The Academic College of Tel Aviv Yaffo

School of Computer Science

M.Sc. Degree Final project

Submitted by: Uri Rosenberg

Guide/Advisor: Dr. Uzi Hadad

Sentiment analysis using Recursive Neural Tensor Network in a distributed environment

Contents

[Abstract 2](#_Toc440297709)

[Introduction 3](#_Toc440297710)

[Previous and Related Works 5](#_Toc440297711)

[Artificial neural networks 5](#_Toc440297712)

[Language model and Word-Vector representation 6](#_Toc440297713)

[SoftMax classifier 7](#_Toc440297714)

[MV-RNN 10](#_Toc440297715)

[RNTN 10](#_Toc440297716)

[Apache Spark 11](#_Toc440297717)

[Downpour SGD and DeepDist 12](#_Toc440297718)

[Distributed RNTN 14](#_Toc440297719)

[RNTN Explained 14](#_Toc440297720)

[RNTN single process implementation 17](#_Toc440297721)

[Distributed RNTN design 18](#_Toc440297722)

[Data partioning 19](#_Toc440297723)

[Cluster management 19](#_Toc440297724)

[Fault tolerance 20](#_Toc440297725)

[Distributed RNTN implementation 20](#_Toc440297726)

[Data 22](#_Toc440297727)

[Experiment and Results 23](#_Toc440297728)

[Conclusions 24](#_Toc440297729)

[References 25](#_Toc440297730)

[Appendix 27](#_Toc440297731)

[A. Class RNTN 27](#_Toc440297732)

[B. Distrntn.py 31](#_Toc440297733)

[C. Full project 37](#_Toc440297734)

# 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). Once executed as a single process on a single machine, we explored implementing RNTN in a distributed fashion. Basing our solution on a general distributed neural network model called Downpour SGD, we present in this work an implementation and analysis of 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 work 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 the aggregated meaning of sentence components (words and n-grams). The resulting ANN is called Recursive Neural Tensor Network (RNTN) and appears to outperform common machine learning sentiment detection algorithms. As a generalized RNTN flow, we have a neuron that combines a pair of leafs (or nodes) from a sentence parse tree using a tensor layer (that captures interaction between combined elements), adds the layer weights and words vectors product. The result is then “squashed” by the activation function. Before propagating the result to the next layer (which repeats the process recursively) a SoftMax classifier classifies the sentiment of the neuron output.

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 SGD5, thus enabling faster and more reliable ANN model learning. It does so by allocating the training data to each node in a cluster along with a replica of the model parameters. Each node computes the required adjustments that needs to be done to the model, and sends these updates to a centralized server which, in turn, updates the model.

In this project we implemented the Downpour SGD paradigm using the Apache Spark framework, a general engine for large-scale data processing, and present a concurrent implementation of RNTN. We show that this method reduced the training time from almost 4 hours to less than 20 minutes and, since the process is distributed over multiple machines, rendered it more resilient.

# 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. They are used to estimate or approximate functions, that depend on a large number of parameters, generally unknown6,7.

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 or node). 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.

The activation function defines the output of the node: will and how much the neuron will fire. Although, there are a number of optional choices of activation functions, we focus on the tanh function8.

Training the neural networks follows the general flow (Backpropagation9):

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.

4. Adjust all the weights according to the error.

5. Return to Step 1 and repeat.

ANN can be trained to produce word embedding. Word embedding is a set of language modeling and feature learning techniques in Natural Language Processing (NLP). The word embedding matrix maps words to vectors of real numbers, thus enabling the harnessing of mathematical procedures to compute various NLP tasks. Word embedding is one method to implement word-vector representation.

## 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:

*“It’s pretty but not good”*

“*It’s good but not pretty”*

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 phrases10.

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 embeddings (sometimes referred to as neural word vectors). Similar to the weights of an ANN, neural word embeddings are words whose vector representation the neural network is trying to capture. It does so by looking at a word and its context from a dataset 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 lower11.

## 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.

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.

For example, consider the sample training data set (consists of *i* = 3 pairs of documents and binary sentiment labels):

<”I am happy”, “positive”>

<”The dog is sad”, “negative”>

<”The cat is happy”, “positive”>

Our vocabulary is of size *n* = 8 (8 distinct words), hence we will index them 0-7. The documents of the training data are represented as follows (with the additional column for the intercept, a word that might appear in the test data but not the training):

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 (I) | 1(am) | 2(happy) | 3(the) | 4(dog) | 5(is) | 6(sad) | 7(cat) |  |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

Since we are only using binary classification we have *k* =2 classes. The coefficient matrix is of size 2 x 9. This is referred to as the *sentiment classification matrix*:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 (I) | 1(am) | 2(happy) | 3(the) | 4(dog) | 5(is) | 6(sad) | 7(cat) |  |
| θ1 |  |  |  |  |  |  |  |  |  |
| θ 2 |  |  |  |  |  |  |  |  |  |

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

http://blog.datumbox.com/wp-content/uploads/2013/11/softmax_tutorial/image002.png

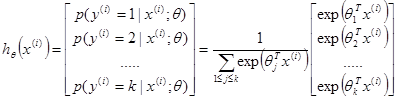
θ*i* -vector stores the coefficients of *i*th category for each of the *n* words

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

*y*(*i*) – label of the *i* th document  (*k* is the number of classes).

*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 using a SoftMax classifier. A detailed description of the algorithm is provided in the RNTN Explained section.

It is important to note that in the paper describing RNTN, the author states that training process took 3-5 hours.

## 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).12

Spark Core is the basis of the whole framework. 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 (RDD), 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 algorithms13.

In Spark terminology, there are 3 components in a distributed execution: the master node, or cluster manager, which allocates data to the cluster nodes and monitors them; the worker nodes, which are the machines that constitutes the cluster; and the executers – the process/threads on each worker (thus enabling concurrent computations both on separate machines and process within each machine).

## Downpour SGD and DeepDist

As stated, training an ANN is inherently sequential: We calculate the error per input in the training set, and adjust weights in a backwards fashion. Even optimized training methods such as Stochastic Gradient Descent (SGD) or minibatch still encompass an iterative nature making it impractical to apply to very large data sets. To apply SGD to large data sets, researchers from Google introduced Downpour SGD5, a variant of asynchronous stochastic gradient descent that uses multiple replicas of a single ANN model.

The basic approach is as follows: the training data is divided 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 multiple machines.

Here is a high level description of the algorithm:

1. Before processing each mini-batch, a model replica asks the parameter server service for an updated copy of its model parameters.
2. After receiving an updated copy of its parameters, the ANN model replica processes a mini-batch of data and computes a parameter gradient
3. ANN model replica sends the gradient to the parameter server, which then applies the gradient to the current value of the model parameters.

Downpour SGD proved valuable in another aspect: using SGD in an undistributed environment, a machine failure blocks the entire training process; however for asynchronous SGD, if one machine in a model replica fails, the other machines with their model replicas continue processing the training data and updating the model parameters via the parameter server.

However, one aspect should be noted: the asynchronous processing in Downpour SGD introduce a great deal of additional stochasticity in the optimization procedure. Clearly, a model replica is computing its gradients based on a set of parameters that are, probably, slightly out of date, while some other model replica have updated the parameters on the parameter server in the meantime.

Downpour SGD was implemented by the researchers in DistBelief – a framework which was not published and is not publicly available. However, Dirk Neumann put out an open source framework called DeepDist14. DeepDist implements Downpour SGD using Spark.

Written in Python, DeepDist follows the general flow for training ANN:

1. Define a *gradient* function. This would run on each worker node (executer) in the cluster and would take as input both the model parameters and training data.
2. Define a *descent* function. This function would run on the master server and would update the model with the gradient emitted by the workers.
3. Use Spark’s API to read the training data: thus, turning the data into a distributed processable RDD.
4. Initiate the DeepDist framework with the ANN: this starts the parameter server.
5. Run DeepDist *train* function with the data (RDD), and previously defined *gradient* and *decent* functions as input. The *train* function defines a *mapPartitions* function which, as a pre training step, makes an HTTP request from the parameter server for an updated copy of the model parameters, executes the *gradient* function on the data and sends the updated model to the centralize server.
6. Using Spark RDD, the data is partitioned and each executer perform the *mapPartitions* function on its data subset.
7. Finally, each model replica is sent to the centralized server which perform the *descent* function, that is: updating the model.

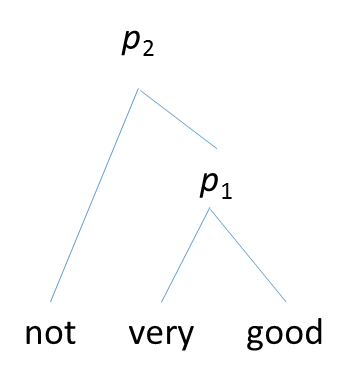
In our project, we used DeepDist as the basis for distributing RNTN.

# Distributed RNTN

## RNTN Explained

In this section we provide an explanation of RNTN and its computational steps.

As an example, consider the trigram “not very good” (the parse tree representation can be seen in Figure 1). Let us mark a middle node as *p*2 (it is a node in the parse tree that follows node *p*1).



Figure

Trigram representation

The RNTN can be described as the following, generalized flow (Figure 2):

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 () described in the softmax section and a is a node (or leaf) vectorized representation, in the example it is the word “not”.

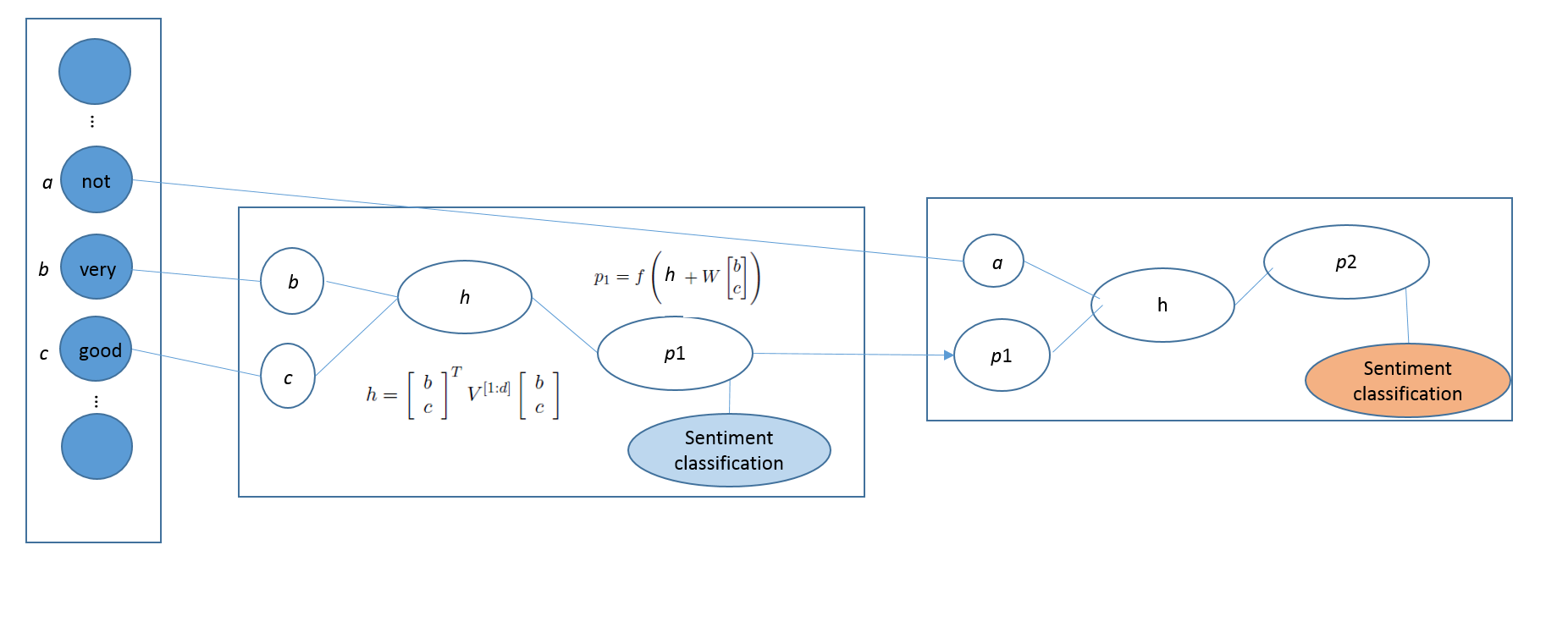
1. Compute the output of a tensor product via the following vectorized notation and the equivalent but more detailed notation for each slice:



(*b,c* are the vector representations of “very” and “good”, respectively)

1. Finally, each layer is computed:

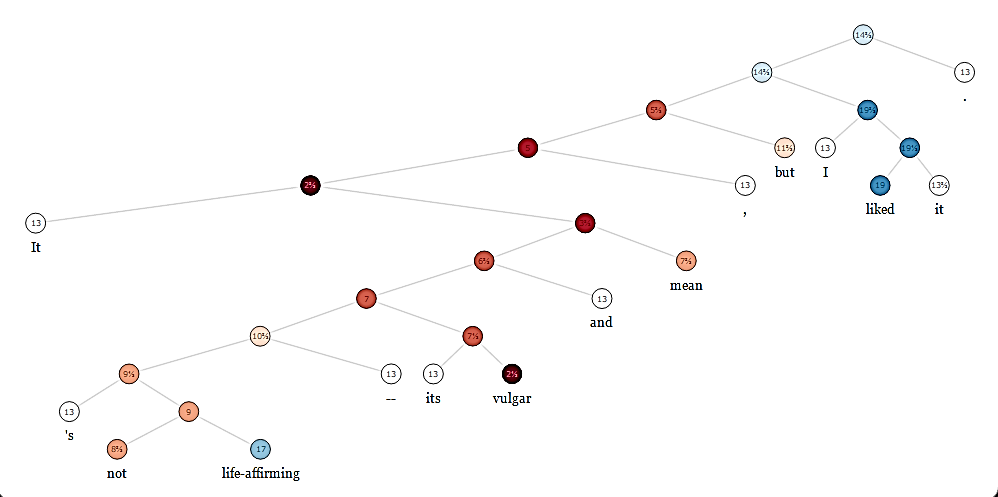
where: *f* is tanh activation function and *W* is the weight of the ANN of the layer.



Figure

Visualization of RNTN flow: on the left we have the input layer with the trigram example presented in Figure 1. The first node takes as input the two word (leafs) in the parse tree. It combines them, compute the product with the weight and simultaneously activates and classifies the output. The activation result is then forward propagated to the next layer.

The result is a parse tree where every node, that is not a leaf, is the composition of the two child nodes representation which is fed to the sentiment SoftMax classifier, activated using the tanh activation function and forward propagated. Since this process is done recursively the algorithm handles input (sentences) of varied length (Figure 3).



Figure

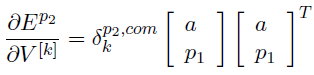
Sample visualization of RNTN for sentiment analysis output: the sentence initially is negative (red nodes) however, the speaker’s overalls’ positive view is detected (top blue node).

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



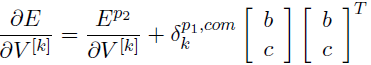
The backpropagation process trains the SoftMax components in a similar fashion described earlier. The remaining derivatives, of *V* and *W*, can only be computed in a top-down manner: from the top, root, node of the tree all the way to the leaf nodes. The full derivative for *V* and *W* is the sum of the derivatives at each of the nodes. Here we describe process of training *V* (*W* training is done similarly).

Additionally, We define the complete incoming error messages for a node *i* as . The partial derivative of *V*, with respect to *k*=1..*d*, at node *p*2 is (*a* is the “not” leaf):



Where  is the *k*th element of the vector.

While the paper describe the entire process, we can sum up that the full derivative for slice *V [k*] for this trigram tree is the sum at each node:



## RNTN single process implementation

The first step in this project was implementing the RNTN algorithm. While the original implementation was done using Matlab, other variants exist: a java implementation as part of Stanfornd’s CoreNLP framework15, and in a framework for distributed learning called deeplearning4j16. However, CoreNLP was not suited for distributed execution and tightly coupled with the framework’s APIs so it could not be used as a standalone implementation. deeplearning4j’s implementation was also considered but closer investigation revealed major concerns as to the correctness of the implementation: RNTN, as a deep learning algorithm, does not only updates the weights of the ANN but also updates the word-embedding matrix (the word vector representation). It was this feature that we failed to see in deeplearning4j’s application. Finally, it was decided to use a Python implementation for two main reasons: first, using Python’s robust mathematical *numpy* module (with its matrix and tensor support) would greatly facilitate the implementation. Second, the need to interface with DeepDist (written in Python) and Spark (which exposes a Python interface) in upcoming steps. We based our implementation on the initial work of R. Socher (the author of RNTN) in a project called *semantic-rntn*, available publicly on github.com.

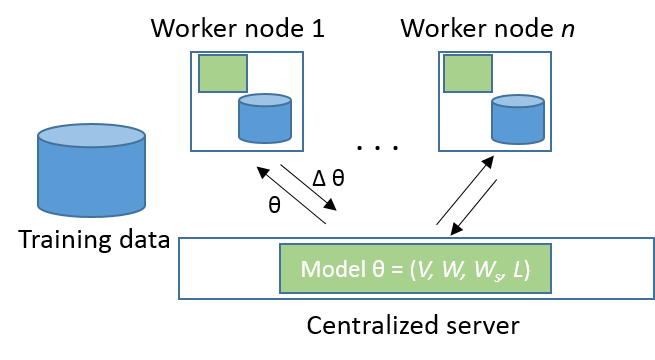
The main component of the implementation is the RNTN class (Appendix A). The class members are the model parameters (such as the word-embedding matrix *L*, the ANN activation weights *W*, *V* and bias *b*, SoftMax parameters *Ws* and *bs*) and differential variables (such as the delta values of the *W* weight matrix *dW*). The class methods are the ones needed to initiate the model and process the data (such as the forward and backward propagation, *forwardProp* and *backProp* respectively). The class has additional methods for operational needs (serializing and de-serializing of models to and from files, respectively).

Finally, it was this implementation that was used as a baseline for comparison and evaluate the performance of the distributed implementation.

## Distributed RNTN design

As a general approach, distributing RNTN needed to address two issues of the training procedure: shorter training time (while maintaining the high accuracy) and fault tolerance. Following Downpour SGD paradigm, we identified the following components (Figure 4):

1. The centralized model server. key functionality:
   1. Store the model
   2. Send a model replica to workers
   3. Asynchronously update the model with the gradient sent from the workers
2. Workers/executers. key functionality:
   1. Each will have a fraction of the data
   2. Get an updated/current copy of the model before computing the gradient
   3. Once computed, send the gradient to the centralized model server



Figure

Diagram of components in distributed RNTN: The Training data set is partitioned across the cluster nodes, each worker node gets a copy of the model parameters θ, calculates the gradient ∆θ, and sends it back to the server for model update.

Following the initial design we needed to fulfil three requirements:

* Data partioning: A mechanism for partioning the data across all worker nodes
* Cluster management: A mechanism to orchestrate/monitor execution on each worker
* Fault tolerance: A mechanism for fault tolerance in case a worker malfunctions.

We identified Spark as a solution for all three requests:

### Data partioning

As stated earlier, Spark’s Resilient Distributed Datasets (RDDs) is a logical collection of data partitioned across machines. There are two ways to create RDDs: parallelizing an existing collection in a driver program, or referencing a dataset in an external storage system (such as a shared files system, HDFS etc…). Parallelized collections are created by calling SparkContext’s *parallelize* method on an existing collection. The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. One important parameter for parallel collections is the number of partitions to cut the dataset into. Spark will run one task for each partition of the cluster. Unless configured otherwise, Spark tries to set the number of partitions automatically based on the cluster size. This provided us with the mechanism to distribute the data across all worker nodes and executers (multiple threads on each worker).

### Cluster management

For cluster management, Spark supports a standalone mode (the native Spark cluster Manager). The Manger, provides a web UI where it shows the cluster nodes along with its number of CPUs and memory (minus one gigabyte left for the OS). Once an application is launched through the Spark submit script, then the application is automatically distributed to all worker nodes and the Manger UI can be used to monitor the cluster. The master and each worker has its own web UI that shows cluster and job statistics. In addition, detailed log output for each job is also written to the work directory of each slave node. Sparks ability to monitor the application execution proved extremely valuable once we started running RNTN on the cluster as we were able to find and address implementation problems.

### Fault tolerance

Spark and its RDD concept is designed to handle failures of any worker nodes in the cluster. RDDs provide an interface based on a limited, coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items. Fault tolerance is achieved by logging the transformations used to build a dataset (its lineage) rather than the actual data. If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to re-compute just that partition (in contrast to other distributed frameworks that require costly data replication, failures can be recovered fairly quickly). Although an interface based on coarse-grained transformations may at first seem limited, RDDs are suitable for many parallel uses, since they apply the same operation to multiple data items. In our work, we used the map transformation to apply the gradient calculation to each data partition, hence rendering the distributed execution fault tolerant.

## Distributed RNTN implementation

Once the RNTN was implemented as a single process and the design was completed, we proceeded to distribute it with DeepDist on a multi-node cluster.

We noticed that DeepDist was missing two features.

The first was a missing exit procedure for its server. DeepDist uses Python *flask* module as a server implementation. Without the ability of shutting the server, not only was the framework incomplete, but it made working with it require us to find and manually shutdown the process that was running rogue which rendered the development tedious. The second enhancement we did was change the way DeepDist serialized the model replica. As described, each executer receives a copy of the model before proceeding with its forward and back propagation. Originally, the model sent was serialized using *pickle*, a Python serialization module. However, *pickle* failed to support lambda expressions (which RNTN uses). Hence, we enhanced DeepDist by replacing *pickle* with *cloudpickle*, a module that is specifically suitable for cluster computing where Python expressions are sent over the network to execute on remote hosts, possibly close to the data. Among other things, *cloudpickle* supports pickling for lambda expressions, functions and classes. We used DeepDist documented example of Word2Vec learning to ensure that our enhancements did not break DeepDist original capabilities (backward compatibility). Our changes are on review to be incorporated into DeepDist.

Finally, we wrote the driver program Distrntn (Appendix B):

Initially, Distrntn loads the different modules (Spark, RNTN classes), then reads the data (the labeled sentence parse trees) and initializes the RNTN class. Following DeepDist workflow, we defined the *gradient* and *descent* functions.

The *gradient* function (similarly to minibatch) forward propagates a training sample through the neural network and compute the cost. The aggregated cost of all samples in the phase are scaled (averaged) and back propagated. Key observation: while the classic backpropagation updates the weights of an ANN, our implementation only calculated the model deltas (the adjustments to the model). The computed deltas were returned as a result and sent to the centralized server to execute the *descent* function. The *descent* function took the deltas as input and updated the model.

Next, DeepDist is initialized with the model and the parameter server location and training begins: iterating training epochs, each epoch randomly shuffles the data (trees), tells Spark context object to parallelize the data (thus fulfilling the data partioning and fault tolerance requirements set by the design) and process the data in parallel (Figure 2).

Finally, in order to launch the execution, Distrntn application is submitted to the cluster using Spark’s submit script (and the third requirement, cluster management, is handled).

However, since the implementation referenced static resources (class member methods, the *forwardProp* and *backProp* functions) each executer node needed these resources made available. Hence, we turned the *semantic-rntn* project into a Python module and, using a custom installer, installed the module on each node in the cluster.

Figure

Diagram depicting Distrntn execution using DeepDist and Spark: Distrntn driver program initiates DeepDist and the spark context with training data, Spark cluster manager, or master node, distributes the RDDs to each worker node. Each node gets the updated model and executes the *gradient* function and transmits the result back to the DeepDist’ server for model updating.

With a working cluster, and a deployed application we proceeded with running the experiment (Appendix C).

## Data

Stanford University makes 3 training data sets available (with 1101 training samples, 2210 samples and a major one with 8544 samples). While the first, smaller data set was used for sanity tests along the project development, it was the larger data set (8544 samples, 2.1 MB) that the experiment used as training data while the medium sized data set (2210 samples) was used as test data to evaluate accuracy.

Each sample consists of a sentence parsed tree and a sentiment score. The data originated from rottentomatoes.com, a movie review website. The Stanford Parser17 was used to parse sentences into parse trees. Finally, Amazon Mechanical Turk was used to label each sentence with a sentiment score.

## Experiment and Results

In order to get a baseline for comparison, we started training our model on a single machine, using the *semantic-rntn* implementation. Tests were run on a CentOS 6.2 using intel Core i5-4300 processor. The training was configured to use 40 epochs, word vector dimensionality of 30 and minibatch size was 30. Parameters for comparison were average time per epoch, total training time and accuracy (calculated as number of correct sentiment classifications divided by the total number of samples). As stated above, the 8544 samples data set was used for training and the 2210 samples data set for testing the model.

The model training took almost 4 hours to train, averaging a training epoch time of around 260 seconds. The accuracy calculated was 0.79 or 79% which seemed consistent with the accuracy reported by the RNTN author (80.7%). This made us feel confident in the implementation and we proceeded with the distributed training on a cluster.

We used a virtual cluster of 25 machines (nodes), each with 4-8 cores (executers) which totaled 156 executers. Again, we’ve set the training configuration to use 40 epochs and word vector dimensionality of 30.

This time, total training took less than 20 minutes, with average time per epoch of 27 seconds. The model accuracy was 76.2% (Table 1).

|  |  |  |
| --- | --- | --- |
|  | single machine, with minibatch=30 | 156 executers cluster |
| Total training time (min) | 212 | 18.5 |
| Average time per epoch(sec) | 260.3 | 27.6 |
| accuracy (correct/total samples) | 0.7945 | 0.7626 |

Table

Results summary

## Conclusions

In this project we explored how RNTN, a deep learning neural network, can be distributed using Spark. We note three key differences between the single-process and distributed implementation:

1. A considerable faster training time: From 3 to 5 hours reported by the authors (and observed by our implementation), we were able to bring the training time to less than 20 minutes.
2. Minor effects on model accuracy: we see that the distributed model’s accuracy is impaired by 3%. As indicated by the authors of Downpour SGD, asynchronously updating the model introduced higher stochasticity to the model, thus damaging its accuracy. However, we too found the resulting model remarkably effective.
3. Fault tolerant training: by proving that Spark’s coarse-grained transformations are expressive enough for implementing a calculation of a deep learning model gradient, we showed that the whole process is resilient in the face of hardware failures.

RNTN has been proved valuable in other fields such as computer vision, where the ability to learn to identify objects in an image can be analogous to the sentiment detection described in this work: pixels (instead of words) are compositioned to objects (nodes in a parse tree) that constitute a whole image (a whole sentence). Since graphic data is much larger than text, we believe that our work can have further impact when applied to such data intensive applications.

# 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.

5Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in Neural Information Processing Systems. 2012.

6Shiffman, Daniel, Shannon Fry, and Zannah Marsh. The nature of code. D. Shiffman, 2012.

7Schalkoff, Robert J. Artificial neural networks. New York: McGraw-Hill, 1997.

8Sibi, 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.

9Hecht-Nielsen, Robert. "Theory of the backpropagation neural network." Neural Networks, 1989. IJCNN., International Joint Conference on. IEEE, 1989.

10de Kok, Daniël, and Harm Brouwer. "Natural language processing for the working programmer." (2011).

11Collobert, 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.

12Xin, 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.

13Matei 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.

14Neumann, Dirk. "DeepDist." DeepDist. N.p., n.d. Web. 05 Dec. 2015. <http://deepdist.com/>.

15Manning, Christopher D., et al. "The Stanford CoreNLP natural language processing toolkit." Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations. 2014.

16Gibson, Adam. "DL4J." Deeplearning4j. Skymind, n.d. Web. 05 Dec. 2015. <http://deeplearning4j.org/>.

17Klein, Dan. "Christopher d." Manning. Accurate unlexicalized parsing. In ACL 3 (2003): 423-430.

# Appendix

## Class RNTN

|  |
| --- |
| import numpy as np  import collections  np.seterr(over='raise',under='raise')  class RNN:  def \_\_init\_\_(self,wvecDim,outputDim,numWords,mbSize=30,rho=1e-6):  self.wvecDim = wvecDim  self.outputDim = outputDim  self.numWords = numWords  self.mbSize = mbSize  self.defaultVec = lambda : np.zeros((wvecDim,))  self.rho = rho  def initParams(self):  # Word vectors  self.L = 0.01\*np.random.randn(self.wvecDim,self.numWords)  # Hidden activation weights  self.V = 0.01\*np.random.randn(self.wvecDim,2\*self.wvecDim,2\*self.wvecDim)  self.W = 0.01\*np.random.randn(self.wvecDim,self.wvecDim\*2)  self.b = np.zeros((self.wvecDim))  # Softmax weights  self.Ws = 0.01\*np.random.randn(self.outputDim,self.wvecDim)  self.bs = np.zeros((self.outputDim))  self.stack = [self.L, self.V, self.W, self.b, self.Ws, self.bs]  # Gradients  self.dV = np.empty((self.wvecDim,2\*self.wvecDim,2\*self.wvecDim))  self.dW = np.empty(self.W.shape)  self.db = np.empty((self.wvecDim))  self.dWs = np.empty(self.Ws.shape)  self.dbs = np.empty((self.outputDim))  def costAndGrad(self,mbdata,test=False):  """  Each datum in the minibatch is a tree.  Forward prop each tree.  Backprop each tree.  Returns  cost  Gradient w.r.t. W, Ws, b, bs  Gradient w.r.t. L in sparse form.  """  cost = 0.0  correct = 0.0  total = 0.0  self.L,self.V,self.W,self.b,self.Ws,self.bs = self.stack  # Zero gradients  self.dV[:] = 0  self.dW[:] = 0  self.db[:] = 0  self.dWs[:] = 0  self.dbs[:] = 0  self.dL = collections.defaultdict(self.defaultVec)  # Forward prop each tree in minibatch  for tree in mbdata:  c,corr,tot = self.forwardProp(tree.root)  cost += c  correct += corr  total += tot  if test:  return (1./len(mbdata))\*cost,correct,total  # Back prop each tree in minibatch  for tree in mbdata:  self.backProp(tree.root)  # scale cost and grad by mb size  scale = (1./self.mbSize)  for v in self.dL.itervalues():  v \*=scale    # Add L2 Regularization  cost += (self.rho/2)\*np.sum(self.V\*\*2)  cost += (self.rho/2)\*np.sum(self.W\*\*2)  cost += (self.rho/2)\*np.sum(self.Ws\*\*2)  return scale\*cost,[self.dL,scale\*(self.dV+self.rho\*self.V),  scale\*(self.dW + self.rho\*self.W),scale\*self.db,  scale\*(self.dWs+self.rho\*self.Ws),scale\*self.dbs]  def forwardProp(self,node):  cost = correct = total = 0.0  if node.isLeaf:  node.hActs = self.L[:,node.word]  node.fprop = True  else:  if not node.left.fprop:  c,corr,tot = self.forwardProp(node.left)  cost += c  correct += corr  total += tot  if not node.right.fprop:  c,corr,tot = self.forwardProp(node.right)  cost += c  correct += corr  total += tot  # Affine  lr = np.hstack([node.left.hActs, node.right.hActs])  node.hActs = np.dot(self.W,lr) + self.b  node.hActs += np.tensordot(self.V,np.outer(lr,lr),axes=([1,2],[0,1]))  # Tanh  node.hActs = np.tanh(node.hActs)  # Softmax  node.probs = np.dot(self.Ws,node.hActs) + self.bs  node.probs -= np.max(node.probs)  node.probs = np.exp(node.probs)  node.probs = node.probs/np.sum(node.probs)  node.fprop = True  return cost - np.log(node.probs[node.label]), correct + (np.argmax(node.probs)==node.label),total + 1  def backProp(self,node,error=None):  # Clear nodes  node.fprop = False  # Softmax grad  deltas = node.probs  deltas[node.label] -= 1.0  self.dWs += np.outer(deltas,node.hActs)  self.dbs += deltas  deltas = np.dot(self.Ws.T,deltas)    if error is not None:  deltas += error  deltas \*= (1-node.hActs\*\*2)  # Leaf nodes update word vecs  if node.isLeaf:  self.dL[node.word] += deltas  return  # Hidden grad  if not node.isLeaf:  lr = np.hstack([node.left.hActs, node.right.hActs])  outer = np.outer(deltas,lr)  self.dV += (np.outer(lr,lr)[...,None]\*deltas).T  self.dW += outer  self.db += deltas  # Error signal to children  deltas = np.dot(self.W.T, deltas)  deltas += np.tensordot(self.V.transpose((0,2,1))+self.V,  outer.T,axes=([1,0],[0,1]))  self.backProp(node.left, deltas[:self.wvecDim])  self.backProp(node.right, deltas[self.wvecDim:])    def updateParams(self,scale,update,log=False):  """  Updates parameters as  p := p - scale \* update.  If log is true, prints root mean square of parameter  and update.  """  if log:  for P,dP in zip(self.stack[1:],update[1:]):  pRMS = np.sqrt(np.mean(P\*\*2))  dpRMS = np.sqrt(np.mean((scale\*dP)\*\*2))  print "weight rms=%f -- update rms=%f"%(pRMS,dpRMS)  self.stack[1:] = [P+scale\*dP for P,dP in zip(self.stack[1:],update[1:])]  # handle dictionary update sparsely  dL = update[0]  for j in dL.iterkeys():  self.L[:,j] += scale\*dL[j]  def toFile(self,fid):  import cPickle as pickle  pickle.dump(self.stack,fid)  def fromFile(self,fid):  import cPickle as pickle  self.stack = pickle.load(fid)  def check\_grad(self,data,epsilon=1e-6):  cost, grad = self.costAndGrad(data)  for W,dW in zip(self.stack[1:],grad[1:]):  W = W[...,None,None] # add dimension since bias is flat  dW = dW[...,None,None]  for i in xrange(W.shape[0]):  for j in xrange(W.shape[1]):  for k in xrange(W.shape[2]):  W[i,j,k] += epsilon  costP,\_ = self.costAndGrad(data)  W[i,j,k] -= epsilon  numGrad = (costP - cost)/epsilon  err = np.abs(dW[i,j,k] - numGrad)  print "Analytic %.9f, Numerical %.9f, Relative Error %.9f"%(dW[i,j,k],numGrad,err)  # check dL separately since dict  dL = grad[0]  L = self.stack[0]  for j in dL.iterkeys():  for i in xrange(L.shape[0]):  L[i,j] += epsilon  costP,\_ = self.costAndGrad(data)  L[i,j] -= epsilon  numGrad = (costP - cost)/epsilon  err = np.abs(dL[j][i] - numGrad)  print "Analytic %.9f, Numerical %.9f, Relative Error %.9f"%(dL[j][i],numGrad,err) |

## Distrntn.py

|  |
| --- |
| \_\_author\_\_ = 'Uri'  import os  import sys  import time  import optparse  import random  import numpy as np  import ConfigParser  import socket  import pickle  import collections  print ("Running on %s"%socket.gethostname())  #Load config parameters  config = ConfigParser.ConfigParser()  config.readfp(open(r'../config'))  SparkPythonPath = config.get('Spark', 'SparkPythonPath')  Py4jPath = config.get('Spark', 'Py4jPath')  appname = config.get('Distrntn ', 'appname')  masterurl = config.get('Distrntn ', 'masterurl')  mode = config.get('Distrntn ', 'mode')  sys.path.append(SparkPythonPath)  sys.path.append(Py4jPath)  print os.environ["PYTHONPATH"]  ######## import spark ########  try:  from pyspark import SparkContext  from pyspark import SparkConf  print ("Successfully imported Spark Modules")  except ImportError as e:  print ("Can not import Spark Modules", e)  sys.exit(1)  ######## import DeepDist ########  try:  from DeepDist import DeepDist  print ("Successfully imported DeepDist Modules")  except ImportError as e:  print ("Can not import DeepDist Modules", e)  sys.exit(1)  ######## import rntn ########  try:  from semantic\_rntn import rntn as nnet  from semantic\_rntn import tree as tr  from semantic\_rntn import sgd as optimizer  print ("Successfully imported rntn Modules")  except ImportError as e:  print ("Can not import rntn Modules", e)  sys.exit(1)  #make sure all modules imported  time.sleep(8)  '''  Setup args for sgd.  Note: Since DeepDist implements stochastic gradient descent the model type (optimizer) has to be sgd.  '''  usage = "usage : %prog [options]"  parser = optparse.OptionParser(usage=usage)  parser.add\_option("--test",action="store\_true",dest="test",default=False)  # Optimizer  parser.add\_option("--minibatch",dest="minibatch",type="int",default=60)  parser.add\_option("--optimizer",dest="optimizer",type="string",  default="sgd")  parser.add\_option("--epochs",dest="epochs",type="int",default=10)  parser.add\_option("--step",dest="step",type="float",default=1e-2)  parser.add\_option("--outputDim",dest="outputDim",type="int",default=5)  parser.add\_option("--wvecDim",dest="wvecDim",type="int",default=30)  parser.add\_option("--outFile",dest="outFile",type="string",  default="models/Distrntn .bin")  parser.add\_option("--inFile",dest="inFile",type="string",  default="models/Distrntn .bin")  parser.add\_option("--data",dest="data",type="string",default="train")  (opts,args)=parser.parse\_args(None)  print "Loading data..."  # load training data  trees = tr.loadTrees()  opts.numWords = len(tr.loadWordMap())  #setup the rntn  rnn = nnet.RNN(opts.wvecDim,opts.outputDim,opts.numWords,opts.minibatch)  rnn.initParams()  sgd = optimizer.SGD(rnn,alpha=opts.step,minibatch=opts.minibatch,  optimizer=opts.optimizer)  #setup spark  if mode == "local":  # Set heap space size for java  #os.environ["\_JAVA\_OPTIONS"] = "-Xmx1g"  conf = (SparkConf()  .setMaster("local[\*]")  .setAppName(appname)  .set("spark.executor.memory", "1g")  .set("spark.driver.memory", "1g")  .set("spark.Python.worker.memory", "1g"))  if mode == "cluster":  conf = (SparkConf()  .setAppName(appname))  sc = SparkContext(conf=conf)  '''  Define the gradient and descent functions as required by DeepDist.  For more info about gradient and descent functions, please see: http://www.DeepDist.com  '''  def gradient(model, tree\_data): # executes on workers  """  Each datum in the minibatch is a tree.  Forward prop each tree.  Backprop each tree.  Returns  cost  Gradient w.r.t. W, Ws, b, bs  Gradient w.r.t. L in sparse form.  """  tree\_data=list(tree\_data)  datasize=len(tree\_data)  if datasize == 0:  return []  cost = 0.0  correct = 0.0  total = 0.0  model.model.L,model.model.V,model.model.W,model.model.b,model.model.Ws,model.model.bs = model.model.stack  # Zero gradients  model.model.dV[:] = 0  model.model.dW[:] = 0  model.model.db[:] = 0  model.model.dWs[:] = 0  model.model.dbs[:] = 0  model.model.dL = collections.defaultdict(model.model.defaultVec)  # Forward prop each tree in minibatch  for tree in tree\_data:  c,corr,tot = model.model.forwardProp(tree.root)  cost += c  correct += corr  total += tot  # Back prop each tree in minibatch  for tree in tree\_data:  model.model.backProp(tree.root)  # scale cost and grad by mb size \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*88  #scale = (1./model.model.mbSize)  scale = (1./datasize)  for v in model.model.dL.itervalues():  v \*=scale  # Add L2 Regularization  cost += (model.model.rho/2)\*np.sum(model.model.V\*\*2)  cost += (model.model.rho/2)\*np.sum(model.model.W\*\*2)  cost += (model.model.rho/2)\*np.sum(model.model.Ws\*\*2)  model.message="updated in gardient"  return [scale\*cost,[model.model.dL,scale\*(model.model.dV+model.model.rho\*model.model.V),  scale\*(model.model.dW + model.model.rho\*model.model.W),scale\*model.model.db,  scale\*(model.model.dWs+model.model.rho\*model.model.Ws),scale\*model.model.dbs],model]  def descent(model, update): # executes on master  if len(update) != 0:  cost=update[0]  grad=update[1]  updatedModel=update[2]  model.model.stack=updatedModel.model.stack  scale = -model.alpha  # compute exponentially weighted cost  if np.isfinite(cost):  if (model.it > 1 and len(model.expcost) > 0):  model.expcost.append(.01\*cost + .99\*model.expcost[-1])  else:  model.expcost.append(cost)  if model.optimizer == 'sgd':  #update = grad  scale = -model.alpha  elif model.optimizer == 'adagrad':  # trace = trace+grad.^2  model.gradt[1:] = [gt+g\*\*2  for gt,g in zip(model.gradt[1:],grad[1:])]  # update = grad.\*trace.^(-1/2)  update = [g\*(1./np.sqrt(gt))  for gt,g in zip(model.gradt[1:],grad[1:])]  # handle dictionary separately  dL = grad[0]  dLt = model.gradt[0]  for j in dL.iterkeys():  dLt[:,j] = dLt[:,j] + dL[j]\*\*2  dL[j] = dL[j] \* (1./np.sqrt(dLt[:,j]))  update = [dL] + update  scale = -model.alpha  #update params  model.model.stack[1:] = [P+scale\*dP for P,dP in zip(model.model.stack[1:],grad[1:])]  #model.model.updateParams(scale,grad,log=False)  # handle dictionary update sparsely  dL = grad[0]  for j in dL.iterkeys():  model.model.L[:,j] += scale\*dL[j]  model.costt.append(cost)  if model.it%1 == 0:  print "Iter %d : Cost=%.4f, ExpCost=%.4f."%(model.it,cost,model.expcost[-1])  start = time.time()  with DeepDist(sgd,masterurl) as dd:  print 'wait for server to come up'  time.sleep(10)  #epoch loop  for e in range(opts.epochs):  startepoch = time.time()  print "Running epoch %d"%e  m = len(trees)  random.shuffle(trees)  for i in xrange(0,240,sgd.minibatch):  sgd.it += 1  mb\_data = sc.parallelize(trees[i:i+sgd.minibatch])  dd.train(mb\_data, gradient, descent)  endepoch= time.time()  print '\*\*\*\*\*\*\*\* time of iteration %f'%(endepoch-startepoch)  end = time.time()  print "Total time: %f"%(end-start)  #output the final model to file  with open(opts.outFile,'w') as fid:  pickle.dump(opts,fid)  pickle.dump(sgd.costt,fid)  rnn.toFile(fid)  sys.exit("program done") |

## Full project

The full project can be reviewed at <https://github.com/urirosenberg/rntn-spark>.

In addition to the code, it includes links to the DeepDist and semantic-rntn sub-projects along with installation instructions.