Syntactic realization with data-driven neural tree grammars

Abstract

A key component in surface realization in natural language generation is to choose concrete syntactic relationships to express a target meaning. We develop a new method for syntactic choice based on learning a stochastic tree grammar in a neural architecture. This framework can exploit state-of-the-art methods for modeling word sequences and generalizing across vocabulary. Our methods also embed elementary tree structures using convolutional coding to generalize over tree structures. We evaluate the models on the task of linearizing unannotated dependency trees, achieving improvements over previous methods.

1 Introduction

Where natural language understanding systems face problems of ambiguity, natural language generation (NLG) systems face problems of choice. A wide coverage NLG system must be able to formulate messages using specialized linguistic elements in the exceptional circumstances where they are appropriate; however, it can only achieve fluency by expressing frequent meanings in routine ways. Empirical methods have thus long been recognized as crucial to NLG; see e.g. Langkilde and Knight (1998).

With traditional stochastic modeling techniques, NLG researchers have had to predict choices using factored models with handcrafted representations and strong independence assumptions, in order to avoid combinatorial explosions and address the sparsity of training data. By contrast, in this paper, we leverage recent advances in deep learning to develop new models for syntactic choice that free engineers from many of these decisions, but still generalize more effectively and match human choices more closely than traditional techniques.

We adopt the characterization of syntactic choice from Bangalore and Rambow (2000) which uses a stochastic tree model and a language model to produce a linearized string from an unordered, unlabeled dependency graph. The first step to producing a linearized string is to assign each node an appropriate supertag—a fragment of a parse tree with a leaf left open for a lexical item—using a stochastic tree model. The resulting assignments are consistent with many possible derivation trees because of adjunct ambiguitity. To finish the linearization, a language model is used to select among them. While improving the language model would improve the linearized string, this work focuses on more accurately predicting the correct supertags from unlabeled dependency trees.

In this work, we make two novel contributions to improve upon the stochastic tree model: a supertag embedding technique which uses convolutional neural networks and a recurrent tree network which can model hierarchical relationships. By embedding supertags and using a recurrent tree network for the relationships between them, we enable more information to be used at each node. We evaluate our contributions in two ways: first, by varying the technique used to embed supertags, and then by comparing a feed-forward model against our recurrent tree model.

Our presentation begins in § 2 with an introduction to tree grammars and a deterministic metholodogy for inducing the elementary trees of the grammar. Next, § 3 presents the techniques we have developed to represent a tree grammar using a neural architecture. Then, in § 4, we describe the specific model we have implemented, and in **??**, we describe the algorithms used to describe the classification results. The experiments in § 5 demonstrate the improvement of the model over baseline results based on previous

work on stochastic surface realization. We conclude with a brief discussion of the future potential for neural architectures to predict grammatical choices in NLG.

2 Tree Grammars

Broadly, tree grammars are a family of tree rewriting formalisms that modify structure to produce strings. The atomic syntactic units are called elementary trees; elementary trees combine using tree-rewrite rules to form derived phrase structure trees describing complex sentences. Inducing a tree grammar involves fixing a formal inventory of structures and operations for elementary trees and then inferring instances of those structures to match corpus data.

2.1 The Grammar Formalism

The canonical tree grammar is perhaps lexicalized tree-adjoining grammar (LTAG) (Joshi and Schabes, 1991). The elementary trees of LTAG consist of two disjoint sets with distinct operations The elementary trees of LTAG are disjoint sets based on the operations they can perform: initial trees can perform substitution operations and auxiliary trees can perform adjunction operations. The substitution operation replaces a non-terminal leaf of a target tree with an identically-labeled root node of an initial tree. The adjunction operation modifies the internal structure of a target tree at by expanding a node identically-labeled with the root and a distinguished foot note in the auxiliary tree. The lexicalization of the the grammar requires each elementary tree to have at least one lexical item as a leaf.

LTAG incurs computational costs because it is mildly context-sensitive in generative power. Several variants reduce the complexity of the formalism by limiting the range of adjunction operations. For example, the Tree Insertion Grammar allows for adjunction as long as it is either a left or right auxiliary tree (Schabes and Waters, 1995). Tree Substitution Grammars, meanwhile, allow for no adjunction and only substitutions (Cohn et al., 2009).

We adopt one particular restricted adjunction, called sister-adjunction or *insertion*, which allows trees to attach to an interior node and add itself as a first or last child (Chiang, 2000). Chiang's sister-adjunction allows for the flat structures in the Penn Treebank while limiting the formalism to context-free power.

2.2 Inducing a tree grammar

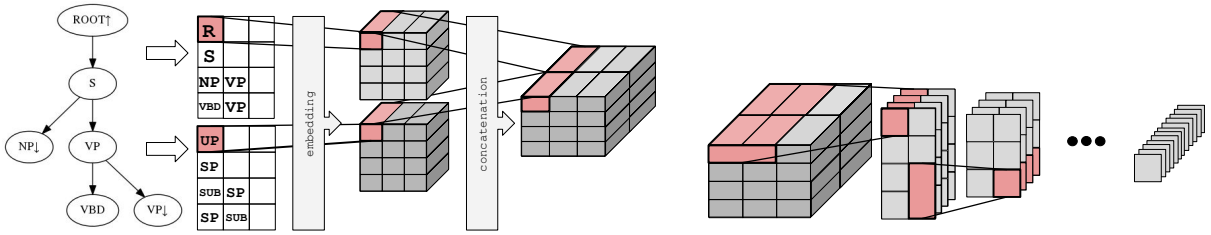
In lexicalized tree grammars, the lexicon and the grammatical rules are one and the same. The set of possible grammatical moves which can be made are simultaneously the set of possible words which can be used next. This means inducing a tree grammar from a dataset is a matter of inferring the set of constructions in the data.

We follow previous work in using bracketed phrase structure corpora and deterministic rules to induce the grammar (Chiang, 2000; Bangalore et al., 2001). Broadly, the methodology is to split the observed trees into the constituents which make it up, according to the grammar formalism. We use head rules (Chiang, 2000; Collins, 1997; Magerman, 1995) to associate internal nodes in a bracketed tree with the lexical item that owns it. We use additional rules to classify some children as complements, corresponding to substitution sites and root notes of complement trees; and other children as adjuncts, corresponding to insertion trees that combine with the parent node, either to the right or to the left of the head; this allows us to segment the tree into units of substitution and insertion.[1]

3 Neural Representations

This work makes two contributions to improve stochastic tree modeling with neural networks. First, supertags are represented as vectors through convolutional coding, opening up the recent repertoire of vector space techniques. Second, hierarchical dependence between nodes are modeled using a recurrent tree network.

[1] One particular downside of deterministically constructing the grammar this way is that it can produce an excess of superfluous elementary trees. We minimize this by collapsing repeated projections in the treebank. Other work has provided Bayesian models for reducing grammar complexity by forcing it to follow Dirichlet or Pitman-Yor processes (Cohn et al., 2010)—an interesting direction for future work.



(A) Tree Node Embedding (B) Convolutional Filter Encoding

Figure 1: Embedding supertags using convolutional neural networks. In (A), a tree is encoded by its features and then embedded. In (B), convolutional layers are used to encode the supertag into a vector.

3.1 Embedding Supertags

To embed a supertag, each of its nodes are embedded, grouped to form a tensor, and then summarized into a single vector using a series of convolutional neural networks. Since the weights of a convolutional neural network are optimized during training, the data should determine which parts of the supertag are important. The more usual alternative approach is to treat each supertag as the atomic unit—essentially a statistical singleton—and map it either to a feature or an embedding vector. We evaluate our contribution and compare these two approaches to get a sense for whether convolutional coding robustly generalizes across supertags.

The goal of embedding supertags is to summarize the tree structure into a single vector. Inside the tree structure, there are two separate pieces of information which need to be summarized: the syntactic category of each node and the role each node plays in the tree grammar. The root receives a special role—either substitution or insertion to represent the supertag's tree operation. The remaining nodes either have the substitution point role or the spine role—they are along the spine from root to the lexical attachment point.

The nodes of a supertag can be independently embedded and combined to form a tensor of embeddings. Specifically, symbols representing the syntactic category and node roles are treated as distinct vocabulary tokens, mapped to integers, and used to retrieve a vector representation that is learned during training. The vectors are grouped into a tensor by placing the root node into the first cell of the first row and left-aligning the decendents in the subsequent rows. The two tensors are combined by concatenating along the embedding dimension. This embed-and-group method is shown in on the left in Figure 1.

Using a series of convolutional neural networks which learn their weights during training, the tensor of embeddings can be reduced to a single vector. To reduce the tensor to a vector, the convolutions are designed with increasingly larger filter sizes. Additionally, the dimensions are reduced alternatingly to also facilitate the capture of features. The entire process is summarized in Eq. 1 with Λ representing the supertags, G representing embedding matrices, and C representing the convolutional neural network layers. Specifically, G_s is the syntactic category embedding matrix and G_r is the node role embedding matrix. Each convolutional layer C is shown with its corresponding height and width as $C^{i,j}$. The encoding first constructs the tensor, T_Λ, through the embed-and-group method. Then, the embedding matrix G_Λ is summarized from T_Λ using the series of convolutional layers.

$$
\begin{aligned}
T_\Lambda &= [G_s(\Lambda_{syntactic\ category});\ G_r(\Lambda_{role})] \\
G_\Lambda &= C^{4,5}(C^{3,1}(C^{1,3}(C^{2,1}(C^{1,2}(T_\Lambda)))))
\end{aligned} \tag{1}
$$

The final product, a vector per supertag, is aggregated with the other vectors and turned into an embedding matrix. This is visualized in on the right Figure 1. During training and test time, supertags are input as indices and their feature representations retrieved as an embedding. Importantly, the convolutional layers are connected to the computational graph during training, so the parameters are optimized with respect to the task.

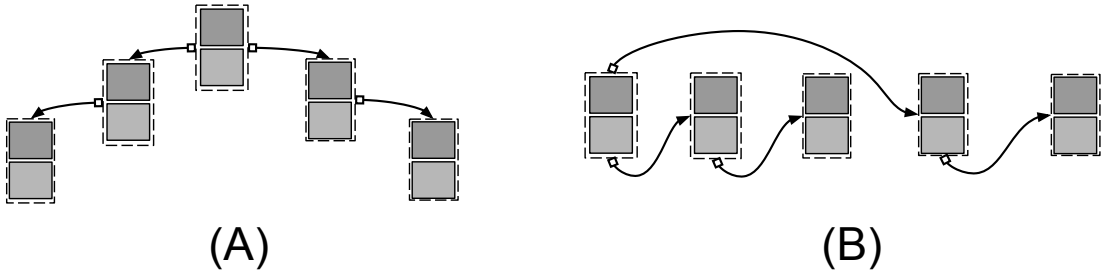


Figure 2: A recurrent tree network. (A) The dependency structure as a tree. (B) the dependency structure as a sequence.

3.2 Recurrent Tree Networks

To model the relationship between the supertags in the derivation tree, we use a recurrent tree network. While recurrent neural networks normally use the previous hidden state in the sequential order of the inputs, recurrent tree networks use the hidden state from the parent. Utilizing the parent's hidden state rather than the sequentially previous hidden state, the recurrent connection can travel down the branches of a tree. An example of a recurrent tree network is shown in 2.

In our recurrent tree network, child nodes gain access to a parent's hidden state through an internal *tree state*. During a tree recurrence, the nodes in the dependency graph are enumerated in a top-down traversal. At each step in the recurrence, the resulting recurrent state is stored in the tree state at the step index. Descendents access the recurrent state using a topological index that is passed in as data.

The formulation is summarized in Equation 2. The input to each time step in the current tree is the data, x_t, and a topological index, p_t. The recurrent tree uses p_t to retrieve the parent's hidden state, s_p, from the tree state, S_{tree}, and applies the recurrence function, $g(\dot{)}$. The resulting recurrent state is the hidden state for child node, s_c. The recurrent state s_c is stored in the tree state, S_{tree}, at index t.

$$\begin{aligned} s_c &= RTN(x_t, p_t) \\ &= g(x_t, S_{tree}[p_t]) \\ &= g(x_t, s_p) \\ S_{tree}[t] &= s_c \end{aligned} \tag{2}$$

The use of topological indices allows for many recurrent tree networks to be run in parallel on a GPU, increasing the efficiency of the implementation. The primary concern for running parallel GPU computations is homogeneity because the same operation will be applied to the entire data structure. Normally, tree operations require flow control, making homogeneity impossible. However, using topological indices and a tree state eliminates the need for flow control by creating locally linear, homogenous computations.

4 Models

In this work, we present two stochastic tree models which represent different modeling decisions. The first, Fergus-N[2], is a reimplementation of Bangalore and Rambow (2000)'s original FERGUS model but implemented in a feed-forward neural network. The second model, Fergus-R[3], utilizes a recurrent tree network. Fergus-N serves as a baseline model to which we compare the recurrent tree network.

For both stochastic tree models, a recurrent neural network language model is used to complete the linearization task. The same language model is to eliminate the confound of language model performance and measure performance differences in the stochastic tree modeling.

[2]Fergus-N is shorthand for Fergus-Neuralized

[3]Fergus-R is shorthand for Fergus-Recurrent

4.1 Model 1: Fergus-N

Fergus-N is a stochastic tree model which uses local parent-child information as inputs to a feed-forward network. Each parent-child pair are treated as independent of all other pairs. The probability of the parent's supertag is computed using an embedding of the pair's lexical material and an embedding of the child's supertag. During training, the negative log probability of the observed parent supertags is minimized for each parent-child pair. This objective function is formally defined in Eq. 3 where tag_p is the parent supertag, tag_c is the child supertag, lex_p is the parent's lexical material, and lex_c is the child's lexical material. Note that the probability of supertags for the leaves of the tree are computed with respect to their parent's lexical material.

$$min_\theta - [\sum_p \sum_{p \rightarrow c} log[P_\theta(tag_p | lex_p, lex_c, tag_c)] + \sum_c log[P_\theta(tag_c | lex_p, lex_c)]] \tag{3}$$

The model is implemented as a feed-forward neural network that computes a probability distribution over parent supertags conditioned on lexical material and the child's supertag. Equation 4 details the model formulation. The lexical material, lex_p and lex_c, are embedded using the word embedding matrix, G_w, concatenated, and mapped to a new vector, ω_{lex}, with a fully connected layer, FC_1. The child supertag, tag_c, is embedded with G_Λ and concatenated the lexical vector, ω_{lex}, forming an intermediate vector representation of the node, ω_{node}. The node vector is repeated for each of the parent's possible supertags, $tagset_p$, and then concatenated with their embeddings to construct the set of treelet vectors, $\Omega_{treelet}$. The vector states for the leaf nodes are similarly constructed, but instead combine the lexical vector, ω_{lex} with the embeddings of the child's possible supertags, $tagset_c$. The final operation induces a probability distribution over the treelet and leaf vectors using a score computed by the vectorized function, $\Psi_{predict}$, as the scalar in a softmax distribution.

$$\begin{aligned}
&\omega_{lex} = FC_1([G_w(lex_p); G_w(lex_c)]) \\
&\omega_{node} = concat([G_\Lambda(tag_c);\ \omega_{lex}]) \\
&\Omega_{treelet} = concat([repeat(\omega_{node}),\ G_\Lambda(tagset_p)]) \\
&\Omega_{leaf} = concat([repeat(\omega_{lex}),\ G_\Lambda(tagset_c)]) \\
&P_\theta(tag_{p,i} | lex_p, lex_c, tag_c) = \frac{exp(\Psi(\omega_{treelet_i}))}{\sum_{j \in |tagset_p|} exp(\Psi(\omega_{treelet_j}))} \\
&P_\theta(tag_{c,i} | lex_p, lex_c) = \frac{exp(\Psi(\omega_{leaf_i}))}{\sum_{j \in |tagset_c|} exp(\Psi(\omega_{leaf_j}))}
\end{aligned} \tag{4}$$

A decoding step uses the probability distributions to compute a high probability assignment for all supertags simueltaneously. There are two primary difficulties that arise in this computation. First, the conditional relationship of parents on children implies that the probability for the root supertag depends on the supertags of the entire tree. Second, while it is easy to maintain local consistency—matching the syntactic category of the child to an appropriate node on the parent—two children may choose substitution supertags which assume attachment to the same position on the parent.

To efficiently decode the supertag classifications, we implement an A* algorithm to incrementally select consistent supertag assignments. At each step, the algorithm uses a priority queue to select subtrees based on their inside-outside scores. The inside score is computed as the sum of the log probabilities of the supertags in the subtree. The outside score is the sum of the best supertag for nodes outside the subtree, similar to Lewis and Steedman (2014). Once selected, the subtree is attached to the possible supertags of its parent that are both locally consistent and consistent among its already attached children. These resulting subtrees are placed into the priority queue and the algorithm iterates to progress the search. The search concludes either when a single complete tree has been found[4].

[4] Although, the data has some noise so that sometimes there is no complete tree that can possibly be formed

4.2 Model 2: Fergus-R

Fergus-R is a stochastic tree model implemented in a top-down recurrent tree network and augmented with soft attention. For each node in the input dependency tree, soft attention—a method which learns a vectorized function to weight a group of vectors and sum into a single vector—is used to summarize its children.

The advantage of using a recurrent tree is that nodes are able to be informed of their ancestors. Each node is further informed using soft attention over its children —a method which learns a vectorized function to weight a group of vectors and sum into a single vector. The soft attention vector and the node's embedded lexical material serve as the input to the recurrent tree. The output of the recurrent tree represents the vectorized state of each node and is combined with each node's possible supertags to form prediction states. Importantly, removing the conditional dependence on descendents' supertags results in the simplified objective function in Eq. 5 where lex_C is the children's lexical information, lex_p is the parent's lexical information, tag_p is the supertag for the parent node, and RTN is the recurrent tree network.

$$min_\theta - [\sum_{(p,C)} P_\theta(tag_p | RTN,\ lex_p,\ lex_C)] \tag{5}$$

The Fergus-R model uses only lexical information as input to calculate the probability distribution over each node's supertags. The specific formulation is detailed in Eq. 6. First, a parent node's children, lex_C, are embedded using the word embedding matrix, G_w, and then summarized with an attention function, Ψ_{attn}, to form the child context vector, ω_C. The child context is concatenated with the embedded lexical information of the parent node, lex_p, and mapped to a new vector space with a fully connected layer, FC_1, to form the lexical context vector, ω_{lex}. The context vector and a topological vector for indexing the internal tree state (see § 3.2) are passed to the recurrent tree network, RTN, to compute the full state vector for the parent node, ω_{node}. Similar to Fergus-N, the state vector is repeated and concatenated with the vectors of the parent node's possible supertags, $tagset_p$, and mapped to a new vector space with a fully connected layer, FC_2. A vector in this vector space is labeled $\omega_{elementary}$ because the combination of supertag and lexical item constitutes an elementary tree. The last step is to compute the probability of each supertag using the vectorized function, $\Psi_{predict}$.

$$\begin{aligned}
&\omega_C = \Psi_{attn}(G_w(lex_C)) \\
&\omega_{lex} = FC_1(concat(\omega_C,\ G_w(lex_p))) \\
&\omega_{node} = RTN(\omega_{lex},\ topology) \\
&\Omega_{elementary} = FC_2(concat(repeat(\omega_{node}),\ G_\Lambda(tagset_p))) \\
&P_\theta(tag_{p,i} \mid RTN,\ lex_p,\ lex_C) = \frac{exp(\Psi_{predict}(\omega_{elementary_i})))}{\sum_{j \in |\Omega|} exp(\Psi_{predict}(\omega_{elementary_j}))}
\end{aligned} \tag{6}$$

Although the same A* algorithm from Fergus-N is used, the decoding for Fergus-R is far simpler. As supertags are incrementally selected in the algorithm, the inside score of the subsequent subtree is computed. Where Fergus-N had to compute a incremental dynamic program to evaluate the inside score, Fergus-R decomposes into a sum of conditionally independent distributions. The resulting setup is a chart parsing problem where the inside score of combining two consistent (non-conflicting) edges is just the sum of their inside scores.

4.3 Linearization

The final step to linearizing the output of Fergus-N and Fergus-R—a dependency tree annotated with supertags and partial attachment information—is a search over possible orderings with a language model. There are many possible orderings due to adjunction operation What remains to be determined is the order in which adjuncts are attached. Following Bangalore and Rambow (2000), a language model is

Model	Test Set Size	Accuracy
Fergus-Penn	100	0.749
Fergus-N Top_{40}	2187	0.699

Figure 3:

used to select between the alternate orderings. The language model used is a two-layer LSTM trained using the Keras library on the surface form of the Penn Treebank. The surface form was minimally cleaned[5] to simulate realistic scenarios.

The difficulty of selecting orderings with a language model is that the possible linearizations can grow exponentially. In particular, our implementations result in a large amount of insertion trees[6]. We approach this problem using a prefix tree which stores the possible linearizations as back-pointers to their last step and the word for the current step. The prefix tree is greedily searched with 32 beams.

5 Experiments

In this work, we make two contributions to stochastic tree modeling and evaluate them on a supertag prediction and linearization task. The evaluation uses the Wall Street Journal sections of the Penn Treebank which have been previously used for statistical tree grammars (Chiang, 2000)[7]. Our data pipeline transforms the treebank into a collection of elementary trees and derivation tree—the structured relationships required to compose them into the observed data (see **??**). Abstracting syntactic information from the derivation tree leads to the unlabeled dependency trees our models assume as input.

We evaluate the convolutional coding of the supertags by comparing it against standard embedding techniques. Specifically, we construct a matrix that has as many rows as supertags and is initialized with a Glorot uniform distribution. Fergus-N and Fergus-R are retrained with the exact same parameters, except that instead of the supertag indices referencing a matrix constructed by a series of convolutions, it references our constructed embedding matrix (which gets optimized during learning). The impact of the embedding will be different for each model—Fergus-N uses embedded supertags in constructing the node state and Fergus-R does not—but any differences between the models and their alternate embedding counterparts can be perceived as evidence for either embedding technique.

To evaluate whether a recurrent tree network can capture hierarchical and long-distance relationships, we compare it against a feed-forward model. Specifically, for each model, we measure how often it predicts the correct supertag or if it predicts a different supertag but with the correct root syntactic category, root node role, or lexical leaf node[8] syntactic category. These four quantities represent different aspects that are important to get right in a stochastic tree model.

We also compare the differences in supertag embeddings and stochastic tree models on the linearization task itself. The linearization tasks licenses more freedom in supertag classifications because it may have the same root and leaf syntactic categories but differ in minor constructions, such as number of arguments. The freedom allows for models to be more generalized and less fit to the specific supertags, but at the same time, it also mutes the distinctions between classification decisions. The metrics used in the linearization task were taken from previous work, who argue that it correlates with generation performance better than n-gram counting methods.

5.1 Results

6 Related Work

7 Conclusion

References

Srinivas Bangalore and Owen Rambow. 2000. Exploiting a probabilistic hierarchical model for generation. In *Proceedings of the 18th conference on Computational linguistics-Volume 1*, pages 42–48. Association for Computational Linguistics.

Srinivas Bangalore, John Chen, and Owen Rambow. 2001. Impact of quality and quantity of corpora on stochastic generation. In *Proceedings of the 2001 Conference on Empirical Methods in Natural Langauge Processing, Pittsburgh, PA*.

David Chiang. 2000. Statistical parsing with an automatically-extracted tree adjoining grammar. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 456–463. Association for Computational Linguistics.

Trevor Cohn, Sharon Goldwater, and Phil Blunsom. 2009. Inducing compact but accurate tree-substitution grammars. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 548–556. Association for Computational Linguistics.

Trevor Cohn, Phil Blunsom, and Sharon Goldwater. 2010. Inducing tree-substitution grammars. *The Journal of Machine Learning Research*, 11:3053–3096.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, pages 16–23. Association for Computational Linguistics.

Aravind K Joshi and Yves Schabes. 1991. Tree-adjoining grammars and lexicalized grammars.

Irene Langkilde and Kevin Knight. 1998. Generation that exploits corpus-based statistical knowledge. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 1*, pages 704–710. Association for Computational Linguistics.

Mike Lewis and Mark Steedman. 2014. Improved ccg parsing with semi-supervised supertagging. *Transactions of the Association for Computational Linguistics*, 2:327–338.

David M Magerman. 1995. Statistical decision-tree models for parsing. In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, pages 276–283. Association for Computational Linguistics.

Yves Schabes and Richard C. Waters. 1995. Tree Insertion Grammar : A Cubic-Time , Parsable Formalism that Lexicalizes Context-Free Grammar without Changing the Trees Produced. *Computational Linguistics*, 21(4):479–513.

Model specifications - layers, learning, etc

Head to supertag mapper

The data used in our experiments was taken from the Wall Street Jounral sections of the Treebank corpus.

For the unlabeled dependency trees, we used the derivation structures which resulted from the deterministic grammar induction (see **??**).

For the Fergus-N and Fergus-R learning problems, we processed the data into their relevant tuples.

Supertags were preprocessed into two data structures: one for the convolutional coding and one to map lexical items to supertags. The supertag data structure used for convolutional coding is a three-dimensional tensor where the first dimension is the supertag index, the second dimension is depth from

[5] With respect to the surface form, the only cleaning operations were to merge proper noun phrases into single tokens. Punctuation and other common cleaning operations were not performed.

[6] Many of the validation examples had more than 2^{40} possible linearizations.

[7] A possible additional data source, the data from the 2011 Shared Task on Surface Realization, was not available

[8] The leaf to which a lexical item would attach

root, and the third is relative sibling index. The tokens for syntactic category and node role are mapped to indices, stored in the data structure, and the data structure is stored in GPU memory.

During

in model intro: - learning parameters in appendix - used keras + theano

in conv coding maybe: - gpu data structure

in experiments - where the data comes from