Deep Learning for Efficient Discriminative Parsing

Ronan Collobert[†]
IDIAP Research Institute
1920 Martigny, CH
ronan.collobert@idiap.ch

Abstract

We propose a new fast *purely discriminative* algorithm for natural language parsing, based on a "deep" recurrent convolutional graph transformer network (GTN). Assuming a decomposition of a parse tree into a stack of "levels", the network predicts a level of the tree taking into account predictions of previous levels. Using only few basic text features, we show similar performance (in F_1 score) to existing *pure* discriminative parsers and existing "benchmark" parsers (like Collins parser, probabilistic context-free grammars based), with a huge speed advantage.

1 Introduction

Parsing has been pursued with tremendous efforts in the Natural Language Processing (NLP) community. Since the introduction of *lexicalized*[1] probabilistic context-free grammar (PCFGs) parsers [1, 2], improvements have been achieved over the years, but *generative* PCFGs parsers of the last decade from Collins [3] and Charniak [4] still remain standard benchmarks. Given the success of discriminative learning algorithms for classical NLP tasks (Part-Of-Speech (POS) tagging, Name Entity Recognition, Chunking...), the generative nature of such parsers has been questioned. First discriminative parsing algorithms [5, 6] did not reach standard PCFG-based generative parsers. Henderson [6] outperforms Collins parser only by using a generative model and performing re-ranking. Charniak [7] also successfully leveraged re-ranking. Pure discriminative parsers from Taskar [8] and Turian [9] finally reached Collins' parser performance, with various simple template features. However, these parsers were slow to train and were both limited to sentences with less than 15 words. Most recent discriminative [10, 11] parsers are based on Conditional Random Fields (CRFs) with PCFG-like features. In the same spirit, Carreras et al [12] use a global-linear model (instead of a CRF), with PCFG and dependency features.

We motivate our work with the fundamental question: how far can we go with discriminative parsing, with as little prior information as possible? We propose a *fast* new discriminative parser which not only does not rely on information extracted from PCFGs, but does not rely on most classical parsing features. In fact, with only few basic text features and Part-Of-Speech (POS), it performs similarly to Taskar and Turian's parsers on small sentences, and similarly to Collins' parser on long sentences.

We trade this reduction of features for a "deeper" architecture, a.k.a. a particular deep neural network. As is the case for choosing parsing features, we acknowledge that training a neural network is a task which requires some experience. From our perspective, this knowledge allows however flexible and generic architectures. Indeed, from a deep learning point of view, our approach is quite conventional, based on a *convolutional* neural network (CNN) adapted for text. CNNs were successful very early for tasks involving sequential data [13]. They have also been applied to NLP [14, 15], but limited to "flat" tagging problems. We combine CNNs with a structured tag inference in a graph, the resulting model being called a *Graph Transformer Network* (GTN) [16]. Again, this is not a sur-

[†]Most of this work as been achieved when Ronan Collobert was at NEC Laboratories America.

[1]Which leverage head words of parsing constituents.

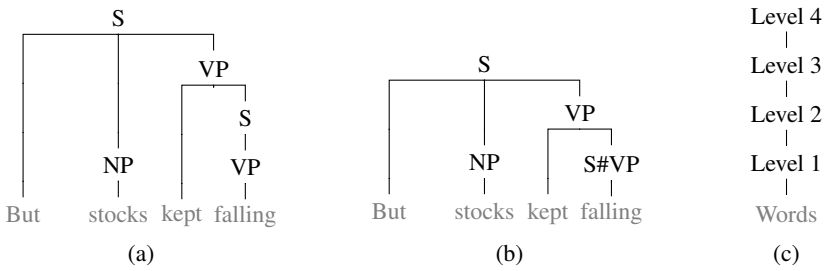

Figure 1: Parse Trees representation. As in Penn Treebank (a), and after concatenating nodes spanning same words (b). In (c) we show our definition of "levels".

prising architecture: GTNs are for deep models what CRFs are for linear models [17], and CRFs had great success in NLP [18, 19, 20].

We convert parse trees into a stack of levels, and then train a recursive GTN which predicts a "level" of the tree based on predictions of previous levels. This approach shares some similarity with the finite-state parsing cascades from Abney [21]. However, Abney's algorithm was limited to partial parsing, because each level of the tree was predicted by its own tagger: the maximum depth of the tree had to be chosen beforehand.

In Section 2 we describe how we convert trees to (and from) a stack of levels. Section 3 describes our GTN architecture for text. Section 4 shows how to implement necessary constraints to get a valid tree from a level decomposition. Evaluation of our system on standard benchmarks is given in Section 5.

2 Parse Trees

We consider linguistic parse trees as described in Figure 1a. The root spans all of the sentence, and is recursively decomposed into sub-constituents (the nodes of the tree) with labels like NP (noun phrase), VP (verb phrase), S (sentence), etc. The tree leaves contain the sentence words. All our experiments were performed using the Penn Treebank dataset [22], on which we applied several standard pre-processing steps: (1) functional labels as well as traces were removed (2) the label PRT was converted into ADVP (see [1]) (3) duplicate constituents (spanning the same words and with the same label) were removed. The resulting dataset contains 26 different labels, that we will denote $\mathcal{L}$ in the rest of the paper.

2.1 Parse Tree Levels

Many NLP tasks involve finding chunks of words in a sentence, which can be viewed as a tagging task. For instance, "Chunking" is a task related to parsing, where one wants to obtain the label of the lowest parse tree node in which each word ends up. For the tree in Figure 1a, the pairs word/chunking tags could be written as: But/O stocks/S-NP kept/B-VP falling/E-VP. We chose here to adopt the IOBES tagging scheme to mark chunk boundaries. Tag "S-NP" is used to mark a noun phrase containing a single word. Otherwise tags "B-NP", "I-NP", and "E-NP" are used to mark the first, intermediate and last words of the noun phrase. An additional tag "O" marks words that are not members of a chunk.

As illustrated in Figure 2, one can rewrite a parse tree as a stack of tag levels (see Figure 1c). We achieve this tree conversion by first transforming the lowest nodes of the parse tree into chunk tags ('Level 1"). Tree nodes which contain sub-nodes are ignored at this stage[2]. Words not into one of the lowest nodes are tagged as "O". We then strip the lowest nodes of the tree, and apply the same principle for "Level 2". We repeat the process until one level contains the root node. We chose a bottom-up approach because one can rely very well on lower level predictions: the chunking

[2]E.g. in Figure 1a, "kept" is not tagged as "S-VP" in Level 1, as the node "VP" still contains sub-nodes "S" and "VP" above "falling".

Level 4	B-S	I-S	I-S	E-S
Level 3	O	O	B-VP	E-VP
Level 2	O	O	O	S-S
Level 1	O	S-NP	O	S-VP
Words	But	stocks	kept	falling

Figure 2: The parse tree shown in Figure 1a, rewritten as four levels of tagging tasks.

task, which describes in an other way the lowest parse tree nodes, has a very good performance record [18].

2.2 From Tagging Levels To Parse Trees

Even if it had success with partial parsing [21], the simplest scheme where one would have a different tagger for each level of the parse tree is not attractive in a full parsing setting. The maximum number of levels would have to be chosen at train time, which limits the maximum sentence length at test time. Instead, we propose to have a *unique* tagger for all parse tree levels:

1. Our tagger starts by predicting Level 1.
2. We then predict next level according to a history of previous levels, with the same tagger.
3. We update the history of levels and go to 2.

This setup fits naturally into the recursive definition of the levels. However, we must insure the predicted tags correspond to a parse tree. In a tree, a parent node fully includes child nodes. Without constraints during the level predictions, one could face a chunk partially spanning another chunk at a lower level, which would break this tree constraint.

We can guarantee that the tagging process corresponds to a valid tree, by adding a constraint enforcing higher level chunks to fully include lower level chunks. This iterative process might however never end, as it can be subject to loops: for instance, the constraint is still satisfied if the tagger predicts the same tags for two consecutive levels. We propose to tackle this problem by (a) modifying the training parse trees such that nodes grow *strictly* as we go up in the tree and (b) enforcing the corresponding constraints in the tagging process.

Tree nodes spanning the same words for several consecutive level are first replaced by one node in the whole training set. The label of this new node is the concatenation of replaced node labels (see Figure 1b). At test time, the inverse operation is performed on nodes having concatenated labels. Considering all possible label combinations would be intractable[3]. We kept in the training set concatenated labels which were occurring at least 30 times (corresponding to the lowest number of occurrences of the less common non-concatenated tag). This added 14 extra labels to the 26 we already had. Adding the extra O tag and using the IOBES tagging scheme led us to 161 ($(26+14) \times 4 + 1$) different tags produced by our tagger. We denote $\mathcal{T}$ this ensemble of tags.

With this additional pre-processing, any tree node is strictly larger (in terms of words it spans) than each of its children. We enforce the corresponding Constraint 1 during the iterative tagging process.

Constraint 1 *Any chunk at level i overlapping a chunk at level $j < i$ must span at least this overlapped chunk, and be larger.*

As a result, the iterative tagging process described above will generate a chunk of size N in at most N levels, given a sentence of N words. At this time, the iterative loop is stopped, and the full tree can be deduced. The process might also be stopped if no new chunks were found (all tags were O). Assuming our simple tree pre-processing has been done, this generic algorithm could be used with any tagger which could handle a history of labels and tagging constraints. Even though the tagging process is *greedy* because there is no *global* inference of the tree, we will see in Section 5 that it can perform surprisingly well. We propose in the next section a tagger based on a convolutional Graph

[3]Note that more than two labels might be concatenated. E.g., the tag SBAR#S#VP is quite common in the training set.

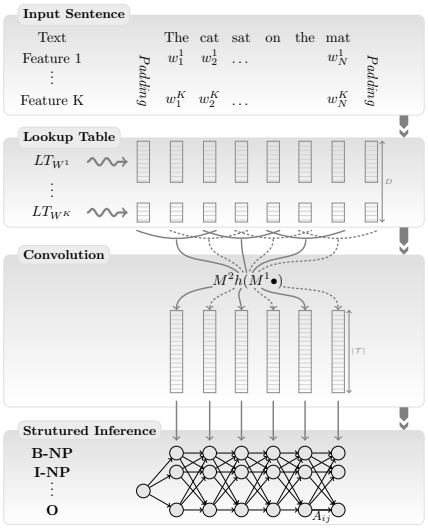

Figure 3: Our neural network architecture. Words and other desired discrete features (caps, tree history, ...) are given as input. The lookup tables embed each feature in a vector space, for each word. This is fed in a convolutional network which outputs a score for each tag and each word. Finally, a graph is output with network scores on the nodes and additional transition scores on the edges. A Viterbi algorithm is performed to infer the word tags.

Transformer Network (GTN) architecture. We will see in Section 4 how we keep track of the history and how we implement Constraint 1 for that tagger.

3 Architecture

We chose to use a variant of the versatile convolutional neural network architecture first proposed in [14] for language modeling, and reintroduced later in [15] for various NLP tasks involving tagging. Our network outputs a graph over which inference is achieved with a Viterbi algorithm. In that respect, one can see the whole architecture (see Figure 3) as an instance of GTNs [16, 23]. All network and graph parameters are trained in a end-to-end way with stochastic gradient maximizing a graph likelihood. We first describe in this section how we adapt neural networks to text data, and then we introduce GTNs training procedure. We will show in Section 4 how one can further adapt this architecture for parsing, by introducing a tree history feature and few graph constraints.

Word Embeddings We consider a fixed-sized word dictionary[4] $\mathcal{W}$. Given a sentence of N words $\{w_1, w_2, \ldots, w_N\}$, each word $w_n \in \mathcal{W}$ is first embedded into a D-dimensional vector space, by applying a lookup-table operation:

$$LT_W(w_n) = W \times \left(0, \cdots 0, \underset{\text{at index } w_n}{1}, 0, \cdots 0\right)^{\mathrm{T}} = W_{w_n}\,, \tag{1}$$

where the matrix $W \in \mathbb{R}^{D \times |\mathcal{W}|}$ represents the parameters to be *trained* in this lookup layer. Each column $W_n \in \mathbb{R}^D$ corresponds to the embedding of the n^{th} word in our dictionary $\mathcal{W}$. Having in mind the matrix-vector notation in (1), the lookup-table applied over the sentence can be seen as an efficient implementation of a convolution with a kernel width of size 1.

In practice, it is common that one wants to represent a word with more than one feature. In our experiments we always took at least the low-caps words and a "caps" feature: $w_n = (w_n^{lowcaps}, w_n^{caps})$. In this case, we apply a different lookup-table for each discrete feature ($LT_{W^{\text{lowcaps}}}$ and $LT_{W^{\text{caps}}}$), and the word embedding becomes the concatenation of the output of all these lookup-tables:

$$LT_{W^{\text{words}}}(w_n) = \left(LT_{W^{\text{lowcaps}}}(w_n^{\text{lowcaps}})^{\mathrm{T}},\, LT_{W^{\text{caps}}}(w_n^{\text{caps}})\right)^{\mathrm{T}}\,. \tag{2}$$

For simplicity, we consider only one lookup-table in the rest of the architecture description.

[4]Unknown words are mapped to a special UNKNOWN word. Also, we map numbers to a NUMBER word.

Word Scoring Scores for all tags $\mathcal{T}$ and all words in the sentence are produced by applying a classical convolutional neural network over the lookup-table embeddings (1). More precisely, we consider all successive windows of text (of size K), sliding over the sentence, from position 1 to N. At position n, the the network is fed with the vector $\boldsymbol{x}_n$ resulting from the concatenation of the embeddings:

$$\boldsymbol{x}_n = \left(W^{\mathrm{T}}_{\boldsymbol{w}_{n-(K-1)/2}}, \ldots, W^{\mathrm{T}}_{\boldsymbol{w}_{n+(K-1)/2}} \right)^{\mathrm{T}} .$$

The words with indices exceeding the sentence boundaries ($n-(K-1)/2 < 1$ or $n+(K-1)/2 > N$) are mapped to a special PADDING word. As any classical neural network, our architecture performs several matrix-vector operations on its inputs, interleaved with some non-linear transfer function $h(\cdot)$. It outputs a vector of size $|\mathcal{T}|$ for each word at position n, interpreted as a score for each tag in $\mathcal{T}$ and each word w_n in the sentence:

$$\boldsymbol{s}(\boldsymbol{x}_n) = M^2 \, h(M^1 \, \boldsymbol{x}_n) \, , \tag{3}$$

where the matrices $M^1 \in \mathbb{R}^{H \times (KD)}$ and $M^2 \in \mathbb{R}^{|\mathcal{T}| \times H}$ are the trained parameters of the network. The number of hidden units H is a hyper-parameter to be tuned. As transfer function, we chose the hyperbolic tangent $h(z) = \tanh(z)$ in our experiments.

Long-Range Dependencies The "window" approach proposed above assume that the tag of a word is solely determined by the surrounding words in the window. As we will see in our experiments, this approach falls short on long sentences. Inspired by [15], we consider a variant of this architecture, where *all* words $\{w_1, w_2, \ldots, w_N\}$ are considered for tagging a given word w_n. To specifiy to the network that we want to tag the word w_n, we introduce an additional lookup-table in (2), which embeds the *relative distance* $(m-n)$ of each word w_m in the sentence with respect to w_n. At each position $1 \leq m \leq N$, the outputs of the all lookup-tables (2) (low caps word, caps, relative distance...) $LT_{W^{\mathrm{words}}}(w_m)$ are first combined together by applying a mapping M^0. We then extract a fixed-size "global" feature vector $\boldsymbol{x}_n$ by performing a $\max$ over the sentence:

$$[\boldsymbol{x}_n]_i = \max_{1 \leq m \leq N} \left[M^0 \, LT_{W^{\mathrm{words}}}(w_m) \right]_i \quad \forall i \tag{4}$$

This feature vector is then fed to scoring layers (3). The matrix M^0 is *trained* by back-propagation, as any other network parameter. We will refer this approach as "sentence approach" in the following.

Structured Tag Inference We know that there are strong dependencies between parsing tags in a sentence: not only are tags organized in chunks, but some tags cannot follow other tags. It is thus natural to infer tags from the scores in (3) using a structured output approach. We introduce a transition score A_{tu} for jumping from tags $t \in \mathcal{T}$ to $u \in \mathcal{T}$ in successive words, and an initial score A_{t0} for starting from the t^{th} tag. The last layer of our network outputs a graph with $|\mathcal{T}| \times N$ nodes G_{tn} (see Figure 3). Each node G_{tn} is assigned a score $\boldsymbol{s}(\boldsymbol{x}_n)_t$ from the previous layer (3) of our architecture. Given a pair of nodes G_{tn} and G_{um}, we add an edge with transition score A_{tu} on the graph. For compactness, we use the sequence notation $[t]_1^N \triangleq \{t_1, \ldots, t_n\}$ for now. We score a tag path $[t]_1^N$ in the graph G, as the sum of scores along $[t]_1^N$ in G:

$$S([w]_1^N, [t]_1^N, \boldsymbol{\theta}) = \sum_{n=1}^{N} \left(A_{t_{n-1} t_n} + s(\boldsymbol{x}_n)_{t_n} \right) , \tag{5}$$

where $\boldsymbol{\theta}$ represents all the trainable parameters of our complete architecture (W, M^1, M^2 and A). The sentence tags $[t^\star]_1^N$ are then inferred by finding the path which leads to the maximal score:

$$[t^\star]_1^N = \underset{[t]_1^N \in \mathcal{T}^N}{\operatorname{argmax}} \, S([w]_1^N, [t]_1^N, \boldsymbol{\theta}) \, . \tag{6}$$

The Viterbi algorithm [24] is the natural choice for this inference. We will show now how to train all the parameters of the network $\boldsymbol{\theta}$ in a end-to-end way.

Training Likelihood Following the GTN's training method introduced in [16, 23], we consider a probabilistic framework, where we maximize a likelihood over all the sentences $[w]_1^N$ in our training set, with respect to $\boldsymbol{\theta}$. The score (5) can be interpreted as a conditional probability over a path by

taking it to the exponential (making it positive) and normalizing with respect to all possible paths (summing to 1 over all paths). Taking the $\log(\cdot)$ leads to the following conditional log-probability:

$$\log p([t]_1^N \mid [w]_1^N, \boldsymbol{\theta}) = S([w]_1^N, [t]_1^N, \boldsymbol{\theta}) - \underset{\forall [u]_1^N \in \mathcal{T}^N}{\text{logadd}} S([w]_1^N, [u]_1^N, \boldsymbol{\theta}) \,, \tag{7}$$

where we adopt the notation $\text{logadd}_i \, z_i = \log(\sum_i e^{z_i})$. This likelihood is the same as the one found in Conditional Random Fields (CRFs) [17] over temporal sequences. The CRF model is however linear (which would correspond in our case to a linear neural network, with fixed word embeddings).

Computing the log-likelihood (7) efficiently is not straightforward, as the number of terms in the logadd grows exponentially with the length of the sentence. Fortunately, in the same spirit as the Viterbi algorithm, one can compute it in linear time with the following classical recursion over n:

$$\begin{aligned}
\delta_n(v) &\triangleq \underset{\{[u]_1^n \cap u_n = v\}}{\text{logadd}} S([w]_1^n, [u]_1^n, \boldsymbol{\theta}) \\
&= \underset{t}{\text{logadd}} \underset{\{[u]_1^n \cap u_{n-1} = t \cap u_n = v\}}{\text{logadd}} S([w]_1^{n-1}, [u]_1^{n-1}, \boldsymbol{\theta}) + A_{u_{n-1} v} + \boldsymbol{s}(\boldsymbol{x}_n)_v \\
&= \boldsymbol{s}(\boldsymbol{x}_n)_v + \underset{t}{\text{logadd}} \, (\delta_{n-1}(t) + A_{tv}) \quad \forall v \in \mathcal{T} \,,
\end{aligned} \tag{8}$$

followed by the termination $\text{logadd}_{\forall [u]_1^N} S([w]_1^N, [u]_1^N, \boldsymbol{\theta}) = \text{logadd}_u \, \delta_N(u)$. As a comparison, the Viterbi algorithm used to perform the inference (6) is achieved with the same recursion, but where the logadd is replaced by a max, and then tracking back the optimal path through each max.

Stochastic Gradient We maximize the log-likelihood (7) using stochastic gradient ascent, which has the main advantage to be extremely scalable [25]. Random training sentences $[w]_1^N$ and their associated tag labeling $[t]_1^N$ are iteratively selected. The following gradient step is then performed:

$$\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} + \lambda \, \partial \log p([t]_1^N \mid [w]_1^N, \boldsymbol{\theta}) / \partial \boldsymbol{\theta} \,, \tag{9}$$

where λ is a chosen learning rate. The gradient in (9) is efficiently computed via a classical back-propagation [26]: the differentiation chain rule is applied to the recursion (8), and then to all network layers (3), including the word embedding layers (1). Derivations are simple (but fastidious) algebra, and will be reported in a longer version of this paper.

4 Chunk History and Tree Constraints

The neural network architecture we presented in Section 3 is made "recursive" by adding an additional feature (and its corresponding lookup-table (1)) describing a history of previous tree levels. For that purpose, we gather all chunks which were discovered in previous tree levels. If several chunks were overlapping at different levels, we consider only the largest one. Assuming Constraint 1 is true, a word can be at most in one of the remaining chunk. This is our history[5] $\mathcal{C}$. The corresponding IOBES tags of each word will be fed as feature to the GTN. For instance, assuming the labeling in Figure 2 was found up to Level 3, the chunks we would consider in $\mathcal{C}$ for tagging Level 4 would be only the NP around "stocks" and the VP around "kept falling". We would discard the S and VP around "falling" as they are included by the larger VP chunk.

Implementing Constraint 1 is now made easy using this history $\mathcal{C}$ and a IOBES tagging scheme. For each chunk $\boldsymbol{c} \in \mathcal{C}$, we adapt the graph output by our network Figure 3 such that any new candidate chunk $\tilde{\boldsymbol{c}}$ overlapping $\boldsymbol{c}$ includes $\boldsymbol{c}$, and is larger than $\boldsymbol{c}$. For each candidate label (say VP) we create three possible paths (see Figure 4) for the duration of $\boldsymbol{c}$: (1) Both $\boldsymbol{c}$ and $\tilde{\boldsymbol{c}}$ starts at the same position. The first tag of $\tilde{\boldsymbol{c}}$ is then B-VP, and remaining tags overlapping with $\boldsymbol{c}$ are maintained at I-VP. In this way, $\tilde{\boldsymbol{c}}$ has to end after $\boldsymbol{c}$. (2) Both $\boldsymbol{c}$ and $\tilde{\boldsymbol{c}}$ end at the same position. The last tag of $\tilde{\boldsymbol{c}}$ is then E-VP, and previous tags overlapping with $\boldsymbol{c}$ are maintained at I-VP. In this way, $\tilde{\boldsymbol{c}}$ has to start before $\boldsymbol{c}$. (3) The candidate chunk $\tilde{\boldsymbol{c}}$ includes $\boldsymbol{c}$ but does not start nor ends at the same position. The path is maintained on I-VP while overlapping $\boldsymbol{c}$. As a result, it will start before and end after $\boldsymbol{c}$. In addition to these $3 \times |\mathcal{L}|$ possible paths overlapping $\boldsymbol{c}$, there is an additional path where no chunk is found over $\boldsymbol{c}$, in which case all tags stay in O while overlapping $\boldsymbol{c}$. Finally, as $\tilde{\boldsymbol{c}}$ must be strictly larger than $\boldsymbol{c}$, any S- tag is discarded for the duration of $\boldsymbol{c}$. Parts of the graph not overlapping with the chunk history $\mathcal{C}$ remain fully connected, as previously described in Section 3.

[5] Some other kind of history could have been chosen (e.g. a feature for each arbitrary chosen $L \in \mathbb{N}$ previous levels). However we still need to "compute" the proposed history for implementing Constraint 1.

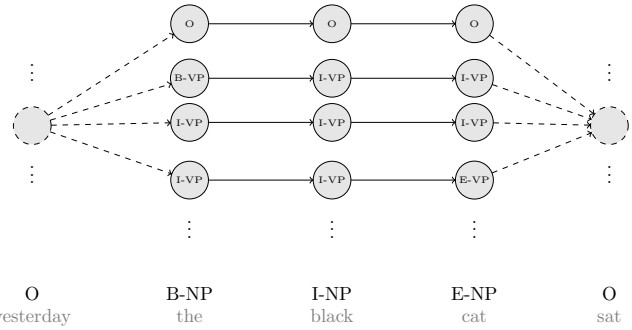

Figure 4: Implementing tree constraints: the chunk history (bottom) contains the NP "the black cat". At the next level the inference graph (top) is constrained for the sections overlapping this chunk, such that new overlapping chunks include "the black cat", and are strictly larger. The top candidate path is taken when no chunk is found (O). The three next candidate paths relate to the candidate label VP. The graph contains similar triplets of paths for all the other labels.

5 Experiments

We conducted our experiments on the standard English Penn Treebank benchmark [22]. Sections 02–21 were used for training, section 22 for validation, and section 23 for testing. Standard preprocessing as described in Section 2 was performed. In addition, the training set trees were transformed such that two nodes spanning the same words were concatenated as described in Section 2.2. We report results on the test set in terms of recall (R), precision (P) and F_1 score. Scores were obtained using the EVALB implementation[6].

Our architecture (see Section 3) was trained on all possible parse tree levels (see Section 2.1), for all sentences available in the training set. Random levels in random sentences were presented to the network until convergence on the validation set. We fed our network with (1) lower cap words (to limit the number of words), (2) a capital letter feature (is low caps, is all caps, had first letter capital, or had one capital) to keep the upper case information (3) the relative distance to the word of interest (only for the "sentence approach") (4) a POS feature[7] (unless otherwise mentioned) (5) the history of previous levels (see Section 4). During training, the true history was given. During testing the history and the tags were obtained recursively from the network outputs, starting from Level 1, (see Section 2.2). All features had a corresponding lookup-table (1) in the network.

Only few hyper-parameters were tried in our models (chosen according to the validation). Lookup-table sizes for the low cap words, caps, POS, relative distance (in the "sentence approach") and history features were respectively 50, 5, 5, 5 and 10. The window size of our convolutional network was $K = 5$. We used the word embeddings obtained from the language model (LM) [15] to initialize the word lookup-table. Finally, we fixed the learning rate $\lambda = 0.01$ during the stochastic gradient procedure (9). The only neural network "tricks" we used were (1) the initialization of the parameters was done according to the fan-in, and (2) the learning rate was divided by the fan-in [27].

Small Scale Experiments First discriminative parse trees were very computationally expensive to train. Taskar et al. [8] proposed a comparison setup for discriminative parsers limited to Penn Treebank sentences with ≤ 15 words. Turian et. al [9] reports almost 5 days of training for their own parser, using parallelization, on this setup. They also report several months of training for Taskar et al.'s parser. In comparison, our parser takes only few hours to train (on a single CPU) on this setup. We report in Table 1 test performance of our *window approach* system ("GTN Parser", with $H = 300$ hidden units) against Taskar and Turian's discriminative parsers. We also report performance of Collins parser [3], a reference in non-discriminative parsers. Not initializing the word lookup table with the language model (LM) and not using POS features performed poorly, similar to experiments reported in [15]. Initializing with the LM but not using POS or using POS

[6] Available at `http://cs.nyu.edu/cs/projects/proteus/evalb`.

[7] Obtained with the tagger available at `http://ml.nec-labs.com/senna`.

Table 1: Comparison of parsers trained and tested on Penn Treebank, on sentences ≤ 15 words, against our GTN parser (window approach).

Model	R	P	F_1
Collins (1999) [3]	88.2	89.2	88.7
Taskar et al. (2004) [8]	89.1	89.1	89.1
Turian and Melamed (2006) [9]	89.3	89.6	89.4
GTN Parser	82.4	82.8	82.6
GTN Parser (LM)	86.1	87.2	86.6
GTN Parser (POS)	87.1	86.2	86.7
GTN Parser (LM+POS)	89.2	89.0	89.1

Table 2: Parsers comparison trained on the full Penn Treebank, and tested on sentences with ≤ 40 and ≤ 100 words. We also report testing time on the test set (Section 23).

	≤ 40 Words			≤ 100 Words			Test Time
	R	P	F_1	R	P	F_1	(sec.)
Magerman (1995) [1]	84.6	84.9	84.8				
Collins (1996) [2]	85.8	86.3	86.1	85.3	85.7	85.5	
Collins (1999) [3]	88.5	88.7	88.6	88.1	88.3	88.2	2640
Charniak (2000) [4]	90.1	90.1	90.1	89.6	89.5	89.6	1020
Charniak & Johnson (2005) [7]			92.0			91.4	
Finkel et al. (2008) [10]	88.8	89.2	89.0	87.8	88.2	88.0	
Petrov et al. (2008) [11]			90.0			89.4	
Carreras et al. (2008) [12]				89.9	91.1	90.5	
GTN Parser (window)	81.3	81.9	81.6	80.3	81.0	80.6	
GTN Parser (window, LM)	84.2	85.7	84.9	83.5	85.1	84.3	
GTN Parser (window, LM+POS)	85.6	86.8	86.2	84.8	86.2	85.5	
GTN Parser (sentence, LM+POS)	88.1	88.8	88.5	87.5	88.3	87.9	76

but not LM gave similar improvements in performance. Combining LM and POS compares well with other parsers.

Large Scale Experiments We also trained (Table 2) our GTN parsers (both the "window" and "sentence" approach) on the full Penn Treebank dataset. Both takes a few days to train on a single CPU in this setup. The number of hidden units was set to $H = 700$. The size of the embedding space obtained with M^0 in the "sentence approach" was 300. Our "window approach" parser compares well against the first lexical PCFG parsers: Magerman (1995) and Collins (1996). The "sentence approach" provides a clear boost and compares well against Collins (1999) parser[8], a standard benchmark in NLP. More refined parsers like Charniak & Johnson (2005) (which takes advantage of re-ranking) or recent discriminative parsers (which are based on PCFGs features) have higher F_1 scores. Our parser performs comparatively well, considering we only used simple text features. Finally, we report some timing results on Penn Treebank test set (many implementations are not available). The GTN parser[9] was an order of magnitude faster than other available parsers.

6 Conclusion

We proposed a new fast and scalable purely discriminative parsing algorithm based on Graph Transformer Networks. With only few basic text features, it performs similarly to existing pure discriminative algorithms, and similarly to Collins (1999) "benchmark" parser. Many paths remain to be explored: richer features (in particular head words, as do *lexicalized* PCFGs), combination with generative parsers, less greedy bottom-up inference (e.g. using K-best decoding), or other alternatives to describe trees.

[8] We picked Bikel's implementation available at `http://www.cis.upenn.edu/~dbikel`.

[9] Our implementation can be downloaded at `http://ml.nec-labs.com/senna/`.

References

[1] D. M. Magerman. Statistical decision-tree models for parsing. In *Proceedings of the 33rd Annual Meeting of the ACL*, pages 276–283, 1995.

[2] M. Collins. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th annual meeting on ACL*, pages 184–191, 1996.

[3] M. Collins. *Head-Driven Statistical Models for Natural Language Parsing*. PhD thesis, University of Pennsylvania, 1999.

[4] E. Charniak. A maximum-entropy-inspired parser. *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, pages 132–139, 2000.

[5] A. Ratnaparkhi. Learning to parse natural language with maximum entropy models. *Machine Learning*, 34(1-3):151–175, 1999.

[6] J. Henderson. Discriminative training of a neural network statistical parser. In *Proceedings of the 42nd Annual Meeting on ACL*, 2004.

[7] E. Charniak and M. Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting on ACL*, pages 173–180, 2005.

[8] B. Taskar, D. Klein, M. Collins, D. Koller, and C. Manning. Max-margin parsing. In *Proceedings of EMNLP*, 2004.

[9] J. Turian and I. D. Melamed. Advances in discriminative parsing. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the ACL*, pages 873–880, 2006.

[10] J. R. Finkel, A. Kleeman, and C. D. Manning. Efficient, feature-based, conditional random field parsing. In *Proceedings of ACL-08: HLT*, pages 959–967. ACL, June 2008.

[11] S. Petrov and D. Klein. Sparse multi-scale grammars for discriminative latent variable parsing. In *EMNLP '08*, pages 867–876. ACL, 2008.

[12] X. Carreras, M. Collins, and T. Koo. Tag, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *CoNLL '08*, pages 9–16. ACL, 2008.

[13] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[14] Y. Bengio and R. Ducharme. A neural probabilistic language model. In *NIPS 13*, 2001.

[15] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, 2008.

[16] L. Bottou, Y. LeCun, and Yoshua Bengio. Global training of document processing systems using graph transformer networks. In *Proceedings of CVPR*, pages 489–493, 1997.

[17] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Eighteenth International Conference on Machine Learning, ICML*, 2001.

[18] F. Sha and F. Pereira. Shallow parsing with conditional random fields. In *NAACL 2003*, pages 134–141, 2003.

[19] A. McCallum and W. Li. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *Proceedings of HLT-NAACL*, pages 188–191, 2003.

[20] T. Cohn and P. Blunsom. Semantic role labelling with tree conditional random fields. In *Ninth Conference on Computational Natural Language (CoNLL)*, 2005.

[21] S. Abney. Partial parsing via finite-state cascades. *Natural Language Engineering*, 23(4):337–344, 1997.

[22] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of English: the penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[23] Y. Le Cun, L. Bottou, Y. Bengio, and P. Haffner. Gradient based learning applied to document recognition. *Proceedings of IEEE*, 86(11):2278–2324, 1998.

[24] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory*, 13:260–269, 1967.

[25] L. Bottou. Stochastic gradient learning in neural networks. In *Proceedings of Neuro-Nîmes 91*, Nimes, France, 1991. EC2.

[26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by back-propagating errors. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. MIT Press, 1986.

[27] D. C. Plaut and G. E. Hinton. Learning sets of filters using back-propagation. *Computer Speech and Language*, 2:35–61, 1987.