FNText: A Fast Neural Model for Efficient Text Classification

Anonymous EMNLP submission

Abstract

In recent years, deep convolutional neural networks have achieved remarkable results in natural language processing (NLP). However, the computational complexity largely increases as the networks go deeper, which causes long training time. To improve the computing efficiency, we focus on shallow neural models and propose a fast neural text classification model FNText, which only contains 3 layers and do not use stacked convolutional layers. FNText employs a feature-wise max pooling operation to obtain the text features, and can integrate additional bi-gram features to improve the performance. Instead of enumerating a bag of bi-grams, FNText takes an average pooling over randomly initialized word vectors to obtain bi-gram features. We also improve the training speed by ignoring parameters with zero-gradients. Experiments show that FNText can be trained on more than 300 million words in less than 20 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/RalnyHouse/FNText.

1 Introduction

Classifying the massive volume of text is an important task in natural language processing (NLP) domains, including web search, information retrieval, sentiment classification. Recently, a trend in the NLP literature is the utilization of neural networks (Bengio et al., 2003; Collobert et al., 2011). The most used tool word2vec (Mikolov et al., 2013b,a) implements two shallow neural models (e.g., continuous bag-of-words and skip-gram) for learning distributed word representations, which are able to capture syntactic and semantic information. Subsampling and hierarchical softmax tricks improve the quality of word vectors and training efficiency. 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 are originally invented to learn filters that can capture invariant features of images based on weight-shared structures. When it comes to texts, CNNs can capture the local semantic information over the adjacent word vectors (Kim, 2014). Each filter is computed independently and thus is well-suited for parallel computation. By contrast, RNNs process sequential words via a recurrent structure, and "memorize" the sequential information into its hidden states. The last-step hidden state or the average of hidden states can be fed to a classifier for label prediction. Both CNNs and RNNs can take word order into account without additional n-grams. However, training the networks always takes a long time. For RNNs, the recurrent structure is not suitable for parallel computing, since the current hidden state depends on the previous one. Although the convolution operator is parallel-friendly, recent models primarily construct a deep architecture with stacked convolutional layers (Hu et al., 2014; dos Santos and Gatti, 2014). These models achieve satisfying prediction performance at the cost of more computation.

In this paper, we focus on shallow neural models and propose a simple architecture called FNText, which do not employ time-consuming convolutional layers and activation functions. Specifically, FNText first concatenates the word vectors of a document into a word-semantics matrix as

the model input (suppose that each dimension of the word vector associates with a semantic feature). Then feature-wise max pooling is applied over the concatenated word vectors to take the maximum values of each dimension. Compared to traditional methods that average or sum over the word vectors, feature-wise max pooling provides two main benefits: less computation during back-propagation and more robust document features. To capture the word order, FNText integrates additional bi-gram features. The features are obtained by averaging over corresponding word vectors, instead of enumerating a bag of bi-grams. Besides, we also provide several tricks to accelerate training of the model, such as ignoring the parameters that are not selected by feature-wise max pooling and selecting 5% most frequent words as the vocabulary. Extensive experiments on real-world datasets show that FNText requires less training time to achieve a satisfying result. On a corpus of 300 million words, FNText can be trained in less than 20 minutes using a standard multicore CPU. In contrast, char-CNN and VDCNN need 5 days and 7 hours, respectively.

2 FNText Architecture

Overview: Fig. 1 shows the whole architecture of FNText. The first layer concatenates word vectors from two different look-up tables and produces two input matrices: one is directly used by the next layer, the other is processed to obtain the bi-gram features via average pooling. Then a feature-wise max pooling layer is employed to generate document features, which takes the largest value (i.e., the most important semantic feature) of each dimension. Finally the document features are fed to a softmax layer for label prediction. The key features of FNText are summarized as follows.

- FNText is a shallow model, which is relatively easy to implement and train.
- FNText employs average pooling to obtain bi-gram features, which avoids enumerating a bag of bi-grams and complex hash trick (Weinberger et al., 2009) for query efficiency.
- FNText do not apply any nonlinear transformation on hidden features, but use feature-wise max pooling to capture the most important semantic feature of each dimension.

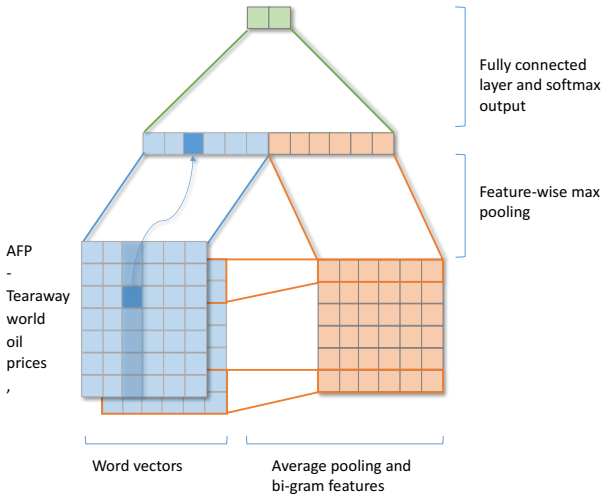

Figure 1: FNText architecture.

2.1 Network architecture

Generating word and bi-gram matrices We start by introducing some basic notations. We denote the vocabulary size by n. For a given document, the word sequence is represented as $\{x_1, x_2, \cdots, x_T\}$, where T is the number of words and $1 \leqslant x_i \leqslant n$. We randomly initialize two matrices $A, A^{bi} \in \mathbb{R}^{n \times k}$ as look-up tables over the words in the vocabulary. Then the word vector matrix $F \in \mathbb{R}^{T \times k}$ can be constructed as follows:

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

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

Feature-wise max pooling After obtaining the input matrices, we employ a feature-wise max pooling layer to generate the document features, which chooses the most important semantic feature of each dimension. For j-th dimension, the maximum value is selected by

$$\begin{aligned} f_j &= \max\{F_{*,j}\} \equiv A_{s_j,j}, \\ g_j &= \max\{G_{*,j}\} \equiv \frac{1}{2}(A^{bi}_{s^{bi}_j,*} + A^{bi}_{s^{bi+}_j,*}), \end{aligned} \quad (2)$$

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

Predicting the document label To predict the category of the document, we employ a softmax layer for multi-class classification. The classifier has several parameters: $W \in \mathbb{R}^{m \times k}$, $W^{bi} \in \mathbb{R}^{m \times k}$

and $b \in \mathbb{R}^m$, where m is the number of predicted categories. The output is computed as follows:

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

Finally, 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

It is often computationally expensive to take word order into account (e.g., the recurrent layer of RNNs or the stacked convolutional layers of CNNs). Instead, we can use a bag of bi-grams as additional features to capture the local semantic information. For instance, given a sentence

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

the bi-grams can be enumerated as:

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

However, enumerating all combinations of bi-grams generates a very large look-up table, which causes poor query performance. To improve the efficiency of building bi-grams, we use an average pooling operation over concatenated word vectors, with a window size of 2. Specifically, for the i-th bi-gram $< x_i, x_{i+1} >$, its representation can be calculated by:

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

Then the bi-gram matrix G can be obtained by concatenating these representations:

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

2.3 Learning algorithms

For a given document, 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. We use the back-propagation algorithm (Plaut et al., 1986) to calculate the partial derivatives. For simplicity, we define $q \in \mathbb{R}^m$ as the 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 w.r.t parameters of FNText can be computed as follows:

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

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 feature-wise max pooling, the position indicators s_j, s^{bi}_j are stored.
- The gradients of look-up tables not pointed by the position indicators are set to zero (see Eq. 10).
- Not only the parameters 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 training if the memory is limited. SGD don't require additional space for saving gradients of the previous step, but needs more training time compared to Adam.

3 Experiments

In this section, we conduct a comprehensive set of experiments that aim to answer four key questions: (1) Can 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 the vocabulary size, using bi-grams or not? (3) Does FNText have potential of

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 several large-scale datasets.

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 on different datasets.

transfer learning? (4) What is the computational complexity and memory usage of FNText?

3.1 Dataset overview

For a fair comparison, we use the same datasets and evaluation protocol (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) indicates that labels are positive or negative, and '.f' (full) indicates that labels are the number of stars. Table 3.1 summarizes the statistics of datasets.

For simplicity, we do not employ 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. If new tokens appear during inference, we just ignore them.

3.2 Parameter settings

We randomly initialize the parameters 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 $U(-\sqrt{\frac{1}{2k}}, \sqrt{\frac{1}{2k}})$. We randomly select 10% of the training set as the validation set and tune hyperparameters on it. There are three hyperparameters k, $epochs$ and $batchs$ that may affect the training process. We test the word vector dimension k of [200, 400, 600, 800], $epochs$ of [1, 2, 3, 4, 5, 10] and $batchs$ of [500, 1000, 2000, 3000]. Table 3.2 lists the parameters used in our experiments. In addition, a default configuration is given for some experiments: for small datasets (less than 500K documents), we set $k = 400$, $epoch = 10$ and $batchs = 500$; for large datasets, we set $k = 800$, $epoch = 2$ and $batchs = 3000$. This group of parameters leads to good results in most cases.

3.3 Accuracy results

We evaluate several configurations of FNText, using bi-grams or not, limiting the vocabulary size or not. We also report results of several baselines:

- Bag-of-words uses the counts of each word in the vocabulary as the document feature.
- Bag-of-ngrams counts up the number of n-grams instead 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) considers the document as a raw signal at a character level, and applies CNNs to learn document representations.
- 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 29 convolutional layers for text classification.
- Fasttext (Grave et al., 2017) employs several tricks to train a simple but efficient shallow neural model for text classification.

Table 3.3 lists all the test accuracies of FNText and compared baselines, where 'bi-gram' means the utilization of bi-grams, '(full)' means using all words in vocabulary, and '(n%)' means that the vocabulary size is limited to n percent. Besides, to facilitate analysis, we plot the relative errors between FNText and compared baselines in Fig. 2. The relative error is computed as follows:

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

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 [%] on several large-scale datasets.

where $Acc.$ is the accuracy of our FNText (with bi-grams and using a full vocabulary) and $\hat{Acc.}$ is the accuracy for a given baseline. A positive relative error means our model is better than the baseline. We can observe that there is no one model that can outperform others on all kinds of datasets. In practice, several factors may play a role in deciding which model is well-suited for a specific problem.

Using vs. not using n-grams For shallow neural models, using n-grams can improve performance in most cases. Bag-of-ngrams outperforms conventional bag-of-words in most cases, verifying the effectiveness of n-gram information. For example, bag-of-ngrams achieves higher accuracies on AG, Sougou, Dbpedia, Yelp P. and Ama. P. datasets, while gets lower accuracies on Yelp F., Yah. A. and Ama. F. datasets. When it comes to neural models, the variants that integrate n-grams as additional features can achieve better performance on all large-scale datasets. Fasttext using bi-gram information improves the performance by 1-4% compared to the original. Our FNText also performs better with bi-grams (e.g. improving the performance by 0.5% and 3.5% on AG and Amz. F., respectively). Integrating bi-grams enables the basic models to take into account the word order and then capture more syntactic and semantic information.

Note that our method for learning n-grams is different from the conventional ones. Bag-of-ngrams selects fixed quantity (e.g. 500000) of the most frequent n-grams from the training set. Fasttext utilizes more n-grams via the hashing trick, which maintains a fast and efficient memory mapping of n-grams (Weinberger et al., 2009). However, the enumerating method cannot process new n-grams that are not in the prepared vocabulary. FNText solves this problem by utilizing average pooling to produce n-gram features in real-time, rather than prepare n-grams in advance. It also requires less memory space compared to enumerating method.

Limiting vs. not limiting vocabulary Although limiting the vocabulary size cannot accelerate training of the model, it can save memory space, which is a common trick for text classification. In (Zhang et al., 2015), bag-of-words only selects 50000 most frequent words from the training set, and bag-of-ngrams limits the vocabulary size to 500000. Limiting vocabulary has little performance loss, since the incomplete vocabulary covers the majority of the texts. For example, the complete vocabulary of the AG dataset contains 90K words, but a vocabulary of 9K (10%) most frequent words covers about 93% of the texts. Similarly, 22K (5%) words covers about 99% of the texts for Sougou dataset, and 65K (5%) words covers about 98% of the texts for Yah. A. dataset. Therefore, limiting the vocabulary size do not reduce the scale of datasets, thus it cannot speed up the training. However, if we use a smaller vocabulary, less parameters are required to represent words, which can save memory.

In our experiments, we explore FNText with different sizes of vocabulary. Table 3.3 shows that limiting vocabulary to 5% or more achieves similar performance compared to that with a complete vocabulary in most cases (one exception is that on

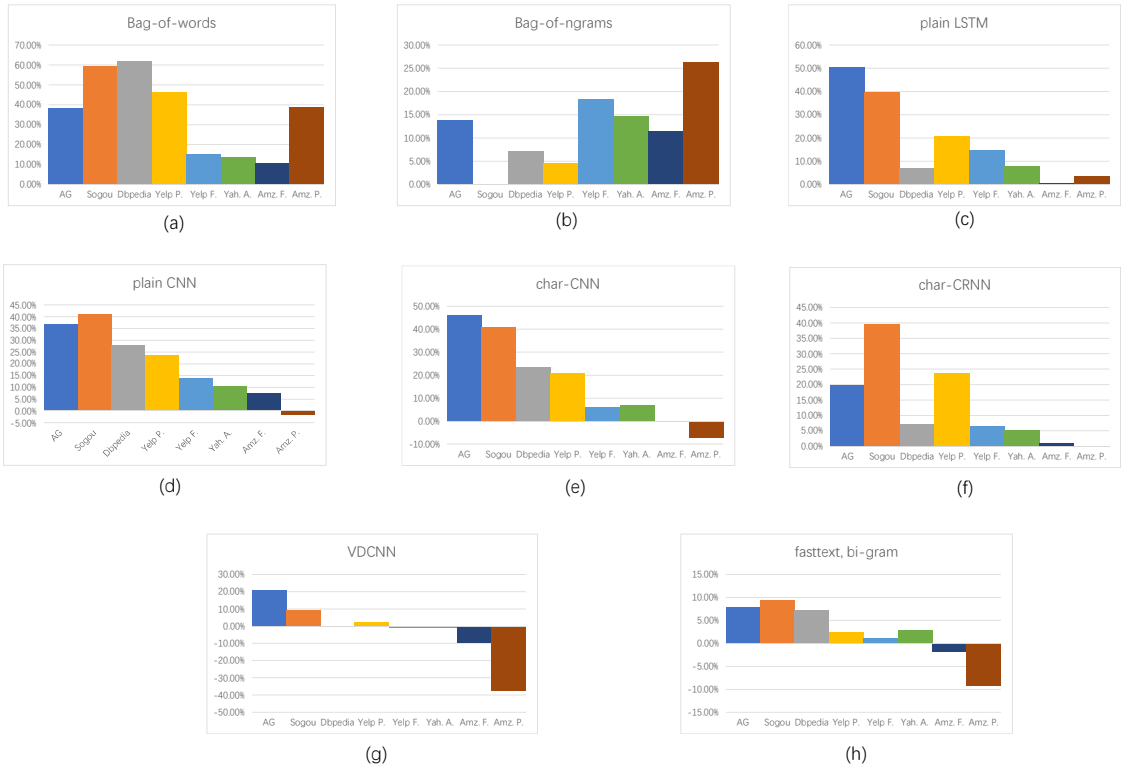

Figure 2: Relative errors when comparing FNText with other baselines. The value greater than zero means our FNText works better, otherwise the compared baseline performs better. (a) Compared to bag-of-words. (b) Compared to bag-of-ngrams. (c) Compared to plain LSTM. (d) Compared to plain CNN. (e) Compared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

AG dataset, which has a small vocabulary). For example, FNText using 5% vocabulary achieves 59.4% accuracy on Amz. F. dataset, while FNText trained on the full dataset just increases the accuracy to 59.5%. Note that too small vocabulary (e.g., 1%) may reduce the accuracy. On all datasets, FNText with 1% vocabulary gets the worse result. This is because some important but low-frequency words for text classification may be discarded in the incomplete vocabulary.

In practice, if there is enough memory, we can use complete vocabulary; otherwise, limiting the vocabulary size to 5% may lead to a good enough result in most cases.

FNText vs. shallow models FNText with bi-grams is better than the models based on word frequency statistics. We can see that Bag-of-ngrams achieves competitive results on Sougou, Dbpedia, Yelp P. and Amz. P. datasets, although it is a simple model. 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, while FNText gets 95.4%. However, when bi-gram features are integrated into FNText, the accuracy can be improved. For Sogou dataset, accuracy improves to 97.1%, which is equal to that of bag-of-ngrams. FNText with bi-grams significantly outperforms bag-of-ngrams on the Yelp. F. dataset. Figs. 2 (a)(b) also show that FNText with bi-grams performs better than models based on word frequency statistics.

Compared to shallow neural models (e.g., plain CNN, plain LSTM and fasttext), FNText with bi-grams gets higher accuracy on most datasets (one exception is that on the Amz. dataset). Basic FNText without bi-grams is on par with plain CNN and plainLSTM on AG and Sougou datasets. As the dataset grows larger, the difference between basic FNText and plain neural models becomes 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 outperforms another neural model fasttext on all datasets. After integrating the bi-gram features, FNText achieves higher accuracies compared to plain CNN and plain LSTM (see Figs. 2(c)(d)) except on Amz. P. dataset. Compared to another neural model fasttext (also integrating bi-grams), FNText achieves better performance in most cases (see Fig. 2(h)).

FNText vs. deep models Compared to plain shallow neural models like LSTM and CNN, deep neural models including char-CNN, char-CRNN, VDCNN achieve significant improvement on several large datasets. Our FNText and fasttext are also shallow nueral models, but can achieves competitive results using the bi-gram features. Specifically, FNText performs better than char-CNN and char-CRNN except on Amz. P. dataset (see Figs. 2(e)(f)). Compared to VDCNN, FNText get higher accuracy on AG, Sogou and Yelp P. datasets (see Fig. 2(g)). For larger datasets such as Amz. datasets, FNText gets lower accuracy, but is still competitive (the difference is 0.1%-3.5%). Note that deep models do not utilize n-grams, since the convolution operator with stride greater than 1 actually encodes the information of word order, and the CNN structure is not suitable for taking n-grams as input. Overall, FNText with bi-grams achieves better performance than deep neural models on small datasets; for large datasets, FNText with bi-grams also achieves competitive results.

3.4 Transfer learning

To explore the potential of transfer learning of our FNText, we conducted a sub-experiment on Yelp P. and Amz. P. datasets, which contain reviews and mainly consist of English. Table 3.4 shows the accuracies on Yelp P. and Amz. P. datasets, where FNText with bi-grams is used. We can observe that FNText trained on Amz. P. can achieve a satisfying result on Yelp P. dataset, verifying the transferability of FNText. The same conclusion can be obtained from the FNText that trained on Yelp.P and evaluated on Amz. P..

3.5 Computation time

Given N documents with the average length T, the forward complexity of FNText without bi-grams is $\mathcal{O}(NTk + Nkm)$, dominated by the computation of the feature-wise max pooling and fully connected layer. The backward complexity is

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

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

$\mathcal{O}(NTk + Nkm)$, using conventional optimization algorithm. When we use the tricks described in Section 2.3 to accelerate training, the complexity becomes $\mathcal{O}(Nk + Nkm)$, since the feature-wise max pooling only chooses k largest values as the document features, and we do not update parameters with zero gradients. For FNText with bi-grams, the forward complexity is $\mathcal{O}(2NTk + 2Nkm)$ and the backward complexity becomes $\mathcal{O}(2Nk + 2Nkm)$, since an average pooling operation is used to obtain the bi-gram features.

In Table 3.5, we list the average training time per epoch on serveral large-scale datasets. We compare FNText with Char-CNN, VDCNN and fasttext. 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 takes nearly $2\times$ time than FNText without bi-grams, which is the same to that of complexity analysis. Compared with fasttext, our FNText contains more parameters and runs slower, since a larger look-up table is used to achieve higher accuracy. Compared with deep models, FNText achieves significant performance gains. Specifically, FNText with bi-grams achieves a speedup of $1800\times$ than big char-CNN on AG dataset, $900\times$ on Amz.P. dataset. Besides, VDCNN runs faster than char-CNN. FNText with bi-grams still achieves a speedup of $53\times$ on Amz. P. dataset, $510\times$ on AG dataset. Overall, FNText runs much faster than deep models at a cost of slightly lower accuracy; compared to fasttext, FNText runs slower but achieves higher accuracy.

3.6 Memory usage

Table 3.5 shows the number of parameters and memory usage of FNText with a full vocabulary. The number of parameters is proportional to the size of vocabulary, and FNText with bi-grams has $2\times$ parameters compared to FNText without bi-

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 #param	FNText #memory	FNText,bi-gram #param	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 parameters and memory usage.

grams. Memory usage is determined by the number of parameters and the size of training set.

Note that there are several methods to reduce the memory usage, e.g., limiting the vocabulary size (see Section 3.3) and using SGD instead of Adam (see Section 2.3). In practice, FNText with bi-grams requires 47.7G memory on Amz.P. dataset. The memory requirement decreases to 17.3G when limiting the vocabulary size to 5%, or 16.3G using SGD instead of Adam. If we adopt both two methods, FNText with bi-grams only needs 6.2G memory, which is acceptable for 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 accelerate training. The main difference between fasttext and our FNText is that fasttext averages the word vectors as the text representation, while FNText employs feature-wise max pooling to construct the text features. During back-propagation, fasttext needs to update all word vectors for a given document, while FNText only updates word vectors selected by feature-wise max pooling. Thus, FNText can use word vectors with a larger hidden dimension (i.e., 800 for FNText and 10 for fasttext) to obtain robust features and achieve better performance. Moreover, FNText applies average pooling to generate n-gram features, which is different from the conventional enumerating method used by fasttext.

5 Conclusion

In this paper, we proposed a fast shallow neural model for text classification. We provided several tricks to improve the prediction performance and training efficiency, such as applying average pooling to obtain bi-grams, using feature-wise max pooling to generate text features and ignoring the parameters with zero gradients. We conducted a comprehensive set of experiments to demonstrate the effectiveness of our FNText on large-scale text classification dataset.

References

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

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

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537.

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, 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.

Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. 2014. Convolutional neural network architectures for matching natural language sentences. In *Advances in neural information processing systems*, pages 2042–2050.

Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 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.

David C Plaut et al. 1986. Experiments on learning by back propagation.

Cicero dos Santos and Maira Gatti. 2014. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of the 25th International Conference on Computational Linguistics*, pages 69–78.

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.