: 92.36
- Dev Acc: 92.56
- Test Acc: 91.69

We also played with using different versions of GloVe file. But Glove.6B.200d.txt gave us the best results. We tried GloVe.6B.50d, GloVe840B.300d.txt also.

Step3: Same code as step2. Except the transition param is our own declared first randomly assigning vectors. Over the course of training, that transition parameter matrix will be learnt.
Q. How will you introduce model weights in the state transition objective?
Ans: By instroducing another transition matrix before the crf graph. Over the course of time, that transition matrix will be learnt.

So after training for 10 epochs, we got

- Training Acc: 92.36
- Dev Acc: 92.56
- Test Acc: 91.69