FNText: A Fast Neural Model for Efficient Text Classification

Anonymous EMNLP submission

Abstract

In recent years, very deep neural models based convolutional neural networks (CNNs) have achieved remarkable results in natural language processing (NLP). However, the computational complexity also largely increases as the networks go deeper, which causes long training time. To raise the efficiency of calculation, this paper focus on shallow neural model and explores a fast neural text classification model FNText, which only contains 3 layers, without activation function and stacked time-consuming convolutional layers. Instead of enumerating a bag of bi-grams, we propose a novel method which utilizes average pooling operation along randomly initializing word vectors to obtain bi-gram features. These additional bi-gram features can further improve the performance of FNText. We improve the training speed by ignoring hyperparameters with zero-gradients. Experiments show that FNText can be trained on more than 300 million words in less than 10 minutes using a standard multicore CPU, and achieves competitive results on several large-scale datasets. Sometimes FNText is on par with very deep neural models. The implementation of FNText is freely available at https://github.com/Ra1nyHouse/FNText.

1 Introduction

Classifying the massive volume of text is an important task in natural language processing (NLP), whose applications include web search, information retrieval, sentiment classification. Recently, neural networks makes advances in NLP problems. The most widely used tool word2vec (Mikolov et al., 2013b,a) implement the very shallow neural models (continuous bag-of-words and skip-gram) for learning high-quality distributed vector representations, which capture syntactic and semantic information from words. Subsampling and hierarchical softmax tricks improve the quality of word vectors and training speed. CNN-kim (Kim, 2014) is builded on top of pre-trained word vectors for sentence-level classification tasks. To process large amount of documents, deeper and more complex models have been proposed, such as character-level CNNs (Zhang et al., 2015), a combination of char-CNN and Recurrent Neural Networks (RNNs) (Xiao and Cho, 2016), a deep ResNet-like neural network VDCNN (Conneau et al., 2017).

CNNs based on shared-weights architecture and translation invariance characteristics, are originally invented for computer vision to learn filters that in traditional algorithms were handcrafted. The based idea has been improved recently, in particular by stacking many layers of convolutions and pooling to learn a hierarchical representation (Szegedy et al.; He et al., 2016), which then feed to predict the category of the image. In array data of images, local groups of values are often highly correlated, thus the local motifs that are easily detected by CNNs. When it comes to texts, CNNs can learn the local semantic information over the stacked word vectors, because sequential words can be seen as the adjacent pixels. Each CNN is computed independently and is suited for parallel computation. By contrast, RNNs process words in sequential order, and "memorize" the sequence information into its hidden states. The last-step hidden state or the average of hidden states can be feed to a classifier for predicting the sequence label. This structure is parallel-processing unfriendly.

Both CNNs and RNNs take word order into account, without additional n-grams. But training the models always takes a long time. With regard to RNNs, the current hidden state is depend on the last-step hidden state, thus the structure is not suitable for parallel computing. For CNNs, the deep stacked convolutional layers cause too much com-

putation. These motivate us to pursue a fast and efficient model.

In this paper, we focus on shallow neural models and propose a simple model called FNText, which is without time-consuming convolutional layers and activation layers. We produce 2D array by concatenating the word vector sequence from a document as our model input. Then a feature-wise max pooling is applied over the concatenating word vectors to take the maximum values corresponding with all dimensions. Compared to traditional methods about averaging or suming the word vector sequence, feature-wise max pooling provides two main benefits: less computation during backpropagation period and more robust document features. To overcome the problem that basic FNText model can not learn word order information. We inventively use a average pooling operation over randomly initializing word vectors to obtain bi-gram features, which further improve the model performance. Moreover, we also give a bag of tricks to accelerate training model, such as ignoring the parameters in look-up tables not selected by feature-wise max pooling during backpropagation period and selecting 5% most frequent words as vocabulary. Extensive experiments on real-world datasets show that FNText works efficiently with less training time. On a corpus with 300 million words, FNText can be trained in less than 20 minutes using a standard multicore CPU. In contrast, char-CNN need 5 days and VDCNN need 7 hours.

2 Fast neural model for text classification (FNText)

Overview of FNText: Fig. 1 shows the whole model architecture of FNText. The first layer concatenates word vectors from 2 different look-up tables and produces 2-channel 2D arrays: one is directly used by next layer, the other is processed by a average pooling operation with window-size 2 to make bi-gram features. Then it is followed by a feature-wise max pooling layer, which choose the important feature on each dimension and produce the document features. Finally the document features are feed to the softmax layer to predict the document label. The key features of FNText are as follows.

- FNText only contains 3 layers, without any activation function and time-consuming convolutional layer, which is relatively easy to implement and train.
- FNText applies average pooling on concatenating word vectors to obtain bi-gram features with word order information. This method avoids enumerating tokens corresponding to bi-grams and complex hash trick (Weinberger et al., 2009) for query efficiency.
- We do not apply any nonlinear transform on word and bi-gram vectors, but directly use a feature-wise max pooling to capture the most important feature for each particular dimension associated with the semantic.

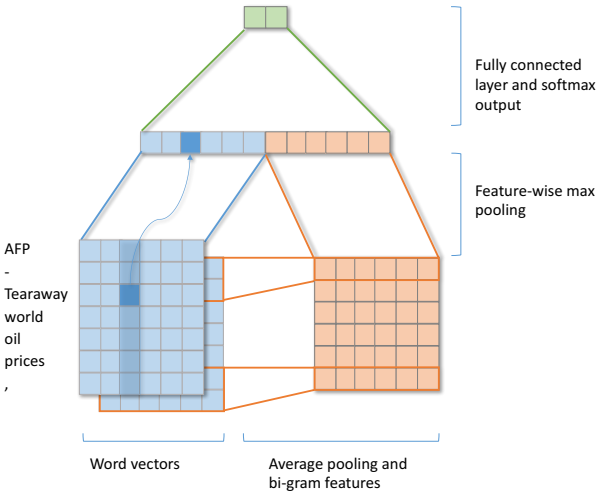

Figure 1: Model architecture of FNText.

2.1 Network architecture

Generating word and bi-gram vectors Suppose that the T-length text is denoted as word indice sequence $\{x_1, x_2, \cdots, x_T\}$, where $1 \leqslant x_i \leqslant n$ and n is the size of vocabulary. We randomly initialize two matrix $A, A^{bi} \in \mathbb{R}^{n \times k}$ as look-up tables over the words in the vocabulary. Then word vectors $F \in \mathbb{R}^{T \times k}$ can be constructed by

$$F = A_{x_1,*} \oplus A_{x_2,*} \oplus \cdots \oplus A_{x_T,*}. \quad (1)$$

The bi-gram vectors $G \in \mathbb{R}^{(T-1) \times k}$ is calculate on top of A^{bi} and we will discuss it detailedly in Section 2.2.

Feature-wise max pooling After obtaining model input, we then apply a feature-wise max

pooling over concatenating word and bi-gram vectors. For j-th dimension, the maximum value is selected by

$$f_j = max\{F_{*,j}\} \equiv F_{r_j,j} \equiv A_{x_{r_j},j} \equiv A_{s_j,j},$$
$$g_j = \max\{G_{*,j}\} \equiv G_{r_j^{bi},j} \equiv \frac{1}{2}(A^{bi}_{s_j^{bi},*} + A^{bi}_{s_j^{bi+},*}). \tag{2}$$

where r_j, s_j, r_j^{bi}, s_j^{bi} and s_j^{bi+} are the position indicators, which are used by learning algorithm to skip the parameters with zero gradients.

Predicting document label To deal with the prediction task, we choose softmax as a multiclass classifier for predicting the category of input text. The classifier contains several hyperparameters: $W \in \mathbb{R}^{m\times k}$, $W^{bi} \in \mathbb{R}^{m\times k}$ and $b \in \mathbb{R}^m$, where m is the number of predicted categories. The output of the classifier is calculated by

$$z = f \cdot W^\top + g \cdot (W^{bi})^\top + b. \tag{3}$$

The probability of category i is

$$p_i = \frac{\exp(z_i)}{\sum_{j=1}^m \exp(z_j)}. \tag{4}$$

2.2 Bi-gram features

Taking explicitly word order into account is often computationally very expensive. Instead, we can use a bag of bi-grams as additional features to capture some partial information about the local word order. For example, given a sentence

"AFP - Tearaway world oil prices, ..."

the bi-grams can be enumerated as

<"AFP", "-" >, <"-", "Tearaway" >, <"Tearaway", "world" >...

However, the enumerating tokens corresponding to bi-grams should be added into the vocabulary. Then the look-up table maybe very large, which causes poor query and computation efficiency.

To improve the efficiency of building bi-grams, we inventively apply a average pooling with window-size 2 over concatenating word vectors from a look-up table A^{bi}. For example, the representation of i-th bi-gram $< x_i, x_{i+1} >$ can be calculated by

$$G_{i,*} = \frac{1}{2}(A^{bi}_{x_i,*} + A^{bi}_{x_{i+1},*}). \tag{5}$$

Then concatenating bi-gram vectors G can be obtained by

$$G = G_{1,*} \oplus G_{1,*} \oplus \cdots \oplus G_{T-1,*}. \tag{6}$$

Our method for constructing bi-grams avoids enumerating and querying operations. Besides, the bi-grams not appeared in training dataset can also be computed by our method.

2.3 Learning algorithms

Given a document x, we minimize the negative log likelihood loss:

$$\min_\Theta \mathscr{L} = -\log p_y, \tag{7}$$

where Θ denotes model parameters, y is the text label. Then we use backpropagation algorithm to calculate desired partial derivatives. For simplicity, we define $q \in \mathbb{R}^m$ as indicator vector, where q_y takes on value 1, and 0 otherwise.

The normalization term of softmax is

$$\mathcal{Z} = \sum_{j=1}^m \exp(z_j). \tag{8}$$

Then the error δ of softmax layer can be calculated by

$$\delta = \nabla_z \mathscr{L} = \frac{1}{\mathcal{Z}} \exp(z) - q. \tag{9}$$

The gradients with respect to hyperparameters of FNText model can be computed as follows:

$$\begin{aligned}
\nabla_{b_i}\mathscr{L} &= \delta_i, \\
\nabla_{W_{i,j}}\mathscr{L} &= \delta_i \cdot \hat{f}_j, \\
\nabla_{W^{bi}_{i,j}}\mathscr{L} &= \delta_i \cdot \hat{g}_j, \\
\nabla_{A_{u,j}}\mathscr{L} &= \begin{cases} \delta^{bi} \cdot W_{*,j} & \text{if } u = s_j, \\ 0 & \text{otherwise,} \end{cases} \\
\nabla_{A^{bi}_{u,j}}\mathscr{L} &= \begin{cases} \delta^{bi} \cdot W^{bi}_{*,j} & \text{if } u = s_j^{bi} = s_j^{bi+}, \\ \frac{1}{2} \cdot \delta^{bi} \cdot W^{bi}_{*,j} & \text{if } u = s_j^{bi} \neq s_J^{bi+} \\ & \quad \text{or } u = s_j^{bi+} \neq s_j^{bi}, \\ 0 & \text{otherwise,} \end{cases}
\end{aligned} \tag{10}$$

where $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant k$ and $1 \leqslant u \leqslant n$.

We use the stochastic optimization algorithm Adam (Kingma and Ba, 2014) to train FNText. To accelerate the training speed, we give some tricks as follows:

- During the forward period of max-over-time pooling, the position indicators s_j, s_j^{bi} are stored.
- Set the gradients of look-up tables not pointed by the position indicators to 0 (see Eq. 10).

- Not only the hyperparameters with zero-gradients are ignored, but also the moments (also named 1-st and 2-nd moment vectors in Adam).

We also recommend Stochastic Gradient Descent (SGD) (Bottou, 2012) for train our models if the memory is limited, because SGD don't need additional space for saving gradients of the previous step. But SGD need more training time compared to the Adam.

3 Experiments

In this section, we conduct a comprehensive set of experiments that aim to answer four key questions: (1) Could FNText achieve competitive results on several large-scale datasets compared with traditional methods and deep neural models? (2) How does performance of FNText vary with model configurations, such as using bi-grams or not, limiting vocabulary size or not? (3) Does FNText have potential of transfer learning? (4) What is the computational complexity and memory usage of FNText?

3.1 Dataset overview

Dataset	Classes	#Train	#Test	#Vocabulary
AG	4	120K	7.6K	90K
Sogou	5	450K	60K	430K
Dbpedia	14	560K	70K	860K
Yelp P.	2	560K	38K	350K
Yelp F.	5	650K	50K	360K
Yah. A.	10	1.4M	60K	1.3M
Amz. F.	5	3M	650K	1.8M
Amz. P.	2	3.6M	400K	2.1M

Table 1: Statistics of the large-scale datasets.

To facilitate comparisons with previous results, we employ the same large-scale datasets and evaluation protocol of Zhang et al. (Zhang et al., 2015). AG and Sogou are news. Dbpedia is an ontology. Yahoo (abbreviated as 'Yah. A.') contains questions and answers from the 'Yahoo! Answers' website. Yelp and Amazon (abbreviated as 'Ama') are reviews where '.p' (polarity) in the names indicates that labels are positive or negative, and '.f' (full) indicates that labels are the number of stars. The statistics of the large-scale datasets are shown in Table. 3.1. Sogou consists of Chinese, and the others mainly consists of English, which are also with other languages in small proportions. Specially, Chinese words in Sogou has been transformed into Pinyin - a phonetic remanization of Chinese by Zhang et al. Thus the models for English can be applied to Sogou dataset without change.

For simplicity, there are not any complicated preprocessing steps or data augmentation. We just convert upper-case letters to lower-case letters, and add all tokens including punctuation and words into the vocabulary. We don't remove the stop-words or sub-sample the frequent words. If there are tokens not in the vocabulary during predicting period, we just ignore these tokens.

3.2 Parameter settings

Dataset	k	$epochs$	$batchs$
AG	400	10	500
Sogou	400	10	500
Dbpedia	400	10	1000
Yelp P.	600	5	1000
Yelp F.	600	3	1000
Yah. A.	600	3	2000
Amz. F.	800	2	3000
Amz. P.	800	2	3000

Table 2: Parameter settings.

We randomly initialize hyperparameters of FNText before training. Look-up tables A and A^{bi} are drawn from the uniform distribution $U(-0.01, 0.01)$. Weights $\{W, W^{bi}, b\}$ in softmax layer are drawn from $U(-\sqrt{\frac{1}{2k}}, \sqrt{\frac{1}{2k}})$. There are three parameters k, $epochs$ and $batchs$ may affect the training process. In our experiments, we randomly select 10% of the training dataset as the validation dataset and search the suitable parameters on validation dataset, where k is from $\{200, 400, 600, 800\}$, $epoch$ is from $\{1, 2, 3, 4, 5, 10\}$ and $batchs$ is from $\{500, 1000, 2000, 3000\}$. Table. 3.2 lists the parameters used in our experiments.

In addition, we give a default configuration of FNText model: for small dataset (less than 500K documents), setting $k = 400$, $epoch = 10$ and $batchs = 500$; for large dataset, setting $k = 800$, $epoch = 2$ and $batchs = 3000$. This group of parameters lead to a good enough result in most cases.

3.3 Accuracy results

We evaluate several configurations of our model, using bi-grams or not, limiting vocabulary size or not. Table 3.3 lists all the test accuracies, where 'bi-gram' stands for utilizing bi-grams, '$(n\%)$' denotes vocabulary size is limited to n percent and

'(full)' means using all words in vocabulary. We also report results with respect to other models for compare:

- Bag-of-words use the counts of each word in vocabulary as the document feature.
- Bag-of-ngrams count up the number of n-grams instead of words. The feature values are computed the same way as in bag-of-words.
- Plain LSTM is the common RNN architecture used in NLP.
- Plain CNN produces the representation of document by applying CNNs over word vectors.
- Char-CNN (Zhang et al., 2015) treats text as a kind of raw signal at character level, and applies CNNs to learn document representation.
- Char-CRNN (Xiao and Cho, 2016) integrates RNNs and CNNs to efficiently encode character-level inputs.
- VDCNN (Conneau et al., 2017) is a deep ResNet-like neural network, which uses up to 29 convolutional layers for text classification.
- Fasttext (Grave et al., 2017) uses bag of tricks for training a simple but efficient shallow neural networks for txt classification.

To facilitate analysis, we present relative errors in Fig. 2. Relative errors are computed by taking the difference between errors on comparison model and FNText model, then divided by the comparison model error. The specific calculation formula is as follows:

$$relative\ error = \frac{Acc. - \hat{Acc.}}{1 - \hat{Acc.}}, \qquad (11)$$

where $Acc.$ denotes our FNText's accuracy (with bi-grams and full vocabulary), $\hat{Acc.}$ denotes the comparison model's accuracy. A positive relative error means our model is better than the comparison model, otherwise a negative relative error denotes our model is worse. Our experiments show that there is not a single model that can work very well for all kinds of datasets. There may have some factors that all play a role in deciding which model is best for a specific problem.

Using vs. not using n-grams Using n-grams can further improve performance in most cases, especially helpful for shallow neural models. For the conventional models based word frequency statistics, experiments results show that variant with n-gram information is better than the original in most cases. For example, bag-of-ngrams model achieves higher accuracy on AG, Sougou, Dbpedia, Yelp P. and Ama. P. datasets, gets lower accuracy on Yelp F., Yah. A. and Ama. F. datasets. When it comes to neural models, the variant combined n-grams as additional features obtains better performance on all large-scale datasets. Fasttext using bi-gram information improves the performance by 1-4% compared with the original. Our FNText model also works better with bi-grams (e.g. improving the performance by 0.5% for AG, 3.5% for Amz. F.). The model with bi-grams overcomes the problem that the basic model is invariant to word order, and captures more syntactic and semantic information. That's why the model with n-grams always perform better.

Note that our method for learning n-grams is different from the conventional method. Bag-of-ngrams model are constructed by selecting fixed quantity (e.g. 500000) of the most frequent n-grams from the training subset. Fasttext do the same way as in the bag-of-ngrams model, while the number of n-grams may be larger. To maintain a fast and memory efficient mapping of n-grams, fasttext utilizes the hashing trick (Weinberger et al., 2009). While problem do exists with this conventional approach as it can not process the n-grams not in the prepared vocabulary. Our method solve this problem by utilizing average pooling to produce n-gram features in real-time, rather than preparing in advance. It also offers lower memory requirement compared with enumerating n-grams.

Limiting vs. not limiting vocabulary Although limiting vocabulary can not accelerate the training process, it can save memory. Limiting vocabulary is a very common trick for text classification. Bag-of-words reported in (Zhang et al., 2015) only selects 50000 most frequent words from training subset, and bag-of-ngrams limits the vocabulary size to 500000. Limiting vocabulary has little performance loss because the incomplete vocabulary covers the majority of the training texts. For example, the complete vocabulary of the AG training dataset contains 90K words, but

Model	Deep	AG	Sogou	Dbpedia	Yelp P.	Yelp F.	Yah. A.	Amz. F.	Amz. P.
bag-of-words (Zhang et al., 2015)		88.8	92.9	96.6	92.2	58.0	68.9	54.6	90.4
bag-of-ngrams (Zhang et al., 2015)		92.0	**97.1**	98.6	95.6	56.3	68.5	54.3	92.0
plain LSTM (Zhang et al., 2015)		86.1	95.2	98.6	94.7	58.2	70.8	59.4	93.9
plain CNN (Zhang et al., 2015)		89.1	95.1	98.2	94.5	58.6	70.0	56.3	94.2
char-CNN (Zhang et al., 2015)	√	87.2	95.1	98.3	94.7	62.0	71.2	59.5	94.5
char-CRNN (Xiao and Cho, 2016)	√	91.4	95.2	98.6	94.5	61.8	71.7	59.2	94.1
VDCNN (Conneau et al., 2017)	√	91.3	96.8	**98.7**	**95.7**	**64.7**	**73.4**	**63.0**	**95.7**
fasttext (Grave et al., 2017)		91.5	93.9	98.1	93.8	60.4	72.0	55.8	91.2
fasttext, bi-gram (Grave et al., 2017)		**92.5**	96.8	98.6	**95.7**	63.9	72.3	60.2	94.6
FNText(1%)		88.4	95.1	98.0	93.5	60.3	71.9	55.8	91.5
FNText(5%)		91.4	95.4	98.3	94.1	60.9	72.6	55.9	91.6
FNText(10%)		92.1	95.4	98.3	94.1	60.9	72.7	55.9	91.6
FNText(full)		**92.5**	**95.4**	**98.3**	**94.1**	**60.9**	**72.7**	**55.9**	**91.6**
FNText, bi-gram(1%)		88.6	97.0	98.5	95.4	63.7	72.4	59.3	94.0
FNText, bi-gram(5%)		91.9	97.1	98.7	95.8	64.3	73.0	59.4	94.1
FNText, bi-gram(10%)		92.7	97.1	98.7	95.8	64.3	73.1	59.5	94.1
FNText, bi-gram(full)		**93.1**	**97.1**	**98.7**	**95.8**	**64.3**	**73.1**	**59.5**	**94.1**

Table 3: Test accuracy [%] of the large-scale datasets.

a vocabulary including 9K (10%) most frequent words covers about 93% of the texts. Similarly, 22K (5%) words covers about 99% of the texts for Sougou training dataset, and 65K (5%) words covers about 98% of the texts for Yah. A. training dataset. Limiting vocabulary can not reduce the scale of training dataset, thus it can not speed up the training. While, if we use smaller vocabulary, less parameters to represent words are needed. In other words, limiting vocabulary can save memory.

In our experiments, we explore FNText model with different size of vocabulary. Table 3.3 shows that limiting vocabulary to 5% or more achieves a very similar performance compared to the model with complete vocabulary in most cases (with one exception, that is AG dataset, which has a very small vocabulary). For example, using FNText with 5% vocabulary to classify Amz. F. dataset achieves 59.4% accuracy, while using full vocabulary just increases the accuracy to 59.5%. Using full vocabulary help little on the performance. The 0.1% difference in accuracy can be explained by random factors. One other thing to note is that too small vocabulary may reduce the accuracy. On all datasets, FNText with 1% vocabulary get the worse result. That is because some words playing a role in classification may not be included in the incomplete vocabulary.

There is a helpful experience for deciding the vocabulary size, that is if you have enough memory, you can use complete vocabulary. Otherwise limiting vocabulary size to 5% may lead to a good enough result in most cases. Or you can select a validation dataset looking for suitable vocabulary size.

FNText vs. shallow models FNText with bi-grams is better than the models based word frequency statistics. Compared to the shallow neural models, FNText with bi-grams gets higher accuracy on most datasets (one exception is on the Amz. datasets). Bag-of-ngrams achieves competitive results on Sougou, Dbpedia, Yelp P., Amz. P. datasets, although it's a very simple model. Our basic FNText without bi-grams gets lower accuracies on these datasets compared to bag-of-ngrams. For example, bag-of-ngrams gets 97.1% accuracy on Sogou dataset, and FNText gets 95.4%, decreasing by 1.7%. However, if we integrate bi-gram features into FNText, we can further improve the accuracy, which is greater than or equal to the accuracy of bag-of-ngrams. For Sogou dataset, accuracy improves to 97.1%, which is equal to the result of bag-of-ngrams. For Yelp. F. dataset, FNText with bi-grams achieves 64.3% accuracy, improving the performance by 8% compared to bag-of-ngrams. Plots comparing the models based word frequency statistics (Figs. 2 (a)(b)) also show that FNText with bi-grams works better.

When it comes to the shallow neural models, basic FNText without bi-grams is on par with plain CNN and plainLSTM models as processing AG and Sougou datasets. As the dataset grows larger, the difference between basic FNText and plain neural models is getting smaller. For the large Amz. datasets, the shallow neural models get higher accuracies (on Amz. P. dataset, plain CNN achieves 94.2%, plain LSTM achieves 93.9% and basic FNText achieves 91.6%). Even so, the basic FNText works better than another neural model

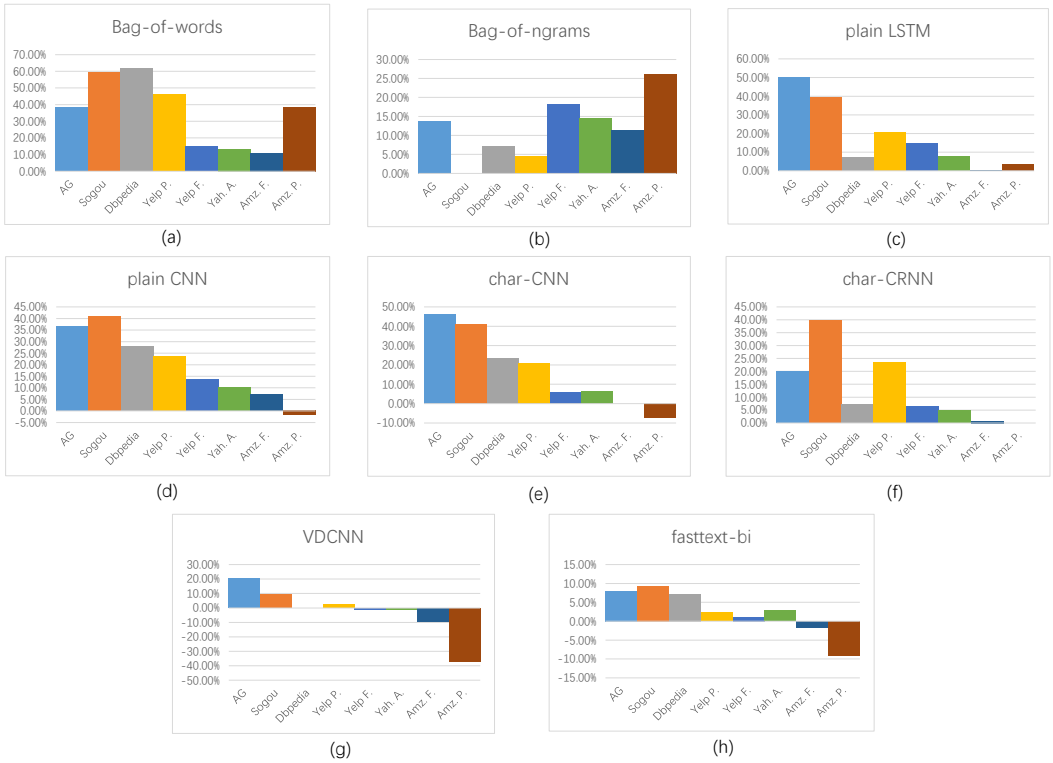

Figure 2: Relative errors with comparison models. The value greater than zero means our FNText works better, otherwise the comparison model performs better. (a) Compared to bag-of-words. (b) Compared to bag-of-ngrams. (c) Compared to plain LSTM. (d) Compared to plain CNN. (e) Compared to char-CNN. (f) Compared to char-CRNN. (g) Compared to VDCNN. (h) Compared to fasttext with bi-grams.

fasttext on all datasets. After learning bi-gram features, the improving FNText achieves higher accuracy compared to plain CNNand plain LSTM (see Figs. 2(c)(d)) except Amz. P. dataset. Compared to another neural model fasttext, which is also with bi-grams, FNText achieves better performance in most cases (see Fig. 2(h)).

FNText vs. deep models FNText with bi-grams is on par with very deep neural models as processing small dataset. For large datasets, FNText with bi-grams also achieves competitive results. Compared to plain shallow neural models like LSTM, CNN, very deep neural models including char-CNN, char-CRNN, VDCNN achieve significant improvement as processing large datasets. While novel shallow neural models with bi-grams such as fasttext and our FNText also achieves competitive results. Specifically, FNText works better than char-CNN and char-CRNN except Amz. P. dataset (see Figs. 2(e)(f)). Compared to VDCNN (see Fig. 2(g)), FNText get higher accuracy as processing AG, Sogou and Yelp P. datasets. For larger datasets such as Amz. datasets, FNText achieves lower accuracy, which is still competitive (difference is between 0.1%-3.5%). You should note that why deep models don't utilize n-grams to further improve performance, it's because the convolution operation with stride greater than 1 actually encodes the word order information. Using n-grams as additional features can't learn more information and the CNN structure is not suitable for receiving n-grams as input.

3.4 Transfer learning

In this section, we explore our FNText for transfer learning applications. We evaluate our method on Yelp P. and Amz. P. datasets. Both of them are reviews and mainly consists of English. Table 3.4 lists the accuracies.

Source vs. Target	Acc.
Yelp P. vs. Yelp P.	95.8
Amz. P. vs. Yelp P.	92.7
Amz. P. & Yelp P. vs. Yelp P.	**96.0**
Amz. P. vs. Amz. P.	94.1
Yelp P. vs. Amz. P.	88.4
Yelp P. & Amz. P. vs. Amz. P.	**94.1**

Table 4: Transfer results on Yelp and Amz. datasets.

3.5 Computational time evaluation

Given N documents with T average length, the forward computational complexity of FNText without bi-grams is $\mathcal{O}(NTk + Nkm)$, dominated by the computation of the max-over-time pooling and fully connected layer. The backward computational complexity is $\mathcal{O}(NVk + Nkm)$ as utilizing conventional optimization algorithm. But if we use the trick described in Section 2.3 to speed up the training, the computational complexity becomes $\mathcal{O}(Nk + Nkm)$, less than the former because the hyperparameters with zero gradients are not updated. When adding bi-grams into FNText, the forward computational complexity is $\mathcal{O}(2NTk + 2Nkm)$ and the backward computational complexity becomes $\mathcal{O}(2Nk + 2Nkm)$ because of adding novel average pooling operation and updating extra hyperparameters for representing bi-grams.

In Table 3.5, we list the training time for a single epoch on the large-scale training datasets. Char-CNN, VDCNN and fasttext are also reported for comparison. Note that both char-CNN and VDCNN are trained on a NVIDIA Tesla K40 GPU, while our FNText and fasttext are trained on a CPU using 20 threads. We observe that FNText with bi-grams cost about $2\times$ time than the basic model without bi-grams. That is because FNText with bi-grams have about $2\times$ hyperparameters, and another average pooling operation for learning bi-grams consumes more time. Compared to fasttext, our FNText models are slower than it. Fasttext takes less than 10 seconds on all datasets, while our FNText models may consume several minutes. The main reason is FNText models contain larger hyperparameters, especially the large look-up tables. This setting leads to a higher accuracy but lower training time. Compared to the very deep models, our FNText models have a great speed-up. Specifically, FNText with bi-grams has a $1800\times$ speed-up compared to big char-CNN on small AG dataset, and it goes down to $900\times$ on large Amz. P. dataset. VDCNN run faster than char-CNN. Our FNText with bi-grams still achieves $53 - 510\times$ speed-up ($53\times$ on Amz. P. dataset, $510\times$ on AG dataset). In general, FNText models run much faster than very deep neural models with little performance loss. While compared to fasttext, FNText models run slower but getting higher accuracy.

3.6 Memory usage

Table 3.5 reports the number of hyperparameters and memory usage about FNText with complete vocabulary. The number of hyperparameters is proportional to the size of vocabulary, and FNText with bi-grams has $2\times$ hyperparameters compared to basic FNText without bi-grams. Memory usage is decided by the number of hyperparameters and the size of training dataset. As processing very large dataset, there should be additional memory for saving the training dataset.

Note that there are several methods to reduce the memory usage. The first one is limiting vocabulary size. Section 3.3 discusses it in detail. The second one is using SGD instead of Adam. Because SGD don't need additional space for saving gradients of the previous step. You can refer Section 2.3 for detail. In practice, as processing Amz. P. dataset, FNText with bi-grams need 47.7G memory. While limiting vocabulary to 5%, the memory requirement decreases to 17.3G. As we use SGD instead of Adam, the memory requirement decrease to 16.3G from 47.7G. If adopt both two methods, FNText with bi-grams only needs 6.2G memory, which can be provided by most personal computers.

4 Related work

There is another shallow neural model named fasttext (Grave et al., 2017) for text classification, which also aims to accelerates training speed for simple text classification tasks. The main difference between them is that fasttext averages the word vectors as the text representation, while our FNText uses a feature-wise max pooling to construct the text feature. During backpropagation period, fasttext need update all word vectors with respect to the document, while our FNText only update a few parameters selected by feature-wise max pooling. Thus we could set a large word vector dimension (FNText always sets 800 vs. fasttext always sets 10) to obtain robust features. That is why FNText always on top of the fasttext. Moreover, we propose a novel method described in next section to learn n-gram features, which is different from the conventional enumerating method used by fasttext. It's easily to be implemented by average pooling and not need complex hash trick to promote query efficiency.

Dataset	char-CNN small	char-CNN big	VDCNN depth=9	VDCNN depth=17	VDCNN depth=29	fasttext bi-gram	FNText	FNText bi-gram
AG	1h	3h	24m	37m	51m	1s	3s	6s
Sogou	-	-	25m	41m	56m	7s	19s	36s
Dbpedia	2h	5h	27m	44m	1h	2s	13s	32s
Yelp P.	-	-	28m	43m	1h9m	3s	16s	46s
Yelp F.	-	-	29m	45m	1h12m	4s	25s	1m
Yah. A.	8h	1d	1h	1h33m	2h	5s	1m	2m
Amz. F.	2d	5d	2h45m	4h20m	7h	9s	2m	5m
Amz. P.	2d	5d	2h45m	4h25m	7h	10s	3m	8m

Table 5: Training time for a single epoch.

Dataset	FNText #hyperparam	FNText #memory	FNText,bi-gram #hyperparam	FNText, bi-gram #memory
AG	36M	0.6G	72M	1.1G
Sogou	172M	4.9G	344M	7.6G
Dbpedia	344M	5.5G	688M	10.5G
Yelp P.	210M	3.7G	420M	6.7G
Yelp F.	216M	4.1G	432M	7.4G
Yah. A.	780M	12.9G	1560M	24.6G
Amz. F.	1440M	20.9G	2880M	41.3G
Amz. P.	1680M	24.6G	3360M	47.7G

Table 6: Number of hyperparameters and usage of memory during the training.

5 Conclusion

In this paper, we propose a fast shallow neural model for text classification. We give a bag of tricks to improve the prediction performance and training efficiency, such as applying average pooling with window-size 2 over concatenating word vectors to obtain bi-grams, using feature-wise max pooling to produce text features, and ignoring the parameters with zero gradients during updating period. We have demonstrated the effectiveness of our method by applying our model to large-scale text classification datasets.

References

Léon Bottou. 2012. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer.

Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Lecun. 2017. Very deep convolutional networks for text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, volume 1, pages 1107–1116.

E Grave, T Mikolov, A Joulin, and P Bojanowski. 2017. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 427–431.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013b. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM.

Yijun Xiao and Kyunghyun Cho. 2016. Efficient character-level document classification by combining convolution and recurrent layers. *arXiv preprint arXiv:1602.00367*.

Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657.