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

JONATHAN K. QUIJAS

Department of Computer Science

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 much success in computer vision and natural language processing tasks. Recent convolutional network architectures employ word embeddings, or words represented as dense vectors, to transform a given input sequence of words into a dense matrix of continuous values akin to image data. This allows for the well known convolution/pooling operations to be applied to text data in a manner similar to convolution and pooling operations on image data. These embeddings may incorporated into the neural network itself as a trainable layer to allow fine-tuning their values to better fit the data. Because of the large number of parameters in these models', they are prone to overfitting if not regularized appropriately or trained on a sufficiently large dataset. In this work, we study the performance of convolutional neural networks when the available training datasets are of moderate size and propose several data augmentation techniques designed for text datasets in an attempt to mitigate overfitting and improve model generalization. Finally, we provide discussion on our empirical results as well as future work.

Table of Contents

Page

Acknowledgements . . . v

Abstract . . . vi

Table of Contents . . . vii

List of Figures . . . x

Chapter

1 Introduction . . . 1

  1.1 Brief Overview of Deep Neural Networks . . . 1

  1.2 Training a Neural Network . . . 1

  1.3 Bias-Variance Tradeoff . . . 2

  1.4 Weight Regularization . . . 3

    1.4.1 Dropout . . . 4

2 Text Classification with Deep Neural Networks . . . 5

  2.1 Word Embeddings . . . 5

  2.2 Convolutional Neural Networks . . . 6

  2.3 Input Representation: Integer Sequences to Word Embeddings . . . 7

3 Model, Dataset, and Final Pipeline Description . . . 9

  3.1 Layer Descriptions . . . 9

    3.1.1 Embedding Layer . . . 9

    3.1.2 Feature Maps: Convolution + Pooling . . . 10

    3.1.3 Gated Recurrent Unit Layer . . . 10

    3.1.4 Dense Layer . . . 11

  3.2 Model Description . . . 12

  3.3 Model Hyper-Parameters . . . 13

  3.4 Dataset Description . . . 14

3.5 Final Pipeline 14
4 Towards Improving Model Performance: Regularization and Data Augmentation 15
4.1 Freezing the Embedding Layer 15
4.2 Dataset Augmentation: Shuffling and Noise Injection 16
4.2.1 Shuffling 17
4.2.2 Noise Injection 17
4.3 Dataset Augmentation: Padding 18
4.3.1 Wrap Padding 20
4.3.2 Keyword Padding 20
4.3.3 Proposed Padding Scheme 22
4.4 Dataset Augmentation: Padding with Similar Keywords 22
4.5 Reducing Data Granularity: From Abstract to Sentences 23
4.6 Embedding Matrix: Avoiding Overfitting 23
5 Experimental Results 24
5.1 Dataset Descriptions 24
5.2 Freezing the Embeddings: Results 25
5.3 Data Augmentation Results 25
5.4 AstroPhysics 26
5.4.1 Baseline 26
5.4.2 Padding 26
5.4.3 Sentence Split 27
5.4.4 Noise Injection 27
5.5 Math 28
5.5.1 Sentence Split 28
5.5.2 Padding 28
5.5.3 Noise Injection 28
6 Concluding Remarks 29
6.1 Significance of the Result 29

6.2 Future Work 29
References 30
Curriculum Vitae 31

List of Figures

1.1 Visualization of overfitting. Training error decreases as epochs progress, eventually reaching zero. Validation error starts increasing, indicating overfitting. The best model is shown at the dotted line, where validation error reached its minimum. 3

2.1 Visualization of embeddings using T-SNE. Words with similar or related meanings tend to lie close to each other in embedding space. 5

2.2 Visualization of a feature map with a single kernel. In this example, the kernel convolves tri-grams, or windows of three words. After convolution with all possible trigram context windows, max pooling is applied to reduce dimensionality. Here, the pool size is 3. This process is repeated for as many filters in the layer e.g. 100 and their outputs are concatenated horizantally to yield a matrix of size *num filters* × (*input length - pool size + 1*). 7

3.1 Visualization of an embedding layer. Word indexes are mapped to word embeddings. This layer outputs a matrix comprised of stacked embeddings, one for each index. 10

3.2 Visualization of a recurrent layer. Each hidden layer $\boldsymbol{h}^t$ is a function of the previous hidden layer $\boldsymbol{h}^{t\text{-}1}$, the present input signal $\boldsymbol{x}^t$. The weights are shared across time steps in order for the model to generalize better. 11

3.3 Model Architecture: embedding layer, followed by two feature maps, and a recurrent layer. At the end, we have a fully connected layer with a softmax activation, which will output class probabilities. 12

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. 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 on using convolutional neural networks for text classification. Especifically, we analyse how well modern neural networks models performed using moderately-sized scientific abstract text datasets 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.

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.

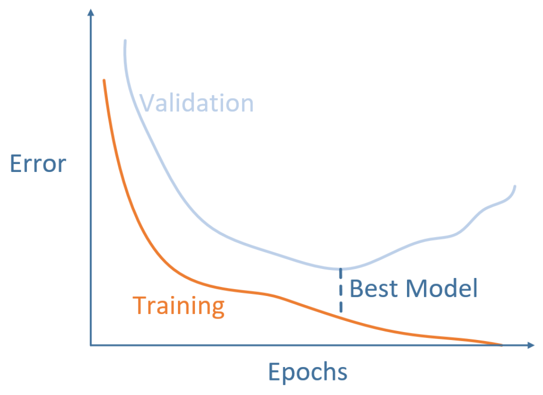
1.3 Bias-Variance Tradeoff

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. When training a neural network, we aim to find a balance between training cost and test cost, between overfitting and underfitting.

Because of the commonly large amount of weights in deep convolutional networks, it is easy to overfit even a moderate size training set[CITE]. It has been shown that when using a large enough dataset, neural networks are trained without overfitting, and thus generalize

Figure 1.1: Visualization of overfitting. Training error decreases as epochs progress, eventually reaching zero. Validation error starts increasing, indicating overfitting. The best model is shown at the dotted line, where validation error reached its minimum.



well to new unseen data. Because of the abundance of data today, it is relatively easy to acquire large datasets, although this is not always the case. In this work, we consider the case then the sizes of available datasets are moderately sized e.g. a tens of thousands. We quantify the effects of regularization used to avoid overfitting in a neural network[REFER TO CHAPTER], and propose dataset augmentation techniques to simulate a larger amount of available data.

1.4 Weight Regularization

One way to regularize a model to impose a 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. $\boldsymbol{L_2}$ 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 and a value of 0 leads to no regularization.

1.4.1 Dropout

A more recent and highly effective way to regularize a neural network is via **dropout**. Dropout "deactivates" a neuron or unit with probability p_{drop}. We deactivate a unit by setting its output to 0. This forces the network to not rely on certain activations, since any unit will only be *active* with probability $1 - p_{drop}$[CITE].

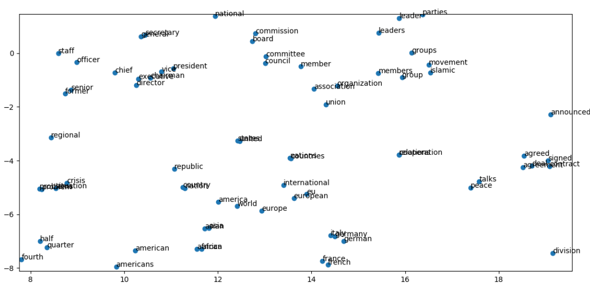
Chapter 2

Text Classification with Deep Neural Networks

2.1 Word Embeddings

Word embeddings are representations of words as vectors. Neural language models, language models learned using a neural network, learn to represent words as continuous, dense vectors. Because of the of the underlying algorithm used to learn these word embeddings, similar words tend to lie closer to each other on embedding space. Thus, word embeddings are said to capture semantic relations, and thus encode more information than just a word identifier e.g. a bag-of-words or one-hot vector representation.

Figure 2.1: Visualization of embeddings using T-SNE. Words with similar or related meanings tend to lie close to each other in embedding space.



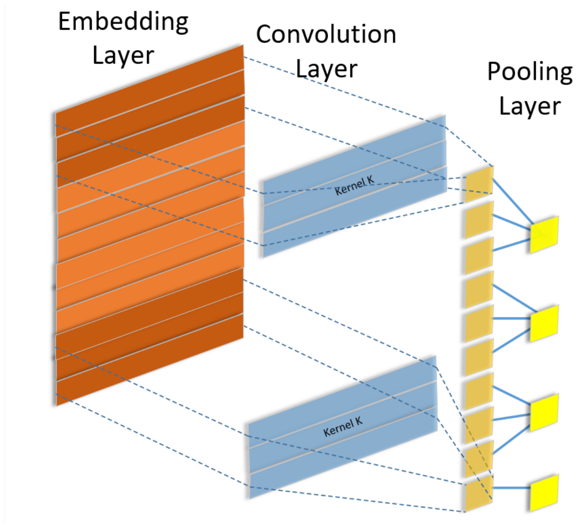
2.2 Convolutional Neural Networks

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 [CITE ZHOU AND CHELLAPA 1988].

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 convolutional neural 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.

Figure 2.2: Visualization of a feature map with a single kernel. In this example, the kernel convolves tri-grams, or windows of three words. After convolution with all possible trigram context windows, max pooling is applied to reduce dimensionality. Here, the pool size is 3. This process is repeated for as many filters in the layer e.g. 100 and their outputs are concatenated horizantally to yield a matrix of size *num filters* $\times$ (*input length - pool size + 1*).



We can take advantage of word embeddings to apply convolutions to text in a fashion similar to convolutions with image data. We apply the convolutions to overlapping subregions of the input text i.e. bi-grams, tri-grams, etc. After convolving these individual and overlapping subregions, we apply a non-linear function such as $max(0, x)$ and reduce data dimensionality by pooling.

2.3 Input Representation: Integer Sequences to Word Embeddings

Given a set of texts $\boldsymbol{s_1}, ..., \boldsymbol{s_m}$, 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 $\boldsymbol{s_i}$ as a sequence of integers $BoW(\boldsymbol{s}) = \boldsymbol{x} = x_1, ..., x_k$, each integer being simply a word index in $\mathbb{V}$. When received as input to a convolutional network, each word index will then be mapped to a corresponding embedding vector.

Chapter 3

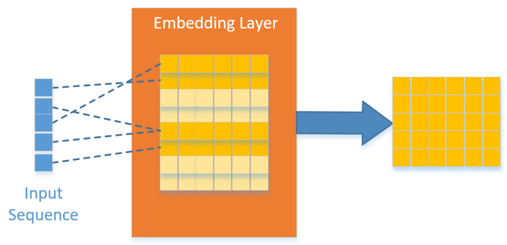
Model, Dataset, and Final Pipeline Description

3.1 Layer Descriptions

3.1.1 Embedding Layer

An **embedding layer** maps an word index, or integer, into its corresponding embedding. This layer is usually the first layer in a neural network that processes input such as text. Given input text s, this layer receives as input a sequence of word indexes $BoW(\boldsymbol{s}) = \boldsymbol{x} = x_1, ..., x_k$ and maps each word into an embedding. Thus, for an input text $\boldsymbol{s}$, 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. These embeddings are further fine-tuned during training. Because of the large number of parameters in this layer (number of words allowed times embedding size), it is usually a good idea to use L_2 regularization or dropout in order to avoid or mitigate overfitting.

Figure 3.1: Visualization of an embedding layer. Word indexes are mapped to word embeddings. This layer outputs a matrix comprised of stacked embeddings, one for each index.



3.1.2 Feature Maps: Convolution + Pooling

We refer to a convolutional layer followed by a pooling layer as a feature map. Since our data are text sequences and thus have temporal structure, we use a one-dimensional convolutional layer to convolve with embedding n-grams through the input sequence [CITE LANG AND HINTON 1988, WAIBEL ET AL 1989, LANG ET AL 1990]. As in [REFER TO CONVOLUTION DIAGRAM], we stride only along the time dimension. This is in contrast to the usual two-dimensional convolution schemes with image data, where the kernels stride along the width and height dimensions of the input data. We use a global max pooling scheme to ouput a single value for each kernel[CITE LIN ET AL 2014]. Besides the huge dimensionality reduction, we do this so that we pass to the next layer the single strongest activation from each kernel i.e. a strong feature selection mechanism.

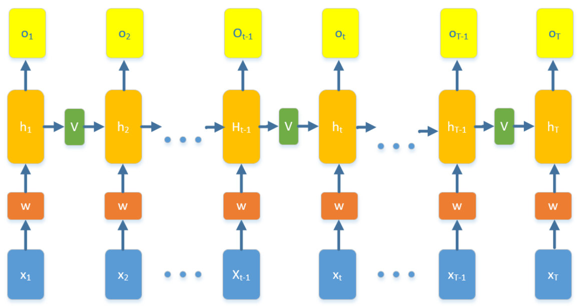
3.1.3 Gated Recurrent Unit Layer

A recurrent layer is designed to process data with temporal structure i.e. sequences [RUMELHART ET AL 1986]. This recurrent layer is in itself a network, where the hidden layer values at time t depend on the previous hidden layer's values, the input corresponding to time t, and a shared parameter set:

$$\boldsymbol{h}^t = f(\boldsymbol{h}^{t-1}, \boldsymbol{x}^t, \boldsymbol{\theta})$$

This is implemented by *unrolling* the layer along the time dimension as a sequence of layers, each corresponding to the next in time. Because of this design, back propagation of error through the unrolled layer can result in the vanishing gradient problem.

Figure 3.2: Visualization of a recurrent layer. Each hidden layer $\boldsymbol{h}^t$ is a function of the previous hidden layer $\boldsymbol{h}^{t\text{-}1}$, the present input signal $\boldsymbol{x}^t$. The weights are shared across time steps in order for the model to generalize better.



A Gated Recurrent Unit layer is a type of recurrent layer designed to combat this condition[CITE CHO ET AL 2014]. It is a simplified version of the earlier LSTM layer[CITE SCHMIDHUBER], also designed to mitigate the vanishing gradient problem, and in practice it performs comparably well. This layer models the latent features computed by the feature map as a time sequence. This means that it does not assume indepences between activation and learn to model the temporal structure of the previous layer's output. This property makes the layer an adequate choice, since we deal with text data which inhibits temporal structure.

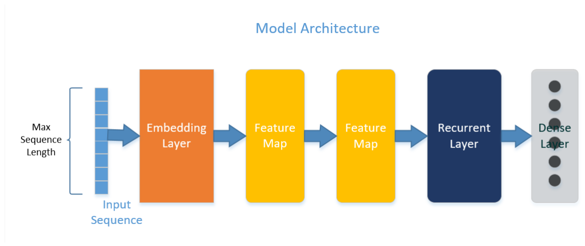
3.1.4 Dense Layer

The last layer in our model is the classical **dense**, or fully connected layer. The number of units in this layer is equal to the number of classes in our dataset. Each unit h_i is a linear combination of the previous layer's output with the unit's corresponding set of weights. The **softmax** activation function is then applied to the units, to finally output class probabilities.

3.2 Model Description

[YIN ET AL 2014] provide an overview of typical models used for text classification. A standard architecture for convolutional neural networks for text sequence classification is the one proposed by [CITE YOON 2014]. This model has an embedding layer followed by a feature map and a dense layer with a softmax activation to output class probabilities. Another popular neural model for text classification is the recurrent neural network. This model is essentially an embedding layer followed by a recurrent layer e.g. LSTM or GRU, then a dense layer to output probabilities. The model we use in this work is a mixture of these two standards: after the embedding layer, we add a feature map to learn the spatial structure of the data, then we add a GRU layer. This design choice was made due to the fact that a recurrent layer always led to better model performance, while the convolutional/pooling operations led to a huge data dimensionality reduction and thus much faster training times. The two types of layers, combined, performed the best.

Figure 3.3: Model Architecture: embedding layer, followed by two feature maps, and a recurrent layer. At the end, we have a fully connected layer with a softmax activation, which will output class probabilities.



Our network model is a variation of the convolutional model proposed by Yoon2014. Our model consists of an embedding layer, two feature maps, a recurrent layer, and a dense layer.

The network's architecture is as follows:

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

3.3 Model Hyper-Parameters

- number of training words
- number of kernels
- kernel size
- pool size
- learning rate
- regularization rate

- dropout rate
- maximum sequence length

3.4 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.

3.5 Final Pipeline

[INCLUDE PIPELINE DIAGRAM]

Chapter 4

Towards Improving Model Performance: Regularization and Data Augmentation

4.1 Freezing the Embedding Layer

When we make our model's embeddings trainable, its number of parameters highly increases. The embedding matrix adds a number of weights equal to the maximum input sequence length times the embedding size. While we observe that finetuning the embeddings yields good model accuracy, it also leads to rapid overfitting of when the dataset is of moderate size. We therefore try to answer the question: can we finetune the embeddings in such a way that we can avoid overfitting but retain the performance boost that comes along with it? We propose some embedding layer training schemes.

- Frozen Embeddings
  - With this scheme, we *freeze* the embedding layer altogether. Not once during training do we update the embedding vectors, so the model relies completely on its other layers' parameters.
- Freeze-Unfreeze
  - With the *freeze-unfreeze* scheme, we train the model with its embedding layer frozen until some stopping criterion is met e.g. early stopping due to validation

error increase. After this, we make the embedding layer trainable.

- Flash-Freeze
  - In the *flash-freeze* scheme, we allow the embedding layer to be trainable for a small number of epochs e.g. two, then freeze it or make it non-trainable.
- Unfrozen Embeddings
  - On the other extreme of the spectrum, we allow the embeddings layer to be trainable from the very beggining. This scheme makes the model converge in a small number of iterations but quickly starts to overfit the data.

Enabling the embedding layer to be trainable drastically increases the amount of parameters in the model. We observe that although fine-tuning the embeddings generally increases accuracy on the test set, the model quickly overfits the training data.

4.2 Dataset Augmentation: Shuffling and Noise Injection

Another way to improve model performance is to **augment** the dataset. Data augmentation refers to any transformation of the input data in a way that the label value does not change. Data augmentation is ubiquitous in computer vision tasks. Example transformations include translations, rotations, and even color intensity jitters via Pricipal Component Analysis[CITE]. All thesse tranformations should be are subtle enough that the overall structure is preserved, but allow for the model to process more distinct training data points.

In order to introduce more variance into our dataset, we propose to **shuffle** and **add noise** to the input sequences. We shuffle by randomly permuting non-overlapping neighborhoods in the input text sequence i.e.**context windows**. We further propose to add small amounts of noise to the input sequence by randomly replacing words with words taken from the training vocabulary.

4.2.1 Shuffling

We propose to augment our dataset by making small *context* changes that don't change the global structure of the input sequences. Concretely, we move a non-overlapping *context window* along the input sequence, randomly shuffling the words indexes inside it. For example using a context window of size 2 i.e. *bi-gram*, an input sequence

$$\boldsymbol{x} = x_1, x_2, x_3, x_4, x_5, x_6$$

could be shuffled to:

$$\boldsymbol{x}' = x_2, x_1, x_3, x_4, \underbrace{x_6, x_5}_{\text{bi-gram}}$$

In the first pass, the first two indexes get shuffled. The second pass led to no changes in ordering. The third and final pass shuffles the last two indexes.

The motivation for this dataset augmentation technique is that small changes in the ordering of the words will simulate a larger training sample; a training input sequence has a low chance of being repeated as its length increases. Smaller context windows preserve the most structure. This method will preserve most of the original context structure in the sequence, as long as the shuffling is not to harsh e.g. a complete random permutation of the sequence.

4.2.2 Noise Injection

One common augmentation technique for image datasets is to add small amount of noise. By replacing a relatively small amount of pixel values with noise, the model gets to process and train on a different but similar instance. The noise should be subtle enough as to not distort the image too much, otherwise we could potentially train the model using mostly noise. We propose to augment our dataset by injecting small amounts of noise to each training sample. Again, we aim to simulate a larger training sample while avoiding harsh changes to the original inputs. Concretely, each word in a text sequence gets replaced with a specified probability with a word randomly chosen from our training vocabulary.

Algorithm 1 Add noise to input sequence

```
1: procedure NoiseInjection($\boldsymbol{x} = [x_1, ..., x_n], \mathbb{V}, p_{noise}$)
2:     for $k = 1$ to $n$ do
3:         if $p_k \sim U(0,1) \leq p_{noise}$ then
4:             $x_k \leftarrow x'_k \in \mathbb{V}$
5:         end if
6:     end for
       return $\boldsymbol{x}$
7: end procedure
```

4.3 Dataset Augmentation: Padding

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]. In order to enforce uniform input size for our neural networks, we apply **zero-padding**.For any arbitrary training instance $BoW(\boldsymbol{s}) = \boldsymbol{x} = x_1, ..., x_k$, 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. The input length introduces another model hyper-parameter 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.

Our network's input is a sequence of integers, each integer being a word index: a number representing a word in our vocabulary $\mathbb{V}$. Word indexes range from 1 to $|\mathbb{V}|$, and the index 0 being left for out-of-vocabulary words. When we pad our input sequence with 0s, we dont add any additional information; we simply create a constant input length. We propose that if instead we add values that characterize or help describe the input sequence more thoroughly, we may 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. In the following subsections, we propose several padding schemes in order reduce the amount of non-informative when training.

4.3.1 Wrap Padding

Our first padding scheme is to *wrap around* the text, repeating words once we reach the padding portion. Refering back to the first example, the input sequence:

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

would then be

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

One thing to observe is that using this simple scheme, we remove all 0's i.e. non-informative padding indexes from the text sequence, but we also don't have selection mechanism and can miss useful words such as missing the word "*graphics*" in this example. Nonetheless, it is a simple first approach to pad our data.

4.3.2 Keyword Padding

One characteristic of the wrap padding scheme described above is that there is no selection mechanism. Words are not selected based on some metric, but rather just *wrapped*. We now propose a simple algorithm to select *keywords* from within the input text sequence based on a cosine similarity based metric, as well as a padding scheme using those keywords.

Using word embeddings, word tokens i.e. strings can now be represented as dense, continuous vectors of dimensionality much smaller than vocabulary size. Another useful property of embeddings 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. We represent a noun phrase **np** as a matrix composed of horizontally stacked vectors of size d, each vector a word in the noun phrase:

$$\mathbf{np} = \begin{bmatrix} \boldsymbol{np}_1 \\ \vdots \\ \boldsymbol{np}_n \end{bmatrix}, \boldsymbol{np}_i \in \mathbb{R}^{1\times d}$$

Given a noun phrase **np** comprised of n unit norm embeddings $\boldsymbol{np}_i$, we can compute its *within noun phrase* cosine similarity by:

$$c = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \begin{bmatrix} \boldsymbol{np}_1 \\ \vdots \\ \boldsymbol{np}_n \end{bmatrix} \times \begin{bmatrix} \boldsymbol{np}_1 \\ \vdots \\ \boldsymbol{np}_n \end{bmatrix}^{\mathsf{T}}$$

Thus, given a set of noun phrases $\{\mathbf{np}^{(1)}, \dots \mathbf{np}^{(m)}\}$ from a sentence, we compute the corresponding set of *within noun phrase* cosine similarities $\boldsymbol{c} = \{c_1 \dots c_m\}$, and we select the noun phrase which corresponds to minimum

$$\mathbf{np}^{(j)}, j = \operatorname{argmin} \boldsymbol{c}$$

.

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]

4.3.3 Proposed Padding Scheme

We propose to pad an input sequence $\boldsymbol{x}_{pad} = x_1, ..., x_k, 0_{k+1}, ..., 0_n$ using the indexes of extracted keywords $x_{key^1}, ..., x_{key^j}$ by concatenating the non-zero entries $x_1, ..., x_k$ with the extracted keyword indexes:

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

4.4 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:

$$\min_{\boldsymbol{w}_i} 1 - \frac{\boldsymbol{w} \cdot \boldsymbol{w}_i}{\|\boldsymbol{w}\| \|\boldsymbol{w}_i\|}, \boldsymbol{w} \neq \boldsymbol{w}_i$$

We use a Locality Sensitive Hashing forest for our approximate nearest neighbor search[CITE]. After having computed the nearest neighbor of each keyword, we pad as follows:

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

4.5 Reducing Data Granularity: From Abstract to Sentences

Our last proposed dataset augmentation scheme is break each training abtract into a set of sentences, and trained the network using this finer text granularity. For example, training a model using single sentences, another using sentence pairs, and another using sentence triplets.

During testing, we break a test abstract into sentence sets and classify each set individually. We then assign the class with the largest mean.

4.6 Embedding Matrix: Avoiding Overfitting

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.

Department	Number of Labels
AstroPhysics	5
Physics	13
Computer Science	20
Mathematics	26
Quant. Bio	0

For our pretrained embeddgins, we used the GloVe embedding set [CITE GLOVE]. This set consists of six billion tokens, each represented as a vector of size 100.

5.2 Freezing the Embeddings: Results

We tested the four different proposed embedding training approaches. The first is to leave the embeddings frozen throughout training i.e. non-trainable parameters. The second approach is to freeze the embeddings until convergeance, then retrain with the embeddings unfrozen. The thid approach is to train with the embeddings unfrozen for a small amount of epochs e.g. three, then freeze the embeddings and retrain the model. The last approach is to simply train with the embeddings as trainable parameters from the very beggining.

We now present our results

Dataset	Training Set Size	Frozen	Not-Frozen	Freeze-Unfreeze	Flash-Freeze
AstroPhysics	17500	.755(6)	.770(8)	0.769(14)	**.775(6)**
Physics	45000	.715(9)	**.781**(10)	.777(39)	.780**(8)**
Computer Science	63400	.621**(9)**	.693(12)	0.695(56)	**.701**(10)
Mathematics	87700	.590(22)	**.685**(21)	.678(72)	.684**(11)**
Quant. Bio	11000	.834(15)	.835(6)	0.836(21)	**.841(5)**

Table 5.1: Accuracy results on all datasets with proposed embedding training schemes. The numbers in parenthesis represent the number of epochs until the model converged.

We observe that the Flash-Freeze method achieves relatively faster convergence, with accuracy competent or better than the accuracy of all other methods.

5.3 Data Augmentation Results

In this section we show our experimental results when using our proposed data augmentation techniques. Our model hyper-parameters are as follows: number of training words=20000, kernel size=3, learning rate=0.001, regularization rate=0.00001, dropout rate = 0.2, maximum sequence length = 250.

5.4 AstroPhysics

[Distribution]

5.4.1 Baseline

NWords	Batch	MaxLen	p_{drop}	Kern	Pool	α	λ	Acc
30000	64	200	0.442	3	5	1e-3	1e-4	0.773
80000	64	250	0.059	3	2	1e-4	1e-5	0.760
80000	64	250	0.056	2	2	1e-3	1e-4	0.759

5.4.2 Padding

Wrap Padding

NWords	Batch	MaxLen	p_{drop}	Kern	Pool	α	λ	Acc
30000	64	200	0.442	3	5	1e-3	1e-4	0.773
80000	64	250	0.059	3	2	1e-4	1e-5	0.761
80000	64	250	0.056	2	2	1e-3	1e-4	0.760

CSO Padding

NWords	Batch	MaxLen	p_{drop}	Kern	Pool	α	λ	Acc
30000	64	200	0.442	3	5	1e-3	1e-4	0.774
80000	64	250	0.059	3	2	1e-4	1e-5	0.752
80000	64	250	0.056	2	2	1e-3	1e-4	0.761

LSH Padding

NWords	Batch	MaxLen	p_{drop}	Kern	Pool	α	λ	Acc
30000	64	200	0.442	3	5	1e-3	1e-4	0.768
80000	64	250	0.059	3	2	1e-4	1e-5	0.752
80000	64	250	0.056	2	2	1e-3	1e-4	0.755

5.4.3 Sentence Split

Mean Vote

NWords	Batch	MaxLen	p_{drop}	Kern	Pool	α	λ	Acc
30000	512	50	0.395	3	3	1e-5	1e-4	0.730
60000	512	40	0.051	4	4	1e-5	1e-6	0.7152
20000	512	40	0.062	5	5	1e-4	1e-6	0.685
20000	512	60	0.247	3	5	1e-3	1e-5	0.710
10000	512	60	0.121	5	3	1e-3	1e-5	0.712
70000	512	40	0.223	4	5	1e-5	1e-5	0.690
40000	128	50	0.225	5	3	1e-5	1e-4	0.734
30000	64	40	0.132	3	4	1e-3	1e-4	0.711
40000	1024	50	0.225	5	3	1e-4	1e-4	0.689

5.4.4 Noise Injection

NWords	Batch	MaxLen	p_{drop}	Kern	Pool	α	λ	p_{aug}	p_{noise}	Acc
80000	64	250	0.548	4	5	1e-4	1e-5	0.653	0.195	0.537
70000	64	300	0.423	5	4	1e-5	1e-3	0.107	.004	0.665
70000	32	200	0.512	3	4	1e-3	1e-5	0.179	.373	0.764
80000	128	200	0.185	4	4	1e-5	1e-6	0.973	0.128	0.680
50000	32	250	0.167	4	4	1e-4	1e-4	0.866	0.364	0.761

5.5 Math

[Distribution] [Distribution]

5.5.1 Sentence Split

Mean Vote

5.5.2 Padding

Wrap Padding

NWords	Batch	MaxLen	p_{drop}	Kern	Pool	α	λ	Loss	Acc
80000	64	250	0.548	4	5	1e-4	1e-5	1.818	0.537
2	7	78	5415						

CSO Padding

NWords	Batch	MaxLen	p_{drop}	Kern	Pool	α	λ	Loss	Acc
80000	64	250	0.548	4	5	1e-4	1e-5	1.818	0.537
2	7	78	5415						

LSH Padding

NWords	Batch	MaxLen	p_{drop}	Kern	Pool	α	λ	Loss	Acc
80000	64	250	0.548	4	5	1e-4	1e-5	1.818	0.537

5.5.3 Noise Injection

NWords	Batch	MaxLen	p_{drop}	Kern	Pool	α	λ	p_{aug}	p_{noise}	Acc	
80000	64	250	0.548	4	5	1e-4	1e-5	0.653	0.195	1.818	0.537

Chapter 6

Concluding Remarks

6.1 Significance of the Result

6.2 Future Work

References

[1] @articlekim2014convolutional, title=Convolutional neural networks for sentence classification, author=Kim, Yoon, journal=arXiv preprint arXiv:1408.5882, year=2014

Curriculum Vitae