ANALYSING THE EFFECTS OF DATA AUGMENTATION AND FREE PARAMETERS FOR TEXT CLASSIFICATION WITH RECURRENT CONVOLUTIONAL NEURAL NETWORKS

JONATHAN K. QUIJAS

Computer Science

APPROVED:

Olac Fuentes, Chair, Ph.D.

Monika Akbar, Ph.D.

David Novick, Ph.D.

Charles Ambler, Ph.D.
Dean of the Graduate School

to my

FAMILY

thanks for everything

ANALYSING THE EFFECTS OF DATA AUGMENTATION AND FREE PARAMETERS FOR TEXT CLASSIFICATION WITH RECURRENT 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

Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2017

Abstract

Convolutional neural networks have seen much success in computer vision and natural language processing tasks. When training convolutional neural networks for text classification tasks, a common technique is to transform an input sequence of words into a dense matrix of word embeddings, or words represented as dense vectors, using table lookup operations. This allows for the inputs to be represented in a way that the well-known convolution/pooling operations can be applied to them in a manner similar to images. These word embeddings may be further incorporated into the neural network itself as a trainable layer to allow fine-tuning, usually leading to improved model predictions. The drastic increase of free parameters however, leads to overfitting if proper regularization is not applied or the size of the training set is not large enough.

We give an overview of popular convolutional and recurrent network architectures, describe their basic functions, and discuss their observed advantages and shortcomings in our experiments. We follow this discussion with an overview of our final choice of architecture, based on a combination of these architectures.

We train our neural networks using abstracts from multiple science and engineering fields; each set of abstracts comprised of multiple topics. The number of publications available for our task is moderate, in the mid thousands for each topic. We analyse the effect of using word embeddings with our models in terms of fit and prediction. We then propose embedding "trainability" schemes to alleviate overfitting, improve test accuracy and reduce training times. We conclude our study proposing several data augmentation techniques designed for text sequences in an attempt to further mitigate overfitting and improve generalization. Finally, we provide discussion on our empirical results and propose promising directions for future work.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work is an early step towards fascilitating interdisciplinary research within the University of Texas at El Paso. In the future, we wish to design a system to automatically assign or recommend faculty to interdisciplinary research communities based on their publication data. For this task, we want to learn models that capture important and high-level features from raw text in order to make decisions as accurate and meaningful to the faculty members as possible. Neural networks are effective at learning said complex features. One important consideration is that neural networks work best when trained using large amounts of data, as is the case with statistical models. This study is to better understand deep convolutional and recurrent networks for the task of classifying scientific publication abstracts and how well they can cope with limited amounts of data. We evaluate the predictive performance of multiple types of neural networks for text classification. We then evaluate the effect of integrating distributed representations of words as free parameters of the model. We then propose several data augmentation techniques to help reduce the effects caused by our moderate-sized training set.

1.1 Text Classification

Text classification is the task of assigning a label to a text document. A document can be any piece of text we wish to process; in this work, we label scientific and engineering publication abstracts. We wish to create a **model** that parametrizes some function that labels a document based on its **features**. The features of a document can be any set of values computed from the document itself. These features are the input representation

used for our classification model. A very popular feature representation of documents is the ***term frequency-inverse document frequency*** (tf-idf) representation, where a document is represented as a vector of frequency-based statistics of its words [18]. Another popular feature representation of documents is the ***Bag-of-Words***(BoW) representation, where a document is represented as a vector of indexes, each index corresponding to a word in the document [11]. A more recent approach to text classification is to use neural networks to automatically extract high-level features that contain context and semantic information [12][6]. In the following sections, we describe neural networks and provide a general overview of how to train them to allow generalization for tasks such as text classification.

1.2 Neural Networks

A neural network is a function $f(\boldsymbol{x}; \boldsymbol{\theta})$ that maps an input $\boldsymbol{x}$ to some response variable y. Neural networks are commonly represented as directed acyclical graphs. Each node is called a hidden unit, namely because it computes a latent variable of its input. We compute a unit's output as the dot-product of inputs with a corresponding set of parameters, followed by a non-linear function applied to the resulting scalar. This non-linearity is referred to in machine learning as the unit's *activation* function. This process is called **forward propagation**, or a forward pass of the neural network. Given input vector $\boldsymbol{x}$, we forward-propagate to compute the the ith hidden unit's output as:

$$h_i = g(\boldsymbol{x} \cdot \boldsymbol{\theta})$$

where g is a non-linear activation function.

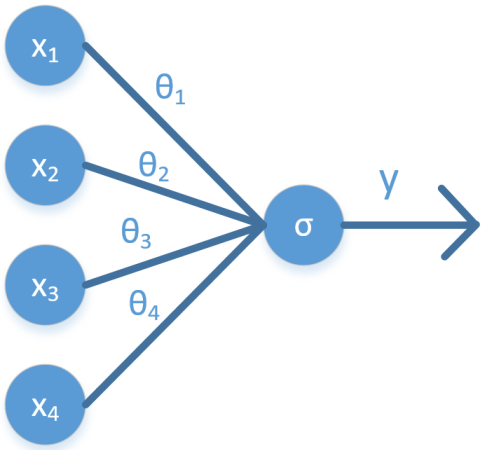

Figure 1.1: Visualization of a single unit neural network. A non-linearity is applied to the linear combination of the inputs x_i with the weights θ_i.

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**:

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

For categorical or discrete output variables found in classification tasks, we normally 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 equivalent to computing the weights that ***minimize*** the cross-entropy cost function.

1.2.1 Minimizing Non-Convex Functions: Gradient-Based Learning

Due to the non-linearities associated with a neural network, the loss function to be minimized becomes non-convex. Non-convex optimization is in general more difficult than convex optimization, so we usually rely on gradient-based methods for the task. Gradient based optimization provides a sound theoretical framework and a practical and well tested methodology for learning deep neural networks. This gradient-based learning process involves *moving*, or updating, the weight values in the direction opposite of the cost function's gradient, repeating this process for a series of **epochs**, or training iterations. These updates should be small, scaled by a *learning rate*, and can make learning a lengthy process. Gradient based learning is also scalable to datasets of enourmous size. The gradients and updates may be computed using small random batches of the training set at a time rather than the entire dataset; this is known as Stochastic Gradient Descent [2].

1.2.2 Bias-Variance Tradeoff

Given a set of identically and independently distributed observations $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$, we wish to compute the set of weights $\hat{\boldsymbol{\theta}}$ that is as close as possible to $\boldsymbol{\theta}$, the set of weights that truly parametrize the data generating process $p_{data}(\boldsymbol{x}; \boldsymbol{\theta})$ and thus minimizes the cost function. This concrete set of parameters (in contrast to a density over possible values) is called a **point estimator**, or statistic, of the true set of parameters. Unless the data generating process is known exactly, the parameter estimators will have error. Two different measures of error in an estimator are its bias and variance. The **bias** measures the expected deviation between $\boldsymbol{\theta}$ and the estimator $\hat{\boldsymbol{\theta}}$, with this expectation being over the data:

$$\begin{aligned} Bias(\hat{\boldsymbol{\theta}}) &= \mathbb{E}(\hat{\boldsymbol{\theta}}) - \boldsymbol{\theta} \\ &= \mathbb{E}[\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}] \end{aligned}$$

Intuitively the bias can be viewed as error caused by generalization in the model. If a model is trained with high bias, it will make strong assumptions of the data generating process. For example, it can assume that the **decision boundary**, or region of separation between classes, is linear when it could be quadratic.

The **variance** is a measure of the deviation from the expected estimator, caused by the randomness in the sample, or how much the estimator will vary as a function of the data sample:

$$Var(\hat{\boldsymbol{\theta}}) = \mathbb{E}[\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}]^2$$

Intuitively, the variance can be seen as error caused by variation in the training set; how much will the set of parameters differ from its expected value given a different training sample. Thus, if a model is trained with high variance, it will learn to model the nuances and variations specific to the training sample with low error but will fail to generalize well to new, unseen test data. During training, we obtain $\boldsymbol{\theta}_{ML}$ by minimizing $J_{train}(\boldsymbol{\theta})$, but we care about having low $J_{test}(\boldsymbol{\theta})$, or low cost on test data points. We seek a model that has a balance between variance (training specific nuances) and bias (generalization). This is known as the bias-variance tradeoff.

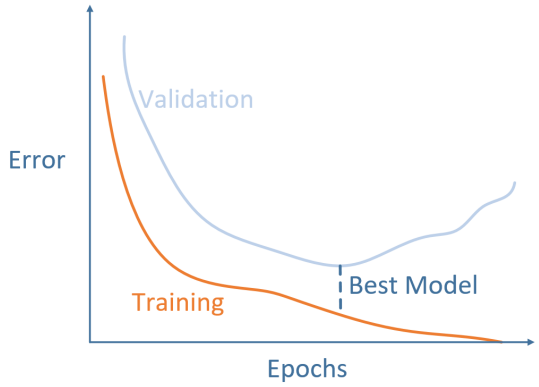

Figure 1.2: Visualization of overfitting. Training error decreases as epochs progress, eventually reaching zero. Validation error starts increasing, indicating overfitting. The best model is shown at the dotted line, where validation error reached its minimum.

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 samples. 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 much. 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 number of weights in deep convolutional networks, it is easy to overfit a moderate size training set [8]. It has been shown that when using a large enough dataset, neural networks are trained without overfitting, and thus generalize well to new unseen data. Because of the abundance of data today, it is usually easy to acquire large datasets, although this is not always the case. In this work, we consider the case when the available datasets are of moderately size, for example tens of thousands of examples.

1.2.3 Weight Regularization

One way to regularize a model is to impose a constraint on its weights. By adding a penalty term based to the cost function, we can prevent the weights from *blowing up* and overfitting 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 weight magnitudes more and a value of 0 leads to no regularization.

1.2.4 Dropout

A recently proposed and highly effective way to regularize a neural network is via **dropout**[8][24]. 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 combinations of activations, since any unit will only be *active* with probability $1 - p_{drop}$.

Chapter 2

Text Classification with Deep Neural Networks

2.1 Brief Overview of Deep Learning

Deep convolutional neural networks have seen much 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. This success of deep neural networks is accredited to their ability to learn high-level features from raw input data[15].

One type of neural networks that is particularly good at learning features from the data is the convolutional neural network (CNN)[5][16][19]. A CNN learns a set of weights, or convolution kernels, that are normally much smaller than the input size. The difference in size forces the weights to have **sparse interaction**, or interaction on a subregion of the input, versus the dense interaction between every input with every output in traditional neural networks (vector-matrix multiplication). This sparse interaction allows for features to be detected locally within the input (for example, an edge detector).

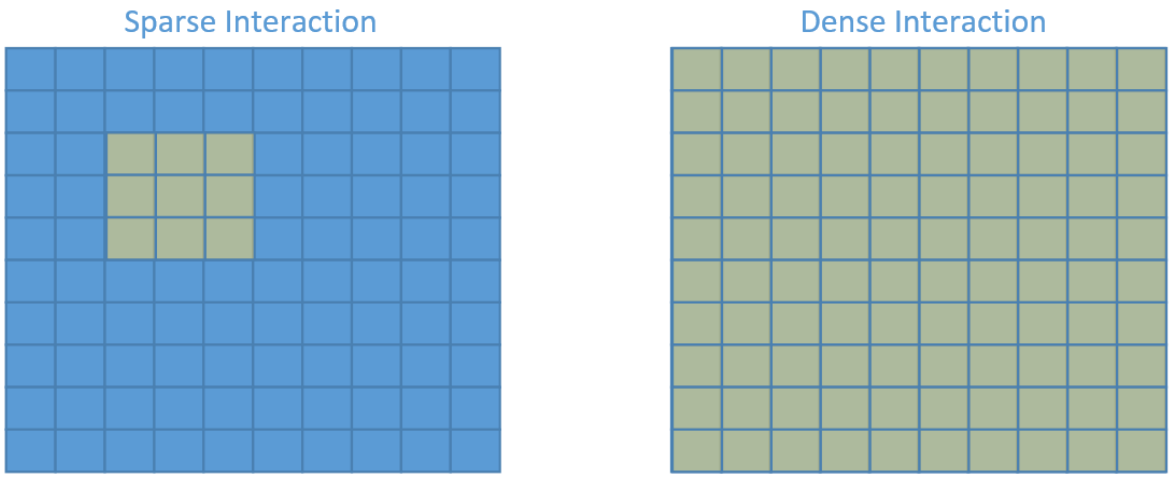

Figure 2.1: Visualization of sparse interaction (left) and dense interaction (right) between inputs and weights. With sparse interaction, the weights only interact on a subregion of the entire input. This is in contrast to the dense interaction of classic neural networks, where there's an interaction between every input and every output.

Parameter sharing allows for distinct outputs to be computed using the same set of weights. This drastically reduces the number of unique weights and allows for significant increases in network depth (i.e. deep networks) without the need to increase the amount of training data. These two design principles make CNN's poweful and efficient feature extractors.

Another type of deep neural network that has seen much success is the recurrent neural network (RNN). RNN's are designed to model data that display temporal structure, such as text and speech. In contrast to traditional neural networks, by *unfolding* time-steps as a computational graph, the network can learn to model data along the time dimension, using at each time step the raw input plus the output of the last time step's layer [23]. Because of the usually very deep structure of the recurrent network, computation of the gradient via back propagation can lead to the *vanishing gradient* problem, where the gradient starts to shrink and become very close to zero. The **Long Short-Term Memory (LSTM)** recur-

rent network proposed by [9] essentially solves this problem. A recent and simpler variation of the LSTM network is the Gated Recurrent Unit (GRU), which performs comparably well [4].

2.2 Word Embeddings

Another significant achievement of deep learning for natural language processing is the neural probabilistic language model. In their work, [1] proposed a neural network to model the probability of a word given its context $P(w_t|w_{t-k}, \ldots, w_{t-1})$.

The curse of dimensionality, manifested by the fact that a test word sequence $(w_{t-k}, \ldots, w_t)$ is likely to be different from all training sequences, was addressed by learning distributed representations of words. Words are represented as dense continuous vectors called **word embeddings**. Using word embeddings to model the likelihood of word sequences allowed generalization because the language model would give high probability to unseen word sequences if they were made up of words semantically similar to already seen words. Because of the underlying algorithm used to learn these word embeddings, similar words tend to lie closer to each other in embedding space. Thus, word embeddings are said to capture semantic relations and encode more information than just a word identifier (e.g. Bag of Words input representation).

2.3 Convolutional Neural Networks

As mentioned previously, 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 combinations 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 [27].

When dealing with image data, images are convolved with multiple filters, each convo-

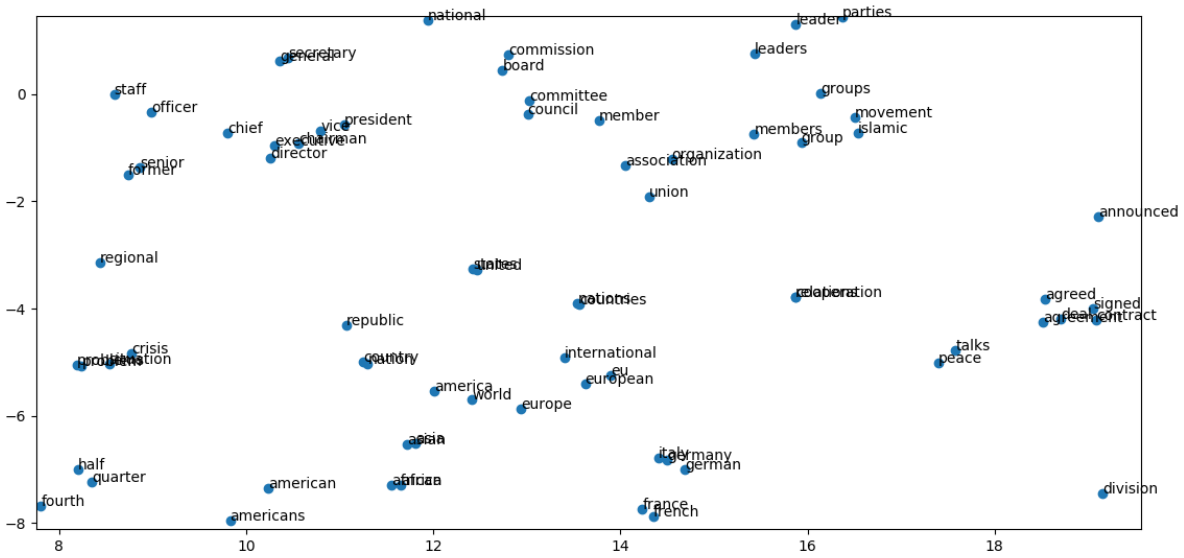

Figure 2.2: Visualization of embeddings using the t-distributed stochastic neighbor embedding (t-SNE) dimensionality reduction algorithm. Words with similar or related meanings tend to lie close to each other in embedding space.

lution applied to overlapping subimages called 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* by convolution of kernels with the inputs followed by a non-linear. The convolution operator of two-dimensional data such as images can be defined as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$

where m and n are the kernel's width and height respectively.

2.3.1 Input Representation: Word Embeddings for Convolutional Networks

We can take advantage of word embeddings to apply convolutions to text in a fashion similar to convolutions with image data and exploit the semantic information in the embeddings [13]. We apply the convolutions to overlapping sub-regions of the input text i.e. bi-grams, tri-grams, etc. After convolution, we apply a non-linear function such as $max(0, x)$ and reduce data dimensionality by pooling such as $x_{pool} = max(x_1, \ldots, x_n)$. Given a set of documents $\boldsymbol{s_1}, ..., \boldsymbol{s_m}$, 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 document $\boldsymbol{s}$ 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 the convolutional network receives this input sequence, each word index will be mapped to a corresponding embedding vector.

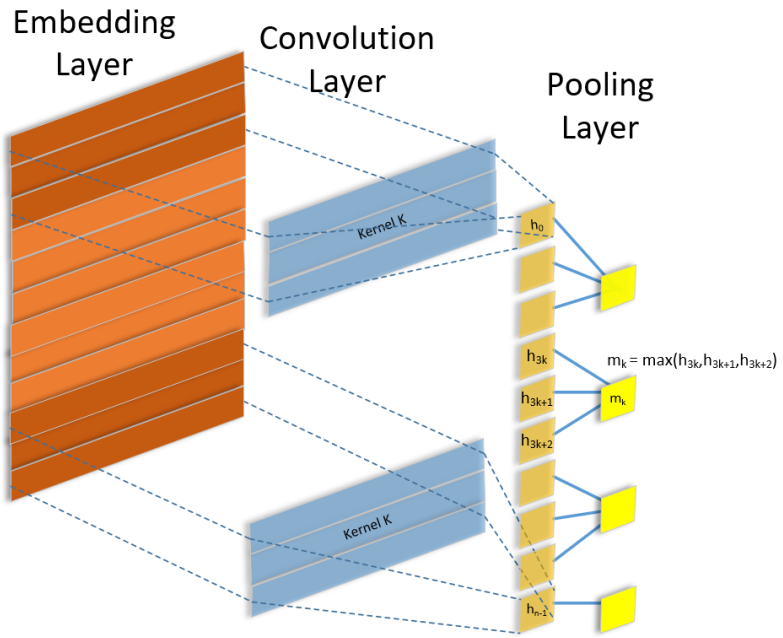

Figure 2.3: Visualization of a feature map with a single kernel. In this example, the kernel convolves tri-grams, or windows of three consecutive words. After convolution with all possible trigram context windows, max pooling is applied to reduce dimensionality. Here, the pool size is 3. This process is repeated as many times as there are kernels in the layer and their outputs are concatenated horizantally to yield a matrix of size *num filters* $\times$ (*input length - pool size + 1*).

Chapter 3

Model, Dataset, and Final Pipeline Description

3.1 Model Description

A standard architecture for convolutional neural networks for text sequence classification is an embedding layer followed by a feature map and a dense layer with a softmax activation to output class probabilities, as proposed by [13].

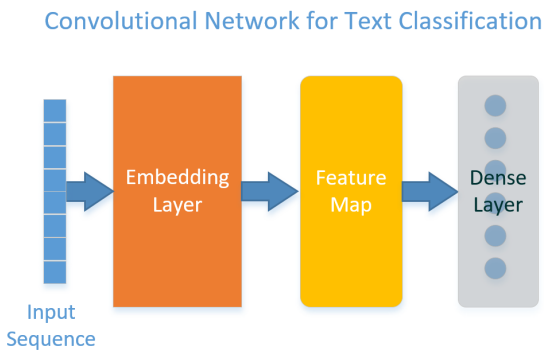

Figure 3.1: A common architecture for a convolutional network for text classification is an embedding layer followed by a feature map, then a dense layer to compute class probabilities.

This model has parameter sharing and sparse interaction properties inherent of convolutional neural networks, and thus is a good choice for efficiently extracting features from the data via convolutions and non-linearities. The pooling operation acts as a feature selection

mechanism and reduces the number of outputs in its layer. This reduction makes forward propagation an even more efficient process. Convolutional neural networks, however, are not designed to consider the data as having temporal structure, as is the case with text sequence data.

Another popular neural model for text classification is the recurrent neural network. This architecture is explicitly designed to treat data that display temporal structure. Because of this, the recurrent neural network is an excelent choice for text sequence data. At a basic level, this architecture is comprised of an embedding layer (to transform input sequences into word embeddings) followed by a recurrent layer, then a dense layer with the softmax activation to output probabilities. In the following sections we provide a technical description of the model layers introduced above. A more in-depth description of typical neural models used for natural language processing is presented by [26].

3.2 Layer Descriptions

3.2.1 Embedding Layer

The **embedding layer** layer is usually the first layer in a convolutional neural network that processes text input for classification tasks. Given an input sequence of words $\boldsymbol{s} = \{w_1, \ldots, w_n\}$, we transform it into a sequence of indexes $BoW(\boldsymbol{s}) = \boldsymbol{x} = x_1, ..., x_n$. The embedding layer maps each index x_i to a corresponding embedding and outputs an embedding matrix $\mathbf{E} \in \mathbb{R}^{n \times d}$, where d is the embedding size. These embeddings may be 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.

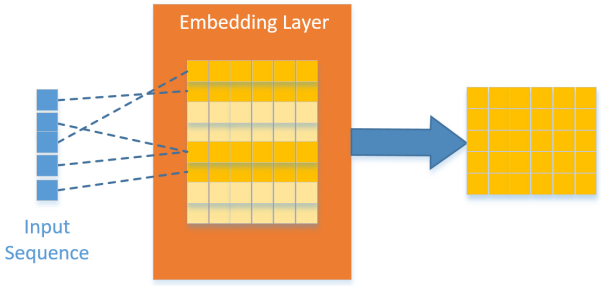

Figure 3.2: Visualization of an embedding layer. Word indexes are mapped to word embeddings via table lookups. This layer outputs a matrix comprised of stacked embeddings, one for each index.

3.2.2 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 with temporal structure, we use a one-dimensional convolutional layer to convolve with embedding n-grams through the input sequence [25]. As in Figure 2.3, 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. Concretely, the convolution between the ith word embedding vector $\boldsymbol{w}_i$ and a kernel $\boldsymbol{k}$ is given by:

$$\boldsymbol{s}(i) = \sum_{j=1}^{d} \boldsymbol{w}_i(j)\boldsymbol{k}(j)$$

,

where d is the embedding size.

After the convolution step, we apply a non-linearity to each computed feature. We chose the rectified linear activation (**ReLU**) function

$$ReLU(x) = max(0, x)$$

Although simple, this non-linearity is quite effective and thus widely used. We then use a

max pooling operation

$$maxpool(\boldsymbol{x}) = \max_i \boldsymbol{x}$$

to ouput a single value for each kernel as done in to [17]. Max pooling is a strong feature selection and dimensionality reduction mechanism. We pass to the next layer the single strongest activation from each kernel.

3.2.3 Gated Recurrent Unit Layer

A recurrent layer is designed to process data with temporal structure[21]. 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:

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

This is implemented by *unrolling* the layer along the time dimension as a sequence of layers, each corresponding to a moment in time. Because of this design, back-propagation of error through the unrolled layer can result in the vanishing gradient problem (i.e. the gradients shrink exponentially as they propagate backwards through the network layers).

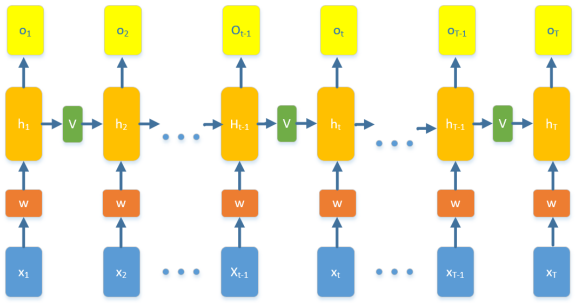

Figure 3.3: Visualization of a recurrent layer. Each hidden layer $\boldsymbol{h}^t$ is a function of the previous hidden layer $\boldsymbol{h}^{t-1}$ and the present input signal $\boldsymbol{x}^t$. The weights are shared across time steps in order for the model to generalize better.

A Gated Recurrent Unit layer is a type of recurrent layer designed to combat this condition [4]. It is a simplified version of the earlier LSTM layer [9], also designed to mitigate the vanishing gradient problem. In practice, this type of recurrent layer normally performs as well as its more complex predecesor. This layer models the latent features computed by the feature map as a time sequence. This means that it does not assume indepencies between computed activations and learns to model the temporal structure of the previous layer's output. This property makes the layer an adequate choice for text sequence data.

3.2.4 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. Given the output of the network's second to last layer $\boldsymbol{h}$, we can compute the unnormalized log probabilities

$$\boldsymbol{z} = \boldsymbol{\theta}^T \boldsymbol{h} + \boldsymbol{b}$$

where $\boldsymbol{b}$ is the bias vector.

Each z_i is a log probability of the input corresponding to class i. The **softmax** activation function is then applied to all the units, to finally output class probabilities:

$$\hat{y}_i = softmax(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

,

$$\hat{y}_i = P\{y = i|\boldsymbol{x}\}$$

In our experiments, a purely convolutional neural network finished training in a significantly shorter amount of time compared to the recurrent network. Because of the convolution and pooling operations, the CNN normally processed the data around 10 times faster than the RNN. The RNN, although slower, always achieved around 4% higher accuracy on all experiments.

After testing both architectures, the model we use in the rest of this work is a mixture of these two standards. After the embedding layer, we add a feature map to extract features from the raw inputs, then we add a GRU layer. This design choice incorporates the sequence modeling power of the RNN with the fast and efficient feature extraction of the CNN. This combination always led to the highest accuracy achieved with the RNN, with the reduced training times of the CNN. Out of all the features computed by a kernel, we select only the single strongest one (i.e. highest value). The pooling layer's output size is therefore equal to the number of kernels in our convolutional layer. In order to create a one-to-one correspondance between the network's inputs (the word indices in the input sequence) and the units in the recurrent layer, we chose to have a number of kernels equal to the network's input size (i.e. length of input sequence). This design choice proved to be very efficient and reached the highest accuracy percentages during our experiments.

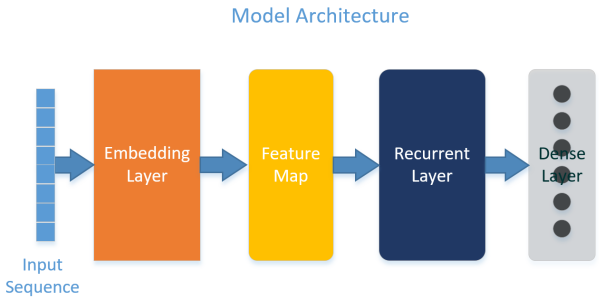

Figure 3.4: Model Architecture: embedding layer, followed by a feature map, and a recurrent layer. At the end, we have a fully connected layer with a softmax activation, which will output class probabilities.

The final network's architecture is as follows:

- Input layer: word index vector
- Embedding layer: maps word index vector to embedding matrix
- Feature Map:

  - Convolution layer: kernel size=5, $ReLU$ activation
  - Global MaxPooling layer: Outputs a scalar per kernel

- Gated Recurrent Unit layer with *tanh* activations
- Dense layer with *softmax* activations

Chapter 4

Towards Improving Model Performance: Embeddings and Data Augmentation

As mentioned in section 3.2.1, we can fine-tune the word embeddings via back-propagation and gradient descent (or some other gradient-based optimizer), just as any other weights in the neural network. In this case, the number of free parameters in our model increases drastically. It is easier to overfit a model with more free parameters when the amount of training data is limited, as is the case in our task. Another important aspect to consider when fine-tuning the embeddings is whether we lose the semantic information originally present in them. It is therefore a good idea to evaluate the effect of training the embeddings. In the following sections, we propose various ways to treat the embedding layer in our neural network in order to mitigate overfitting. We also consider reducing the dimensionality of the embeddings via Principal Component Analysis (PCA). Finally, we propose data augmentation techniques inspired by computer vision tasks but designed for text data with temporal structure, as is the case with our data.

4.1 Freezing the Embedding Layer

When we make our model's embeddings trainable, its number of free parameters highly increases. With inputs of size n, and embeddings of size d, we introduce $n \times d$ new parameters into the model. We therefore propose multiple embedding *trainability* schemes and

evaluate their effects on overall performance.

- Frozen Embeddings
  - With this scheme, we *freeze* the embedding layer altogether. We do not update the embedding vectors, so the model relies completely on its other parameters to fit the data.
- Freeze-Unfreeze
  - With the *freeze-unfreeze* scheme, we train the model with its embedding layer frozen until some stopping criterion is met. After this, we *unfreeze*, or make the embedding layer trainable, and retrain the model.
- Flash-Freeze
  - In the *flash-freeze* scheme, we allow the embedding layer to be trainable for a small number of epochs (e.g. three), then freeze it and retrain the model.
- Unfrozen Embeddings
  - We allow the embeddings layer to be trainable from the beggining to the end. We will refer to a model trained with unfrozen embeddings and without any of our proposed data augmentation schemes as the **baseline** model.

4.2 Principal Component Analysis and Dimensionality Reduction of the Embeddings

Principal Component Analysis (PCA) is a very popular linear transformation commonly used to reduce dimensionality and eliminate noise in a dataset [22]. It projects the data to a representation where the variables are linearly uncorrelated. Concretely, PCA projects the data onto new axes called principal components. These axes, or dimensions, are orthogonal

and selected in a way that minimizes reconstruction mean square error. Given a data matrix $\boldsymbol{X}$ with n dimensions, its principal components are the eigenvectors of the covariance matrix $\boldsymbol{X}^T\boldsymbol{X}$. The first principal component (the one with the largest associated eigenvalue) is the direction of most variation. The second principal component is the direction with the next largest variation, and so on. We can then choose to retain only the $k < n$ principal components that encode the most variation in the data.

We propose to apply PCA to the word embeddings in order to linearly decorrelate their dimensions and reduce their dimensionality. By reducing the dimensionality while preserving the majority of the variance in them (e.g. 95%), we will reduce the number of parameters introduced to the model while retaining most of the information from the original embeddings.

4.3 Dataset Augmentation: Shuffling and Noise Injection

Another common practice aimed to reducing overfitting 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 small translations, rotations, mirroring, and color intensity jitters via Pricipal Component Analysis [14]. All these tranformations should be subtle enough that the overall structure is preserved, but allow for the model to process distinct training data points.

The data augmentation techniques mentioned above are common practice in computer vision tasks such as image classification. In order to introduce more variance into our dataset, we propose data augmentation techniques inspired by computer vision tasks and designed for text sequence data. We propose to **shuffle** by randomly changing the order of words within a **context window**, or non-overlapping neighborhood of words through

the input text sequence. We further propose to inject small amounts of **noise** to the input sequence by randomly replacing words with words taken from the training vocabulary.

As long as the overall structure of the data (and label) is preserved, data augmentation normally leads to improved task performance [3][7] [10][14]. With that same rationale, we expect our proposed augmentation techniques to yield increased accuracy percentages. In other words, we expect that our changes will be subtle enough for the label to be preserved, but effective enough that we introduce more variation into our dataset and help cope with the limited number of training examples.

4.3.1 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 *context window* along non-overlapping neighborhoods the input sequence, randomly shuffling the words indexes inside it. For example using a context window of size 2 (bi-grams), an input sequence

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

could be shuffled to:

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

As it is common practice in computer vision tasks to apply small translations and rotations to images, we perform this operation to slightly perturb the temporal structure of our text sequence data. The rationale for this dataset augmentation technique is that small changes in the ordering of the words will result in 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 too harsh (e.g. a complete random permutation of the sequence).

4.3.2 Noise Injection

One common augmentation technique for image datasets is to add a small amount of noise. By replacing a relatively small number 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 noise to each training sample. Again, we aim to simulate a larger training sample while avoiding harsh changes to the original inputs. Specifically, each word in a text sequence is replaced with a specified probability with a word randomly chosen from our training vocabulary.

Algorithm 1 Add noise to input sequence

procedure NOISEINJECTION($\boldsymbol{x} = [x_1, ..., x_n], \mathbb{V}, p_{noise}$)
  for $k = 1$ to n **do**
    if $p_k \sim U(0, 1) \leq p_{noise}$ **then**
      $x_k \leftarrow x'_k \in \mathbb{V}$
    end if
  end for
  return $\boldsymbol{x}$
end procedure

4.4 Dataset Augmentation: Padding

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. it is 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 $|\mathbb{V}|$, with the index 0 being left for out-of-vocabulary words. When we pad our input sequence with 0s, we do not add any 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 information available during 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" (a publication about computer graphics). In this simple case, the label is determined by the word "graphics". To the human reader, this single informative word is enough to predict the label correctly although it 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 pad the sequence using this informative word instead of some meaningless token (e.g. 0's), we could increase the likelihood of the model extracting features from it. In other words, we make important words be present more frequently. We propose to pad input sequences with values found already within the input text sequence instead of 0's only.

4.4.1 Wrap Padding

Our proposed 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 (non-informative padding indexes) from the text sequence, but we do not have a selection mechanism and can miss useful words such as the word "*graphics*" in this example. Nonetheless, it is a simple approach to pad our data and hopefully increase the likelihood of encountering informative words during feature extraction.

4.5 Reducing Data Granularity: From Abstract to Sentences

Our last proposed dataset augmentation scheme is to break each training abtract into a set of sentences, and train 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.

Chapter 5

Experimental Results

5.1 Dataset Descriptions

We gathered a scientific publication abstract dataset from Arxiv.org. We obtained publications from the physics, mathematics, computer science, and quantitative biology disciplines. For each discipline, we considered each **topic** as a class. The topics for each discipline are listed as follows:

Astrophysics

- Astrophysics of Galaxies
- Earth and Planetary Astrophysics
- High Energy Astrophysical Phenomena
- Instrumentation and Methods for Astrophysics
- Solar and Stellar Astrophysics

Quantitative Biology

- Biomolecules
- Neurons and Cognition
- Populations and Evolution
- Quantitative Methods

Physics

- Accelerator Physics
- Atomic Physics
- Biological Physics
- Chemical Physics
- Classical Physics
- Computational Physics
- Data Analysis, Statistics and Probability
- Fluid Dynamics
- General Physics
- Instrumentation and Detectors
- Optics
- Physics and Society
- Plasma Physics

Computer Science

- Artificial Intelligence
- Computation and Language
- Computational Complexity
- Game Theory
- Computer Vision
- Computers and Society
- Cryptography and Security

- Data Structures and Algorithms
- Databases
- Discrete Mathematics
- Distributed, Parallel, and Cluster Computing
- Information Retrieval
- Information Technology
- Machine Learning
- Logic in Computer Science
- Networking and Internet Architecture
- Neural and Evolutionary Computing
- Social and Information Networks
- Software Engineering
- Systems and Control

Mathematics

- Algebraic Geometry
- Algebraic Topology
- Analysis of Partial Differential Equations
- Category Theory
- Classical Analysis and Ordinary Differential Equations
- Combinatorics
- Commutative Algebra
- Complex Variables

- Dynamical Systems
- Functional Analysis
- Geometric Topology
- Group Theory
- K-Theory and Homology
- Logic
- Mathematical Physics
- Metric Geometry
- Number Theory
- Numerical Analysis
- Operator Algebras
- Optimization and Control
- Probability
- Quantum Algebra
- Representation Theory
- Rings and Algebras
- Statistics Theory
- Symplectic Geometry

Each class has 5000 examples. Because of this class balance, it is appropriate to report the prediction accuracy on the test set rather than using precision, recall, and f-measure. We use a 70/30 split for training and testing. We apply simple preprocessing to the texts by removing non-alphanumeric characters, and converting to lower case characters. The dataset distribution is shown in the table below.

Table 5.1: Dataset distribution. We obtained 5000 abstracts for each class. We use 70% of the data for training and 30% for testing.

Department	Number of Labels	Training Size	Testing Size
Astrophysics	5	17500	7500
Physics	13	45015	19293
Computer Science	20	63396	27170
Mathematics	26	87705	37588
Quant. Bio	5	11006	4717

We used the GloVe embedding set for our pretrained embeddings [20]. This set consists of 400,000 words, 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 method 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 third proposed approach is to train with the embeddings unfrozen for a small number 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 beggining to the end.

Table 5.2: Accuracy results on all datasets with proposed embedding training schemes. The numbers in parenthesis represent the number of epochs until the model converged.

Dataset	Frozen	Not-Frozen	Freeze-Unfreeze	Flash-Freeze
Astrophysics	0.751±0.005(6.6)	0.765±0.003(7.1)	0.761±0.002(7.4)	**0.773±.001(6)**
Physics	0.735±0.007(9.7)	0.764±0.004(9.5)	0.767±0.002(9.8)	**0.779±0.001(6)**
Computer Science	0.644±0.005(12.0)	0.682±0.004(10.5)	0.684±0.003(14.7)	**0.701±0.002(6)**
Mathematics	0.609±0.006(15.3)	0.648±0.003(13.3)	0.657±0.004(15.6)	**0.665±0.001(6)**
Quant. Biology	0.812±0.006(7.2)	0.828±0.005(7.4)	0.820±0.003(7.8)	**0.834±0.002(6)**
Average Accuracy	0.710	0.737	0.738	**0.750**
Average Epochs	10.2	9.6	11.1	**6**

We observe that the Flash-Freeze method achieves faster convergence, with accuracy competent or better than the accuracy of all other methods. We observed that when using the *flash-freeze* embedding training method, we reached a good stopping point after a small number of epochs (e.g. 6). This is useful because it removes the need to use a validation split for early stopping. From here on, all our models are trained using the *flash-freeze* technique for 6 epochs: 3 epochs training the embeddings and 3 epochs with the embeddings frozen.

All other methods required a validation split for early stopping. We used a 90/10 split for training and validation respectively.

5.3 Data Augmentation Results

In this section we show the experimental results obtained using the 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. We refer the model trained using the *non-frozen* scheme and without any of the proposed augmentation techniques as the **baseline**.

5.4 Astrophysics

Table 5.3: Test accuracy on the astrophysics dataset.

Astrophysics Test Accuracy					
	No Change	Embedding PCA	Wrap Padding	PCA & Padding	Sentence Split
No Aug	0.773	0.774	0.778	0.777	0.769
Noise	0.771	**0.779**	0.768	0.777	0.765
Shuffle	0.772	0.768	0.770	0.764	0.766
Ensemble	0.771	0.777	0.770	0.777	0.770
Baseline: 0.765					

5.5 Physics

Table 5.4: Test accuracy on the physics dataset.

Physics Test Accuracy					
	No Change	Embedding PCA	Wrap Padding	PCA & Padding	Sentence Split
No Aug	0.789	0.789	**0.790**	**0.790**	0.786
Noise	0.774	0.757	0.783	0.782	0.743
Shuffle	0.772	0.761	0.787	0.787	0.766
Ensemble	0.783	0.775	0.789	0.789	0.765
Baseline: 0.764					

5.6 Mathematics

Table 5.5: Test accuracy on the mathematics dataset.

Mathematics Test Accuracy					
	No Change	Embedding PCA	Wrap Padding	PCA & Padding	Sentence Split
No Aug	0.663	0.653	**0.664**	0.657	0.654
Noise	0.616	0.612	0.618	0.624	0.618
Shuffle	0.625	0.643	0.640	0.646	0.636
Ensemble	0.638	0.631	0.635	0.639	0.641
Baseline: 0.648					

5.7 Computer Science

Table 5.6: Test accuracy on the computer science dataset.

Computer Science Test Accuracy					
	No Change	Embedding PCA	Wrap Padding	PCA & Padding	Sentence Split
No Aug	**0.703**	0.690	0.701	0.699	0.701
Noise	0.655	0.665	0.688	0.675	0.655
Shuffle	0.682	0.663	0.701	0.685	0.685
Ensemble	0.678	0.680	0.696	0.689	0.687
Baseline: 0.682					

5.8 Quantitative Biology

Table 5.7: Test accuracy on the quantitative biology dataset.

Quantitative Biology Test Accuracy					
	No Change	Embedding PCA	Wrap Padding	PCA & Padding	Sentence Split
No Aug	0.836	0.837	0.839	0.838	0.833
Noise	0.827	0.835	0.837	0.834	0.812
Shuffle	0.831	0.827	0.831	0.829	0.820
Ensemble	0.839	0.840	**0.840**	0.840	0.830
Baseline: 0.828					

We conclude from our experiments that the incorporating of the embeddings as free parameters of the model helped achieve higher accuracy than without them. We expected that the *Freeze-Unfreeze* would achieve the highest test accuracy. The *Flash Freeze* embedding trainability scheme achieved the highest test accuracy in the least number of training epochs. It is not clear from our experiments if the data augmentation schemes proposed lead to improvements. In our tests, these techniques yielded slightly lower accuracy than models trained without them.

Chapter 6

Conclusions

We presented an evaluation of recurrent convolutional neural networks for classification of publication abstracts from multiple science and engineering disciplines. We proposed techniques that resulted in a small increase in test accuracy compared to our baseline model.

One conclusion we can make from this study is that recurrent convolutional networks are efficient and effective feature extractors even with moderate amounts of training data. The combination of convolution and global max pooling provided efficient feature extraction, while the recurrent layer learned to model the temporal structure of our data. The model with this combination of layers reached the highest test accuracy in our experiments.

We proposed several embeding *trainability* schemes to manipulate the word embeddings during training. The results we obtained were different than what we had anticipated. Since the embeddings were already learned (pre-trained) from a previous task using a very large dataset to represent words in a distributed way and to capture semantic relations, we originally believed that changing their values would only overfit the training set and thus significantly reduce testing set accuracy. We expected that the *freeze-unfreeze* method would achieve faster convergeance and higher accuracy compared to all other proposed embedding methods. With this scheme, we hoped the model would converge then increase accuracy by allowing the embeddings to be trainable until convergeance determined by a stopping criterion. Although the model test accuracy did increase with this scheme, the *flash-freeze* method outperformed all others. By allowing the embeddings to be fine-tuned for a small number of iterations (three in our tests), then allowing the model to rely only on its weights (frozen embeddings), we observed we could finish training after a constant

number of epochs. For our task, three epochs were enough to finish training. Not only did the *flash-freeze* method achieve the highest overall performance, but it also removed the need for a validation set to use for early termination. This increased, albeit by not much, the amount of data available for training purposes since we could stop training after a constant number of epochs. A validation set for early stopping was crucial before we incorporated the *freeze-flash* stopping mechanism; the model would quickly overfit and test accuracy would decrease.

One interesting observation from our experiments is that the use of PCA on the embeddings yielded favorable results. The dimensionality reduction and linear decorrelation of the embedding variables did not affect the model's generalization when a large enough variance percentage was retained. This allowed us to reduce the number of free parameters while retaining competent performance compared to using the entire embedding set.

Our tests were inconclusive in determining whether the proposed noise injection and word shuffling augmentations yielded improved results. Overall, we obtained slightly lower test accuracy with models trained using these augmentation schemes. We believe that the model learns the structure of the data and therefore develops a sort of translation invariance. This could explain why shuffling the order of words as a data augmentation technique yielded results similar than without the shuffling. A larger dataset and more tests are necessary to decide if this technique is indeed beneficial or not.

We proposed to split the abstracts into sentences and train the model using this finer data granularity. This would increase the amount of training examples the model could perform weight updates with. By increaseing the number of examples available for training, we hoped to obtain better test accuracy. Our results were contrary to this, as we obtained higher accuracy when we treated an entire abstract as a single training example. We believe this is due to the noise present in our data, as well as the increased likelihood of class ambiguity when processing a single sentence versus an entire paragraph.

6.1 Future Work

We would like to extend our research to incorporate datasets of various sizes. This will give us further insight on how well recurrent convolutional neural networks learn with a range of different training set sizes. Another further augmentation technique we would like to further study is padding.

The application of PCA to the embedding matrix yielded unexpected and favorable results. Because of the projection of the embeddings into a different set of axes, we believed we would lose the performance gain of using pre-trained embeddings. The results were contrary to this, as we obtained similar performance withe the PCA embeddings with a smaller number of free parameters. It would be interesting to apply non-linear dimensionality reduction techniques (e.g. principal curves, autoencoders) and evaluate the effects.

In our work, we padded an input sequence with zeros to enforce uniform input size. We proposed to pad our inputs by *wrapping* around the text to remove zero-padding. It is unclear from our experiments if this augmentation helped improve our model's performance directly. We would like to extend our proposed padding to have a selection mechanism to pad with words that could potentially increase the information content in a training sample (i.e. keywords). Having such a selection mechanism, we could pad an input sequence using words selected as informative or meaningful based on some specified criteria rather than simply wrapping around the sequence.

Lastly, we are interested in seeing the effect of synonyms as a data augmentation procedure. This could be to replace words with their actual synonyms using table lookups, or to compute nearest neighbors in embedding space and replace words based on this criterion.

References

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

[2] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[4] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[5] Alexey Dosovitskiy, Jost Tobias Springenberg, Martin Riedmiller, and Thomas Brox. Discriminative unsupervised feature learning with convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 766–774, 2014.

[6] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[8] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

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

[10] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. In *Advances in Neural Information Processing Systems*, pages 2017–2025, 2015.

[11] Thorsten Joachims. *Learning to classify text using support vector machines: Methods, theory and algorithms*. Kluwer Academic Publishers, 2002.

[12] Rie Johnson and Tong Zhang. Effective use of word order for text categorization with convolutional neural networks. *arXiv preprint arXiv:1412.1058*, 2014.

[13] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[15] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[16] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[17] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013.

[18] Hans Peter Luhn. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of research and development*, 1(4):309–317, 1957.

[19] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014.

[20] 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.

[21] David E Rumelhart, Paul Smolensky, James L McClelland, and G Hinton. Sequential thought processes in pdp models. *Parallel distributed processing: explorations in the microstructures of cognition*, 2:3–57, 1986.

[22] Jonathon Shlens. A tutorial on principal component analysis. *CoRR*, abs/1404.1100, 2014.

[23] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150, 1995.

[24] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[25] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and K Lang. Phoneme recognition: neural networks vs. hidden markov models vs. hidden markov models. In *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*, pages 107–110. IEEE, 1988.

[26] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of cnn and rnn for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.

[27] Y-T Zhou, Rama Chellappa, Aseem Vaid, and B Keith Jenkins. Image restoration using a neural network. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(7):1141–1151, 1988.

Curriculum Vitae

Jonathan Quijas was born on September 18, 1991. He graduated from Franklin High School, El Paso, Texas, in the spring of 2009. He entered The University of Texas at El Paso in August that same year. While pursuing his bachelor's degree in Computer Science, he worked as a clerical assistant at the same departments front office, and later obtained a programming position at the University of Texas at El Paso's Lockheed Martin Storefront. He joined the University's Vision and Learning Laboratory in summer of 2011, where focused his studies on Computer Science towards Machine Learnig and Computer Vision. He obtained his bachelor's degree in December 2013 and shortly after enrolled as a graduate student pursuing a Master's degree in Computer Science in Spring 2014.

Permanent address: 1000 Desierto Seco Drive
El Paso, Texas 79912

This thesis was typed by Jonathan Quijas.