# Inside-Outside and Forward-Backward Algorithms Are Just Backprop (Tutorial Paper)

**Jason Eisner**
Department of Computer Science
Johns Hopkins University
jason@cs.jhu.edu

## Abstract

A probabilistic or weighted grammar implies a posterior probability distribution over possible parses of a given input sentence. One often needs to extract information from this distribution, by computing the expected counts (in the unknown parse) of various grammar rules, constituents, transitions, or states. This requires an algorithm such as inside-outside or forward-backward that is tailored to the grammar formalism. Conveniently, each such algorithm can be obtained by automatically differentiating an "inside" algorithm that merely computes the log-probability of the evidence (the sentence). This mechanical procedure produces correct and efficient code. As for any other instance of back-propagation, it can be carried out manually or by software. This pedagogical paper carefully spells out the construction and relates it to traditional and non-traditional views of these algorithms.

## 1 Introduction

The inside-outside algorithm (Baker, 1979) is a core method in natural language processing. Given a sentence, it computes the expected count of each possible grammatical substructure at each position in the sentence. Such expected counts are commonly used (1) to train grammar weights from data, (2) to select low-risk parses, and (3) as soft features that characterize sentence positions for other NLP tasks.

The algorithm can be derived directly but is generally perceived as tricky. This paper explains how it can be obtained simply and automatically by back-propagation—more precisely, by differentiating the inside algorithm. In the same way, the forward-backward algorithm (Baum, 1972) can be gotten by differentiating the backward algorithm.

Back-propagation is now widely known in the natural language processing and machine learning communities, thanks to the recent surge of interest in neural networks. Thus, it now seems useful to call attention to its role in some of NLP's core algorithms for structured prediction.

### 1.1 Why the connection matters

The connection is fundamental. However, in the present author's experience, it is not as widely known as it should be, even among experienced researchers in this area. Other pedagogical presentations treat the inside-outside algorithm as if it were *sui generis* within NLP, deriving it "directly" as a challenging dynamic programming method that sums over exponentially many parses. That treatment follows the original papers (Baker, 1979; Jelinek, 1985; see Lari and Young, 1991 for history). While certainly valuable, it ignores the point that the algorithm is working with a log-linear (exponential-family) distribution. *All* such distributions share the property that a certain gradient is a vector of expected feature counts. The inside-outside algorithm can be viewed as following a *standard* recipe—back-propagation—for computing this gradient.

That insight is practically useful when deriving new algorithms. The original inside algorithm applies to probabilistic context-free grammars in Chomsky Normal Form. However, *other* inside algorithms are frequently constructed for other parsing strategies or other grammar formalisms (see section 8 for examples). It is very handy that these can be algorithmically differentiated to obtain the corresponding inside-outside algorithms. The core of this paper (section 5) demonstrates by example how to do this *manually*, by working through the derivation of standard inside-outside. Alternatively, one can implement one's new inside algorithm using a software framework that supports *automatic* differentiation in a general-purpose programming language (see `www.autodiff.org`) or a neural network (e.g., Bergstra et al., 2010)—hopefully without too much overhead. Then the rest comes for free.

Note that we use the name "inside-outside" (or "forward-backward") to denote just an algorithm that computes certain expected counts. Such an algorithm runs an inside pass and then an outside pass, and then combines their results. The resulting counts are broadly useful, as we noted at the start of the paper (see section 4 for details). Thus, we use "inside-outside" narrowly to mean computing these counts. We do not use it to refer to the larger method that computes the counts repeatedly in order to iteratively reestimate grammar parameters: we call that method by its generic name, Expectation-Maximization (section 4).

### 1.2 Contents of the paper

After concisely stating the formal setting (section 2) and the inside algorithm (section 3), we discuss the expected counts, their uses, and their relation to the gradient (section 4). Finally, we show how to differentiate the inside algorithm to obtain the new algorithm that computes this gradient (section 5).

For readers who are using this paper to learn the algorithms, section 6 gives interpretations of the $\beta$ and $\alpha$ quantities that arise and relates them to the traditional dynamic programming presentation.

As a bonus, section 7 then offers a supplementary perspective. Here the inside algorithm is presented as normalizing any weighted parse forest and, further, converting it into a PCFG that can be sampled from. The inside-outside algorithm is then explained as computing this sampler's probabilities of hitting various anchored constituents and rules. Similarly, the backward algorithm can be regarded as normalizing a weighted "trellis" graph and, further, converting it into a non-stationary Markov model; the forward-backward algorithm computes hitting probabilities in this model.

Section 8 discusses other settings where the same approach can be applied, starting with the forward-backward algorithm for Hidden Markov Models. Two appendices work through some additional variants of the algorithms.

### 1.3 Related work

Other papers have also provided significant insight into this subject. In particular, Goodman (1998, 1999) unifies most parsing algorithms as semiring-weighted theorem proving, with discussion of both inside and outside computations. Klein and Manning (2001) regard the resulting proof forests—traditionally called parse forests—as weighted hypergraphs. Li and Eisner (2009) show how to compute various expectations and gradients over such hypergraphs, by techniques including the inside-outside algorithm, and clarify the "wonderful" connection between expected counts and gradients. Eisner et al. (2005, section 5) observe without details that for real-weighted proof systems, the expected counts of the axioms (in our setting, grammar rules) can be obtained by applying back-propagation. They detail two ways to apply back-propagation, noting inside-outside as an example.

Graphical models are like context-free grammars in that they also specify log-linear distributions over structures.[1] Darwiche (2003) shows how to compute marginal posteriors (i.e., expected counts) in a graphical model by the same technique given here.

## 2 Definitions and Notation

Assume a given alphabet $\Sigma$ of **terminal symbols** and a disjoint finite alphabet $\mathcal{N}$ of **nonterminal symbols** that includes the special symbol ROOT.

A **derivation** $T$ is a rooted, ordered tree whose leaves are labeled with elements of $\Sigma$ and whose internal nodes are labeled with elements of $\mathcal{N}$. We say that the internal node $t$ uses the **production rule** $A \to \sigma$ if $A \in \mathcal{N}$ is the label of $t$ and $\sigma \in (\Sigma \cup \mathcal{N})^*$ is the sequence of labels of its children (in order). We denote this rule by $T_t$.

In this paper, we focus mainly on derivations in **Chomsky Normal Form (CNF)**—those for which each rule $T_t$ has the form $A \to B\ C$ or $A \to w$ for some $A, B, C \in \mathcal{N}$ and $w \in \Sigma$. We write $\mathcal{R}$ for the set of all possible rules of these forms, and $\mathcal{R}[A]$ for the subset with $A$ to the left of the arrow. However, the following definitions generalize naturally to other choices of $\mathcal{R}$.

A **weighted context-free grammar (WCFG)** in Chomsky Normal Form is a function $\mathcal{G} : \mathcal{R} \to \mathbb{R}_{\geq 0}$. Thus, $\mathcal{G}$ assigns a **weight** to each CNF rule. We extend it to assign a weight to each CNF derivation, by

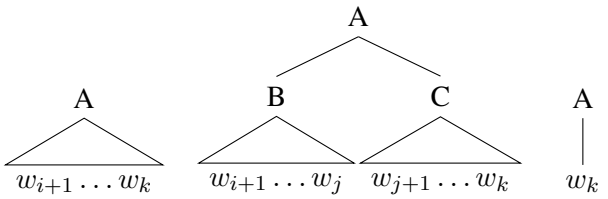
[1] Indeed, the two formalisms can be unified under a broader formalism such as case-factor diagrams (McAllester et al., 2004) or probabilistic programming (Sato, 1995; Sato and Kameya, 2008).

defining $\mathcal{G}(T) = \prod_{t \in T} \mathcal{G}(T_t)$, where $t$ ranges over the internal nodes of $T$.

A **probabilistic context-free grammar (PCFG)** is a WCFG $\mathcal{G}$ in which $(\forall A \in \mathcal{N}) \sum_{R \in \mathcal{R}[A]} \mathcal{G}(R) = 1$. In this case, $\mathcal{G}(T)$ is a probability measure over all derivations.[2]

The CNF derivation $T$ is called a **parse** of $\mathbf{w} \in \Sigma^*$ if ROOT is the label of its root and $\mathbf{w}$ is its **fringe**, i.e., the sequence of labels of its leaves. We refer to $\mathbf{w}$ as a **sentence** and denote its length by $n$; $\mathcal{T}(\mathbf{w})$ denotes the set of all parses of $\mathbf{w}$.

The triple $\langle A, i, k \rangle$ is mnemonically written as $A_i^k$ and pronounced as "$A$ from $i$ to $k$." We say that a parse of $\mathbf{w}$ uses the **anchored nonterminal** $A_i^k$ or the **anchored rule** $A_i^k \to B_i^j \; C_j^k$ or $A_{k-1}^k \to w_k$ if it contains the respective configurations




## 3 The Inside Algorithm

The inside algorithm (Algorithm 1) returns the *total* weight $Z$ of all parses of sentence $\mathbf{w}$ according to a WCFG $\mathcal{G}$. It is the natural extension to weighted CFGs of the CKY algorithm, a recognition algorithm for unweighted CFGs in Chomsky Normal Form (Kasami, 1965; Younger, 1967).

The importance of $Z$ is that a probability distribution over the parses $T \in \mathcal{T}(\mathbf{w})$ is given by

$$p(T \mid \mathbf{w}) \stackrel{\text{def}}{=} \mathcal{G}(T)/Z \tag{1}$$

When $\mathcal{G}$ is a PCFG representing a prior distribution on parses $T$, (1) is its posterior after observing the fringe $\mathbf{w}$. When $\mathcal{G}$ is a WCFG, (1) directly defines a conditional distribution on parses.

$Z$ is a sum of exponentially many products, since $|\mathcal{T}(\mathbf{w})|$ is exponential in $n = |\mathbf{w}|$. Fortunately, many of the sub-products are shared across multiple summands, and can be factored out using the distributive property. This strategy leads to the above polynomial-time dynamic programming algorithm, which interleaves sums and products.

[2]For this statement to hold even for "non-tight" PCFGs (Chi, 1999), we must consider the uncountable space of all finite *and infinite* derivations. That requires equipping this space with an appropriate $\sigma$-algebra and defining the measure $\mathcal{G}$ more precisely.

**Algorithm 1** The inside algorithm

```
1:  function INSIDE(G, w)
2:    initialize all β[···] to 0
3:    for k := 1 to n :                    ▷ width-1 constituents
4:      for A ∈ N :
5:        β[A_{k-1}^k] += G(A → w_k)
6:    for width := 2 to n :                ▷ wider constituents
7:      for i := 0 to n − width :          ▷ start point
8:        k := i + width                   ▷ end point
9:        for j := i + 1 to k − 1 :        ▷ midpoint
10:         for A, B, C ∈ N :
11:           β[A_i^k] += G(A → B C) β[B_i^j] β[C_j^k]
12:   return Z := β[ROOT_0^n]
```

Along the way, the inside algorithm computes useful intermediate quantities. Each **inner weight** $\beta[A_i^k]$ is the total weight of all derivations with root $A$ and fringe $w_{i+1} w_{i+2} \ldots w_k$. This implies the correctness of the return value, and is rather easy to establish by induction on the width $k - i$.

Note that if a parse contains any 0-weight rules, then that parse also has weight 0 and so does not contribute to $Z$. In effect, such rules and parses are excluded by $\mathcal{G}$. Such rules can in fact be skipped at lines 5 and 11, where they clearly have no effect. This further reduces runtime from $O(n^3 |\mathcal{N}|^3)$ to $O(n^3 |\mathcal{G}|)$, where $|\mathcal{G}|$ denotes the number of rules of nonzero weight.

## 4 Expected Counts and Derivatives

### 4.1 The goal of inside-outside

The inside-outside algorithm aims to extract useful information from the distribution (1). Given a sentence $\mathbf{w}$, it computes the **expected count** of each rule $R \in \mathcal{R}$ in a random parse $T$ drawn from that distribution:

$$c(R) \stackrel{\text{def}}{=} \sum_T \Big( p(T \mid \mathbf{w}) \sum_{t \in T} \delta(T_t = R) \Big) \tag{2}$$

For example, when $\mathcal{G}$ is a PCFG, $c(A \to B \; C)$ is the posterior expectation of the number of times that $A$ expanded as $B \; C$ while generating the sentence $\mathbf{w}$. Why are these expected counts useful? As Baker

(1979) saw, summing them over all observed sentences constitutes the E step within the Expectation-Maximization (EM) method (Dempster et al., 1977). EM adjusts the rule probabilities $\mathcal{G}$ to locally maximize likelihood (i.e., the probability of the observed sentences under $\mathcal{G}$).

### 4.2 The log-linear view

Of course, another way to locally maximize likelihood is to follow the gradient of log-likelihood. It has often been pointed out that EM is related to gradient ascent (e.g., Salakhutdinov et al., 2003; Berg-Kirkpatrick et al., 2010).

We now observe that (1) is an exponential-family model, implying a close relationship between expected counts and this gradient.

When we re-express the distribution (1) in the standard log-linear form, we see that its natural parameters are given by $\theta_R \stackrel{\text{def}}{=} \log \mathcal{G}(R)$ for $R \in \mathcal{R}$:

$$\begin{aligned} p(T \mid \mathbf{w}) &= \mathcal{G}(T)/Z \\ &= \frac{1}{Z} \prod_{t \in T} \mathcal{G}(T_t) = \frac{1}{Z} \exp \sum_{t \in T} \theta_{T_t} \\ &= \frac{1}{Z} \exp \sum_{R \in \mathcal{R}} \theta_R \cdot f_R(T) \end{aligned} \tag{3}$$

Here each $f_R$ is a feature function: $f_R(T) \in \mathbb{N}$ counts the occurrences of rule $R$ in parse $T$.

By standard properties of log-linear models, $\partial(\log Z)/\partial\theta_R$ equals the expectation of $f_R(T)$ under distribution (3). But the latter is precisely the expected count $c(R)$ that we desire. Thus, expected counts can be obtained as the gradient of $\log Z$. We will show in section 5 that this is precisely how the inside-outside algorithm operates.

### 4.3 Anchored probabilities

Along the way, the classical inside-outside algorithm finds the expected counts of the *anchored* rules. It finds $c(A \to B\ C)$ as $\sum_{i,j,k} c(A_i^k \to B_i^j\ C_j^k)$, where $c(A_i^k \to B_i^j\ C_j^k)$ denotes the expected count of that anchored rule, or equivalently, the expected number of times that $A \to B\ C$ is used at the particular position described by $i, j, k$. A simple extension (section 6.3) will find $c(A_i^k)$, the expected count of an anchored constituent.[3]

A CNF parse never uses a rule or constituent more than once at a given position, so an anchored expected count is always in $[0, 1]$. In fact, it is the *probability* that a random parse uses this anchored rule or anchored constituent.

These **anchored probabilities** are independently useful. They can be used in subsequent NLP tasks as **soft features** that characterize each portion of the sentence by its likely syntactic behavior. If $c(A_i^k) = 0.9$, then (according to $\mathcal{G}$) the substring $w_{i+1} \ldots w_k$ is probably a constituent of type $A$. If also $\sum_j c(A_i^k \to B_i^j\ C_j^k) = 0.75$, this $A$ probably splits into subconstituents $B$ and $C$.

Even for the parsing task itself, the anchored probabilities are useful for **decoding**—that is, selecting a single "best" parse tree $\hat{T}$. If the system will be rewarded for finding correct constituents, the *expected reward* of $\hat{T}$ is the sum of the anchored probabilities of the anchored constituents included in $\hat{T}$. The $\hat{T}$ that maximizes this sum[4] can be selected by a Viterbi-style algorithm, once all the anchored probabilities have been computed (Goodman, 1996; Matsuzaki et al., 2005).

## 5 Deriving the Inside-Outside Algorithm

### 5.1 Back-propagation

Section 4.2 showed that the expected counts can be obtained as the partial derivatives of $\log Z$. However, we will start by obtaining the partial derivatives of $Z$. This will lead to a more standard presentation of the inside-outside algorithm, exposing quantities such as the outer weights $\alpha$ that are both intuitive and useful.

The inside algorithm can be regarded as evaluating an **arithmetic circuit** that has many inputs $\{\mathcal{G}(R) : R \in \mathcal{R}\}$ and one output $Z$. Each non-input node of the circuit is a $\beta$ value, which is defined as a sum of products of certain other $\beta$ values and $\mathcal{G}$ values. The circuit's size and structure are determined by $\mathbf{w}$. The nested loops in Algorithm 1 simply iterate through the nodes of this circuit in a topologically sorted order, computing the value at each node.

Given any arithmetic circuit that represents a differentiable function $Z$, **automatic differentiation** extends it with a new **adjoint circuit** that computes

[3] A slower method is $c(A_i^k) = \sum_{B,C,j} c(A_i^k \to B_i^j\ C_j^k)$.

[4] An example of minimum Bayes risk decoding.

the gradient of $Z$—that is, the partial derivatives of the output $Z$ with respect to the inputs.

In the common case of **reverse-mode** automatic differentiation (Griewank and Corliss, 1991), the adjoint circuit employs a **back-propagation** strategy (Werbos, 1974).[5] For *each* node $x$ in the original circuit (not just the input nodes), the adjoint circuit includes an **adjoint node** $\eth x$ whose value is the partial derivative $\partial Z / \partial x$. Beginning with the obvious fact that $\eth Z = 1$, the adjoint circuit next computes the partials of $Z$ with respect to the nodes that directly influence $Z$, and then with respect to the nodes that influence those, gradually working back toward the inputs. Thus, the earlier $x$ is computed, the later $\eth x$ is computed.

### 5.2 Differentiating the inside algorithm

Back-propagation is popular for optimizing the parameters of neural networks, which involve nonlinear functions (LeCun, 1985; Rumelhart et al., 1986). We do not need to give a full presentation here, because the inside algorithm's circuit consists entirely of multiply-adds. In this simple case, automatic differentiation just augments each operation

$$x \mathrel{+}= y_1 \cdot y_2 \tag{4}$$

with the pair of adjoint operations

$$\eth y_1 \mathrel{+}= \eth x \cdot y_2 \tag{5}$$

$$\eth y_2 \mathrel{+}= y_1 \cdot \eth x \tag{6}$$

Intuitively, (5) recognizes that increasing $y_1$ by a small increment $\epsilon$ will increase $x$ by $\epsilon \cdot y_2$, which in turn increases $Z$ by $\eth x \cdot \epsilon \cdot y_2$. This suggests that $\eth y_1 = \eth x \cdot y_2$. However, increasing $y_1$ may affect $Z$ through more than just $x$, since $y_1$ may also feed into other equations like (4). Differentiability implies that the combined effect of these influences of $y_1$ on $Z$ is additive as $\epsilon \to 0$, accounting for the $+=$ in (5) (which is unrelated to the $+=$ in (4)).

This pattern in (4)–(6) extends to three-way products as well. Thus, the key step at line 11 of the inside algorithm,

$$\beta[A_i^k] \mathrel{+}= \mathcal{G}(A \to B\ C)\beta[B_i^j]\beta[C_j^k] \tag{7}$$

yields three adjoint summands

$$\eth\mathcal{G}(A \to B\ C) \mathrel{+}= \eth\beta[A_i^k]\beta[B_i^j]\beta[C_j^k] \tag{8}$$

$$\eth\beta[B_i^j] \mathrel{+}= \mathcal{G}(A \to B\ C)\eth\beta[A_i^k]\beta[C_j^k] \tag{9}$$

$$\eth\beta[C_j^k] \mathrel{+}= \mathcal{G}(A \to B\ C)\beta[B_i^j]\eth\beta[A_i^k] \tag{10}$$

Similarly, the initialization step in line 5,

$$\beta[A_{k-1}^k] \mathrel{+}= \mathcal{G}(A \to w_k) \tag{11}$$

yields the single (obvious) adjoint summand

$$\eth\mathcal{G}(A \to w_k) \mathrel{+}= \eth\beta[A_{k-1}^k] \tag{12}$$

Importantly, computing the adjoints increases the runtime by only a constant factor.

The adjoint of an inner weight, $\eth\beta[\cdots]$, corresponds to the traditional **outer weight**, written as $\alpha[\cdots]$. We will adopt this notation below. Furthermore, for consistency, we will write $\eth\mathcal{G}(R)$ as $\alpha[R]$—the "outer weight" of rule $R$.

### 5.3 The inside-outside algorithm

We need only one more move to derive the inside-outside algorithm. So far, we have obtained $\alpha[R] \stackrel{\text{def}}{=} \eth\mathcal{G}(R)$, the partial of $Z$ with respect to $\mathcal{G}(R) = \exp\theta_R$. We log-transform both $Z$ and $\mathcal{G}(R)$ to arrive finally at the expected count:

$$\begin{aligned} c(R) &= \frac{\partial \log Z}{\partial \theta_R} && \triangleright \textit{ from section 4.2} \\ &= \frac{\partial \log Z}{\partial Z} \cdot \frac{\partial Z}{\partial \mathcal{G}(R)} \cdot \frac{\partial \mathcal{G}(R)}{\partial \theta_R} && \triangleright \textit{ chain rule} \\ &= (1/Z) \cdot \alpha[R] \cdot \mathcal{G}(R) && (13) \end{aligned}$$

The final algorithm appears as Algorithm 2. This visits all of the inside nodes in a topologically sorted order by calling Algorithm 1, then visits all of their adjoints in a topologically sorted order (roughly the reverse), and finally applies (13).

## 6 Detailed Discussion

### 6.1 Relationship to the traditional version

An "off-the-shelf" application of automatic differentiation produces efficient code. Indeed, we would have gotten slightly more efficient code if we had applied the technique directly to the problem of section 4.2—finding the gradient of *log*-likelihood. This version is shown as Algorithm 3

[5] In cases like ours, where the original circuit has variable size, it is sometimes referred to as "backprop through structure" (Williams and Zipser, 1989; Goller and Küchler, 2005).

**Algorithm 2** The inside-outside algorithm

1: **procedure** INSIDE-OUTSIDE($\mathcal{G}$, $\mathbf{w}$)
2: $Z := \text{INSIDE}(\mathcal{G}, \mathbf{w})$ ▷ *side effect: sets* $\beta[\cdots]$
3: initialize all $\alpha[\cdots]$ to 0
4: $\alpha[\text{ROOT}_0^n]$ += 1 ▷ *sets* $\eth Z = 1$
5: **for** *width* := $n$ downto 2 : ▷ *wide to narrow*
6: **for** $i := 0$ to $n -$ *width* : ▷ *start point*
7: $k := i +$ *width* ▷ *end point*
8: **for** $j := i + 1$ to $k - 1$ : ▷ *midpoint*
9: **for** $A, B, C \in \mathcal{N}$ :
10: $\alpha[A \to B\ C]$ += $\alpha[A_i^k]\beta[B_i^j]\beta[C_j^k]$
11: $\alpha[B_i^j]$ += $\mathcal{G}(A \to B\ C)\alpha[A_i^k]\beta[C_j^k]$
12: $\alpha[C_j^k]$ += $\mathcal{G}(A \to B\ C)\beta[B_i^j]\alpha[A_i^k]$
13: **for** $k := 1$ to $n$ : ▷ *width-1 constituents*
14: **for** $A \in \mathcal{N}$ :
15: $\alpha[A \to w_k]$ += $\alpha[A_{k-1}^k]$
16: **for** $R \in \mathcal{R}$ : ▷ *expected rule counts*
17: $c(R) := \alpha[R] \cdot \mathcal{G}(R)/Z$

in Appendix A. It computes adjoint quantities of the form $\frac{\alpha}{Z}[x] = \partial(\log Z)/\partial x$ rather than $\alpha[x] = \partial Z/\partial x$. As a result, it divides *once* by $Z$ at line 4, whereas Algorithm 2 must do so *many times* at line 17 (to implement the correction in (13)).

Even Algorithm 2 is slightly more efficient than the traditional version (Lari and Young, 1990), thanks to the new quantity $\alpha[R]$. The traditional version (Algorithm 4) leaves out line 17, instead replacing lines 10 and 15 respectively with

$$c(A \to B\ C)\ \text{+=}\ \frac{\alpha[A_i^k]\mathcal{G}(A \to B\ C)\beta[B_i^j]\beta[C_j^k]}{Z} \quad (14)$$

$$c(A \to w_k)\ \text{+=}\ \frac{\alpha[A_{k-1}^k]\mathcal{G}(A \to w_k)}{Z} \quad (15)$$

Our automatically derived Algorithm 2 efficiently omits the common factor $\mathcal{G}(R)/Z$ from each summand above. It uses $\alpha[R]$ to accumulate the resulting intermediate sum (over positions of $R$), and finally multiplies the sum by $\mathcal{G}(R)/Z$ at line 17.[6]

### 6.2 Obtaining the anchored probabilities

If one wants the *anchored* rule probabilities discussed in section 4.3, they are precisely the summands in (14) and (15). To derive this fact via gradients, just revise our previous construction to use anchored rules $R' \in \mathcal{R}$. Extending the notation of section 4.2, let $f_{R'}(T) \in \{0, 1\}$ denote the count of $R'$ in parse $T$. We wish to find its expectation $c(R')$. Intuitively, this should equal $\partial(\log Z)/\partial\theta_{R'}$, which intuitively should work out to the summand in question via anchored versions of (8) and (13). This indeed holds if we revise (3) to use anchored features $f_{R'}$ of the tree $T$:

$$p(T \mid \mathbf{w}) = \frac{1}{Z} \exp \sum_{R' \in \mathcal{R}'} \theta_{R'} \cdot f_{R'}(T) \quad (16)$$

and then set $\theta_{R'} \stackrel{\text{def}}{=} \log \mathcal{G}(R)$ whenever $R'$ is an anchoring of $R$. Notice that (16) is a more expressive model than a WCFG, since it permits the same rule $R$ to have different weights at different positions—an idea we will revisit in section 8.4. However, here we take the gradient of its $Z$ only at the point in parameter space that corresponds to the actual WCFG $\mathcal{G}$—obtained by setting $\theta_{R'}$ to the same value, $\log G(R)$, for all anchorings $R'$ of $R$.

### 6.3 Interpreting the outer weights

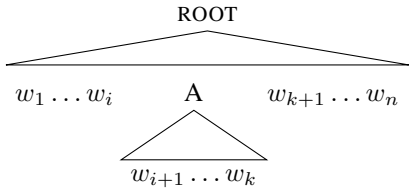
Section 5.3 exposes some interesting quantities. The outer weight of a *constituent* is $\alpha[A_i^k] = \eth\beta[A_i^k]$. The outer weight of a *rule*—novel to our presentation—is $\alpha[R] = \eth\mathcal{G}(R)$. We now connect these partial derivatives to a traditional view of the outer weights.

As we will see, $\beta[A_i^k] \cdot \eth\beta[A_i^k]$ (inner times outer) is the **total weight** of all parses that contain $A_i^k$. Then (1) implies that the total *probability* of these parses is $\beta[A_i^k] \cdot \eth\beta[A_i^k]/Z$. This gives the marginal probability of the anchored nonterminal, i.e., $c(A_i^k)$.

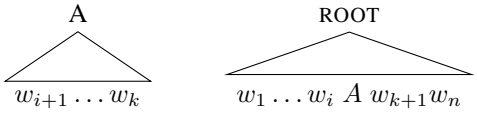
Analogously, $\mathcal{G}(R) \cdot \eth\mathcal{G}(R)$ is the total weight of all parses that contain $R$, where a parse is counted multiple times in this total if it contains $R$ multiple times. Dividing by $Z$ as before gives the expected count $c(R)$, which is indeed what line 17 does.

What does an outer weight signify on its own, and how does it help compute the total weight? Consider a parse $T$ that contains the anchored nonterminal $A_i^k$, so that $T$ has the form

[6] In practice, to avoid allocating separate memory for $\alpha[R]$, it can be stored in $c(R)$ and then multiplied in place by $\mathcal{G}(R)/Z$ at line 17 to obtain the true $c(R)$. Current automatic differentiation packages do not discover optimizations of this sort, as far as this author knows.




Its weight $\mathcal{G}(T)$ is the product of weights of all rules in the parse. This can be factored into an **inside product** that considers the rules dominated by $A_i^k$, times an **outside product** that considers all the other rules in $T$. That is, $T$ is obtained by combining an "inside derivation" with an incomplete "outside derivation," of the respective forms

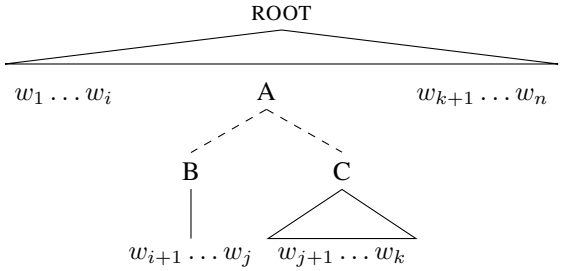



and the weight of $T$ is the product of their weights.

The many parses $T$ that contain $A_i^k$ can be obtained by pairwise combinations of inside and outside derivations of the forms shown. Thus—thanks to the distributive property—the total weight of these parses is $\beta[A_i^k] \cdot \alpha[A_i^k]$, where the inner weight $\beta[A_i^k]$ sums the inside product over all such inside derivations, and the outer weight $\alpha[A_i^k]$ sums the outside product over all such outside derivations. Both sums are accomplished in practice by dynamic programming recurrences that build from smaller to larger derivations. Namely, Algorithm 1 line 11 extends from smaller inside derivations to $\beta[A_i^k]$, while Algorithm 2 lines 11–12 extend from $\alpha[A_i^k]$ to larger outside derivations.

The previous paragraph is part of the traditional ("difficult") explanation of the inside-outside algorithm. However, it gives an alternative route to the gradient interpretation. It implies that $Z$ can be found as $\beta[A_i^k] \cdot \alpha[A_i^k]$ plus the weights of some other parses that do not involve $\beta[A_i^k]$. It follows that $\partial Z / \partial \beta[A_i^k]$ is indeed $\alpha[A_i^k]$.

Similarly, how about the outer weight and total weight *of a rule*? Consider a parse $T$ that contains rule $R$ at a particular position $i, j, k$. $\mathcal{G}(T)$ can be factored into the rule probability $\mathcal{G}(R)$—which can be regarded as an "inside product" with only one factor—times an "outside product" that considers all the other rule tokens in $T$. This decomposition helps us find the total weight as before. Each of the many instances of a parse $T$ with a token of $R$ marked within it can be obtained by combining $R$ with the incomplete "outside derivation" that lacks the token of $R$ at that position, which has the form




Summing the weights of these instances (a tree with $c$ copies of $R$ is added $c$ times), the distributive property implies the total is $\mathcal{G}(R) \cdot \alpha[R]$, where the outer weight $\alpha[R]$ sums over the "outside derivations." Algorithm 2 accumulates that sum at line 10 (for a binary rule as drawn above) or line 15 (unary rule).

We can verify from this fact that $\partial Z / \partial \mathcal{G}(R)$ is indeed $\alpha[R]$, by checking the effect on $Z$ of increasing $\mathcal{G}(R)$ slightly. Let us express $Z = \sum_{c=0}^{\infty} Z_c$, where $Z_c$ is the total weight of parses that contain exactly $c$ copies of $R$. Increasing $\mathcal{G}(R)$ by a *multiplicative* factor of $1 + \epsilon$ will increase $Z$ to $\sum_c (1+\epsilon)^c Z_c$, which $\approx \sum_c (1 + c\epsilon) Z_c$ for $\epsilon \approx 0$. Put another way, increasing $\mathcal{G}(R)$ by *adding* $\epsilon \mathcal{G}(R)$ results in increasing $Z$ by about $\sum_c c\epsilon Z_c$. Therefore (at least when $G(R) \neq 0$), we conclude $\partial Z / \partial G(R) = \left(\sum_c c Z_c\right) / \mathcal{G}(R) = (\mathcal{G}(R) \cdot \alpha[R]) / \mathcal{G}(R) = \alpha[R]$, since $\sum_c c Z_c$ is the total weight computed in the previous paragraph.

## 7 The Forest PCFG

For additional understanding, this section presents a different motivation for the inside algorithm—which then leads to an attractive independent derivation of the inside-outside algorithm.

### 7.1 Constructing the parse forest as a PCFG

The inner weights serve to enable the construction of a convenient representation—as a PCFG—of the distribution (1). Using a *grammar* to represent the packed forest of all parses of $\mathbf{w}$ was originally discussed by Bar-Hillel et al. (1961) and Billot and Lang (1989). The construction below Nederhof and Satta (2003) takes the weighted version of such a grammar and "renormalizes" it into a PCFG (Thompson, 1974; Abney et al., 1999; Chi, 1999).

This PCFG $\mathcal{G}'$ generates *only* parses of $\mathbf{w}$: that is, it assigns weight 0 to any derivation with fringe $\neq \mathbf{w}$. $\mathcal{G}'$ uses the original terminals $\Sigma$, but a different nonterminal set—namely the anchored nonterminals $\mathcal{N}' = \{A_i^k : A \in \mathcal{N}, 0 \leq i < k \leq n\}$, with root symbol $\text{ROOT}' = \text{ROOT}_0^n$. The CNF rules over these nonterminals are the anchored rules $\mathcal{R}'$. The function $\mathcal{G}' : \mathcal{R}' \to \mathbb{R}_{\geq 0}$ can now be defined in terms of the inner weights:

$$\mathcal{G}'(A_i^k \to B_i^j \; C_j^k) \stackrel{\text{def}}{=} \frac{\mathcal{G}(A \to B \; C) \cdot \beta[B_i^j] \cdot \beta[C_j^k]}{\beta[A_i^k]} \tag{17}$$

$$\mathcal{G}'(A_{k-1}^k \to w_k) \stackrel{\text{def}}{=} \frac{G(A \to w_k)}{\beta[A_{k-1}^k]} = 1 \tag{18}$$

for $A, B, C \in \mathcal{N}$ and $0 \leq i < j < k \leq n$, and $G'(\cdots) = 0$ otherwise. All trees $T$ with $\mathcal{G}'(T) > 0$ have fringe $\mathbf{w}$. Note that $|\mathcal{G}'| = O(n^3|\mathcal{G}|)$.

Thus, $\mathcal{G}'$ defines the weights of the rules in $\mathcal{R}'[A_i^k]$ to be the *relative* contributions made to $\beta[A_i^k]$ in Algorithm 1 by its summands.[7] This ensures that they sum to 1, making $\mathcal{G}'$ a PCFG.

### 7.2 Sampling from the forest PCFG

One use of $\mathcal{G}'$ is to enable easy sampling from (1), since PCFGs are *designed* to allow sampling (by a straightforward top-down recursive procedure). In effect, the top-down sampler recursively subdivides the total probability mass $Z$. For example, suppose that 60% of the total $Z = \beta[\text{ROOT}_0^n]$ was contributed by the various derivations in which the two child subtrees of ROOT have root labels $B, C$ and fringes $w_1 \ldots w_j, w_{j+1} \ldots w_n$. Then the first step of sampling from $\mathcal{G}'$ has a 60% chance of expanding $\text{ROOT}'$ into $B_0^j$ and $C_j^n$. If that happens, the sampler then recursively chooses how to expand those nonterminals according to *their* inner weights.

By thus sampling a tree $T'$ from $\mathcal{G}'$, and then simplifying each internal node label $A_i^k$ to $A$, we obtain a parse $T$ of $\mathbf{w}$, distributed according to $p(T \mid \mathbf{w})$ as desired. This method is equivalent to traditional presentations of sampling from $p(T \mid \mathbf{w})$ (Bod, 1995, p. 56; Goodman, 1998, section 4.4.1; Finkel et al., 2006). Our presentation merely exposes the construction of $\mathcal{G}'$ as an intermediate step.[8]

### 7.3 Expected counts from the forest PCFG

The above reparameterization of $p(T \mid \mathbf{w})$ as a PCFG $\mathcal{G}'$ makes it easier to find $c(A_i^k)$. The probability of finding $A_i^k$ in a parse must be the probability of encountering it when sampling a parse top-down from $\mathcal{G}'$ (the **hitting probability**).

Observe that the top-down sampling procedure starts at $\text{ROOT}'$. If it reaches $A_i^k$, it has probability $\mathcal{G}'(A_i^k \to B_i^j \; C_j^k)$ of reaching $B_i^j$ *as well as* $C_j^k$ on the next step.

Thus, the hitting probability $c(A_i^k)$ of an anchored nonterminal is the total probability of all "paths" from $\text{ROOT}'$ to $A_i^k$. To find all such totals, we initialize all $c(\cdots) = 0$ and then set $c(\text{ROOT}') = 1$. Once we know $c(A_i^k)$, we can *extend* those paths to its successor vertices, much as in the forward algorithm for HMMs.

Clearly, the probability that $c(A_i^k)$ expands using a particular anchored rule $R' \in \mathcal{R}'[A_i^k]$ during top-down sampling is

$$c(R') = c(A_i^k) \cdot \mathcal{G}'(R') \tag{19}$$

which we add into the expected count of the unanchored version $R$:

$$c(R) \; \texttt{+=} \; c(R') \tag{20}$$

This anchored rule hits the successors of $A_i^k$. E.g.:

$$c(B_i^j) \; \texttt{+=} \; c(A_i^k \to B_i^j \; C_j^k) \tag{21}$$

$$c(C_j^k) \; \texttt{+=} \; c(A_i^k \to B_i^j \; C_j^k) \tag{22}$$

This view leads to a correct algorithm that directly computes $c$ values without using $\alpha$ values at all. This Algorithm 5 looks exactly like Algorithm 2 or 3, except it uses the above lines that modify $c(\cdots)$ in place of the lines that modify $\alpha[\cdots]$ or $\frac{\alpha}{Z}[\cdots]$. We can in fact regard Algorithm 3 as simply the result of rearranging Algorithm 5 to avoid some of the multiplications and divisions needed to construct $\mathcal{G}'$, exploiting the fact that they cancel out as paths are extended.

[7] When $\beta[A_i^k] = 0$, these relative contributions are indeterminate (quotients are $0/0$). Then $\mathcal{G}'$ can specify *any* probability distribution over the rules $\mathcal{R}'[A_i^k]$. That distribution will never be used: the PCFG $\mathcal{G}'$ has probability 0 of reaching the anchored nonterminal $A_i^k$ and needing to expand it.

[8] Exposing $\mathcal{G}'$ can be computationally advantageous, in fact. After preprocessing a PCFG, samples can be drawn in time $O(n)$ per tree independent of the size of the grammar, by using alias sampling (Vose, 1991) to draw each rule in $O(1)$ time.

For example, in Algorithm 5, update (22) above expands via (19) and (17) into

$$c(C_j^k) \mathrel{+}= c(A_i^k) \cdot \frac{\mathcal{G}(A \to B\ C) \cdot \beta[B_i^j] \cdot \beta[C_j^k]}{\beta[A_i^k]} \quad (23)$$

Algorithm 3 rearranges this into

$$\frac{c(C_j^k)}{\beta[C_j^k]} \mathrel{+}= \frac{c(A_i^k)}{\beta[A_i^k]} \cdot \mathcal{G}(A \to B\ C) \cdot \beta[B_i^j] \quad (24)$$

and then systematically uses $\frac{\alpha}{Z}[x]$ to store $\frac{c(x)}{\beta[x]}$. It similarly transforms all other $c$ updates into $\frac{\alpha}{Z}$ updates as well. Algorithm 3 then recovers $c(x) := \frac{\alpha}{Z}[x] \cdot \beta[x]$ at the end of the algorithm, for all $x \in \mathcal{R}$, and could do the same for all $x \in \mathcal{R}'$ or $x \in \mathcal{N}'$.

# 8 Other Settings

Many types of weighted grammar are used in computational linguistics. Hence one often needs to construct new inside and inside-outside algorithms (Goodman, 1998, 1999). As examples, the weighted version of Earley's (1970) algorithm (Stolcke, 1995) handles arbitrary WCFGs (not restricted to CNF). Vijay-Shanker and Weir (1993) treat tree-adjoining grammar. Eisner (1996) handles projective dependency grammars. Smith and Smith (2007) and Koo et al. (2007) handle non-projective dependency grammars, using an inside algorithm with a different structure (not a dynamic programming algorithm).

In typical settings, each $T$ is a **derivation tree** (Vijay-Shankar et al., 1987)—a tree-structured recipe for assembling some syntactic description of the input sentence. The parse probability $p(T \mid \mathbf{w})$ takes the form $\frac{1}{Z} \prod_{t \in T} \exp \theta_t$, where $t$ ranges over certain configurations in $T$. $Z$ is the total weight of all $T$. Then the core insight of section 4.2 holds up: given an "inside" algorithm to compute $\log Z$, we can differentiate to obtain an "inside-outside" algorithm that computes $\nabla \log Z$, which yields up the expected counts of the configurations.

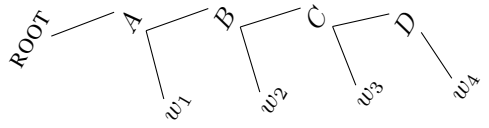
In this section we review several such settings. Parsing with a WCFG in CNF (Algorithms 1–2) is merely the simplest case that fully illustrates the "ins and outs," as it were.

In all these cases, the EM algorithm is not the only use of the expected counts: Sections 1 and 4.3 mentioned more important uses. EM itself is only useful for training a generative grammar, at best.[9]

[9]EM locally maximizes $\log Z$, which equals the likelihood $\log p(\mathbf{w})$ in the case of a generative grammar. Even in this case, EM may not be the best choice: once we are already computing $\nabla \log Z$, any continuous optimization algorithm (batch or online) can exploit that gradient to improve $\log Z$.

## 8.1 The forward-backward algorithm

A **Hidden Markov Model (HMM)** is essentially a simple PCFG. Its nonterminal symbols $\mathcal{N}$ are often called **states** or **tags**. The rule set $\mathcal{R}$ now consists not of CNF rules as in section 2, but all rules of the form ROOT $\to A$ or $A \to w\ B$ or $A \to w$ (for $A, B \in \mathcal{N}$ and $w \in \Sigma$). As a result, a parse of a length-4 sentence $\mathbf{w}$ must have the form




which is said to **tag** each word $w_j$ with its parent state. All parses share this right-branching tree structure, so they differ only in their choice of tags.

(Traditionally, the weight $\mathcal{G}(A \to w\ B)$ or $\mathcal{G}(A \to w)$ is defined as the product of an **emission probability**, $p(w \mid A)$, and a **transition probability**, $p(B \mid A)$ or $p(\text{HALT} \mid A)$. However, the following algorithms do not require this. Indeed, the algorithms do not require that $\mathcal{G}$ is a PCFG—any right-branching WCFG will do.)

In this setting, the inside algorithm (which computes $\beta$ bottom-up) is known as the **backward algorithm**, because it proceeds from right to left. The subsequent outside pass (which computes $\alpha$ top-down) is known as the **forward algorithm**.

Thanks to the fixed tree structure, this specialized version of inside-outside can run in total time of only $O(n|\mathcal{N}|^2)$. Of course, once we work out the fast backward algorithm (directly or from the inside algorithm), the forward-backward algorithm comes for free by algorithmic differentiation, with $\alpha[x]$ denoting $\eth x = \partial Z / \partial x$. Pseudocode appears as Algorithms 6–7 in Appendix B.

The forest of all taggings of $\mathbf{w}$ may be compactly represented as a directed acyclic graph—the **trellis**. The forward-backward algorithms can be nicely understood with reference to this trellis, shown as Figure 1 in Appendix B. Each maximal path in the trellis corresponds to a tagging; $\alpha$ and $\beta$ quantities sum the weights of prefix and suffix paths. $\mathcal{G}$ defines a probability distribution over these taggings

via (1). Following section 7, one can determine transition probabilities to the edges of the trellis so that a random walk on the trellis samples a tagging, from left to right, according to (1). The backward algorithm serves to compute $\beta$ values from which these transition probabilities are found (cf. section 7.1). Under these probabilities, the trellis becomes a nonstationary Markov model over the taggings. The forward pass now finds the hitting probabilities in this Markov model (cf. section 7.3), which describe how often the random walk will reach specific anchored nonterminals or traverse edges between them. These are the expected counts of tags and tag bigrams at specific positions.

### 8.2 Other grammar formalisms

Many grammar formalisms have weighted versions that produce exponential-family distributions over tree-structured derivations:

- PCFGs or WCFGs whose nonterminals are lexicalized or extended with other attributes, including unification-based grammars (Johnson et al., 1999)
- Categorial grammars, which use an unbounded set of nonterminals $\mathcal{N}$ (bounded for any given input sentence)
- Tree substitution grammars and tree adjoining grammars (Schabes, 1992), in which the derivation tree is distinct from the derived tree
- History-based stochasticizations such as the structured language model (Jelinek, 2004)
- Projective and non-projective dependency grammars as mentioned earlier
- Semi-Markov models for chunking (Sarawagi and Cohen, 2004), with runtime $O(n^2)$

Each formalism has one or more inside algorithms[10] that efficiently compute $Z$, typically in time $O(n^2)$ to $O(n^6)$ via dynamic programming. These inside algorithms can all be differentiated using the same recipe.

The trick continues to work when one of these inside algorithms is extended to the case of prefix parsing or lattice parsing (Nederhof and Satta, 2003), where the input sentence is not fully observed, or to partially supervised parsing (Pereira and Schabes, 1992; Matsuzaki et al., 2005), where the output tree is not fully *un*observed.

[10]Even basic WCFGs admit *multiple* algorithms—Earley's algorithm, unary cycle elimination, the "hook trick," and more (see Goodman, 1998; Eisner and Blatz, 2007).

### 8.3 Synchronous grammars

In a synchronous grammar (Shieber and Schabes, 1990), a parse $T$ is a derivation tree that produces aligned syntactic representations of a *pair* of sentences. These cases are handled as before. The input to the inside algorithm is a pair of aligned or unaligned sentences, or a single sentence.

The simplest synchronous grammar is a finite-state transducer. Here $T$ is an alignment of the two sentences, tagged with states. Eisner (2002) generalized the forward-backward algorithm to this setting and drew a connection to gradients.

### 8.4 Conditional grammars

In the conditional random field (CRF) approach, $p(T \mid \mathbf{w})$ is defined as $\mathcal{G}_{\mathbf{w}}(T)/Z$, where $\mathcal{G}_{\mathbf{w}}$ is a specialized grammar constructed given the input $\mathbf{w}$. Generally $\mathcal{G}_{\mathbf{w}}$ defines weights for the *anchored* rules $\mathcal{R}'$, allowing the weight of a rule to be position-specific. (This slightly affects lines 5 and 11 of Algorithm 1.) Any weighted grammar formalism may be used: e.g., the formalisms in section 2 and section 8.1 respectively yield the CRF-CFG (Finkel et al., 2008) and the popular linear-chain CRF (Sutton and McCallum, 2011).

Nothing in our approach changes. In fact, supervised training of a CRF usually follows the gradient $\nabla \log p(T^* \mid \mathbf{w})$. This equals the vector of rule counts observed in the supervised tree $T^*$, minus the expected rule counts $\nabla \log Z$—as equivalently computed by backprop *or* inside-outside.

### 8.5 Pruned, prioritized, and beam parsing

For speed, it is common in practice to perform only a subset of the updates at line 11 of Algorithm 1. This approximate algorithm computes an underestimate $\hat{Z}$ of $Z$ (via underestimates $\hat{\beta}[A_i^k]$), because only a subset of the inside circuit is used. By storing this dynamically determined smaller circuit and performing back-propagation through it (Eisner et al., 2005), we can compute the partial derivatives $\partial\hat{Z}/\partial\hat{\beta}[A_i^k]$ and $\partial\hat{Z}/\partial\mathcal{G}(R)$. These approximations may be used in all formulas in order to compute the expected counts of constituents and rules

in the **pruned parse forest**, which consists of the parse trees—with total weight $\hat{Z}$—explored by the approximate algorithm.

Many clever single-pass or multi-pass strategies exist for including the most important updates. Strategies are usually based either on direct pruning of constituents or prioritized updates with early stopping. A key insight is that $\beta[A_i^k] \cdot \eth\beta[A_i^k]$ is the total contribution of $\beta[A_i^k]$ to $Z$, so one should be sure to include the update $\beta[A_i^k]$ += $\Delta$ if $\Delta \cdot \alpha[A_i^k]$ is predicted to be large.

Weighted automata are popular for parsing, particularly dependency parsing (Nivre, 2003). In this case, $T$ is similar to a derivation tree: it is a sequence of operations (e.g., shift and reduce) that constructs a syntactic representation. Here an approximate $\hat{Z}$ is usually computed by beam search, and can be differentiated as above.

### 8.6 Inside-outside should be as fast as inside

In all cases, it is good practice to derive one's inside-outside algorithm by differentiation—manual or automatic—to ensure correctness and efficiency.

It should be a red flag if a proposed inside-outside algorithm is asymptotically slower than its inside algorithm.[11] Why? Because automatic differentiation produces an adjoint circuit that is at most twice the size of the original circuit (assuming binary operators). That means $\nabla \log Z$ always *can* be evaluated with the same asymptotic runtime as $\log Z$.

Good researchers do sometimes slip up and publish less efficient algorithms. Koo et al. (2007) present the $O(n^3)$ algorithms for non-projective dependency parsing: they point out that a contemporaneous IWPT paper with the same $O(n^3)$ inside algorithm had somehow raised inside-outside's runtime from $O(n^3)$ to $O(n^5)$. Similarly, Vieira et al. (2016) provide efficient algorithms for variable-order linear-chain CRFs, but note that a JMLR paper with the same forward algorithm had raised the grammar constant in forward-backward's runtime.

[11] Unless inside-outside has been *deliberately* slowed down to reduce its *space* requirements, as in the discard-and-recompute scheme of Zweig and Padmanabhan (2000). Absent such a scheme, inside-outside may need asymptotically more space than inside: though the adjoint circuit is not much bigger than the original, evaluating it may require keeping more of the original in memory at once (unless time is traded for space).

## 9 Conclusions

Computational linguists have been back-propagating through their arithmetic circuits for a long time without realizing it—indeed, since before Rumelhart et al. (1986) popularized the use of this technique to train neural networks. Recognizing this connection can help us to understand, teach, develop, and implement many core algorithms of the field.

## Acknowledgments

Thanks to anonymous referees and to Tim Vieira for useful comments that improved the paper.

## References

Steven Abney, David McAllester, and Fernando Pereira. Relating probabilistic grammars and automata. In *Proceedings of ACL*, pages 542–557, 1999.

J. K. Baker. Trainable grammars for speech recognition. In Jared J. Wolf and Dennis H. Klatt, editors, *Speech Communication Papers Presented at the 97th Meeting of the Acoustical Society of America*, MIT, Cambridge, MA, June 1979.

Yehoshua Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172, 1961. Reprinted in Y. Bar-Hillel. (1964). *Language and Information: Selected Essays on their Theory and Application*, Addison-Wesley 1964, 116–150.

L. E. Baum. An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process. *Inequalities*, 3, 1972.

Taylor Berg-Kirkpatrick, Alexandre Bouchard-Côté, DeNero, John DeNero, and Dan Klein. Painless unsupervised learning with features. In *Proceedings of NAACL*, June 2010.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math compiler in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 3–10, 2010.

S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of ACL*, pages 143–151, April 1989.

Rens Bod. *Enriching Linguistics with Statistics: Performance Models of Natural Language*. PhD thesis, University of Amsterdam, Academische Pers, Amsterdam, 1995. ILLC Dissertation Series 1995-14.

Zhiyi Chi. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25(1): 131–160, 1999.

Adnan Darwiche. A differential approach to inference in Bayesian networks. *Journal of the Association for Computing Machinery*, 50(3):280–305, 2003.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statist. Soc. Ser. B*, 39(1):1–38, 1977. With discussion.

J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

Jason Eisner. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pages 340–345, Copenhagen, August 1996.

Jason Eisner. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of ACL*, pages 1–8, 2002.

Jason Eisner and John Blatz. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proceedings of the 11th Conference on Formal Grammar*, pages 45–85, 2007.

Jason Eisner, Eric Goldlust, and Noah A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of HLT-EMNLP*, pages 281–290, 2005.

Jenny Rose Finkel, Christopher D. Manning, and Andrew Y. Ng. Solving the problem of cascading errors: Approximate Bayesian inference for linguistic annotation pipelines. In *Proceedings of EMNLP*, pages 618–626, 2006.

Jenny Rose Finkel, Alex Kleeman, and Christopher D. Manning. Efficient, feature-based, conditional random field parsing. In *Proceedings of ACL-08: HLT*, pages 959–967, June 2008.

Christoph Goller and Andreas Küchler. Learning task-dependent distributed representations by backpropagation through structure. Report AR-95-02, Fakultät Für Informatik, Technischen Universität München, 2005.

Joshua Goodman. Efficient algorithms for parsing the DOP model. In *Proceedings of EMNLP*, 1996.

Joshua Goodman. *Parsing Inside-Out*. PhD thesis, Harvard University, May 1998.

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

Andreas Griewank and George Corliss, editors. *Automatic Differentiation of Algorithms*. SIAM, Philadelphia, 1991.

Frederick Jelinek. Markov source modelling of test generation. In *NATO Advanced Study Institute: Impact of Processing Techniques on Communication*, pages 569–598. Martinus Nijhoff, 1985.

Frederick Jelinek. Stochastic analysis of structured language modeling. In Mark Johnson, Sanjeev P. Khudanpur, Mari Ostendorf, and Roni Rosenfeld, editors, *Mathematical Founations of Speech and Languge Procesing*, number 138 in The IMA Volumes in Mathematics and its Applications, pages 37–71. Springer, 2004.

Mark Johnson, Stuart Geman, Stephen Canon, Zhiyi Chi, and Stefan Riezler. Estimators for stochastic 'unification-based' grammars. In *Proceedings of ACL*, pages 535–549, University of Maryland, 1999.

Tadao Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical report, Air Force Cambridge Research Laboratory, 1965.

Dan Klein and Christopher D. Manning. Parsing and hypergraphs. In *Proceedings of the International Workshop on Parsing Technologies (IWPT)*, 2001.

Terry Koo, Amir Globerson, Xavier Carreras, and Michael Collins. Structured prediction models via the matrix-tree theorem. In *Proceedings of EMNLP-CoNLL*, pages 141–150, June 2007.

K. Lari and S. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.

K. Lari and S. Young. Applications of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 5:237–257, 1991.

Yann LeCun. Une procédure d'apprentissage pour réseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks). In *Proceedings of Cognitiva 85*, pages 599–604, Paris, France, 1985.

Zhifei Li and Jason Eisner. First- and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of EMNLP*, pages 40–51, 2009.

Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. Probabilistic CFG with latent annotations. In *Proceedings of ACL*, pages 75–82, 2005.

David McAllester, Michael Collins, and Fernando Pereira. Case-factor diagrams for structured probabilistic modeling. In *Proceedings of the Twentieth*

*Conference on Uncertainty in Artificial Intelligence (UAI)*, 2004.

Mehryar Mohri. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 324:177–201, March 2000.

Mark-Jan Nederhof and Giorgio Satta. Probabilistic parsing as intersection. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 137–148, April 2003.

Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 149–160, 2003.

Fernando Pereira and Yves Schabes. Inside-outside reestimation from partially bracketed corpora. In *Proceedings of ACL*, 1992.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. MIT Press, 1986.

Ruslan Salakhutdinov, Sam Roweis, and Zoubin Ghahramani. Optimization with EM and expectation-conjugate-gradient. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, Washington, DC, 2003.

Sunita Sarawagi and William W Cohen. Semi-Markov conditional random fields for information extraction. In *Proceedings of NIPS*, 2004.

Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of ICLP*, pages 715–729, 1995.

Taisuke Sato and Yoshitaka Kameya. New advances in logic-based probabilistic modeling by PRISM. In *Probabilistic Inductive Logic Programming*, pages 118–155. Sringer, 2008.

Yves Schabes. Stochastic lexicalized tree-adjoining grammars. In *Proceedings of COLING*, 1992.

Stuart Shieber and Yves Schabes. Synchronous tree-adjoining grammars. In *Proceedings of COLING*, 1990.

David A. Smith and Noah A. Smith. Probabilistic models of nonprojective dependency trees. In *Proceedings of EMNLP-CoNLL*, pages 132–140, 2007.

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

Charles Sutton and Andrew McCallum. An introduction to conditional random fields. *Foundations and Trends in Machine Learning*, 4(4):267–373, 2011.

Richard A. Thompson. Determination of probabilistic grammars for functionally specified probability-measure languages. *IEEE Transactions on Computers*, C-23(6):603–614, 1974.

Tim Vieira, Ryan Cotterell, and Jason Eisner. Speed-accuracy tradeoffs in tagging with variable-order CRFs and structured sparsity. In *Proceedings of EMNLP*, Austin, TX, November 2016.

K. Vijay-Shankar, David J. Weir, and Aravind K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *Proceedings of ACL*, pages 104–111, 1987.

K. Vijay-Shanker and David J. Weir. Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4):591–636, 1993.

Michael D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on Software Engineering*, 17(9):972–975, September 1991.

P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

D. H. Younger. Recognition and parsing of context-free languages in time $n^3$. *Information and Control*, 10(2): 189–208, February 1967.

G. Zweig and M. Padmanabhan. Exact alpha-beta computation in logarithmic space with application to map word graph construction. In *Proceedings of ICSLP*, 2000.

## A Pseudocode for Inside-Outside Variants

The core of this paper is Algorithms 1 and 2 in the main text. For easy reference, this appendix provides concrete pseudocode for some close variants of Algorithm 2 that are discussed in the main text, highlighting the differences from Algorithm 2. All of these variants compute the same expected counts $c$, with only small constant-factor differences in efficiency.

Algorithm 3 is the clean, efficient version of inside-outside that obtains the expected counts $\nabla \log Z$ by direct implementation of backprop. We may regard this as the fundamental form of the algorithm. The differences from Algorithm 2 are highlighted in red and discussed in section 6.1.

Since $\log Z$ is replacing $Z$ as the output being differentiated, the adjoint quantity $\eth x$ is redefined as $\partial(\log Z)/\partial x$ and stored in $\frac{\alpha}{Z}[x]$. The name $\frac{\alpha}{Z}$ is chosen because it is $\frac{1}{Z}$ times the traditional $\alpha$. The computations in the loops are not affected by this rescaling: they perform the same operations as in Algorithm 2. Only the start and end of the algorithm are different (lines 4 and 17).

To emphasize the pattern of reverse-mode automatic differentiation, Algorithm 3 takes care to compute the adjoint quantities in exactly the reverse of the order in which Algorithm 1 computed the original quantities. The resulting iteration order in the $i$, $j$, and $k$ loops is highlighted in blue. Computing adjoints in reverse order always works and can be regarded as the default strategy. However, this is merely a cosmetic change: the version in Algorithm 2 is just as valid, because it too visits the nodes of the adjoint circuit in a topologically sorted order. Indeed, since each of these blue loops is parallelizable, it is clear that the order cannot matter. What is crucial is that the overall narrow-to-wide order of Algorithm 1, which is *not* parallelizable, is reversed by both Algorithm 2 and Algorithm 3.

Algorithm 4 is a more traditional version of Algorithm 2. The differences from Algorithm 2 are highlighted in red and discussed in section 6.1.

Finally, Algorithm 5 is a version that is derived on independent principles (section 7.3). It directly

**Algorithm 3** A cleaner variant of the inside-outside algorithm

1: **procedure** INSIDE-OUTSIDE($\mathcal{G}$, $\mathbf{w}$)
2: $Z :=$ INSIDE($\mathcal{G}, \mathbf{w}$) ▷ *side effect: sets* $\beta[\cdots]$
3: initialize all $\alpha[\cdots]$ to 0
4: $\frac{\alpha}{Z}[\text{ROOT}_0^n]$ += $\frac{1}{Z}$ ▷ *sets* $\eth Z = 1/Z$
5: **for** *width* $:= n$ downto 2 : ▷ *wide to narrow*
6: **for** $i := n - \mathit{width}$ downto 0 : ▷ *start point*
7: $k := i + \mathit{width}$ ▷ *end point*
8: **for** $j := k - 1$ downto $i + 1$ : ▷ *midpoint*
9: **for** $A, B, C \in \mathcal{N}$ :
10: $\frac{\alpha}{Z}[A \to B\ C]$ += $\frac{\alpha}{Z}[A_i^k]\beta[B_i^j]\beta[C_j^k]$
11: $\frac{\alpha}{Z}[B_i^j]$ += $\mathcal{G}(A \to B\ C)\frac{\alpha}{Z}[A_i^k]\beta[C_j^k]$
12: $\frac{\alpha}{Z}[C_j^k]$ += $\mathcal{G}(A \to B\ C)\beta[B_i^j]\frac{\alpha}{Z}[A_i^k]$
13: **for** $k := n$ downto 1 : ▷ *width-1 constituents*
14: **for** $A \in \mathcal{N}$ : p
15: $\frac{\alpha}{Z}[A \to w_k]$ += $\frac{\alpha}{Z}[A_{k-1}^k]$
16: **for** $R \in \mathcal{R}$ : ▷ *expected rule counts*
17: $c(R) := \frac{\alpha}{Z}[R] \cdot \mathcal{G}(R)$ ▷ *no division by Z*

**Algorithm 4** A more traditional variant of the inside-outside algorithm

1: **procedure** INSIDE-OUTSIDE($\mathcal{G}$, $\mathbf{w}$)
2: $Z :=$ INSIDE($\mathcal{G}, \mathbf{w}$) ▷ *side effect: sets* $\beta[\cdots]$
3: initialize all $\alpha[\cdots]$ to 0
4: $\alpha[\text{ROOT}_0^n]$ += 1 ▷ *sets* $\eth Z = 1$
5: **for** *width* $:= n$ downto 2 : ▷ *wide to narrow*
6: **for** $i := 0$ to $n - \mathit{width}$ : ▷ *start point*
7: $k := i + \mathit{width}$ ▷ *end point*
8: **for** $j := i + 1$ to $k - 1$ : ▷ *midpoint*
9: **for** $A, B, C \in \mathcal{N}$ :
10: $c(A \to B\ C)$ += $\frac{\alpha[A_i^k]\mathcal{G}(A \to B\ C)\beta[B_i^j]\beta[C_j^k]}{Z}$
11: $\alpha[B_i^j]$ += $\mathcal{G}(A \to B\ C)\alpha[A_i^k]\beta[C_j^k]$
12: $\alpha[C_j^k]$ += $\mathcal{G}(A \to B\ C)\beta[B_i^j]\alpha[A_i^k]$
13: **for** $k := 1$ to $n$ : ▷ *width-1 constituents*
14: **for** $A \in \mathcal{N}$ :
15: $c(A \to w_k)$ += $\frac{\alpha[A_{k-1}^k]\mathcal{G}(A \to w_k)}{Z}$
16: **for** $R \in \mathcal{R}$ :
17: do nothing ▷ $c(R)$ *has already been computed*

computes the hitting probabilities of anchored constituents and rules in a PCFG representation of the parse forest. This may be regarded as the most natural way to obtain the algorithm without using gradients. However, as section 7.3 notes, it can be fairly easily rearranged into Algorithm 3, which is slightly more efficient. The differences from Algorithms 2 and 3 are highlighted in red.

To rearrange Algorithm 5 into Algorithm 3, the key is to compute not the count $c(x)$, but the ratio $\frac{c(x)}{\beta[x]}$ (or $\frac{c(x)}{\mathcal{G}(x)}$ when $x$ is a rule), storing this ratio in the variable $\frac{\alpha}{Z}[x]$. This ratio $\frac{\alpha}{Z}[x]$ can be interpreted as $\partial(\log Z)/\partial x$ as previously discussed.

**Algorithm 5** An inside-outside variant motivated as finding hitting probabilities

**procedure** INSIDE-OUTSIDE($\mathcal{G}$, $\mathbf{w}$)
  $Z :=$ INSIDE($\mathcal{G}, \mathbf{w}$) ▷ *side effect: sets* $\beta[\cdots]$
  initialize all $c[\cdots]$ to 0
  $c(\text{ROOT}_0^n)$ += 1
  **for** *width* $:= n$ downto 2 : ▷ *wide to narrow*
    **for** $i := 0$ to $n -$ *width* : ▷ *start point*
      $k := i +$ *width* ▷ *end point*
      **for** $j := i + 1$ to $k - 1$ : ▷ *midpoint*
        **for** $A, B, C \in \mathcal{N}$ :
          $c(A_i^k \to B_i^j\ C_j^k) :=$ ▷ *eqs.* (19), (17)
            $c(A_i^k) \cdot \frac{\mathcal{G}(A \to B\ C) \cdot \beta[B_i^j] \cdot \beta[C_j^k]}{\beta[A_i^k]}$
          $c(A \to B\ C)$ += $c(A_i^k \to B_i^j\ C_j^k)$
          $c(B_i^j)$ += $c(A_i^k \to B_i^j\ C_j^k)$
          $c(C_j^k)$ += $c(A_i^k \to B_i^j\ C_j^k)$
  **for** $k := 1$ to $n$ : ▷ *width-1 constituents*
    **for** $A \in \mathcal{N}$ :
      $c(A_{k-1}^k \to w_k) := c(A_{k-1}^k)$ ▷ *eqs.* (19),(18)
      $c(A \to w_k)$ += $c(A_{k-1}^k \to w_k)$
  **for** $R \in \mathcal{R}$ : ▷ *expected rule counts*
    do nothing ▷ $c(R)$ *has already been computed*

## B Pseudocode for Forward-Backward

Algorithm 6 is the backward algorithm, as introduced in section 8.1. It is an efficient specialization of the inside algorithm (Algorithm 1) to right-branching trees.

Notation: We use $\text{ROOT}^0$ to denote the root anchored nonterminal, and $A^j$ (for $A \in \mathcal{N}$ and $1 \le j \le n$) to denote an anchored nonterminal $A$ that serves as the tag of $w_j$. That is, $A_j$ is anchored so that its left child is $w_j$. (Since this $A^j$ actually dominates all of $w_j \ldots w_n$ in the right-branching tree, it would be called $A_{j-1}^n$ if we were running the full inside algorithm.)

Line 7 builds up the right-branching tree by combining a word from $j-1$ to $j$ (namely $w_j$) with a phrase from $j$ to $n$ (namely $B^{j+1}$). This line is a specialization of line 11 in Algorithm 1, which combines a phrase from $i$ to $j$ with another phrase from $j$ to $k$. Thanks to the right-branching constraint, the backward algorithm only has to loop over $O(n)$ triples of the form $(j-1, j, n)$ (with fixed $n$)—whereas the inside algorithm must loop over $O(n^3)$ triples of the form $(i, j, k)$.

**Algorithm 6** The backward algorithm

**function** BACKWARD($\mathcal{G}$, $\mathbf{w}$)
  initialize all $\beta[\cdots]$ to 0
  **for** $A \in \mathcal{N}$ : ▷ *stopping rules*
    $\beta[A^n]$ += $\mathcal{G}(A \to w_n)$
  **for** $j := n - 1$ downto 1 :
    **for** $A, B \in \mathcal{N}$ : ▷ *transition rules*
      $\beta[A^j]$ += $\mathcal{G}(A \to w_j\ B)\ \beta[B^{j+1}]$
  **for** $A \in \mathcal{N}$ : ▷ *starting rules*
    $\beta[\text{ROOT}^0]$ += $\mathcal{G}(\text{ROOT} \to A)\ \beta[A^1]$
  **return** $Z := \beta[\text{ROOT}^0]$

The forward-backward algorithm, Algorithm 7, is derived mechanically by differentiating Algorithm 6, by exactly the same procedure as in section 5. As a result, it is a specialization of Algorithm 2.

This presentation of the forward-backward algorithm finds the expected counts of rules $R \in \mathcal{R}$. However, section 8.1 mentions that each rule $R$ can be regarded as consisting of an emission action $R_\text{e}$ followed by a transition action $R_\text{t}$. We may want to find the expected counts of the various actions. These can of course be found by summing the expected counts of all rules containing a given action. However, this step can also be handled naturally by backprop, in the common case where each $\mathcal{G}(R)$ is defined as a product $p_{R_\text{e}} \cdot p_{R_\text{t}}$ of the conditional probabilities of the emission and transition. In this case, $\theta_R = \log \mathcal{G}(R)$ from section 4.2 can be re-expressed

**Algorithm 7** The forward-backward algorithm

**procedure** FORWARD-BACKWARD($\mathcal{G}$, $\mathbf{w}$)
  $Z :=$ BACKWARD($\mathcal{G}, \mathbf{w}$) ▷ *also sets* $\beta[\cdots]$
  initialize all $\alpha[\cdots]$ to 0
  $\alpha[\text{ROOT}^0]$ += 1 ▷ *sets* $\eth Z = 1$
  **for** $A \in \mathcal{N}$ : ▷ *starting rules*
    $\alpha[\text{ROOT} \to A]$ += $\alpha[\text{ROOT}^0]\ \beta[A^1]$
    $\alpha[A^1]$ += $\alpha[\text{ROOT}^0]\ \mathcal{G}(\text{ROOT} \to A)$
  **for** $j := 1$ to $n-1$ :
    **for** $A, B \in \mathcal{N}$ : ▷ *transition rules*
      $\alpha[A \to w_j\ B]$ += $\alpha[A^j]\ \beta[B^{j+1}]$
      $\alpha[B^{j+1}]$ += $\alpha[A^j]\ \mathcal{G}(A \to w_j\ B)$
  **for** $A \in \mathcal{N}$ : ▷ *stopping rules*
    $\mathcal{G}(A \to w_n)$ += $\alpha[A^n]$
  **for** $R \in \mathcal{R}$ : ▷ *expected rule counts*
    $c(R) := \alpha[R] \cdot \mathcal{G}(R)/Z$

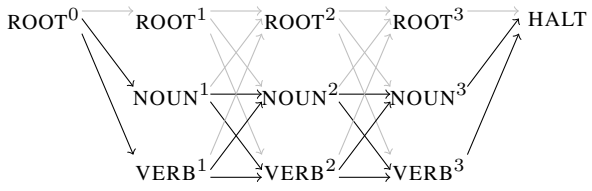



**Figure 1:** The trellis of taggings of a length-3 sentence, under an HMM where $\mathcal{N} = \{\text{ROOT}, \text{NOUN}, \text{VERB}\}$. (Although the trellis shows that ROOT may be used as an ordinary tag, often in practice it is allowed only at the root. This can be arranged by giving weight 0 to rules involving $\text{ROOT}^j$ for $j > 0$, corresponding to the gray edges.)

as the sum of two parameters, $\theta_{R_e} + \theta_{R_t}$, which represent the logs of these conditional probabilities. Then the expected emission and transition counts are given by $\partial(\log Z)/\partial\theta_{R_e}$ and $\partial(\log Z)/\partial\theta_{R_t}$.

It is traditional to view the forward-backward algorithm as running over a "trellis" of taggings (Figure 1), which represents the forest of parses. Since a nonterminal $A^j$ that is anchored at position $j$ necessarily emits $w_j$, the trellis representation does not bother to show the emissions. It is simply a directed graph showing the transitions. Every parse (tagging) of $\mathbf{w}$ corresponds to a path in Figure 1. Specifically, edge $A^j \to B^{j+1}$ in the trellis represents the anchored rule $A^j \to w_j\ B^{j+1}$, without showing $w_j$. Similarly, $A^n \to$ HALT represents the anchored rule $A^n \to w_n$, without showing $w_n$, and ROOT $\to A^1$ represents the anchored rule ROOT $\to A^1$. The weight of a trellis edge corresponding to an anchoring of rule $R$ is given by $\mathcal{G}(R)$. The weight $\mathcal{G}(T)$ of a tagging $T$ is then the product weight of the path that corresponds to that tagging.

On this view, the inner weight $\beta[A^j]$ can be regarded as a **suffix weight**: it sums up the weight of all paths from $A^j$ to HALT. Algorithm 6 can be transparently viewed as computing all suffix weights from right to left by dynamic programming. $Z = \text{ROOT}^0$ sums the weight of all paths from $\text{ROOT}^0$ to HALT. Similarly, the outer weight $\alpha[A^j]$ can be regarded as a **prefix weight**, computed symmetrically within Algorithm 7.

The constructions of section 7 are easier to understand in this setting. Here is the interpretation. It is possible to replace the non-negative *weights* on the trellis edges with *probabilities*, in such a way that the product weight of each path is not changed. Indeed, the method is essentially identical to the "weight pushing" algorithm for weighted finite-state automata (Mohri, 2000).

The probabilistic version of the trellis is a representation of a new weighted grammar $\mathcal{G}'$—an HMM (hence a type of PCFG) that generates only taggings of $\mathbf{w}$, with the probabilities given by (1).

In the probabilistic version of the trellis, the edges from a node have total probability of 1. Thus it is the graph of a Markov chain, whose states are the anchored nonterminals $\mathcal{N}'$. Sampling a tagging of $\mathbf{w}$ is now as simple as taking a random walk from $\text{ROOT}^0$ until HALT is reached. The forward pass can be interpreted as a straightforward use of dynamic programming to compute the hitting probabilities of the nodes in the trellis, as well as the probabilities of traversing a node's out-edges once the node is hit.

But how were the trellis probabilities found in the first place? The edge $A^j \to B^{j+1}$ originally had weight $\mathcal{G}(A \to w_j\ B)$. In the probabilistic version of the trellis, it has probability $\frac{\mathcal{G}(A \to w_j\ B) \cdot \beta[B^{j+1}]}{\beta[A^j]}$. This represents the total weight of paths from $A^j$ that start with this edge, as a fraction of the total weight $\beta[A^j]$ of all paths from $A^j$. (The edges involving $\text{ROOT}^0$ and HALT edges are handled similarly.) Computing the necessary $\beta$ weights to determine these probabilities is the essential function of the backward algorithm.