Table of Contents

[Abstract 3](#_Toc435354092)

[Introduction 3](#_Toc435354093)

[Previous and Related Works 3](#_Toc435354094)

[Artificial neural networks 3](#_Toc435354095)

[Language model and Word-Vector representation 5](#_Toc435354096)

[SoftMax classifier 5](#_Toc435354097)

[MV-RNN 7](#_Toc435354098)

[RNTN 8](#_Toc435354099)

[Apache Spark 9](#_Toc435354100)

[Implementation 9](#_Toc435354101)

[Results 9](#_Toc435354102)

[Discussion 9](#_Toc435354103)

[References 9](#_Toc435354104)

[Appendix 11](#_Toc435354105)

# Abstract

Capturing the meaning of words in semantic word spaces has been very useful. Yet, using them without understanding compositionality in tasks such as sentiment analysis, failed to capture complex, although common, human expressions. Furthermore, in the modern big data applications there is a need to maximize performance using distributed systems. To tackle these problems we implemented an innovative, state-of-the-art approach of sentiment analysis called Recursive Neural Tensor Network (RNTN) with the Apache Spark framework.

Since RNTN is a general, deep-learning framework, used in other fields such as computer vision, we believe that this research (applying RNTN in a distributed manner for sentiment analysis) is also valuable for additional applications.

# Introduction

Sentiment analysis incorporates the practice of natural language processing, text analysis and computational linguistics to classify subjective information in source materials1. The ability to determine the attitude of a speaker or a writer with respect to some topic in an unmanned fashion is drawing increasing interest in various fields: business intelligence, stock trading, public relation etc. However, most bag-of-word based machine learning predictions (such as Softmax, KNN classifiers) ignore sentence structure, namely: order of words and their context2. Furthermore, they often fail in a fine-grained classification (multiple sentiment classes and not just binary “positive”/”negative” classification).

Artificial Neural networks (ANN) are often a good candidate for such complex, multi-feature analysis3. In order to handle input of variable size (sentences of varying length) a Recursive Neural Networks (RNN) is often used. In the evolution of using RNNs several approaches were explored: using RNN to learn a word-phrase vector representation4 and, in the context of sentiment analysis, a matrix-vector RNN (MV-RNN) was used to learn both the representation and the interactions between words in a phrase2.

In this paper we implemented a novel approach, introduced by researchers from Stanford University, using tensors instead of the MV-RNN. Tensors are aimed, in a single, more powerful composition function, to capture aggregated meaning from smaller constituents more accurately than many input specific ones.

However, RNTN (like all ANN) suffers a huge drawback in real-life applications: even with optimization techniques, feature reduction and a small dataset, ANN training is a long, time consuming process due to its iterative nature. Google, which heavily relies on ANN, introduced a paradigm for ANN training in a distributed manner called Downpour\*, thus enabling faster and more accurate ANN models.

In this project we implemented the Downpour paradigm (using the Apache Spark framework) and present a concurrent implementation of the RNTN algorithm.

# Previous and Related Works

## Artificial neural networks

Inspired by biological neural networks (the central nervous systems of animals, in particular the brain), artificial neural networks (ANNs) are a family of statistical learning algorithms that are used to estimate or approximate functions, that can depend on a large number of inputs, and are generally unknown5,6.

A neural network is not just a complex system, it is an adaptive one: it changes its internal structure based on the information flowing through it.

A perceptron is the most basic building block of a neural network (a single Neuron). A perceptron consists of one or more inputs, a processor, and a single output.

The generalized Perceptron Algorithm:

1. For every input, multiply that input by its weight.

2. Sum all of the weighted inputs.

3. Compute the output of the perceptron based on that sum passed through an **activation function**.

Although, there are a number of optional choices of activation functions, we focus on the tanh function7.

Training the neural networks follows the general flow (Backpropagation8):

1. Provide the perceptron with inputs for which there is a known answer.

2. Ask the perceptron to guess an answer.

3. Compute the error. (Did it get the answer right or wrong?)

4. Adjust all the weights according to the error.

5. Return to Step 1 and repeat.

## Language model and Word-Vector representation

While methods such as *tf-idf* perform well for modeling large documents, they are a bag-of-words approach and do not capture context, for example:

*“a pretty bad performance by the actress”*

“*a bad performance by the pretty actress”*

would have very similar vector representation while

“*the country of my birth*”

“*the place where I was born*”

(although have similar meanings) would have very different representations.

Statistical language model assigns a probability to a sequence of *m* words P(w1…,w*m*) by means of a probability distribution.

The language model provides context to distinguish between words and phrases9.

The question that arises is: How to represent a word as a vector based on a language model?

To answer this we use neural word vectors. These vectors are trained in an unsupervised fashion to capture distributional similarities, then can also be fine-tuned and trained to specific tasks such as sentiment detection.

Originally, this method used an ANN to learn the Word-Vector representation by looking at a word and its context as a positive training sample, then a random word in that same context gives a negative training sample. The ANN computes a score for both sentences and uses backpropagation to adjust the word-vector itself (as opposed to the weights) so that the score of the negative sample is lower10.

## SoftMax classifier

A SoftMax classifier is a supervised learning algorithm, which can be used in several problems including text classification. It is a regression model that generalizes the logistic regression to classification problems where the output can take more than two possible values.

SoftMax regression requires the estimation of a coefficient theta for every word and category combination.

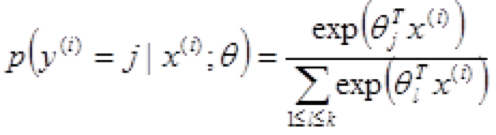
The sign and the value of this coefficient show whether the existence of the particular word within a document has a positive or negative effect towards its classification to the category.

Training dataset consist of *m* (*xi,yi*) pairs, where *xi* is the *i*th document vector representation and *yi* is the document label, and *k* be the number of all possible classes.

Let {w1,…,wn} be the set of n words that can appear within our texts.

All the documents within our training dataset will be represented as vectors with 0s and 1s that indicate whether each word of our vocabulary exists within the document. In addition, all vectors will include an additional “1” element for the intercept term.

In SoftMax Regression the probability given a document x to be classified as y is equal to:



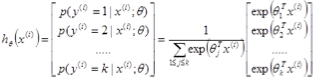
θ*i* -vector stores the coefficients of *i*th category for each of the n words

*x*(*i*) – *i* th document, as a vector

*y*(*i*) – label of the *i* th document 

*j* – value of the label (1≤*j*≤*k*)

Thus the hypothesis function will return a *k* dimensional vector with the estimated probabilities:



Training requires us to minimize the following cost function:



where:

*m* – number of documents

*i* – sample iterator (sample is a <doc,label> tuple)

1{\*} is the indicator function, so that 1{a true statement} = 1, and 1{a false statement} = 0. For example, 1{2 + 2 = 4} evaluates to 1; whereas 1{1 + 1 = 5} evaluates to 0

We minimize the cost function using the partial derivative:



The RNTN uses SoftMax classifier at each computational node (neuron) to classify the composite meaning.

## MV-RNN

As stated above, a key difficulty in NLP related tasks in general, and sentiment classification in particular, is capturing the meaning of composite sentences.

MV-RNN4 is a step in the evolution of machine learning algorithms that try to attack that problem by representing each term, and later each combined terms, as both a vector and a matrix. The vector representation describes the semantic of the word/composite (similar to the method described as the Word-Vector representation above). The Matrix component of the representation however tries to represent the notion of words-as-operators.

For example, consider the sentences:

“a *very good movie*”

“*a good movie*”

“not a good movie”

In the first sentence the term “very” acts as an operator on the “good” term (amplifies the meaning of “good”), in the second the “a” term has a neutral effect and in the third “not” acts as a negative operator.

At each layer of the RNN the composition function combined terms in the parse tree to represent both the matrix and vector (in two separate composition functions) thus, aggregating the meaning of the whole sentence.

## RNTN

Following the MV-RNN method, the researchers were motivated by the following question: “Can a single, more powerful composition function perform better and compose aggregate meaning from smaller constituents more accurately than many input specific ones?”2

As a result, a new model called the Recursive Neural Tensor Network (RNTN) was proposed. The main idea is to use the same, tensor-based composition function for all nodes.

Tensors are geometric objects that describe linear relations between vectors, scalars, and other tensors. In their research they have implemented the RNTN to classify movie reviews into 5 sentiment classes: very bad, bad, neutral, good and very good.

The RNTN can be described as the following, generalized flow:

1. Each word is represented as a *d*-dimensional vector. Initialize all word vectors by randomlysampling each value from a uniform distribution. All the word vectors are stacked in the word-embedding matrix *L*.
2. Use the word vectors as parameters to optimize and as feature inputs to a *SoftMax* classifier. For classification into five classes, we compute the posterior probability over labels given the word vector via:



Where *Ws* is the sentiment classification matrix ()

1. Compute the output of a tensor product via the following vectorized notation and the equivalent but more detailed notation for each slice:



(*b,c* are the vector representation)

1. Finally, each layer is computed:

where: *f* is tanh function.

The error as a function of the RNTN parameters θ = (*V, W, Ws, L*) for a sentence is:



The backpropagation process will be described in detail in the final paper.

## Apache Spark

At its base, Apache Spark is an in-memory data processing framework with two distinct capabilities: the first it is capable of running in a distributed mode (thus handling large amounts of data) and second it's in-memory primitives provide performance of up to 100 times faster than MapReduce (for certain applications).13

Spark Core is the foundation of the overall project. With its language-integrated API in Java, Python and Scala, it provides distributed task dispatching, scheduling, and basic I/O functionalities. The central programming abstraction is called Resilient Distributed Datasets, a logical collection of data partitioned across machines. RDDs can be created by referencing datasets in external storage systems, or by applying coarse-grained transformations (e.g. map, filter, reduce, join) on existing RDDs.

By allowing user programs to load data into a cluster's memory and query it repeatedly, Spark is well suited to machine learning algorithms14.

## Downpour SGD

Stochastic gradient descent (SGD) is perhaps the most commonly used optimization procedure for training deep neural networks [25, 26, 3]. Unfortunately, the traditional formulation of SGD is inherently sequential, making it impractical to apply to very large data sets where the time required to move through the data in an entirely serial fashion is prohibitive. To apply SGD to large data sets, we introduce Downpour SGD, a variant of asynchronous stochastic gradient descent that uses multiple replicas of a single DistBelief model. The basic approach is as follows: We divide the training data into a number of subsets and run a copy of the model on each of these subsets. The models communicate updates through a centralized parameter server, which keeps the current state of all parameters for the model, sharded across many machines (e.g., if we have 10 parameter server shards, each shard is responsible for storing and applying updates to 1/10th of the model parameters) (Figure 2). This approach is asynchronous in two distinct aspects: the model replicas run independently of each other, and the parameter server shards also run independently of one another. In the simplest implementation, before processing each mini-batch, a model replica asks the parameter server service for an updated copy of its model parameters. Because DistBelief models are themselves partitioned across multiple machines, each machine needs to communicate with just the subset of parameter server shards that hold the model parameters relevant to its partition. After receiving an updated copy of its parameters, the DistBelief model replica processes a mini-batch of data to compute a parameter gradient, and sends the gradient to the parameter server, which then applies the gradient to the current value of the model parameters. It is possible to reduce the communication overhead of Downpour SGD by limiting each model replica to request updated parameters only every nf etch steps and send updated gradient values only every npush steps (where nf etch might not be equal to npush). In fact, the process of fetching 4 parameters, pushing gradients, and processing training data can be carried out in three only weakly synchronized threads (see the Appendix for pseudocode). In the experiments reported below we fixed nf etch = npush = 1 for simplicity and ease of comparison to traditional SGD. Downpour SGD is more robust to machines failures than standard (synchronous) SGD. For synchronous SGD, if one machine fails, the entire training process is delayed; whereas for asynchronous SGD, if one machine in a model replica fails, the other model replicas continue processing their training data and updating the model parameters via the parameter servers. On the other hand, the multiple forms of asynchronous processing in Downpour SGD introduce a great deal of additional stochasticity in the optimization procedure. Most obviously, a model replica is almost certainly computing its gradients based on a set of parameters that are slightly out of date, in that some other model replica will likely have updated the parameters on the parameter server in the meantime. But there are several other sources of stochasticity beyond this: Because the parameter server shards act independently, there is no guarantee that at any given moment the parameters on each shard of the parameter server have undergone the same number of updates, or that the updates were applied in the same order. Moreover, because the model replicas are permitted to fetch parameters and push gradients in separate threads, there may be additional subtle inconsistencies in the timestamps of parameters. There is little theoretical grounding for the safety of these operations for nonconvex problems, but in practice we found relaxing consistency requirements to be remarkably effective. One technique that we have found to greatly increase the robustness of Downpour SGD is the use of the Adagrad [10] adaptive learning rate procedure. Rather than using a single fixed learning rate on the parameter sever (η in Figure 2), Adagrad uses a separate adaptive learning rate for each parameter. Let ηi,K be the learning rate of the i-th parameter at iteration K and ∆wi,K its gradient, then we set: ηi,K = γ/qPK j=1 ∆wi,j2. Because these learning rates are computed only from the summed squared gradients of each parameter, Adagrad is easily implemented locally within each parameter server shard. The value of γ, the constant scaling factor for all learning rates, is generally larger (perhaps by an order of magnitude) than the best fixed learning rate used without Adagrad. The use of Adagrad extends the maximum number of model replicas that can productively work simultaneously, and combined with a practice of “warmstarting” model training with only a single model replica before unleashing the other replicas, it has virtually eliminated stability concerns in training deep networks using Downpour SGD (see results in Section 5)

# Implementation

# Results

# Discussion

# References

1Wikipedia. (2015/2/17). Sentiment Analysis. (Wikimedia Foundation, Inc.) Retrieved 2 22, 2015, from Wikipedia: The Free Encyclopedia: http://en.wikipedia.org/wiki/Sentiment\_analysis

2Socher, Richard, et al. "Recursive deep models for semantic compositionality over a sentiment treebank." Proceedings of the conference on empirical methods in natural language processing (EMNLP). Vol. 1631. 2013.

3Socher, Richard, et al. "Semi-supervised recursive autoencoders for predicting sentiment distributions." Proceedings of the Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, 2011.

4Socher, Richard, et al. "Semantic compositionality through recursive matrix-vector spaces." Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning. Association for Computational Linguistics, 2012.

5Shiffman, Daniel, Shannon Fry, and Zannah Marsh. The nature of code. D. Shiffman, 2012.

6Schalkoff, Robert J. Artificial neural networks. New York: McGraw-Hill, 1997.

7Sibi, P., S. Allwyn Jones, and P. Siddarth. "Analysis of Different Activation Functions Using Back Propagation Neural Networks." Journal of Theoretical and Applied Information Technology 47.3 (2013): 1264-1268.

8Hecht-Nielsen, Robert. "Theory of the backpropagation neural network." Neural Networks, 1989. IJCNN., International Joint Conference on. IEEE, 1989.

9de Kok, Daniël, and Harm Brouwer. "Natural language processing for the working programmer." (2011).

10Collobert, Ronan, and Jason Weston. "A unified architecture for natural language processing: Deep neural networks with multitask learning."Proceedings of the 25th international conference on Machine learning. ACM, 2008.

11Shvachko, Konstantin, et al. "The hadoop distributed file system." Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010.

12Dean J, Ghemawat S: MapReduce: A Flexible Data Processing Tool. Communications of the ACM 2010, 53(1):72-77.

13Xin, Reynold S., et al. "Shark: SQL and rich analytics at scale." Proceedings of the 2013 ACM SIGMOD International Conference on Management of data. ACM, 2013.

14Matei Zaharia. Spark: In-Memory Cluster Computing for Iterative and Interactive Applications. Invited Talk at NIPS 2011 Big Learning Workshop: Algorithms, Systems, and Tools for Learning at Scale.

# Appendix