Deep Learning for Efficient Discriminative Parsing

Ronan Collobert[†]
IDIAP Research Institute
Martigny, Switzerland
ronan@collobert.com

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 which leverage word representations from Collobert and Weston (2008), 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 (Magerman, 1995; Collins, 1996), improvements have been achieved over the years, but *generative* PCFGs parsers of the last decade from Collins (1999) and Charniak (2000) 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 (Ratnaparkhi, 1999; Henderson, 2004) did not reach standard PCFG-based generative parsers. Henderson (2004) outperforms Collins parser only by using a generative model and performing re-ranking. Charniak and Johnson (2005) also successfully leveraged re-ranking. Pure discriminative parsers from Taskar et al. (2004) and Turian and Melamed (2006) 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 parsers (Finkel et al., 2008; Petrov and Klein, 2008) are based on Conditional Random Fields (CRFs) with PCFG-like features. In the same spirit, Carreras et al. (2008) 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 task-specific 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.

There are two main achievements in this paper. (1) We trade the reduction of features for a "deeper" architecture, a.k.a. a particular deep neural network, which takes advantage of word representations from Collobert and Weston (2008) trained on a large unlabeled corpus. (2) We show the task of parsing can be *efficiently* implemented by seeing it as a recursive tagging task. We convert parse trees into a stack of levels, and then train a single neural network 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 (1997). 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 cho-

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

[1]Which leverage head words of parsing constituents.

Appearing in Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) 2011, Fort Lauderdale, FL, USA. Volume 15 of JMLR: W&CP 15. Copyright 2011 by the authors.

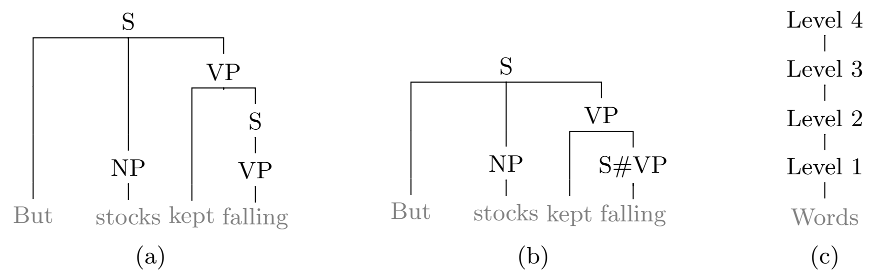

Figure 1: Parse Tree representations. As in Penn Treebank (a), and after concatenating nodes spanning same words (b). In (c) we show our definition of "levels".

sen beforehand.

We acknowledge that training a neural network is a task which requires some experience, which differs from the experience required for choosing good parsing features in more classical approaches. 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 (Lang and Hinton, 1988). They have also been applied to NLP (Bengio et al., 2001; Collobert and Weston, 2008; Collobert et al., 2011), 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) (Bottou et al., 1997). Again, this is not a surprising architecture: GTNs are for deep models what CRFs are for linear models (Lafferty et al., 2001), and CRFs had great success in NLP (Sha and Pereira, 2003; McCallum and Li, 2003; Cohn and Blunsom, 2005). We show how GTNs can be adapted to parsing, by simply constraining the inference graph at each parsing level prediction.

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 (Marcus et al., 1993), 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 Magerman, 1995) (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 1c and Figure 2, one can rewrite a parse tree as a stack of tag levels. 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 task, which describes in an other way the lowest parse tree nodes, has a very good performance record (Sha and Pereira, 2003).

[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.

2.2 From Tagging Levels To Parse Trees

Even if it had success with partial parsing (Abney, 1997), 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[4] 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 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 by Bengio et al. (2001) for language modeling, and reintroduced later by Collobert and Weston (2008) 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 (Bottou et al., 1997; Le Cun et al., 1998). In the NLP field, this type of architecture has been used with success by Collobert et al. (2011) for "flat" tagging tasks. 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. More details on the

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

[4] With the IOBES tagging scheme, each label (e.g. VP) is expanded into 4 different tags (e.g. B-VP, I-VP, E-VP, S-VP), as described in Section 2.1.

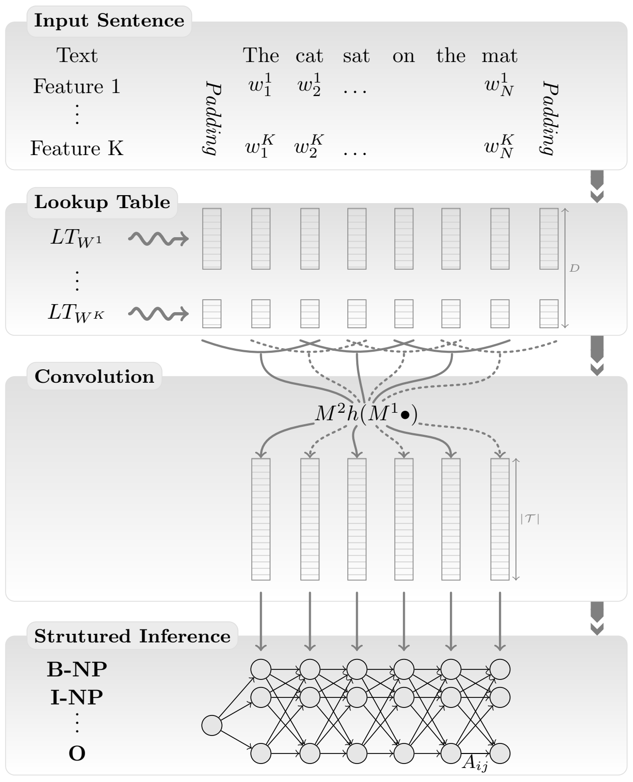

Figure 3: Our neural network architecture. Words and other desired discrete features (caps, tree history, ...) are given as input. The lookup-tables embed each feature in a vector space, for each word. This is fed in a convolutional network which outputs a score for each tag and each word. Finally, a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

derivations are provided in the supplementary material attached to this paper. 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.

3.1 Word Embeddings

We consider a fixed-sized word dictionary[5] $\mathcal{W}$. 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:

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

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. Parameters W are thus initialized randomly and trained as any other neural network layer. However, we show in the experiments that one can obtain a significant performance boost by initializing[6] these embeddings with the word representations found by Collobert and Weston (2008). These representations have been trained on a large unlabeled corpus (Wikipedia), using a language modeling task. They contain useful syntactic and semantic information, which appears to be useful for parsing. This corroborates improvements obtained in the same way by Collobert & Weston on various NLP tagging tasks.

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:

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

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

3.2 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

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

[6]Only the initialization differs. The parameters are trained in any case.

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 in our experiments a (fast) "hard" version of the hyperbolic tangent:

$$h(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases} . \tag{4}$$

3.3 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 Collobert and Weston (2008), 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 indicate 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...) $LT_{W^{\text{words}}}(w_m)$ are first combined together by applying a mapping M^0. We then extract a fixed-size "global" feature vector[7] $\boldsymbol{x}_n$ by performing a max over the sentence:

$$[\boldsymbol{x}_n]_i = \max_{1\leq m\leq N} \left[M^0\, LT_{W^{\text{words}}}(w_m)\right]_i \qquad \forall i \tag{5}$$

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.

3.4 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 tag $t \in \mathcal{T}$ 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 $G_{u(n+1)}$, 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:

$$S([w]_1^N, [t]_1^N, \boldsymbol{\theta}) = \sum_{n=1}^{N} \left(A_{t_{n-1}t_n} + s(\boldsymbol{x}_n)_{t_n}\right)\,, \tag{6}$$

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:

$$[t^\star]_1^N = \underset{[t]_1^N\in\mathcal{T}^N}{\operatorname{argmax}} S([w]_1^N, [t]_1^N, \boldsymbol{\theta})\,. \tag{7}$$

The Viterbi (1967) algorithm 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.

3.5 Training Likelihood

Following the GTN's training method introduced in (Bottou et al., 1997; Le Cun et al., 1998), 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 (6) 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:

$$\begin{aligned} \log p([t]_1^N \,|\, [w]_1^N, \boldsymbol{\theta}) = {} & S([w]_1^N,\, [t]_1^N,\, \boldsymbol{\theta}) \\ & - \underset{\forall [u]_1^N\in\mathcal{T}^N}{\operatorname{logadd}}\; S([w]_1^N,\, [u]_1^N,\, \boldsymbol{\theta})\,, \end{aligned} \tag{8}$$

where we adopt the notation $\operatorname{logadd}_i z_i = \log(\sum_i e^{z_i})$. This likelihood is the same as the one found in Conditional Random Fields (CRFs) (Lafferty et al., 2001) 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 (8) 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 fol-

[7] Here, the concatenation of lookup-tables outputs $LT_{W^{\text{words}}}$ includes relative position embeddings with respect to word n. Because of this notation shortcut, the right-hand side of (5) depends on n implicitly.

lowing classical recursion over n:

$$\begin{aligned}\delta_n(v) &\triangleq \underset{\{[u]_1^n \,\cap\, u_n = v\}}{\operatorname{logadd}} S([w]_1^n,\, [u]_1^n,\, \boldsymbol{\theta}) \quad \forall v \in \mathcal{T} \\ &= \boldsymbol{s}(\boldsymbol{x}_n)_v + \underset{t}{\operatorname{logadd}}\, (\delta_{n-1}(t) + A_{tv})\ ,\end{aligned} \tag{9}$$

followed by the termination

$$\underset{\forall [u]_1^N}{\operatorname{logadd}} S([w]_1^N,\, [u]_1^N,\, \tilde{\boldsymbol{\theta}}) = \underset{u}{\operatorname{logadd}}\ \delta_N(u)\,.$$

As a comparison, the Viterbi algorithm used to perform the inference (7) 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.

3.6 Stochastic Gradient

We maximize the log-likelihood (8) using stochastic gradient ascent, which has the main advantage to be extremely scalable (Bottou, 1991). Random training sentences $[w]_1^N$ and their associated tag labeling $[t]_1^N$ are iteratively selected. The following gradient step is then performed:

$$\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} + \lambda \frac{\partial \log p([t]_1^N \mid [w]_1^N,\, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\ , \tag{10}$$

where λ is a chosen learning rate. The gradient in (10) is efficiently computed via a classical backpropagation (Rumelhart et al., 1986): the differentiation chain rule is applied to the recursion (9), and then to all network layers (3), including the word embedding layers (1). Derivations are simple (but fastidious) algebra which can be found in the supplementary material 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 chunks. This is our history[8] $\mathcal{C}$. 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.

[8]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.

We now implement Constraint 1 by constraining the inference graph introduced in Section 3.4 using the chunk history $\mathcal{C}$. For each chunk $\boldsymbol{c} \in \mathcal{C}$, we adapt the graph output by our network in Figure 3 such that any new candidate chunk $\tilde{\boldsymbol{c}}$ overlapping $\boldsymbol{c}$ includes $\boldsymbol{c}$, and is *strictly* larger than $\boldsymbol{c}$. Because the chunk history $\mathcal{C}$ includes the *largest* chunks up to the last predicted tree level, the new candidate chunk $\tilde{\boldsymbol{c}}$ will be strictly larger than *any* chunk predicted in previous tree levels. Constraint 1 is then always satisfied.

Constraining the inference graph can be achieved by noticing that the condition "$\tilde{\boldsymbol{c}}$ strictly includes $\boldsymbol{c}$" is equivalent to say that the new chunk $\tilde{\boldsymbol{c}}$ satisfies one of the following conditions:

- Starts at the same position but ends after $\boldsymbol{c}$
- Starts before $\boldsymbol{c}$, and ends at the same position
- Starts before and ends after $\boldsymbol{c}$.

Using a IOBES tagging scheme, we implement (see Figure 4) these three conditions by allowing only three corresponding possible paths $\tilde{\boldsymbol{c}}$ in the inference graph, for each candidate label (e.g. *VP*):

- The first tag of $\tilde{\boldsymbol{c}}$ is B-*VP*, and remaining tags overlapping with $\boldsymbol{c}$ are maintained at I-*VP*
- The last tag of $\tilde{\boldsymbol{c}}$ is E-*VP*, and previous tags overlapping with $\boldsymbol{c}$ are maintained at I-*VP*
- The path $\tilde{\boldsymbol{c}}$ is maintained on I-*VP* while overlapping $\boldsymbol{c}$.

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 Experiments

We conducted our experiments on the standard English Penn Treebank benchmark (Marcus et al., 1993). 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

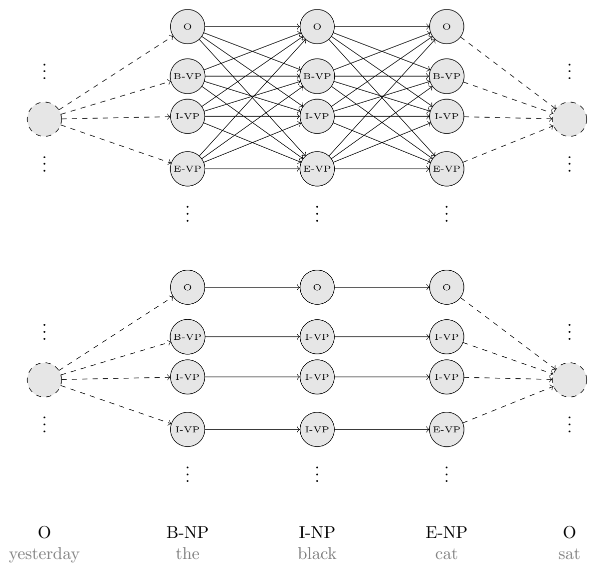

Figure 4: Implementing tree constraints: the chunk history (bottom) contains the NP "the black cat". The top inference graph is unconstrained (as it would be if no chunk were found in the history) and given only for comparison. The bottom graph is constrained such that new overlapping chunks strictly include the existing chunk "the black cat".

on the test set in terms of recall (R), precision (P) and F_1 score. Scores were obtained using the EVALB implementation[9].

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[10] (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

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

[10]Obtained with our own 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)	88.2	89.2	88.7
Taskar et al. (2004)	89.1	89.1	89.1
Turian and Melamed (2006)	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

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 $K = 5$. The word dictionary size was $100,000$. We used the word embeddings obtained from the language model (LM) (Collobert and Weston, 2008) to initialize the word lookup-table. Finally, we fixed the learning rate $\lambda = 0.01$ during the stochastic gradient procedure (10). 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 (Plaut and Hinton, 1987).

5.1 Small Scale Experiments

First discriminative parse trees were very computationally expensive to train. Taskar et al. (2004) proposed a comparison setup for discriminative parsers limited to Penn Treebank sentences with ≤ 15 words. Turian and Melamed (2006) 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 (1999) parser, 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 by Collobert and Weston (2008). It is known that POS is an fundamental feature for all existing parsers. The LM is crucial for the performance of the architecture, as most of the capacity of the network lies into the word lookup-table ($100,000$ words $\times$ dimension 50). Without the LM, rare words cannot be properly trained.[11] Initializing with the LM but not using POS or using POS but not LM gave similar improvements in performance. Combining LM and POS compares well with other parsers.

5.2 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" (leveraging long-range dependencies) provides a clear boost and compares well against Collins (1999) parser[12], 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 was an order of magnitude faster than other available parsers[13].

6 Conclusion

We proposed a new fast and scalable purely discriminative parsing algorithm based on Graph Transformer Networks. With only few basic text features (thanks to word representations from Collobert and Weston (2008)), 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.

Acknowledgments

The author would like to thank NEC Laboratories America for its support, as well as Léon Bottou and Vincent Etter for their comments.

[11]About 15% of the most common words appear 90% of the time, so many words are rare.

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

[13]Available at `http://ml.nec-labs.com/senna`.

References

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

Y. Bengio, R. Ducharme, and P. Vincent. A neural probabilistic language model. In *NIPS 13*, 2001.

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

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.

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.

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.

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.

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

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

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

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

R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *JMLR*, (to appear), 2011.

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.

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

J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting

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)	84.6	84.9	84.8				
Collins (1996)	85.8	86.3	86.1	85.3	85.7	85.5	
Collins (1999)	88.5	88.7	88.6	88.1	88.3	88.2	2640
Charniak (2000)	90.1	90.1	90.1	89.6	89.5	89.6	1020
Charniak and Johnson (2005)			92.0			91.4	
Finkel et al. (2008)	88.8	89.2	89.0	87.8	88.2	88.0	
Petrov and Klein (2008)			90.0			89.4	
Carreras et al. (2008)				90.7	91.4	91.1	
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

and labeling sequence data. In *Eighteenth International Conference on Machine Learning, ICML*, 2001.

K.J. Lang and G.E. Hinton. The development of the time-delay neural network architecture for speech recognition. Technical Report CMU-CS-88-152, Carnegie Mellon University, 1988.

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.

Y. LeCun. A learning scheme for asymmetric threshold networks. In *Proceedings of Cognitiva 85*, pages 599–604, Paris, France, 1985.

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

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.

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.

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

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

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

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.

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

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

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.

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

A Neural Network Gradients

We consider a neural network $f_{\boldsymbol{\theta}}(\cdot)$, with parameters $\boldsymbol{\theta}$. We maximize the likelihood (8), with respect to the parameters $\boldsymbol{\theta}$, using stochastic gradient. By negating the likelihood, we now assume it corresponds to minimize a cost $C(f_{\boldsymbol{\theta}}(\cdot))$, with respect to $\boldsymbol{\theta}$.

Following the classical "back-propagation" derivations (LeCun, 1985; Rumelhart et al., 1986) and the modular approach shown in (Bottou, 1991), any feed-forward neural network with L layers, like the one shown in Figure 3, can be seen as a composition of functions $f^l_{\boldsymbol{\theta}}(\cdot)$, corresponding to each layer l:

$$f_{\boldsymbol{\theta}}(\cdot) = f^L_{\boldsymbol{\theta}}(f^{L-1}_{\boldsymbol{\theta}}(\dots f^1_{\boldsymbol{\theta}}(\cdot)\dots))$$

Partionning the parameters of the network with respect to each layers $1 \leq l \leq L$, we write:

$$\boldsymbol{\theta} = (\boldsymbol{\theta}^1,\, \dots,\, \boldsymbol{\theta}^l,\, \dots,\, \boldsymbol{\theta}^L)\,.$$

We are now interested in computing the gradients of the cost with respect to each $\boldsymbol{\theta}^l$. Applying the chain rule (generalized to vectors) we obtain the classical backpropagation recursion:

$$\frac{\partial C}{\partial \boldsymbol{\theta}^l} = \frac{\partial f^l_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}^l}\frac{\partial C}{\partial f^l_{\boldsymbol{\theta}}} \qquad (11)$$

$$\frac{\partial C}{\partial f^{l-1}_{\boldsymbol{\theta}}} = \frac{\partial f^l_{\boldsymbol{\theta}}}{\partial f^{l-1}_{\boldsymbol{\theta}}}\frac{\partial C}{\partial f^l_{\boldsymbol{\theta}}}\,. \qquad (12)$$

In other words, we first initialize the recursion by computing the gradient of the cost with respect to the last layer output $\partial C/\partial f^L_{\boldsymbol{\theta}}$. Then each layer l computes the gradient respect to its own parameters with (11), given the gradient coming from its output $\partial C/\partial f^l_{\boldsymbol{\theta}}$. To perform the backpropagation, it also computes the gradient with respect to its own inputs, as shown in (12). We now derive the gradients for each layer we used in this paper. For simplifying the notation, we denote $\langle A\rangle_i$ the i^{th} column vector of matrix A.

Lookup Table Layer Given a matrix of parameters $\boldsymbol{\theta}^1 = W^1$ and word (or discrete feature) indices $[w]_1^T$, the layer outputs the matrix:

$$f^l_{\boldsymbol{\theta}}([w]_l^T) = \begin{pmatrix} W^1_{w_1} & W^1_{w_2} & \dots & W^1_{w_T}\end{pmatrix}\,.$$

The gradients of the weights W_i are given by:

$$\frac{\partial C}{\partial W^1_i} = \sum_{\{1\leq t\leq T\,/\,w_t=i\}} \langle\frac{\partial C}{\partial f^l_{\boldsymbol{\theta}}}\rangle_i$$

This sum equals zero if the index i in the lookup table does not corresponds to a word in the sequence. In this case, the i^{th} column of W does not need to be updated. As a Lookup Table Layer is always the first layer, we do not need to compute its gradients with respect to the inputs.

Linear Layer Our convolutional architecture performs a series of "Linear" operations, as described in (3). Given parameters $\boldsymbol{\theta}^l = (W^l, \boldsymbol{b}^l)$, and an input *vector* $f^{l-1}_{\boldsymbol{\theta}}$ the output is given by:

$$f^l_{\boldsymbol{\theta}} = W^l f^{l-1}_{\boldsymbol{\theta}} + \boldsymbol{b}^l\,. \qquad (13)$$

The gradients with respect to the parameters are then obtained with:

$$\frac{\partial C}{\partial W^l} = \left[\frac{\partial C}{\partial f^l_{\boldsymbol{\theta}}}\right]\left[f^{l-1}_{\boldsymbol{\theta}}\right]^{\mathrm{T}} \quad\text{and}\quad \frac{\partial C}{\partial \boldsymbol{b}^l} = \frac{\partial C}{\partial f^l_{\boldsymbol{\theta}}}\,, \qquad (14)$$

and the gradients with respect to the inputs are computed with:

$$\frac{\partial C}{\partial f^{l-1}_{\boldsymbol{\theta}}} = \left[W^l\right]^{\mathrm{T}}\frac{\partial C}{\partial f^l_{\boldsymbol{\theta}}}\,. \qquad (15)$$

Max Layer Given a *matrix* $f^{l-1}_{\boldsymbol{\theta}}$, the Max Layer computes

$$\left[f^l_{\boldsymbol{\theta}}\right]_i = \max_m \left[\langle f^{l-1}_{\boldsymbol{\theta}}\rangle_m\right]_i \text{ and } a_i = \operatorname*{argmax}_t \left[\langle f^{l-1}_{\boldsymbol{\theta}}\rangle_m\right]_i \;\forall i\,,$$

where a_i stores the index of the largest value. We only need to compute the gradient with respect to the inputs, as this layer has no parameters. The gradient is given by

$$\left[\langle\frac{\partial C}{\partial f^{l-1}_{\boldsymbol{\theta}}}\rangle_m\right]_i = \begin{cases} \left[\langle\frac{\partial C}{\partial f^l_{\boldsymbol{\theta}}}\rangle_m\right]_i & \text{if } m = a_i \\ 0 & \text{otherwise}\end{cases}\,.$$

HardTanh Layer Given a *vector* $f^{l-1}_{\boldsymbol{\theta}}$, and the definition of the HardTanh (4) we get

$$\left[\frac{\partial C}{\partial f^{l-1}_{\boldsymbol{\theta}}}\right]_i = \begin{cases} 0 & \text{if } \left[f^{l-1}_{\boldsymbol{\theta}}\right]_i < -1 \\ \left[\frac{\partial C}{\partial f^l_{\boldsymbol{\theta}}}\right]_i & \text{if } -1 <= \left[f^{l-1}_{\boldsymbol{\theta}}\right]_i <= 1 \\ 0 & \text{if } \left[f^{l-1}_{\boldsymbol{\theta}}\right]_i > 1\end{cases}\,,$$

if we ignore non-differentiability points.

Sentence-Level Log-Likelihood The network outputs a matrix where each element $[f_{\boldsymbol{\theta}}]_{t,\,n}$ gives a score for tag t at word n. Given a tag sequence $[t]_1^N$ and a input sequence $[\boldsymbol{x}]_1^N$, we maximize the likelihood (8), which corresponds to minimizing the score

$$C(f_{\boldsymbol{\theta}}, A) = \underbrace{\operatorname*{logadd}_{\forall [u]_1^N} s([\boldsymbol{x}]_1^N,\,[u]_1^N,\,\tilde{\boldsymbol{\theta}})}_{C_{logadd}} - s([\boldsymbol{x}]_1^N,\,[t]_1^N,\,\tilde{\boldsymbol{\theta}})\,,$$

with

$$s([\boldsymbol{x}]_1^N,\,[t]_1^N,\,\tilde{\boldsymbol{\theta}}) = \sum_{n=1}^N \left([A]_{[t]_{n-1},\,[t]_n} + [f_{\boldsymbol{\theta}}]_{[t]_n,\,n}\right)\,.$$

We first initialize all gradients to zero

$$\frac{\partial C}{\partial \left[f_{\boldsymbol{\theta}}\right]_{t,\,n}} = 0 \;\forall t, n \;\text{ and }\; \frac{\partial C}{\partial \left[A\right]_{t,\,u}} = 0 \quad \forall t, u\,.$$

We then *accumulate* gradients over the second part of the cost $-s([\boldsymbol{x}]_1^N,\,[t]_1^N,\,\tilde{\boldsymbol{\theta}})$, which gives:

$$\left.\begin{array}{l} \dfrac{\partial C}{\partial \left[f_{\boldsymbol{\theta}}\right]_{[t]_n,\,n}} \mathrel{+}= 1 \\ \dfrac{\partial C}{\partial \left[A\right]_{[t]_{n-1},\,[t]_n}} \mathrel{+}= 1 \end{array}\right. \qquad \forall n\,.$$

We now need to accumulate the gradients over the first part of the cost, that is C_{logadd}. We differentiate C_{logadd} by applying the chain rule through the recursion (9). First we initialize our recursion with

$$\frac{\partial C_{logadd}}{\partial \delta_N(t)} = \frac{e^{\delta_N(t)}}{\sum_u e^{\delta_N(u)}} \quad \forall t\,.$$

We then compute iteratively:

$$\frac{\partial C_{logadd}}{\partial \delta_{n-1}(t)} = \sum_u \frac{\partial C_{logadd}}{\partial \delta_n(u)} \frac{e^{\delta_{n-1}(t)+[A]_{t,\,u}}}{\sum_v e^{\delta_{n-1}(v)+[A]_{v,\,u}}}\,,$$

where at each step n of the recursion we accumulate of the gradients with respect to the inputs $f_{\boldsymbol{\theta}}$, and the transition scores $[A]_{t,\,u}$:

$$\frac{\partial C}{\partial \left[f_{\boldsymbol{\theta}}\right]_{t,\,n}} \mathrel{+}= \frac{\partial C_{logadd}}{\partial \delta_n(t)} \, \frac{\partial \delta_n(t)}{\partial \left[f_{\boldsymbol{\theta}}\right]_{t,\,n}} = \frac{\partial C_{logadd}}{\partial \delta_n(t)}\,,$$

and

$$\begin{aligned} \frac{\partial C}{\partial \left[A\right]_{t,\,u}} \mathrel{+}=& \frac{\partial C_{logadd}}{\partial \delta_n(u)} \, \frac{\partial \delta_n(u)}{\partial \left[A\right]_{t,\,u}} \\ =& \frac{\partial C_{logadd}}{\partial \delta_n(u)} \, \frac{e^{\delta_{n-1}(t)+[A]_{t,\,u}}}{\sum_v e^{\delta_{n-1}(v)+[A]_{v,\,u}}}\,. \end{aligned}$$

Initialization and Learning Rate We employed only two classical "tricks" for training our neural networks: the initialization and update of the parameters of each network layer were done according to the "fan-in" of the layer, that is the number of inputs used to compute each output of this layer (Plaut and Hinton, 1987). The fan-in for the lookup table (1) and the l^{th} linear layer (3) (also rewritten in (13)) are respectively 1 and $|f_{\boldsymbol{\theta}}^{l-1}|$ (where $|z|$ is the number of dimensions of vector z). The initial parameters of the network were drawn from a centered uniform distribution, with a variance equal to the inverse of the square-root of the fan-in. The learning rate in (10) was divided by the fan-in, but stayed fixed during the training.