Dependency parsing with neural scoring and randomized greedy inference

6.806/6.864 Final Project
Leon Lin and Tom Yan

December 14, 2015

Abstract

In this project we treat dependency parsing as structured prediction, using 1) a simple feedforward network for scoring parse trees and 2) the randomized greedy inference algorithm of Zhang and Lei (2014) for finding the parse tree with the highest score. We show that learning the weights of the neural network in an end-to-end fashion with max-margin training produces better results than training independently of the inference algorithm. We also compare performance to various other baselines.[1]

Neural networks have been widely used to score or evaluate parts of parse trees in order to make local decisions (see for example Chen and Manning[2] or Durrett and Klein[3]). They allow for richer scoring than, for example, learned feature weights used in a log-linear model.

A rich, higher-order scoring function complicates inference, however. To navigate this trade-off, we use the greedy algorithm of Zhang and Lei[4] for inference, which they show performs strongly on parsing despite being only an approximate method. This hill-climbing algorithm, which treats parsing and structured prediction, is fairly general and can be applied to a scoring function with higher-order or global components. It has previously been used successfully to optimize over a scoring function based on low-rank tensors.[5]

[1] Code at https://github.mit.edu/leonxlin/tom-leon

[2] Danqi Chen and Christopher D Manning. A Fast and Accurate Dependency Parser using Neural Networks. Proceedings of EMNLP 2014.

[3] Greg Durrett and Dan Klein. Neural CRF Parsing. ACL 2015.

[4] Yuan Zhang, Tao Lei, Regina Barzilay and Tommi Jaakkola. Greed is Good if Randomized: New Inference for Dependency Parsing. EMNLP 2014.

[5] Tao Lei, Yu Xin, Yuan Zhang, Regina Barzilay, and Tommi Jaakkola. Low-rank tensors for scoring dependency structures. ACL 2014.

As far as we know the combination (suggested to us by Prof. Jaakkola) of the neural scoring function for arcs and greedy inference algorithm to find the highest-scoring tree is novel.

In this project we consider two ways to learn the weights of the neural network for scoring. First, they can be trained to simply classify arcs either good (1) or bad (0) based on which arcs are present in the training data. The inference algorithm is not used until decoding.

The second training method is max-margin training, which has been shown to work well in a number of NLP tasks.[6] The neural network is trained to separate the scores of correct parse trees and incorrect parse trees based on the number of incorrect arcs. The training examples are chosen using greedy inference.

1 Model

We decompose the score of a parse tree y for a sentence x, where y is represented as a set of head-modifier integer index pairs, into the sum of the scores of the arcs in the tree:

$$S(x, y) = \sum_{(h,m) \in y} S(x, h, m).$$

A simple scoring function might be a linear combination of some extracted features $\phi(x, h, m)$, say, $S(x, h, m) = \theta \cdot \phi(x, h, m)$.

In our set-up, $S(x, h, m)$ is the output of a two-layer fully connected neural network with $\phi(x, h, m)$ as input and a single scalar score as output. To counteract sparsity problems and overfitting, we used a compressed ReLU hidden layer of size 25 or 50, at most a quarter the size of the input layer.

If the set of possible parse trees for x is $\mathcal{T}(x)$, then the parsing problem asks for

$$\arg\max_{y \in \mathcal{T}(x)} S(x, y).$$

We use greedy inference to find this best parse tree. Briefly: The algorithm begins with a random parse tree and makes arc modifications that each improve the score until a local maximum is attained. This process is repeated some number of times with different random starting parse trees. The best final tree among all of these iterations is returned.

[6] Jonathan K. Kummerfeld, Taylor Berg-Kirkpatrick, and Dan Klein. An Empirical Analysis of Optimization for Max-Margin NLP. EMNLP 2015.

2 Training

With both training methods, we used stochastic gradient descent with early stopping based on a held-out validation set.

Under the arc classification method, the neural network is trained to minimize the squared loss between its score and 1 or 0, depending on whether the arc was present in the gold tree of the sentence. False arcs are obtained using negative sampling. During training, the network is fed an equal number of correct and incorrect example arcs: one correct head and one incorrect head for each possible modifier.

(Note that it doesn't necessarily make sense to model S using logistic regression, since it wouldn't define a probability distribution over $\mathcal{T}(x)$.)

Under max-margin training, we want that given a sentence x with gold parse tree $\tilde{y}$, we have for any parse tree $y \in \mathcal{T}(x)$,

$$S(x, \tilde{y}) \geq S(x, y) + \delta(y, \tilde{y}),$$

where $\delta(y, \tilde{y})$ is the number of words with different heads in y and $\tilde{y}$. For each training sentence x, the neural network weights θ are updated as follows:

$$\theta \leftarrow \theta - \eta \nabla L(x, \hat{y}, \tilde{y})$$

where

$$L(x, y, \tilde{y}) = \max\left(0, S(x, y) + \delta(y, \tilde{y}) - S(x, \tilde{y})\right)$$

can be treated as a function of θ and

$$\hat{y} = \arg\max_{y \in \mathcal{T}(x)} L(x, y, \tilde{y})$$

is the "worst violator" at each training step, found using greedy inference.

3 Results and discussion

We report results on a data set of 39279 training sentences, 1334 validation (dev) sentences, and 2399 test sentences.[7]

We trained and tested models using the following bundles of features for each arc:

[7] This data set was provided to us by Yuan Zhang and is the data set used for the CoNLL 2012 shared task. It is our understanding that the data is derived from the Penn Treebank and uses the standard division of Sections 2–20 for training, Section 24 for validation, and Section 23 for test.

Unlexicalized, no context Arc length, parts of speech (one-hot[8]) of the head and modifier (101 features)

Unlexicalized, 5-gram Above features, plus parts of speech of words within 2 positions of either the head or modifier (581 features)

Lexicalized Above features, plus 50-dimensional GLoVe word vectors[9] for the head and modifier each, plus 2 flags for an out-of-vocabulary head or modifier (683 features)

Lexicalized, more embeddings Above features, plus GLoVe word vectors for words within 2 positions of either the head or modifier (1091 features)

Higher-order Arc length; parts of speech and word vectors for the head, modifier, the head word's head, and the modifier's next sibling to the right; parts of speech of words immediately adjacent to either the head or modifier (669 features)

First we verify that the simple two-layer feedforward network outperforms a linear model with $S(x, h, m) = \theta \cdot \phi(x, h, m)$ where $\phi(x, h, m)$ contains arc length, the respective parts of speech of the head and modifier, and the conjunction of the parts of speech (given that our data set has 50 parts of speech, this is 2601 features). These features can be compared to the "Unlexicalized, no context" setting for a neural network. This linear model fit with L_1 regularization achieves a UAS less than 50%.

The results using the neural network trained with the arc classification method and the max-margin approach are shown in Figure 1. We find that in general the max-margin approach improves upon the arc classification method. Furthermore, the max-margin approach attained its peak results on the dev set usually after only 1 or 2 epochs (i.e., passes over the training set), whereas the arc classification method took anywhere between 10 and 20 epochs. (On the other hand, the max-margin approach was always much slower in terms of runtime, requiring $O(n^2)$ invocations of the scoring function, where n is the sentence length, per sentence per epoch.)

One explanation for the max-margin approach's superior performance might be that because it trains the scoring function on only parse trees

[8] We considered learning dense representations of parts of speech, but did not pursue that route, since each part of speech is reasonably well represented in the training data to begin with.

[9] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global Vectors for Word Representation. 2014.

	Arc classification		Max-margin training	
	Dev	Test	Dev	Test
Unlexicalized, no context (A)	69.54	**69.62**	68.30	68.36
Unlexicalized, no context (B)	69.25	69.15	71.28	**71.55**
Unlexicalized, 5-gram	79.12	79.30	81.39	**82.17**
Lexicalized	81.60	82.08	84.14*	**84.49***
Lexicalized, more embeddings	80.73	81.44		
Higher-order				

Figure 1: Unlabeled attachment scores for arc classification training and max-margin training. For "Unlexicalized, no context (A)", 20 restarts were used in greedy inference; for (B), 300 restarts; for all other feature settings, 100 restarts. Some results are not reported because training took too long. See the text for details. *These results were obtained by initializing the network weights to those at the end of arc classification training but only training with max-margin for one additional epoch, due to time constraints.

erroneously judged to be better than they are, the scoring function is able to learn finer and subtler distinctions between good and bad arcs. For a fairer match-up, the arc classification training method could be modified to only train specifically on (true and false) arcs that are currently scored badly.

Another hypothesis is that when the score function is trained along with the inference algorithm that will be used for decoding, the score function can adapt to the idiosyncrasies of the approximate inference algorithm and perhaps learn ways to smooth out local maxima, for example, to make inference itself more likely to succeed. This could be tested more thoroughly by varying the strength of the inference algorithm (e.g., by varying the number of random restarts allowed) during both training and testing to see which training method is more robust to such changes.

4 Implementation challenges

Unfortunately our implementation of the model became infeasibly slow to train and test as more features were added, especially higher-order features. For example, it took an hour for an unlexicalized higher-order model to parse 250 sentences with 10 restarts each. (Extrapolating, lexicalized max-margin training on the full training set with a reasonable number of restarts and

epochs would take months.) Thus we did not get the opportunity to iterate on our model to see whether higher-order features, hyperparameter changes, and so on could have brought performance closer to the state of the art.

A profile of the program revealed that more than 96% of computing time was devoted to evaluating the score function, which was carried out by the Theano package (and invoked tens of millions of times per training epoch by the randomized greedy inference algorithm). We could not expect to speed up Theano itself, except perhaps by using GPUs, but instead considered ways to cut down on the number of calls to the score function.

Here are some of the steps we took:

Hardware We stopped training the models on our own laptops and used the rosetta CSAIL machines in Prof. Barzilay's lab instead. Mostly we used the two 40-core machines; using a GPU made performance worse for reasons that are still unclear to us.

Partial scoring During hill-climbing, the inference algorithm only needs to make comparisons between local candidate tree modifications; thus it should only evaluate a carefully designed, lighter partial score function rather than attempt to score entire trees that differ very little from each other. This cut runtimes down significantly.

Caching/memoization With first-order features, there are a small number possible arc feature vectors in a sentence, so that the output of the scoring function on each possible arc can be cached. During hill-climbing, the inference algorithm often needs to score the same arc again, for example following a random restart. This method cut runtimes down significantly for first-order parsing, but was not yet implemented for higher-order features.

Parallelization We attempted to run greedy restarts in parallel processes, but the overhead of starting processes killed any performance gain. Running restarts in separate processes also hinders score function caching. Instead, running greedy inference on different sentences in parallel proved more effective. This is fine for testing, but during training the score function changes with every update, so in order to parallelize sentence processing, we had to use minibatch training.

Minibatch training Instead of updating the neural network weights after processing each sentence, we only update (in proportion to the sum of gradients) after every 30 sentences. This facilitates parallelization.

"Vectorization" A common piece of advice is to consolidate calls to vector operations such as the score function, which operate efficiently over large arrays. We did this for scoring sentences (scoring all arcs at once), but could not find another place to apply this without getting in the way of either caching or parallelization.

Pruning This is an approach we have not yet attempted.

5 Division of work

Tom implemented the neural network, wrote the first version of the high-level code for the max-margin and arc classification training algorithms, contributed to the code for extracting features from the corpus, obtained and processed the GLoVe word embeddings, drafted this report, and picked this project.

Leon implemented the random greedy inference algorithm, contributed to and debugged the code for the training algorithms, spent a lot of time trying to make things run faster, wrote unit tests, wrote code to extract feature vectors from the corpus, ran the training and test jobs on the rosetta computers, and wrote most of this report.