Natural Language Processing
Sentiment Analysis

Silin DU

Department of Management Science and Engineering, Tsinghua University
dsl21@mails.tsinghua.edu.cn

April 29, 2022

清華大學
Tsinghua University

Contents

1 Sentiment Analysis

1.1 Environments and Project Structure

We use PyTorch to implement a RNN-based model to perform the sentiment analysis task. The required environments are as follows.

- torch-1.11.0
- torchtext-0.12.0
- pytreebank-0.2.7: used to load the datasets.

The structure of our project is as follows.

- /codes: contain all the codes.
  - test.py: the entrance of our project.
  - train.py: define the class to train the model.
  - model.py: define the model.
  - utils.py: load the data for training and evaluations.
  - config.yaml: store the configurations for model trianing.
- /weight: the directory to save models, training logs and results.
- /data: the directory of the datasets and pretrained emebddings.

We provide our training logs and evaluation results in /weight directory. Several configurations needed to be predefined in config.yaml, shown in Table 1.

To run our codes, use the following command.

```
python test.py --config='config.yaml'
```

1.2 Details

In this subsection, we introduce the details of our models and experimental results.

We implement a RNN-based model using LSTM units, shown in Figure 1. This model takes the word embeddings as inputs, and learns the sequential information in word sequences. Then the output of the LSTM layer is aggregated by an attention layer, which can better capture the relationships between different words. Finally, a fully-connected network is used to decode the sentiment class labels.

We preprocess the Stanford Sentiment Treebank datasets as follows.

1. We use the `pytreebank` package to load the datasets from local files.
2. We use the English tokenizer from `torchtext.data.utils` to tokenize all the sentences, and organize them with their labels in the form like $([\text{word}_1, ..., \text{word}_n], \text{label})$.
3. Then we use `torchtext.vocab.build_vocab_from_iterator` to get the unique words/tokens in the dataset. We do not omit any words. Finally, we get a vocabulary containing around 1.6K words.
4. To make use of the pretrained embeddings, we extract the pretrained embeddings of each word and organize them according to their word IDs in our vocabulary. Around

Table 1: Configurations

Name	Description
Model Saving	
model_name	Model name used to save a model.
model_dir	Directory to save models, training logs and results.
Data	
dataset_path	Directory of the dataset.
data_dir	Directory of corpus.
embedding_path	Path of pretrained word embeddings.
Model Setting	
hidden_dim	Size of hidden layers.
num_layer	Number of layers.
dropout	Dropout rate.
pretrain	Whether to use pretrained embeddings, expected to `True` of `False`.
Training Setting	
batch_size	Batch size used in training, validation and testing.
learning_rate	Learning rate during training.
epochs	Total epochs of training.
weight_decay	The regularization parameter used in `torch.optim`, can be expressed like `2e-5`.
lr_decay	Whether to use learning rate decay, expected to be `True` of `False`.

1200 tokens are out of the vocabulary of pretrained embeddings, most of which are compound words. For those words, we use the embeddings of `'<unk>'` token. We will randomly initialize the embedding matrix of our data in the experiments without pretrained embeddings.

5. We observe that sentences in the datasets contain around 20 words on average. Therefore, we set the maximum length of sentences as 25, and do some padding for those sentences with less than 25 words.
6. Finally, we add a random embedding for `'<pad>'` token into our embedding matrix.

During the experiments, we consider the following parameter settings.

- Train with or without dropout. If used, the dropout rate is 0.2.
- Train with different hidden size, 256 or 512.
- Train with a different number of the hidden layer, 1 or 3.
- Train with/without pretrained word embedding.

There're total $2^4 = 16$ combinations of different parameter settings. We fix the pretrained embeddings during training. In each experiment, We train our model on the training set

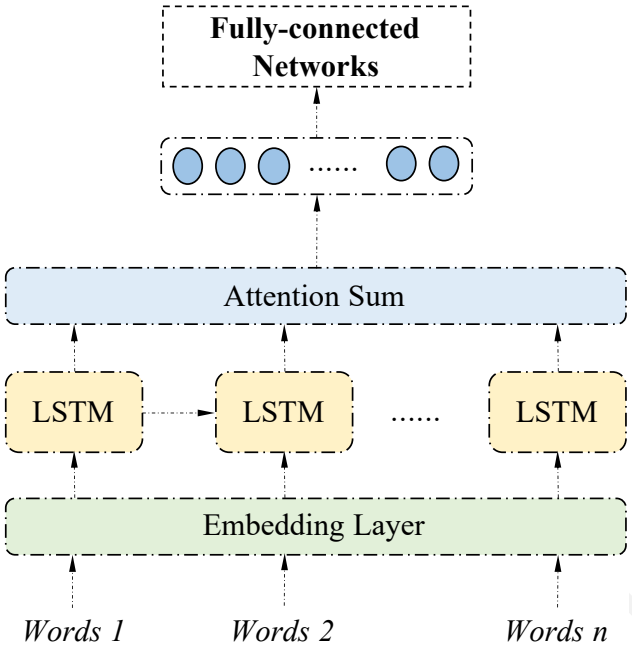

Figure 1: Model Structure

with the batch size of 128 and finetune the learning rate and the regularization parameter. The model is trained through 30 epochs, and the one who performs best on the validation set will be saved.

The experimental results are shown in Table 2.

Table 2: Experimental results of our models

Number of Experiments	Dropout	Hidden Size	Layer	Embedding	Accuracy
One-layer LSTM					
1	0	256	1	No	0.353
2	0	256	1	Yes	0.425
3	0	512	1	No	0.342
4	0	512	1	Yes	0.429
5	0.2	256	1	No	0.371
6	0.2	256	1	Yes	**0.447**
7	0.2	512	1	No	0.361
8	0.2	512	1	Yes	0.438
Three-layer LSTM					
9	0	256	3	No	0.341
10	0	256	3	Yes	0.419

Table 2 —— Next Page

Table 2 —— Continued

Number of Experiments	Dropout	Hidden Size	Layer	Embedding	Accuracy
11	0	512	3	No	0.364
12	0	512	3	Yes	0.415
13	0.2	256	3	No	0.339
14	0.2	256	3	Yes	0.424
15	0.2	512	3	No	0.346
16	0.2	512	3	Yes	0.425
Joined Model Multi-tasking[1] [2]	-	-	-	-	0.448

1.3 Discussion and Summary

From Table 2, we observe that

- Pretrained embeddings are quite essential for model performances, where each model with pretrained embeddings gets over 0.41 accuracy scores.
- Since the training set only contains around 8000 samples, all models suffer from the overfitting problems. Then we find that the dropout technique is useful (Experiment 1,2,3,4 *v.s.* 5,6,7,8).
- The increase of the number of hidden layers will cause more severe overfitting problems, which might hurt the performances (Experiment 1-8 *v.s.* 9-16).
- There's no significant difference between the models with 256 or 512 hidden units.
- The model in experiment 6 performs best with 0.447 classification accuracy, which has 256 hidden units and 1 hidden layer.
- Our models get competitive results compared with Nejat et al.(2017) [2].

2 Improved Version

2.1 Environments and Project Structure

The structure of this project is roughly the same. Codes are in the code directory.

- test_improved.py: the entrance of our project.
- train_improved.py: define the class to train the model.
- model_improved.py: define the improved model.
- utils_improved.py: load the data for training and evaluations.

[1]Rank 23-rd, from: https://paperswithcode.com/sota/sentiment-analysis-on-sst-5-fine-grained

Also, there's a new configuration in config.yaml named `token_label_dim`, which specifies the dimension of the sentiment labels of each token after transformation.

To run our codes, use the following command

```
python test_improved.py --config='config.yaml'
```

2.2 Details

We try to use the sentiment labels of each word in the Stanford Sentiment Treebank datasets. We first transform the sentiment label of each word into dense vectors. After that we concatenate these dense vectors with word embeddings and feed them into the LSTM layer, shown in Figure 2.

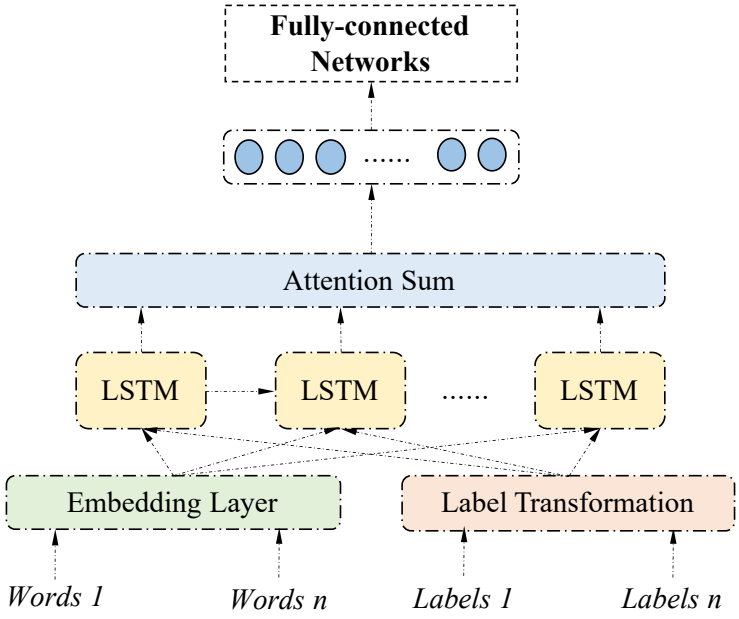

Figure 2: The Structure of Improved Model

Here we use a linear transformation layer with 128 hidden units to process the labels of each word, whcih are represented in one-hot vectors. We train our model on the training set with the batch size of 128. The model is trained through 30 epochs, and the one who performs best on the validation set will be saved.

The experimental results of our new models are shown in Table 3. We only perform our experiments on the basis of two best models in Table 2, i.e. models in experiment 5 and 6.

2.3 Discussion and Summary

From Table 3, we can observe that

Table 3: Experimental results of improved models

Number of Experiments	Dropout	Hidden Size	Layer	Embedding	Accuracy
Original Version					
5	0.2	256	1	No	0.371
6	0.2	256	1	Yes	0.447
Improved Version					
17	0.2	256	1	No	0.391
18	0.2	256	1	Yes	0.453
Joined Model Multi-tasking [2]	-	-	-	-	0.448
GRU-RNN-WORD2VEC[2] [1]	-	-	-	-	0.450
RNTN[3] [3]	-	-	-	-	0.457

- Our new models outperform the original models shown in Table 2 and get competitive results compared with Mu and Viswanath(2018) [1] and Socher et al.(2013) [3], indicating that the new information from the labels of each word is useful.
- Our improved model with pretrained embeddings can get around 2% improvements, while the one without pretrained embeddings can gain more than 6% improvements. These show the effectiveness of our new models.
- However, our new models only utilize the labels of each word, omitting the structural information in the datasets, which might be one of the future directions.
- Also, how to deal with the compound words which are out of the vocabulary of the pretrained embeddings is worth exploring carefully.

References

[1] Mu, J., and Viswanath, P. All-but-the-top: Simple and effective post-processing for word representations. In *6th International Conference on Learning Representations, ICLR 2018* (2018).

[2] Nejat, B., Carenini, G., and Ng, R. Exploring joint neural model for sentence level discourse parsing and sentiment analysis. In *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue* (2017), pp. 289–298.

[3] Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. Recursive deep models for semantic compositionality over a sentiment

[2]Rank 22-nd, from: https://paperswithcode.com/sota/sentiment-analysis-on-sst-5-fine-grained
[3]Rank 21-st, from: https://paperswithcode.com/sota/sentiment-analysis-on-sst-5-fine-grained

treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing* (2013), pp. 1631–1642.