Accurate Dependency Parsing with a Stacked Multilayer Perceptron

Giuseppe Attardi[1], Felice Dell'Orletta[1], Maria Simi[1], and Joseph Turian[2]

[1] Dipartimento di Informatica, Università di Pisa
[2] Dept. IRO, Université de Montréal
{attardi, dellorle, simi}@di.unipi.it, turian@iro.montreal.ca

Abstract. DeSR is a statistical transition-based dependency parser which learns from annotated corpora which actions to perform for building parse trees while scanning a sentence. We describe recent improvements to the parser, in particular stacked parsing, exploiting a beam search strategy and using a Multilayer Perceptron classifier. For the Evalita 2009 Dependency Parsing task DesR was configured to use a combination of stacked parsers. The stacked combination achieved the best accuracy scores in both the main and pilot subtasks. The contribution to the result of various choices is analyzed, in particular for taking advantage of the peculiar features of the TUT Treebank.

Keywords: parser, dependency parsing, perceptron, classifier, natural language.

1 Introduction

Dependency-based syntactic parsing is the task of uncovering the dependency tree of a sentence, which consists of labeled links representing dependency relationships between words. The task can be cast into a series of classification problems, picking a pair of tokens and deciding whether to create a dependency link between them.

A statistical *transition-based parser* uses training data to learn which actions to perform for building a dependency graph while scanning a sentence. The state of the art in the field is well represented by the results in the CoNLL Shared tasks.

The DeSR parser was among the most accurate transition-based parsers at the CoNLL 2008 Shared Task [9] and presents distinctive features like a deterministic incremental algorithm with linear complexity and the ability to handle non-projective dependencies, which often arise in languages like Italian.

The best accuracy though was obtained using an SVM classifier, or rather multiple binary SVM classifiers, which are quite expensive to train and require a lot of memory. DeSR underwent an extensive architecture redesign to incorporate the following features:

1. to exploit a beam search strategy to achieve better accuracy
2. integration in the Tanl linguistic pipeline
3. a Multilayer Perceptron classifier to improve speed and achieve a smaller memory footprint.

We will present the new architecture and then discuss the use of DeSR in the Evalita 2009 parsing task.

2 The DeSR Multilingual Dependency Parser

DESR (Dependency Shift Reduce) is a transition-based statistical parser which builds dependency trees while scanning a sentence and invoking at each step a classifier to select a proper *Shift*/*Reduce* parsing action for the current state based on a set of representative features of the state [1]. The parsing rules described in [2] allow parsing to be performed deterministically in a single pass.

The parser consists in two parts: (i) a training module which learns from a training corpus a classifier model, and (ii) the actual parser which builds the dependency tree for a sentence by performing the actions suggested in each state by the model.

The state of the parser is represented by a triple $\langle S, I, A \rangle$, where I is a sequence of tokens still remaining in the input. Initially I contains the sequence of tokens for the sentence being parsed: each token contains the word w_i as well as a set of word features p_i, consisting typically of the POS tag, the word lemma and possibly morphological features. S is the stack containing analyzed tokens; A is a set of labeled dependencies constructed so far. At each step in the algorithm, the parser selects a parsing rule to apply and modifies its state accordingly.

3 Beam Search

An eager version of the algorithm needs just to maintain a single state, which gets updated during the parsing process. The beam search algorithm instead requires exploring several alternatives states. In order to maintain a minimal memory footprint, we developed a design which limits the amount of data required to represent states, by sharing most data and performing copy on write.

Each parser state maintains its own version of the stack and of the input queue, but tokens stored in them are shared. The only operations that modify tokens are reductions. To limit the amount of copying, tokens consist in two parts: a fixed part containing the static attributes and a list of children to represent the varying tree structure. A *reduce* operation involves copying the tree structure of the affected token, while sharing the rest. States created when alternative branches in the search are explored also share as much as possible. States are discarded when a dead end is reached or a state has a likelihood score below the lowest one in the beam. A reference count is maintained in each state so that all its preceding states can also be deleted, up to a join with another live branch. When a state is deleted, the tokens it is using, which are not shared with other states, also get deleted. No reference counting is required for this, since shared tokens can be detected by the fact that two consecutive states have the same token in the same sentence position.

Beam search maintains two vectors: the current states and the next states. At each iteration, the best scoring state is extracted from the current states, its successors are created according to all possible actions and added to the next states, if they find space in it depending on the value of the state likelihood. This is a score which is computed as the sum of the log likelihoods of all previous states in the sequence that lead to that state. The likelihood of a state is obtained from the probability distribution computed by classifiers like Maximum Entropy or Multilayer Perceptron.

When an iteration completes, next states and best states are swapped and the process repeats.

There is no need to maintain multiple states during training, since training is fully deterministic, given a specific reduction strategy.

DeSR strategy is different from the arc-eager strategy by Nivre [7], but it similarly chooses to delay reductions as much as possible. Because of such determinism, a single state is maintained during training. The code for the training and parsing procedures can however mostly be shared by exploiting polymorphism: the classes `TrainState` and `ParseState` both inherit from State, and they differ only in the implementation of method `State::copy(Token*)`: only the `ParseState` version does indeed perform a copy of the token.

The parser adopts a factory pattern for extensibility. Adding a variant using a different learning algorithm requires providing a class which implements an abstract interface, consisting of just two methods (`train` and `parse`) and registering the parser factory by means of a call to `REGISTER_PARSER`. The new learning algorithm becomes available just by linking it with the rest of the code.

The parser code is mostly language independent, but a few abstract classes are present that can be specialized for performing language specific processing, for instance reading from different corpus format or extracting morphological features from tokens.

4 The Tanl Pipeline

Tanl (Text Analysis and Natural Language) is a full linguistic pipeline that has been built around DeSR to perform complete document analysis from pure text to semantic annotation.

The pipeline is designed around an extensible data representation for tokens, which are the basic unit of document representation. Tokens are passed along the stages each stage in the pipeline adds annotations in terms of new attributes for the

Instead of providing special tools and code generators to create a processing structure, Tanl relies on the use of a popular scripting language, so that any arbitrary combination can be assembled with a few lines of scripting code and tested immediately. For instance here is an example of how a pipeline can be created and started in Python:

```
ss = SentSplitter('italian.spl').pipe(stdin)
wt = Tokenizer().pipe(ss)
pt = PosTagger('italian.pos').pipe(wt)
pa = Parser('italian.par').pipe(pt)
foreach sentence in pa:
   print sentence
```

The first four lines set up a pipeline consisting of a `SentenceSplitter`, a `Tokenizer`, a `PosTagger` and a `Parser`. No processing has happened so far. The last instruction is a loop that drives the processing, requesting the next sentence from the `Parser`, which in turn requests the next vector of tokens from the `PosTagger` and so on.

5 Multilayer Perceptron

A new classifier has been developed for DeSR, based on a Multilayer Perceptron. The classifier used in the Evalita experiments involves a hidden layer with 320 hidden units. The Multilayer Perceptron classifier was developed through symbolic programming techniques and then converted to C++. The initial version was developed using Theano [10], a Python library and optimizing compiler for manipulating and evaluating expressions, especially matrix-valued ones. Theano uses operator overloading, so that expressions written in Python are turned into symbolic expressions. Theano can perform symbolic transformations on these expressions, like computing derivatives or gradients. The Theano compiler generates C code for evaluating symbolic expressions, which can be run to compute numeric results.

In our case the gradient of the loss function was computed in order to perform the steps of the back-propagation learning procedure of neural networks. After successful testing of the Theano version, the expressions for the gradient were obtained from Theano itself and rewritten in C++ using the Boost uBLAS library [11].

The MLP model was as follows:

$$Pr(Y \mid x) = \psi(b_2 + W_2\, h(x))$$
$$h(x) = \sigma\,(b_1 + W_1\, x)$$

$Pr(Y \mid x)$ is a probability distributions over possible decisions Y, $h(x)$ is the hidden-layer, a real-valued vector with 320 dimensions, σ is the softsign function [5]:

$$\sigma(z) = z/(1 + |z|)$$

ψ is the softmax squashing function:

$$\psi_i(z) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

W_1, W_2, b_1, and b_2 are all parameters tuned during training to minimize the cross-entropy between the true probability distribution over parser decisions and the predicted distribution. (The true probability is 1 for the correct parser decision and 0 for all others.) We minimized this cross-entropy using stochastic gradient descent. We used early stopping, choosing the epoch that achieved the lowest per-decision error on held-out validation data. We also chose the learning rate to minimize this validation error.

The accuracy is further increased by using a beam search algorithm that exploits the probability distribution computed by the MLP classifier. A beam size of 10 was sufficient, since increasing the beam did not lead to accuracy improvements.

Stacked Parser

In DeSR, the set of features to use in parsing is configurable through a parameter file and consists in a choice of tags from various tokens, in particular form, lemma, POS, morphology and dependency label. The set of features used in the evaluation was the same used in Evalita 2007, chosen through a process of feature selection [3].

While DeSR achieves competitive accuracy with the state of the art, it shares the typical defects of transition-based parsers, which often make mistakes on long

distance dependencies. In order to mitigate this problem, we developed a technique that allows the parser to learn from its own mistakes. The approach, described in [4], consists in running a first parser, then extracting hints from its output and feed them to another parser running in the opposite direction. The second parser sees tokens that the first parser would only see later, but knowing the output of the first parser can decide whether the dependency it created was wrong. The second parser, which tries to recognize the errors by the first parser, is trained on a special version of the corpus, which contains not only the standard dependency annotations, but also the dependencies that the first parser produced analyzing the corpus itself.

This approach has been called *stacked parsing* [8], since one parser is stacked on top of another. The difference in our solution is that we use the same type of parser, with linear complexity, in both stages, rather than a combination of a transition-based parser and a graph-based parser.

We call the stacked parser a *Reverse Revision* parser, since it operates in the reverse direction and tries to revise the output of the first parser. In [4] we showed that the reverse revision parser significantly improves the accuracy of the base parser in almost all languages from CoNLL X and from Conll 2007.

A further improvement can be obtained by combining the output of these two parsers with some other variant, for instance a simple parse in the reverse direction.

In the literature, parser combination is performed by a quadratic algorithm for computing the Maximum Spanning Tree (MST) of the graph consisting of the output of all parsers. In [4] we proposed a greedy algorithm that exploits the fact that the input graphs are already trees, and hence the process can proceed top down from the tree roots. Despite the process is not guaranteed to produce a global optimum, in practice it often outperforms the accuracy of the MST algorithm. In the Evalita 2009 experiments discussed below the algorithm was able to reduce the error rate up to 8% in the pilot task.

6 Experimental Results

Table 1 reports the Labeled Attachment Score (LAS) obtained in the official runs on the Evalita 2009 Dependency Parsing Task, for both the Main and Pilot subtasks.

The Main subtask used a training corpus of 2201 sentences from the Turin University Treebank (TUT) and 200 sentences from the Passage project, for a total of 66,059 tokens. The pilot task used as training 2,999 sentences from the ISST-Tanl corpus for a total of 66,418 tokens.

Table 1. Accuracy on the Evalita 2009 Dependency Parsing Tasks.

Subtask	*Run*	*LAS*	*Best*
Main	mainDepPar_SemaWiki	**88.67**	**88.73**
Pilot	pilotDepPar_SemaWiki	**83.38**	**83.38**

In the main subtask, we exploited the additional annotations provided in TUT, including extra morphological features and semantic annotations. Those features were extracted with a Python script from the FEATS field of the original corpus and added as two new columns, named EXTRA and SEM, to the parser input.

The EXTRA column contains remaining morphological features, like:

```
ADFIRM, ADVERS, CAUS, CONCESS, CONCL, COND, CONSEQ, DOUBT, EXCLUDING,
EXPLIC, FINAL, INTERJ, INTERR|LOC, INTERR|MANNER, INTERR|REASON,
INTERR|TIME, LOC, LOCUTION, MANNER, ORDIN, QUANT, REASON, REFL
```

while the SEM column contains semantic features such as:

```
ASSOCIATION, CITY, COMPANY, CONTINENT, CURRENCY, DENOMINATION,
DISTRICT, GEOGR-AREA, ILLNESS, ISLAND, MOUNTAIN, NAME, ORGANIZATION,
PERSON, PLACE, POLITICIAN, PRICE, REGION, SEA, STATE, SURNAME, TAX
```

For supplying to DeSR data with the additional columns it was enough to provide a corpus description file, extending the standard description of the CoNLL-X format including the descriptions for the two extra columns.

The feature model used for TUT exploited these columns as follows:

```
LEMMA     -2 -1 0 1 2 3 prev(0) leftChild(-1) leftChild(0)
          rightChild(-1) rightChild(0)
POSTAG    -2 -1 0 1 2 3 next(-1) leftChild(-1) leftChild(0)
          rightChild(-1) rightChild(0)
CPOSTAG   -1 0 1
FEATS     -1 0 1
DEPREL    leftChild(-1) leftChild(0) rightChild(-1)
EXTRA     -1 0 1
SEM       -2 -1 0 1 2 3
```

Each line describes which token attribute to use as a feature and for which tokens to extract it: tokens are numbered according to their relative position on the stack and input queue of the parser in each state.

For example the first line tells to use the LEMMA of the top two tokens on the stack (negative index) and of the next four token on the input, the LEMMA of the token preceding the one on the input plus the LEMMA of both right and left children of the top stack token and of the next token. The last two lines specify from which tokens to consider features for the two additional columns EXTRA e SEM.

Notice that the model is partially unlexicalized, since it does not use the FORM of tokens. A cutoff was applied to features appearing less than twice in training. Training with this model produced 129,721 training events consisting in 30,696 features.

The stacked Reverse Revision parser used the following additional features:

```
PHLEMMA     1 0 1
PHPOS       1 0 1
PDEP        1 0 1
PLOC        1 0 1
```

where PHLEMMA and PHPOS are respectively the lemma and POS of the predicted head, PDEP is the predicted dependency and PLOC expresses whether the predicted head occurs before or after the token. Notice we reduced the number of such guided features with respect to experiments reported previously [4].

The submitted run uses a combination of the following parsers:

Type	*Algo.*	*Features*	*Cutoff*	*LAS*
Reverse Revision	SVM	SEM -2 -1 0 1 2 3	2	87.63
Reverse	SVM	SEM -1 0 1	0	88.41
Reverse Revision	MLP	SEM -2 -1 0 1 2 3	2	87.06
Combination				**88.67**

The combination was performed used the voting algorithm described in [4].

For the ISST corpus we used the same model except for the EXTRA and SEM features and a four parser combination:

Type	*Algo.*	*Features*	*Cutoff*	*LAS*
Reverse Revision	SVM		2	81.84
Reverse Revision	SVM	LexChildNonWord	0	81.76
Reverse	SVM	FORM -1 0 1	2	81.12
Reverse	SVM	VerbCount	2	80.78
Combination				**83.38**

The major difference is in the third component which uses the FORM of the previous and two next tokens. The second component uses no cutoff and an additional feature for noting when token has a child which is a symbol, like a punctuation. The fourth variant uses a feature that represents the count of verbs preceding the token.

Notice that adding the form as feature did not produce any improvement.

The choice of the components to use in the submitted run was made on measurement performed on the development test set, selecting the best among 70 experiments on TUT and 27 on ISST. For TUT we randomly split the training corpus into 2215 sentence for train and 186 for development. For ISST we used the supplied development set of 248 sentences.

7 Discussion

We performed some further experiments to assess the impact of the additional features present in the TUT corpus on the overall accuracy.

Running the same parsers disabling the extra features achieves this score:

Run	*Extra Features*	*LAS*
mainDepPar_SemaWiki	no	87.48
mainDepPar_SemaWiki	yes	88.67

Since the evaluation test sets contains portions from the ISST corpus, which has some annotation differences with respect to the TUT corpus used in training, it is interesting to examine the accuracy on the two portions separately both for the Main task and the Pilot task:

Run	*Subset*	*Sentences*	*LAS*
mainDepPar_SemaWiki	TUT	140	**91.75**
mainDepPar_SemaWiki	ISST	100	82.60
pilotDepPar_SemaWiki	TUT	100	**84.67**
pilotDepPar_SemaWiki	ISST	160	82.66

The result on the TUT subset is quite remarkable, especially considering that scores close to 90% could be achieved so far only for languages with much bigger treebanks.

8 Conclusions

The recent extensions to the DeSR parser have proved quite effective in improving its accuracy and effectiveness. In particular we exploited a stacked combination of MLP and SVM parsers, a beam search strategy and a suitable feature model. With these setting the parser achieved the best accuracy in both the Main an Pilot subtasks, showing that it can adapt quite well to different corpora.

Acknowledgments. This research was funded in part by Fondazione Cassa di Risparmio di Pisa within project SemaWiki.

References

1. Attardi, G.: Experiments with a Multilanguage non-projective dependency parser. In: Proc. of the Tenth CoNLL, (2006)
2. Attardi, G., Chanev, A., Ciaramita, M., Dell'Orletta, F., Simi, M.: Multilingual Dependency Parsing and Domain Adaptation using DeSR. In: Proc. of the CoNLL Shared Task Session of EMNLP-CoNLL 2007 (2007)
3. Attardi, G., Simi, M.: DeSR at the Evalita Dependency Parsing Task. In: Proc. of Workshop of Evalita 2007. Intelligenza Aritificiale, 4(2) (2007)
4. Attardi, G., Dell'Orletta, F.: Reverse Revision and Linear Tree Combination for Dependency Parsing. In: Proc. of NAACL HLT 2009 (2009)
5. Bergstra, J., Desjardins, G., Lamblin, P., Bengio, Y.: Quadratic Polynomials Learn Better Image Features. TR-1337, DIRO, Université de Montréal (2009)
6. Villemonte de La Clergerie, É., Hamon, O., Mostefa, D., Ayache, C., Paroubek, P., Vilnat, A.: PASSAGE: from French Parser Evaluation to Large Sized Treebank. In: Proc. of LREC 2008 (2008)
7. Nivre, J., Scholz, M.: Deterministic Dependency Parsing of English Text. In: Proc. of COLING 2004, pp. 64—70 (2004)
8. Nivre, J., McDonald, R.: Integrating Graph-Based and Transition-Based Dependency Parsers. In: Proc. of ACL-08: HLT, pp. 950--958 (2009)
9. Surdeanu, M., et al.: The CoNLL-2008 shared task on joint parsing of syntactic and semantic dependencies. In: Proc. of CoNLL 2008, pp. 159--157 (2008)
10. Theano v0.1, http://lgcm.iro.umontreal.ca:8000/theano/
11. uBLAS. Boost Basic Linear Algebra, http://www.boost.org/doc/libs/1_40_0/libs/numeric/ublas/doc/index.htm