Learning Context-Free Languages with Nondeterministic Stack RNNs

Brian DuSell
University of Notre Dame
bdusell1@nd.edu

David Chiang
University of Notre Dame
dchiang@nd.edu

Abstract

We present a differentiable stack data structure that simultaneously and tractably encodes an exponential number of stack configurations, based on Lang's algorithm for simulating nondeterministic pushdown automata. We call the combination of this data structure with a recurrent neural network (RNN) controller a Nondeterministic Stack RNN. We compare our model against existing stack RNNs on various formal languages, demonstrating that our model converges more reliably to algorithmic behavior on deterministic tasks, and achieves lower cross-entropy on inherently nondeterministic tasks.

1 Introduction

Although recent neural models of language have made advances in learning syntactic behavior, research continues to suggest that inductive bias plays a key role in data efficiency and human-like syntactic generalization (van Schijndel et al., 2019; Hu et al., 2020). Based on the long-held observation that language exhibits hierarchical structure, previous work has proposed coupling recurrent neural networks (RNNs) with differentiable stack data structures (Joulin and Mikolov, 2015; Grefenstette et al., 2015) to give them some of the computational power of pushdown automata (PDAs), the class of automata that recognize context-free languages (CFLs). However, previously proposed differentiable stack data structures only model deterministic stacks, which store only one version of the stack contents at a time, theoretically limiting the power of these stack RNNs to the deterministic CFLs.

A sentence's syntactic structure often cannot be fully resolved until its conclusion (if ever), requiring a human listener to track multiple possibilities while hearing the sentence. Past work in psycholinguistics has suggested that models that keep multiple candidate parses in memory at once can explain human reading times better than models which assume harsher computational constraints. This ability also plays an important role in calculating expectations that facilitate more efficient language processing (Levy, 2008). Current neural language models do not track multiple parses, if they learn syntax generalizations at all (Futrell et al., 2019; Wilcox et al., 2019; McCoy et al., 2020).

We propose a new differentiable stack data structure that explicitly models a nondeterministic PDA, adapting an algorithm by Lang (1974) and reformulating it in terms of tensor operations. The algorithm is able to represent an exponential number of stack configurations at once using cubic time and quadratic space complexity. As with existing stack RNN architectures, we combine this data structure with an RNN controller, and we call the resulting model a Nondeterministic Stack RNN (NS-RNN).

We predict that nondeterminism can help language processing in two ways. First, it will improve trainability, since all possible sequences of stack operations contribute to the objective function, not just the sequence used by the current model. Second, it will improve expressivity, as it is able to model concurrent parses in ways that a deterministic stack cannot. We demonstrate these claims by comparing the NS-RNN to deterministic stack RNNs on formal language modeling tasks of varying complexity. To show that nondeterminism aids training, we show that the NS-RNN achieves lower cross-entropy, in fewer parameter updates, on some deterministic CFLs. To show that nondeterminism improves expressivity, we show that the NS-RNN achieves lower cross-entropy on nondeterministic CFLs, including the "hardest context-free language" (Greibach, 1973), a language which is at least as difficult to parse as any other CFL and inherently requires nondeterminism. Our code is available at https://github.com/bdusell/nondeterministic-stack-rnn.

Proceedings of the 24th Conference on Computational Natural Language Learning, pages 507–519
Online, November 19-20, 2020.
https://doi.org/10.18653/v1/P17

2 Background and Motivation

In all differentiable stack-augmented networks that we are aware of (including ours), a network called the *controller*, which is some kind of RNN (typically an LSTM), is augmented with a differentiable stack, which has no parameters of its own. At each time step, the controller emits weights for various stack operations, which at minimum include push and pop. To maintain differentiability, the weights need to be continuous; different designs for the stack interpret fractionally-weighted operations differently. The stack then executes the fractional operations and produces a stack *reading*, which is a vector that represents the top of the updated stack. The stack reading is used as an extra input to the next hidden state update.

Designs for differentiable stacks have proceeded generally along two lines. One approach, which we call *superposition* (Joulin and Mikolov, 2015), treats fractional weights as probabilities. The other, which we call *stratification* (Sun et al., 1995; Grefenstette et al., 2015), treats fractional weights as "thicknesses."

Superposition In the model of Joulin and Mikolov (2015), the controller emits at each time step a probability distribution over three stack operations: push a new vector, pop the top vector, and no-op. The stack simulates all three operations at once, setting each stack element to the weighted interpolation of the elements above, at, and below it in the previous time step, weighted by push, no-op, and pop probabilities respectively. Thus, each stack element is a superposition of possible values for that element. Because stack elements depend only on a fixed number of elements from the previous time step, the stack update can largely be parallelized. Yogatama et al. (2018) developed an extension to this model that allows a variable number of pops per time step, up to a fixed limit K. Suzgun et al. (2019) also proposed a modification of the controller parameterization.

Stratification The model proposed by Sun et al. (1995) and later studied by Grefenstette et al. (2015) takes a different approach, assigning a *strength* between 0 and 1 to each stack element. If the stack elements were the layers of a cake, then the strengths would represent the thickness of each layer. At each time step, the controller emits a push weight between 0 and 1 which determines the strength of a new vector pushed onto the stack, and a pop weight between 0 and 1 which determines how much to slice off the top of the stack. The stack reading is computed by examining the top layer of unit thickness and interpolating the vectors proportional to their strengths. This relies on min and max operations, which can have zero gradients. In practice, the model can get trapped in local optima and requires random restarts (Hao et al., 2018). This model also affords less opportunity for parallelization because of the interdependence of stack elements within the same time step. Hao et al. (2018) proposed an extension that uses memory buffers to allow variable-length transductions.

Nondeterminism In all the above models, the stack is essentially deterministic in design. In order to recognize a nondeterministic CFL like $\{ww^{\mathrm{R}}\}$ from left to right, it must be possible, at each time step, for the stack to track all prefixes of the input string read so far. None of the foregoing models, to our knowledge, can represent a set of possiblities like this. Even for deterministic CFLs, this has consequences for trainability; at each time step, training can only update the model from the vantage point of a single stack configuration, making the model prone to getting stuck in local minima.

To overcome this weakness, we propose incorporating a nondeterministic stack, which affords the model a global view of the space of possible ways to use the stack. Our controller emits a probability distribution over stack operations, as in the superposition approach. However, whereas superposition only maintains the per-element marginal distributions over the stack elements, we propose to maintain the full distribution over the whole stack contents. We marginalize the distribution as late as possible, when the controller queries the stack for the current top stack symbol.

In the following sections, we explain our model and compare it against those of Joulin and Mikolov (2015) and Grefenstette et al. (2015). Despite taking longer in wall-clock time to train, our model learns to solve the tasks optimally with a higher rate of success.

3 Pushdown Automata

In this section, we give a definition of nondeterministic PDAs (§3.2), describe how to process strings with nondeterministic PDAs in cubic time (§3.3), and reformulate this algorithm in terms of tensor operations (§3.4).

3.1 Notation

Let ϵ be the empty string. Let $\mathbb{1}[\phi]$ be 1 when proposition ϕ is true, 0 otherwise. If A is a matrix, let $A_{i:}$ and $A_{:j}$ be the ith row and jth column, respectively, and define analogous notation for tensors.

3.2 Definition

A *weighted pushdown automaton (PDA)* is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, \bot)$, where:

- Q is a finite set of states.
- Σ is a finite input alphabet.
- Γ is a finite stack alphabet.
- $\delta\colon Q \times \Gamma \times \Sigma \times Q \times \Gamma^* \to \mathbb{R}_{\geq 0}$ maps transitions, which we write as $q, x \xrightarrow{a} r, y$, to weights.
- $q_0 \in Q$ is the start state.
- $\bot \in \Gamma$ is the initial stack symbol.

In this paper, we do not allow non-scanning transitions (that is, those where $a = \epsilon$). Although this does not reduce the weak generative capacity of PDAs (Autebert et al., 1997), it could affect their ability to learn; we leave exploration of non-scanning transitions for future work.

For simplicity, we will assume that all transitions have one of the three forms:

$$\begin{aligned} q, x &\xrightarrow{a} r, xy && \text{push } y \text{ on top of } x \\ q, x &\xrightarrow{a} r, y && \text{replace } x \text{ with } y \\ q, x &\xrightarrow{a} r, \epsilon && \text{pop } x. \end{aligned}$$

This also does not reduce the weak generative capacity of PDAs.

Given an input string $w \in \Sigma^*$ of length n, a *configuration* is a triple (i, q, β), where $i \in [0, n]$ is an input position indicating that all symbols up to and including w_i have been scanned, $q \in Q$ is a state, and $\beta \in \Gamma^*$ is the content of the stack (written bottom to top). For all i, q, r, β, x, y, we say that $(i{-}1, q, \beta x)$ *yields* $(i, r, \beta y)$ if $\delta(q, x \xrightarrow{w_i} r, y) > 0$. A *run* is a sequence of configurations starting with $(0, q_0, \bot)$ where each configuration (except the last) yields the next configuration.

Because our model does not use the PDA to accept or reject strings, we omit the usual definitions for the language accepted by a PDA. This is also why our definition lacks accept states.

As an example, consider the following PDA, for the language $\{ww^{\mathrm{R}} \mid w \in \{\mathtt{0}, \mathtt{1}\}^*\}$:

$$\begin{aligned} M &= (Q, \Sigma, \Gamma, \delta, q_1, \bot) \\ Q &= \{q_1, q_2\} \\ \Sigma &= \{\mathtt{0}, \mathtt{1}\} \\ \Gamma &= \{\mathtt{0}, \mathtt{1}, \bot\} \end{aligned}$$

where δ contains the transitions

$$\begin{aligned} q_1, x &\xrightarrow{a} q_1, xa && x \in \Gamma, a \in \Sigma \\ q_1, a &\xrightarrow{a} q_2, \epsilon && a \in \Sigma \\ q_2, a &\xrightarrow{a} q_2, \epsilon && a \in \Sigma. \end{aligned}$$

This PDA has a possible configuration with an empty stack ($\bot$) iff the input string read so far is of the form ww^{R}.

To make a weighted PDA probabilistic, we require that all transition weights be nonnegative and, for all a, q, x:

$$\sum_{r \in Q} \sum_{y \in \Gamma^*} \delta(q, x \xrightarrow{a} r, y) = 1.$$

Whereas many definitions make the model generate symbols (Abney et al., 1999), our definition makes the PDA operations conditional on the input symbol a. The difference is not very important, because the RNN controller will eventually assume responsibility for reading and writing symbols, but our definition makes the shift to an RNN controller below slightly simpler.

3.3 Recognition

Lang (1974) gives an algorithm for simulating all runs of a nondeterministic PDA, related to Earley's algorithm (Earley, 1970). At any point in time, there can be exponentially many possibilities for the contents of the stack. In spite of this, Lang's algorithm is able to represent the set of all possibilities using only quadratic space. As this set is regular, its representation can be thought of as a weighted finite automaton, which we call the *stack WFA*, similar to the graph-structured stack used in GLR parsing (Tomita, 1987).

Figure 1 depicts Lang's algorithm as a set of inference rules, similar to a deductive parser (Shieber et al., 1995; Goodman, 1999), although the visual presentation is rather different. Each inference rule is drawn as a fragment of the stack WFA. If the transitions drawn with solid lines are present in the stack WFA, and the side conditions in the right column are met, then the transition drawn with a

dashed line can be added to the stack WFA. The algorithm repeatedly applies inference rules to add states and transitions to the stack WFA; no states or transitions are ever deleted.

Each state of the stack WFA is of the form (i, q, x), where i is a position in the input string, q is a PDA state, and x is the top stack symbol. We briefly explain each of the inference rules:

Axiom creates an initial state and pushes $\perp$ onto the stack.

Push pushes a y on top of an x. Unlike Lang's original algorithm, this inference rule applies whether or not state $(j{-}1, q, x)$ is reachable.

Replace pops a z and pushes a y, by backing up the z transition (without deleting it) and adding a new y transition.

Pop pops a z, by backing up the z transition as well as the preceding y transition (without deleting them) and adding a new y transition.

The set of accept states of the stack WFA changes from time step to time step; at step j, the accept states are $\{(j, q, x) \mid q \in Q, x \in \Gamma\}$. The language recognized by the stack WFA at time j is the set of possible stack contents at time j.

An example run of the algorithm is shown in Figure 2, using our example PDA and the string `0110`. At time step $j = 3$, the PDA reads 1 and either pushes a 1 (path ending in state $(3, q_1, 1)$) or pops a 1 (path ending in state $(3, q_2, \mathtt{0})$). Similarly at time step $j = 4$, and the existence of a state with top stack symbol $\perp$ indicates that the string is of the form ww^{R}.

The total running time of the algorithm is proportional to the number of ways that the inference rules can be instantiated. Since the Pop rule contains three string positions (i, j, and k), the time complexity is $O(n^3)$. The total space requirement is characterized by the number of possible WFA transitions. Since transitions connect two states, each with a string position (i and j), the space complexity is $O(n^2)$.

3.4 Inner and Forward Weights

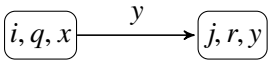
To implement this algorithm in a typical neural-network framework, we reformulate it in terms of tensor operations. We use the assumption that all transitions are scanning, although it would be possible to extend the model to handle non-scanning transitions using matrix inversions (Stolcke, 1995).

Define $\mathrm{Act}(\Gamma) = \bullet\Gamma \cup \Gamma \cup \{\epsilon\}$ to be a set of possible stack actions: if $y \in \Gamma$, then $\bullet y$ means "push y," y means "replace with y," and ϵ means "pop."

Given an input string w, we pack the transition weights of the PDA into a tensor Δ with dimensions $n \times |Q| \times |\Gamma| \times |Q| \times |\mathrm{Act}(\Gamma)|$:

$$\begin{aligned} \Delta[j][q, x \to r, \bullet y] &= \delta(q, x \xrightarrow{w_j} r, xy) \\ \Delta[j][s, z \to r, y] &= \delta(s, z \xrightarrow{w_j} r, y) \\ \Delta[j][s, z \to r, \epsilon] &= \delta(s, z \xrightarrow{w_j} r, \epsilon). \end{aligned} \tag{1}$$

We compute the transition weights of the stack WFA (except for the initial transition) as a tensor of *inner weights* γ, with dimensions $n \times n \times |Q| \times |\Gamma| \times |Q| \times |\Gamma|$. Each element, which we write as $\gamma[i \to j][q, x \to r, y]$, is the weight of the stack WFA transition



The equations defining γ are shown in Figure 3. Because these equations are a recurrence relation, we cannot compute γ all at once, but (for example) in order of increasing j.

Additionally, we compute a tensor α of *forward weights* of the stack WFA. This tensor has dimensions $n \times |Q| \times |\Gamma|$, and its elements are defined by the recurrence

$$\alpha[1][r, y] = \mathbb{1}[r = q_0 \wedge y = \perp]$$

$$\alpha[j][r, y] = \sum_{i=1}^{j-1} \sum_{q,x} \alpha[i][q, x]\, \gamma[i \to j][q, x \to r, y] \qquad (2 \le j \le n).$$

The weight $\alpha[j][r, y]$ is the total weight of reaching a configuration $(r, j, \beta y)$ for any β from the initial configuration, and we can use α to compute the probability distribution over top stack symbols at time step j:

$$\tau^{(j)}(y) = \frac{\sum_r \alpha[j][r, y]}{\sum_{y'} \sum_r \alpha[j][r, y']}.$$

4 Neural Pushdown Automata

Now we couple the tensor formulation of Lang's algorithm for nondeterministic PDAs with an RNN controller.

4.1 Model

The controller can be any type of RNN; in our experiments, we used a LSTM RNN. At each time step, it computes a hidden vector $\mathbf{h}^{(j)}$ with d dimensions from the previous hidden vector, an input

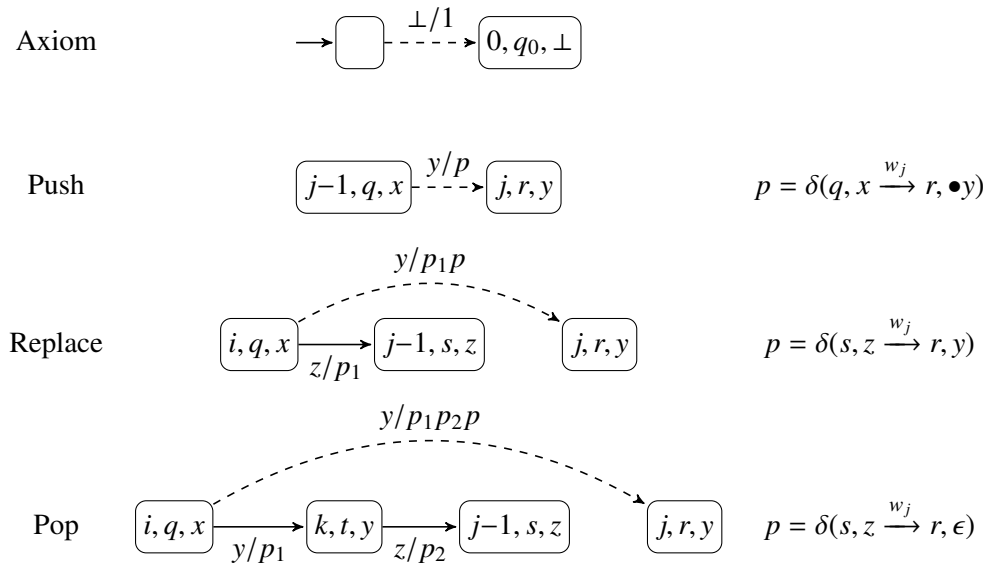


Figure 1: Lang's algorithm drawn as operations on the stack WFA. Solid edges indicate existing transitions; dashed edges indicate transitions that are added as a result of the stack operation.

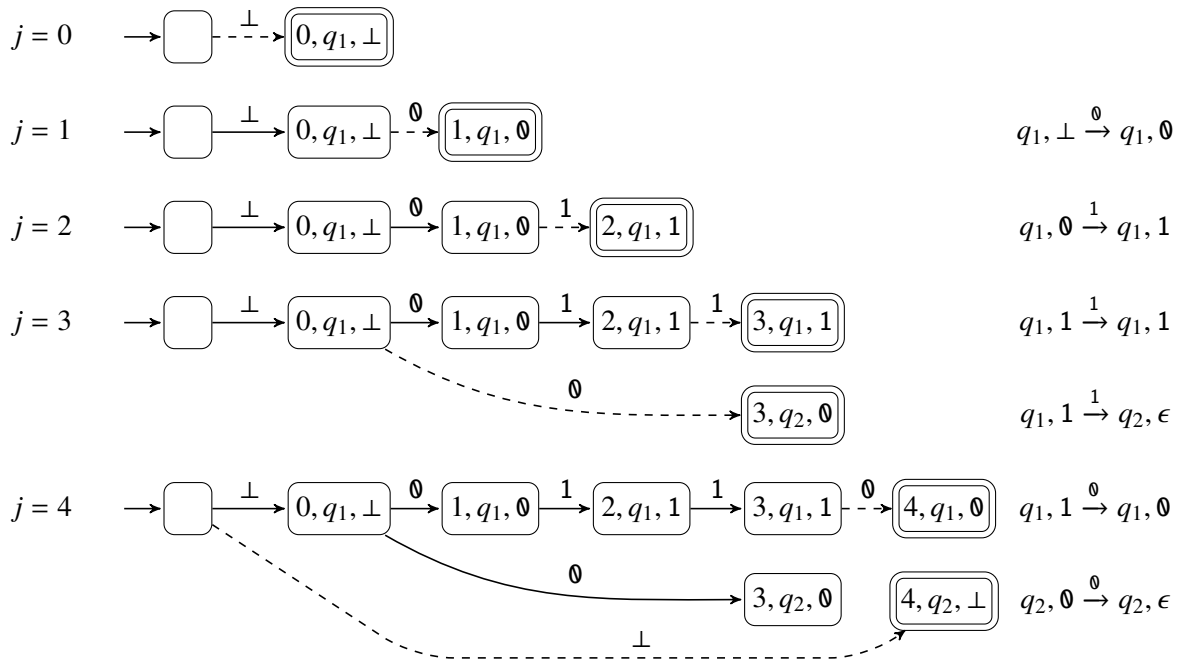


Figure 2: Run of Lang's algorithm on our example PDA and the string `0110`. The PDA transitions used are shown at right.

For $1 \le i < j \le n$,

$$\begin{aligned}
&\gamma[i \to j][q, x \to r, y] = \\
&\quad \mathbb{1}[i = j{-}1]\ \Delta[j][q, x \to r, \bullet y] && \text{Push} \\
&\quad + \sum_{s,z} \gamma[i \to j{-}1][q, x \to s, z]\ \Delta[j][s, z \to r, y] && \text{Replace} \\
&\quad + \sum_{k=i+1}^{j-2} \sum_{t} \sum_{s,z} \gamma[i \to k][q, x \to t, y]\ \gamma[k \to j{-}1][t, y \to s, z]\ \Delta[j][s, z \to r, \epsilon] && \text{Pop}
\end{aligned}$$

Figure 3: Equations for computing inner weights.

vector $\mathbf{x}^{(j)}$, and the distribution over current top stack symbols, $\tau^{(j)}$, defined above:

$$\mathbf{h}^{(j)} = R\left(\mathbf{h}^{(j-1)}, \begin{bmatrix}\mathbf{x}^{(j)} \\ \tau^{(j)}\end{bmatrix}\right)$$

where R can be any RNN unit. This state is used to compute an output vector $\mathbf{y}^{(j)}$ as usual:

$$\mathbf{y}^{(j)} = \text{softmax}\left(\mathbf{A}\mathbf{h}^{(j)} + \mathbf{b}\right)$$

where $\mathbf{A}$ and $\mathbf{b}$ are parameters with dimensions $|\Sigma| \times d$ and $|\Sigma|$, respectively. In addition, the state is used to compute a conditional distribution over actions, $\Delta[j]$:

$$\mathbf{z}^{(j)}_{qxry} = \exp\left(\mathbf{C}_{qxry:}\mathbf{h}^{(j)} + \mathbf{D}_{qxry}\right)$$

$$\Delta[j][q, x \to r, y] = \frac{\mathbf{z}^{(j)}_{qxry}}{\sum_{r',y'} \mathbf{z}^{(j)}_{qxr'y'}}$$

where $\mathbf{C}$ and $\mathbf{D}$ are tensors of parameters with dimensions $|Q| \times |\Gamma| \times |Q| \times |\text{Act}(\Gamma)| \times d$ and $|Q|\times|\Gamma|\times|Q|\times|\text{Act}(\Gamma)|$, respectively. (This is just an affine transformation followed by a softmax over r and y.) These equations replace equations (1).

4.2 Implementation

We implemented the NS-RNN using PyTorch (Paszke et al., 2019), and doing so efficiently required a few crucial tricks. The first was a workaround to update the γ and α tensors in-place in a way that was compatible with PyTorch's automatic differentiation; this was necessary to achieve the theoretical quadratic space complexity. The second was an efficient implementation of a differentiable `einsum` operation[1] that supports the log semiring (as well as other semirings), which allowed us to implement the equations of Figure 3 in a reasonably fast, memory-efficient way that avoids underflow. Our `einsum` implementation splits the operation into fixed-size blocks where the multiplication and summation of terms can be fully parallelized. This enforces a reasonable upper bound on memory usage while suffering only a slight decrease in speed compared to fully parallelizing the entire `einsum` operation.

[1] https://github.com/bdusell/semiring-einsum

5 Experiments

In this section, we describe our experiments comparing our NS-RNN and three baseline language models on several formal languages.

5.1 Tasks

Marked reversal The language of palindromes with an explicit middle marker, with strings of the form $w\#w^{\text{R}}$, where $w \in \{\mathtt{0}, \mathtt{1}\}^*$. This task should be easily solvable by a model with a deterministic stack, as the model can push the string w to the stack, change states upon reading `#`, and predict w^{R} by popping w from the stack in reverse.

Unmarked reversal The language of (even-length) palindromes without a middle marker, with strings of the form ww^{R}, where $w \in \{\mathtt{0}, \mathtt{1}\}^*$. When the length of w can vary, a language model reading the string from left to right must use nondeterminism to guess where the boundary between w and w^{R} lies. At each position, it must either push the input symbol to the stack, or else guess that the middle point has been reached and start popping symbols from the stack. An optimal language model will interpolate among all possible split points to produce a final prediction.

Padded reversal Like the unmarked reversal language, but with a long stretch of repeated symbols in the middle, with strings of the form wa^pw^{R}, where $w \in \{\mathtt{0}, \mathtt{1}\}^*$, $a \in \{\mathtt{0}, \mathtt{1}\}$, and $p \ge 0$. The

purpose of the padding is to confuse a language model attempting to guess where the middle of the palindrome is based on the content of the string. In the general case of unmarked reversal, a language model can disregard split points where a valid palindrome does not occur locally. Since all substrings of a^p are palindromes, the language model must deal with a larger number of candidates simultaneously.

Dyck language The language D_2 of strings with two kinds of balanced brackets.

Hardest CFL Designed by Greibach (1973) to be at least as difficult to parse as any other CFL:

$$\begin{aligned} L_0 = \{ & x_1 \, , y_1 \, , z_1 \, ; \cdots x_n \, , y_n \, , z_n \, ; \mid \\ & n \geq 0, \\ & y_1 \cdots y_n \in \$D_2, \\ & x_i, z_i \in \{ \, , \, \$, (,), [,] \}^* \}. \end{aligned}$$

Intuitively, L_0 contains strings formed by dividing a member of $\$D_2$ into pieces (y_i) and interleaving them with "decoy" pieces (substrings of x_i and z_i). While processing the string, the machine has to nondeterministically guess whether each piece is genuine or a decoy. Greibach shows that for any CFL L, there is a string homomorphism h such that a parser for L_0 can be run on $h(w)$ to find a parse for w. See Appendix A for more information.

5.2 Data

For each task, we construct a probabilistic context-free grammar (PCFG) for the language (see Appendix B for the full grammars and their parameters). We then randomly sample a training set of 10,000 examples from the PCFG, filtering samples so that the length of a string is in the interval $[40, 80]$ (see Appendix C for our sampling method). The training set remains the same throughout the training process and is not re-sampled from epoch to epoch, since we want to test how well the model can infer the probability distribution from a finite sample.

We sample a validation set of 1,000 examples from the same distribution and a test set with string lengths varying from 40 to 100, with 100 examples per length. The validation set is randomized in each experiment, but for each task, the test set remains the same across all models and random restarts. For simplicity, we do not filter training samples from the validation or test sets, assuming that the chance of overlap is very small.

5.3 Evaluation

Since, in these languages, the next symbol cannot always be predicted deterministically from previous symbols, we do not use prediction accuracy as in previous work. Instead, we compute per-symbol cross-entropy on a set of strings S. Let p be any distribution over strings; then:

$$H(S, p) = \frac{\sum_{w \in S} -\log p(s)}{\sum_{w \in S} |w|}.$$

We compute the cross-entropy for both the stack RNN and the distribution from which S is sampled and report the difference. This can be seen as an approximation of the KL divergence of the stack RNN from the true distribution.

Technically, because the RNN models do not predict the end of the string, they estimate $p(w \mid |w|)$, not $p(w)$. However, they do not actually use any knowledge of the length, so it seems reasonable to compare the RNN's estimate of $p(w \mid |w|)$ with the true $p(w)$. (This is why, when we bin by length in Figure 5, some of the differences are negative.)

A benefit of using cross-entropy instead of prediction accuracy is that we can easily incorporate new tasks as long as they are expressed as a PCFG. We do not, for example, need to define a language-dependent subsequence of symbols to evaluate on.

5.4 Baselines

We compare our NS-RNN against three baselines: an LSTM, the Stack LSTM of Joulin and Mikolov (2015) ("JM"), and the Stack LSTM of Grefenstette et al. (2015) ("Gref"). We deviate slightly from the original definitions of these models in order to standardize the controller-stack interface to the one defined in Section 4.1, and to isolate the effects of differences in the stack data structure, rather than the controller mechanism. For all three stack models, we use an LSTM controller whose initial hidden state is fixed to 0, and we use only one stack for the JM and Gref models. (In early experiments, we found that using multiple stacks did not make a meaningful difference in performance.) For JM, we include a bias term in the layers that compute the stack actions and network output. We do allow the no-op operation, and the stack reading consists of only the top stack cell. For Gref, we set the controller output $\mathbf{o}'_t$ equal to the hidden state $\mathbf{h}_t$, so we compute the stack actions, pushed vector, and network output directly from the hidden state. We encode all input symbols as one-hot vectors; there are no embedding layers.

5.5 Hyperparameters

For all models, we use a single-layer LSTM with 20 hidden units. We selected this number because we found that an LSTM of this size could not completely solve the marked reversal task, indicating that the hidden state is a memory bottleneck. For each task, we perform a hyperparameter grid search for each model. We search for the initial learning rate, which has a large impact on performance, from the set $\{0.01, 0.005, 0.001, 0.0005\}$. For JM and Gref, we search for stack embedding sizes in $\{2, 20, 40\}$. We manually choose a small number of PDA states and stack symbol types for the NS-RNN for each task. For marked reversal, unmarked reversal, and Dyck, we use 2 states and 2 stack symbol types. For padded reversal, we use 3 states and 2 stack symbol types. For the hardest CFL, we use 3 states and 3 stack symbol types.

As noted by Grefenstette et al. (2015), initialization can play a large role in whether a Stack LSTM converges on algorithmic behavior or becomes trapped in a local optimum. To mitigate this, for each hyperparameter setting in the grid search, we run five random restarts and select the hyperparameter setting with the lowest average difference in cross entropy on the validation set. This gives us a picture not only of the model's performance, but of its rate of success. We initialize all fully-connected layers except for the recurrent LSTM layer with Xavier uniform initialization (Glorot and Bengio, 2010), and all other parameters uniformly from $[-0.1, 0.1]$.

We train all models with Adam (Kingma and Ba, 2015) and clip gradients whose magnitude is above 5. We use mini-batches of size 10; to generate a batch, we first select a length and then sample 10 strings of that length. We train models until convergence, multiplying the learning rate by 0.9 after 5 epochs of no improvement in cross-entropy on the validation set, and stopping after 10 epochs of no improvement.

6 Results

We show plots of the difference in cross entropy on the validation set between each model and the source distribution in Figure 4. For all tasks, stack-based models outperform the LSTM baseline, indicating that the tasks are effective benchmarks for differentiable stacks. For the marked reversal, unmarked reversal, and hardest CFL tasks, our model consistently achieves cross-entropy closer to the source distribution than any other model. Even for the marked reversal task, which can be solved deterministically, the NS-RNN, besides achieving lower cross-entropy on average, learns to solve the task in fewer updates and with much higher reliability across random restarts. In the case of the mildly nondeterministic unmarked reversal and highly nondeterministic hardest CFL tasks, the NS-RNN converges on the lowest validation cross-entropy. On the Dyck language, which is a deterministic task, all stack models converge quickly on the source distribution. We hypothesize that this is because the Dyck language represents a case where stack usage is locally advantageous everywhere, so it is particularly conducive for learning stack-like behavior. On the other hand, we note that our model struggles on padded reversal, in which stack-friendly signals are intentionally made very distant. Although the NS-RNN outperforms the LSTM baseline, the JM model solves the task most effectively, though still imperfectly.

In order to show how each model performs when evaluated on strings longer than those seen during training, in Figure 5, we show cross-entropy on separately sampled test data as a function of string length. All test sets are identical across models and random restarts, and there are 100 samples per length. The NS-RNN consistently does well on string lengths it was trained on, but it is sometimes surpassed by other stack models on strings that are outside the distribution of lengths it was trained on. This suggests that the NS-RNN conforms more tightly to the real distribution seen during training.

7 Conclusion

We presented the NS-RNN, a neural language model with a differentiable stack that explicitly models nondeterminism. We showed that it offers improved trainability and modeling power over previous stack-based neural language models; the NS-RNN learns to solve some deterministic tasks more effectively than other stack-LSTMs, and achieves the best results on a challenging nondeterministic context-free language. However, we note that the NS-RNN struggled on a task where signals in the data were distant, and did not generalize to longer lengths as well as other stack-LSTMs; we hope to address these shortcomings in future work. We believe that the NS-RNN will prove to be a powerful tool for learning and modeling ambiguous syntax in natural language.

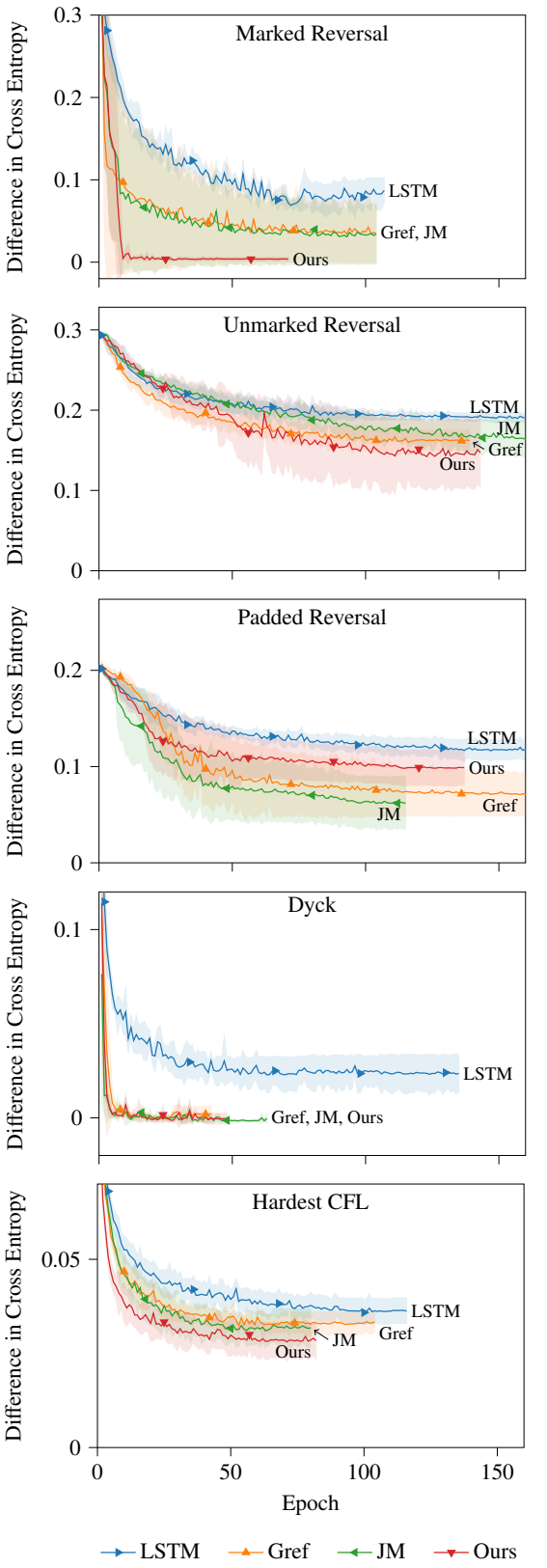


Figure 4: Cross-entropy difference in nats between model and source distribution on validation set, as a function of training time. Lines are averages of five random restarts, and shaded regions are standard deviations. After a random restart converges, the value of its last epoch is used in the average for later epochs.

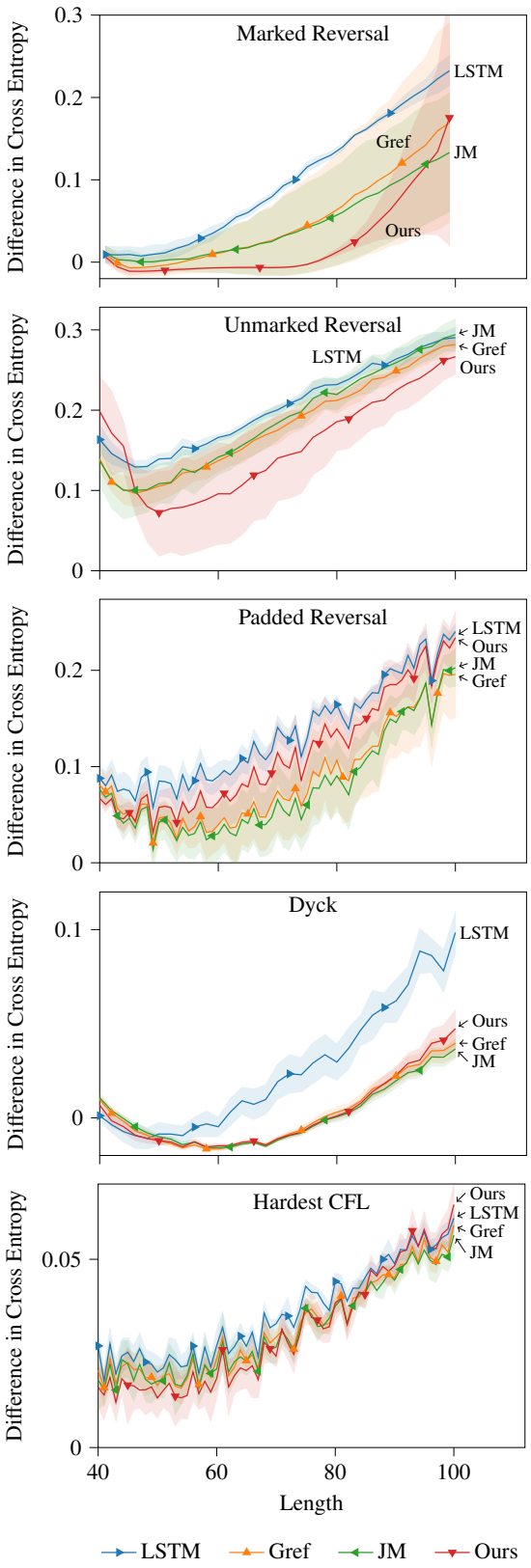


Figure 5: Cross-entropy difference in nats on the test set, binned by string length. Some models achieve a negative difference, for reasons explained in §5.3. Each line is the average of the same five random restarts shown in Figure 4.

Acknowledgements

This research was supported in part by a Google Faculty Research Award. We would like to thank Justin DeBenedetto and Darcey Riley for their helpful comments, and the Center for Research Computing at the University of Notre Dame for providing the computing infrastructure for our experiments.

References

Steven Abney, David McAllester, and Fernando Pereira. 1999. Relating probabilistic grammars and automata. In *Proc. ACL*, pages 542–549.

Salvador Aguinaga, David Chiang, and Tim Weninger. 2019. Learning hyperedge replacement grammars for graph generation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 41(3):625–638.

Jean-Michel Autebert, Jean Berstel, and Luc Boasson. 1997. Context-free languages and pushdown automata. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, pages 111–174. Springer.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Comm. ACM*, 13(2):94–102.

Richard Futrell, Ethan Wilcox, Takashi Morita, Peng Qian, Miguel Ballesteros, and Roger Levy. 2019. Neural language models as psycholinguistic subjects: Representations of syntactic state. In *Proc. NAACL HLT*, pages 32–42.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proc. AISTATS*, pages 249–256.

Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics*, 25(4):573–605.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to transduce with unbounded memory. In *Proc. NeurIPS*, volume 2, pages 1828–1836.

Sheila A. Greibach. 1973. The hardest context-free language. *SIAM J. Comput.*, 2(4):304–310.

Yiding Hao, William Merrill, Dana Angluin, Robert Frank, Noah Amsel, Andrew Benz, and Simon Mendelsohn. 2018. Context-free transductions with neural stacks. In *Proc. BlackboxNLP*, pages 306–315.

Jennifer Hu, Jon Gauthier, Peng Qian, Ethan Wilcox, and Roger Levy. 2020. A systematic assessment of syntactic generalization in neural language models. In *Proc. ACL*, pages 1725–1744.

Armand Joulin and Tomas Mikolov. 2015. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Proc. NeurIPS*, volume 1, pages 190–198.

Diederik P. Kingma and Jimmy Lei Ba. 2015. Adam: A method for stochastic optimization. In *Proc. ICLR*.

Bernard Lang. 1974. Deterministic techniques for efficient non-deterministic parsers. In *Proc. Colloquium on Automata, Languages, and Programming*, pages 255–269.

Roger Levy. 2008. Expectation-based syntactic comprehension. *Cognition*, 106:1126–77.

Richard McCoy, Robert H. Frank, and Tal Linzen. 2020. Does syntax need to grow on trees? Sources of hierarchical inductive bias in sequence-to-sequence networks. *Trans. ACL*, 8:125–140.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Proc. NeurIPS*, pages 8024–8035.

Marten van Schijndel, Aaron Mueller, and Tal Linzen. 2019. Quantity doesn't buy quality syntax with neural language models. In *Proc. EMNLP-IJCNLP*, pages 5831–5837.

Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1):3–36.

Andreas Stolcke. 1995. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201.

G. Z. Sun, C. Lee Giles, H. H. Chen, and Y. C. Lee. 1995. The neural network pushdown automaton: Model, stack, and learning simulations. Technical Report UMIACS-TR-93-77 and CS-TR-3118, University of Maryland. Revised version.

Mirac Suzgun, Sebastian Gehrmann, Yonatan Belinkov, and Stuart M. Shieber. 2019. Memory-augmented recurrent neural networks can learn generalized Dyck languages. arXiv:1922.03329.

Masaru Tomita. 1987. An efficient augmented context-free parsing algorithm. *Computational Linguistics*, 13(1–2):31–46.

Ethan Wilcox, Roger Levy, and Richard Futrell. 2019. Hierarchical representation in neural language models: Suppression and recovery of expectations. In *Proc. BlackboxNLP*, pages 181–190.

Dani Yogatama, Yishu Miao, Gábor Melis, Wang Ling, Adhiguna Kuncoro, Chris Dyer, and Phil Blunsom. 2018. Memory architectures in recurrent neural network language models. In *Proc. ICLR*.

A The Hardest CFL

Greibach (1973) describes a CFL, L_0, which is the "hardest" CFL in the sense that an efficient parser for L_0 is also an efficient parser for any other CFL L. It is defined as follows. (We deviate from Greibach's original notation for the sake of clarity.) Every string in L_0 is of the following form:

$$\alpha_1 ; \alpha_2 ; \cdots \alpha_n ; \in L_0$$

that is, a sequence of strings α_i, each terminated by ;. No α_i can contain ;. Each α_i, in turn, is divided into three parts, separated by commas:

$$\alpha_i = x_i , y_i , z_i$$

The middle part, y_i, is a substring of a string in D_2. The brackets in y_i do not need to be balanced, but all of the y_i's concatenated must form a string in D_2, prefixed by \$. The catch is that x_i and z_i can be any sequence of bracket, comma, and \$ symbols, so it is impossible to tell, in a single α_i, where y_i begins and ends. A parser must nondeterministically guess where each y_i is, and cannot verify a guess until the end of the string is reached.

The design of L_0 is justified as follows. Suppose we have a parser for L_0 which, as part of its output, identifies the start and end of each y_i. Given a CFG G in Greibach normal form (GNF), we can adapt the parser for L_0 to parse $\mathcal{L}(G)$ by constructing a string homomorphism h, such that $w \in \mathcal{L}(G)$ iff $h(w) \in L_0$, and the concatenated y_i's encode a leftmost derivation of w under G.

The homomorphism h always exists and can be constructed from G as follows. Let the nonterminals of G be $V = \{A_1, \ldots, A_{|V|}\}$. Recall that in GNF, every rule is of the form $A_i \to a A_{j_1} \cdots A_{j_m}$ and S does not appear on any right-hand side. Define

$$\text{push}(A_i) = (\texttt{[}^i\texttt{(}$$

$$\text{pop}(A_i) = \begin{cases} \texttt{)]}^i\texttt{)} & A_i \neq S \\ \texttt{\$} & A_i = S. \end{cases}$$

We encode each rule of G as

$$\text{rule}(A_i \to a A_{j_1} \cdots A_{j_m}) = \text{pop}(A_i)\ \text{push}(A_{j_1}) \cdots \text{push}(A_{j_m}).$$

Finally, we can define h as

$$h(b) = \left(\bigodot_{(A \to b\gamma) \in G} \text{rule}(A \to b\gamma) \right) ;$$

where $\odot$ concatenates strings together delimited by commas. Then there is a valid string of y_i's iff there is a valid derivation of w with respect to G.

B PCFGs for Generating Data

We list here the production rules and weights for the PCFG used for each of our tasks. Let $f(\mu) = 1 - \frac{1}{\mu+1}$, which is the probability of failure associated with a negative binomial distribution with a mean of μ failures before one success. For a recursive PCFG rule, a probability of $f(\mu)$ results in an average of μ applications of the recursive rule.

B.1 Marked reversal

We set $\mu = 60$.

$$\begin{aligned} S &\to \texttt{0}S\texttt{0} \ / \ \tfrac{1}{2} f(\mu) \\ S &\to \texttt{1}S\texttt{1} \ / \ \tfrac{1}{2} f(\mu) \\ S &\to \texttt{\#} \ / \ 1 - f(\mu) \end{aligned}$$

B.2 Unmarked reversal

We set $\mu = 60$.

$$\begin{aligned} S &\to \texttt{0}S\texttt{0} \ / \ \tfrac{1}{2} f(\mu) \\ S &\to \texttt{1}S\texttt{1} \ / \ \tfrac{1}{2} f(\mu) \\ S &\to \epsilon \ / \ 1 - f(\mu) \end{aligned}$$

B.3 Padded reversal

Let μ_c be the mean length of the reversed content, and let μ_p be the mean padding length. We set $\mu_c = 60$ and $\mu_p = 30$.

$$\begin{aligned} S &\to \texttt{0}S\texttt{0} \ / \ \tfrac{1}{2} f(\mu_c) \\ S &\to \texttt{1}S\texttt{1} \ / \ \tfrac{1}{2} f(\mu_c) \\ S &\to T_0 \ / \ \tfrac{1}{2}(1 - f(\mu_c)) \\ S &\to T_1 \ / \ \tfrac{1}{2}(1 - f(\mu_c)) \\ T_0 &\to \texttt{0}T_0 \ / \ f(\mu_p) \\ T_0 &\to \epsilon \ / \ 1 - f(\mu_p) \\ T_1 &\to \texttt{1}T_1 \ / \ f(\mu_p) \\ T_1 &\to \epsilon \ / \ 1 - f(\mu_p) \end{aligned}$$

B.4 Dyck language

Let μ_s be the mean number of splits, and let μ_n be the mean nesting depth. We set $\mu_s = 1$ and $\mu_n = 40$.

$$\begin{aligned} S &\to ST \ / \ f(\mu_s) \\ S &\to T \ / \ 1 - f(\mu_s) \\ T &\to \texttt{(}S\texttt{)} \ / \ \tfrac{1}{2} f(\mu_n) \\ T &\to \texttt{[}S\texttt{]} \ / \ \tfrac{1}{2} f(\mu_n) \\ T &\to \texttt{()} \ / \ \tfrac{1}{2}(1 - f(\mu_n)) \\ T &\to \texttt{[]} \ / \ \tfrac{1}{2}(1 - f(\mu_n)) \end{aligned}$$

B.5 Hardest CFL

Let μ_c be the mean number of commas, μ_{sf} be the mean short filler length, μ_{lf} be the mean long filler length, p_s be the probability of a semicolon, μ_s be the mean number of bracket splits, and μ_n be the mean bracket nesting depth. We set $\mu_c = 0.5$, $\mu_{sf} = 0.5$, $\mu_{lf} = 2$, $p_s = 0.25$, $\mu_s = 1.5$, and $\mu_n = 3$.

$$
\begin{array}{ll}
S' \to R\$Q\,S\,L; & / \ 1 \\
L \to L',U & / \ 1 \\
L' \to ,VL' & / \ f(\mu_c) \\
L' \to \epsilon & / \ 1 - f(\mu_c) \\
R \to U,R' & / \ 1 \\
R' \to R'V, & / \ f(\mu_c) \\
R' \to \epsilon & / \ 1 - f(\mu_c) \\
U \to WU & / \ f(\mu_{sf}) \\
U \to \epsilon & / \ 1 - f(\mu_{sf}) \\
V \to WV & / \ f(\mu_{lf} - 1) \\
V \to W & / \ 1 - f(\mu_{lf} - 1) \\
W \to (& / \ 0.2 \\
W \to) & / \ 0.2 \\
W \to [& / \ 0.2 \\
W \to] & / \ 0.2 \\
W \to \$ & / \ 0.2 \\
Q \to L;R & / \ p_s \\
Q \to \epsilon & / \ 1 - p_s \\
S \to S\,Q\,T & / \ f(\mu_s) \\
S \to T & / \ 1 - f(\mu_s) \\
T \to (Q\,S\,Q) & / \ \frac{1}{2} f(\mu_n) \\
T \to [Q\,S\,Q] & / \ \frac{1}{2} f(\mu_n) \\
T \to (Q) & / \ \frac{1}{2}(1 - f(\mu_n)) \\
T \to [Q] & / \ \frac{1}{2}(1 - f(\mu_n))
\end{array}
$$

C Sampling Strings with Fixed Length from a PCFG

For practical reasons, we restrict strings we sample from PCFGs to those whose lengths lie within a certain interval, say $[\ell_{\min}, \ell_{\max}]$. The lengths of strings sampled randomly from PCFGs tend to have high variance, and we often want data sets to consist of strings of a certain length (e.g. longer strings in the test set than in the training set).

To do this, we first sample a length ℓ uniformly from $[\ell_{\min}, \ell_{\max}]$. Then we use an efficient dynamic programming algorithm to sample strings directly from the distribution of strings in the PCFG with length ℓ. This algorithm is adapted from an algorithm presented by Aguinaga et al. (2019) for sampling graphs of a specific size from a hyperedge replacement grammar.

The algorithm operates in two phases. The first (Algorithm 1) computes a table T such that every entry $T[A, \ell]$ contains the total probability of sampling a string from the PCFG with length ℓ. The second (Algorithm 2) uses T to randomly sample a string from the PCFG (using S as the nonterminal parameter X), restricted to those with a length of exactly ℓ.

Let nonterminals(β) be an ordered sequence consisting of the nonterminals in β. Let Compositions(ℓ, n) be a function that returns a (possibly empty) list of all compositions of ℓ that are of length n (that is, all ordered sequences of n positive integers that add up to ℓ).

Algorithm 1 Computing the probability table T

Require: G has no ϵ-rules or unary rules

1: **function** ComputeWeights(G, T, X, ℓ)
2: **for all** rules $X \to \beta \ / \ p$ in G **do**
3: $N \leftarrow$ nonterminals(β)
4: $\ell' = \ell - |\beta| + |N|$
5: **for** C in Compositions($\ell', |N|$) **do**
6: $t[\beta, C] \leftarrow p \times \prod_{i=1}^{|N|} T[N_i, C_i]$
7: **return** t
8: **function** ComputeTable(G, n)
9: **for** ℓ from 1 to n **do**
10: **for all** nonterminals X **do**
11: $t \leftarrow$ ComputeWeights(G, T, X, ℓ)
12: $T[X, \ell] = \sum_{\beta, C} t[\beta, C]$
13: **return** T

Because this algorithm only works on PCFGs that are free of ϵ-rules and unary rules, we automatically refactor our PCFGs to remove them before providing them to the algorithm.

Some of our PCFGs do not generate any strings for certain lengths, which is detected at line 3 of Algorithm 2. In this case, we restart the sampling procedure from the beginning. This means that the distribution we are effectively sampling from is as follows. Let $G(w)$ be the probability of w under PCFG G, and let $G(\ell)$ be the probability of all strings of length ℓ, that is,

$$G(\ell) = \sum_{w \text{ s.t. } |w| = \ell} G(w).$$

Algorithm 2 Sampling a string using T

Require: T is the output of ComputeTable(G, ℓ)

function SampleSized(G, T, X, ℓ)
  if $T[X, \ell] = 0$ **then**
    error
  $t \leftarrow$ ComputeWeights(G, T, X, ℓ)
  sample (β, C) with probability $\frac{t[\beta, C]}{T[X, \ell]}$
  $s \leftarrow \epsilon$
  $i \leftarrow 1$
  for j from 1 to $|\beta|$ **do**
    if β_j is a terminal **then**
      append β_j to s
    else
      $s' \leftarrow$ SampleSized(G, T, β_j, C_i)
      append s' to s
      $i \leftarrow i + 1$
  return s

Then the distribution we are sampling from is

$$p_{\text{sample}}(w) = \frac{1}{|\{\ell \mid G(\ell) > 0\}|} \frac{G(w)}{G(|w|)}.$$

When computing the lower-bound cross-entropy of the validation and test sets, we must compute $p_{\text{sample}}(w)$ for each string w. Finding $G(w)$ requires re-parsing w with respect to G and summing the probabilities of all valid parses using the Inside algorithm. We can look up the value of $G(|w|)$ in the table entry $T[S, |w|]$ produced in the sampling algorithm.