# LING 185B Final Paper

Angela Yang

June 2024

## 1 Abstract

The forward-backward algorithm is an procedure used to find the expected counts of rules given weighted context-free grammar and a sentence to parse. This algorithm is used to learn the most plausible parses, optimize weights, and provide "soft counts" of rules to expect. This algorithm is usually derived manually to fit the needs of certain kinds of models. This paper aims to prove that the forward-backward algorithm can be distilled into an instance of back-propagation on a set of input rules.

## 2 Introduction

The inside-outside algorithm is a method used to compute the expected counts of grammatical infixes at each position in the sentence. The forward-backward algorithm is a simpler form of the inside-outside algorithm which computes the expected counts of the infixes that are suffixes. As Eisner postulates, both the inside-outside and forward-backward algorithm can be derived more simply by generalizing them as variations of backprop. Where Eisner has demonstrated this with the inside-outside algorithm, I will be extending the example to forward-backward algorithm, which can be derived from the backward algorithm (Baum, 1972).

In the field of machine learning, back-propagation is a relatively fast algorithm used to compute the gradient of a cost function. It's famously a core component of neural network learning; in essence, back-propagation allows for the cost or error of the model's results to adjust the nodes' weights relative to its influence. Where we had used the EM algorithm in Homework 2 to converge on optimal weights that accounted for sentences in the language using the expected counts in the E step, an alternative method of optimization can be found with back-propagation.

## 3 Definitions and Notation

I will be largely aligning with Eisner's notations, as my goal is to extend the concepts analogously to the backward algorithm. So, instead of using the typical model of Hidden

Markov Models directly, I'll be drawing its parallels in a simpler version of CFGs.

Assume a given alphabet $\Sigma$ of **emissions** (which are analogous to terminal symbols in CFGs) and a disjoint finite alphabet $\mathcal{N}$ of **states** that includes the special symbol ROOT, which decomposes into the first emission and the starting state.

A **derivation** $T$ is a rooted, ordered tree whose leaves are members of $\Sigma$ and whose non-leaf nodes are members of $\mathcal{N}$. The symbol $w$ represents the sentence that we are trying to find the derivation for. What Eisner refers to as a **production rule** is a superset of a transition and state-emission pair in the backward algorithm. In the context of the grammar tree, a production rule would denote a rule that results in a branch or the last emission. Formally, we will represent this as $\mathcal{R}$ which consist of all rules of the form $\text{ROOT} \to A$ or $A \to wB$ or $A \to w$ (for $A, B \in \mathcal{N}$ and $w \in \Sigma$).

A **weighted context-free grammar (WCFG)** can be shown as a function $\mathcal{G} : \mathcal{R} \to \mathbb{R}_{\geq 0}$. Each rule in the grammar has a weight. We take the product of all rules applied to derive the weight of the particular derivation: $\mathcal{G}(T) = \Pi_{t \in T} \mathcal{G}(T_t)$, where $t$ ranges over all internal nodes of $T$. If the weight represents a probability from 0 to 1, this is a **probabilistic context-free grammar (PCFG)**.

Lastly, there is the notation of an **anchored nonterminal** $A^i$ or **anchored rule** $\text{ROOT} \to A^i$ or $A^i \to wB^{i+1}$ or $A^i \to w$, where the superscript denotes the position in $w$ where the backward traversal begins.

# 4 The Backward Algorithm

The backward algorithm returns the total weight $Z$ of all parses of $w$ with grammar $\mathcal{G}$. It is a special case of the inside algorithm in a right-branching WCFG. The pseudocode for this algorithm is shown in Algorithm 1.

**Algorithm 1** The Backward Algorithm

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

To begin to digest this algorithm to its core components, we begin with why $Z$ is significant. As previously stated, $Z$ is the total weight of all parses given a $w$ and $\mathcal{G}$. The probability of each parse $T$, which is a particular tree structure, given $w$ can be defined to be equal to the following:

$$p(T|w) := \mathcal{G}(T)/Z \tag{1}$$

Since $\mathcal{G}(T)$ can be calculated as the product of all rule instances' weights of a given tree, and the backward algorithm provides $Z$, we can calculate the probability of a given parse.

The three for-loops calculate the probability of the stopping rules, transition rules,

and starting rules respectively. Each node's backwards value $\beta[A^j]$ contains the recursive call to the backwards value of the suffix $\beta[B^j + 1]$. One can conceptualize the interleaved sums and products as a sort of Disjunctive Normal Form, in which a particular parse would have rules multiplied together when all nodes are required, and thus their weights are ANDed together, and each parse is a possibility in a massive OR statement, so their weights are added together.

# 5 Expected Counts and Derivatives

## 5.1 The Goal of Forward-Backward

The forward-backward algorithm's objective is to derive the expected counts of each rule across all parses $T$ using the distribution found in the previous section (1). We utilized this function in Homework 2 to calculate the E step in the Expectation Maximization (EM) algorithm made to optimize probabilities. The M step then took the expected counts and calculated the new probabilities based on the proportion of rules relative to one another.

## 5.2 Anchored Probabilities

The forward-backward algorithm is a recursive process which finds the expected counts of rules at specific positions in $w$. Such rules are called **anchored rules**, and are denoted $A^i$ where $i$ is the position and $A$ is the nonterminal which generates the suffix beginning at that position. As Eisner points out, CFGs never use a particular constituent more than once in a particular position so the expected count for $A^i$ is always either 0 or 1.

## 5.3 Log-Linear Distribution

There is another way to find the optimal weights of a weighted context-free-grammar using the gradient of log-likelihood. Starting from equation 1, we can alternatively expand $\mathcal{G}(R)$ using this definition: $\theta_R := log\mathcal{G}(R)$.

$$\begin{aligned} p(T|w) &= \mathcal{G}(T)/Z \\ &= \frac{1}{Z}\prod_{t\in T}\mathcal{G}(T_t) = \frac{1}{Z}exp\sum_{t\in T}\theta_{T_t} \\ &= \frac{1}{Z}exp\sum_{R\in\mathcal{R}}\theta_R \cdot f_R(T) \end{aligned} \tag{2}$$

This equation might look familiar as the dot product of the weights and feature vector is something we have encountered in our overview of log-linear models.

Most importantly, the partial derivative of log Z with respect to the weight vector $\theta$ gives us the expected counts under a log-linear distribution.

$$c(R) = \frac{\partial(logZ)}{\partial\theta_R} \tag{3}$$

This will prove useful in converting the updated weights to an expected count.

# 6 Adjusting the Forward-Backward Algorithm

## 6.1 Back-propagation

Let's approach how a backward algorithm might fit into the typical example of back-propagation. Back-propagation is a method in which weights in algorithmic circuit are updated based on the quantified cost of its results. Back-propagation can be run over and over on the same circuit to find the local minimum cost. The process consists of four steps:

1. **Forward Pass:** This is where the input is fed into the circuit and the output is computed with the current weights.
2. **Cost calculation**: We compare the current output to the desired output to determine a cost
3. **Backward Propagation**: This cost is then propagated backwards through the weights of the circuit. At each weight, we compute the gradient of the cost with respect to that weight.
4. **Adjustment of Weights**: Using the gradient of the error, each weight is adjusted in the direction that minimizes the cost. We may use methods such as stochastic gradient descent for this step.

In the context of the backward algorithm, since the structure itself is the one being adjusted and dependent on $w$, we can think of this as evaluating an **algorithmic circuit** where the inputs are the rule weights $\mathcal{G}(R) : R \in \mathcal{R}$, the internal nodes are backwards values of suffixes of $w$ also known as $\beta$ values, and the output is the total weight $Z$. This circuit is traversed in topological order and in Eisner's analogy, represents the forward pass of the back-propagation algorithm.

There are a few things that differ between our version of backprop for the backward algorithm and the vanilla variant introduced above. Instead of having a traditional supervised value to compare the output $Z$ to, we use $Z$ itself as the cost. Another adjustment made is that $Z$ must now be maximized, since $logZ$ is the likelihood of a particular parse, and we are trying to maximize likelihood. Thus, we find a local maximum and move in the opposite direction as the typical back-prop algorithm's gradient.

The back-propagation step is represented by an **adjoint circuit** which forms the same structure but is calculated in the reverse direction as the algorithmic circuit. Each node also calculates $\partial Z/\partial x$ where $x$ is the weight of the node in the original circuit. Finally, we have the last step, in which the original circuit's nodes are adjusted based on the adjoint circuit's gradients.

## 6.2 Mathematical Lemmas

Now that we have the backward algorithm (Algorithm 1), there are a few changes we need to make to the forward-backward algorithm utilize back-propagation and ultimately calculate the expected counts. Firstly, the we replace the weight functions with their respective adjoint operations. The weight

functions are sums of products in the form:

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

The respective adjoint expressions are these:

$$\begin{aligned} \frac{\partial Z}{\partial y_1} &+= \frac{\partial Z}{\partial x} \cdot y_1 \\ \frac{\partial Z}{\partial y_2} &+= \frac{\partial Z}{\partial x} \cdot y_2 \end{aligned} \tag{5}$$

Intuitively, $y_1$ is adjusted however much $x$ is adjusted, weighted on how much "influence" $y_1$ has on $x$. We can see this pattern applied to lines 9 and 7 in Algorithm 1 to produce lines 6-7 and 10-11 in Algorithm 2 respectively, as well as the stopping rule lines.

Lastly, we may use the expected counts expression derived in (3) and the definition $\theta_R := log\mathcal{G}(R)$ to synthesize a new expression which is independent of $\theta_R$:

$$\begin{aligned} c(R) &= \frac{\partial log Z}{\partial \theta_R} \\ &= \frac{\partial log Z}{\partial Z} \cdot \frac{\partial Z}{\partial \mathcal{G}(R)} \cdot \frac{\partial \mathcal{G}(R)}{\partial \theta_R} \\ &= (1/Z) \cdot \alpha[R] \cdot \mathcal{G}(R) \end{aligned} \tag{6}$$

This expression can be seen in the returned expected rule counts at the end of Algorithm 2.

# 7 Overview of the Reframed Algorithm

So here we have it! Algorithm 2 shows the pseudocode for a version of the forward-backward algorithm that utilizes back-propagation. To further connect this concept to Homework 2, there are some parallels that can be made between the optimization of weights with back-propagation versus using the EM algorithm. For the E-step in Homework 2, we used a formula customized for Hidden Markov Models that accrued the expected counts of each of the four kinds of operations (start, end, transmit, emit) at certain states to calculate expected counts. Analogously, the last line of Algorithm 2 calculates expected counts using the gradient of the total weight with respect to the rule weights. For the M-step, we had calculated new rule probabilities using the expected counts given the originating state. In Algorithm 2, we can see parallels in how the adjoint operation steps update the forward values.

**Algorithm 2** The Forward-Backward Algorithm

```
1:  function FORWARD-BACKWARD(G, w)
2:      Z :=BACKWARD(G, w)                    ▷ sets β[···]
3:      initialize all α[···] to 0
4:      α[ROOT^0] += 1                        ▷ sets ∂Z = 1
5:      for A ∈ N do                          ▷ starting rules
6:          α[ROOT → A] += α[ROOT^0]β[A^1]
7:          α[A^1] += α[ROOT^0]G(ROOT → A)
8:      for j := 1 to n − 1 do
9:          for A, B ∈ N do                   ▷ transition rules
10:             α[A → w_j B] += α[A^j]β[B^{j+1}]
11:             α[B^{j+1}] += α[A^j]G(A → w_j B)
12:     for A ∈ N do                          ▷ stopping rules
13:         G(A → w_n) += α[A^n]
14:     for R ∈ R do                          ▷ expected rule counts
15:         c(R) := α[R] · G(R)/Z
```

# 8 Significance

The generalization of inside-outside and forward-backwards as back-propagation is a hugely advantageous re-framing. In linguistics, the forward-backward algorithm is usually derived manually, tailored to the context of a particular problem. However, the underlying distribution under all of these algorithms is a log-linear (and therefore exponential) distribution. These distributions all have the property of the derivative with respect to the weights being equivalent to a vector of expected counts.

By attacking the problem from the perspective of back-propagation, we can apply the automatic differentiation software already in existence to the WCFG contexts.

# References


[1] Eisner, J. (2016). Inside-Outside and Forward-Backward Algorithms Are Just Backprop (tutorial paper). Association for Computational Linguistics. https://doi.org/10.18653/v1/w16-5901

[2] Nielsen, M. A. (2015). Neural networks and deep learning. http://neuralnetworksanddeeplearning.com, Ch. 2