CONTEXT-FOFE BASED DEEP LEARNING MODELS FOR TEXT CLASSIFICATION AND MODELING

YUPING LIN

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

GRADUATE PROGRAM IN
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
AUGUST 2017

Abstract

Text classification is a fundamental task in natural language processing. Many recently proposed deep learning models have leveraged context information in documents and achieved great successes. However, most of these models use complicated recurrent structures to handle the variable-length text and to record context information, which are hard to train. In this case, we propose a simple and efficient encoding scheme called context-FOFE that can encode context of variable-length documents into fixed-size representations. Our encoding is unique and reversible for any text sequence. Based on the encoded representations of documents, we further use two feed-forward neural network models and a generative HOPE model for text classification and modeling. We tested the models on the 20 Newsgroups text classification dataset and the IMDB sentiment analysis dataset. Experimental results show that our models can achieve competitive performance as the existing best models while using much simpler context encoding mechanism and network structure.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Hui Jiang for his continuous support through my Master study and research. His motivation, enthusiasm and immense knowledge are what I admired and encourage me to keep on studying. I also deeply appreciate his patience and valuable guidance helping me in both research and writing this thesis.

I would also like to express my gratitude to all the professors who taught me during my undergraduate and graduate study at York University, for the knowledge and help they gave me during my study.

I am especially thankful to York University and the department of computer science for their support and providing all the facilities to enrich my graduate study experience.

My special thanks also goes to my parents. Their understanding, support and encouragement give me the opportunity to pursue my degree.

I would also like to thank all my friends and colleagues in iFLYTEK Laboratory

for Neural Computing and Machine Learning (iNCML) at York University who assisted me in research and writing this thesis. I have learned a lot from many discussions with them.

Table of Contents

List of Tables

List of Figures

Abbreviations

BoW	**B**ag **o**f **W**ords
BPTT	**B**ack-**P**ropagation **T**hrough **T**ime
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
EM	**E**xpectation-**M**aximization (algorithm)
FNN	**F**eed-forward **N**eural **N**etwork
FOFE	**F**ixed-size **O**rdinally **F**orgetting **E**ncoding
GPU	**G**raphics **P**rocessing **U**nit
HOPE	**H**ybrid **O**rthogonal **P**rojection and **E**stimation
LDA	**L**atent **D**irichlet **A**llocation
LSTM	**L**ong **S**hort-**T**erm **M**emory (network)
MLE	**M**aximum **L**ikelihood **E**stimation
NLP	**N**atural **L**anguage **P**rocessing
NN	**N**eural **N**etwork

PCA	**P**rinciple **C**omponent **A**nalysis
RCNN	**R**ecurrent **C**onvolutional **N**eural **N**etwork
ReLU	**Re**ctified **L**inear **U**nit
RNN	**R**ecurrent **N**eural **N**etwork
SGD	**S**tochastic **G**radient **D**escent
SVM	**S**upport **V**ector **M**achine
SWE	**S**emantic **W**ord **E**mbedding
vMF	**v**on **M**ises-**F**isher (distribution)

1 Introduction

Text classification (or text categorization) is an important task in Natural Language Processing (NLP) that aims to automatically assign natural language text documents to predefined categories according to their content. In terms of a classification problem, the task is defined as follow. Given a set of N training samples $X = \{x_1, \ldots, x_N\}$, with each of them associated with a label from a set of K discrete values $C = \{1, \ldots, K\}$ indicating the document categories, we need to build a classification model that captures the underlying relationship between text documents and category labels. The classification model is expected to take as input an unseen text document, then correctly predict its category label. In this case, performance of the classification model is usually measured by its prediction accuracy on a test dataset that has no overlap with the training samples.

As a fundamental task in NLP, text classification has applications in a wide variety of NLP tasks and real-world problems. Here are some of the common applications:

- **Document organization, browsing and retrieval:** Nowadays there are many huge text document collections not only in the libraries but also on the web. And the number and size of the collections still keep growing. Examples of such large text document collections include digital libraries, web article collections and scientific literature databases. To ensure efficient browsing and retrieval, they need to be well-organized. However, it will cost too much human labor to manually read and label such a large number of text documents. In this case, a text classifier can help to automate the categorization process in order to greatly ease the human effort needed for organization and maintenance.

- **News and article recommendation:** Today many people like to read news and articles on-line because they are comprehensive, fast and convenient. Many of the news websites and blogs not only collect and organize news and articles but also do recommendation. Here is where text classification techniques can be used to group the vast volume of news and articles generated everyday, according to their topic and similarity, and then recommend related news and articles to users. Similar techniques can also be used in search engines.

- **Spam filtering:** Spam filtering is one of the earliest real-world applications of

text classification techniques. The goal is to help people identify junk emails in an automatic way. To do this, a very simple way is to define a blacklist of certain phrases and patterns along with a set of rules, then follow the rules to identify junk emails. A more advanced way is to treat junk emails and normal emails as two categories of text documents, and train classification models to distinguish them.

- **Sentiment analysis:** Sentiment analysis is a popular NLP task in recent years that aims to extract and identify subjective information like opinion, attitude and emotion from natural language text. A typical scenario is to classify customer reviews into different satisfaction categories such as *positive*, *neutral* and *negative*, which can be used as a reference for marketing and production. To a certain extent, this can be viewed as a text classification problem and text classification models can be used to tackle this problem.

- **Topic and trend identification:** With the growing popularity of social media platforms, more and more people share their thoughts, ideas and opinions on-line. Many social media platforms like Twitter and Weibo use hashtags to specify the topic or theme of posts. From a data mining point of view, text classification techniques can be used to automatically identify the hot topics, people's opinions and the trend of people's interest.

Over the last two decades, extensive research has been done on the topic of text classification and many models have been proposed. Among these models, some of the traditional key models [1] include:

- **Decision Trees:** Decision trees are traditional methods in machine learning that decide the output value by answering a series of true/false questions along a tree-like structure. Each interior node of the tree associates with one question, and each branch corresponds to one answer. The questions are often about features of the testing sample. Hence based on the feature combinations of the testing sample, there will be a path leading to a leaf node that represents the classification decision.

- **Rule-based models:** This type of model attempts to classify texts into different categories based on a set of rules about word patterns. The rules are usually handcrafted and complicated. However, due to the complexity and flexibility of human language, this type of model only works well on small datasets and is hard to be generalized to practical situations.

- **SVM classifiers:** Support vector machines (SVM) [6] are effective models for classification problems. They directly find separation boundaries with maximum separation margins between different classes, without the need to estimate data distributions of different classes. With the learned separation

boundaries, new data are classified based on which side of the separation boundaries they belong to.

- **Bayesian classifiers:** Bayesian classifiers are typical generative models for text classification. They first learn the probability distribution of text document features under different classes, then perform classification based on the posterior probability computed for each class given the features of the testing document.

In recent years, with the successes of artificial neural networks (NN) in automatic speech recognition and computer vision, deep learning methods that use various types of neural networks have become popular in the field of NLP, including text classification. NNs are proven to be universal approximators that can approximate any function mappings given large enough model size [7]. Comparing with the traditional machine learning methods, NNs usually can learn better on high-dimensional data while using less sophisticated features. Some of the NN models even adopt the end-to-end modeling strategy, in which the models take only raw data as inputs and learn to solve the problem directly during model training. A typical representation for the raw text data is the one-hot representation, in which each word in the vocabulary is represented by a vector of vocabulary size. In a one-hot vector, all elements are set to be 0 except that the element in the corresponding

position of the representing word in the vocabulary list is set to 1. That is, for every one-hot vector there is one and only one element that has high signal and the position of the high signal indicates the word it represents. With the end-to-end modeling strategy, the useful features are automatically learned within the models and hence saves the labor for feature engineering. In this thesis, we mainly focus on studying deep learning methods for the text classification and text modeling tasks.

1.1 Motivation

NLP problems are generally challenging, and so is text classification. Natural language texts are very high-dimensional, sparse and discrete signals in nature. They are also highly contextual. This imposes the need for large volume of training data and good feature representations. In this case, a good feature representation needs to embed as much useful information as possible in an easy to learn manner, while still meeting the requirements of the models. Such a good feature representation, no doubt, can help the model learn better from limited number of training samples. Moreover, most machine learning models, including NNs, need to have fixed size input. However, text documents are often of different lengths. Therefore we also need a representation that can convert documents of any lengths into fixed size vectors.

The bag-of-words (BoW) [13] is a simple but commonly used feature representation for text documents. In the BoW representation, a piece of text is represented as a multiset of words in the vocabulary list, that is, a list of all words in the vocabulary together with their frequencies in the text. For example, with a vocabulary list as {"to", "be", "or", "not"}, the sentence "to be or not to be" is represented in BoW as "$[2, 2, 1, 1]$", a vector of vocabulary size where each position of the vector corresponds to a word in the vocabulary list in the same position. Obviously, this model ignores much important information like grammar, word relations and word ordering in the text, which limits the system performance. Another simple trick to convert text documents of varying lengths to fixed size is to truncate long text sequences and pad short text sequences [17, 44]. Nevertheless, this approach is not elegant and the alteration of text sequences may change their original meaning. Some recurrent neural network (RNN) based models [20, 22, 38] solve this problem by reading the text documents word by word, and memorize the information using their built-in recurrent structures. This is a more natural approach for reading text documents but it is specifically for RNN based models.

To distinguish text documents of different categories, a reasonable assumption is: **Documents of different categories usually have different compositions of words, phrases and sentences; while documents of the same category**

usually have similar compositions of words, phrases and sentences. This assumption implies that we can model documents of different categories by capturing their composition patterns. These composition patterns are however hidden within the context of text documents, which need to be discovered by the classification models through the learning process. Some recent text classification models already more or less leverage contextual information of text documents and achieve good results. For example, the Convolutional Neural Network (CNN) based model in [19] uses a convolution window to encode partial context, and the Recurrent Convolutional Neural Network (RCNN) model in [22] uses a recurrent structure to encode full context information. We view this as a strong indication for the importance of context information in the text classification task. However, so far there is no efficient text representation method that can encode full context of text documents into fixed size representations and is applicable to most machine learning models.

In this work, our main goal is to design an efficient text representation method that can encode full context of text documents into fixed size representations. Additionally, we also want our method to be generally applicable to many machine learning models. With the desired context representations, our second goal is to design a generative model that can model the probability distribution of text doc-

uments. The most popular generative model for text documents is the Latent Dirichlet Allocation (LDA) model published by Blei et at. in 2003 [4]. In LDA, all words are treated as independent features and are exchangeable within the document. This approach ignores the word ordering information and hence cannot capture the context patterns in the text documents. Noticing this, we believe a new generative model that can capture the context patterns of text documents will better model the text distributions.

1.2 Contribution

In this thesis, we propose a novel feature representation method, namely, the context-FOFE encoding scheme, that can encode full context information of text documents of arbitrary length into fixed size vectors. Our method is an extension of the Fixed-size Ordinally-Forgetting Encoding (FOFE) scheme proposed by Zhang et al. in 2015 [41, 42]. FOFE is a simple and efficient encoding scheme that can encode any text sequence into a fixed size vector representation. The encoded representation of any variable-length text sequence is almost unique, given a sufficiently large vocabulary. Moreover, the encoding process is simple and does not involve any training. FOFE memorizes the word order in a text sequence using a simple ordinally-forgetting mechanism that relates weights of the words with their

positions in the sequence. This mechanism is a recursive process in which weights of the previously read words are decreased each time a new word is read in, by a preset hyper-parameter called the forgetting factor. Clearly, this mechanism puts more weights on the latter words of a text sequence. In practice, when encoding long text sequence like text document, this will result in unwanted biased attention on the latter part of text document since the former words are gradually forgotten as their weights become insignificant.

Our context-FOFE encoding scheme solves this issue by encoding the context around every word position of a text document, and aggregating them to be a matrix. For each word position, which hereafter is referred to as the center word of the context around it, we represent its context as a left-context and a right-context, where left context denotes word sequence from document beginning to the center word and right-context denotes word sequence from document end backward to the center word. In order to capture both the context patterns before and after the center word, we use FOFE to encode the left-context and the right-context respectively. Our context-FOFE encoding scheme has the following advantages. Firstly, our encoded representation for each document is unique and reversible, hence there is no information loss. Secondly, our representation comprises of fixed size vectors, which are generally applicable to many machine learning models. Thirdly, by sam-

pling context patterns at each word position, our representation provides slightly different views of a document, which can potentially make the learning of context patterns easier. Fourthly, the different views of a document effectively provide more data for training. And last but not least, our method is efficient and fast, because it does not need to learn any parameter and is only controlled by a single hyper-parameter called forgetting factor.

To examine the effectiveness of our new feature representation for text documents, we use a regular feed-forward neural network (FNN) model on our encoded document representations to perform a text classification task and sentiment analysis task. We designed a position-wise training model and a document-wise training model to leverage position-wise encoded features. Experimental results show that our models achieved better performance than the existing state-of-the-art [22] on the 20Newsgroups document categorization dataset [23], and close to the state-of-the-art [8] performance on the IMDB sentiment analysis dataset [28]. These results show the effectiveness of our new feature representation method.

In this thesis, we also studied the use of our new document representation method on document modeling. We decided to use the Hybrid Orthogonal Projection and Estimation (HOPE) model [43] to model the probability distribution of context patterns in our document representations. HOPE is a powerful genera-

tive model for high-dimensional data. It unifies the feature extraction stage and the data modeling stage into a single learning process so that context patterns are automatically extracted while being probabilistically modeled. We evaluate our models on the same datasets used with the above FNN models. We propose to model the context patterns of each document category separately, and classify new documents based on probabilistic scores against each category's model. Experimental results show that our generative model greatly outperforms the LDA based model [14]. This confirmed our assumption that context patterns are important and result in better features for text modeling.

1.3 Outline

This thesis is divided into seven chapters. Chapter 1 describes the task of text classification, as well as our motivations and contributions of this work. Chapter 2 introduces background knowledges about artificial neural networks (NN) and some related works to this thesis. Chapter 3 proposes our novel context-FOFE encoding scheme that can uniquely encode variable-length text sequences into fixed size representations. Chapter 4 designs two regular feed-forward neural network models based on the context-FOFE encoded representations to perform the text classification task. Chapter 5 uses the context-FOFE and the HOPE framework to build a

generative model for text modeling. Chapter 6 presents experimental results of the proposed models on a document categorization task and a sentiment analysis task. Finally, chapter 7 concludes this thesis.

2 Background and Related Works

2.1 Word Embeddings

Word embeddings (or word vectors) are vector representations of words in a continuous vector space. It is an important and commonly used technique in NLP that maps the discretely distributed words in the vocabulary to fixed-size vectors in a continuous space such that the similarities between words are reflected by their distances. The aim of word embeddings is to quantify and categorize semantic similarities between linguistic items based on their distributional properties in a large amount of language data. An important distributional property is the idea that "a word is characterized by the company it keeps" [9], which serves as the underlining assumption for many of the word embedding generating models. The development of word embedding technique began in the 2000s and various models have been proposed [3, 10, 25, 26]. The most popular model nowadays is the word2vec model [30–32] that trains word embeddings in a language modeling task.

Since word embeddings reflect similarities between words, they are popularly used in many NLP models to relate words in the semantic space.

2.2 Artificial Neural Networks

The idea of neural networks (NN) originates from attempts to find mathematical representations of biological information processing systems [29, 35, 40]. The first neural network was published in 1943 [29]. However, neural networks achieved limited success before the publication of the back-propagation algorithm in the 1980s [36], which makes the training of multi-layered neural networks feasible. Since then, NNs have been gradually applied to many different problems. And nowadays, thanks to the massive computational power of CPUs and GPUs, as well as extensive availability of training data, NNs have shown success in many diverse applications, such as:

- speech recognition,
- object recognition,
- machine translation,
- decision making,
- and many more ...

2.2.1 Basic units

NNs have many different structures. But in general, they are built with two basic components: artificial neurons and connections. Artificial neuron (or simply neuron), as the name suggests, is an imitation of the biological neuron. An artificial neuron can have multiple input connections as well as multiple output connections with other neurons. Each connection is associated with an adaptive weight w that is to be learned from data. Every artificial neuron also has an activation function $f(\cdot)$ that controls the output of the neuron:

$$z_j = f(a_j) = f(\sum_{\forall i, \exists w_{ij}} (w_{ij} \cdot z_i) + b_j) \tag{2.1}$$

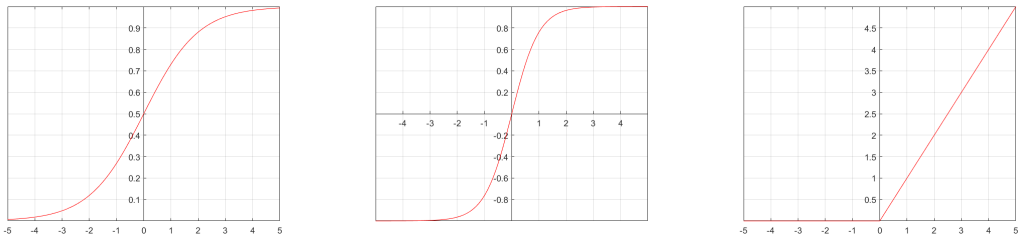
where z_j is the output of neuron j, a_j is the accumulated signal before the activation function, w_{ij} is the adaptive weight associated with the connection from neuron i to neuron j if existed, and b_j is the bias associated with neuron j. In this way, signals from other neurons are adjusted by the weights and then accumulated before the activation function. If the accumulated signal is strong enough, the activation function will trigger an output of this neuron and send it to other neurons through the output connections.

2.2.2 Activation functions

Activation function is a gate that controls the output of a neuron. There are many activation functions and some of the commonly used ones are:

- **sigmoid:** $f(x) = \frac{1}{1+e^{-x}}$
- **tanh:** $f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$
- **Rectified Linear Unit (ReLU):** $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

and their corresponding plots are shown in figure 2.1.



(a) sigmoid (b) tanh (c) ReLU

Figure 2.1: Commonly used activation functions.

Activation functions are generally non-linear. Non-linearity allows neural networks to compute nontrivial problems using only a small number of neurons. Moreover, it has been proven that a neural network with at least two layers of non-

linearity can be a universal function approximator [7]. However, this layered non-linearity makes the error function non-convex, which has no direct method for an optimal solution. In this case, iterative optimization algorithms can be used to estimate a sub-optimal solution.

Activation functions usually also need to be continuously differentiable. This is particularly required for gradient-based optimization algorithms, as error back-propagation needs to compute derivatives of the activation functions. ReLU [12], as an exception, is not differentiable at the origin but it is differentiable at every other point, which does not affect the gradient-based optimization. Another desirable property of activation functions is the property of approximating identity near the origin. When activation functions have this property, weights can learn efficiently when they are initialized with small random values. But when activation functions don't have this property, special care may be needed for initializing the weights [37].

2.2.3 Structural variants

Neural networks are built with many neurons. These neurons are generally organized layer by layer, with connections between different layers. The network layers usually can be divided into an input layer, an output layer, and multiple hidden layers. As their name suggests, the input layer is used to encapsulate input vec-

tors, while the output layer is used to produce network outputs. Although the way how NNs generalize from observations is not yet fully understood, the multiple hidden layers, can be viewed as sequential abstractions of the inputed information; deep neural networks today typically consist of more than one hidden layer. During training, the information first flow from the input layer via the hidden layers to the output layer, which is called the forward phase; and then flow back from the output layer via the hidden layers to the input layer, which is called the error back-propagation phase.

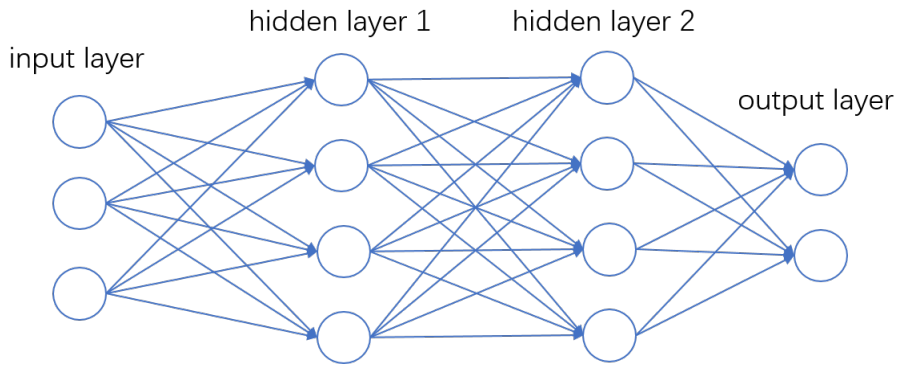


Figure 2.2: Typical structure of feed-forward neural networks (FNN).

There are many structural variants of NNs. Among them, the regular feed-forward neural networks (FNN) are of the simplest kind. The typical structure of FNNs is shown in figure 2.2. As shown in the figure, FNNs have an input layer, multiple hidden layer and an output layer, arranged sequentially. The adjacent

layers in the network are fully connected and there is no intra-connection within each layer. Note that there is no cyclic connection in the network, and this is why it is called feed-forward neural network.

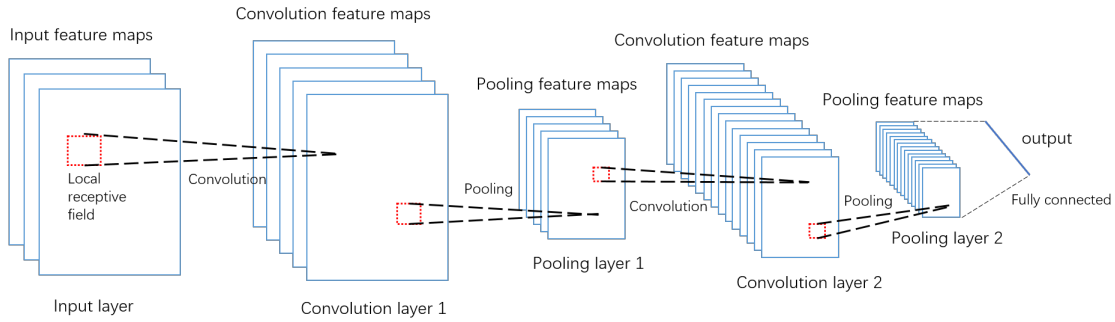


Figure 2.3: Typical structure of convolutional neural networks (CNN).

Another well-known type of neural network variant is the convolutional neural network (CNN) [24], which is specially designed to handle images. As shown in figure 2.3, CNNs typically contain several pairs of convolution layer and pooling (subsampling) layer. Each layer has a number of feature maps of the same size, with each of them representing one kind of feature extracted from different locations of the previous layer's feature maps. In the convolution layer, CNNs swipe over the input image feature maps and apply convolution operation on small regions of the image called local receptive fields; these convolution operations on each of the small regions share the same convolution weights. This mechanism of convolution on small regions is designed to extract local features of the images. In the pooling

layer, with the local features detected on the previous convolution layer, CNNs perform pooling operations on small regions to select or combine the local features in the convolution layer. This pooling operation has the purpose of generalizing lower-level features as well as to reduce resolution of the feature maps. These mechanisms together, can help CNNs to handle the local shifts and distortions often seen in images.

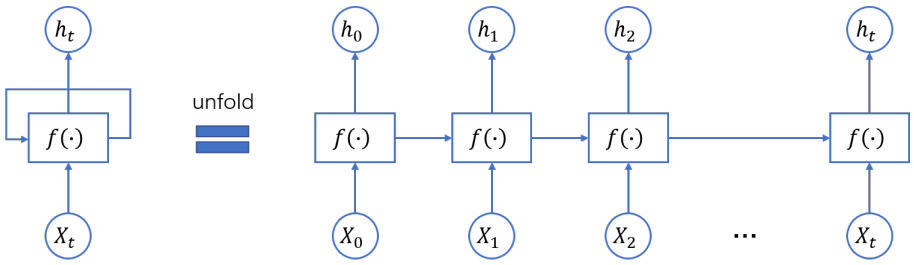


Figure 2.4: Typical structure of recurrent neural networks (RNN).

In the FNNs and CNNs, the network output depends only on the current input and no historical information is used. Recurrent neural network (RNN) is a different type of network architecture that can take into account historical information by its recurrent structure. As shown in figure 2.4, RNN has a cyclic connection that connects the hidden layer back to itself. In this way, not only the current input, but also the previous state of the hidden layer contributes to the current state of the hidden layer; hence the state of hidden layer serves as a memory of the historical information. During training, RNNs usually need to be unfolded

to allow error information to back-propagate to inputs at every timestamp. This back-propagation through previous states operation is usually referred to as back-propagation through time (BPTT) [39]. Since error information in RNNs need to back-propagate through a sequence of unfolded previous layers, it is usually much harder to train RNNs than regular FNNs. And moreover, gradients will often gradually vanish or explode along a series of multiplications in the back-propagation procedure, which becomes a major difficulty in training RNNs [16].

The Long Short Term Memory network (LSTM) [15] is a kind of RNN that is specially designed to avoid the vanishing gradient problem. LSTM has the chain of repeating modules structure like standard RNNs. But within each module, instead of using a single activation function on the input and the previous hidden state, LSTM uses a complicated gating mechanism to control the forgetting and updating of the hidden state and the output. Hence unlike RNN that can only capture the short-term dependencies, LSTM is also capable of capturing long-term dependencies.

2.2.4 Optimization algorithms

In order to guide the learning process towards a better solution, we need to set an evaluation metric for the learning. Such an evaluation metric can be defined

as a function that maps a parameter setting to a real number. Conventionally in machine learning, this number is designed to represent some "cost" that needs to be minimized. Hence the defined function is often called the cost function, or also referred to as loss function, error function or objective function. The task of optimization in machine learning is to find the set of model parameters that minimizes the loss function.

There are many optimization algorithms that exist for the training of neural networks. Stochastic gradient descent (SGD) [5] is probably the most famous one. In the original gradient descent algorithm, the learning process iteratively takes steps proportional to the negative of the gradient of the loss function at the current point. Let $L(\cdot)$ denote the loss function and $\boldsymbol{W}^t$ denote the model weights at the t-th iterative step, a gradient descent updating step can be represented as:

$$\boldsymbol{W}^{t+1} = \boldsymbol{W}^t - \gamma \cdot \nabla L(\boldsymbol{W}^t) \tag{2.2}$$

where γ is a small step size called the learning rate, and $\nabla L(\boldsymbol{W}^t)$ denotes the gradient of the loss function. The loss function $L(\boldsymbol{W})$, in practice usually comprises a sum of terms, one for each training sample in the dataset:

$$L(\boldsymbol{W}) = \frac{1}{N} \sum_{n=1}^{N} L_n(\boldsymbol{W}) \tag{2.3}$$

where $L_n(\boldsymbol{W})$ is the loss function on the n-th data sample with the models weights $\boldsymbol{W}$. In each iterative update step, the algorithm needs to compute the gradients

summed over all the data samples, which is a heavy computation for large datasets. In this case, stochastic gradient descent (SGD) approximates the true gradient by the gradient of a randomly picked single data sample (or a mini-batch of data samples in practice):

$$\boldsymbol{W}^{t+1} = \boldsymbol{W}^t - \gamma \cdot \nabla L_n(\boldsymbol{W}^t) \tag{2.4}$$

In this way, the algorithm avoids the heavy summation computation in each iterative update step. And moreover, the randomness introduced in each update step can help the model to move out of some local minima. Although using an inaccurate gradient in each update step, it has been proven that SGD can almost surely converge to a local minimum, under some minor assumptions [5]. In practice, we refer an epoch as a complete traversal of the whole dataset. In the start of every epoch, the dataset is randomly divided into a number of mini-batches and one update step will be performed on each mini-batch. This implementation is sometimes referred to as mini-batch gradient descent.

Adam [21] is another efficient gradient-based optimization algorithm. It has a more complex mechanism for weight updating that combines with momentum and second moments of the gradients. In the t-th iteration, the algorithm updates

model weights according to the following formulae:

$$\boldsymbol{m}^{t+1} = \beta_1 \cdot \boldsymbol{m}^t + (1 - \beta_1) \cdot \nabla L(\boldsymbol{W}^t) \tag{2.5}$$

$$\boldsymbol{v}^{t+1} = \beta_2 \cdot \boldsymbol{v}^t + (1 - \beta_2) \cdot (\nabla L(\boldsymbol{W}^t))^2 \tag{2.6}$$

$$\hat{\boldsymbol{m}} = \frac{\boldsymbol{m}^{t+1}}{1 - {\beta_1}^t} \tag{2.7}$$

$$\hat{\boldsymbol{v}} = \frac{\boldsymbol{v}^{t+1}}{1 - {\beta_2}^t} \tag{2.8}$$

$$\boldsymbol{W}^{t+1} = \boldsymbol{W}^t - \gamma \cdot \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{v}}} + \epsilon} \tag{2.9}$$

where ϵ is a small number used to prevent division by 0, and β_1 and β_2 are the forgetting factors for the moving averages of the first and second moments of the gradients. The recommended values for these parameters are: $\epsilon = 1e-8$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$ as in [21]. We use the these recommended values in our implementation. Adam is a per-parameter adaptive learning rate method because the gradient of each model weight is individually and automatically adjusted based on the history of the first and second moments of the gradient. However, a learning rate annealing schedule is still necessary for training.

Learning rate is one of the common but very important hyper-parameters in the training of NNs. As shown in equation 2.2, learning rate is a hyper-parameter to control the step of updates. If learning rate is set too large, the learning may not converge. On the other hand, if the learning rate is too small, the learning speed

will be very slow. Learning rate typically needs to decrease along with the training of NNs. A scheme that specifies how learning rate decreases is call the learning rate annealing schedule. There are many learning rate annealing schedules, ranging from simple schedule like multiplying with a constant factor (< 1) to more complicated mechanism like only halving the learning rate when no improvement is observed.

2.2.5 Error back-propagation

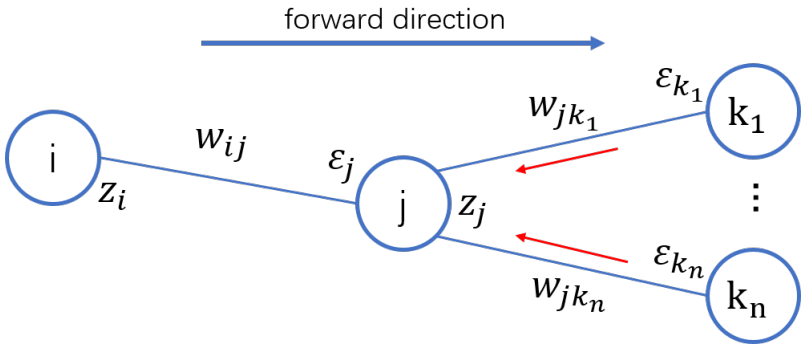


Figure 2.5: Illustration of the back-propagation mechanism. The blue arrow denotes the forward direction of information flow, and the red arrows denotes the direction of the error back-propagation.

As we have seen from the above subsection, the training of neural networks often involves the computation of gradients for all the adaptive weights at each update step, which is usually the most computationally expensive part in the training process. Back-propagation [36] is a well-known algorithm that can compute gradients

of all adaptive weights simultaneously. It is commonly used as a part of iterative optimization algorithms because of its efficiency in computing gradients. As shown in equation 2.3, loss functions usually comprise of a sum of terms. In this case, it is sufficient for us to first derive the gradients for only one training sample and then generalize to more samples. Consider an arbitrary neuron j, as shown in the figure 2.5. The desired gradient of the loss function with respect to an adaptive weight w_{ij} can be expressed, according to the chain rule for partial derivatives, as:

$$\frac{\partial L_n(\boldsymbol{W})}{\partial w_{ij}} = \frac{\partial L_n(\boldsymbol{W})}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ij}} = \frac{\partial L_n(\boldsymbol{W})}{\partial a_j} \cdot z_i \tag{2.10}$$

Let ε_j denote $\frac{\partial L_n(\boldsymbol{W})}{\partial a_j}$, the formula to compute the gradient of weight w_{ij} becomes:

$$\frac{\partial L_n(\boldsymbol{W})}{\partial w_{ij}} = \varepsilon_j \cdot z_i \tag{2.11}$$

Now we turn to the term ε_j, which can also be decomposed by the chain rule as:

$$\begin{aligned} \varepsilon_j = \frac{\partial L_n(\boldsymbol{W})}{\partial a_j} &= \frac{\partial L_n(\boldsymbol{W})}{\partial z_j} \cdot \frac{\partial z_j}{\partial a_j} \\ &= f'(a_j) \cdot \sum_{\forall k, \exists w_{jk}} \left(\frac{\partial L_n(\boldsymbol{W})}{\partial a_k} \cdot \frac{\partial a_k}{\partial z_j} \right) \\ &= f'(a_j) \cdot \sum_{\forall k, \exists w_{jk}} \left(\varepsilon_k \cdot w_{jk} \right) \end{aligned} \tag{2.12}$$

We refer the term ε_j as *error*. According to the above formula, the error term can be computed backward using the error terms of the neurons in the next layer. This is why it is called the error back-propagation algorithm. In general, the back-propagation algorithm uses the following procedure:

1. Feed the inputs into the neural network, compute all z's and a's in the forward phase using formula 2.1;

2. Get the error terms of the output neurons, which can be computed based on the specific loss function used;

3. Back-propagate errors from the output neurons to the other neurons layer by layer, according to formula 2.12;

4. Compute the gradients with respect to the adaptive weights with formula 2.11;

5. Update model weights following formula 2.2;

2.3 Related Works

In this section, we will briefly describe some of the related works that we are going to compare our model performance with. These related works include a LDA based generative model and some neural network based models.

2.3.1 LDA and its based model for text classification

The Latent Dirichlet allocation (LDA) model [4] is a popular generative model for text modeling. It is a three-level hierarchical model in which each document is represented as a random mixture of latent topics, and each latent topic is characterized

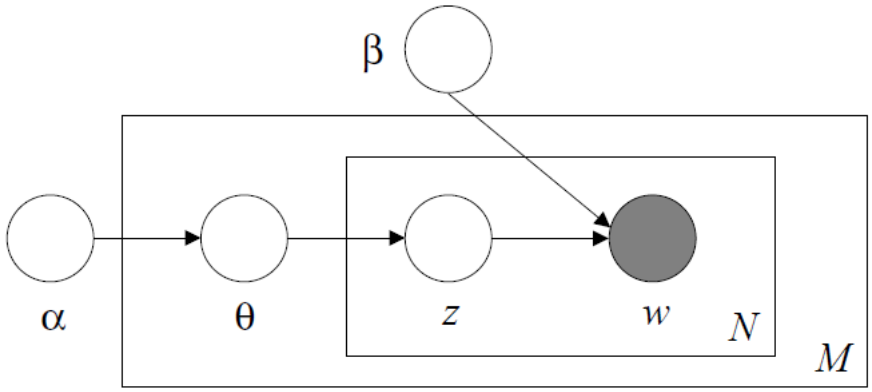


Figure 2.6: Graphical model representation of LDA. The boxes are "plates" representing replicates. The outer plate represents documents, while the inner plate represents the repeated choice of topics and words within a document. (*This figure is from [4]*).

by a distribution over words. By introducing the level of mixture of latent topics, LDA is able to associate multiple topics with a document. Comparing with many other clustering models that restrict a document to be associated with a single topic, the flexibility of repeatedly selecting and switching topics within a document makes LDA more powerful to model real-world documents. However, like most of the other models, LDA follows the assumption of exchangeability for simplicity, which ignores the order of words in a document.

LDA assumes the following generative process to generate a document:

1. Choose document length $N \sim \text{Poisson}\,(\lambda)$, where λ is the average document length in the current setting.

2. Choose topic selecting probabilities $\boldsymbol{\theta} \sim \text{Dir}\,(\boldsymbol{\alpha})$, where $\boldsymbol{\alpha}$ is a parameter vector of positive reals.

3. For each of the N words w_n:

   (a) Choose a topic $z_n \sim \text{Multinomial}\,(\boldsymbol{\theta})$.

   (b) Choose a word $w_n \sim \text{Multinomial}\,(\boldsymbol{\beta}(z_n, :))$, a multinomial probability distribution conditioned on the topic z_n.

In the above generative process, $\boldsymbol{\theta}$ is a K-dimensional variable generated from

a Dirichlet distribution:

$$p(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) = \frac{\Gamma(\sum_{i=1}^{K} \alpha_i)}{\prod_{i=1}^{K} \Gamma(\alpha_i)} \prod_{i=1}^{K} \theta_i^{\alpha_i - 1} \tag{2.13}$$

where $\boldsymbol{\alpha}$ is a K-dimensional parameter vector with components $\alpha_i > 0$, K is the total number of possible topics, and $\Gamma(\cdot)$ is the Gamma function. According to the property of Dirichlet distribution, $\boldsymbol{\theta}$ takes values in the $(K-1)$-dimensional simplex so that $\forall i, \theta_i \geq 0 \,\&\, \sum_{i=1}^{K} \theta_i = 1$. Therefore $\boldsymbol{\theta}$ is used to represent probabilities of selecting each topic in a topic selection step during the generation of this document. After defining the $\boldsymbol{\theta}$, the process then repeatedly generates a word until it reaches the document length. In the generation of each word, the process first selects a topic z_n from all K topics following the multinomial distribution Multinomial $(\boldsymbol{\theta})$. Then based on the selected topic z_n and $\boldsymbol{\beta}$, which is a $K \times V$ matrix that stores the conditional word probabilities for each topic (i.e., $\beta_{ij} = p(w_j \mid z_i)$), the process generates a word according to a topic-conditioned multinomial distribution Multinomial $(\boldsymbol{\beta}(z_n, :))$.

In the paper [14], Hingmire et al. proposed a document classification algorithm called ClassifyLDA that is based on LDA. In the algorithm, they first construct a topic model using LDA on the corpus, then they manually assign a class label to each of the learned topics according to expert knowledge. After that, they aggregate all the topics within each class into a single topic for that class using

the aggregation property of the Dirichlet distribution. With the aggregated topics associated with the classes, the system then can automatically classify an unlabeled document depending on its "closeness" to the class topics. Based on this algorithm, they further extend to build a naïve Bayes classifier for text classification and use the well-known Expectation-Maximization (EM) algorithm for optimization.

2.3.2 Neural network based models for text classification

Many of the current best NN-based models for text classification typically use some recurrent structure to handle the variable-length text documents and to capture context information for training.

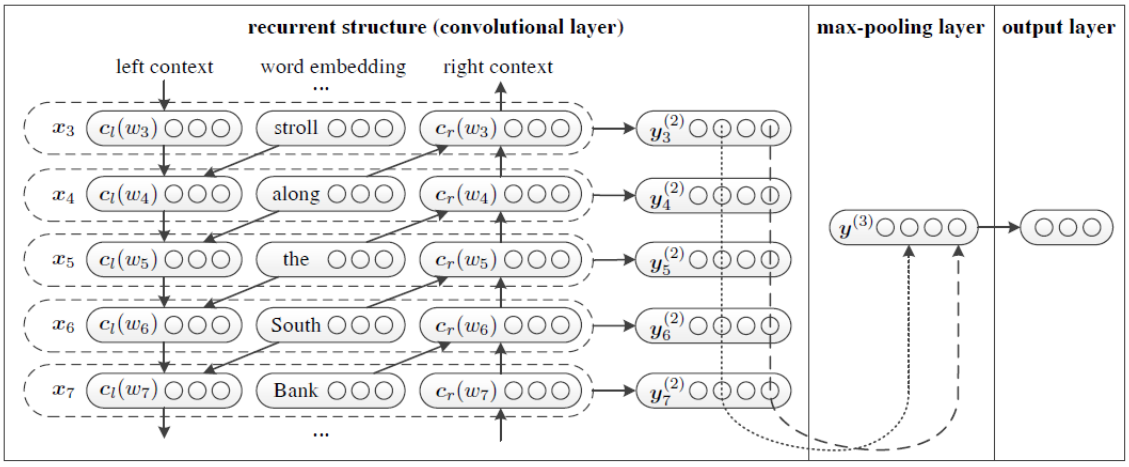


Figure 2.7: Structure of the recurrent convolutional neural network (RCNN). (*This figure is from [22]*).

In [22], Lai et al. proposed a recurrent convolutional neural network (RCNN) that combines a recurrent structure and a max-pooling layer to extract useful features from variable-length text documents. As shown in figure 2.7, a bi-directional recurrent structure is used to recursively encode context information around each word in the text document. The left and right context vectors are encoded using the following formula:

$$c_l(w_i) = f(W^{(l)}c_l(w_{i-1}) + W^{sl}e(w_{i-1}))$$

$$c_r(w_i) = f(W^{(r)}c_r(w_{i+1}) + W^{sr}e(w_{i+1}))$$

where $c_l(w_i)$ and $c_r(w_i)$ are the left and right context vectors of word w_i, $e(w_i)$ denotes the word embedding of word w_i and $W^{(l)}$, $W^{(r)}$, W^{sl} and W^{sr} are weights to be learned. The corresponding left-context, center word and the right context vectors are concatenated to generate the context representation x_i for each center word. Then each of the context representations is multiplied with a weight matrix and fed into a *tanh* activation function to generate a latent semantic vector y_i. After that, a max-pooling layer is used to select useful features element-wise from the latent semantic vectors. Based on the selected features, a fully connected layer is used to generate outputs. They called the recurrent structure the convolutional layer though there is no convolution operation performed.

In [20], Johnson et al. used a bi-directional LSTM to capture the forward and

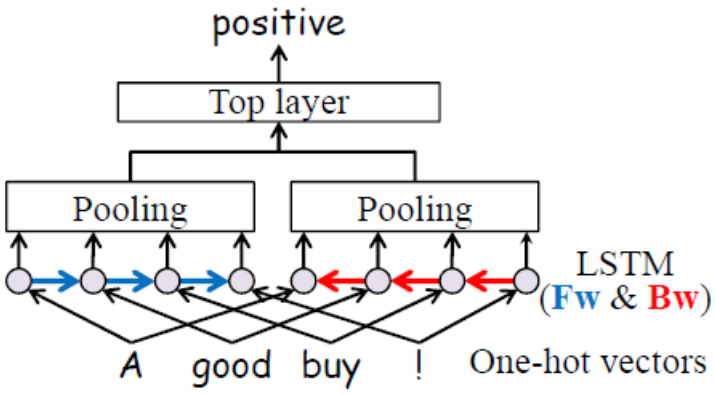


Figure 2.8: Structure of the one-hot bidirectional LSTM with pooling model (oh-2LSTMp). (*This figure is from [20]*).

backward context of a text sequence. In their model, as shown in figure 2.8, they removed the embedding layer and directly built the LSTMs on one-hot vectors. The pooling layers then pool over all the intermediate outputs of LSTMs. After pooling, a fully connected layer is then applied to generate outputs.

Instead of learning model directly from labeled data, Dai et al. [8] used a semi-supervised approach to first unsupervisedly train a LSTM model for the language modeling task, which is a task to predict the next word in natural language text sequence based on the previous words, and then they fine-tune the pre-trained LSTM on the text classification datasets. In this way, they can make use of the enormous amount of unlabeled data to help the model start from a better initialization point.

3 Context-FOFE Encoding Scheme

Most of the machine learning algorithms need to have fixed-size inputs. However, natural language sentences are usually of variable length. Some models try to handle this problem by truncating the long sentences and padding the short sentences [17, 44], which is unnatural and suffers from losing information. Bag of words (BoW) representation is a simple and commonly used method that can be used to convert variable-length text into a fixed-size vector, nevertheless, this representation discards the most important word ordering information in the text, thus limiting the system performance.

Moreover, one of our goals in this work is to design a representation method for variable-length text documents that can encode their informative contexts. The context patterns, according to our assumption stated in section 1.1, are very good features for text classification. However, these context patterns are hidden within text documents and need to be discovered by classification models through a learning process. In this sense, our desired representation is better to be simple but

informative.

To meet these demands, we propose a recursive encoding scheme that can convert variable-length text into fixed-size representation while retaining the word ordering information. Furthermore, the encoded representation is unique and reversible for every piece of text. We call this encoding scheme Context-FOFE, for it is based on the Fixed-size Ordinally Forgetting Encoding (FOFE) scheme [41, 42], and focuses on keeping all the context (word ordering) information.

In the following sections, we will first introduce the FOFE encoding scheme, which forms the basis of our context encoding scheme; then we will describe how our context-FOFE encoding scheme encodes context information; lastly we will present an efficient implementation of our encoding scheme.

3.1 The FOFE Encoding Scheme

FOFE is a simple recursive encoding scheme that can almost uniquely encode any sequence of words (or discrete symbols), with their ordering information, into a fixed-size representation. FOFE memorizes the word order in a text sequence using a simple ordinally-forgetting mechanism that relates weights of the words with their positions in the sequence. In the papers [41, 42], Zhang et al. have shown that FOFE has the equivalent ability as RNNs to encode historical information.

And moreover, since the encoding is based on a simple recursive formula, there is no parameter needs to be learned.

FOFE's ordinally-forgetting mechanism works as follows. Given a sequence of words $S = \{w_1, w_2, \ldots, w_T\}$, each word w_t is first represented as a 1-of-V one-hot vector e_t, where $t(1 \leq t \leq T)$ denotes the sequential reading step and V denotes the vocabulary size. FOFE then encodes the sequence word by word from the beginning to the end of the sequence based on the following simple recursive formula (with $Z_0 = \mathbf{0}$):

$$Z_t = \alpha \cdot Z_{t-1} + e_t \tag{3.1}$$

Where Z_t denotes the FOFE code for the partial sequence up to word w_t and $\alpha(0 < \alpha < 1)$ is a constant forgetting factor that controls the contribution of historical information to the current encoding. The constant α is called the forgetting factor because the weights of the previous words are gradually reduced by α as the reading moves forward. By using such an ordinally-forgetting mechanism, FOFE efficiently relates the weights of words with their positions in the sequence. According to the weights in the encoded representation, it is also possible to revert the FOFE code back to the original word sequence.

Figure 3.1 shows examples of using FOFE to encode word sequence. In the figure, the table to the left contains the vocabulary list and the corresponding one-

WORD	ONE-HOT
w_0	1000000
w_1	0100000
w_2	0010000
w_3	0001000
w_4	0000100
w_5	0000010
w_6	0000001

PARTIAL SEQUENCE	FOFE CODE
[]	0,0,0,0,0,0,0
w_6	0,0,0,0,0,0,1
w_6, w_4	$0,0,0,0,1,0,\alpha$
w_6, w_4, w_5	$0,0,0,0,\alpha,1,\alpha^2$
w_6, w_4, w_5, w_0	$1,0,0,0,\alpha^2,\alpha,\alpha^3$
w_6, w_4, w_5, w_0, w_5	$\alpha,0,0,0,\alpha^3,1+\alpha^2,\alpha^4$
$w_6, w_4, w_5, w_0, w_5, w_4$	$\alpha^2,0,0,0,1+\alpha^4,\alpha+\alpha^3,\alpha^5$

Figure 3.1: Illustration of the FOFE encoding scheme.

hot vectors of each word. The one-hot vectors are of vocabulary size, with each word corresponding to a different position in the vector. The table to the right shows the FOFE code for each intermediate partial sequence. Each time a new word is read in, the FOFE code is first multiplied with the forgetting factor α, and then added with the corresponding one-hot vector of the read-in word. Following this procedure, the sentence "to be or not to be", with a vocabulary list as {"to", "be", "or", "not"}, is represented in FOFE as "$[\alpha+\alpha^5, 1+\alpha^4, \alpha^3, \alpha^2]$". Comparing with its bag-of-words representation "$[2, 2, 1, 1]$", which we have shown in section 1.1, obviously the FOFE code can better represent the sentence as its weights reflect the word orders in the sentence.

The FOFE code for any sequence of words is almost unique, given the vocabulary is sufficiently large to include all possible words in the text sequences.

Here the unique encoding means a one-one mapping between text sequences and FOFE codes. That is, for any two different text sequences, they will have different FOFE codes; for any sequence of words, given its FOFE code, the vocabulary list and forgetting factor used, it should be able to unambiguously recover the original sequence. The uniqueness property of FOFE encoding has been proven both theoretically and experimentally. Theoretically, when $0 < \alpha \leq 0.5$, according to the formula of the sum of geometric series, we have:

$$\sum_{i=1}^{t} \alpha^i = \frac{\alpha(1-\alpha^t)}{1-\alpha} < \lim_{t\to\infty} \frac{\alpha(1-\alpha^t)}{1-\alpha} = \frac{\alpha}{1-\alpha} \leq 1$$

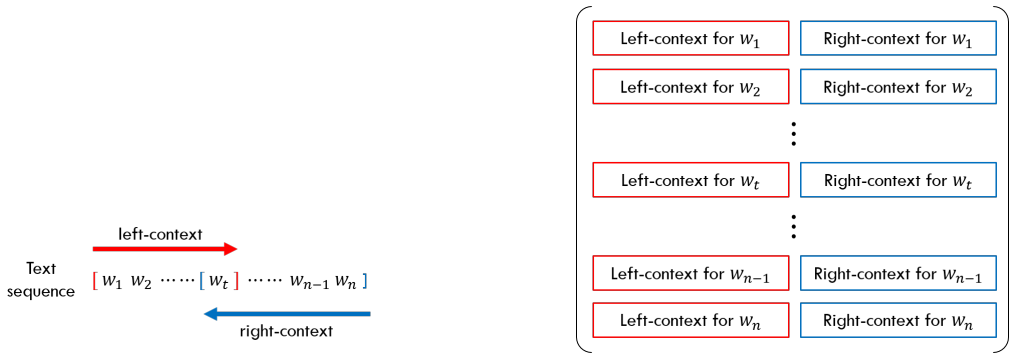
Hence at any intermediate time step, there can be only one weight that is greater than or equal to 1, which corresponds to the word that just read in. Based on this fact, it is easy to decode the FOFE code to the original word sequence; just reverse the encoding procedure. Experimentally, the authors have tested all the possible sequences up to 20 words for number of collisions. A collision is defined as the event in which the maximum element-wise difference between two FOFE codes is less than a set small error ϵ. In the experiments, the authors found that the number of collisions is extremely small, thousands of collisions out of 10^{20} tested sequences, even with no restriction on the possible word sequences. In real-world situation, with the syntactic and semantic restrictions of natural language, the number of collisions would be much less. This phenomenon can be heuristically explained by

the sparsity of natural language. Here the sparsity of natural language refers to the facts that a piece of natural language text usually only uses a small portion of the vocabulary, and normally there is very few sentences in a text document having exactly the same sequence of words. In this case, we can conclude that the FOFE encoding scheme can encode any sequence of words into almost unique representation, given a sufficiently large vocabulary.

3.2 The Context-FOFE Encoding Scheme

As introduced in section 3.1, FOFE can almost uniquely encode text of variable length into a fixed-size vector representation while keeping its word ordering information. Having all these exciting properties, we may think of using FOFE to encode text documents. Since FOFE already keeps the word ordering information of the text document, the context information should also have been encoded. This sounds reasonable in theory. However, in practice the FOFE code can only remember recent history because of the fact that weight α^t becomes insignificant as the power term t gets larger. Moreover, FOFE puts more weight on the later words of a text sequence because FOFE's encoding process is a recursive process in which weights of the previous words are decreased each time a new word is read in. This biased weight distribution will let the learning model unnecessarily focus on the

later part of the text document.



(a) (b)

Figure 3.2: Context-FOFE encoding scheme. (a) Context around word w_t. Words are represented as one-hot vectors. (b) Context matrix. The context vectors are of the same size.

To overcome this difficulty when encoding long text sequences like documents, and to fully encode context information with unbiased focus, we propose a variant of the FOFE encoding scheme, namely, the Context-FOFE encoding scheme. This scheme encodes the context information of a text document into a matrix. As shown in figure 3.2a, the context around word w_t is divided into a left-context and a right-context, where the left-context denotes the forward word sequence from w_1 to w_t, and the right-context denotes the backward word sequence from w_n to w_t. We use FOFE to encode the left-context and the right-context, respectively. The encoded left and right context vectors are then concatenated horizontally to

generate a row vector that represents the context around the center word. For each word in a text document, we can generate a row vector. And by concatenating these row vectors vertically according to the word order, we can form a matrix of size $N \times 2V$, where N is the length of the text document and V is the vocabulary size, as shown in figure 3.2b. We call this matrix as context matrix and use it to represent the corresponding text document.

As a simple encoding scheme that can uniquely encode context information of text documents into a matrix containing fixed-size vectors, our context-FOFE keeps all the merits of the FOFE encoding scheme. Firstly, our context-FOFE encoding scheme enhances FOFE to have complete uniqueness and reversibility on the encoded representation, under the same condition of having a sufficiently large vocabulary. Since our matrix representation of text document consist of FOFE codes for all the word positions, that is, all the partial word sequences, it is very easy to revert back to the original text sequence. And because for any encoded representation, we can unambiguously recover the original text sequence, we say our method can encode any text document into completely unique representation. Secondly, our encoded representations of text documents are generally applicable to many machine learning models that require fixed-size inputs. Our matrix representation of text document comprises fixed-size vectors; each one of these fixed-size

vectors can be fed as input to many machine learning models. Thirdly, the same as FOFE, our encoding process is a simple recursive process that is only controlled by a single hyper-parameter called forgetting factor. This makes our method an efficient and fast encoding method.

Furthermore, our context-FOFE encoding scheme has the following extra benefits. Firstly, our method solves the biased focus problem arose when encoding long text sequences with FOFE. Instead of sampling context patterns once only with a focus on the latter words, we sample context patterns at every word position and hence every position in the document is focused on. Secondly, The context patterns sampled at every word position can provide slightly different views of a document, which can potentially make the learning of context patterns easier. Finally, by sampling the slightly different views of a document at each word position, our method effectively augments the existing training data to provide more data for training.

3.3 Efficient Implementation of Context-FOFE

Although our context-FOFE encoding scheme is based on a recursive encoding formula, in implementation we don't have to follow the recursive process iteratively to encode text documents. Note that our context-FOFE encoding only has one single parameter, the forgetting factor α; and according to the ordinally-forgetting

mechanism, weights of words are gradually reduced by multiplying with α. We can know that the partial weight of a word resulting from its appearance t positions before is α^t. Based on this fact, we can first construct a weight matrix containing positional weights for all partial sequences, as:

$$\begin{bmatrix} 1 & & & & \\ \alpha & 1 & & & \\ \alpha^2 & \alpha & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ \alpha^{L-1} & \alpha^{L-2} & \alpha^{L-3} & \cdots & 1 \end{bmatrix} \tag{3.2}$$

where L denotes length of the text document. Note that this is only a weight matrix and does not involve any particular word sequence. Hence this matrix can be generally constructed according to the length of the text sequence. Then to encode a text sequence, we can just multiple this weight matrix with the matrix of

one-hot vectors of the text sequence:

$$\textit{left context} = \begin{bmatrix} 1 & & & & \\ \alpha & 1 & & & \\ \alpha^2 & \alpha & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ \alpha^{L-1} & \alpha^{L-2} & \alpha^{L-3} & \cdots & 1 \end{bmatrix} \times [\text{rows of 1-hot vectors}] \tag{3.3}$$

$$\textit{right context} = \begin{bmatrix} 1 & & & & \\ \alpha & 1 & & & \\ \alpha^2 & \alpha & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ \alpha^{L-1} & \alpha^{L-2} & \alpha^{L-3} & \cdots & 1 \end{bmatrix}^T \times [\text{rows of 1-hot vectors}] \tag{3.4}$$

$$\tag{3.5}$$

where the one-hot vectors should be in the same order as the corresponding words in the text sequence.

Noticing that weights α^t are gradually reduced and become insignificant when t gets larger, we can obtain even better efficiency both on speed and memory by removing insignificant weights. To do this, we can set a threshold to prune all

weights less than the threshold in the weight matrix, as:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \alpha & 1 & 0 & 0 \\ \alpha^2 & \alpha & 1 & 0 \\ \alpha^3 & \alpha^2 & \alpha & 1 \end{bmatrix} \underset{pruned}{\Longrightarrow} \begin{bmatrix} 1 & 0 & 0 & 0 \\ \alpha & 1 & 0 & 0 \\ 0 & \alpha & 1 & 0 \\ 0 & 0 & \alpha & 1 \end{bmatrix} \tag{3.6}$$

assuming α^2 is below the threshold. Clearly, this pruning operation is equivalent to setting a context encoding window. For a commonly used forgetting factor $\alpha = 0.7$ and a threshold 0.001, the effective context window size is 20 including the center word, for either the left context or the right context.

4 Context-FOFE Based FNNs for Text Classification

In chapter 3 we presented our novel context-FOFE encoding scheme to convert text documents of variable length into fixed-size representations that as well keep all the context information. Moreover, our context-FOFE representation has extra benefits like: uniqueness, reversibility and effectively providing more training data. In this chapter, we are going to design NN models to test our feature representation method on the text classification task. We decide to use the regular feed-forward neural network (FNN) rather than the recurrent neural network (RNN) because each row of the context-FOFE encoded representation has already captured the surrounding context information, hence no need for the internal memory of the previous context and the sequential reading of the text. Moreover, as we already know, FNNs have much simpler structure than RNNs and hence are much easier and faster to train.

Our context-FOFE encoding scheme encodes text document into a matrix that contains many fixed-size vectors, with each vector representing the context around a word. With these many fixed-size vectors in every document, naturally we can have two ways to train the NN model. The first way is to treat each vector representation in a document as an individual training instance and train on them independently; the second way is to treat the whole matrix representation as an individual training instance, and feed in the vector representations of all the word positions within the document as a whole for training. The NNs for the two ways of training share similar structure and the difference only occurs on aggregating position-wise hidden representations to a vector representation of the whole document.

4.1 Position-wise Trained Model

According to the context-FOFE encoding scheme, each word position generates a tuple of left-context FOFE code and right-context FOFE code. In position-wise training, each tuple of the context-FOFE codes is treated as a training instance. And the tuples are shuffled over the whole dataset. In this way, we mix up the relationships between adjacent tuples and view each tuple of the context-FOFE codes as an individual text sequence. This effectively provides more text sequences for training, even though many of them are similar.

Although the context-FOFE code has already kept all the context information necessary for training, the representation itself is sparse and high-dimensional. Moreover, there is no semantic relation between words. In this case, it is necessary to multiply the FOFE codes with a projection matrix to perform the dimension reduction. In addition, the projection matrix can be initialized by a word embedding matrix to further relate the words in the semantic space.

As shown in figure 4.1, the left-context FOFE codes and the right-context FOFE codes of a mini-batch of the training instances are first projected onto the semantic space by multiplying with a shared word embedding matrix. The shared word embedding matrix can be fine-tuned during training to better adapt to the specific dataset, and hence it can be viewed as part of the model weights. The left and right projected contexts are then concatenated horizontally, with the corresponding rows aligned, to form a $B \times 2D$ context matrix, which are the features of the training instances in the semantic space. After that the context matrix is fed into the regular FNN. Each layer of the FNN serves as a feature abstractor that abstracts the features to a higher level. We call the highly abstracted feature representations in the last hidden layer the hidden representations of the sample instances. On top of these highly abstracted representations of the training instances, we then use a

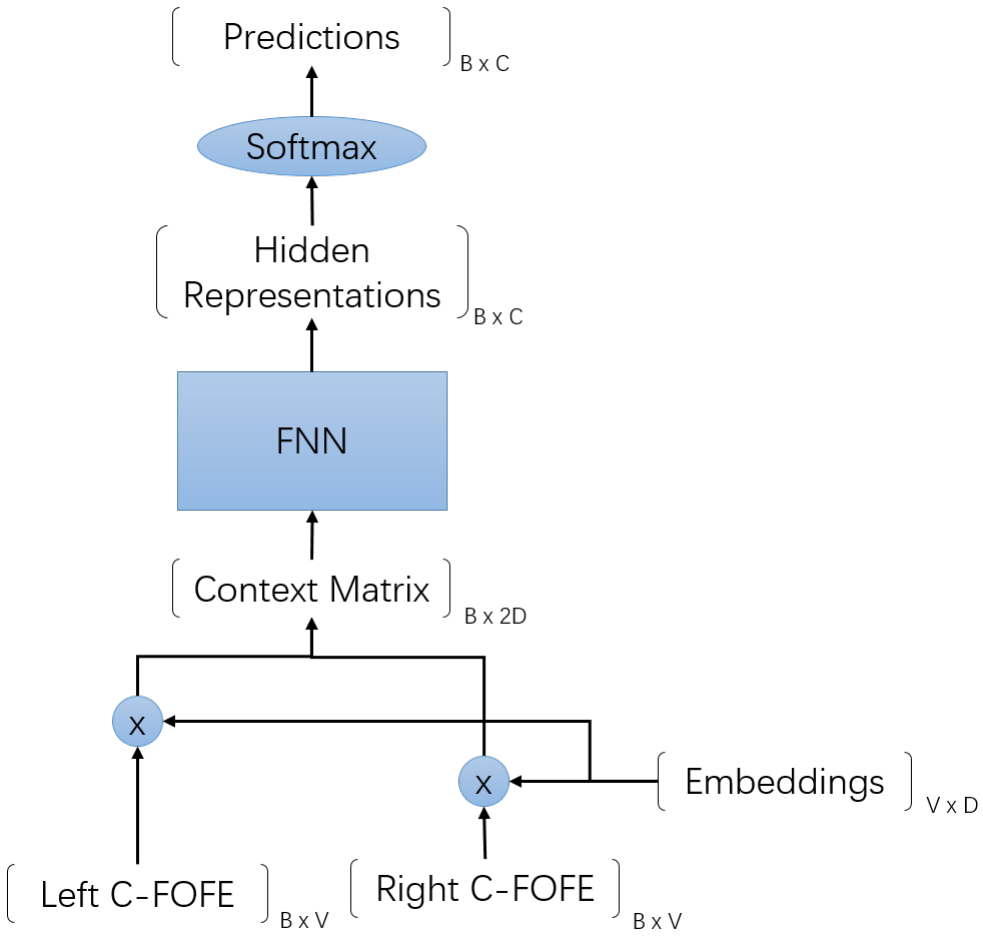


Figure 4.1: Structure of position-wise trained FNN. B denotes the mini-batch size; V denotes the vocabulary size; D denotes the word embedding dimension and C denotes the number of classification categories.

softmax layer to predict the labels of the training instances:

$$p_i = softmax(\boldsymbol{h}) = \frac{e^{h_i}}{\sum_k^C e^{h_k}} \quad \text{for } i = 1, \cdots, C \tag{4.1}$$

where h_i denotes the i-th element of the hidden representation, and p_i denotes the i-th element of the output vector. The softmax function normalizes the outputs to have a sum equal to 1, hence the outputs can be viewed as the predicted likelihood scores for each of the categories.

Note that we have labels for the training text documents but not for every word position. In this case, to obtain training labels for all training instances, we simply assign the document label to all the word positions within the document. This is reasonable because we assume context patterns are distinct in different text categories but similar in the same text category. Therefore training instances within a document probably have similar category label as the document. According to this labeling mechanism, the loss function for training is defined as the cross-entropy function over all the training instances:

$$L = -\frac{1}{N} \sum_{i=1}^{N} \boldsymbol{T}_i log(\boldsymbol{P}_i) \tag{4.2}$$

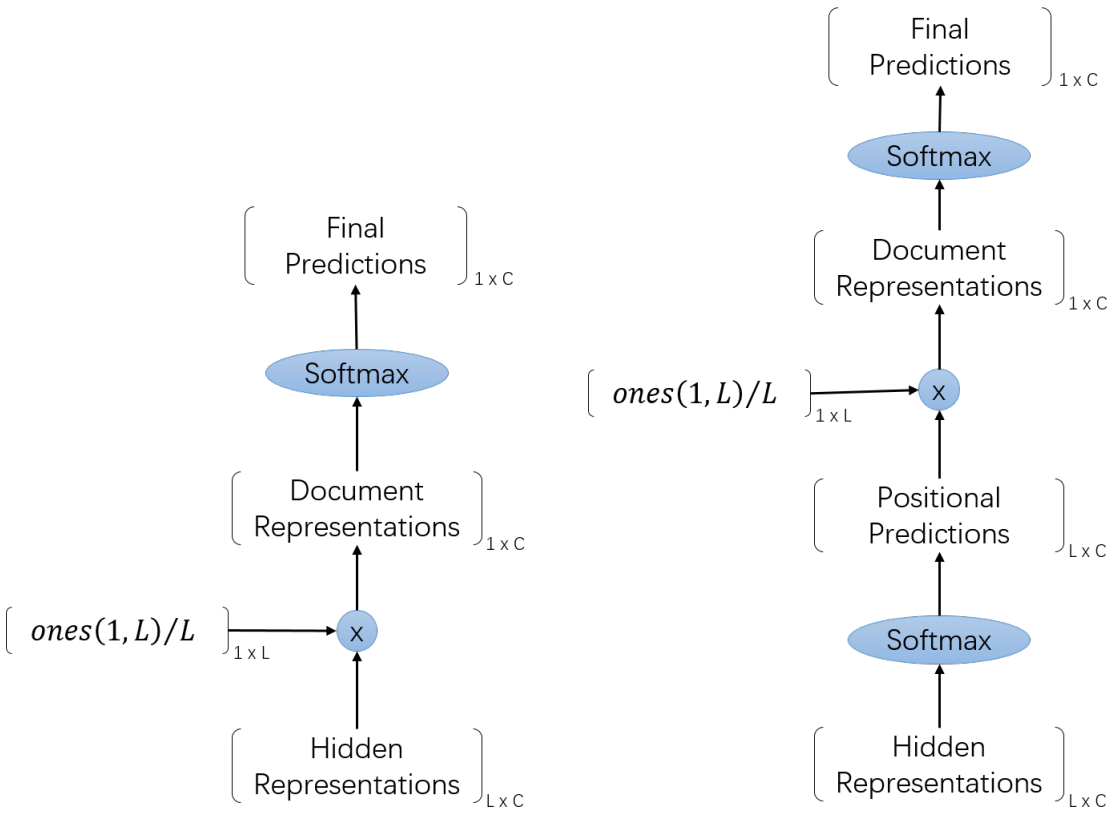
where N is the total number of word positions in the whole dataset, $\boldsymbol{T}_i$ is the label vector for the i-th training instance and $\boldsymbol{P}_i$ is the softmax output vector for the i-th training instance.

4.1.1 Voting strategy

Our position-wise FNN model is trained to predict the word positional labels. However, in testing we need to predict the document label. In order to obtain the document label, we need a mechanism to aggregate the information from word positions within the document. We call such a mechanism a voting strategy, for it is a mechanism for every word position to contribute their scores against each category to the final score.

Since we can feed all the word positions of a document into the trained model to produce the corresponding hidden representations, a very natural way to aggregate the position-wise information is to take the element-wise average of the hidden representations, and use the averaged hidden representation to represent the document, as shown in figure 4.2a. With the document hidden representation vector, it is then easy to generate the category predictions; just feed it into the softmax layer. We call this simple mechanism the averaged voting strategy.

Remember that the outputs of the softmax layer for each word position can be viewed as the predicted likelihood scores for each of the categories. With the likelihood scores from all the word positions, we can compute the document level likelihood scores for each of the categories, as an average of the position-wise likelihood scores. we call this mechanism the softmax-normed voting strategy. As shown



(a) averaged voting

(b) softmax-normed voting

Figure 4.2: Voting strategies for context-FOFE based FNNs. L denotes the document length.

in figure 4.2b, there are two softmax layers. The first one is to get the position-wise predicted likelihood scores, and the second one is to generate the final document level prediction scores. Comparing with the averaged voting, the softmax-normed voting can balance the contributions of word positions to the document level representation by normalizing the weights across categories on each position. But having two softmax layers stacked together makes it more complicated than the averaged voting.

4.2 Document-wise Trained Model

In the position-wise training, the model is trained with shuffled position-wise training instances but tested document-wise. This may lead to a mismatch between the loss function and the performance goal. To avoid this mismatch and gain potential performance improvement, we can train the model document-wise.

In document-wise training, each context-FOFE matrix as a whole is treated as a training instance. Hence the training data are shuffled on the document level instead of the word position level. Furthermore, as part of a whole, the context vectors of all the word positions within the document are needed to be fed into the network at the same time. Also, the back-propagated gradients of this training instance need to be averaged over all word positions.

Since now we directly predict the document label during training, the cross-entropy loss function is defined as:

$$L = -\frac{1}{D}\sum_{i=1}^{D} \boldsymbol{T}_i log(\boldsymbol{P}_i) \tag{4.3}$$

Where D is the total number of text documents in the whole dataset, $\boldsymbol{T}_i$ is the label vector for the i-th training document and $\boldsymbol{P}_i$ is the final softmax outputted likelihood score vector for the i-th training document.

As shown in figure 4.3, the model structure of the document-wise trained model is similar to the position-wise trained model. In the lower part of the model structure, the model also takes the left & right context-FOFE matrix as inputs. But the content of the matrices are different. In the position-wise trained model, the context-FOFE matrices comprise of a mini-batch of randomly shuffled position-wise context vectors; while in the document-wise trained model, the context-FOFE matrices are the original feature representation matrix of a document. The inputted context-FOFE matrices are first projected onto the semantic space with a shared word embedding matrix, as in the position-wise trained model. Then the projected semantic representations are aligned and concatenated horizontally to form the context matrix in the semantic space. This concatenated context matrix is then fed into the regular FNN to generate the abstracted hidden representations.

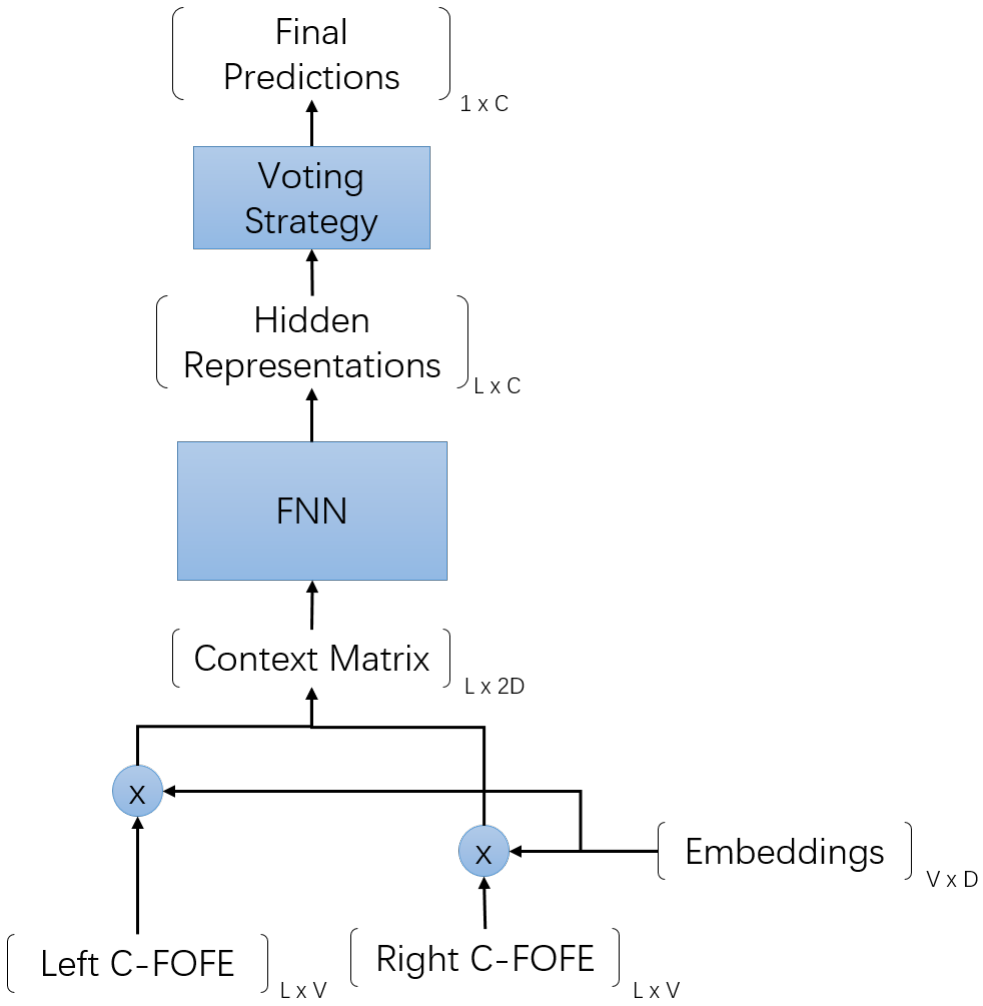


Figure 4.3: Structure of document-wise trained FNN. L denotes the document length; V denotes the vocabulary size; D denotes the word embedding dimension and C denotes the number of classification categories.

4.2.1 Voting strategy

Different from what we did in the position-wise trained model, in document-wise training, we also need to predict the document label in the training phase. In this case, we need to use the voting strategy also during training. We use the same two voting strategies, as described in section 4.1.1, on the generated hidden representations of all the word positions. Since the voting strategy is applied in the training phase, we also need to back-propagate the training errors through the voting layer. In this case, having two softmax layers stacked together makes the structure of the softmax-normed voting more complicated and thus harder to learn.

When testing, the context-FOFE matrix of a document is fed into the trained model. By simply going through the forward pass, the prediction of the given text document will then be generated at the output layer.

4.3 Bag-of-words Based FNN Model

In order to further investigate the effectiveness of our new context encoding scheme, we also designed a bag-of-words (BoW) based FNN model as a baseline for performance comparison. BoW [13] is a simple and commonly used representation for text documents. It keeps the words and their frequencies in a text document but doesn't record the context (word ordering) information.

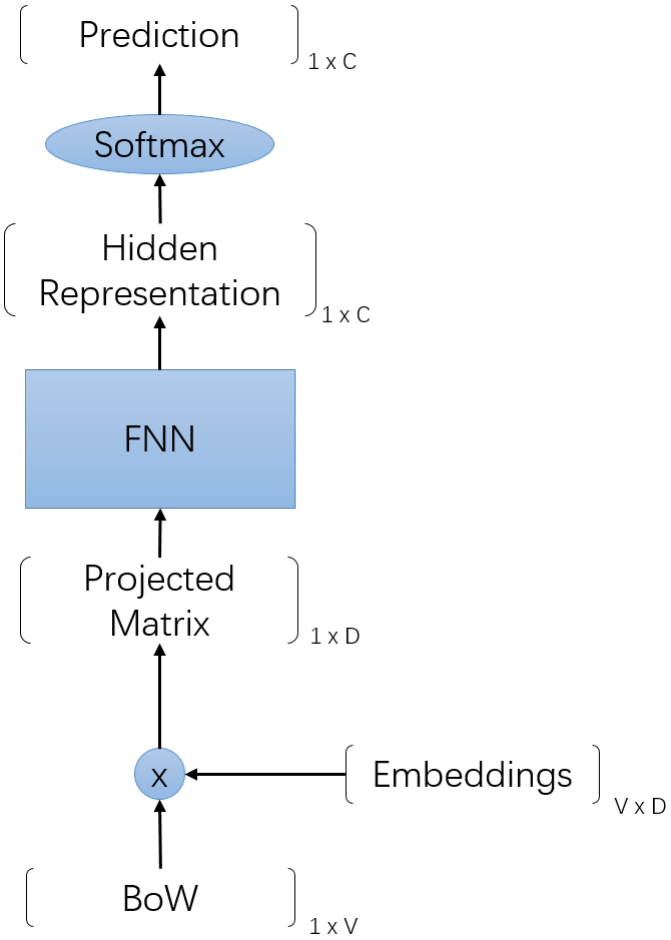


Figure 4.4: Structure of bag-of-words based FNN. V denotes the vocabulary size; D denotes the word embedding dimension and C denotes the number of classification categories.

We use a similar network structure as the context-FOFE based FNNs introduced in the previous sections and replace the context-FOFE encoded inputs with the BoW encoded inputs. As shown in figure 4.4, the inputted BoW representation of a document is first projected onto semantic space with a shared word embedding matrix. Then a multi-layer FNN is used on the projected representation to extract a hidden representation of the document. Based on the hidden representation, a softmax layer will then generate the prediction score for each of the categories. Note that BoW represents a document with a vector but not a matrix, no voting strategy is needed.

5 Context-FOFE Based HOPE Models for Text Modeling

Real-word data, like natural language text, are usually high-dimensional data. However, it is much harder to analyze data in high-dimensional space than in low-dimensional space. As the dimensionality increases, the volume of the space expands so fast that the available data becomes too sparse. The amount of data needed to obtain a statistically sound and reliable result often grows exponentially with the dimensionality. This difficulty is known as the curse of dimensionality [2]. To handle the curse of dimensionality, machine learning systems generally need to perform dimension reduction on the raw data to project the majority of signals to a lower-dimensional space; then to learn models based on the projected signals in the lower-dimensional space. These two stages are referred to as the feature extraction stage and the data modeling stage respectively.

There are many existing techniques for the feature extraction stage. Among

them, Principle Component Analysis (PCA) [33] is probably the most commonly used technique. PCA can be viewed as a linear orthogonal projection from the original data space to a lower-dimensional feature space such that the variance of the projected data is maximized. The intuition behind this is that the larger variation a dimension has, the more informative this dimension is, according to the information theory. Hence the projected dimensions are iteratively selected as the dimension that maximize the variance of the projected uncovered signals, up to the desired dimensionality M of the projected feature space. In practice, These projected dimensions can be represented by the eigenvectors of the covariance matrix that correspond to the M largest eigenvalues.

In the feature extraction stage, the process of dimension reduction often has the combined objectives as to: reduce the data dimension, remove random noise from data, select distinctive features and de-correlate features. However, in many of the traditional machine learning systems, the feature extraction stage and the data modeling stage are conducted independently, optimized based on rather different criteria. This may lead to a limitation in the data modeling performance. On the other hand, with the recent successes, neural network models that use the end-to-end learning strategy have demonstrated their capabilities to automatically extract and learn useful features in a single unified learning process. This suggests

a better way to perform feature extraction and data modeling for machine learning problems.

5.1 The HOPE Framework

Hybrid Orthogonal Projection and Estimation (HOPE) [43] is such a unified generative modeling framework for high-dimensional data. It is called unified because it embeds both the feature extraction and the data modeling stages in a typical machine learning process into one single framework. In such a way, these two stages can be jointly learned and optimized to better cope with high-dimensional real-world data.

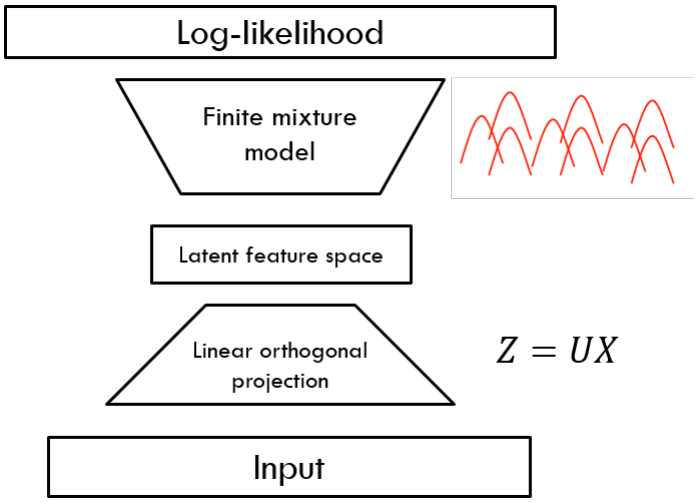


Figure 5.1: Structure of the HOPE framework.

The framework structure of HOPE is shown in figure 5.1. HOPE uses a simple linear orthogonal projection for the feature extraction stage. Similar to the PCA method, it projects data from the input data space to a lower-dimensional space that is called the latent feature space. Assuming that we have a full size $D \times D$ orthogonal matrix $\hat{\boldsymbol{U}}$, satisfying $\hat{\boldsymbol{U}}^T\hat{\boldsymbol{U}} = \hat{\boldsymbol{U}}\hat{\boldsymbol{U}}^T = \boldsymbol{I}$, a data sample $\boldsymbol{x}$ in the original D-dimensional data space can be decomposed based on all orthogonal row vectors of $\hat{\boldsymbol{U}}$, denoted as $\boldsymbol{u}_i$ with $i = 1, \ldots, D$, as follows:

$$\boldsymbol{x} = \underbrace{(\boldsymbol{x} \cdot \boldsymbol{u}_1)\,\boldsymbol{u}_1 + \cdots + (\boldsymbol{x} \cdot \boldsymbol{u}_M)\,\boldsymbol{u}_M}_{\textit{signal component}} + \underbrace{(\boldsymbol{x} \cdot \boldsymbol{u}_{M+1})\,\boldsymbol{u}_{M+1} + \cdots + (\boldsymbol{x} \cdot \boldsymbol{u}_D)\,\boldsymbol{u}_D}_{\textit{noise component}} \quad (5.1)$$

This decomposition can be viewed as a combination of a signal component and a noise component. Let $\boldsymbol{U}$ be an $M \times D$ matrix containing the first M row vectors and $\boldsymbol{V}$ be a $(D - M) \times D$ matrix containing all remaining row vectors, the orthogonal projection can also be viewed as a projection of the signals to an M-dimensional space plus a projection of the residual noise to the remaining $(D - M)$-dimensional space:

$$[\boldsymbol{z}; \boldsymbol{n}] = [\boldsymbol{U}; \boldsymbol{V}]\,\boldsymbol{x} = \hat{\boldsymbol{U}}\boldsymbol{x} \quad (5.2)$$

Where $\boldsymbol{z}$ is the projected signal component and $\boldsymbol{n}$ is the projected noise component.

One of the training goals is to learn a proper $\boldsymbol{U}$ that projects the data sample $\boldsymbol{x}$ onto a lower M-dimensional space while keeping as much signal as possible. This will be jointly learned with the statistical modeling stage. During training,

orthonormality constraint needs to be imposed on $\boldsymbol{U}$ to reduce feature correlation. If this projection is properly learned, it is reasonable to assume that the projected signal component $\boldsymbol{z}$, and the projected noise component $\boldsymbol{n}$, are independent in the latent feature space. Therefore, the probability distribution of the original data can be derived as:

$$p(\boldsymbol{x}) = p(\boldsymbol{z}) \cdot p(\boldsymbol{n}) \tag{5.3}$$

Based on this formula, the probability distribution of the original data can be modeled by modeling the probability distributions of the signal and noise components independently in lower-dimensional spaces.

While using a relatively simple feature extraction method, HOPE uses a powerful statistical model, namely the finite mixture model of the exponential family distributions, to model the projected signals.

$$p(\boldsymbol{z}) = \sum_{k=1}^{K} \pi_k \cdot f_k(\boldsymbol{z} \mid \boldsymbol{\theta}_k) \tag{5.4}$$

This is important since in most real-world applications, the projected signal component $\boldsymbol{z}$ still locates in a fairly high-dimensional space and real-world data tend to follow a rather complex distribution in high-dimensional spaces. Theoretically, a finite mixture model can approximate any arbitrary statistical distribution given a sufficiently large number of mixture components. On the other hand, HOPE

models the projected noise with an isotropic covariance Gaussian distribution:

$$p(\boldsymbol{n}) \sim \mathcal{N}(\boldsymbol{n} \mid \boldsymbol{0}, \sigma^2 \boldsymbol{I}) \tag{5.5}$$

Where σ^2 is a variance parameter to be learned from data.

HOPE is essentially a generative model that unifies the feature extraction stage and the data modeling stage into a single learning process. In this sense, as with many of other generative models, its model parameters, including both the projection matrix $\boldsymbol{U}$ and the mixture model, can be estimated based on the maximum likelihood criterion. This enables the unsupervised learning mode of the HOPE model. Assuming a training set is available as $\boldsymbol{X} = \{\boldsymbol{x}_n \mid n = 1, \ldots, N\}$, the joint log-likelihood function related to all HOPE parameters, including the projection matrix $\boldsymbol{U}$, the mixture model $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_k \mid k = 1, \ldots, K\}$ and the residual noise variance σ^2, can be expressed as follows:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{U}, \boldsymbol{\Theta}, \sigma^2 \mid \boldsymbol{X}) &= \sum_{n=1}^{N} \ln p(\boldsymbol{x}_n) = \sum_{n=1}^{N} \Big[\ln p(\boldsymbol{z}_n) + \ln p(\boldsymbol{n}_n) \Big] \\
&= \underbrace{\sum_{n=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \cdot f_k(\boldsymbol{U}\boldsymbol{x}_n \mid \boldsymbol{\theta}_k) \right)}_{\mathcal{L}_1(\boldsymbol{U}, \boldsymbol{\Theta})} + \underbrace{\sum_{n=1}^{N} \ln \Big(\mathcal{N}(\boldsymbol{n}_n \mid \boldsymbol{0}, \sigma^2 \boldsymbol{I}) \Big)}_{\mathcal{L}_2(\boldsymbol{U}, \sigma^2)}
\end{aligned} \tag{5.6}$$

The learning process, which is to estimate a set of HOPE parameters $\{\boldsymbol{U}^*, \boldsymbol{\Theta}^*, \sigma^{2*}\}$

that maximize the above joint log-likelihood function, can be formulated as:

$$\{\boldsymbol{U}^*, \boldsymbol{\Theta}^*, \sigma^{2^*}\} = \underset{\boldsymbol{U},\boldsymbol{\Theta},\sigma^2}{\arg\max}\ \mathcal{L}(\boldsymbol{U}, \boldsymbol{\Theta}, \sigma^2 \mid \boldsymbol{X}) \tag{5.7}$$

subjecting to an orthogonal constraint:

$$\boldsymbol{U}\boldsymbol{U}^T = \boldsymbol{I} \tag{5.8}$$

For computational efficiency, the orthogonal constraint can be cast as a penalty term to the objective function. Hence the above constrained optimization problem is converted into an unconstrained one as follows:

$$\{\boldsymbol{U}^*, \boldsymbol{\Theta}^*, \sigma^{2^*}\} = \underset{\boldsymbol{U},\boldsymbol{\Theta},\sigma^2}{\arg\max}\ \Big[\mathcal{L}(\boldsymbol{U}, \boldsymbol{\Theta}, \sigma^2 \mid \boldsymbol{X}) - \beta \cdot \mathcal{D}(\boldsymbol{U})\Big] \tag{5.9}$$

Where β ($\beta > 0$) is a contribution control parameter for the penalty term, and the penalty term $\mathcal{D}(\boldsymbol{U})$ is a differentiable function defined as:

$$\mathcal{D}(\boldsymbol{U}) = \sum_{i=1}^{M} \sum_{j=i+1}^{M} \frac{|\boldsymbol{u}_i \cdot \boldsymbol{u}_j|}{|\boldsymbol{u}_i| \cdot |\boldsymbol{u}_j|} \tag{5.10}$$

In implementation, the norms of all row vectors of $\boldsymbol{U}$ are normalized to one after each update.

However, since the signal component $\boldsymbol{z}$ follows a mixture distribution, no closed-form solution is available for both the projection matrix $\boldsymbol{U}$ and the mixture model parameters $\boldsymbol{\Theta}$. In this case, iterative optimization algorithms like stochastic gradient descent (SGD), can be used to jointly learn the HOPE parameters by maximizing the joint log-likelihood function. Algorithm 1 shows the pseudo code for the

Algorithm 1 SGD-based Maximum Likelihood Learning Algorithm for HOPE

randomly initialize $\mathbf{u}_i$ $(i = 1, \cdots, M)$, π_k and $\boldsymbol{\theta}_k$ $(k = 1, \cdots, K)$

for *epoch* $= 1$ **to** T **do**

for *minibatch* $\mathbf{X}$ **in** training set **do**

$$\mathbf{U} \leftarrow \mathbf{U} + \epsilon \cdot \left(\frac{\partial \mathcal{L}_1(\mathbf{U}, \boldsymbol{\Theta})}{\partial \mathbf{U}} + \frac{\partial \mathcal{L}_2(\mathbf{U}, \sigma)}{\partial \mathbf{U}} - \beta \cdot \frac{\partial \mathcal{D}(\mathbf{U})}{\partial \mathbf{U}} \right)$$

$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_k + \epsilon \cdot \frac{\partial \mathcal{L}_1(\mathbf{U}, \boldsymbol{\Theta})}{\partial \boldsymbol{\theta}_k} \quad (\forall k)$$

$$\pi_k \leftarrow \pi_k + \epsilon \cdot \frac{\partial \mathcal{L}_1(\mathbf{U}, \boldsymbol{\Theta})}{\partial \pi_k} \quad (\forall k)$$

$$\sigma^2 \leftarrow \frac{1}{N(D-M)} \sum_{n=1}^{N} \mathbf{n}_n^T \mathbf{n}_n$$

$$\pi_k \leftarrow \frac{\pi_k}{\sum_j \pi_j} \quad (\forall k) \quad \text{and} \quad \mathbf{u}_i \leftarrow \frac{\mathbf{u}_i}{|\mathbf{u}_i|} \quad (\forall i)$$

end for

end for

SGD-based maximum likelihood learning algorithm. In each training step, the partial derivatives of the log-likelihood function $\mathcal{L}(\boldsymbol{U}, \boldsymbol{\Theta}, \sigma^2 \mid \boldsymbol{X})$ with respect to the projection matrix $\boldsymbol{U}$, the mixture model parameters $\boldsymbol{\theta}_k$'s and π_k's, are computed and updated to the parameters with a learning rate ϵ.

5.2 Comparative HOPE Models for Text Classification

The HOPE model, as introduced in the above section, is essentially a generative model that models the data distribution but not the separation boundaries. There-

fore we cannot perform the text classification task directly on a HOPE model. In this case, we propose to use a comparative evaluation strategy. That is, we train a group of HOPE models, with each of them specializes in modeling a category of data, and then classify new data based on the likelihood scores given from each of the HOPE models.

We train each HOPE model based on the maximum likelihood estimation (MLE) criterion. Before training, we first divide the labeled training data into subsets according to their category labels: $\boldsymbol{X}_c = \{\boldsymbol{x}_i \mid \boldsymbol{x}_i \in \boldsymbol{X},\ \mathit{label}(\boldsymbol{x}_i) = c\}$ $(c = 1, \ldots, C)$, where $\boldsymbol{X}_c$ is a set of all the data samples in category c, and C is the total number of categories. Then for each category, we train a HOPE model to model the distribution of data samples within this category by estimating a set of model parameters that maximizes the log-likelihood function:

$$\{\boldsymbol{U}_c^*, \boldsymbol{\Theta}_c^*, {\sigma^2}_c^*\} = \underset{\boldsymbol{U},\boldsymbol{\Theta},\sigma^2}{\arg\max} \left[\mathcal{L}(\boldsymbol{U}, \boldsymbol{\Theta}, \sigma^2 \mid \boldsymbol{X}_c) - \beta \cdot \mathcal{D}(\boldsymbol{U})\right] \tag{5.11}$$

In this way, finally we will have C number of well-trained HOPE models that each of them specializes in modeling the distributions of context patterns within one particular category.

We follow the context-FOFE encoding scheme, as proposed in chapter 3, to represent each text document as a context-FOFE matrix. During training, we treat each row of these matrices as a training instance because each row already has the

context information, i.e., composition patterns of words and phrases that we want to model. Moreover, to introduce semantic relations between words, we can multiply the context-FOFE codes with a word embedding matrix before feeding them into the HOPE model for training. However, due to the orthonormality constraint on the projection matrix $\boldsymbol{U}$, the word embedding matrix needs to be fixed during training. In implementation, after multiplying with the word embedding matrix, we also normalize the whole input dataset to have zero mean and unit variance on each element-wise dimension, which helps to get more balanced values along the origin after the orthogonal projection.

In our implementation, we follow [43] to use a mixture of the von Mises-Fisher (vMF) distributions to model the signal component $\boldsymbol{z}$:

$$p(\boldsymbol{z}) = \sum_{k=1}^{K} \pi_k \cdot f_k(\boldsymbol{z} \mid \boldsymbol{\theta}_k) = \sum_{k=1}^{K} \pi_k \cdot C_M(|\boldsymbol{\mu}_k|) \cdot e^{\boldsymbol{z} \cdot \boldsymbol{\mu}_k} \tag{5.12}$$

In the above formula, $\boldsymbol{\theta}_k$ denotes the model parameters of the k-th vMF component, $\boldsymbol{\mu}_k$ is the mean direction of the k-th vMF component and $C_M(\kappa)$ is the probability normalization term of the vMF distribution defined as:

$$C_M(\kappa) = \frac{\kappa^{M/2-1}}{(2\pi)^{M/2} I_{M/2-1}(\kappa)} \tag{5.13}$$

where $I_v(\cdot)$ denotes the modified Bessel function of the first kind at order v. The mixture of vMF distributions can be viewed as a generalized finite Gaussian mixture

model over a high-dimensional spherical surface. Therefore the projected signals $\boldsymbol{z}$ need to be located on the surface of an M-dimensional sphere, i.e., $|\boldsymbol{z}| = 1$. In this case, $\boldsymbol{z}$ needs to be normalized to have unit length each time after the orthogonal projection.

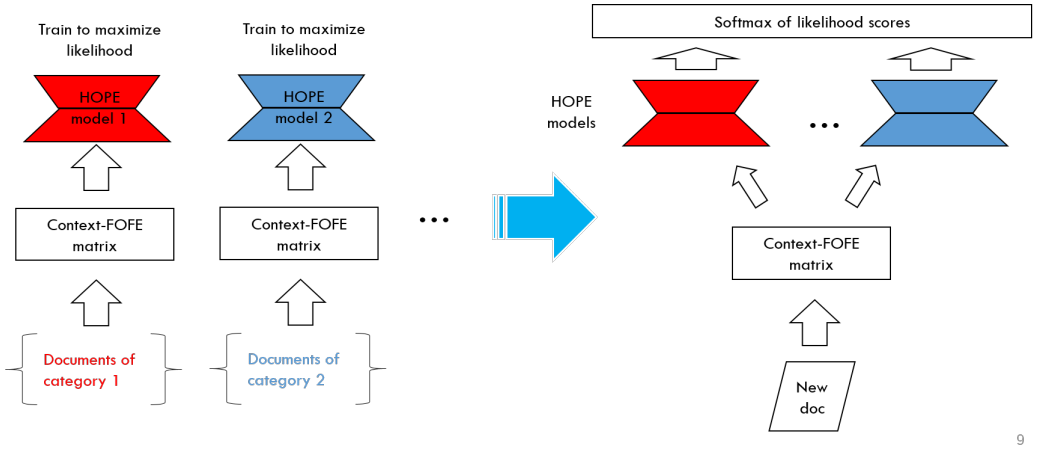


Figure 5.2: The comparative evaluation strategy of HOPE models.

After training a HOPE model for each category, we have a group of "experts" $\{\{\boldsymbol{U}_c^*, \boldsymbol{\Theta}_c^*, \sigma^{2*}_c\} \mid c = 1, \ldots, C\}$ that each of them captures the distribution of a different category of data. When testing, we employ the comparative evaluation strategy. Given a text document $\boldsymbol{y}$, the document is fed into each of the models and each model will then produce a log-likelihood score, according to the following

formula:

$$\mathcal{L}(\boldsymbol{U}_c^*, \boldsymbol{\Theta}_c^*, {\sigma^2}_c^* \mid \boldsymbol{y}) = \sum_{n=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \cdot C_M(|\boldsymbol{\mu}_k|) \cdot e^{\boldsymbol{U} \cdot \boldsymbol{y} \cdot \boldsymbol{\mu}_k} \right) + \sum_{n=1}^{N} \ln \left(\mathcal{N}(\boldsymbol{n}_n \mid \boldsymbol{0}, \sigma^2 \boldsymbol{I}) \right) \tag{5.14}$$

where N now is the length of the text document $\boldsymbol{y}$. The log-likelihood scores reflect how similar the distribution of the testing document $\boldsymbol{y}$ is to each category. Nevertheless, we need to note that, in order to maintain a fair comparison of the log-likelihood scores, all the HOPE models need to be trained to a similar level. Based on these log-likelihood scores, the document $\boldsymbol{y}$ is then classified to the category that has the highest score:

$$\mathit{label}(\boldsymbol{y}) = \arg\max_{c} \mathcal{L}(\boldsymbol{U}_c^*, \boldsymbol{\Theta}_c^*, {\sigma^2}_c^* \mid \boldsymbol{y}) \tag{5.15}$$

that is, the category that the context pattern distribution of the testing document is most similar to.

6 Experiments

In this chapter, we present experiments conducted on a document classification task and a sentiment analysis task with our proposed models. The chapter is divided into five sections, in which section 6.1 introduces the common setups for the experiments, sections 6.2, 6.3 and 6.4 examine some important parameters for each of the proposed models, and finally section 6.5 presents our best models and the performance comparison with existing models.

6.1 Experimental Setup

We use the following datasets in the experiments:

- **20NG**[1]**:** This is the 20 Newsgroups dataset [23]. It is a popular dataset for text classification and text clustering tasks. We use the "bydate" version of the dataset. This dataset is a collection of 18846 newsgroup documents,

[1]The dataset is available from http://qwone.com/ jason/20Newsgroups/

partitioned nearly evenly across 20 different newsgroups. Table 6.1 shows the summary of the dataset. As we can see, some of the newsgroups are very closely related to each other, while some of the others are highly unrelated.

- **IMDB[2]:** The IMDB movie review dataset [28] is a popular dataset for binary sentiment classification task. The dataset consists of 50,000 movie reviews from IMDB, allowing no more than 30 reviews per movie. The dataset uses 25,000 highly polar samples for training and the other 25,000 samples for testing. The training set and testing set are equally divided into two classes: *positive* and *negative*. The dataset also provides 50,000 unlabeled samples for unsupervised learning.

Table 6.1: Summary of the 20 Newsgroups dataset

Topic	**Newsgroup**	**#Train document**	**#Test document**
comp	comp.graphics	584	389
	comp.os.ms-windows.misc	591	394
	comp.sys.ibm.pc.hardware	590	392
	comp.sys.mac.hardware	578	385
	comp.windows.x	593	395

Continued on next page

[2]The dataset is available from http://ai.stanford.edu/ amaas/data/sentiment/

Table 6.1 – *Continued from previous page*

	sum	2936	1955
politics	talk.politics.misc	465	310
	talk.politics.guns	546	364
	talk.politics.mideast	564	376
	sum	1575	1050
rec	rec.autos	594	396
	rec.motorcycles	598	398
	rec.sport.baseball	597	397
	rec.sport.hockey	600	399
	sum	2389	1590
religion	talk.religion.misc	377	251
	alt.atheism	480	319
	soc.religion.christian	599	398
	sum	1456	968
sci	sci.crypt	595	396
	sci.electronics	591	393
	sci.med	594	396

Continued on next page

Table 6.1 – *Continued from previous page*

	sci.space	593	394
	sum	2373	1579
misc	misc.forsale	585	390

We perform document classification task on the 20 Newsgroups dataset and sentiment analysis task on the IMDB dataset. For the 20 Newsgroups dataset, in addition to testing on the original 20 categories division, we also follow [14, 22] to test on the first four major categories (comp, politics, rec, and religion) as shown in table 6.1. For all our experiments on the IMDB dataset, only labeled data are used.

To be comparable with existing works, the performance of the 20NG(4 categories) task is evaluated based on the Macro-F1 score, which is the average of the F1 scores for each category. All the other tasks in this work are evaluated based on the classification accuracy, which is the number of correctly classified documents over the total number of documents.

We use three different sets of word embedding vectors in the experiments:

- **gloveVec-300D**[3]**:** These word embeddings are 300-dimensional pre-trained word vectors obtained from the GloVe model [34]. The word vectors are

[3]The pre-trained word embedding is available from https://nlp.stanford.edu/projects/glove/

trained on a Common Crawl corpus that contains 840 billion tokens.

- **SWE-300D:** These word embeddings are 300-dimensional vectors trained with the Semantic Word Embedding (SWE) model [27]. The word vectors are trained with the default model setting on enwik9, a corpus that consists of the first one billion characters from Wikipedia.

- **SWE-50D:** These word embeddings are 50-dimensional vectors trained using the same setting as SWE-300D.

In the experiments, these word embeddings are updated along with the training of FNN models but remain fixed during the training of HOPE models.

We also performed some preprocessings on the raw data. For the 20 Newsgroups dataset, we first removed all headers (e.g. "From", "Reply-To", "Organization") except for the "Subject". We then converted all letters in the text to lower case and removed all punctuations and special characters, that is, only numbers and lower case word letters are remained. We preprocessed the IMDB dataset in a similar way, except that we kept an additional six common punctuations: {",", ".", ":", ";", "?", "!"} that may affect sentiment values.

With the preprocessed data, we then compared them with the vocabulary of the word embeddings to get the effective vocabulary for each set of embeddings. We filtered out unused word vectors and denoted all out-of-vocabulary words in the

text documents as <UNKNOWN>. We added an all 0 vector specially for <UNKNOWN> and this word vector is fixed during training. The effective vocabulary size of the glove word vectors is 76761 for 20 Newsgroups and 78789 for IMDB. And the effective vocabulary size of the SWE word vectors is 60027 for 20 Newsgroups and 70356 for IMDB. All these numbers are including <UNKNOWN>.

All datasets are divided into a training set and a testing set according to their pre-defined split. Furthermore, we randomly select 10% data samples from the training set to form a validation set and use only the remaining 90% data samples for training. The validation set is used as a held out dataset to estimate the model performance for every intermediate epoch. The estimated performance on the validation set can help the model to determine the status of learning process. We use such a learning rate annealing schedule that during training, the learning rate starts from the initialized value and only decreases when the total "loss" value on the validation set is not lower than that of the previous epoch. Whenever the learning rate decreases, the updates performed in the current epoch are discarded, and the model will start training again from the saved previous epoch with a halved learning rate. The learning process finally terminates either after a limited number of reverting-backs or reaching a preset maximum number of training epochs. This learning rate annealing scheme helps the model to retreat from possibly wrong

directions and provides early stopping mechanism to avoid the problem of over-fitting.

We initialize the weights of our FNN models using the following formula described in [11]:

$$\boldsymbol{W} \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \quad \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \tag{6.1}$$

where U stands for uniform distribution and n_j is the size of the network's j-th layer. We use the popular rectified linear unit (ReLU) as activation function, as described in section 2.2.2, along with the batch normalization [18] technique to get more stable distributions before each layer. Moreover, we implemented the Adam optimization algorithm [21] to adaptively update the model weights.

6.2 Examining Position-wise Trained FNNs

In this section we present experiments with the position-wise trained FNNs, on each of the evaluation tasks. The model details are described in section 4.1. In these experiments, the models are trained with a maximum number of 20 epochs. In this section we will examine some important settings for the model including: word embeddings, vocabulary, forgetting factor for the context-FOFE encoding scheme and voting strategy.

In order to investigate the effect of using different word embeddings on the

FNNs, we conducted a series of experiments using different sets of word embeddings. The experimental results are shown in figure 6.1. As we can see, the model generally can achieve better performance by using larger word embedding dimension, except for the 20NG(4 categories) task, where the model achieved better performance on a smaller word embedding dimension of 50. This may be because the categories in the 20NG(4 categories) task are more distinct and hence information in the 50-dimensional word embeddings is already adequate for the task. In this case, we can known that feeding too much information into the model may disturb the learning process. In these experiments, we also found that using different kind of existing word embeddings often introduce not much difference to the model performance.

We also tried to vary the vocabulary size used in the model. We conduct this series of experiments on the IMDB dataset with the GloVe vectors. We shrink the vocabulary size by pruning the words that appear less than a set frequency lower-bound in the whole corpus. The experimental results are shown in figure 6.2, where "Glove_filtered_MFq30" stands for the GloVe vectors with all words having frequency less than 30 filtered out. These experiments show that vocabulary size is not always the larger the better. The model achieves the best performance when using only about 10,000 words, with all of their frequencies higher than 30. Low frequency words are noisy to the model and they typically cannot be learned well

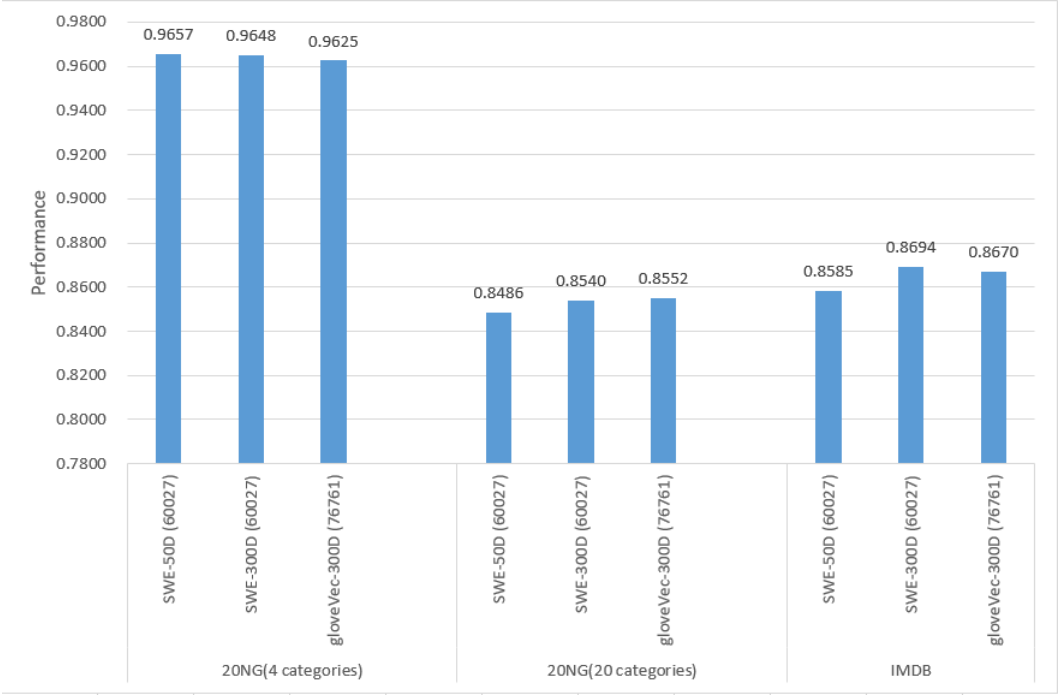


Figure 6.1: Effects of different word embeddings on the performance of the position-wise trained model.

as they don't have enough training samples. By filtering the less frequent words, the model can concentrate on the learning of the major features.

In our proposed context-FOFE encoding scheme, we use a parameter called forgetting factor to control the contribution of history and the effective encoding window in implementation, as described in section 3.3. To investigate the effective of this parameter on the model performance, we performed a series of experiments with different values of the forgetting factor. As shown in figure 6.3, the model

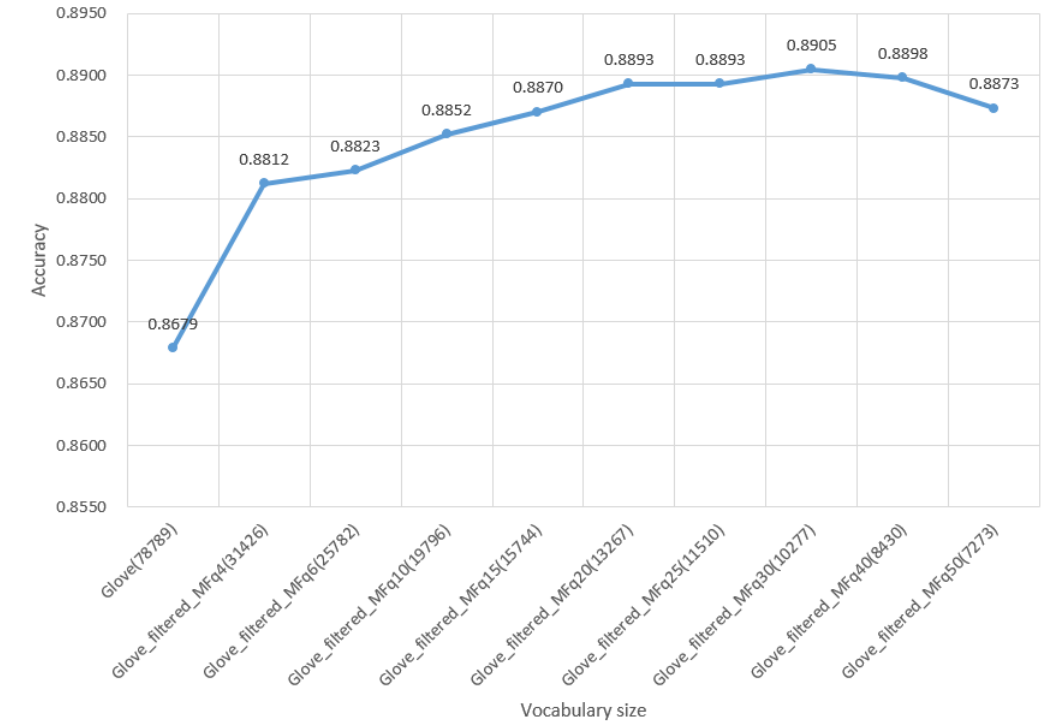


Figure 6.2: Effects of different vocabulary size on the performance of the position-wise trained model for the IMDB dataset.

achieves better performance on all tasks when the forgetting factor is set to 0.6. In our implementation, we treat FOFE weights less than 0.001 as neglectable and remove them for computational efficiency. In this case, a forgetting factor of 0.6 is equivalent to a context encoding window of size 14 including the center word.

As mentioned in chapter 3, we use two different voting strategies, the averaged voting and the softmax-normed voting, to aggregate the position-wise hidden representations to a document representation vector. Figure 6.4 shows the performance

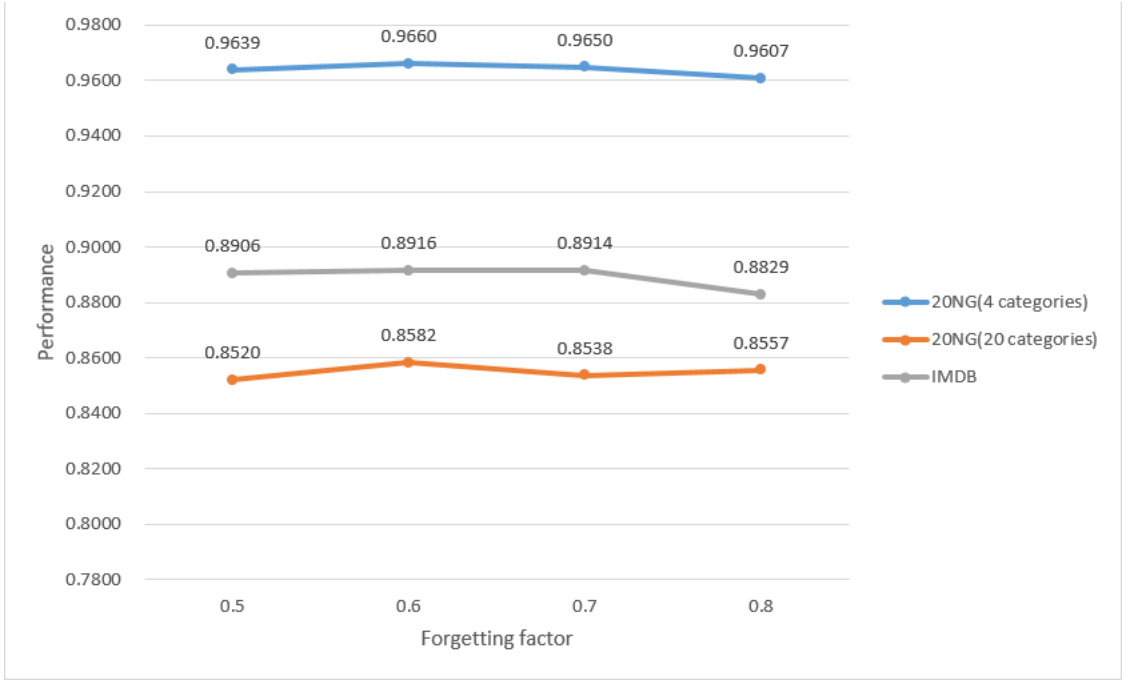


Figure 6.3: Effects of different forgetting factor on the performance of the position-wise trained model.

of the models using the two voting strategies along with the learning process. In the figure, we can find that the softmax-normed voting generally have better performance than the averaged voting. This is probably because normalizing the weights across categories on each position can balance the contributions of positions to the document representation.

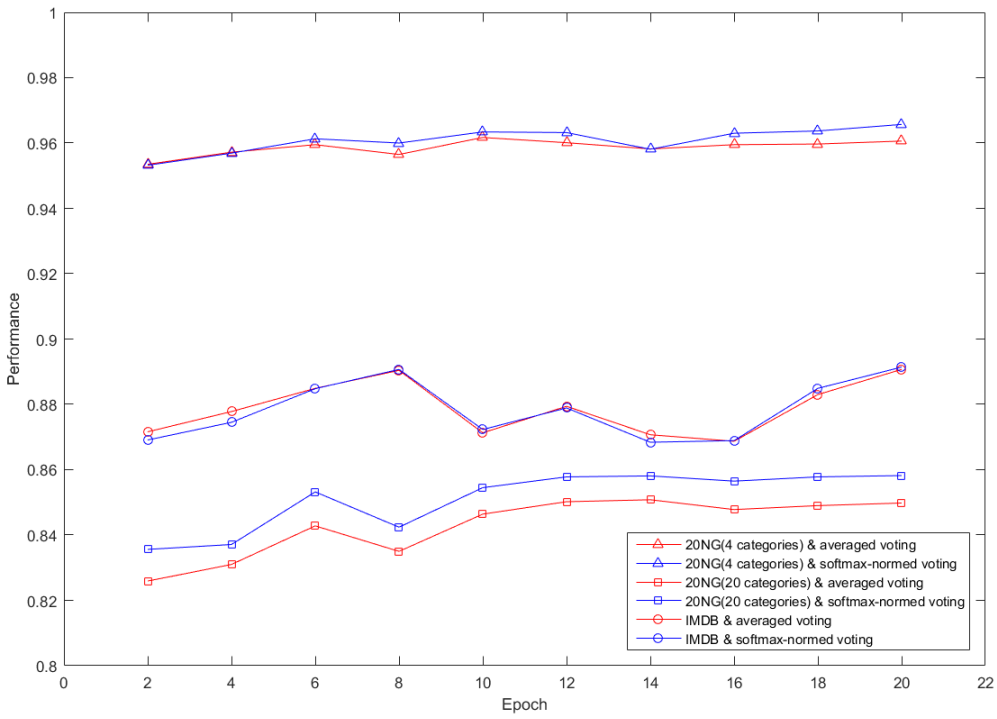


Figure 6.4: Effects of different voting strategy on the performance of the position-wise trained model.

6.3 Examining Document-wise Trained FNNs

This section presents experiments conducted with the document-wise trained FNNs. We follow the best settings in the previous experiments done on the position-wise trained models to select the word embeddings, vocabulary and forgetting factor. Specifically, we use a forgetting factor of 0.6 for all models, SWE-50D

for the 20NG(4 categories) task, gloveVec-300D for the 20NG(20 categories) task and Glove_filtered_MFq30 for the IMDB task. The models are all trained with a maximum epochs of 20. In this section, we will examine the effect of mini-batch size and the voting strategies on the model performance.

We conducted a series of experiments varying only the mini-batch size. The experimental results are shown in figure 6.5, 6.6 and 6.7. In these experiments, we found that using different mini-batch size (number of documents in a mini-batch) has a large impact on the model performance, while mini-batch size (number of positions in a mini-batch) in the position-wise trained models doesn't affect the model performance much. This is probably because in document-wise training, we have much less individual training samples, which means that a small change to the mini-batch size will greatly change the number of mini-batches in a training epoch.

Remember that in the document-wise training, we also use the voting strategies to aggregate position-wise hidden representations. Moreover, the document-wise trained models also need to back-propagate error through the voting layers. Therefore we can expect that the voting strategy used would have greater impact on the document-wise trained models. Figure 6.8 shows the performance of the models using the two voting strategies along with the learning process. As we can see from the figure, the model using the softmax-normed voting strategy performs significantly

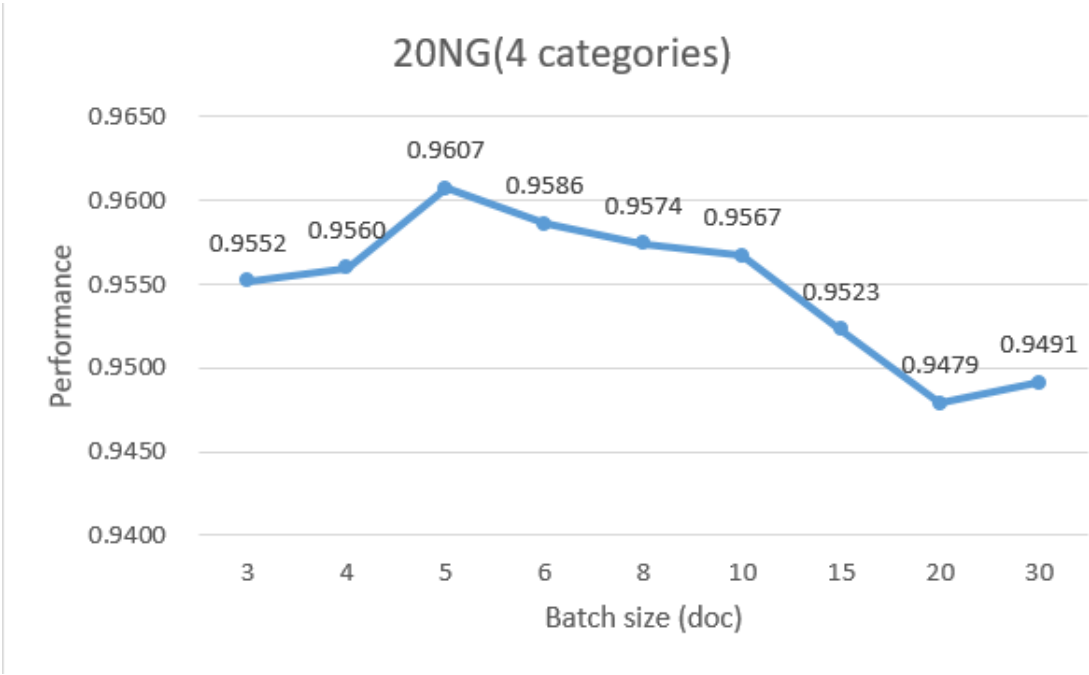


Figure 6.5: Effects of different mini-batch size on the performance of the document-wise trained model on the 20 Newsgroups 4 categories task.

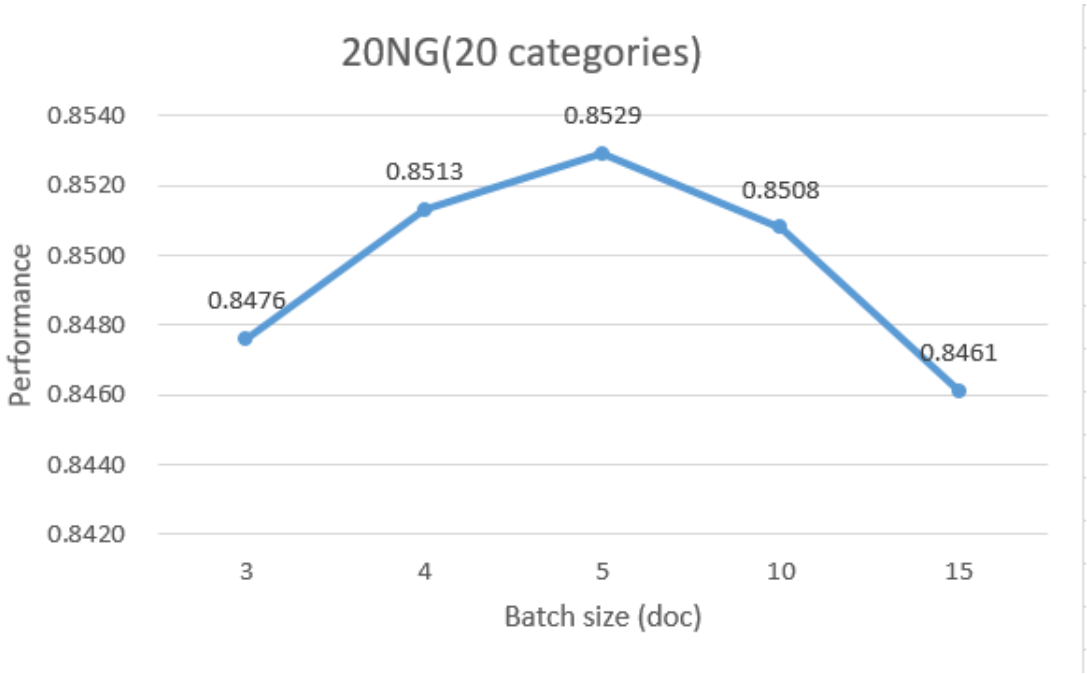


Figure 6.6: Effects of different mini-batch size on the performance of the document-wise trained model on the 20 Newsgroups 20 categories task.

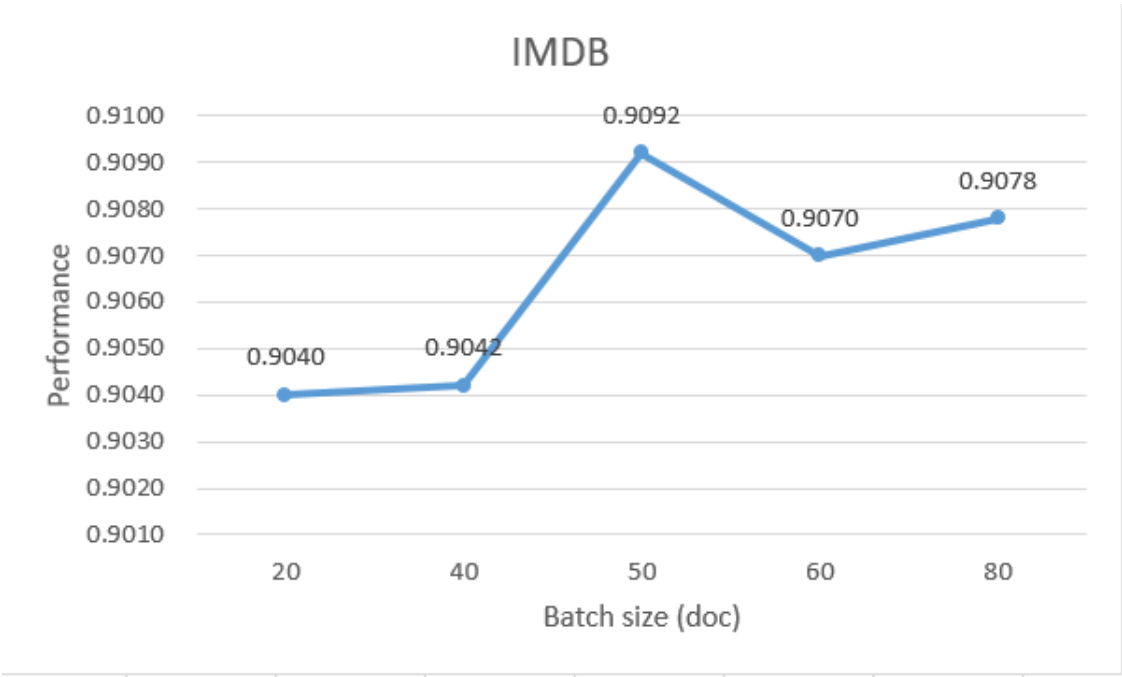


Figure 6.7: Effects of different mini-batch size on the performance of the document-wise trained model on the IMDB task.

worse than the model using the averaged voting, as opposite to what we observed in the position-wise trained models. We think the reason for this probably is that the stacked softmax layers in the softmax-normed voting is too complicated for the model to learn efficiently. Hence the model using the simpler averaged voting layer can learn much better.

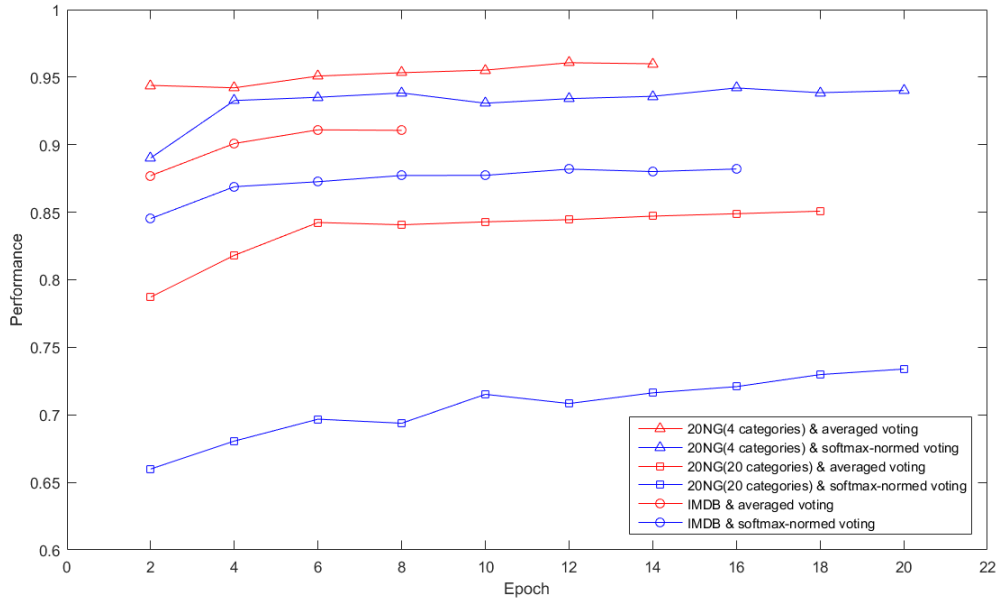


Figure 6.8: Effects of different voting strategy on the performance of the document-wise trained model.

Furthermore, from the figure 6.8, we can see that document-wise trained models can achieve much better result on the IMDB sentiment classification task while on the other two document classification tasks, position-wise trained models performs

better. This is reasonable because sentiment values often differs a lot from parts of the text to the whole text. Therefore training the text document position-wise may introduce a lot of noise to the sentiment values. On the other hand, document category and topic information are usually consistent throughout the whole document, hence by training position-wise, we are in effect augmenting the data to enhance the learning.

6.4 Examining Comparative HOPE Models

In this section, we present experiments conducted on the comparative HOPE models. Recall from section 5.2 that we train a HOPE model for each classification category to model the probability distribution of data in this category. Then we classify new documents by comparing their likelihood scores computed from each of the category models. Since each category model is trained separately, they need to be trained to a similar level to ensure comparability. In this case, we use the same parameter settings for all the categories. All models in these experiments are trained with a maximum of 30 epochs.

Recall that HOPE has an orthogonal projection step to project signals to a lower-dimensional space called latent feature space. In implementation, the dimension of the latent feature space is controlled by the parameter M. We conducted a

series of experiments to test on different M values to study its effect on the model performance. Experimental results are shown in figure 6.9. The model achieves relatively better performance when M ranges from 60 to 70. Noticing that the input dimension is 100, as we use the SWE-50D word embedding for the 20NG(4 categories) task, we can conclude that a good choice for the latent feature space dimension M is between 60% and 70% of the original input space dimension.

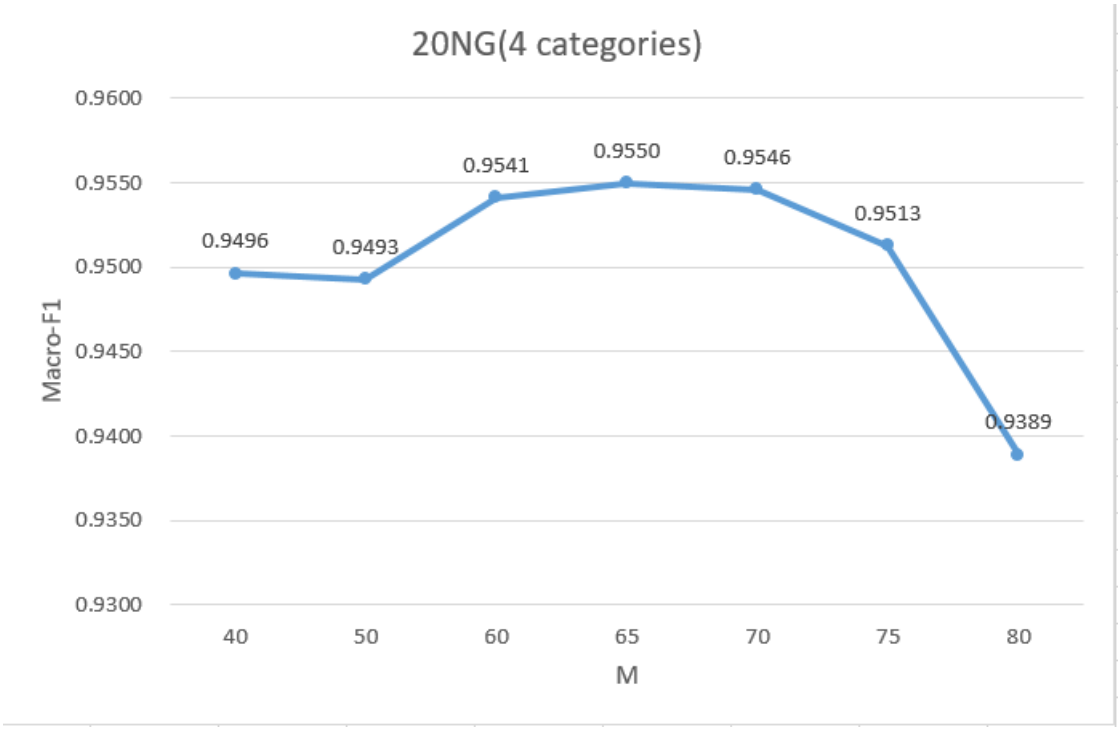


Figure 6.9: Effects of different M on the performance of the comparative HOPE model.

We also conducted a series of experiments on varying the parameter K, which

is the number of mixture components for the finite mixture model used in HOPE. Theoretically, the more mixture components, the better the distribution is modeled. However, using too large K value may results in over-fitting, as the model starts to model the distribution of this particular set of data samples. As we can see from the experimental results in figure 6.10, the model initially performs better when increasing K but the performance starts to drop when K exceeds 10000. Hence we can say that performance can be improved by using more mixture components until the number of mixture components is sufficiently large. In this experiment, a sufficiently large number of mixture components is around 10000. Additionally, from the experimental results, we can draw another conclusion that the model performance is less sensitive to K than to M.

Finally, we test the model with different forgetting factor values. As we can see from figure 6.11, the model achieves better performance when using a forgetting factor of 0.7, which in effect is equivalent to a context encoding window of size 20 including the center word. This context encoding window size is much longer than the one that best suits the previously examined FNNs, which indicates that HOPE model uses longer context dependency.

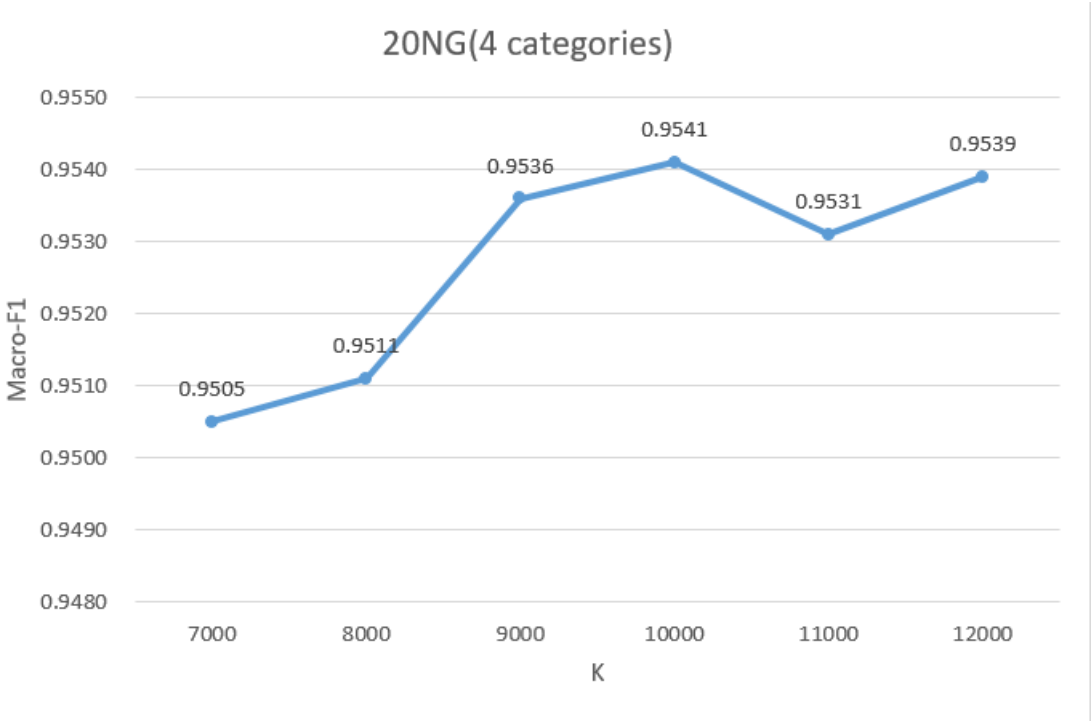


Figure 6.10: Effects of different K on the performance of the comparative HOPE model.

6.5 Comparing with Existing Models

In this section, we present our best models and compare them with the existing state-of-the-art models. Tables 6.2, 6.3 and 6.4 are the best settings for each of our proposed models on each of the testing tasks. As we can see from the tables, the position-wise trained FNNs achieve the best performance on the 20 Newsgroups document classification tasks, while the document-wise trained FNN has the best performance on the IMDB sentiment classification task. This again shows that our

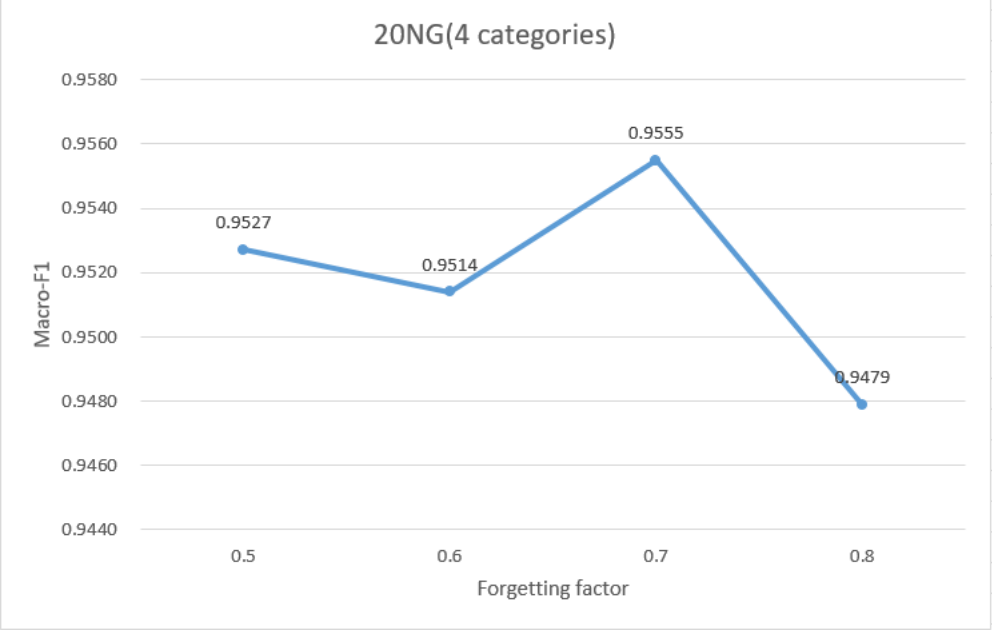


Figure 6.11: Effects of different forgetting factor on the performance of the comparative HOPE model.

position-wise trained FNN model is suitable for tasks that have a consistent label throughout the whole text document, while our document-wise trained FNN model is suitable for tasks that need to consider the whole text document. Furthermore, our comparative HOPE model achieves pretty good performance in the 20 Newsgroups(4 categories) task though the performance are not so good in the other two tasks. This is probably because the 20 closely related categories in the 20 Newsgroups(20 categories) task and the inconsistent position-wise sentiment values in the IMDB task confuse the comparative HOPE model. This implies that comparing

likelihood scores may not be a good strategy for general classification tasks.

Table 6.2: Performance of position-wise trained FNNs on different tasks.

Task	Embedding	Struct	Forgetting Factor	Learning Rate	Batch Size	Voting Strategy	Epoch	Performance
20NG(4 categories)	SWE-50D	$[100, 2048, 1024(\times 4), 4]$	0.6	0.0005	11000	softmax-normed	12	0.9661
20NG(20 categories)	gloveVec-300D	$[600, 2048, 1024(\times 5), 20]$	0.6	0.0005	11000	softmax-normed	20	0.8582
IMDB	Glove_filtered_MFq30	$[600, 2048, 1024(\times 3), 2]$	0.6	0.0005	7000	softmax-normed	16	0.8916

Table 6.3: Performance of document-wise trained FNNs on different tasks.

Task	Embedding	Struct	Forgetting Factor	Learning Rate	Weight Decay	Batch Size	Voting Strategy	Epoch	Performance
20NG(4 categories)	SWE-50D	$[100, 2048, 1024(\times 4), 4]$	0.6	0.0001	0	5	averaged	12	0.9607
20NG(20 categories)	gloveVec-300D	$[600, 2048, 1024(\times 5), 20]$	0.6	0.0001	0	5	averaged	16	0.8529
IMDB	Glove_filtered_MFq30	$[600, 2048, 1024(\times 3), 2]$	0.6	0.0001	1.1	50	averaged	14	0.9165

Table 6.4: Performance of comparative HOPE models on different tasks.

Task	Embedding	Forgetting Factor	Learning Rate	M	K	Batch Size	Epoch	Performance
20NG(4 categories)	SWE-50D	0.7	0.00001	65	10000	300	30	0.9555
20NG(20 categories)	gloveVec-300D	0.7	0.00001	390	10000	300	26	0.7924
IMDB	Glove_filtered_MFq30	0.7	0.00005	360	10000	250	30	0.8235

In table 6.5, we compare the performance of our models to the existing models. As shown in the table, our position-wise trained FNN model outperforms the previous state-of-the-art on the 20NG(4 categories) task. And the performance of our FNN models on the other two tasks are competitive with the state-of-the-art models. Moreover, our proposed generative model, the comparative HOPE models based on the context-FOFE encoding, has greatly out performed the LDA-based

model, which indicates that documents can be better modeled by using the context information.

Table 6.5: Performance comparison with existing models.

Model	**20NG(4 categories) (Macro-F1)**	**20NG(20 categories) (Accuracy)**	**IMDB (Accuracy)**
ClassifyLDA-EM [14]	0.9360	-	-
RCNN [22]	0.9649	-	-
SA-LSTM [8]	-	0.8440	**0.9276**
oh-2LSTMp [20]	-	**0.8668**	0.9186
contextFOFE-FNN (position-wise)	**0.9661**	0.8582	0.8916
contextFOFE-FNN (document-wise)	0.9607	0.8529	0.9165
contextFOFE-HOPE	0.9555	0.7924	0.8235

To investigate the effectiveness of our context-FOFE encoding scheme, we also compare the performance of our context-FOFE based FNN models with the bag-of-words (BoW) based FNN model as described in section 4.3. As shown in table 6.6, encoding context information using our context-FOFE encoding scheme can bring in very large performance improvements in the 20 Newsgroups document classification tasks and relatively smaller improvement in the IMDB sentiment analysis task. These results demonstrated that context is an important and very useful feature for text classification tasks, and our context-FOFE encoding scheme can effectively encode context information in text documents.

Table 6.6: Performance comparison with the bag-of-words based FNN model.

Model	**20NG(4 categories) (Macro-F1)**	**20NG(20 categories) (Accuracy)**	**IMDB (Accuracy)**
BoW-FNN	0.8755	0.6633	0.8821
contextFOFE-FNN (position-wise)	**0.9661**	0.8582	0.8916
contextFOFE-FNN (document-wise)	0.9607	0.8529	0.9165

7 Conclusions

7.1 Conclusions

In this thesis, we proposed the context-FOFE encoding scheme to encode text sequences of variable length into fixed size representations in which all context information is retained. Our context-FOFE encoding scheme has the following advantages:

- It generates unique representation for every different text sequence and can be easily reverted to get back the original text sequence, given a large enough vocabulary.
- Its representation comprises a list of fixed-size vectors, which can be easily applied to models that require fixed size input, individually or as a whole.
- Its encoding is based on a simple recursive formula and no parameter need to learn, hence the encoding process is efficient and fast.

- It encodes contexts at each word position to produce slightly different views of a text sequence, hence effectively providing more data for training

Based on the context-FOFE encoded document representations, we designed two simple regular feed-forward neural network models for the text classification task. One is trained with individual context vectors corresponding to word positions, while the other one is trained with the whole context matrix corresponding to a document. The models were tested on the 20 Newsgroups dataset [23] for document categorization and the IMDB dataset [28] for sentiment classification. Experimental results showed that our position-wise trained FNN model outperforms the state-of-the-art [22] on the 20 Newsgroups 4 categories document categorization task. The experiments of our models on the other datasets also showed competitive results with the existing best models [8, 20, 22] that use rather complex neural network structures.

We further used a unified data modeling framework called HOPE [43] to build a generative model on the context-FOFE encoded document representations. The generative model was also tested on the 20 Newsgroups dataset for document categorization. In order to perform the document classification task, we proposed to train a generative model for each of the classification categories, and classify documents by comparing the likelihood scores produced from the category models.

Experimental results showed that our generative model outperforms the LDA based model [14]. Moreover, the performance of our generative model is competitive with other discriminative models.

Over all, the experimental results demonstrated that based on our context-FOFE encoded representations, we can achieve competitive or even better performance on text classification tasks using much simpler network models instead of the more complex RNN or LSTM based models. Furthermore, our generative model showed that using context information can result in much better text modeling performance. All these results support our assumption that contextual information is an important feature in the text classification task and text modeling task.

7.2 Future Works

There are still possibilities for future extension of this work. One of the possibilities is to combine the HOPE generative model with the FNN classifiers. In this combination, we can use the HOPE model as a feature extractor. Since the HOPE model is essentially a generative model that models the data distribution by many mixture components, we may use the likelihood scores on all the mixture components as features that can represent the statistical properties of the data. In this way, FNN classifiers trained on these features may have better generalization ability for the

data. Moreover, since HOPE can be trained in an unsupervised way, we can adopt the semi-supervised learning approach to unsupervisedly train the HOPE model on large amount of unlabeled data to better model the data distribution.

Bibliography

[1] Charu C Aggarwal and ChengXiang Zhai. A survey of text classification algorithms. In *Mining Text Data*, pages 163–222. Springer, 2012.

[2] Richard Bellman. Adaptive control processes: a guided tour. 1961.

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

[4] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3(Jan):993–1022, 2003.

[5] Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. revised, oct 2012.

[6] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[7] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

[8] Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. In *Advances in Neural Information Processing Systems*, pages 3079–3087, 2015.

[9] John R Firth. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*, 1957.

[10] Amir Globerson, Gal Chechik, Fernando Pereira, and Naftali Tishby. Euclidean embedding of co-occurrence data. *Journal of Machine Learning Research*, 8 (Oct):2265–2295, 2007.

[11] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[12] Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947, 2000.

[13] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[14] Swapnil Hingmire, Sandeep Chougule, Girish K Palshikar, and Sutanu Chakraborti. Document classification by topic labeling. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 877–880. ACM, 2013.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

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

[18] Sergey Ioffe and Christian Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[19] Rie Johnson and Tong Zhang. Effective use of word order for text categorization with convolutional neural networks. *CoRR*, abs/1412.1058, 2014.

[20] Rie Johnson and Tong Zhang. Supervised and semi-supervised text categorization using lstm for region embeddings. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 526–534, 2016.

[21] Diederik P. Kingma and Jimmy Ba. Adam: a method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[22] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In *AAAI*, volume 333, pages 2267–2273, 2015.

[23] Ken Lang. Newsweeder: learning to filter netnews. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 331–339, 1995.

[24] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10): 1995, 1995.

[25] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185, 2014.

[26] Omer Levy, Yoav Goldberg, and Israel Ramat-Gan. Linguistic regularities in sparse and explicit word representations. In *CoNLL*, pages 171–180, 2014.

[27] Quan Liu, Hui Jiang, Si Wei, Zhen-Hua Ling, and Yu Hu. Learning semantic word embeddings based on ordinal knowledge constraints. In *Proceedings of the 53th Annual Meeting of the Association for Computational Linguistics*, July 2015.

[28] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[29] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

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

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

[32] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *hlt-Naacl*, volume 13, pages 746–751, 2013.

[33] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[34] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[35] Frank Rosenblatt. Principles of neurodynamics: perceptions and the theory of brain mechanisms. 1962.

[36] David E Rumelhart, Geoffrey E Hintont, and Ronald J Williams. Learning representations by back-propagating errors. *NATURE*, 323:9, 1986.

[37] David Sussillo and LF Abbott. Random walk initialization for training very deep feedforward networks. *arXiv preprint arXiv:1412.6558*, 2014.

[38] Duyu Tang, Bing Qin, and Ting Liu. Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1422–1432, 2015.

[39] Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, 1(4):339–356, 1988.

[40] Bernard Widrow and Marcian E Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, pages 96–104. IRE, 1960.

[41] Shiliang Zhang, Hui Jiang, Mingbin Xu, Junfeng Hou, and LiRong Dai. A fixed-size encoding method for variable-length sequences with its application to neural network language models. *CoRR*, abs/1505.01504, 2015.

[42] Shiliang Zhang, Hui Jiang, Mingbin Xu, Junfeng Hou, and Lirong Dai. The fixed-size ordinally-forgetting encoding method for neural network language models. In *Proceedings of ACL*, 2015.

[43] Shiliang Zhang, Hui Jiang, and Lirong Dai. Hybrid orthogonal projection and estimation (hope): a new framework to learn neural networks. *Journal of Machine Learning Research*, 17(37):1–33, 2016.

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