Syntactic realization with data-driven neural tree grammars

Brian McMahan and **Matthew Stone**
Computer Science, Rutgers University
brian.mcmahan, matthew.stone@rutgers.edu

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. We also induce embeddings to generalize over elementary tree structures and exploit a tree recurrence over the input structure to model long-distance influences between NLG choices. We evaluate the models on the task of linearizing unannotated dependency trees, documenting the contribution of our modeling techniques to improvements in both accuracy and run time.

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, match human choices more closely, and enable more efficient computations than traditional techniques.

We adopt the characterization of syntactic choice from Bangalore and Rambow (2000): the problem is to use 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 item an appropriate *supertag*—a fragment of a parse tree with a leaf left open for the lexical item. This process involves applying a learned model to make predictions for the syntax of each item and then searching over the predictions to find a consistent assignment for the entire sentence. The resulting assignments allow for many possible surface realization outputs because they can underdetermine the order and attachment of adjuncts. To finish the linearization, a language model is used to select the most likely surface form from among the alternatives. While improving the language model would improve the linearized string, we focus here on more accurately predicting the correct supertags from unlabeled dependency trees.

Our work exploits deep learning to improve the model of supertag assignment in two ways. First, we analyze the use of embedding techniques to generalize across supertags. Neural networks offer a number of architectures that can cluster tree fragments during training; such models learn to treat related structures similarly, and we show that they improve supertag assignments. Second, we analyze the use of tree recurrences to track hierarchical relationships within the generation process. Such networks can track more of the generation context than a simple feed-forward model; as a side effect, they can simplify the problem of computing consistent supertag assignments for an entire sentence. 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 methodology 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 models we have implemented and the algorithms used to exploit the models in NLG. 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 NLG choices.

2 Tree Grammars

Broadly, tree grammars are a family of tree rewriting formalisms that produce strings as a side effect of composing primitive hierarchical structures. The basic 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 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: 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 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.

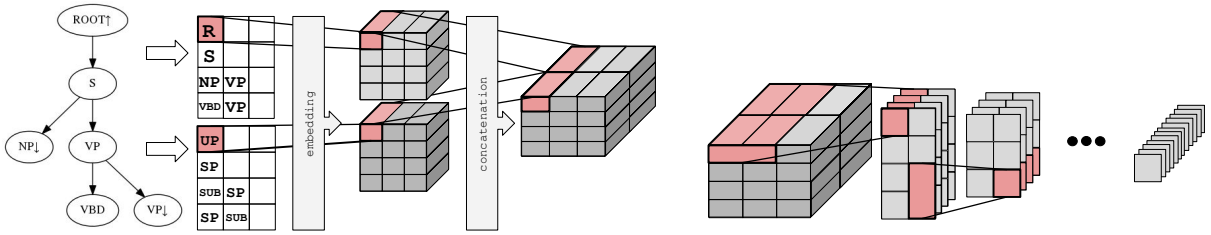
LTAG incurs computational costs because it is mildly context-sensitive in generative power. Several variants reduce the complexity of the formalism by limiting the range of adjunction operations. For example, the Tree Insertion Grammar allows for adjunction as long as it is either a left or right auxiliary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

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 data set 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 (Bangalore et al., 2001; Chiang, 2000). 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]

[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 Neural Representations

The grammar induction of § 2 allows us to construct an inventory of supertags to match a corpus. For NLG, we also need to predict how likely a supertag is to fit a specific lexical item in the context of a generation task. We approach this problem using neural networks. In particular, this work makes two contributions to improve stochastic tree modeling with neural networks. First, we represent supertags as vectors through embedding techniques that enable to model to generalize over complex, but related structures. Second, we address the hierarchical dependence between choices using a recurrent tree network that can capture long-distance influences as well as local ones. We now describe these representations in more detail.

3.1 Embedding Supertags

Different supertags for the same word can encode differences in the item's own combinatorial syntax, differences in argument structure, and differences in word order. Accordingly, words have many related supertags, with substantial overlaps in structure, and, presumably, corresponding similarities in their patterns of occurrence. A traditional machine learning approach to supertag prediction would treat individual supertags as atoms for classification; generalizing across supertags would require linking model parameters to handcrafted features or back-off categories.

By contrast, neural techniques work by embedding such tokens into a vector space. This process learns an abstract representation of tokens that clusters similar items together and makes further predictions as a function of those items' learned features. The resulting abilities to generalize across sparse data seems to be one of the most important reasons for the success of deep learning in NLP.

The simplest way to embed supertags is to treat each structure as a distinct token that indexes a corresponding learned vector. This places no constraints on the learned similarity function, but it also ignores the hierarchical structure of the elementary trees themselves. Previous work on deep learning with graph structures suggests convolutional neural networks can exploit similarities in structure. Thus, we developed analogous techniques to encode supertags based on their underlying tree structure. In particular, to embed a supertag, we embed each node, group the resulting vectors to form a tensor, and then summarize the tensor into a single vector using a series of convolutional neural networks.

Note that each elementary tree is a complex structure with nodes labeled by category and assigned a role that enables further tree operations. The root node's role represents the overall action associated with that elementary tree—either substitution or insertion. 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, and thus provide targets for further insertion.

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 descendants 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.

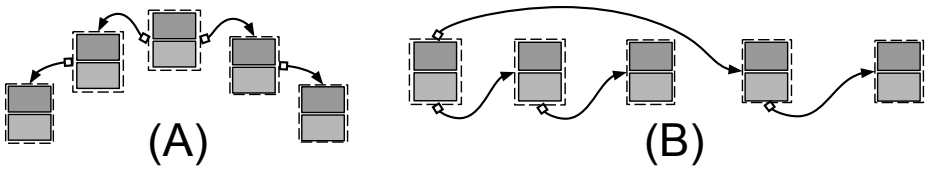


Figure 2: A recurrent tree network. (A) The dependency structure as a tree. (B) the dependency structure as a sequence.

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 in 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.

3.2 Recurrent Tree Networks

Our models predict supertags as a function of the target word and its context. Neural networks make it possible to generalize over such contexts by learning to represent them with a hidden state vector that aggregates and clusters information from the relevant history. Our approach is to do this using 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 figure 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, homogeneous computations.

4 Models

To analyze the representations we describe in § 2 and § 3, we developed two alternative architectures for predicting supertags in context. The first is a feed-forward neural network designed to solve a closely analogous task to the supertagging step of Bangalore and Rambow (2000)'s original FERGUS model. We call it Fergus-N (for Neuralized). The second uses a recurrent tree network to model the generation context. Because it has this richer context representation, it takes advantage of a slightly different characterization of the supertag prediction problem to streamline the problem solving involved in using the model. We call this Fergus-R (for Recurrent).

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

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 is treated as independent of all others. The probability of the parent's supertag is predicted using an embedding of the pair's lexical material and an embedding of the child's supertag. (Our experiments compare the different embedding options surveyed in § 3.) Training maximizes the likelihood of the training data according to the model. Formally, our objective is to minimize the negative log probability of the observed parent supertags for each parent-child pair, as formally defined in Eq. 3.

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

Here 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.

The model is implemented as a feed-forward neural network. 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.

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

At generation time, we are given a full dependency tree. A decoding step is necessary to compute a high probability assignment for all supertags simultaneously. 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 succeeds when the first complete tree has been found.[2]

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 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.

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

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 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[3] 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.[4] 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

Using the representations of § 3, the models of § 4 can be instantiated in six different ways. We can use a feed-forward Fergus-N architecture or a recurrent Fergus-R architecture. Each architecture can embed supertags *minimally*, by learning a scalar corresponding to each supertag; *atomically*, by learning an embedding vector corresponding to each supertag; or *structurally*, by using convolutional coding over each supertag's tree structure to form a vector. In each case, the vector (a size-one vector in the *mimimal* condition) is concatenated as described in § 4.

5.1 Training

We trained six such models using a common experimental platform. We started from the Wall Street Journal sections of the Penn Treebank, which have been previously used for evaluating statistical tree grammars (Chiang, 2000).[5] Our data pipeline breaks each sentence in the treebank into component elementary trees and then represents the sentence in terms of a derivation tree, specifying the tree-rewriting operations required to construct the actual treebank surface tree from the basic supertags. Abstracting syntactic information from the derivation tree leads to the unlabeled dependency trees our models assume as input.

From this input, we extracted the atomic supertag prediction instances and trained a network defined by each of the architectures of § 4 and each of the supertag representations of § 3. As always, we used Section 02-21 for development, Sections 22 for training, and Section 23 for testing. A complete description of network organization and training parameters is given in the appendix. The code and complete experimental setup are publically available[6].

5.2 Performance Metrics

We evaluate the performance of the models in several ways. First, we look at the accuracy of the supertag predictions directly output by each model. Second, we look at the accuracy of the final supertags obtained by decoding the model predictions to the best-ranked consistent global assignment. These metrics directly assess the ability of the models to successfully learn the target distributions.

Next, we evaluate the models on the full NLG task, including linearization. The linearization task allows more freedom in supertag classifications because supertags may differ in minor ways, such as the projections present along the spine, which will not affect generation output for a particular target input. The freedom means models may not be penalized based on decisions that don't matter—thus, at the same time, it also mutes the distinctions between classification decisions. We report a modified edit distance measure, Generation String Accuracy, following (Bangalore et al., 2000). Since linearization uses a beam search, we report statistics both for the top-ranked beam and for the empirically based beam

[3] 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.

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

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

[6] `https://github.com/braingineer/neural_tree_grammar`

		Accuracy		
Model	**Embedding**	**Raw Model**	**After Decoding**	**Running Time**
Fergus-N	Structural	58.17%	57.40%	1.97s
	Atomic	60.69%	55.56%	1.81s
	Minimal	52.09%	54.18%	2.02s
Fergus-R	Structural	67.62%	57.04%	0.30s
	Atomic	82.65%	62.73%	0.36s
	Minimal	10.13%	19.66%	0.54s

Table 1: For each supertag and embedding pair, the mean accuracy of supertag classification directly output by the model and in the consistent global assignment output by A* decoding. Also shown is the median running time—which includes model computation and A* search. The structural embeddings are computed with convolutional coding, the atomic embeddings as rows in a matrix, and the minimal embeddings as scalars in a vector.

among the candidates computed during search. The difference gives an indication of the effect of the language model in guiding the decisions that remain after supertagging.

Finally, we report statistics about the run time of different generation steps. This allows us to assess the complexity of the different decoding steps involved in generation, to reveal any tradeoffs among the models between speed and accuracy.

5.3 Results

Table 1 shows the results of supertag prediction. All differences between model are significant using a Paired-Sample t-test ($p < 10^{-5}$) The structural and atomic embedding methods consistently perform better, suggesting that the clustering capabilities of neural methods is a crucial part of their effectiveness. For post-decoding performance, Fergus-N utilizes the structural embeddings more than the atomic embeddings. This merits further investigation: it might be because Fergus-N predicts one supertag as a function of another, and so the compositional relationships among the two trees are more important—or because Fergus-R's contextualized decisions depend on similarities among supertags (involving argument structure or information structure) that are difficult for the convolutional coding to represent or learn. Additionally, the minimal embeddings suggests that Fergus-N's architecture might provide enough structure to make some predictions greatly simplified.

The overall best results come from Fergus-R, suggesting that it is worthwhile to take additional context into account in this task. At the same time, the median time taken to classify and decode a sentence with Fergus-R is just one sixth that of Fergus-N. We suspect that there is a general lesson in this speedup: because neural models can be more flexible about the information they take into account in decisions, it's especially advantageous in designing neural architectures to break a problem down into decisions that can be combined easily.

Finally, decoding the network generally leads to lower accuracy. It seems that our models are not doing a good job of using the predictions they make to triangulate to accurate and consistent supertags. This suggests that the models could be improved by taking more or better information into account in decoding. This is more pronounced in the atomic embeddings than the structural embeddings, which suggests that the lack of structure in the vector representation allows for the model to learn clustering relationships that don't correlate with the structural requirements.

Figure 2 shows the NLG evaluation results for the different models. All differences in model are significant using an Independent[7] t-test ($p < 10^{-5}$). For both models, the differences between structural embeddings (using convolutional coding) and atomic embeddings (using standard vector embedding techniques) were not significant, while the differences between the two embeddings and minimal embeddings were significant ($p < 10^{-5}$). The performance confirms our expectation that differences in supertag accuracy after decoding correlate with NLG accuracy overall, but that differences in NLG performance are attenuated. We note by comparison that Bangalore and Rambow report an accuracy of

[7] An Indepedent t-test was used instead of a Paired-Sample t-test because of intermittent failures during linearization that resulted in slightly different numbers of observations.

		Accuracy	
Model	**Embedding**	**Top Scoring**	**Best Performance**
Fergus-N	Structural	65.80%	72.58%
	Atomic	65.52%	71.82%
	Minimal	63.79%	71.09%
Fergus-R	Structural	68.22%	74.70%
	Atomic	69.29%	75.56%
	Minimal	58.23%	65.04%

Table 2: Shown above as accuracy is the percentage of tokens in the linearized strings that are in correct positions according to an edit distance measure.

74.9% in their best evaluation of FERGUS—on a data set of just 100 sentences with an average length of 16.7. Our evaluation, on 2400 sentences with an average length of 22, is more strenuous.

6 Related Work

There are several lines of related work which explore stochastic tree models from the standpoint of parsing and understanding. While using the same methods, NLG has different goals and we think the perspective is instructive. Where parsing infers most probable underlying structure, generation infers the most likely way of expressing a semantic structure. This divergence of goals leads to different concerns, alternatives, and emphasis.

The works most similar to ours explicitly model tree structures, but focus on resolving the uncertainty invovled with the latent structure of an observed sentence. For example, the top down tree structure of Zhang et al. (2016) expresses the generation of a dependency structure as the decisions of a set of long short-term memory networks. For each decision, the possible options are different tree structures which can produce the target linear form. In contrast, the generation problem is concerned with different linear forms that can result from the same tree structure. In more extensive tasks, the generation problem can include simulated interpretion to inform decisions; using the ease of structural inference from linear form quantifies the understandability of a sentence.

Although the methodology presented in this work is closely related to several recent neural networks models for long-distance relationships, it differs distinctly in its treatment of state and search. Specifically, forward-planning in a generation task produces a growing horizon of syntactic choices while shrinking the horizon of semantic goals. At each step, syntactic operations grow the number of available syntactic choices while limiting the number of semantic goals left to express. In contrast, parsing and understanding begin with the surface form and construct the organized semantic content, either for a downstream decision or just for the structure itself. The most notable works in this line of research are the recurrent neural network grammars (Dyer et al., 2016), a shift-reduce parser and interpreter (Bowman et al., 2016), and a dynamic network for composing other neural network modules (Andreas et al., 2016). Interestingly, there is a common theme of using indexable and dynamic data structures in neural architecture to make long-distance decisions.

7 Conclusion

This paper has explored issues in deep learning of probabilistic tree grammars from the standpoint of natural language generation. For NLG, we need models that predict high-probability structures to encode deep linguistic relationships—rather than to infer deep relationships from surface cues. This problem brings new challenges for learning, as it requires us to represent new kinds of linguistic elements and new kinds of structural context in order to capture the regularities involved. Despite these challenges, however, the problem continues to have the mix of data sparsity, rich primitives and combinatorial interactions that has made deep learning attractive for use in natural language parsing and understanding.

Of the range of models we surveyed here, the best combines a top down tree recurrence to cluster contexts with appropriate embedding methods to cluster syntactic and lexical elements. Our evaluations suggest that the model is more accurate and faster than alternative techniques. However, it would still be

Model Parameter	Value
Fergus-N Parameters	
Fully connected layer size	256
Batch size	128
Fergus-R Parameters	
Fully connected layer size	256
Hidden state size	128
Batch size	16
Embedding Parameters	
Convolution filter size	48
Syntactic category embedding size	32
Node role embedding size	32
Word embedding size (Pennington et al., 2014)	300

Model Parameter	Value
Language Model Parameters	
Hidden state size	368
Batch size	32
Optimization Parameters	
Optimization Algorithm	ADAM
Fergus-R and Fergus-N Learning Rate	1e-4
Language Model Learning Rate	0.01
Fully-Connected Dropout Rate	0.5
Recurrent Weight Dropout Rate	0.2
L_2 Weight Decay	1e-6
Max gradient norm	10.0
Gradient clip threshold	5.0

Table 3: The parameters for the Fergus-R, Fergus-N, and language models. The exact specifications in configuration files can be found in the code repository that accompanies this paper.

good to analyze the performance of the model more deeply. Can we get better results in the key decoding step? How do human readers find the output of the system?

Looking forward, we see this research a step towards learned models that capture more of the NLG task. We plan to explore similar techniques in planning surface text from more properly semantic inputs or even from abstract communicative goals. Further, we plan to integrate learned methods with knowledge-based techniques to offer designers more control over system output in specific applications. Developing methods appropriate to such settings will require researchers to revisit the core problems of generalizing across linguistic structures and contexts—and, we hope, to build on and extend the provisional solutions we have explored here.

Acknowledgments

This research was supported in part by NSF IIS-1526723 and by a sabbatical leave from Rutgers to Stone.

A Appendix

In this appendix, we describe some design decisions that took place which did not have a place in the paper but should still be reported. In particular, this section covers the data processing, parameters used in the final model, and design decisions for the restricting the supertag distributions. As a reminder, the code, tex sources, and experimental setup are hosted on github[8].

A.1 Data

The corpus was preprocessed using the Stanford Parser to fix common issues and remove extraneous information. The resulting parse trees were then annotated by marking the head words, the dependents, and the adjuncts. The annotated trees were split at adjunction and substitution positions to form the grammar. For the dependency structures used in the models, we use the derivation tree formed by the grammar for each parse tree. Derivation trees were processed to parent-child pairs and top-down traversals for Fergus-N and Fergus-R. All of these steps are released with the code.

A.2 Model parameters

All of the models were implemented in the Keras (Chollet, 2015) and Theano (Theano Development Team, 2016) libraries. The specific parameters that were used are shown in Table 3. The parameters were selected by measured performance on the development portion of the data set. In the accompanying code repository, the full experiment parameters—including programmatic parameters controlling the experimental design—are specified in configuration files.

[8] https://github.com/braingineer/neural_tree_grammar

References

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Learning to compose neural networks for question answering. *CoRR*, abs/1601.01705.

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, Owen Rambow, and Steve Whittaker. 2000. Evaluation metrics for generation. In *Proceedings of the first international conference on Natural language generation-Volume 14*, pages 1–8. 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*.

Samuel R Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D Manning, and Christopher Potts. 2016. A fast unified model for parsing and sentence understanding. *arXiv preprint arXiv:1603.06021*.

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.

François Chollet. 2015. Keras. `https://github.com/fchollet/keras`.

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.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. 2016. Recurrent neural network grammars. *arXiv preprint arXiv:1602.07776*.

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.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*, 12:1532–1543.

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.

Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May.

Xingxing Zhang, Liang Lu, and Mirella Lapata. 2016. Top-down tree long short-term memory networks. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 310–320, San Diego, California, June. Association for Computational Linguistics.