Abstract

Introduction

Problem

As already defined, the objective of this project is to develop a dependency parser based on Artificial Neural Networks. For this purpose we are faced with the following challenges:

1. Find an Artificial Neural Network architecture that can help us to model the dependency parser in order to uncover the dependency structure for the unseen sentences.
2. Find a suitable feature representation that goes beyond word counting, that can incorporate the context of the constituent words of a sentence.

In this project we're implementing a transition based parser. A transition based parser must find the existing transitions that model the dependency relations within words in a given sentence. For this task we have received three data sets:

1. Training
2. Test
3. Development

Each data set consist of sentences and on its turn in each sentence contaings, beside the constituent words, the following additional elements like:

- the head of the word in the sentence
- The POS tag of the word
- The dependency relation that describes the translation

However, the dataset doesn't contain the transition type, so we need to find it first in order to be able to train our network.

In a machine learning setup the training data set is used to train the model, in other words, to find the parameters of the model that can help us to uncover, in our case, the existing relations within the words in the sentence. The training dataset should contain the network input set of features and also the corresponding tag (target) to the input features.

The output of our model must be able to predict the following:

1. The transition within two words
2. The head of the a pair of words
3. The dependency relation

Besides the prediction phase, which is in fact the aimed parser, we still need to be able to feed the ANN with a set features that are able not only to represent a single word using a numerical value, but we need to incorporate in the feature the context of a word, not only within the sentence but also within the corpus. To achieve such a goal we need to envision a word representation that can incorporate the context backward and forward ideally in an infinite basis or at least with a window that can be parameterized in order to cover as much context as possible. For this purpose we can't use a simple feed forward ANN, we need to use a ANN that can "scan" a sentence forward and backwards keeping track of its history, the past, but also its future, shortly, the context of the input sequence.

Our ANN model thus should capable of:

1. Find a representation a word in a vectorized manner: a word to vector approach
2. Find a representation of a word that includes context.
3. Find an ANN that can predict the three labels stated above.

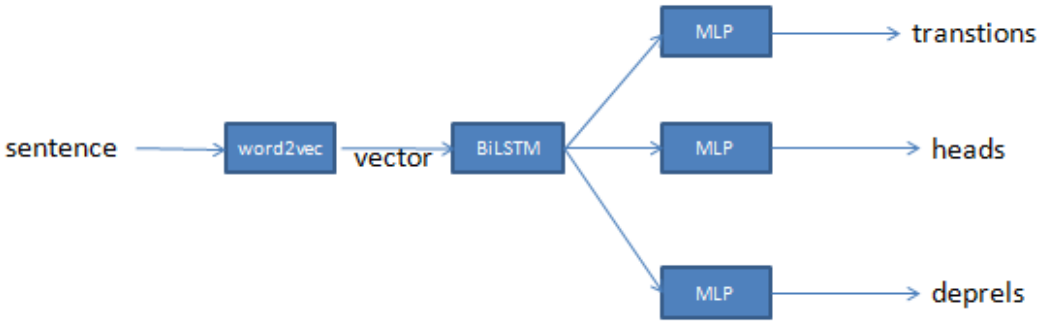
4. Define a the input features and the output labels for the training phase
5. Define the input features for the test and develop phase
6. Offer an easy an intuitive manner to predict, thus, parse, an unseen sentence.

Approach

In order to tackle our problem we have made the following decision for the architecture of our model:

1. Word vectorization we're using word2vec, which transform a word to a vector *[insert reference to paper]*. For this we're using the gensim implementation of word2vec.
2. A multilayer perceptron (MLP) for the prediction of the transitions, heads and dependency relations
3. A bidirectional Long Short Term Memory Network ANN (BiLSTM) which is special type recurrent neural network (RNN).

In our implementation we're following the approach of Kiperwasser and Goldberg [insert reference]. The image bellow depicts globally the architecture of the parser.



Let's zoom in on each of the architectural components of our parser.

Word vectorization: Word2Vec

Word2Vec is an unsupervised (feature labels are not know) algorithm which learns word vectors and word context vectors. We use word2vec to find a vector representation of words which we use as input of our BiLSTM RNN. But why do we need to encode our sentences? Well, because ANN cannot work with strings directly, so we need to encode them in a numerical way, in this case, not only numerical but also vectorial. Word2Vec learns the word embeddings and by learning word embeddings we prevent the high sparcity problem generated by the representations of words as unique id's.

We're using the gensim word2vec implementation which is quite straightforward and easy to use. It generates vectors of size 189, which are customizable, for instance we can define the size of our context window, in other words, we can define how far in the past and in the future word2vec should look for the context of a word.

As a final note, word2vec comes in two flavors: CBOW(continuous bag of words model) and the Skip-Gram model. CBOW predicts target words, while skip-gram does the inverse and predicts source context-words from the target words. [https://www.tensorflow.org/tutorials/word2vec/]

BiLSTM

BiLSTM are a special type of RNN that can parse a sequence input in two directions. RNN's fall in the category of unsupervised learning, which means that the model doesn't need feature labels in order to learn.

A BiLSTM consists of two RNN's: RNN-f and RNN-R. One networks reads the sequence in the regular order and other on the inverse order. By doing this the BiLSTM is able to capture the entire history of the sentence but also the entire future capturing like this the whole context of word

[honestly, by reading what word2vec and bilstm do, I can't see why whe should use word2vec and biLSTM together, it should be one or the other, and due to time constraints we should stick to word2vec as the encoding-vectorization of the words. From here on we should only focus on the configuration of the MLP's.

The BiLSTM is used as follows. For each word/pos tag vector we calculate its embeddings using BiLSTM. The input feature of the MLP become a concatenation of the BiLSTM output for the word vector and pos vector:

V_i = e(w_i) + e(tag_i), where v, w and tag are vectors .

TODO: add network setup

[insert reference]

Multilayer perceptron (MLP)

Up to this point we've been busy in our parser architecture defining how the ANN features representation. Also, we have used so far two unsupervised models. From this point on we move on to supervised models.

MLP are also known as feed forward ANN which means that the activation of the nodes occurs in a forward wise. For a MLP we need features with each respective labels in order to learn the parameters of the model. We are using a MPL with one hidden layer and a set of vectorized word-pos tags features.

For the dependency parse we want to learn the transitions, the words heads and the dependency relation.

In a transition parser we have the following transitions:

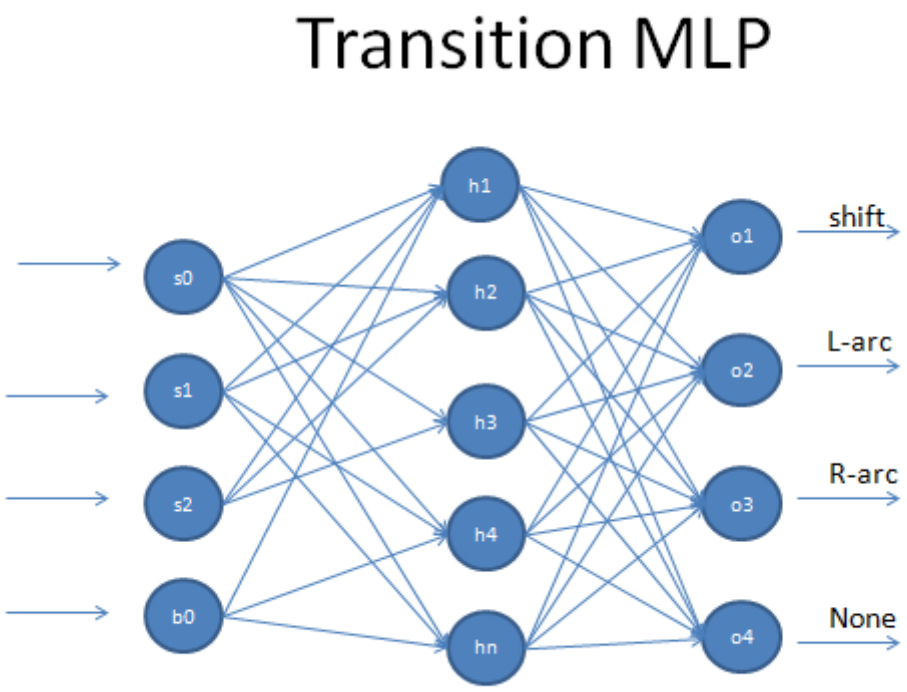
1. Shift
2. Left arc
3. Right arc

While parsing a sentence, if two words are not dependent on each other (one is the head of the other) we perform a shift operation in order to push a new word from the buffer into the stack. In order to learn the transitions we define the following input features for the MLP:

X: [s0, S1, S2, B0]

Y:[transition {*shift, left arc, right arc*}

Where, S0, S1 and S2 are the top three elements of the stack, while B0 is the first element of the buffer. The MLP topoloty is shown bellow:



There are two ways to generate the features:

1. We can generate all the possible combinations s0, s1, s2 and b0, and will mean that some of those combination are not seen in the sentence, so we must have provisions for the inexistent combinations, for instance the combination [car, airplane, the, about] is never seen in the stack, so the transition for such a combination is “none.
2. We generate in our training data set only the possible transitions, so we have exactly three output class, therefore the None class is not needed and our ANN topology only has three output neurons.

For the test phase again we envision two possibilities:

1. Online testing: We parse the sentence per word basis and every parsing action determines the next one. For instance, our initialization of the stack buffer looks like this:

Transition	**Stack**	**Buffer**	**Action**
	[ROOT]	koala eats leafs and barks	
S	[ROOT] A	A koala eats leafs and barks	
S	[ROOT] A koala	eats leafs and barks	
Left-arc(det)	[ROOT] koala	eats leafs and barks	det(koala, a)
S	[ROOT] koala eats	leafs and barks	
left-arc(nsubj)	[ROOT] eats	leafs and barks	nsubj(eats,koala)
S	[ROOT] eats leafs	and barks	
righ-arc(dobj)	[ROOT] eats	and barks	dobj(eats, leafs)
S	[ROOT] eats and	barks	
S	[ROOT] eats and barks		

So we call our model with the following features: [s0=' ',s1= ' ',s2= 'ROOT',b0= 'A], and it predicts a shift, and our stack looks like third line of the table above, and again we call the model with the features [s0= ' ',s1= 'ROOT',s2='A'= 'koala'],and again a shift is predicted. And so on until the whole sentence has been parsed. [Is this feasible?]

2. Offline training: we generate a Cartesian product of all the possible word combinations in order to predict all the transitions. [I find this too hard although if should be faster]

[Does this make sense or am I talking bullshit? At least that's the way I see it]

Similarly we would have a MLP with the same topology to learn the heads and the dependency relations. [we might need to use regression instead of classification to predict the heads numbers because in a real situation we would not know the maximum number of words in a sentence].

Features configuration

The datasets don't contain the transitions, so we need to handcraft them using the transition parsing algorithm. For the test and dev phase see above how to do it.

Experiments

We can think about different experiments with different possibilities:

1. Increase or decrease the number of hidden layers.
2. Use different sizes of word vectors
3. Increase or decrease the window size of the word2vec.
4. Use the MLP with as input
    a. only the word2vec features
    b. Word2vec + biLSTM features
    c. Only biLSTM features.
    d. online vs offline training/testing

…And measure the performance

Discussion and conclusion