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 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 reward = accuracy − λ·runtime. The parameter λ ≥ 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 w₁ · · · wₙ form plausible
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 \(\beta\), known as the chart. The score \(\beta_{ikx}\) 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 \(\beta_{ijy}\) and \(\beta_{jkz}\) (lines 17–22). Additionally, we track a witness (backpointer) for each \(\beta_{ikx}\), 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 \(\pi\) to decide whether to compute the several values \(\beta_{ikx}\) 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 \(w\), policy \(\pi\)
2: **Output:** completed chart \(\beta\), derivation \(d\)
3: \(\beta := 0\)
4: **for** \(k := 1\) **to** \(n\):
5: **for** \(x\) such that \(x \rightarrow w_k \in \text{domain}(G)\):
6: \(s := G(x \rightarrow w_k | w, k)\)
7: **if** \(s > \beta_{k-1,k,x}\):
8: \(\beta_{k-1,k,x} := s\)
9: \(\text{witness}(k-1, k, x) := (k-1, k, w_k)\)
10: **for** width := 2 **to** \(n\):
11: **for** \(i := 0\) **to** \(n\) - width:
12: \(k := i + \text{width}\) \(\triangleright\) Current span is \((i,k)\)
13: \(\triangleright\) Policy determines whether to fill in this span
14: **if** \(\pi(w, i, k) = \text{prune}\):
15: **continue**
16: \(\triangleright\) Fill in span by considering each split point \(j\)
17: **for** \(j := i + 1\) **to** \(k - 1\):
18: **for** \(x \rightarrow y z \in \text{domain}(G)\):
19: \(s := \beta_{ijy} \cdot \beta_{jkz} \cdot G(x \rightarrow y z | w, i, j, k)\)
20: **if** \(s > \beta_{ikx}\):
21: \(\beta_{ikx} := s\)
22: \(\text{witness}(i, k, x) := (j, y, z)\)
23: \(\hat{d} := \text{follow backpointers from (0, n, ROOT)}\)
24: **return** \((\beta, \hat{d})\)

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 \(\pi\), we call \(\text{PARSE}(G, x_i, \pi)\) on each training sentence \(x_i\), 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:

\[ \mathcal{R}(\pi) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{E}[r(\text{PARSE}(G, x_i, \pi))] \]

The expectation in the definition may be dropped if \(\text{PARSE}\), \(\pi\), 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 $\lambda$ 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., pruning narrow constituents prevents wider ones from being built), and some even improve accuracy by suppressing an incorrect but high-scoring parse.

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 $t$. 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. In our setting, $s_0$ is derived from a randomly sampled training sentence, but the rest of the trajectory is then deterministically computed by $\pi_t$ 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 $\pi_t$ thereafter—a rollout—to measure the resulting reward $\hat{r}[\bar{a}]$ (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 $\hat{Q}_t$ of pairs $\langle s, \hat{r} \rangle$, where $s$ is a state that was encountered in some roll-in and $\hat{r}$ maps the possible actions $A(s)$ in that state to their measured rewards.

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

---

Our experiments do not make use of the current state of the chart. We discuss this decision in §8. Alg. 2 is simpler than in Chang et al. (2015b) because it omits oracle rollouts, which we do not use in our experiments.
inputs (states), outputs (actions), and costs are obtained by interacting with the environment. LOLS concretes 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 \( \pi_i \) 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 \( T = \frac{n(n+1)}{2} - 1 - n \) 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 \( \pi_i \), 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 \( \pi \) as defining a pruning...
mask \( m \), an array that maps each of the \( T \) spans \((i, k)\) to a decision \( m_{ik} \) (1 = keep, 0 = 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” \( e.\text{head} \leftarrow e.\text{Tail} \) with one output and a list of inputs. We regard each quantity \( \beta_{ikx}, m_{ik}, \) or \( G(...) \) in Alg. 1 as the value of a corresponding hypergraph vertex \( \dot{\beta}_{ikx}, m_{ik}, \) or \( \dot{G}(...) \). Thus, \( \text{value}(\dot{v}) = v \) for any vertex \( \dot{v} \). Each \( m_{ik} \)’s value is computed by the policy \( \pi \) or chosen by a rollout intervention. Each \( G \)’s value is given by the grammar.

Possible values of \( \dot{\beta}_{ikx} \), by contrast, are computed at line 19 if \( k - i > 1 \). To record the dependence of \( \beta_{ikx} \) on other quantities, our hypergraph includes the hyperedge \( \dot{\beta}_{ikx} \leftarrow (\dot{\beta}_{ijy}, \dot{\beta}_{jxz}, m_{ik}, \dot{g}) \) for each \( 0 \leq i < j < k \leq n \) and \( x \rightarrow y z \in \text{domain}(G) \), where \( \dot{g} \) denotes the vertex \( \dot{G}(x \rightarrow y z | w, i, j, k) \).

If \( k - i = 1 \), then possible values of \( \beta_{ikx} \) are instead computed at line 6, which does not access any other \( \beta \) values or the pruning mask. Thus our hypergraph includes the hyperedge \( v_{ikx} \leftarrow (\dot{g}) \) whenever \( i = k - 1, 0 \leq i < k \leq n \), and \( x \rightarrow w_k \in \text{domain}(G) \), with \( \dot{g} = \dot{G}(x \rightarrow w_k | w, k) \).

With this setup, the value \( \beta_{ikx} \) 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 \( \dot{\beta}_{ikx} \leftarrow T \) and the previously computed values of the vertices in the tail \( T \). For a vertex \( \dot{v} \), we write \( \text{In}(\dot{v}) \) and \( \text{Out}(\dot{v}) \) 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

Change propagation (CP) 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 \( r(\text{PARSE}(G, x_1, \pi)) \) 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 \( m \) 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, most interventions change < 2% of the \( \beta \) values in the chart. The reason is that \( \beta_{ikx} \) is a maximum over many quantities, only one of which “wins.” Changing a given \( \beta_{ijy} \) usually does not affect this maximum, and so changes are unlikely to propagate from vertex \( \beta_{ijy} \) to \( \beta_{ikx} \). 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.
was not pruned. In this section, we relax that to
A simple accuracy
where

Algorithm 3 Change propagation algorithm

1: Global: Alg. 1’s vertex values/witnesses (roll-in)
2: procedure CHANGE(\(v, v\))
3: \(\triangleright\) Change the value of a leaf vertex \(v\) to \(v\).
4: \(value(\hat{v}) := v\); \(witness(\hat{v}) := LEAF\)
5: \(Q := \emptyset; Q.push(\hat{v}) \triangleright work queue (“agenda”)\)
6: while \(Q \neq \emptyset\) \(\triangleright\) propagate until convergence
7: \(\hat{u} := Q.pop() \triangleright\) narrower consists first
8: if \(witness(\hat{u}) = \text{NULL}\) : \(\triangleright\) value is unknown
9: \(\text{RECOMPUTE(}\hat{u})\)
10: for \(e \in \text{Out}(\hat{u})\) \(\triangleright\) Propagate new value of \(\hat{u}\)
11: \(s := e.\text{head}; s := \prod_{u' \in e.\text{Tail}} value(u')\)
12: if \(s > value(s)\) : \(\triangleright\) increase value
13: \(value(s) := s;\) \(witness(s) := e\)
14: \(Q.push(s)\)
15: else if \(witness(s) = e\) and \(s < value(s)\):
16: \(witness(s) := \text{NULL}\) \(\triangleright\) value may decrease
17: \(Q.push(s)\) \(\triangleright\) so, recompute upon pop
18: procedure RECOMPUTE(s)
19: for \(e \in \text{ln}(s)\) \(\triangleright\) max over incoming hyperedges
20: \(s := \prod_{u \in e.\text{Tail}} value(u)\)
21: if \(s > value(s)\) :
22: \(value(s) = s;\) \(witness(s) = e\)

5.3 Dynamic programming
The naive rollout algorithm runs the parser \(T\) times—
onece for each variation of the pruning mask. The reader may be reminded of the finite difference approximiation 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-\(O\) sense) as computing the reward once. The overall algorithm is \(O(|E| + T)\) where |\(E\)| is the total number of hyperedges, whereas the naive algorithm is \(O(|E'| \cdot T)\) where |\(E'\)| 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 \(\bar{r}(\bar{d})\) of the highest-scoring derivation \(\bar{d}\) that was not pruned. In this section, we relax that to expected recall, \(\bar{r} = \sum dp(d)r(d)\). Here we interpret the pruned hypergraph’s values as an unnormalized probability distribution over derivations, where the probability \(p(d) = \bar{p}(d)/Z\) of a derivation is proportional to its score \(\bar{p}(d) = \prod_{u \in \text{leaves}(d)} value(u)\).

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 \(r(\bar{d})\) when probability mass is concentrated on a few derivations, which is common with heavy pruning.

We can re-express \(\bar{r}\) as \(\bar{r}/Z\), where
\[
\bar{r} = \sum dp(d)r(d)\quad Z = \sum dp(d) .
\]

These can be computed by a variant of the inside algorithm (Li and Eisner, 2009). Since \(\bar{p}(d)\) is a product of rule weights and pruning mask bits at \(d\)’s leaves (§5.1), each appearing at most once, both \(\bar{r}\) and \(Z\) vary linearly in any one of these inputs provided that all other inputs are held constant. Thus, the exact effect on \(\bar{r}\) 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 \(\Delta \in \{0, \pm 1\}\) (to flip this bit), the new value of \(\bar{r}\) would be exactly
\[
\bar{r} + \Delta \cdot \partial \bar{r}/\partial m_{ik} .
\]

It remains to compute these partial derivatives. All partials can be jointly computed by back-propagation: equivalent to an outside algorithm (Eisner, 2016) in the expectation semiring (Li and Eisner, 2009).

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 \(\partial \bar{r}/\partial m_{ik}\) 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.

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

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. For efficiency during training, we restrict the length of sentences to $\leq 40$. 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) gives 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 accuracy $- \lambda \cdot$ runtime, where accuracy is given by labeled F$_1$ percentage and 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 accuracy on a sentence as F$_1$ of just that sentence, and to approximate runtime as $||\beta||_0$, the number of constituents that were built$^{10}$

$^7$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 ($X \rightarrow X$), and empty subtrees left by other operations. Relabel ADVP and FRT | ADVP tags to FRT.

$^9$On the $7^{th}$ day, LOLS rested and performance was good.

$^{10}$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 $T/(2n)$.

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

12We estimate $\lambda$ by first fitting a parametric model $y_i = h(x_i) \hat{=} y_{\text{max}} \cdot \text{sigmoid}(a \cdot \log(x_i + c) + b)$ to the baseline runtime-accuracy curve on development data by minimizing mean squared error. We then use the derivative $h'$ to estimate each $\lambda_i = h'(x_i)$, where $x_i$ is the runtime of baseline policy $i$. Our parametric model is a sigmoidal curve, with accuracy $\rightarrow y_{\text{max}}$ asymptotically as runtime $\rightarrow \infty$, and accuracy $\rightarrow 0$ as runtime $\rightarrow 0$. It obtains an excellent fit by placing accuracy and runtime on the log-logit scale—that is, log$(x_i + c)$ and log$(y_i/y_{\text{max}})$ transforms are used to convert our bounded random variables $x_i$ and $y_i$ to unbounded ones—and assuming they are linearly related.

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 $\lambda$ 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.

7.1 Discussion

As previous work has shown, learning to prune gives us excellent parsers with less than $< 2\%$ overhead 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 don’t 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 constituents are “safest” to prune, nor can it learn strategies such as “skip all long sentences.” We dis-

by upweighting the resulting training examples by $T/(2n)$.

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

12We estimate $\lambda$ by first fitting a parametric model $y_i = h(x_i) \hat{=} y_{\text{max}} \cdot \text{sigmoid}(a \cdot \log(x_i + c) + b)$ to the baseline runtime-accuracy curve on development data by minimizing mean squared error. We then use the derivative $h'$ to estimate each $\lambda_i = h'(x_i)$, where $x_i$ is the runtime of baseline policy $i$. Our parametric model is a sigmoidal curve, with accuracy $\rightarrow y_{\text{max}}$ asymptotically as runtime $\rightarrow \infty$, and accuracy $\rightarrow 0$ as runtime $\rightarrow 0$. It obtains an excellent fit by placing accuracy and runtime on the log-logit scale—that is, log$(x_i + c)$ and log$(y_i/y_{\text{max}})$ transforms are used to convert our bounded random variables $x_i$ and $y_i$ to unbounded ones—and assuming they are linearly related.

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 $\lambda$ 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.

7.1 Discussion

As previous work has shown, learning to prune gives us excellent parsers with less than $< 2\%$ overhead 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 don’t 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 constituents are “safest” to prune, nor can it learn strategies such as “skip all long sentences.” We dis-

by upweighting the resulting training examples by $T/(2n)$.

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

12We estimate $\lambda$ by first fitting a parametric model $y_i = h(x_i) \hat{=} y_{\text{max}} \cdot \text{sigmoid}(a \cdot \log(x_i + c) + b)$ to the baseline runtime-accuracy curve on development data by minimizing mean squared error. We then use the derivative $h'$ to estimate each $\lambda_i = h'(x_i)$, where $x_i$ is the runtime of baseline policy $i$. Our parametric model is a sigmoidal curve, with accuracy $\rightarrow y_{\text{max}}$ asymptotically as runtime $\rightarrow \infty$, and accuracy $\rightarrow 0$ as runtime $\rightarrow 0$. It obtains an excellent fit by placing accuracy and runtime on the log-logit scale—that is, log$(x_i + c)$ and log$(y_i/y_{\text{max}})$ transforms are used to convert our bounded random variables $x_i$ and $y_i$ to unbounded ones—and assuming they are linearly related.
cuss why LOLS doesn’t 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- $\lambda$ 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 didn’t improve from its initial policy.

We find that when \( \lambda \) 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 many iterations within our 6-day limit; usually it was still improving when we terminated it. By contrast, for the more accurate small-\( \lambda \) parsers, 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 $\hat{Q}_i$ tell us what actions LOLS is trying to improve in its current policy $\pi_i$ for a given $\lambda$, although there is no guarantee that the learner in §4 will succeed at classifying $\hat{Q}_i$ correctly (due to limited features, regularization, and the effects of dataset aggregation).

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

$$E[\text{regret}] = p(\text{diff}) \cdot \left( p(\text{gold} | \text{diff}) \cdot E[\text{regret} | \text{gold, diff}] + p(\text{non-gold} | \text{diff}) \cdot E[\text{regret} | \text{non-gold, diff}] \right),$$

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

**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 $\pi, \lambda$, and grammar. The LOLS oracle evolves along with the policy $\pi$, since it identifies the best action given $\pi$. We thus evaluate the oracle baseline against two LOLS oracles: the one used at the start of LOLS training (derived from the initial policy $\pi_I$ that was trained on baseline supervision), and the one obtained at the end (derived from the LOLS-trained policy $\pi_e$ 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 $\lambda$ 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 $\lambda$, and this difference comes especially from the top row. With a fine grammar and small $\lambda$, 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 $\lambda$:** Aggressive pruning (large $\lambda$) 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 $\tau_{DP}$ 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 $\lambda$) but not when it is mostly pruned (large $\lambda$).

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

**Pruning** methods 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, Jiang et al. (2012) and Berant and Liang (2015) train prioritization heuristics (using policy gradient). Some previous work uses coarse-to-fine methods to automatically derive 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). 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 $\pi(w, i, k)$. This permits our fast CP and DP rollout algorithms. It also reduces the time and space cost of dataset aggregation.

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

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.
LOLS attempts to do \textit{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 \textit{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.\textsuperscript{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 haven’t (yet) been designed.

\section{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.

\section*{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 for suggestions that improved this paper.

\section*{References}


