CS 224D: Deep Learning for NLP[1]

Lecture Notes: Part V [2]

Spring 2015

[1] Course Instructor: Richard Socher

[2] Author: Francois Chaubard, Richard Socher

Keyphrases: RNN, Recursive Neural Networks, MV-RNN, RNTN

This set of notes discusses and describes the many variants on the RNN (Recursive Neural Networks) and their application and successes in the field of NLP.

1 *Recursive Neural Networks*

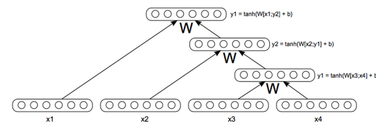

Figure 1: A standard Recursive Neural Network

In these notes, we introduce and discuss a new type of model that is indeed a superset of the previously discussed Recurrent Neural Network. Recursive Neural Networks (RNNs) are perfect for settings that have nested hierarchy and an intrinsic recursive structure. Well if we think about a sentence, doesn't this have such a structure? Take the sentence "A small crowd quietly enters the historical church". First, we break apart the sentence into its respective Noun Phrase, Verb Phrase, "A small crowd" and "quietly enters the historical church", respectively. But there is a noun phrase, verb phrase within that verb phrase right? "quietly enters" and "historical church". etc, etc. Seems pretty recursive to me.

The syntactic rules of language are highly recursive. So we take advantage of that recursive structure with a model that respects it! Another added benefit of modeling sentences with RNN's is that we can now input sentences of arbitrary length, which was a huge head scratcher for using Neural Nets in NLP, with very clever tricks to make the input vector of the sentence to be of equal size despite the length of the sentences not being equal. (see Bengio et al., 2003; Henderson, 2003; Collobert & Weston, 2008)

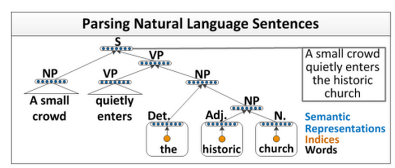

Figure 2: Parse Tree of a sentence.

Let's imagine our task is to take a sentence and represent it as a vector in the same semantic space as the words themselves. So that phrases like "I went to the mall yesterday", "We went shopping last week", and "They went to the store", would all be pretty close in distance to each other. Well we have seen ways to train unigram word vectors, should we do the same for bigrams, trigrams, etc. This very well may work but there are two major issues with this thinking. 1) There are literally an infinite amount of possible combinations of words. Storing and training an infinite amount of vectors would just be absurd. 2) Some combinations of words while they might be completely reasonable to hear in language, may never be represented in our training/dev corpus. So we would never learn them.

We need a way to take a sentence and its respective words vectors, and derive what the embedded vector should be. Now lets first ask a very debated question. Is it naive to believe that the vector space that we used to represent all words, is sufficiently expressive to also be able to represent all sentences of any length? While this may be unintuitive, the performance of these models suggest that this is actually a reasonable thing to do.

Let's first discuss the difference between semantic and grammatical understanding of a sentence. Semantic analysis is an understanding of the meaning of a sentence, being able to represent the phrase as a vector in a structured semantic space, where similar sentences are very nearby, and unrelated sentences are very far away. The grammatical understanding is one where we have identified the underlying grammatical structure of the sentence, which part of the sentence depends on which other part, what words are modifying what other words, etc. The output of such an understanding is usually represented as a parse tree as displayed in Figure 2.

Now for the million dollar question. If we want to know the semantic representation, is it an advantage, nay, required, to have a grammatical understanding? Well some might disagree but for now we will treat this semantic composition task the following way. First, we need to understand words. Then, we need to know the way words are put together, Then, finally, we can get to a meaning of a phrase or sentence by leveraging these two previous concepts.

So lets begin with our first model built on this principle. Let's imagine we were given a sentence, and we knew the parse tree for that sentence, such as the one displayed in Figure 2, could we figure out an encoding for the sentence and also perhaps a sentiment score just from the word vectors that are in the sentence? We observe how a Simple RNN can perform this task. (As will you in PSet 3!)

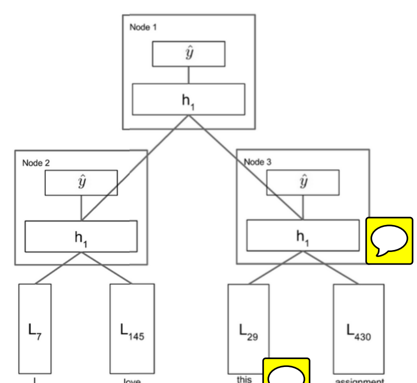

Figure 3: An example standard RNN applied to a parsed sentence "I love this assignment"

1.1 *A simple single layer RNN*

Lets walk through the model displayed in Figure 3 above. We first take a sentence parse tree and the sentence word vectors and begin to walk up the tree. The lowest node in the graph is Node 3, so we concatenate L_{29} and L_{430} to form a vector $\in \mathbb{R}^{2d}$ and feed it into our network to compute:

$$h^{(1)} = tanh(W^{(1)} \begin{bmatrix} L_{29} \\ L_{430} \end{bmatrix} + b^{(1)}) \tag{1}$$

Since $W^{(1)} \in \mathbb{R}^{d\times 2d}$ and $b^{(1)} \in \mathbb{R}^d$, $h^{(1)} \in \mathbb{R}^d$. We can now think of $h^{(1)}$ as a point in the same word vector space for the bigram "this assignment", in which we did not need to learn this representation separately, but rather derived it from its constituting word vectors.

We now take $h^{(1)}$ and put it through a softmax layer to get a score over a set of sentiment classes, a discrete set of known classes that represent some meaning. In the case of positive/negative sentiment analysis, we would have 5 classes, class 0 implies strongly negative, class 1 implies negative, class 2 is neutral, class 3 is positive, and finally class 4 is strongly positive.

Now we do the same thing with the "I" and "love" to produce the vector $h^{(1)}$ for the phrase "I love". Again, we compute a score over the semantic classes again for that phrase. Finally, for the most interesting step, we need to merge the two phrases "I love" and "this assignment". Here we are concatenating word phrases, rather than word vectors! We do this in the same manner, concatenating the two $h^{(1)}$ vectors and compute

$$h^{(1)} = \tanh(W^{(1)} \begin{bmatrix} h^{(1)}_{Left} \\ h^{(1)}_{Right} \end{bmatrix} + b^{(1)}) \tag{2}$$

Now we have a vector in the word vector space that represents the full sentence "I love this assignment". Furthermore, we can put this $h^{(1)}$ through the same softmax layer as before, and compute sentiment probabilities for the full sentence. Of course the model will only do this reliably once trained, but that left for you to do in PSet3. :)

Now lets take a step back. First, is it naive to think we can use the same matrix W to concatenate all words together and get a very expressive $h^{(1)}$ and yet again use that same matrix W to concatenate all phrase vectors to get even deeper phrases? These criticisms are valid and we can address them in the following twist on the simple RNN.

1.2 *Syntactically Untied SU-RNN*

As we discussed in the criticisms of the previous section, using the same W to bring together a Noun Phrase and Verb Phrase and to bring together a Prepositional Phrase and another word vector seems intuitively wrong. And maybe we are bluntly merging all of these functionalities into too weak of a model.

What we can do to remedy this shortcoming is to "syntactically untie" the weights of these different tasks. By this we mean, there is no reason to expect the optimal W for one category of inputs to be at all related to the optimal W for another category of inputs. So we let these W's be different and relax this constraint. While this for sure increases our weight matrices to learn, the performance boost we gain is non-trivial.

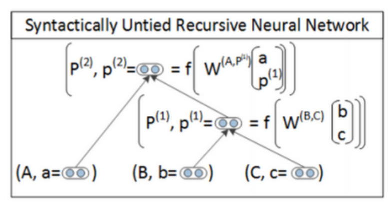

Figure 4: Using different W's for different categories of inputs is more natural than having just one W for all categories

As in FIgure 4 above, we notice our model is now conditioned upon what the syntactic categories of the inputs are. Note, we determine what the categories are via a very simple Probabilistic Context Free Grammar (PCFG) which is more or less learned by computing summary statistics over the Penn Tree Bank to learn rules such as ”The” is always a DT, etc, etc. No deeper understanding of this part is really necessary, just know itś really simple.

The only major other difference in this model is that we initialize the W's to the identity. This way the default thing to do is to average the two word vectors coming in. Slowly but surely, the model learns which vector is more important and also any rotation or scaling of the vectors that improve performance. We observe in Figure 5 that the trained weight matrices learn actually meaning! For example, the DT-NP rule or Determiner followed by a Noun Phrase such as ”The cat” or ”A man”, puts more emphasis on the Noun Phrase than on the Determiner. (this is obvious because the right diagonals are red meaning higher weightl). This is called the notion of soft head words, which is something that Linguists have long observed to be true for sometime, however the model learned this on its own just by looking at data. Pretty cool!

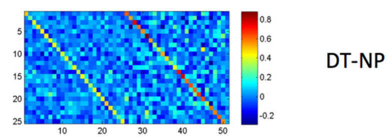

Figure 5: The learnt W weights for DT-NP composition match Linguists theory

The SU-RNN does indeed outperform previously discussed models but perhaps it is still not expressive enough. If we think of modifying words, such as adverbs like ”very”, any interpolation with this word vector and the following one, is definitely not what the understood nature of ”very” is.

As an adverb, it's literal definition is ”used for emphasis”. How can we have a vector that emphasizes any other vector that is to follow when we are solely performing a linear interpolation? How can we construct a vector that will ”scale” any other vector this way? Truth is we can not. We need to have some form of multiplication of word on another word. We uncover two such compositions below that enable this. The first utilizes word matrices and the other utilizes a Quadratic equation over the typical Affine.

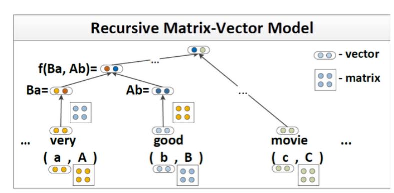

Figure 6: An example MV-RNN

1.3 *MV-RNN's (Matrix-Vector Recursive Neural Networks)*

We now augment our word representation, to not only include a word vector, but also a word matrix! So the word ”very” will have a word vector $v_{very} \in \mathbb{R}^d$ but also $V_{very} \in \mathbb{R}^{d \times d}$. This gives us the expressive ability to not only embed what a word means, but we also learn the way that words ”modify” other words. The word matrix enables the latter. To feed two words, a and b, into a RNN, we take their word matrices A and B, to form our input vector x as the concatenation of vector Ab and Ba. For our example of ”very”, V_{very} could just be the identity times any scalar above one. Which would scale any neighboring word vector by that number! This is the type of expressive ability we desired. While the new word representation explodes our feature space, we can express much better the way words modify each other.

By observing the errors the model makes, we see even the MV-RNN still can not express certain relations. We observe three major classes of mistakes.

First, Negated Positives. When we say something positive but one word turns it negative, the model can not weigh that one word strong enough to flip the sentiment of the entire sentence. Figure 7 shows such an example where the swap of the word "most" to "least" should flip the entire sentiment of the sentence, but the MV-RNN does not capture this successfully.

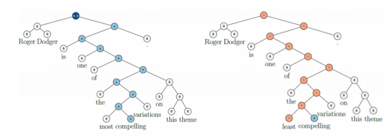

Figure 7: Negated Positives

The second class of mistakes is the Negated Negative case. Where we say something is not bad, or not dull, as in Figure 8. The MV-RNN can not recognize that the word "not" lessens the sentiment from negative to neutral.

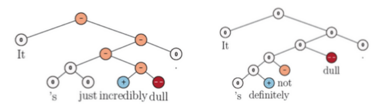

Figure 8: Negated Negatives

The final class of errors we observe is the "X but Y conjunction" displayed in Figure 9. Here the X might be negative BUT if the Y is positive then the model's sentiment output for the sentence should be positive! MV-RNNs struggle with this.

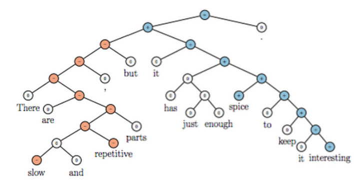

Figure 9: Using a Recursive Neural Net can correctly classify the sentiment of the contrastive conjunction X but Y but the MV-RNN can not

Thus, we must look for an even more expressive composition algorithm that will be able to fully capture these types of high level compositions.

1.4 RNTNs (Recursive Neural Tensor Network)

The final RNN we will cover here is by far the most successful on the three types of errors we left off with. The Recursive Neural Tensor Network does away with the notion of a word matrix, and furthermore, does away with the traditional affine transformation pre-tanh/sigmoid concept. To compose two word vectors or phrase vectors, we again concatenate them to form a vector $\in \mathbb{R}^{2d}$ but instead of putting it through an affine function then a nonlinear, we put it through a quadratic first, then a nonlinear, such as:

$$h^{(1)} = tanh(x^T V x + W x) \tag{3}$$

Note that V is a 3rd order tensor in $\in \mathbb{R}^{2d \times 2d \times d}$. We compute $x^T V[i] x \; \forall i \in [1, 2, ...d]$ slices of the tensor outputting a vector $\in \mathbb{R}^d$. We then add Wx and put it through a nonlinear function. The quadratic shows that we can indeed allow for the multiplicative type of interaction between the word vectors without needing to maintain and learn word matrices!

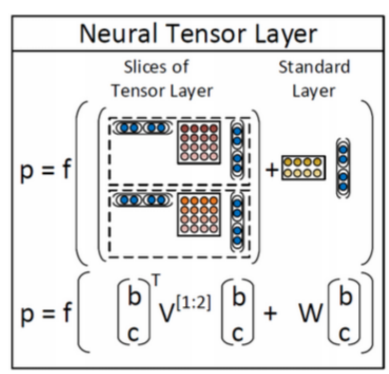

Figure 10: One slice of a RNTN. Note there would be d of these slices.

Model	Accuracy	
	Negated Positive	Negated Negative
biNB	19.0	27.3
RNN	33.3	45.5
MV-RNN	52.4	54.6
RNTN	**71.4**	**81.8**

Model	Accuracy	
	Negated Positive	Negated Negative

As we see in Figure 11, the RNTN is the only model that is capable of succeeding on these very hard datasets.

We will continue next time with a model that actually outperforms the RNTN in some aspects and it does not require an input parse tree! This model is the Dynamic Convolutional Neural Network, and we will talk about that soon. Good luck on your midterm!