ANALYSING THE EFFECTS OF DATA AUGMENTATION AND HYPER-PARAMETERS FOR TEXT CLASSIFICATION WITH CONVOLUTIONAL NEURAL NETWORKS

JONATHAN K. QUIJAS

Department of Computer Science

APPROVED:

Olac Fuentes, Chair, Ph.D.

Monika Akbaré, Ph.D.

David Novick, Ph.D.

Pablo Arenaz, Ph.D.
Dean of the Graduate School

to my

FAMILY

thanks for everything

ANALYSING THE EFFECTS OF DATA AUGMENTATION AND HYPER-PARAMETERS FOR TEXT CLASSIFICATION WITH CONVOLUTIONAL NEURAL NETWORKS

by

JONATHAN K. QUIJAS

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2017

Acknowledgements

Abstract

Convolutional neural networks have seen very large success in computer vision problems and, more recently, natural language processing tasks. We examine the effects of hyperparameters on convolutional neural networks when applied in a text classification task. We also propose a keyword extraction method using a cosine similarity-based metric and a dataset augmentation application. We show that when we augment our dataset using this keyword extraction keyword, we can improve model generalization and loss on test data.

Table of Contents

Chapter 1

Introduction

1.1 Brief Overview of Deep Neural Networks

Deep convolutional neural networks have seen an enormous amount of success on a wide array of application, from scene interpretation to self-driving vehicles and art generation[CITE]. Natural language processing tasks are no exception to the range of problems deep learning can solve, from sentiment analysis to language modeling. In this work, we focus our efforts to studying convolutional neural networks for text classification. Especifically, we analyse how well modern neural networks models performed using scientific abstract text data from multiple disciplines such as astro-physics and computer science. The term modern in this context refers to neural models that use popular and recently (re)discovered techniques to achieve state-of-the-art performance on most machine learning benchmark datasets[CITE].

1.2 Training a Neural Network

A neural network is a function $f(\boldsymbol{x};\boldsymbol{\theta})$ that maps its input $\boldsymbol{x}$ to some response variable y. When we *train* a neural network, we *learn* the model parameters, or weights, $\boldsymbol{\theta}$ that minimize some cost function $J(\boldsymbol{\theta})$. For a regression task, where the model's output is a continuous variable, a common cost function is the **Mean Square Error**:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta}))^2$$

For categorical or discrete output variables found in e.g. classification tasks, we use the

Categorical Cross-Entropy:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},y\sim\hat{p}_{data}} \log p(y|\boldsymbol{x};\boldsymbol{\theta})$$

Given a *training* set of observations $\boldsymbol{x}_i$ and their true labels y_i, we compute weights that minimize the cost, or error, via maximum likelihood (ML) estimation:

$$\boldsymbol{\theta}_{ML} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i}^{m} \log P(y_i|\boldsymbol{x}_i;\boldsymbol{\theta})$$

,

which one can see is the equivalent of computing the weights that ***minimize*** the cross-entropy cost function.

1.3 Bias-Variance Tradeoff

When learning a neural network's weights, we use a *training set* so that we can later generalize previously unseen data with high accuracy (or some other determined metric). That means that during training, we obtain $\boldsymbol{\theta}_{ML}$ by minimizing $J_{train}(\boldsymbol{\theta})$, but we care about having low $J_{test}(\boldsymbol{\theta})$ i.e. low cost on test data points.

Overfitting occurs when a network is able to predict its training set extremely well i.e. very close to zero error, but fails to predict unseen data points. This is because the network's weights have been extremely fine-tuned to *fit* its training data, but do not fit or represent data points outside of its training sample. An overfitted model is said to have large **variance** and small **bias**. Conversely, underfitting occurs when the model fails to predict the training set because it generalizes too harshly. This model is said to have large bias and small variance.

Because of the commonly large amount of weights in deep convolutional networks, it is easy to overfit even a moderate size training set[CITE]. Many techniques exist to avoid overfitting in a neural network [EXPAND]

In this work, we study the effects of multiple regularization techniques used to avoid overfitting in a neural network[REFER TO CHAPTER].

-Describe and cite conv nets for text classification -Describe the conv net pipeline -Add figure of pipeline -Comment that LSTM have made huge progress -Describe difficulty with smaller datasets Convolutional neural networks obtain state of the art results on image and text processing tasks.

1.4 Data Augmentation: Increasing Training Sample Size

-Describe usual augmentation schemes for vision tasks -describe why they work, small changes, same class -propose augmentation scheme and refer to corresponding chapter

Chapter 2

Text Classification with Deep Neural Networks

2.1 Word Embeddings

A very common and simple vector respresentation for words is the one-hot representation. The length of a one-hot vector is the size of the data vocabulary i.e. how many distinct words are found in our data set. For any word, its one-hot vector is zeros everywhere except for a 1 at the word's index. This representation, although simple, fails to capture any meaning other than an identifier. Neural language models i.e. a language model learned using a network, learn to represent words as continuous, dense vectors. These dense, continouos vector representations are commonly called word embeddings. Due to the of the underlying algorithm used to learn these word embeddings, similar words tend to lie closer to each other on embedding space. Because of this, word embeddings are said to capture semantic relations, and thus encode more information than just a word identifier.

[SHOW EMBEDDING PROJECTION]

2.2 Convolutional Neural Networks

Convolutional neural networks are known for their abilities to learn high-level features from raw data. As input signals advance forward through the network, they produce latent signals as linear conbinations with learned parameters, have non-linearities applied to them, and have a pooling or selection mechanism based on simple functions such as the average or maximum operations.

When dealing with image data, images are convolved with multiple filters, each convolution applied to overlapping subimages called as receptive fields. This localized convolution process leads to discovery of low level features of images in the training set such as edges. As data flows forward through the model, higher level features are discovered e.g. wheels or headlights in a vehicle image dataset.

These networks are comprised of *feature maps*. A feature map is a **convolution** layer paired with a **pooling** layer afterwards. The convolution stage creates *activations*, whereas the pooling stage reduces dimensionality and creates translation invariance[CITE].

[INCLUDE IMAGE]

We can take advantage of word embeddings and apply convolutions to text in a fashion akin to convolutions with image data. In a similar manner, we apply convolutions to sub-regions of the input text i.e. bi-grams, tri-grams, etc. This

[DEVELOP]

2.3 Input Representation: Integer Sequences to Word Embeddings

Given a set of texts $\boldsymbol{w_1}, ..., \boldsymbol{w_m}$, we build a vocabulary $\mathbb{V}$ and a bag-of-words model from $\mathbb{V}$ to create a mapping $BoW : \mathbb{V} \mapsto \{1, ..., |\mathbb{V}|\}$. We represent a training text $\boldsymbol{w_i}$ as a sequence of integers, each integer being simply a word index in $\mathbb{V}$. Of course, the training texts will be of variable length. In order to enforce uniform input size for our neural networks, we apply **zero-padding**. For any arbitrary training instance $BoW(\boldsymbol{w}) = \boldsymbol{x} = x_1, ..., x_k$, we enforce that $k = n$, for the specified input size n. Thus, if $k<n$, we transform it into $\boldsymbol{x}_{pad} = x_1, ..., x_k, 0_{k+1}, ..., 0_n$. Conversely, if $k>n$, we simply truncate $\boldsymbol{x}$ to be of size n.

Having converted a text into a sequence of word indexes i.e. integers, we then convert this sequence into an embedding matrix. In order to convert a word into a dense, realvalued vector, we use a token-to-embedding dictionary to map a token to its corresponding embedding form.

Thus, for an input text $\boldsymbol{w}$, we transform it into a sequence of integers $\boldsymbol{x}$, and from there into $\mathbf{E} \in \mathbb{R}^{n \times d}$, where d is the embedding size.

Algorithm 1 Extract a keyword noun phrase from input sentence

```
procedure CSO({{np^1_1, ..., np^1_{|np^1|}}, ..., {np^n_1, ..., np^n_{|np^n|}}})
    for k = 1 to n do
        C^k = [np^k_1 ; ... ; np^k_{|np_k|}] × [np^k_1 ; ... ; np^k_{|np_k|}]^T
        c_k = 1/|np_i|^2 Σ_{i=1}^{|np_i|} Σ_{j=1}^{|np_i|} C^k_(i,j)
    end for
     return argmin_k c
end procedure
```

$$\boldsymbol{C^k} = \begin{bmatrix} np_1^k \\ \vdots \\ np_{|np_k|}^k \end{bmatrix} \times \begin{bmatrix} np_1^k \\ \vdots \\ np_{|np_k|}^k \end{bmatrix}^T$$

$$c_k = \frac{1}{|np_i|^2} \sum_{i=1}^{|np_i|} \sum_{j=1}^{|np_i|} \boldsymbol{C^k_{(i,j)}}$$

The procedure *CSO* receives as input a set of noun phrases $\mathbf{np^i} = \{np_1^i, ..., np_{|\mathbf{np^i}|}^i\}$ corresponding to the ith sentence. Each noun phrase np_1^i is comprised of one or more word embedding vectors, corresponding to the noun phrase's tokens mapped to their vector representations. For each noun phrase np_1^i, we compute the average cosine similarity between all the noun phrase's tokens. The algorithm returns the noun phrase which minimizes its average cosine similarity metric. The reasoning behind this noun phrase selection algorithm is as follows. A noun phrase with a very informative token i.e. word embedding will contain a token which will stand out from the rest of the tokens in the noun phrase. We assume that this can be measured by low cosine similarity between an informative token and all its other noun phrase tokens. We therefore compute, for all noun phrases in a sentence, the mean *within-noun-phrase* average cosine similarity, as described in algorithm 1.

Chapter 3

Model Performance with Regularization and Data Augmentation

Regularization is any approach used to reduce test error but not training error. One way to regularize a model is to introduce bias and reduce variance by imposing any constraint on its weights. By adding a penalty term to the cost function, we can shrink weights to avoid them from blowing up in magnitude and overfitting to the training set. L2 regularization adds the L2 norm of the weights to the cost function:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},y\sim\hat{p}_{data}} \log p(y|\boldsymbol{x};\boldsymbol{\theta}) + \lambda\|\boldsymbol{\theta}\|_2$$

where λ is the regularization control parameter. Higher values of λ penalize more, where a value of 0 degenerates to no regularization. A more recent and highly effective way to regularize a neural network is via **dropout**. Dropout "deactivates" a neuron or hidden unit with probability p_{drop}. In practice, a units output is set to 0 with probability p_{drop}.

Another approach that has received much recent attention is to augment the data set. **Data augmentation** refers to applying transformations to the input data in a way that the label value does not change.

3.1 Dropout

Dropout is one of the most popular regularization mechanism used in deep networks today. Simply put, dropout *deactivates* a unit with probability p_{drop}. This forces the network to not rely on certain activations, since any unit will only be *active* with probability $1 - p_{drop}$[CITE].

[IMAGE]

3.2 Dataset Augmentation

Data augmentation is ubiquitous in computer vision tasks. Example transformations include small translations, rotations, and even color intensity jitters via Pricipal Component Analysis[CITE]. In our work, we use sequence data i.e. scientific publication abstract texts.

3.2.1 Zero-Padding: Uniform Input Length

Although more complex neural models are designed to cope with variable length input, in practice a more common and simple approach is to pad data to be of some specified length as described in chapter [CITE CHAPTER]. The input length is a hyperparameter that should be fine-tuned, but a reasonable approach is to pad enough to fully accomodate the length of most input sequences i.e. its preferrable to pad than truncate and lose information.

3.2.2 Padding with Informative Values

When we pad our input sequence with 0s, we dont add any additional information; we simply create a constant input length. If instead we add values that characterize or help describe the input sequence more thoroughly, we increase the amount of useful information available during neural network training.

Consider a very simple input sequence, and assume its true label can be determined from a single word:

"This paper is about computer graphics"

where the label is "Graphics" i.e. a publication about computer graphics. In this simple case, the label is determined by the word "graphics". A single, **very informative** word is enough for the classifier to predict the label correctly. This very informative word however is only 1/6 of the entire text.

Now consider a the padded version, where we enfore an input length of 10:

"This paper is about computer graphics PAD PAD PAD PAD"

In the padded version, the word graphics is now only 1/10 of the entire text. If we could find this very informative word from within the text and pad using it instead of some meaningless token e.g. 0's, we could increase the information of this word over the entire input sequence by making it comprise a larger amount of the input sequence. In other words, make important words be present more frequently.

We propose to pad input sequences with **meaningful** values found already within the input text sequence instead of 0's only. In the following section, we will develop on how to extract these meaningful words via a cosine similarity-based metric and apply them to pad our data.

3.3 Keyword Extraction: Cosine Similarity Optimization (CSO)

As described in [CITE SECTION], we use a token-to-embedding dictionary to map an input sequence of word index i.e. integers into an embedding matrix i.e. a sequence of embedding vectors. This vectorial represention of words can be obtained after learning a language model using a neural network[CITE]. One property of this word embedding space is that words can now be represented as dense, continuous vectors of dimensionality much smaller than vocabulary size. Another important property is that semantic relationships are captured in this embedding space and can be measured between any two words using the cosine similarity metric[CITE]. This is an attractive property that allows for term comparison by semantic relationships. These relationships can range from complete similarity i.e. the same term, to term orthogonality i.e. unrelated words, and even capture the direction of the relation e.g. negative values of cosine similarity.

The cosine similarity between two vectors can be computed as follows:

$$\frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}$$

We can therefore quantify similarity between words in a sentece as a function of their cosine similarity with respect to each other. We refer to this measure as the *within-sentence* cosine similarity. Similarly, we can measure this similarity for words within a noun phrase in a sentence. We call this the *within-noun-phrase* cosine similarity.

3.3.1 Extracting Keywords: Pipeline

Given input text sequence: Given a document, our keyword extraction algorithm works as follows:

- Remove non-alphanumeric characters and convert to lowercase
- Tokenize each sentence into a list of words
- Perform Part of Speech tagging on each list of words i.e. sentence
- Chunk each sentence into a set of noun phrases
- Map each word in each noun phrase to its corresponding word embedding vector
- Perform Cosine Similarity Optimization (CSO) to extract a keyword noun phrase for each sentence

3.3.2 Extracting Keywords: Algorithm

The procedure CSO receives as input a set of noun phrases $np^i = \{np^i_1, ..., np^i_{|\boldsymbol{np^i}|}\}$ corresponding to a sentence, where np^i_k is the kth word embedding in the sentences ith noun phrase.

Given an input chunked sentence, CSO computes the average cosine similarity between all words in a noun phrase, for all noun phrases in the sentence. The algorithm returns the noun phrase which **optimizes** its average cosine similarity metric.

The reasoning behind the cosine similarity minimization is as follows: An informative noun phrase within a sentence will contain a word which will stand out from the rest of the words in the noun phrase. We quantify this notion of *standing out from the rest* using the mean within-noun-phrase cosine similarity. We therefore compute, for all noun phrases in a sentence, the mean *within-noun-phrase* average cosine similarity, as described in [REFER TO ALGORITHM]. We select the noun phrase that minimizes this measure.

In practice, we find both maximization and minimization of the mean *within-nounphrase* average cosine similarity leads to interesting results i.e. reasonable words to be considered keywords, so we include them both for padding. [INCLUDE SNAPSHOT OF CSO OUTPUT]

3.3.3 Proposed Padding Scheme

As described in [CSO SECTION], can use a cosine similarity-based metric to extract keywords from text. Consider an input text sequence S that yields keyword indexes $BoW(CSO(S)) = x_{key^1}, ..., x_{key^j}$ and input $\boldsymbol{x}_{pad} = x_1, ..., x_k, 0_{k+1}, ..., 0_n$, we concatenate the non-zero entries $x_1, ..., x_k$ with the extracted keyword indexes:

$$\boldsymbol{x}_{cso} = x_1, ..., x_k, x_{key^1}, ..., x_{key^j}, 0_{k+j+1}, ..., 0_n$$

3.4 Dataset Augmentation: Padding with Similar Keywords

Our CSO padding scheme can only pad with words already present in the input text sequence. It could prove beneficial to further pad the input with words that are *similar* to its keywords i.e. words extracted via CSO. We therefore further pad the input sequence with the nearest neighbor of each keyword. In this context, we consider the nearest neighbor of a word embeddings $\boldsymbol{w}$ as:

$$\operatorname*{argmin}_{\boldsymbol{w}_i} 1 - \frac{\boldsymbol{w} \cdot \boldsymbol{w_i}}{\|\boldsymbol{w}\| \|\boldsymbol{w_i}\|}$$

We use a Locality Sensitive Hashing forest for our approximate nearest neighbor search[CITE].

Chapter 4

Model, Dataset, and Final Pipeline Description

In this chapter we describe our model architecture. We start by describing our choice of layers, followed by our network's final architecture. We conclude the chapter by describing our dataset.

4.1 Layer Descriptions

4.1.1 Embedding Layer

This layer maps an word index, or integer, into its corresponding embedding. This is a layer in the neural network because the embeddings can be further fine-tuned during training. Because of the large number of parameters in this layer (number of words allowed times embedding size), we add a L_2 regularization.

4.1.2 Feature Maps: Convolution + Pooling

We refer to a pair of convolutional layer followed by a pooling layer as a feature map[CITE].

4.1.3 Gated Recurrent Unit Layer

4.2 Model Description

The network's general architecture is as follows:

- Input layer: word index vector
- Embedding layer: maps word index vector to embedding matrix
- Dropout layer
- Feature Map 1:
  - Convolution layer: number of kernels:32, activation: rectified linear
  - MaxPooling layer
- Dropout layer
- Feature Map 2:
  - Convolution layer: number of kernels:32, activation: rectified linear
  - MaxPooling layer
- Dropout layer
- Gated Recurrent Unit layer
- Dropout layer
- Dense layer: output size: number of classes, activation: softmax

[IMAGE OF ARCHITECTURE]

4.3 Dataset Description

[DESCRIBE AND CITE EMBEDDINGS]

We gathered scientific paper abstracts from the online repository Arxiv.org. We scraped papers for 5 departments: computer science, mathematics, astrophysics, physics, quantitative biology, and quantititative finance.

4.4 Final Pipeline

[INCLUDE PIPELINE DIAGRAM]

Chapter 5

Experimental Results

In this chapter we will show our results.

5.1 Dataset Descriptions

We scraped a scientific publication abstract dataset from Arxiv.org. Concretely, we gathered publications from the physics, mathematics, computer science, quantitative biology, and quantitative finance departments. For each department, we considered each **topic** as a class. For example, when considering computer science publications, we considered the "computational complexity" topic as one class, "artificial intelligence" as another class, and so on. We apply simple preprocessing to the texts by removing non-alphanumeric characters, and converting to lower case characters. [Overlap Figure] [Class Distribution Figures]

5.2 Results

In this section we show our experimental results. [Add tables and most relevant plots]

Chapter 6

Concluding Remarks

6.1 Significance of the Result

6.2 Future Work

References

[1] S. Cook, "The complexity of theorem-proving procedures," *Proceedings of the 3rd ACM Symposium on Theory of Computing*, Shaker Heights, Ohio, 1971, pp. 151–158.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, 1979.

[3] R. Karp, "Reducibility among combinatorial problems," in: R. Miller and J. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85–103.

[4] R. B. Kearfott and V. Kreinovich (eds.), *Applications of Interval Computations*, Kluwer Academic Publishers, Norwell, MA, 1996.

[5] V. Kreinovich, A. V. Lakeyev and S. I. Noskov, "Optimal solution of interval linear systems is intractable (NP-hard)," *Interval Computations*, 1993, No. 1, pp. 6–14.

[6] V. Kreinovich, A. V. Lakeyev and J. Rohn , "Computational complexity of interval algebraic problems: some are feasible and some are computationally intractable: a survey," in: G. Alefeld and A. Frommer (eds.), *Scientific Computing and Validated Numerics*, Akademie-Verlag, Berlin, 1996, pp. 293–306.

[7] V. Kreinovich, A. V. Lakeyev, J. Rohn and P. Kahl, *Feasible? Intractable? On Computational Complexity of Data Processing and Interval Computations*, Kluwer Academic Publishers, Norwell, MA, 1996 (to appear).

[8] U. Kulisch and W. L. Miranker, *Computer Arithmetic in Theory and Practice*, Academic Press, NY, 1981.

[9] A. V. Lakeyev and V. Kreinovich, "If input intervals are small enough, then interval computations are almost always easy," *Reliable Computing*, Supplement (Extended

Abstracts of APIC'95: International Workshop on Applications of Interval Computations), 1995, pp. 134–139.

[10] L. Levin, "Universal sequential search problems," *Problems of Information Transmission*, 1973, Vol. 9, No. 3, pp. 265–266.

[11] R. E. Moore, "Automatic error analysis in digital computation," *Technical Report LMSD-48421*, Lockheed Missiles and Space Co., Palo Alto, CA, January 1959.

[12] R. E. Moore, *Interval Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1966.

[13] A. Neumaier, *Interval Methods for Systems of Equations*, Cambridge University Press, Cambridge, 1990.

[14] S. G. Rabinovich, *Measurement Errors: Theory and Practice*, American Institute of Physics, NY, 1995.

Curriculum Vitae

Patrick Thor Kahl was born on July 12, 1961. The first son of Ulf Thor Gustav Kahl and Carolyn Kahl, he graduated from Coronado High School, El Paso, Texas, in the spring of 1979. He entered Auburn University in the fall of 1979, and, in the spring of 1982, The University of Texas at El Paso. In 1985 he joined the United States Navy where he served for eight years, most of it aboard the submarine USS Narwhal (SSN671). In the fall of 1993, after being honorably discharged from the navy, Patrick resumed his studies at The University of Texas at El Paso. While pursuing his bachelor's degree in Computer Science he worked as a Teaching Assistant, and as a programmer at the National Solar Observatory at Sunspot, New Mexico. He received his bachelor's degree in Computer Science in the summer of 1994.

In the fall of 1994, he entered the Graduate School of The University of Texas at El Paso. While pursuing a master's degree in Computer Science he worked as a Teaching and Research Assistant, and as the Laboratory Instructor for the 1995 Real-Time Programming Seminar at the University of Puerto Rico, Mayagüez Campus. He was a member of the Knowledge Representation Group and the Rio Grande Chapter of the Association for Computing Machinery.

Permanent address: 6216 Sylvania Way
El Paso, Texas 79912-4927