Learning to Prune: Exploring the Frontier of Fast and Accurate Parsing

Tim Vieira and **Jason Eisner**
Department of Computer Science, Johns Hopkins University
{timv,jason}@cs.jhu.edu

Abstract

Pruning hypotheses during dynamic programming is commonly used to speed up inference in settings such as parsing. Unlike prior work, we train a pruning policy under an objective that measures end-to-end performance: we search for a fast *and* accurate policy. This poses a difficult machine learning problem, which we tackle with the LOLS algorithm. LOLS training must continually compute the effects of changing pruning decisions: we show how to make this efficient in the constituency parsing setting, via dynamic programming and change propagation algorithms. We find that optimizing end-to-end performance in this way leads to a better Pareto frontier—i.e., parsers which are more accurate for a given runtime.

1 Introduction

Decades of research have been dedicated to heuristics for speeding up inference in natural language processing tasks, such as constituency parsing (Pauls and Klein, 2009; Caraballo and Charniak, 1998) and machine translation (Petrov et al., 2008; Xu et al., 2013). Such research is necessary because of a trend toward richer models, which improve accuracy at the cost of slower inference. For example, state-of-the-art constituency parsers use grammars with millions of rules, while dependency parsers routinely use millions of features. Without heuristics, these parsers take minutes to process a single sentence.

To speed up inference, we will *learn* a pruning policy. During inference, the pruning policy is invoked to decide whether to keep or prune various parts of the search space, based on features of the input and (potentially) the state of the inference process.

Our approach searches for a policy with maximum end-to-end performance (reward) on training data, where the reward is a linear combination of problem-specific measures of accuracy and runtime, namely $\mathsf{reward} = \mathsf{accuracy} - \lambda \cdot \mathsf{runtime}$. The parameter $\lambda \geq 0$ specifies the relative importance of runtime and accuracy. By adjusting λ, we obtain policies with different speed-accuracy tradeoffs.

For learning, we use Locally Optimal Learning to Search (LOLS) (Chang et al., 2015b), an algorithm for learning sequential decision-making policies, which accounts for the end-to-end performance of the entire decision sequence jointly. Unfortunately, executing LOLS naively in our setting is prohibitive because it would run inference from scratch millions of times under different policies, training examples, and variations of the decision sequence. Thus, this paper presents efficient algorithms for *repeated* inference, which are applicable to a wide variety of NLP tasks, including parsing, machine translation and sequence tagging. These algorithms, based on change propagation and dynamic programming, dramatically reduce time spent evaluating similar decision sequences by leveraging problem structure and sharing work among evaluations.

We evaluate our approach by learning pruning heuristics for constituency parsing. In this setting, our approach is the first to account for end-to-end performance of the pruning policy, without making independence assumptions about the reward function, as in prior work (Bodenstab et al., 2011). In the larger context of learning-to-search for structured prediction, our work is unusual in that it learns to control a dynamic programming algorithm (i.e., graph-based parsing) rather than a greedy algorithm (e.g., transition-based parsing). Our experiments show that accounting for end-to-end performance in training leads to better policies along the entire Pareto frontier of accuracy and runtime.

2 Weighted CKY with pruning

A simple yet effective approach to speeding up parsing was proposed by Bodenstab et al. (2011), who trained a pruning policy π to classify whether or not spans of the input sentence $w_1 \cdots w_n$ form plausible

Transactions of the Association for Computational Linguistics, vol. 5, pp. 263–278, 2017. Action Editor: Marco Kuhlmann.
Submission batch: 5/2016; Revision batch: 9/2016; Published 8/2017.

constituents based on features of the input sentence. These predictions enable a parsing algorithm, such as CKY, to skip expensive steps during its execution: unlikely constituents are *pruned*. Only plausible constituents are *kept*, and the parser assembles the highest-scoring parse from the available constituents.

Alg. 1 provides pseudocode for weighted CKY with pruning. Weighted CKY aims to find the highest-scoring **derivation** (parse tree) of a given sentence, where a given **grammar** specifies a non-negative **score** for each **derivation rule** and a derivation's score is the product of the scores of the rules it uses.[1] CKY uses a dynamic programming strategy to fill in a three-dimensional array $\boldsymbol{\beta}$, known as the **chart**. The score β_{ikx} is the score of the highest-scoring subderivation with fringe $w_{i+1} \ldots w_k$ and root x. This value is computed by looping over the possible ways to assemble such a subderivation from smaller subderivations with scores β_{ijy} and β_{jkz} (lines 17–22). Additionally, we track a **witness** (backpointer) for each β_{ikx}, so that we can easily reconstruct the corresponding subderivation at line 23. The chart is initialized with lexical grammar rules (lines 3–9), which derive words from grammar symbols.

The key difference between *pruned* and *unpruned* CKY is an additional "if" statement (line 14), which queries the pruning policy π to decide whether to compute the several values β_{ikx} associated with a **span** (i, k). Note that width-1 and width-n spans are always kept because all valid parses require them.

3 End-to-end training

Bodenstab et al. (2011) train their pruning policy as a supervised classifier of spans. They derive direct supervision as follows: try to keep a span if it appears in the gold-standard parse, and prune it otherwise. They found that using an asymmetric weighting scheme helped find the right balance between false positives and false negatives. Intuitively, failing to prune is only a slight slowdown, whereas pruning a good item can ruin the accuracy of the parse.

[1] As is common practice, we assume the grammar has been binarized. We focus on *pre-trained* grammars, leaving co-adaptation of the grammar and pruning policy to future work. As indicated at lines 6 and 19, a rule's score may be made to depend on the context in which that rule is applied (Finkel et al., 2008), although the pre-trained grammars in our present experiments are ordinary PCFGs for which this is not the case.

Algorithm 1 PARSE: Weighted CKY with pruning

1: **Input:** grammar G, sentence $\boldsymbol{w}$, policy π
   Output: completed chart $\boldsymbol{\beta}$, derivation d
2: ▷ *Initialize chart*
3: $\boldsymbol{\beta} := \mathbf{0}$
4: **for** $k := 1$ **to** n :
5: **for** x such that $(x \to w_k) \in \text{rules}(G)$:
6: $s := G(x \to w_k \mid \boldsymbol{w}, k)$
7: **if** $s > \beta_{k-1,k,x}$:
8: $\beta_{k-1,k,x} := s$
9: *witness*$(k-1, k, x) := (k-1, k, w_k)$
10: **for** width $:= 2$ **to** n :
11: **for** $i := 0$ **to** $n -$ width :
12: $k := i +$ width ▷ *Current span is* (i, k)
13: ▷ *Policy determines whether to fill in this span*
14: **if** $\pi(\boldsymbol{w}, i, k) =$ prune :
15: **continue**
16: ▷ *Fill in span by considering each split point* j
17: **for** $j := i + 1$ **to** $k - 1$:
18: **for** $(x \to y\ z) \in \text{rules}(G)$:
19: $s := \beta_{ijy} \cdot \beta_{jkz} \cdot G(x \to y\ z \mid \boldsymbol{w}, i, j, k)$
20: **if** $s > \beta_{ikx}$:
21: $\beta_{ikx} := s$
22: *witness*$(i, k, x) := (j, y, z)$
23: $\widehat{d} :=$ follow backpointers from $(0, n, \text{ROOT})$
24: **return** $(\boldsymbol{\beta}, \widehat{d})$

Our end-to-end training approach improves upon asymmetric weighting by *jointly* evaluating the sequence of pruning decisions, measuring its effect on the test-time evaluation metric by actually running pruned CKY (Alg. 1). To estimate the value of a pruning policy π, we call $\text{PARSE}(G, \boldsymbol{w}^{(i)}, \pi)$ on each training sentence $\boldsymbol{w}^{(i)}$, and apply the **reward function**, $r = \text{accuracy} - \lambda \cdot \text{runtime}$. The **empirical value** of a policy is its average reward on the training set:

$$\mathcal{R}(\pi) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{E}\left[r(\text{PARSE}(G, \boldsymbol{w}^{(i)}, \pi))\right] \qquad (1)$$

The expectation in the definition may be dropped if PARSE, π, and r are all deterministic, as in our setting.[2] Our definition of r depends on the user parameter $\lambda \geq 0$, which specifies the amount of accuracy the user would sacrifice to save one unit of

[2] Parsers may break ties randomly or use Monte Carlo methods. The reward function r can be nondeterministic when it involves wallclock time or human judgments.

runtime. Training under a range of values for λ gives rise to policies covering a number of operating points along the Pareto frontier of accuracy and runtime.

End-to-end training gives us a principled way to decide what to prune. Rather than artificially labeling each pruning decision as inherently good or bad, we evaluate its effect in the context of the particular sentence and the other pruning decisions. Actions that prune a gold constituent are not equally bad—some cause cascading errors, while others are "worked around" in the sense that the grammar still selects a mostly-gold parse. Similarly, actions that prune a non-gold constituent are not equally good—some provide more overall speedup (e.g., pruning narrow constituents prevents wider ones from being built), and some even improve accuracy by suppressing an incorrect but high-scoring parse.

More generally, the gold vs. non-gold distinction is not even available in NLP tasks where one is pruning potential elements of a *latent* structure, such as an alignment (Xu et al., 2013) or a finer-grained parse (Matsuzaki et al., 2005). Yet our approach can still be used in such settings, by evaluating the reward on the downstream task that the latent structure serves.

Past work on optimizing end-to-end performance is discussed in §8. One might try to scale these techniques to learning to prune, but in this work we take a different approach. Given a policy, we can easily find small ways to improve it on specific sentences by varying individual pruning actions (e.g., if π currently prunes a span then try keeping it instead). Given a batch of improved action sequences (trajectories), the remaining step is to search for a policy which produces the improved trajectories. Conveniently, this can be reduced to a classification problem, much like the asymmetric weighting approach, except that the supervised labels and misclassification costs are not fixed across iterations, but rather are derived from interaction with the environment (i.e., PARSE and the reward function). This idea is formalized as a learning algorithm called Locally Optimal Learning to Search (Chang et al., 2015b), described in §4.

The counterfactual interventions we require—evaluating how reward would change if we changed one action—can be computed more efficiently using our novel algorithms (§5) than by the default strategy of running the parser repeatedly from scratch. The key is to reuse work among evaluations, which is possible because LOLS only makes tiny changes.

4 Learning algorithm

Pruned inference is a sequential decision process. The process begins in an initial state s_0. In pruned CKY, s_0 specifies the state of Alg. 1 at line 10, after the chart has been initialized from some selected sentence. Next, the policy is invoked to choose action $a_0 = \pi(s_0)$—in our case at line 14—which affects what the parser does next. Eventually the parser reaches some state s_1 from which it calls the policy to choose action $a_1 = \pi(s_1)$, and so on. When the policy is invoked at state s_t, it selects action a_t based on features extracted from the current state s_t—a snapshot of the input sentence, grammar and parse chart at time t.[3] We call the state-action sequence $s_0\, a_0\, s_1\, a_1 \cdots s_T$ a **trajectory**, where T is the **trajectory length**. At the final state, the reward function is evaluated, $r(s_T)$.

The LOLS algorithm for learning a policy is given in Alg. 2,[4] with a graphical illustration in Fig. 1. At a high level, LOLS alternates between **evaluating** and **improving** the current policy π_i.

The evaluation phase first samples a trajectory from π_i, called a **roll-in**: $s_0\, a_0\, s_1\, a_1 \cdots s_T \sim \text{ROLL-IN}(\pi_i)$. In our setting, s_0 is derived from a randomly sampled training sentence, but the rest of the trajectory is then deterministically computed by π_i given s_0. Then we revisit *each* state s in the roll-in (line 7), and try *each* available action $\bar{a} \in A(s)$ (line 9), executing π_i thereafter—a **rollout**—to measure the resulting reward $\widehat{r}[\bar{a}]$ (line 10). Our parser is deterministic, so a single rollout is an unbiased, 0-variance estimate of the expected reward. This process is repeated many times, yielding a large list $\widehat{Q}_i$ of pairs $\langle s, \widehat{\boldsymbol{r}} \rangle$, where s is a state that was encountered in some roll-in and $\widehat{\boldsymbol{r}}$ maps the possible actions $A(s)$ in that state to their measured rewards.

The improvement phase now trains a new policy π_{i+1} to try to choose high-reward actions, seeking a policy that will "on average" get high rewards $\boldsymbol{r}[\pi_{i+1}(s)]$. Good generalization is important: the policy must select high-reward actions even in states s that are not represented in $\widehat{Q}_i$, in case they are

[3] Our experiments do not make use of the current state of the chart. We discuss this decision in §8.

[4] Alg. 2 is simpler than in Chang et al. (2015b) because it omits oracle rollouts, which we do not use in our experiments.

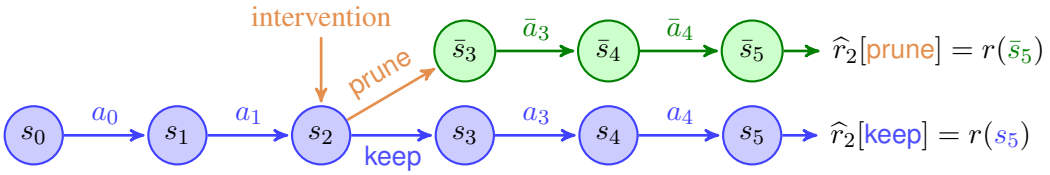


Figure 1: Example LOLS iteration (lines 6–10). Roll-in with the current policy π_i (starting with a random sentence), $s_0\, a_0\, s_1\, a_1 \cdots s_5 \sim \text{ROLL-IN}(\pi_i)$. Perform interventions at each state along the roll-in (only $t = 2$ is shown). The intervention tries alternative actions at each state (e.g., $\bar{a}_2 = \text{prune}$ at s_2). We rollout after the intervention by following π_i until a terminal state, $\bar{s}_3\, \bar{a}_3\, \bar{s}_4\, \bar{a}_4\, \bar{s}_5 \sim \text{ROLLOUT}(\pi_i, s_2, \bar{a}_2)$, and evaluate the reward of the terminal state $r(\bar{s}_5)$.

Algorithm 2 LOLS algorithm for learning to prune.

1: $\pi_1 := \text{INITIALIZEPOLICY}(\dots)$
2: **for** $i := 1$ **to** number of iterations :
3: ▷ *Evaluate: Collect dataset for* π_i
4: $\widehat{Q}_i := \emptyset$
5: **for** $j := 1$ **to** minibatch size :
6: $s_0\, a_0\, s_1\, a_1 \cdots s_T \sim \text{ROLL-IN}(\pi_i)$ ▷ *Sample*
7: **for** $t := 0$ **to** $T-1$:
8: ▷ *Intervene: Evaluate each action at* s_t
9: **for** $\bar{a}_t \in A(s_t)$: ▷ *Possible actions*
10: $\widehat{r}_t[\bar{a}_t] \sim \text{ROLLOUT}(\pi_i, s_t, \bar{a}_t)$
11: $\widehat{Q}_i$.append($\langle s_t, \widehat{\boldsymbol{r}}_t \rangle$)
12: ▷ *Improve: Train with dataset aggregation*
13: $\pi_{i+1} \leftarrow \text{TRAIN}\left(\bigcup_{k=1}^{i} \widehat{Q}_k\right)$
14: ▷ *Finalize: Pick the best policy over all iterations*
15: **return** $\text{argmax}_{i'}\, \mathcal{R}(\pi_{i'})$

encountered when running the new policy π_{i+1} (or when parsing test sentences). Thus, beyond just regularizing the training objective, we apply **dataset aggregation** (Ross et al., 2011): we take the training set to include not just $\widehat{Q}_i$ but also the examples from previous iterations (line 13). This also ensures that the sequence of policies $\pi_1, \pi_2, \dots$ will be "stable" (Ross and Bagnell, 2011) and will eventually converge.

So line 13 seeks to find a good classifier π_{i+1} using a training set: a possible classifier π would receive from each training example $\langle s, \widehat{\boldsymbol{r}} \rangle$ a reward of $\widehat{\boldsymbol{r}}[\pi(s)]$. In our case, where $A(s) = \{\text{keep}, \text{prune}\}$, this cost-sensitive classification problem is equivalent to training an ordinary binary classifier, after converting each training example $\langle s, \widehat{\boldsymbol{r}} \rangle$ to $\langle s, \text{argmax}_a\, \widehat{\boldsymbol{r}}[a] \rangle$ and giving this example a weight of $|\widehat{r}_{t,\text{keep}} - \widehat{r}_{t,\text{prune}}|$. Our specific classifier is described in §6.

In summary, the evaluation phase of LOLS collects training data for a cost-sensitive classifier, where the inputs (states), outputs (actions), and costs are obtained by *interacting with the environment*. LOLS concocts a training set and repeatedly revises it, similar to the well-known Expectation-Maximization algorithm. This enables end-to-end training of systems with discrete decisions and nondecomposable reward functions. LOLS gives us a principled framework for *deriving* (nonstationary) "supervision" even in tricky cases such as latent-variable inference (mentioned in §3). LOLS has strong theoretical guarantees, though in pathological cases, it may take exponential time to converge (Chang et al., 2015b).

The inner loop of the evaluation phase performs roll-ins, interventions and rollouts. Roll-ins ensure that the policy is (eventually) trained under the distribution of states it tends to encounter at test time. Interventions and rollouts force π_i to explore the effect of currently disfavored actions.

5 Efficient rollouts

Unlike most applications of LOLS and related algorithms, such as SEARN (Daumé III, 2006) and DAGGER (Ross et al., 2011), executing the policy is a major bottleneck in training. Because our dynamic programming parser explores many possibilities (unlike a greedy, transition-based decoder) its trajectories are quite long. This not only slows down each rollout: it means we must do more rollouts.

In our case, the trajectory has length $T = \frac{n \cdot (n+1)}{2} - 1 - n$ for a sentence of length n, where T is also the number of pruning decisions: one for each span other than the root and width-1 spans. LOLS must then perform T rollouts on this example. This means that to evaluate policy π_i, we must parse each sentence in the minibatch hundreds of times (e.g., 189 for $n = 20$, 434 for $n = 30$, and 779 for $n = 40$).

We can regard each policy π as defining a **pruning**

mask $\boldsymbol{m}$, an array that maps each of the T spans (i, k) to a decision m_{ik} ($1 = \mathsf{keep}, 0 = \mathsf{prune}$). Each rollout tries flipping a different bit in this mask.

We could spend less time on each sentence by sampling only some of its T rollouts (see §6). Regardless, the rollouts we do on a given sentence are related: in this section we show how to get further speedups by sharing work among them. In §5.2, we leverage the fact that rollouts will be similar to one another (differing by a single pruning decision). In §5.3, we show that the reward of all T rollouts can be computed simultaneously by dynamic programming under some assumptions about the structure of the reward function (described later). We found these algorithms to be crucial to training in a "reasonable" amount of time (see the empirical comparison in §7.2).

5.1 Background: Parsing as hypergraphs

It is convenient to present our efficient rollout algorithms in terms of the hypergraph structure of Alg. 1 (Klein and Manning, 2001; Huang, 2008; Li and Eisner, 2009; Eisner and Blatz, 2007). A hypergraph describes the information flow among related quantities in a dynamic programming algorithm. Many computational tricks apply generically to hypergraphs.

A hypergraph **edge** e (or **hyperedge**) is a "generalized arrow" $e.\mathsf{head} \leftarrow\!\prec e.\mathsf{Tail}$ with one output and a *list* of inputs. We regard each quantity β_{ikx}, m_{ik}, or $G(\ldots)$ in Alg. 1 as the value of a corresponding hypergraph **vertex** $\dot{\beta}_{ikx}, \dot{m}_{ik}$, or $\dot{G}(\ldots)$. Thus, *value*$(\dot{v}) = v$ for any vertex $\dot{v}$. Each $\dot{m}_{ik}$'s value is computed by the policy π or chosen by a rollout intervention. Each $\dot{G}$'s value is given by the grammar.

Values of $\dot{\beta}_{ikx}$, by contrast, are computed at line 19 if $k - i > 1$. To record the dependence of β_{ikx} on other quantities, our hypergraph includes the hyperedge $\dot{\beta}_{ikx} \leftarrow\!\prec (\dot{\beta}_{ijy}, \dot{\beta}_{jkz}, \dot{m}_{ik}, \dot{g})$ for each $0 \le i < j < k \le n$ and $(x \to y\ z) \in \text{rules}(G)$, where $\dot{g}$ denotes the vertex $\dot{G}(x \to y\ z \mid \boldsymbol{w}, i, j, k)$.

If $k - i = 1$, then values of β_{ikx} are instead computed at line 6, which does not access any other β values or the pruning mask. Thus our hypergraph includes the hyperedge $v_{ikx} \leftarrow\!\prec (\dot{g})$ whenever $i = k-1$, $0 \le i < k \le n$, and $(x \to w_k) \in \text{rules}(G)$, with $\dot{g} = \dot{G}(x \to w_k \mid \boldsymbol{w}, k)$.

With this setup, the value β_{ikx} is the maximum score of any derivation of vertex $\dot{\beta}_{ikx}$ (a tree rooted at $\dot{\beta}_{ikx}$, representing a subderivation), where the **score** of a derivation is the *product* of its leaf values. Alg. 1 computes it by considering hyperedges $\dot{\beta}_{ikx} \leftarrow\!\prec T$ and the previously computed values of the vertices in the tail T. For a vertex $\dot{v}$, we write $\mathsf{In}(\dot{v})$ and $\mathsf{Out}(\dot{v})$ for its sets of incoming and outgoing hyperedges. Our algorithms follow these hyperedges implicitly, without the overhead of materializing or storing them.

5.2 Change propagation (CP)

Change propagation is an efficient method for incrementally *re-evaluating* a computation under a change to its inputs (Acar and Ley-Wild, 2008; Filardo and Eisner, 2012). In our setting, each roll-in at Alg. 2 line 6 evaluates the reward $r(\text{PARSE}(G, x_i, \pi))$ from (1), which involves computing an entire parse chart via Alg. 1. The inner loop at line 10 performs T interventions per roll-in, which ask how reward would have changed if one bit in the pruning mask $\boldsymbol{m}$ had been different. Rather than reparsing from scratch (T times) to determine this, we can simply adjust the initial roll-in computation (T times).

CP is efficient when only a small fraction of the computation needs to be adjusted. In principle, flipping a single pruning bit can change up to 50% of the chart, so one might expect the bookkeeping overhead of CP to outweigh the gains. In practice, however, 90% of the interventions change $< 10\%$ of the β values in the chart. The reason is that β_{ikx} is a maximum over many quantities, only one of which "wins." Changing a given β_{ijy} rarely affects this maximum, and so changes are unlikely to propagate from vertex $\dot{\beta}_{ijy}$ to $\dot{\beta}_{ikx}$. Since changes are not very contagious, the "epidemic of changes" does not spread far.

Alg. 3 provides pseudocode for updating the highest-scoring derivation found by Alg. 1. We remark that the RECOMPUTE is called only when we flip a bit from keep to prune, which removes hyperedges and potentially decreases vertex values. The reverse flip only adds hyperedges, which increases vertex values via a running max (lines 12–14).

After determining the effect of flipping a bit, we must restore the original chart before trying a different bit (the next rollout). The simplest approach is to call Alg. 3 again to flip the bit back.[5]

[5] Our implementation uses a slightly faster method which accumulates an "undo list" of changes that it makes to the chart to quickly revert the modified chart to the original roll-in state.

Algorithm 3 Change propagation algorithm

Global: Alg. 1's vertex values/witnesses (roll-in)
procedure CHANGE($\dot{v}$, v)
  ▷ *Change the value of a leaf vertex $\dot{v}$ to v*
  value$(\dot{v}) := v$; *witness*$(\dot{v}) =$ LEAF
  $Q := \emptyset$; Q.push$(\dot{v})$ ▷ *Work queue ("agenda")*
  while $Q \neq \emptyset$: ▷ *Propagate until convergence*
    $\dot{u} := Q$.pop() ▷ *Narrower constituents first*
    if *witness*$(\dot{u}) =$ NULL : ▷ *Value is unknown*
      RECOMPUTE$(\dot{u})$ ▷ *Get value & witness*
    for $e \in$ Out$(\dot{u})$: ▷ *Propagate new value of $\dot{u}$*
      $\dot{s} := e$.head; $s := \prod_{\dot{u}' \in e.\text{Tail}}$ *value*$(\dot{u}')$
      if $s >$ *value*$(\dot{s})$: ▷ *Increase value*
        value$(\dot{s}) := s$; *witness*$(\dot{s}) := e$
        Q.push$(\dot{s})$
      else if *witness*$(\dot{s}) = e$ **and** $s <$ *value*$(\dot{s})$:
        witness$(\dot{s}) :=$ NULL ▷ *Value may decrease*
        Q.push$(\dot{s})$ ▷ *so, recompute upon pop*
procedure RECOMPUTE($\dot{s}$)
  for $e \in$ In$(\dot{s})$: ▷ *Max over incoming hyperedges*
    $s := \prod_{\dot{u} \in e.\text{Tail}}$ *value*$(\dot{u})$
    if $s >$ *value*$(\dot{s})$:
      value$(\dot{s}) = s$; *witness*$(\dot{s}) = e$

5.3 Dynamic programming (DP)

The naive rollout algorithm runs the parser T times—once for each variation of the pruning mask. The reader may be reminded of the finite difference approximation to the gradient of a function, which also measures the effects from perturbing each input value individually. In fact, for certain reward functions, the naive algorithm can be *precisely* regarded as computing a gradient—and thus we can use a more efficient algorithm, back-propagation, which finds the entire gradient vector of reward as fast (in the big-$\mathcal{O}$ sense) as computing the reward once. The overall algorithm is $\mathcal{O}(|E| + T)$ where $|E|$ is the total number of hyperedges, whereas the naive algorithm is $\mathcal{O}(|E'| \cdot T)$ where $|E'| \leq |E|$ is the maximum number of hyperedges actually visited on any rollout.

What accuracy measure must we use? Let $r(d)$ denote the recall of a derivation d—the fraction of gold constituents that appear as vertices in the derivation. A simple accuracy metric would be **1-best recall**, the recall $r(\widehat{d})$ of the highest-scoring derivation $\widehat{d}$ that was not pruned. In this section, we relax that to **expected recall**,[6] $\bar{r} = \sum_d p(d) r(d)$. Here we interpret the pruned hypergraph's values as an unnormalized probability distribution over derivations, where the probability $p(d) = \tilde{p}(d)/Z$ of a derivation is proportional to its score $\tilde{p}(d) = \prod_{\dot{u} \in \textit{leaves}(d)}$ *value*$(\dot{u})$.

Though $\bar{r}$ is not *quite* our evaluation metric, it captures *more* information about the parse forest, and so may offer some regularizing effect when used in a training criterion (see §7.1). In any case, $\bar{r}$ is close to $r(\widehat{d})$ when probability mass is concentrated on a few derivations, which is common with heavy pruning.

We can re-express $\bar{r}$ as $\tilde{r}/Z$, where

$$\tilde{r} = \sum_d \tilde{p}(d) r(d) \qquad Z = \sum_d \tilde{p}(d) \tag{2}$$

These can be efficiently computed by dynamic programming (DP), specifically by a variant of the inside algorithm (Li and Eisner, 2009). Since $\tilde{p}(d)$ is a product of rule weights and pruning mask bits at d's leaves (§5.1), each appearing at most once, both $\tilde{r}$ and Z vary *linearly* in any one of these inputs provided that all other inputs are held constant. Thus, the exact effect on $\tilde{r}$ or Z of changing an input m_{ik} can be found from the partial derivatives with respect to it. In particular, if we increased m_{ik} by $\Delta \in \{-1, 1\}$ (to flip this bit), the new value of $\bar{r}$ would be *exactly*

$$\frac{\tilde{r} + \Delta \cdot \partial \tilde{r}/\partial m_{ik}}{Z + \Delta \cdot \partial Z/\partial m_{ik}} \tag{3}$$

It remains to compute these partial derivatives. *All* partials can be jointly computed by back-propagation, which equivalent to another dynamic program known as the outside algorithm (Eisner, 2016).

The inside algorithm only needs to visit the $|E'|$ unpruned edges, but the outside algorithm must also visit some pruned edges, to determine the effect of "unpruning" them (changing their m_{ik} input from 0 to 1) by finding $\partial \tilde{r}/\partial m_{ik}$ and $\partial Z/\partial m_{ik}$. On the other hand, these partials are 0 when some *other* input to the hyperedge is 0. This case is common when the hypergraph is heavily pruned ($|E'| \ll |E|$), and means that back-propagation need not descend further through that hyperedge.

[6] In theory, we could anneal from expected to 1-best recall (Smith and Eisner, 2006). We experimented extensively with annealing but found it to be too numerically unstable for our purposes, even with high-precision arithmetic libraries.

Note that the DP method computes only the accuracies of rollouts—not the runtimes. In this paper, we will combine DP with a very simple runtime measure that is trivial to roll out (see §7). An alternative would be to use CP to roll out the runtimes. This is very efficient: to measure just runtime, CP only needs to update the record of which constituents or edges are built, and not their scores, so the changes are easier to compute than in §5.2, and peter out more quickly.

6 Parser details[7]

Setup: We use the standard English parsing setup: the Penn Treebank (Marcus et al., 1993) with the standard train/dev/test split, and standard tree normalization.[8] For efficiency during training, we restrict the length of sentences to ≤ 40. We do not restrict the length of test sentences. We experiment with two grammars: *coarse*, the "no frills" left-binarized treebank grammar, and *fine*, a variant of the Berkeley split-merge level-6 grammar (Petrov et al., 2006) as provided by Dunlop (2014, ch. 5). The parsing algorithms used during *training* are described in §5. Our *test-time* parsing algorithm uses the left-child loop implementation of CKY (Dunlop et al., 2010). All algorithms allow unary rules (though not chains). We evaluate accuracy at test time with the F_1 score from the official `EVALB` script (Sekine and Collins, 1997).

Training: Note that we never retrain the grammar weights—we train only the pruning policy. To TRAIN our classifiers (Alg. 2 line 13), we use L_2-regularized logistic regression, trained with L-BFGS optimization. We always rescale the example weights in the training set to sum to 1 (otherwise as LOLS proceeds, dataset aggregation overwhelms the regularizer). For the baseline (defined in next section), we determine the regularization coefficient by sweeping $\{2^{-11}, 2^{-12}, 2^{-13}, 2^{-14}, 2^{-15}\}$ and picking the best value (2^{-13}) based on the dev frontier. We re-used this regularization parameter for LOLS. The number of LOLS iterations is determined by a 6-day training-time limit[9] (meaning some jobs run many fewer iterations than others). For LOLS minibatch size we use 10K on the coarse grammar and 5K on the fine grammar. At line 15 of Alg. 2, we return the policy that maximized reward on *development* data, using the reward function from training.

Features: We use similar features to Bodenstab et al. (2011), but we have removed features that depend on part-of-speech tags. We use the following 16 feature templates for span (i, k) with $1 < k-i < N$: bias, sentence length, boundary words, conjunctions of boundary words, conjunctions of word shapes, span shape, width bucket. Shape features map a word or phrase into a string of character classes (uppercase, lowercase, numeric, spaces); we truncate substrings of identical classes to length two; punctuation chars are never modified in any way. Width buckets use the following partition: $2, 3, 4, 5, [6, 10], [11, 20], [21, \infty)$. We use feature hashing (Weinberger et al., 2009) with `MurmurHash3` (Appleby, 2008) and project to 2^{22} features. Conjunctions are taken at positions $(i-1, i), (k, k+1), (i-1, k+1)$ and (i, k). We use special begin and end symbols when a template accesses positions beyond the sentence boundary.

Hall et al. (2014) give examples motivating our feature templates and show experimentally that they are effective in multiple languages. Boundary words are strong surface cues for phrase boundaries. Span shape features are also useful as they (minimally) check for matched parentheses and quotation marks.

7 Experimental design and results

Reward functions and surrogates: Each user has a personal reward function. In this paper, we choose to specify our *true* reward as $\mathsf{accuracy} - \lambda \cdot \mathsf{runtime}$, where $\mathsf{accuracy}$ is given by labeled F_1 percentage and $\mathsf{runtime}$ by mega-pushes (mpush), millions of calls per sentence to lines 6 and 19 of Alg. 1, which is in practice proportional to seconds per sentence (correlation > 0.95) and is more replicable. We *evaluate* accordingly (on test data)—but during LOLS *training* we approximate these metrics. We compare:

- r_{CP} (fast): Use change propagation (§5.2) to compute $\mathsf{accuracy}$ on a sentence as F_1 of just that sentence, and to approximate $\mathsf{runtime}$ as $||\boldsymbol{\beta}||_0$,

[7] Code for experiments is available at `http://github.com/timvieira/learning-to-prune`.

[8] Data train/dev/test split (by section) 2–21 / 22 / 23. Normalization operations: Remove function tags, traces, spurious unary edges ($X \to X$), and empty subtrees left by other operations. Relabel `ADVP` and `PRT|ADVP` tags to `PRT`.

[9] On the 7th day, LOLS rested and performance was good.

the number of constituents that were built.[10]

- r_{DP} (faster): Use dynamic programming (§5.3) to approximate accuracy on a sentence as *expected* recall.[11] This time we approximate runtime more crudely as $||\boldsymbol{m}||_0$, the number of nonzeros in the pruning mask for the sentence (i.e., the number of spans whose constituents the policy would be willing to keep *if* they were built).

We use these surrogates because they admit efficient rollout algorithms. Less important, they preserve the training objective (1) as an average over sentences. (Our true F_1 metric on a corpus cannot be computed in this way, though it could reasonably be estimated by averaging over minibatches of sentences in (1).)

Controlled experimental design: Our baseline system is an adaptation of Bodenstab et al. (2011) to learning-to-prune, as described in §3 and §6. Our goal is to determine whether such systems can be improved by LOLS training. We repeat the following design for both reward surrogates (r_{CP} and r_{DP}) and for both grammars (*coarse* and *fine*).

① We start by training a number of baseline models by sweeping the asymmetric weighting parameter. For the coarse grammar we train 8 such models, and for the fine grammar 12.

② For each baseline policy, we estimate a value of λ for which that policy is optimal (among baseline policies) according to surrogate reward.[12]

③ For each baseline policy, we run LOLS with the same surrogate reward function (defined by λ) for which that baseline policy was optimal. We initialize LOLS by setting π_0 to the baseline policy. Furthermore, we include the baseline policy's weighted training set $\widehat{Q}_0$ in the $\bigcup$ at line 13. Fig. 2 shows that LOLS learns to improve on the baseline, as evaluated on development data.

④ But do these surrogate reward improvements also improve our true reward? For each baseline policy, we use dev data to estimate a value of λ for which that policy is optimal according to our *true* reward function. We use blind test data to compare the baseline policy to its corresponding LOLS policy on this true reward function, testing significance with a paired permutation test. The improvements hold up, as shown in Fig. 3.

The rationale behind this design is that a user who actually wishes to maximize $\text{accuracy} - \lambda \cdot \text{runtime}$, for some specific λ, could reasonably start by choosing the best baseline policy for this reward function, and then try to improve that baseline by running LOLS with the same reward function. Our experiments show this procedure works for a range of λ values.

In the real world, a user's true objective might instead be some nonlinear function of runtime and accuracy. For example, when accuracy is "good enough," it may be more important to improve runtime, and vice-versa. LOLS could be used with such a nonlinear reward function as well. In fact, a user does not even have to quantify their *global* preferences by writing down such a function. Rather, they could select manually among the baseline policies, choosing one with an attractive speed-accuracy tradeoff, and then specify λ to indicate a *local* direction of desired improvement (like the green arrows in Fig. 2), modifying this direction periodically as LOLS runs.

7.1 Discussion

As previous work has shown, learning to prune gives us excellent parsers with less than $< 2\%$ overhead

[10] When using r_{CP}, we speed up LOLS by doing $\leq 2n$ rollouts per sentence of length n. We sample these uniformly without replacement from the T possible rollouts (§5), and compensate by upweighting the resulting training examples by $T/(2n)$.

[11] Considering all nodes in the *binarized tree*, except for the root, width-1 constituents, and children of unary rules.

[12] We estimate λ by first fitting a parametric model $y_i = h(x_i) \triangleq y_{\max} \cdot \text{sigmoid}(a \cdot \log(x_i + c) + b)$ to the baseline runtime-accuracy measurements on dev data (shown in green in Fig. 2) by minimizing mean squared error. We then use the fitted curve's slope h' to estimate each $\lambda_i = h'(x_i)$, where x_i is the runtime of baseline i. The resulting choice of reward function $y - \lambda_i \cdot x$ increases along the green arrow in Fig. 2, and is indeed maximized (subject to $y \leq h(x)$, and in the region where h is concave) at $x = x_i$. As a sanity check, notice since λ_i is a derivative of the function $y = h(x)$, its units are in units of y (accuracy) per unit of x (runtime), as appropriate for use in the expression $y - \lambda_i \cdot x$. Indeed, this procedure will construct the same reward function regardless of the units we use to express x. Our specific parametric model h is a sigmoidal curve, with accuracy $\to y_{\max}$ asymptotically as runtime $\to \infty$. It obtains an excellent fit by placing accuracy and runtime on the log-logit scale—that is, $\log(x_i + c)$ and $\text{logit}(y_i/y_{\max})$ transforms are used to convert our *bounded* random variables x_i and y_i to *unbounded* ones—and then assuming they are linearly related.

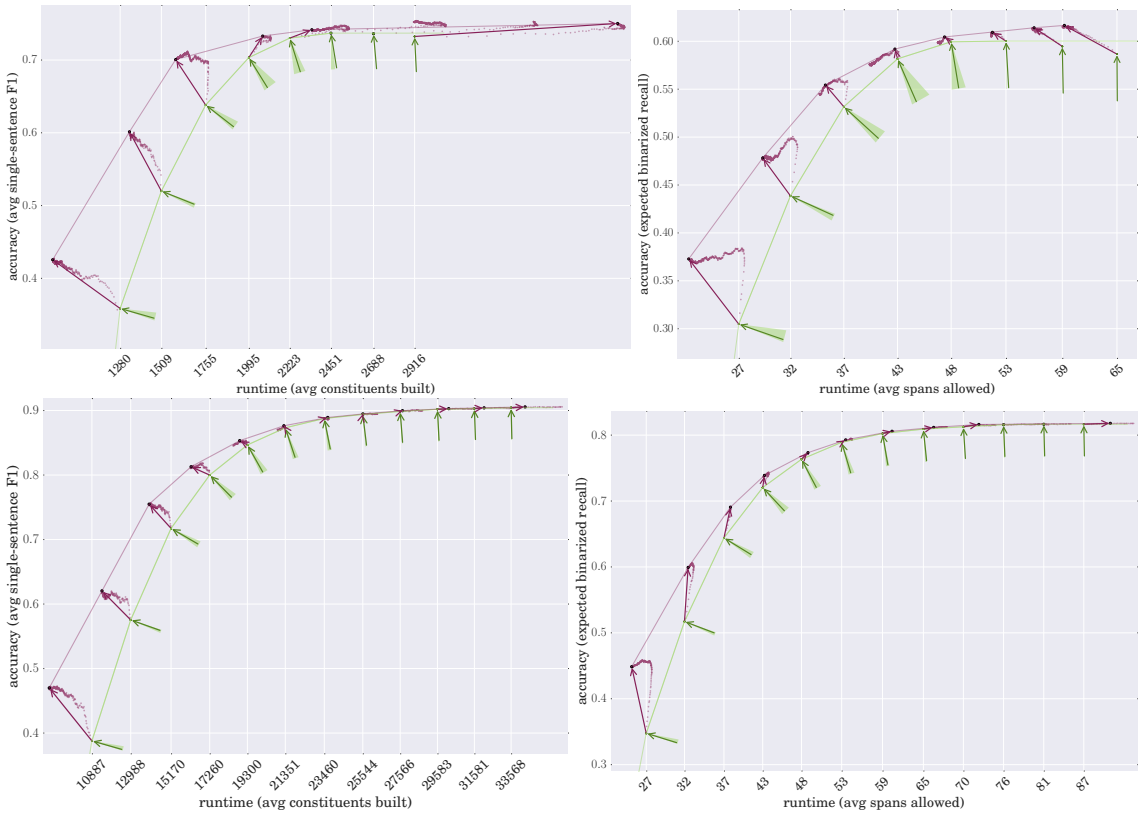


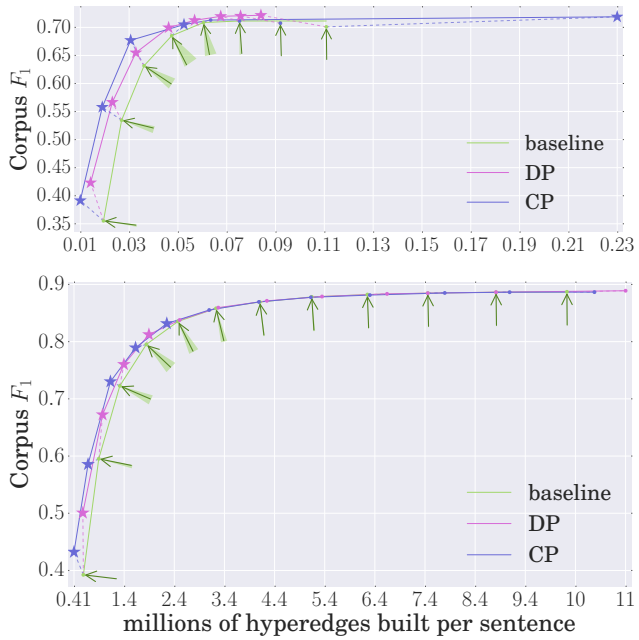
Figure 2: Depiction of LOLS pushing out the frontier of surrogate objectives, r_{CP} (left) and r_{DP} (right), on dev data with coarse (top) and fine (bottom) grammars. Green elements are associated with the baseline and purple elements with LOLS. ① The green curve shows the performance of the baseline policies. ② For each baseline policy, a green arrow points along the gradient of surrogate reward, as defined by the λ that would identify that baseline as optimal. (In case a user wants a different value of λ but is unwilling to search for a better baseline policy outside our set, the green cones around each baseline arrow show the range of λs that would select that baseline from our set.) ③ The LOLS trajectory is shown as a series of purple points, and the purple arrow points from the baseline policy to the policy selected by LOLS with early stopping (§6). This improves surrogate reward if the purple arrow has a positive inner product with the green arrow. LOLS cannot move exactly in the direction of the green arrow because it is constrained to find points that correspond to actual parsers. Typically, LOLS ends up improving accuracy, either along with runtime or at the expense of runtime.

for deciding what to prune (i.e., pruning feature extraction and span classification). Even the baseline pruner has access to features unavailable to the grammar, and so it learns to override the grammar, improving an unpruned coarse parser's accuracy from 61.1 to as high as 70.1% F_1 on test data (i.e., beneficial search error). It is also 8.1x faster![13] LOLS simply does a better job at figuring out where to prune, raising accuracy 2.1 points to 72.2 (while maintaining a 7.4x speedup). Where pruning is more aggressive, LOLS has even more impact on accuracy.

Even on the fine grammar, where there is less room to improve accuracy, the most accurate LOLS system improves an unpruned parser by $+0.16\%$ F_1 with a 8.6x speedup. For comparison, the most accurate baseline drops -0.03% F_1 with a 9.7x speedup.

With the fine grammar, we do not see much improvement over the baseline in the accuracy > 85 regions. This is because the supervision specified by asymmetric weighting is similar to what LOLS surmises via rollouts. However, in lower-accuracy regions we see that LOLS can significantly improve reward over its baseline policy. This is because the baseline supervision does not teach which plausible

[13] We measure runtime as best of 10 runs (recommended by Dunlop (2014)). All parser timing experiments were performed on a Linux laptop with the following specs: Intel® Core™ i5-2540M 2.60GHz CPU, 8GB memory, $32K/256K/3072K$ $L_1/L_2/L_3$ cache. Code is written in the Cython language.



		baseline			r_{CP}			r_{DP}		
	λ	F_1	mpush	kw/s	F_1	mpush	kw/s	F_1	mpush	kw/s
coarse	unpruned	61.1	2.39	1.4						
	5.17	70.1	0.111	10.4	**71.9**	**0.230**	6.6	**72.2**	**0.0841**	10.1
	12.1	70.7	0.0921	10.3	–	–	–	**72.0**	**0.0758**	10.7
	30.6	71.1	0.0753	11.0	–	–	–	**72.0**	**0.0676**	12.6
	80.5	70.9	0.0606	12.8	71.3	0.0635	13.1	**71.3**	**0.0569**	12.5
	221	68.5	0.0475	13.7	**70.5**	**0.0525**	13.2	**70.0**	**0.0462**	14.7
	651	63.2	0.0362	14.2	**67.7**	**0.0307**	13.8	**65.5**	**0.0328**	15.8
	1760	53.5	0.0269	15.3	**55.8**	**0.0190**	15.8	**56.7**	**0.0231**	16.0
	3260	35.5	0.0195	16.7	**39.1**	**0.00998**	20.3	**42.3**	**0.0142**	19.6
fine	unpruned	88.7	145.	0.03						
	0.0736	88.7	10.2	0.24	88.7	10.8	0.23	88.9	11.4	0.22
	0.116	88.6	8.82	0.27	88.6	9.09	0.27	–	–	–
	0.189	88.5	7.46	0.33	88.5	7.80	0.31	–	–	–
	0.321	88.2	6.25	0.37	88.1	6.30	0.36	88.4	6.65	0.34
	0.571	87.8	5.15	0.42	87.7	5.13	0.42	87.9	5.35	0.41
	1.07	86.9	4.11	0.51	86.9	4.09	0.50	87.1	4.25	0.49
	2.13	85.8	3.24	0.62	85.5	3.10	0.62	85.9	3.28	0.60
	4.38	83.5	2.48	0.77	**83.2**	**2.26**	0.76	83.8	2.51	0.71
	9.32	79.5	1.85	0.93	**78.9**	**1.63**	0.96	**81.2**	**1.90**	0.93
	20.5	72.3	1.32	1.20	**73.0**	**1.13**	1.19	**76.0**	**1.40**	1.16
	43.5	59.5	0.901	1.58	**58.6**	**0.688**	1.74	**67.2**	**0.977**	1.56
	73.4	39.2	0.595	2.03	**43.2**	**0.408**	2.58	**50.1**	**0.584**	2.23

Figure 3: Test set results on coarse (top) and fine (bottom) grammars. Each curve or column represents a different training regimen. Accuracy is measured in F_1 percentage; runtime is measured by millions of hyperedges built per sentence. ④ Here, the green arrows point in the direction of true reward. Dashed lines connect each green baseline point to the two LOLS-improved points. Starred points and bold values indicate a significant improvement over the baseline *reward* (paired permutation test, $p < 0.05$). In no case was there a statistically significant decrease. In 4 cases (marked with '–') the policy chosen by early stopping was the initial baseline policy. We also report words per second $\times 10^3$ (kw/s).

constituents are "safest" to prune, nor can it learn strategies such as "skip all long sentences." We discuss why LOLS does not help as much in the high accuracy regions further in §7.3.

In a few cases in Fig. 2, LOLS finds no policy that improves surrogate reward on dev data. In these cases, surrogate reward does improve slightly on training data (not shown), but early stopping just keeps the initial (baseline) policy since it is just as good on dev data. Adding a bit of additional random exploration might help break out of this initialization.

Interestingly, the r_{DP} LOLS policies find higher-accuracy policies than the corresponding r_{CP} policies, despite a greater mismatch in surrogate accuracy definitions. We suspect that r_{DP}'s approach of trying to improve *expected* accuracy may provide a useful regularizing effect, which smooths out the reward signal and provides a useful bias (§5.3).

The most pronounced qualitative difference due to LOLS training is substantially lower rates of parse failure in the mid- to high- λ-range on both grammars (not shown). Since LOLS does end-to-end training, it can advise the learner that a certain pruning decision catastrophically results in no parse being found.

7.2 Training speed and convergence

Part of the contribution of this paper is faster algorithms for performing LOLS rollouts during training (§5). Compared to the naive strategy of running the parser from scratch T times, r_{CP} achieves speedups of 4.9–6.6x on the coarse grammar and 1.9–2.4x on the fine grammar. r_{DP} is even faster, 10.4–11.9x on coarse and 10.5–13.8x on fine. Most of the speedup comes from longer sentences, which take up most of the runtime for all methods. Our new algorithms enable us to train on fairly long sentences ($\leq$ 40). We note that our implementations of r_{CP} and r_{DP} are not as highly optimized as our test-time parser, so there may be room for improvement.

Orthogonal to the cost per rollout is the number of training iterations. LOLS may take many steps to converge if trajectories are long (i.e., T is large)

because each iteration of LOLS training attempts to improve the current policy *by a single action*. In our setting, T is quite large (discussed extensively in §5), but we are able to circumvent slow convergence by initializing the policy (via the baseline method). This means that LOLS can focus on *fine-tuning* a policy which is already quite good. In fact, in 4 cases, LOLS did not improve from its initial policy.

We find that when λ is large—the cases where we get meaningful improvements because the initial policy is far from locally optimal—LOLS steadily and smoothly improves the surrogate reward on both training and development data. Because these are fast parsers, LOLS was able to run on the order of 10 (fine grammar) or 100 (coarse grammar) epochs within our 6-day limit; usually it was still improving when we terminated it. By contrast, for the slower and more accurate small-λ parsers (which completed fewer training epochs), LOLS still improves surrogate reward on training data, but without systematically improving on development data—often the reward on development fluctuates, and early stopping simply picks the best of this small set of "random" variants.

7.3 Understanding the LOLS training signal

In §3, we argued that LOLS gives a more appropriate training signal for pruning than the baseline method of consulting the gold parse, because it uses rollouts to measure the full effect of each pruning decision in the context of the other decisions made by the policy.

To better understand the results of our previous experiments, we analyze how often a rollout does determine that the baseline supervision for a span is suboptimal, and how suboptimal it is in those cases.

We specifically consider LOLS rollouts that evaluate the r_{CP} surrogate (because r_{DP} is a cruder approximation to true reward). These rollouts $\widehat{Q}_i$ tell us what actions LOLS is *trying* to improve in its current policy π_i for a given λ, although there is no guarantee that the learner in §4 will succeed at classifying $\widehat{Q}_i$ correctly (due to limited features, regularization, and the effects of dataset aggregation).

We define regret of the baseline oracle. Let $\mathsf{best}(s) \triangleq \operatorname{argmax}_a \textsc{rollout}(\pi, s, a)$ and $\mathsf{regret}(s) \triangleq (\textsc{rollout}(\pi, s, \mathsf{best}(s)) - \textsc{rollout}(\pi, s, \mathsf{gold}(s)))$. Note that $\mathsf{regret}(s) \geq 0$ for all s, and let $\mathsf{diff}(s)$ be the event that $\mathsf{regret}(s) > 0$ strictly. We are interested in analyzing the expected regret over all gold and non-gold spans, which we break down as

$$\begin{aligned}\mathbb{E}[\mathsf{regret}] = p(\mathsf{diff}) \qquad (4)\\ \cdot \big(p(\mathsf{gold} \mid \mathsf{diff}) \cdot \mathbb{E}[\mathsf{regret} \mid \mathsf{gold}, \mathsf{diff}]\\ + \; p(\neg\,\mathsf{gold} \mid \mathsf{diff}) \cdot \mathbb{E}[\mathsf{regret} \mid \neg\,\mathsf{gold}, \mathsf{diff}]\big)\end{aligned}$$

where expectations are taken over $s \sim \textsc{roll-in}(\pi)$.

Empirical analysis of regret: To show where the benefit of the LOLS oracle comes from, Fig. 4 graphs the various quantities that enter into the definition (4) of baseline regret, for different π, λ, and grammar. The LOLS oracle evolves along with the policy π, since it identifies the best action *given* π. We thus evaluate the oracle baseline against two LOLS oracles: the one used at the start of LOLS training (derived from the initial policy π_1 that was trained on baseline supervision), and the one obtained at the end (derived from the LOLS-trained policy π_* selected by early stopping). These comparisons are shown by solid and dashed lines respectively.

Class imbalance (black curves): In all graphs, the aggregate curves primarily reflect the non-gold spans, since only 8% of spans are gold.

Gold spans (gold curves): The top graphs show that a substantial fraction of the gold spans should be pruned (whereas the baseline tries to keep them all), although the middle row shows that the benefit of pruning them is small. In most of these cases, pruning a gold span improves speed but leaves accuracy unchanged—because that gold span was missed anyway by the highest-scoring parse. Such cases become both more frequent and more beneficial as λ increases and we prune more heavily. In a minority of cases, however, pruning a gold span also improves accuracy (through beneficial search error).

Non-gold spans (purple curves): Conversely, the top graphs show that a few non-gold spans should be kept (whereas the baseline tries to prune them all), and the middle row shows a large benefit from keeping them. They are needed to recover from catastrophic errors and get a mostly-correct parse.

Coarse vs. fine (left vs. right): The two grammars differ mainly for small λ, and this difference comes especially from the top row. With a fine grammar and small λ, the baseline parses are more accurate, so LOLS has less room for improvement: fewer

gold spans go unused, and fewer non-gold spans are needed for recovery.

Effect of λ: Aggressive pruning (large λ) reduces accuracy, so its effect on the top row is similar to that of using a coarse grammar. Aggressive pruning also has an effect on the middle row: there is more benefit to be derived from pruning unused gold spans (surprisingly), and especially from keeping those non-gold spans that are helpful (presumably they enable recovery from more severe parse errors). These effects are considerably sharper with r_{DP} reward (not shown here), which more smoothly evaluates the entire weighted pruned parse forest rather than trying to coordinate actions to ensure a good single 1-best tree; the baseline oracle is excellent at choosing the action that gets the better forest when the forest is mostly present (small λ) but not when it is mostly pruned (large λ).

Effect on retraining the policy: The black lines in the bottom graphs show the overall regret (on training data) if we were to *perfectly* follow the baseline oracle rather than the LOLS oracle. In practice, retraining the policy to match the oracle will not match it perfectly in either case. Thus the baseline method has a further disadvantage: when it trains a policy, its training objective weights all gold or all non-gold examples equally, whereas LOLS invests greater effort in matching the oracle on those states where doing so would give greater downstream reward.

8 Related work

Our experiments have focused on using LOLS to improve a reasonable baseline. Fig. 5 shows that our resulting parser fits reasonably among state-of-the-art constituency parsers trained and tested on the Penn Treebank. These parsers include a variety of techniques that improve speed or accuracy. Many are quite orthogonal to our work here—e.g., the SpMV method (which is necessary for Bodenstab's parser to beat ours) is a set of cache-efficient optimizations (Dunlop, 2014) that could be added to our parser (just as it was added to Bodenstab's), while Hall et al. (2014) and Fernández-González and Martins (2015) replace the grammar with faster scoring models that have more conditional independence. Overall, other fast parsers could also be trained using LOLS, so that

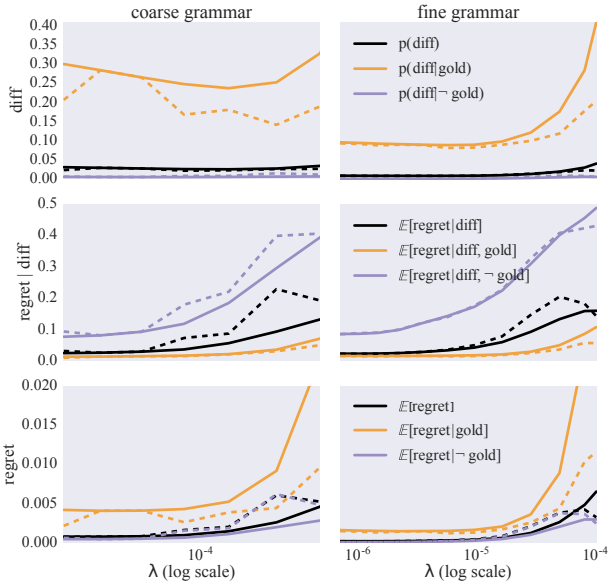


Figure 4: Comparison of the LOLS and baseline training signals based on the regret decomposition in Eq. (4) as we vary π, λ, and grammar. Solid lines show where the baseline oracle is suboptimal on its own system π_1 and dashed lines show where it is suboptimal on the LOLS-improved system π_*. Each plot shows an overall quantity in black as well as that quantity broken down by gold and non-gold spans. **Top:** Fraction of states in which oracles differ. **Middle:** Expected regret per state in which oracles differ. **Bottom:** Expected regret per state. See §7.3 for discussion.

they quickly find parses that are accurate, or at least helpful to the accuracy of some downstream task.

Pruning methods[14] can use classifiers not only to select spans but also to prune at other granularities (Roark and Hollingshead, 2008; Bodenstab et al., 2011). *Prioritization* methods do not prune substructures, but instead delay their processing until they are needed—if ever (Caraballo and Charniak, 1998).

This paper focuses on *learning* pruning heuristics that have trainable parameters. In the same way, Stoyanov and Eisner (2012) learn to turn off unneeded factors in a graphical model, and Jiang et al. (2012) and Berant and Liang (2015) train prioritization heuristics (using policy gradient). In both of those 2012 papers, we explicitly sought to maximize $\text{accuracy} - \lambda \cdot \text{runtime}$ as we do here. Some previous "coarse-to-fine" work does not optimize heuris-

[14] We focus here on parsing, but pruning is generally useful in structured prediction. E.g., Xu et al. (2013) train a classifier to prune (latent) alignments in a machine translation system.

System	F_1	words/sec
Dyer et al. (2016a); Dyer et al. (2016b)	93.3	–
Zhu et al. (2013)	90.4	1290
Fernández-González and Martins (2015)	90.2	957
Petrov and Klein (2007)	90.1	169
Crabbé (2015)	90.0	2150
Our most accurate parser	88.9	218
Bodenstab (2012) w/ SpMV	88.8	1581
Bodenstab (2012) w/o SpMV	88.7	188
Hall et al. (2014)	88.6	12

Figure 5: Comparison among fast and accurate parsers. Runtimes are computed on different machines and parsers are implemented in different programming languages, so runtime is not a controlled comparison.

tics directly but rather derives heuristics for pruning (Charniak et al., 2006; Petrov and Klein, 2007; Weiss and Taskar, 2010; Rush and Petrov, 2012) or prioritization (Klein and Manning, 2003; Pauls and Klein, 2009) from a coarser version of the model. Combining these automatic methods with LOLS would require first enriching their heuristics with trainable parameters, or parameterizing the coarse-to-fine hierarchy itself as in the "feature pruning" work of He et al. (2013) and Strubell et al. (2015).

Dynamic features are ones that depend on previous actions. In our setting, a policy could in principle benefit from considering the full state of the chart at Alg. 1 line 14. While coarse-to-fine methods implicitly use certain dynamic features, training with dynamic features is a fairly new goal that is challenging to treat efficiently. It has usually been treated with some form of simple imitation learning, using a heuristic training signal much as in our baseline (Jiang, 2014; He et al., 2013). LOLS would be a more principled way to train such features, but for efficiency, our present paper restricts to *static features* that only access the state via $\pi(\boldsymbol{w}, i, k)$. This permits our fast CP and DP rollout algorithms. It also reduces the time and space cost of dataset aggregation.[15]

LOLS attempts to do *end-to-end* training of a sequential decision-making system, without falling back on black-box optimization tools (Och, 2003; Chung and Galley, 2012) that ignore the sequential structure. In NLP, sequential decisions are more commonly trained with *step-by-step* supervision (Kuhlmann et al., 2011), using methods such as local classification (Punyakanok and Roth, 2001) or beam search with early update (Collins and Roark, 2004). LOLS tackles the harder setting where the only training signal is a joint assessment of the entire sequence of actions. It is an alternative to policy gradient, which does not scale well to our long trajectories because of high variance in the estimated gradient and because random exploration around (even good) pruning policies most often results in no parse at all. LOLS uses controlled comparisons, resulting in more precise "credit assignment" and tighter exploration.

We would be remiss not to note that current transition-based parsers—for constituency parsing (Zhu et al., 2013; Crabbé, 2015) as well as dependency parsing (Chen and Manning, 2014)—are both incredibly fast and surprisingly accurate. This may appear to undermine the motivation for our work, or at least for its application to fast parsing.[16] However, transition-based parsers do not produce marginal probabilities of substructures, which can be useful features for downstream tasks. Indeed, the transition-based approach is essentially greedy and so it may fail on tasks with more ambiguity than parsing. Current transition-based parsers also require step-by-step supervision, whereas our method can also be used to train in the presence of incomplete supervision, latent structure, or indirect feedback. Our method could also be used immediately to speed up dynamic programming methods for MT, synchronous parsing, parsing with non-context-free grammar formalisms, and other structured prediction problems for which transition systems have not (yet) been designed.

9 Conclusions

We presented an approach to learning pruning policies that optimizes end-to-end performance on a user-specified speed-accuracy tradeoff. We developed two novel algorithms for efficiently measuring how varying policy actions affects reward. In the case of parsing, given a performance criterion and a good baseline policy for that criterion, the learner consistently manages to find a higher-reward policy. We hope this work inspires a new generation of fast and accurate structured prediction models with tunable runtimes.

[15] LOLS repeatedly evaluates actions given $(\boldsymbol{w}, i, k)$. We consolidate the resulting training examples by summing their reward vectors $\widehat{\boldsymbol{r}}$, so the aggregated dataset does not grow over time.

[16] Of course, LOLS can also train transition-based parsers (Chang et al., 2015a), or even vary their beam width dynamically.

Acknowledgments

This material is based in part on research sponsored by the National Science Foundation under Grant No. 0964681 and DARPA under agreement number FA8750-13-2-0017 (DEFT program). We'd like to thank Nathaniel Wesley Filardo, Adam Teichert, Matt Gormley and Hal Daumé III for helpful discussions. Finally, we thank TACL action editor Marco Kuhlmann and the anonymous reviewers and copy editor for suggestions that improved this paper.

References

Umut A. Acar and Ruy Ley-Wild. 2008. Self-adjusting computation with Delta ML. In Pieter Koopman and Doaitse Swierstra, editors, *Advanced Functional Programming*, pages 1–38.

Austin Appleby. 2008. Murmurhash3. `https://sites.google.com/site/murmurhash`.

Jonathan Berant and Percy Liang. 2015. Imitation learning of agenda-based semantic parsers. *Transactions of the Association for Computational Linguistics*, 3:545–558.

Nathan Bodenstab, Aaron Dunlop, Keith Hall, and Brian Roark. 2011. Beam-width prediction for efficient CYK parsing. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Nathan Matthew Bodenstab. 2012. *Prioritization and Pruning: Efficient Inference with Weighted Context-Free Grammars*. Ph.D. thesis, Oregon Health and Science University.

Sharon A. Caraballo and Eugene Charniak. 1998. New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics*, 24(2):275–298.

Kai-Wei Chang, He He, Hal Daumé III, and John Langford. 2015a. Learning to search for dependencies. *Computing Research Repository*, arXiv:1503.05615.

Kai-Wei Chang, Akshay Krishnamurthy, Alekh Agarwal, Hal Daumé III, and John Langford. 2015b. Learning to search better than your teacher. In *Proceedings of the International Conference on Machine Learning*.

Eugene Charniak, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R. Shrivaths, Jeremy Moore, Michael Pozar, and Theresa Vu. 2006. Multilevel coarse-to-fine PCFG parsing. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology*.

Danqi Chen and Christopher D. Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Tagyoung Chung and Michel Galley. 2012. Direct error rate minimization for statistical machine translation. In *Proceedings of the Workshop on Statistical Machine Translation*.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Benoit Crabbé. 2015. Multilingual discriminative lexicalized phrase structure parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Harold Charles Daumé III. 2006. *Practical Structured Learning Techniques for Natural Language Processing*. Ph.D. thesis, University of Southern California.

Aaron Dunlop, Nathan Bodenstab, and Brian Roark. 2010. Reducing the grammar constant: An analysis of CYK parsing efficiency. Technical report, CSLU-2010-02, OHSU.

Aaron Joseph Dunlop. 2014. *Efficient Latent-Variable Grammars: Learning and Inference*. Ph.D. thesis, Oregon Health and Science University.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016a. Recurrent neural network grammars. *Computing Research Repository*, arxiv:1602.07776.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016b. Recurrent neural network grammars. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics*.

Jason Eisner and John Blatz. 2007. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proceedings of the Conference on Formal Grammar*. CSLI Publications.

Jason Eisner. 2016. Inside-outside and forward-backward algorithms are just backprop. In *Proceedings of the EMNLP Workshop on Structured Prediction for NLP*.

Daniel Fernández-González and André F. T. Martins. 2015. Parsing as reduction. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Nathaniel Wesley Filardo and Jason Eisner. 2012. A flexible solver for finite arithmetic circuits. In *Technical Communications of the International Conference on Logic Programming*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*.

Jenny Rose Finkel, Alex Kleeman, and Christopher D. Manning. 2008. Efficient, feature-based, conditional random field parsing. In *Proceedings of the Conference of the Association for Computational Linguistics*.

David Hall, Greg Durrett, and Dan Klein. 2014. Less grammar, more features. In *Proceedings of the Conference of the Association for Computational Linguistics*.

He He, Hal Daumé III, and Jason Eisner. 2013. Dynamic feature selection for dependency parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Liang Huang. 2008. Advanced dynamic programming in semiring and hypergraph frameworks. Material accompanying tutorials at COLING'08 and NAACL'09.

Jiarong Jiang, Adam Teichert, Hal Daumé III, and Jason Eisner. 2012. Learned prioritization for trading off accuracy and speed. In *Advances in Neural Information Processing Systems*.

Jiarong Jiang. 2014. *Efficient Non-deterministic Search in Structured Prediction: A Case Study in Syntactic Parsing*. Ph.D. thesis, University of Maryland.

Dan Klein and Christopher D. Manning. 2001. Parsing and hypergraphs. In *International Workshop on Parsing Technologies*.

Dan Klein and Christopher D. Manning. 2003. A* parsing: Fast exact Viterbi parse selection. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology*.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Zhifei Li and Jason Eisner. 2009. First- and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: The Penn treebank. *Computational Linguistics*, 19(2).

Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. 2005. Probabilistic CFG with latent annotations. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Franz Josef Och. 2003. Minimum error rate training in statistical machine translation. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Adam Pauls and Dan Klein. 2009. Hierarchical search for parsing. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology*.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology*.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Slav Petrov, Aria Haghighi, and Dan Klein. 2008. Coarse-to-fine syntactic machine translation using language projections. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Vasin Punyakanok and Dan Roth. 2001. The use of classifiers in sequential inference. In *Advances in Neural Information Processing Systems*.

Brian Roark and Kristy Hollingshead. 2008. Classifying chart cells for quadratic complexity context-free inference. In *Proceedings of the International Conference on Computational Linguistics*.

Stéphane Ross and J. Andrew Bagnell. 2011. Stability conditions for online learnability. *Computing Research Repository*, arXiv:1108.3154.

Stéphane Ross, Geoff J. Gordon, and J. Andrew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Workshop on Artificial Intelligence and Statistics*.

Alexander M. Rush and Slav Petrov. 2012. Vine pruning for efficient multi-pass dependency parsing. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics*.

Satoshi Sekine and Michael Collins. 1997. Evalb bracket scoring program. `http://nlp.cs.nyu.edu/evalb`.

David A. Smith and Jason Eisner. 2006. Minimum risk annealing for training log-linear models. In *Proceedings of the International Conference on Computational Linguistics*.

Veselin Stoyanov and Jason Eisner. 2012. Fast and accurate prediction via evidence-specific MRF structure. In *ICML Workshop on Inferning: Interactions between Inference and Learning*, Edinburgh, June. 6 pages.

Emma Strubell, Luke Vilnis, Kate Silverstein, and Andrew McCallum. 2015. Learning dynamic feature selection for fast sequential prediction. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the International Conference on Machine Learning*.

David Weiss and Ben Taskar. 2010. Structured prediction cascades. In *Proceedings of the Workshop on Artificial Intelligence and Statistics*.

Wenduan Xu, Yue Zhang, Philip Williams, and Philipp Koehn. 2013. Learning to prune: Context-sensitive pruning for syntactic MT. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *Proceedings of the Conference of the Association for Computational Linguistics*.