Test Classification with CNN and RNN

Abstract—This report is aimed to present experiments with convolutional neural network (CNN) and recurrent neural network (RNN) trained on top of pre-trained word vectors for text classification tasks. A useful tool for deep learning—*Pytorch* was utilized to deal with vectors and matrix. For initialization, both static word embedding like word2vec, glove and fasttext and non-static random word embedding were taken into consideration. Besides, inspired by Yoon Kim, I have tried to use multi-channel CNN to help the word vector better adapt to data set to improve classification efficiency. Also, referring to Jason Brownlee's experiments, I have tried to pay attention to pad sequence into fixed length, and zero out the rest when dealing with our embedding. When it comes to experiments, I used Dev-test set. And parameters like embedding dimension and max length of sequences were adjusted. Finally, the best accuracy of validation set was achieved at 55.38 % for CNN and 59.72% for RNN.

I. Introduction

Text classification is a predictive modeling problem where we have some sequence of inputs over space or time and the task is to predict a category for the sequence. In this report, we focus on text classification of movie review of the Rotten Tomatoes, to label phrases on a scale of five values: negative, somewhat negative, neutral, somewhat positive and positive. Each movie review is a variable sequence of words and the sentiment of each movie review must be classified.

In the past few years, neural network based methods have obtained great progress on a variety of natural language processing tasks. Therefore, in this project, we tried to achieve text classification with deep learning model CNN and RNN.

When it comes to toolikit, we used *pytorch*. *Pytorch* is a deep learning framework, which can replace *numpy*. It not only inherits many advantages of *numpy*, but also supports GPUs computing. So it excels *numpy* in computing efficiency. As it can quickly build and train the deep neural network model, we used *pytorch* in this project.

II. Initialization

Before training the neural network models, we need to map each movie review into a real vector domain, a popular technique when working with text called word embedding. Here, both static and non-static word embedding were taken into consideration. Word2vec and glove were used to pre-train word vectors and they won't change when training the neural network, which means they are "static"; on the contrary, random embedding usually will be fine-tuned when training, so it is "non-static".

Before embedding, there are some tips we need to pay attention to so that we can train our neural networks smoothly later:

- Meaningless stopwords need to be eliminated first.
- We need to make sure that length of each sample(sequence) keep the same. Obviously, the sequence length (number of words) in each review varies, so we will constrain each review a fixed number of words, truncating long reviews and pad the shorter reviews with zero values. The fixed length can be the longest of all the samples, or it can be assigned manually.
- Inspired by Jason BrownLee[1] , we assume that most frequent words can offer further gains in performance. Therefore, we also tried to limit the total number of words that we are interested in modeling to the 5000 most frequent words, and zero out the rest.
- As embedding dim may influence the classification performance, we need to pre-train word vectors of different dim so that we can compare their performance later.
- Undoubtedly, there will be words in the validation set or test set that never appear in training set. We

will use zero vectors to present them or assign word vectors randomly to them. Besides, as *fasttext* tool can create vectors for out-of-dictionary words, I also tried to use this tool for word embedding.

A. *Word2Vec*

Word2Vec model uses neural network to train word vectors. While the CBOW algorithm predicts target word based on context words, the Skip-grams algorithm learns the representation of context words from the target word. Given a training word sequence $w_1, w_2, w_3 \ldots w_T$,we need to maximize the log probability $J(\theta) = \frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j\le c, j\ne 0}\log p(w_{t+j}|w_t)$, where the conditional probability can be calculated using the softmax function: $p(w_o|w_I) = \frac{\exp({u_{w_o}}^T v_{w_I})}{\sum_{w=1}^{W}\exp({u_w}^T v_{w_I})}$

In this way, when training to maximize the log probability, we can also get the word embedding.

Here, we used word2vec model of the genism package. And skip-gram algorithm was used to construct word2vec model. Finally, we can get vector of a certain word using model.wv[word].

B. *Glove*

As word2vec model has drawback that frequent words like "the", "and", "a" have an outsized effect on the representation learning, glove was introduced by researchers. Glove means global vectors for word representation. It combines overall statistics features and local context information(the sliding windows).

To achieve this, glove introduces co-occurrence probabilities matrix. As a result, the objective function becomes $J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \widetilde{w_j} + b_i + b_j - logX_{ij})^2$, where X_{ij} means frequency of word i co-occurring with word j. The ratio of X_{ij} and X_i can represent P_{ij}—probability of word j appearing in context of word i. And $w \in R^d$ represents a word embedding of dimension d, while $\widetilde{w} \in R^d$ represents a context word embedding of dimension d. b_i and b_j are two scalars bias items. And f represents the weighting function.

In this project, we used code of StanfordNLP[2] to train our corpus, and load the word vectors using genism package.

C. *Fasttext*

While word2vec can not map a vector of a word out of dictionary, fasttext has solved this problem using n-gram features. Model framework of fasttext is similar to CBOW of word2vec. The difference lies in the predicted objectives—while CBOW predicts the center word, fasttext directly predicts the category of the text. What's more, as bag of words model can not give information of word order, it introduces N-gram feature. It uses vectors to represent N-gram features to take local word order into account.

Here, we used fasttext model of genism package to train our corpus.

D. *Random Embedding*

Besides pre-train the word vectors, we can also randomly initialize the embedding. Using the torch.nn.embedding(m,n) function, we can achieve random embedding of m words of n dimensions. Later, we can fine-tune there vectors when training the neural network to make them batter adapt to our task.

III. Model

Construction of CNN and RNN by *pytorch* can be divided into three steps: Firstly, we need to preprocess and load our dataset; then we define our own neural networks; after that, we train our networks and compute loss and accuracy.

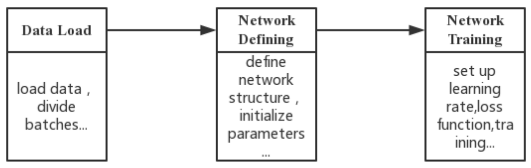
Data Load
load data , divide batches...
Network Defining
define network structure , initialize parameters ...
Network Training
set up learning rate,loss function,training...

Fig 1. Construction of NN by *Pytorch*

A. *Data Load*

Here, we constructed mapping of word to index and index to word, so we can get the word index list of a sentence, given our dictionary. Once we have an index list of a sentence, we can get vectors of it by using the *torch.nn.embedding* function.

Also, with the help of *torch.utils.data* package, we can depart our dataset to several batches. This is important later in the training section, as if we don't divide batches, we may face problems like memory error.

B. *CNN*

1. *Algorithm of Convolutional Neural Network*

Convolutional Neural network(CNN) utilizes layers with convolving filters that are applied to local features. It was originally invented for computer vision field. Later, CNN models have subsequently been shown to be effective for NLP tasks. Yoon Kim[3] trained a simple CNN with one layer of convolution on top of word vectors obtained from an unsupervised neural language model and achieve text classification. In this report, we will construct our CNN model referring work of Yoon Kim.

Here, we quote a picture in Yoon Kim's paper to explain CNN for text classification.

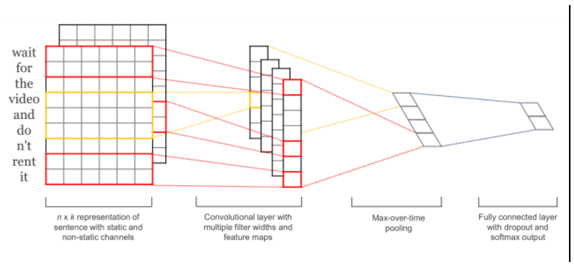
wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Fig 2. Model architecture with two channels [3]

In the first picture above, each word corresponds to one line as a word vector. Then, there is a convolutional layer with multiple filter widths and feature maps. Just as the figure shows, there is something different of this CNN model, compared to those in computer version. The width of convolution kernel is fixed as a dimension of word vector. After that is a max pooling layer. Finally, a fully connected layer with drop out is lied and we can get the output through softmax.

2. *Define our CNN*

With the aid of *Pytorch*, we can define our CNN model easily and quickly. Firstly, for initialization, we need to set up parameters like filter number, filter size and embedding dim. Here, we can use *nn.Conv2d* to define a series of filters and use *nn.ModuleList* to concatenate them to form our convolutional layer. And *nn.Linear* was used to define our fully-connected layer.

It's noticeable that we pay attention to **dropout**. Dropout means in the process of network training, the neural network cell is temporarily discarded from the network according to a certain probability. In this way, we've trained different networks and saved time. So it can avoid overfitting of neural network. Therefore, we use *nn.Dropout* to set up dropout mechanism in our model.

Next is the definition of forward propagation process. Functions encapsulated in *torch.nn.functional* like *relu* and *max_pool2d* are used to express the forward propagation process.

3. *Train our CNN*

When it comes to training of our CNN, we need to set up parameters like optimizer, loss function and learning rate. For optimizer, we have SGD, Momentum, RMSprop and Adam. Among them, Adam is the newest and advanced. So we chose Adam optimizer first and compare them later. Talking about loss function, we used cross-entropy loss function to compute difference between target labels and our prediction. For each epoch, training steps can be described as follows:

Tab 1. Training steps of CNN

Clear the optimizer's gradient :	optimizer.zero_grad()
Predict labels of training samples:	logit =net(feature)
Compute loss:	loss = F.cross_entropy(logit, target)
Compute gradients:	loss.backward()
Update parameters:	optimizer.step()

C. *RNN*

1. *Algorithm of Recurrent Neural Network*

Although CNN models perform well in many NLP tasks, there are some limitations with them. One limitation lies in that they can't model longer sequence information in a fixed filter size field of vision. On the other hand, adjustment of super parameter like filter size is also very cumbersome. By contrast, recurrent neural network(RNN) can better express context information. Compared with common RNN models, bi-directional LSTM can capture long and bidirectional "n-gram" information. So we mainly focus on bi-directional LSTM in this report.

The computation process of LSTM unit can be depicted as follows:

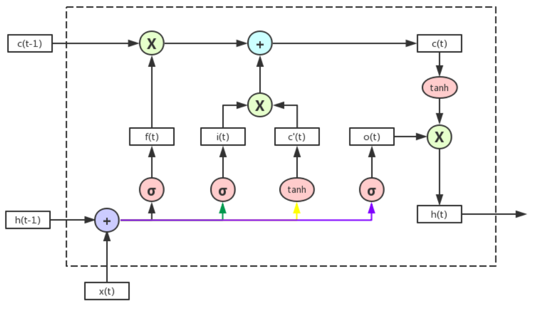

Fig 3. Loop unit structure of LSTM network

Tab 2. Computation process of LSTM unit

Calculate three gates and candidate state c'(t) by using the external state h(t-1) and the input at the present—x(t).
Update the memory cell c(t) by forget gate f(t) and input gate i(t)
Transmit state to external state h(t) by combining o(t) and c(t)

The formula can be simply described as:

$$c(t) = f(t) \odot c(t-1) + i(t) \odot c'(t)$$
$$h(t) = o(t) \odot \tanh(c(t))$$

2. *Define our RNN*

With help of *Pytorch*, it's also convenient to define a RNN model. Similar to steps to define a CNN model, we need to initialize parameters like embedding dim and layer number. But we don't need to define convolution kernel, instead we need to assign size of hidden layer units. *nn.LSTM* is provided to build a LSTM model directly. We can use the bidirectional attribute to decide whether the model is bidirectional or not.

Same to what we have done to our CNN model, we also add dropout layers to our RNN model to avoid overfitting. When it comes to the definition of propagation process, we first need to initialize the internal state *c(0)* and external state *h(0)*.Then, we just call *nn.lstm* to complete the forward propagation process.

3. *Train our RNN*

Similarly, we assign optimizer and loss function in this section. However, while training a LSTM model, I met a problem. Due to the intermediate variables in neural networks and the huge amount of intermediate parameters generated when using optimizer algorithm, a memory error occurs. To solve this problem, several tips can be taken:

- Decrease our batch size properly
- When we predict the labels, we can use "with torch.no_grad()" to save space
- Delete some intermediate parameters

IV. Experiment

The corpus is divided into Development set and Test set, go a step further, we divided the Development set into Training set and Dev-Test set. Hence, we do training on training set, adjusting parameters and choosing the best classifier base on the results from Dev-Test set.

There are many parameters influencing the performance of CNN and RNN. We'll talk about some of them in the following sections.

A. *Different embedding methods*

Here, with reference to work of Jason BrownLee[1], we temporarily set the embedding dim as 32, and fixed length at 10 and do tests with different embedding methods. It's noticeable that inspired by Yoon Kim's work, we experiment with two 'channels' of word vectors—one that is what we have pre-trained in Word2Vec, Glove and Fasttext, and the other is randomly initialized. This is similar to three channels of RGB in CNN of computer version.

The results are as follows:

Tab 3.Result of CNN

Embedding	Word2Vec	Glove	Fasttext	Random
Train_acc	54.48%	52.82%	53.27%	50.88%
Val_acc	54.28%	52.80%	52.99%	50.65%

Embedding	Word2Vec multichannel	Glove multichannel	Fasttext multichannel
Train_acc	55.94%	56.14%	55.71%
Val_acc	55.35%	55.26%	54.95%

Tab 4.Result of RNN(LSTM)

Embedding	Word2Vec	Glove	Fasttext	Random
Train_acc	55.45%	54.53%	56.33%	52.82%
Val_acc	55.39%	54.17%	55.17%	51.88%

B. *Different fixed length of sequence*

Given results above, we used Word2Vec for word embedding and do the next experiments. In this section, we pad word sequence into different length and compare the results. As the max length of a review is 52, which is far from the average, so we do not use the max length, because it will introduce more noise.

Tab 5.Result of CNN

Fixed Length	5	10	15	avg_len(7)
Train_acc	53.63%	54.48%	54.76%	54.23%
Val_acc	53.51%	54.28%	54.61%	53.88%

From the table we can see that long vectors, which mean more words, more information, can improve the performance of our CNN classifier.

C. *Different embedding dim*

Here, we use fixed length of 10 and embedding method of Word2Vec to do the several experiments and get following results. We refers to work of Jason BrownLee[1] and select 32 as the base embedding dim.

Tab 6.Result of CNN

Embedding dim	16	32	64	96
Train_acc	53.65%	54.48%	55.61%	55.67%
Val_acc	53.62%	54.28%	54.91%	55.07%

Tab 7.Result of RNN(LSTM)

Embedding dim	16	32	64	96
Train_acc	54.34%	55.45%	58.83%	59.50%
Val_acc	54.31%	55.39%	57.90%	58.08%

From the table we can see generally larger embedding dim help store more information and improve the classification accuracy.

D. *Different batch size*

Batch size will also influence the network optimization. Generally, the larger the batch size is, the smaller the variance of the random gradient is, the smaller the noise is, and the more stable the training is. Here, we tried batch size of 64,128,256 and 512 and the results are below.

Tab 8.Result of CNN

Batch size	64	128	256	512
Train_acc	56.11%	56.03%	56.10%	55.67%
Val_acc	55.23%	55.38%	55.35%	55.07%

Tab 9.Result of RNN

Batch size	64	128	256	512
Train_acc	62.64%	61.50%	60.57%	59.50%
Val_acc	59.72%	59.45%	58.86%	58.14%

When we have small batch sizes, the accuracy may be higher, but it's less stable. In this case, we often need to adjust learning rate at the same time to get better performance.

E. *Different Optimizer*

As there are adjustment mechanisms of learning rate packaged in optimizers provided by *Pytorch*, we discuss different optimizers instead of different learning rates. Here, taking as example, we tried SGD, Momentum, RMSprop and Adam in our experiments, with word2vec embedding, embedding dim 96, fixed length of 10 and batch size 512. We finally get following figures:

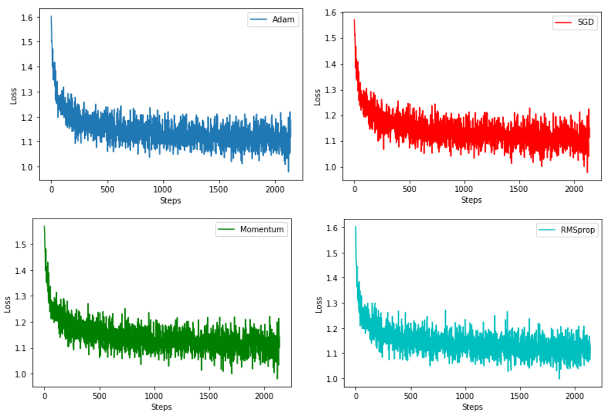

Fig 3. Loss of different optimizers training CNN

The figures show that these optimizers don't have obvious differences in convergence speed or performance. Maybe we need to continue to adjust some parameters to get a better and more stable result.

V. Conclusion

In this report, we used *Pytorch* to construct CNN and RNN to achieve text classification. We first load data of movie reviews of Stanford using *torch.utils.data* package. Then we defined and built network by functions and packages in *Pytorch*. In experiments, we have tried different parameters like embedding dim, text length an so on. Finally, the best accuracy was achieved at 59.72% by LSTM. As there are so many parameters influencing performance of network, in the future we can try more optimization methods to improve.

References

[1]https://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-python-keras/

[2] https://github.com/stanfordnlp/GloVe

[3] Kim, Y. . (2014). Convolutional neural networks for sentence classification. Eprint Arxiv.