Learning to Search Better than Your Teacher

Kai-Wei Chang KCHANG10@ILLINOIS.EDU
University of Illinois at Urbana Champaign, IL

Akshay Krishnamurthy AKSHAYKR@CS.CMU.EDU
Carnegie Mellon University, Pittsburgh, PA

Alekh Agarwal ALEKHA@MICROSOFT.COM
Microsoft Research, New York, NY

Hal Daumé III HAL@UMIACS.UMD.EDU
University of Maryland, College Park, MD

John Langford JCL@MICROSOFT.COM
Microsoft Research, New York, NY

Abstract

Methods for learning to search for structured prediction typically imitate a reference policy, with existing theoretical guarantees demonstrating low regret compared to that reference. This is unsatisfactory in many applications where the reference policy is suboptimal and the goal of learning is to improve upon it. Can learning to search work even when the reference is poor?

We provide a new learning to search algorithm, LOLS, which does well relative to the reference policy, but *additionally* guarantees low regret compared to *deviations* from the learned policy: a local-optimality guarantee. Consequently, LOLS can improve upon the reference policy, unlike previous algorithms. This enables us to develop *structured contextual bandits*, a partial information structured prediction setting with many potential applications.

1. Introduction

In structured prediction problems, a learner makes joint predictions over a set of interdependent output variables and observes a joint loss. For example, in a parsing task, the output is a parse tree over a sentence. Achieving optimal performance commonly requires the prediction of each output variable to depend on neighboring variables. One approach to structured prediction is *learning to search* (L2S) (Collins & Roark, 2004; Daumé III & Marcu, 2005; Daumé III et al., 2009; Ross et al., 2011; Doppa et al., 2014; Ross & Bagnell, 2014), which solves the problem by:

1. converting structured prediction into a search problem with specified search space and actions;
2. defining structured features over each state to capture the interdependency between output variables;
3. constructing a reference policy based on training data;
4. learning a policy that *imitates* the reference policy.

Empirically, L2S approaches have been shown to be competitive with other structured prediction approaches both in accuracy and running time (see e.g. Daumé III et al. (2014)). Theoretically, existing L2S algorithms guarantee that if the learning step performs well, then the learned policy is almost as good as the reference policy, implicitly assuming that the reference policy attains good performance. Good reference policies are typically derived using labels in the training data, such as assigning each word to its correct POS tag. However, when the reference policy is suboptimal, which can arise for reasons such as computational constraints, nothing can be said for existing approaches.

This problem is most obviously manifest in a "structured contextual bandit"[1] setting. For example, one might want to predict how the landing page of a high profile web-

Proceedings of the 32^{nd} International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37.

[1]The key difference from (1) contextual bandits is that the action space is exponentially large (in the length of trajectories in the search space); and from (2) reinforcement learning is that a baseline reference policy exists before learning starts.

site should be displayed; this involves many interdependent predictions: items to show, position and size of those items, font, color, layout, etc. It may be plausible to derive a quality signal for the displayed page based on user feedback, and we may have access to a reasonable reference policy (namely the existing rule-based system that renders the current web page). But, applying L2S techniques results in nonsense—learning something almost as good as the existing policy is useless as we can just keep using the current system and obtain that guarantee. Unlike the full feedback settings, label information is not even available during learning to define a substantially better reference. The goal of learning here is to improve upon the current system, which is most likely far from optimal. This naturally leads to the question: *is learning to search useless when the reference policy is poor?*

This is the core question of the paper, which we address first with a new L2S algorithm, LOLS (Locally Optimal Learning to Search) in Section 2. LOLS operates in an online fashion and achieves a bound on a convex combination of regret-to-reference and regret-to-own-one-step-deviations. The first part ensures that good reference policies can be leveraged effectively; the second part ensures that even if the reference policy is very sub-optimal, the learned policy is approximately "locally optimal" in a sense made formal in Section 3.

LOLS operates according to a general schematic that encompases many past L2S algorithms (see Section 2), including Searn (Daumé III et al., 2009), DAgger (Ross et al., 2011) and AggreVaTe (Ross & Bagnell, 2014). A secondary contribution of this paper is a theoretical analysis of both good and bad ways of instantiating this schematic under a variety of conditions, including: whether the reference policy is optimal or not, and whether the reference policy is in the hypothesis class or not. We find that, while past algorithms achieve good regret guarantees *when the reference policy is optimal*, they can fail rather dramatically when it is not. LOLS, on the other hand, has superior performance to other L2S algorithms when the reference policy performs poorly but local hill-climbing in policy space is effective. In Section 5, we empirically confirm that LOLS can significantly outperform the reference policy in practice on real-world datasets.

In Section 4 we extend LOLS to address the structured contextual bandit setting, giving a natural modification to the algorithm as well as the corresponding regret analysis.

The proofs of our main results, and the details of the cost-sensitive classifier used in experiments are deferred to the appendix. The algorithm LOLS, the new kind of regret guarantee it satisfies, the modifications for the structured contextual bandit setting, and all experiments are new here.

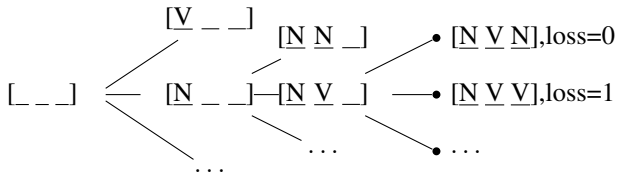


Figure 1. An illustration of the search space of a sequential tagging example that assigns a part-of-speech tag sequence to the sentence "John saw Mary." Each state represents a partial labeling. The start state $b = [\_\ \_\ \_]$ and the set of end states $E = \{[\underline{N}\ \underline{V}\ \underline{N}], [\underline{N}\ \underline{V}\ \underline{V}], \ldots\}$. Each end state is associated with a loss. A policy chooses an action at each state in the search space to specify the next state.

2. Learning to Search

A structured prediction problem consists of an *input space* $\mathcal{X}$, an *output space* $\mathcal{Y}$, a fixed but unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, and a non-negative *loss function* $\ell(\boldsymbol{y}^*, \hat{\boldsymbol{y}}) \to \mathbb{R}^{\geq 0}$ which measures the distance between the true ($\boldsymbol{y}^*$) and predicted ($\hat{\boldsymbol{y}}$) outputs. The goal of structured learning is to use N samples $(\boldsymbol{x}_i, \boldsymbol{y}_i)_{i=1}^N$ to learn a mapping $f : \mathcal{X} \to \mathcal{Y}$ that minimizes the expected structured loss under $\mathcal{D}$.

In the learning to search framework, an input $\boldsymbol{x} \in \mathcal{X}$ induces a search space, consisting of an initial state b (which we will take to also encode $\boldsymbol{x}$), a set of end states and a transition function that takes state/action pairs s, a and deterministically transitions to a new state s'. For each end state e, there is a corresponding structured output $\boldsymbol{y}_e$ and for convenience we define the loss $\ell(e) = \ell(\boldsymbol{y}^*, \boldsymbol{y}_e)$ where $\boldsymbol{y}^*$ will be clear from context. We futher define a feature generating function Φ that maps states to feature vectors in $\mathbb{R}^d$. The features express both the input $\boldsymbol{x}$ and previous predictions (actions). Fig. 1 shows an example search space[2].

An agent follows a *policy* $\pi \in \Pi$, which chooses an *action* $a \in A(s)$ at each non-terminal state s. An action specifies the next state from s. We consider policies that only access state s through its feature vector $\Phi(s)$, meaning that $\pi(s)$ is a mapping from $\mathbb{R}^d$ to the set of actions $A(s)$. A *trajectory* is a complete sequence of state/action pairs from the starting state b to an end state e. Trajectories can be generated by repeatedly executing a policy π in the search space. Without loss of generality, we assume the lengths of trajectories are fixed and equal to T. The expected loss of a policy $J(\pi)$ is the expected loss of the end state of the trajectory $e \sim \pi$, where $e \in E$ is an end state reached by following the policy[3]. Throughout, expectations are taken with

[2] Doppa et al. (2014) discuss several approaches for defining a search space. The theoretical properties of our approach do not depend on which search space definition is used.

[3] Some imitation learning literature (e.g., (Ross et al., 2011; He et al., 2012)) defines the loss of a policy as an accumulation of the costs of states and actions in the trajectory generated by the policy. For simplicity, we define the loss only based on the end

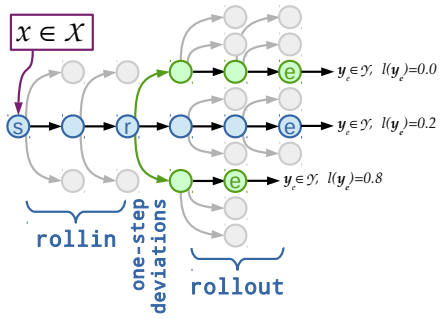


Figure 2. An example search space. The exploration begins at the start state s and chooses the middle among three actions by the **roll-in** policy twice. Grey nodes are not explored. At state r the learning algorithm considers the chosen action (middle) *and* both one-step deviations from that action (top and bottom). Each of these deviations is completed using the **roll-out** policy until an end state is reached, at which point the loss is collected. Here, we learn that deviating to the top action (instead of middle) at state r decreases the loss by 0.2.

respect to draws of $(\boldsymbol{x}, \boldsymbol{y})$ from the training distribution, as well as any internal randomness in the learning algorithm.

An optimal policy chooses the action leading to the minimal expected loss at each state. For losses decomposable over the states in a trajectory, generating an optimal policy is trivial given $\boldsymbol{y}^*$ (e.g., the sequence tagging example in (Daumé III et al., 2009)). In general, finding the optimal action at states not in the optimal trajectory can be tricky (e.g., (Goldberg & Nivre, 2013; Goldberg et al., 2014)).

Finally, like most other L2S algorithms, LOLS assumes access to a cost-sensitive classification algorithm. A cost-sensitive classifier predicts a label $\hat{y}$ given an example $\boldsymbol{x}$, and receives a loss $\boldsymbol{c}_{\boldsymbol{x}}(\hat{y})$, where $\boldsymbol{c}_{\boldsymbol{x}}$ is a vector containing the cost for each possible label. In order to perform online updates, we assume access to a no-regret online cost-sensitive learner, which we formally define below.

Definition 1. *Given a hypothesis class $\mathcal{H} : \mathcal{X} \to [K]$, the regret of an online cost-sensitive classification algorithm which produces hypotheses $h_1, \ldots, h_M$ on cost-sensitive example sequence $\{(\boldsymbol{x}_1, \boldsymbol{c}_1), \ldots, (\boldsymbol{x}_M, \boldsymbol{c}_M)\}$ is*

$$\text{Regret}^{\text{cs}}_M = \sum_{m=1}^{M} \boldsymbol{c}_m(h_m(\boldsymbol{x}_m)) - \min_{h \in \mathcal{H}} \sum_{m=1}^{M} \boldsymbol{c}_m(h(\boldsymbol{x}_m)). \tag{1}$$

An algorithm is no-regret if $\text{Regret}^{\text{cs}}_M = o(M)$.

Such no-regret guarantees can be obtained, for instance, by applying the SECOC technique (Langford & Beygelzimer, 2005) on top of any importance weighted binary classification algorithm that operates in an online fashion, examples being the perceptron algorithm or online ridge regression.

state. However, our theorems can be generalized.

Algorithm 1 Locally Optimal Learning to Search (LOLS)

Require: Dataset $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$ drawn from $\mathcal{D}$ and $\beta \geq 0$: a mixture parameter for roll-out.

```
Initialize a policy π_0.
for all i ∈ {1, 2, ..., N} (loop over each instance) do
    Generate a reference policy π^ref based on y_i.
    Initialize Γ = ∅.
    for all t ∈ {0, 1, 2, ..., T − 1} do
        Roll-in by executing π_i^in = π̂_i for t rounds and reach s_t.
        for all a ∈ A(s_t) do
            Let π_i^out = π^ref with probability β, otherwise π̂_i.
            Evaluate cost c_{i,t}(a) by rolling-out with π_i^out for T − t − 1 steps.
        end for
        Generate a feature vector Φ(x_i, s_t).
        Set Γ = Γ ∪ {⟨c_{i,t}, Φ(x_i, s_t)⟩}.
    end for
    π̂_{i+1} ← Train(π̂_i, Γ) (Update).
end for
Return the average policy across π̂_0, π̂_1, ... π̂_N.
```

LOLS (see Algorithm 1) learns a policy $\hat{\pi} \in \Pi$ to approximately minimize $J(\pi)$,[4] assuming access to a reference policy π^{ref} (which may or may not be optimal). The algorithm proceeds in an online fashion generating a sequence of learned policies $\hat{\pi}_0, \hat{\pi}_1, \hat{\pi}_2, \ldots$. At round i, a structured sample $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ is observed, and the configuration of a search space is generated along with the reference policy π^{ref}. Based on $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, LOLS constructs T cost-sensitive multiclass examples using a roll-in policy π_i^{in} and a roll-out policy π_i^{out}. The roll-in policy is used to generate an initial trajectory and the roll-out policy is used to derive the expected loss. More specifically, for each decision point $t \in [0, T)$, LOLS executes π_i^{in} for t rounds reaching a state $s_t \sim \pi_i^{\text{in}}$. Then, a cost-sensitive multiclass example is generated using the features $\Phi(s_t)$. Classes in the multiclass example correspond to available actions in state s_t. The cost $c(a)$ assigned to action a is the difference in loss between taking action a and the best action.

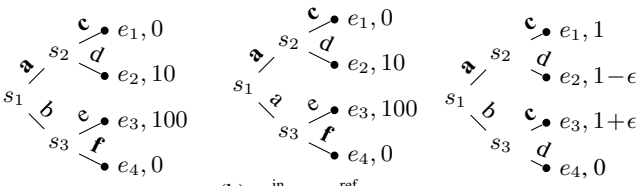
$$c(a) = \ell(e(a)) - \min_{a'} \ell(e(a')), \tag{2}$$

where $e(a)$ is the end state reached with rollout by π_i^{out} after taking action a in state s_t. LOLS collects the T examples from the different roll-out points and feeds the set of examples Γ into an online cost-sensitive multiclass learner, thereby updating the learned policy from $\hat{\pi}_i$ to $\hat{\pi}_{i+1}$. By default, we use the learned policy $\hat{\pi}_i$ for roll-in and a mixture

[4] We can parameterize the policy $\hat{\pi}$ using a weight vector $\boldsymbol{w} \in \mathbb{R}^d$ such that a cost-sensitive classifier can be used to choose an action based on the features at each state. We do not consider using different weight vectors at different states.

roll-out → / ↓ roll-in	**Reference**	**Mixture**	**Learned**
Reference	Inconsistent		
Learned	Not locally opt.	Good	RL

Table 1. Effect of different roll-in and roll-out policies. The strategies marked with "*Inconsistent*" might generate a learned policy with a large structured regret, and the strategies marked with "*Not locally opt.*" could be much worse than its one step deviation. The strategy marked with "*RL*" reduces the structure learning problem to a reinforcement learning problem, which is much harder. The strategy marked with "*Good*" is favored.



(a) $\pi_i^{\text{in}} = \pi_i^{\text{out}} = \pi^{\text{ref}}$ (b) $\pi_i^{\text{in}} = \pi^{\text{ref}}$, representation constrained (c) $\pi_i^{\text{out}} = \pi^{\text{ref}}$

Figure 3. Counterexamples of $\pi_i^{\text{in}} = \pi^{\text{ref}}$ and $\pi_i^{\text{out}} = \pi^{\text{ref}}$. All three examples have 7 states. The loss of each end state is specified in the figure. A policy chooses actions to traverse through the search space until it reaches an end state. Legal policies are bit-vectors, so that a policy with a weight on a goes up in s_1 of Figure 3(a) while a weight on b sends it down. Since features uniquely identify actions of the policy in this case, we just mark the edges with corresponding features for simplicity. The reference policy is bold-faced. In Figure 3(b), the features are the same on either branch from s_1, so that the learned policy can do no better than pick randomly between the two. In Figure 3(c), states s_2 and s_3 share the same feature set (i.e., $\Phi(s_2) = \Phi(s_3)$). Therefore, a policy chooses the same set of actions at states s_2 and s_3. Please see text for details.

policy for roll-out. For each roll-out, the mixture policy either executes π^{ref} to an end-state with probability β or $\hat{\pi}_i$ with probability $1 - \beta$. LOLS converts into a batch algorithm with a standard online-to-batch conversion where the final model $\bar{\pi}$ is generated by averaging $\hat{\pi}_i$ across all rounds (i.e., picking one of $\hat{\pi}_1, \dots \hat{\pi}_N$ uniformly at random).

3. Theoretical Analysis

In this section, we analyze LOLS and answer the questions raised in Section 1. Throughout this section we use $\bar{\pi}$ to denote the average policy obtained by first choosing $n \in [1, N]$ uniformly at random and then acting according to π_n.We begin with discussing the choices of roll-in and roll-out policies. Table 1 summarizes the results of using different strategies for roll-in and roll-out.

3.1. The *Bad* Choices

An obvious *bad* choice is roll-in and roll-out with the learned policy, because the learner is blind to the reference policy. It reduces the structured learning problem to a reinforcement learning problem, which is much harder. To build intuition, we show two other *bad* cases.

Roll-in with π^{ref} is *bad*. Roll-in with a reference policy causes the state distribution to be unrealistically good. As a result, the learned policy never learns to correct for previous mistakes, performing poorly when testing. A related discussion can be found at Theorem 2.1 in (Ross & Bagnell, 2010). We show a theorem below.

Theorem 1. *For $\pi_i^{in} = \pi^{ref}$, there is a distribution D over $(\boldsymbol{x}, \boldsymbol{y})$ such that the induced cost-sensitive regret $\text{Regret}_M^{\text{cs}} = o(M)$ but $J(\bar{\pi}) - J(\pi^{ref}) = \Omega(1)$.*

Proof. We demonstrate examples where the claim is true.

We start with the case where $\pi_i^{\text{out}} = \pi_i^{\text{in}} = \pi^{\text{ref}}$. In this case, suppose we have one structured example, whose search space is defined as in Figure 3(a). From state s_1, there are two possible actions: a and b (we will use actions and features interchangeably since features uniquely identify actions here); the (optimal) reference policy takes action a. From state s_2, there are again two actions (c and d); the reference takes c. Finally, even though the reference policy would never visit s_3, from that state it chooses action f. When rolling in with π^{ref}, the cost-sensitive examples are generated only at state s_1 (if we take a one-step deviation on s_1) and s_2 but *never* at s_3 (since that would require a two deviations, one at s_1 and one at s_3). As a result, we can never learn how to make predictions at state s_3. Furthermore, under a rollout with π^{ref}, both actions from state s_1 lead to a loss of zero. The learner can therefore learn to take action c at state s_2 and b at state s_1, and achieve *zero* cost-sensitive regret, thereby "thinking" it is doing a good job. Unfortunately, when this policy is actually run, it performs as badly as possible (by taking action e half the time in s_3), which results in the large structured regret.

Next we consider the case where π_i^{out} is either the learned policy or a mixture with π^{ref}. When applied to the example in Figure 3(b), our feature representation is not expressive enough to differentiate between the two actions at state s_1, so the learned policy can do no better than pick randomly between the top and bottom branches from this state. The algorithm either rolls in with π^{ref} on s_1 and generates a cost-sensitive example at s_2, or generates a cost-sensitive example on s_1 and then completes a roll out with π_i^{out}. Crucially, the algorithm still never generates a cost-sensitive example at the state s_3 (since it would have already taken a one-step deviation to reach s_3 and is constrained to do a roll out from s_3). As a result, if the learned policy were to

choose the action e in s_3, it leads to a zero cost-sensitive regret but large structured regret. □

Despite these negative results, rolling in with the learned policy is robust to both the above failure modes. In Figure 3(a), if the learned policy picks action b in state s_1, then we can roll in to the state s_3, then generate a cost-sensitive example and learn that f is a better action than e. Similarly, we also observe a cost-sensitive example in s_3 in the example of Figure 3(b), which clearly demonstrates the benefits of rolling in with the learned policy as opposed to π^{ref}.

Roll-out with π^{ref} is *bad* if π^{ref} is not optimal. When the reference policy is not optimal *or* the reference policy is not in the hypothesis class, roll-out with π^{ref} can make the learner blind to compounding errors. The following theorem holds. We state this in terms of "local optimality": a policy is locally optimal if changing any *one* decision it makes never improves its performance.

Theorem 2. *For $\pi_i^{out} = \pi^{ref}$, there is a distribution D over $(\boldsymbol{x}, \boldsymbol{y})$ such that the induced cost-sensitive regret $\text{Regret}_M^{\text{CS}} = o(M)$ but $\bar{\pi}$ has arbitrarily large structured regret to one-step deviations.*

Proof. Suppose we have only one structured example, whose search space is defined as in Figure 3(c) and the reference policy chooses a or c depending on the node. If we roll-out with π^{ref}, we observe expected losses 1 and $1+\epsilon$ for actions a and b at state s_1, respectively. Therefore, the policy with zero cost-sensitive classification regret chooses actions a and d depending on the node. However, a one step deviation ($a \to b$) does radically better and can be learned by instead rolling out with a mixture policy. □

The above theorems show the bad cases and motivate a good L2S algorithm which generates a learned policy that competes with the reference policy and deviations from the learned policy. In the following section, we show that Algorithm 1 is such an algorithm.

3.2. Regret Guarantees

Let $Q^\pi(s_t, a)$ represent the expected loss of executing action a at state s_t and then executing policy π until reaching an end state. T is the number of decisions required before reaching an end state. For notational simplicity, we use $Q^\pi(s_t, \pi')$ as a shorthand for $Q^\pi(s_t, \pi'(s_t))$, where $\pi'(s_t)$ is the action that π' takes at state s_t. Finally, we use d_π^t to denote the distribution over states at time t when acting according to the policy π. The expected loss of a policy is:

$$J(\pi) = \mathbb{E}_{s\sim d_\pi^t}\left[Q^\pi(s,\pi)\right], \tag{3}$$

for any $t \in [0, T]$. In words, this is the expected cost of rolling in with π up to some time t, taking π's action at time t and then completing the roll out with π.

Our main regret guarantee for Algorithm 1 shows that LOLS minimizes a combination of regret to the reference policy π^{ref} and regret its own one-step deviations. In order to concisely present the result, we present an additional definition which captures the regret of our approach:

$$\delta_N = \frac{1}{NT}\sum_{i=1}^{N}\sum_{t=1}^{T}\mathbb{E}_{s\sim d_{\hat{\pi}_i}^t}\Big[Q^{\pi_i^{\text{out}}}(s,\hat{\pi}_i) - \Big(\beta\min_a Q^{\pi^{\text{ref}}}(s,a) + (1-\beta)\min_a Q^{\hat{\pi}_i}(s,a)\Big)\Big], \tag{4}$$

where $\pi_i^{\text{out}} = \beta\pi^{\text{ref}} + (1-\beta)\hat{\pi}_i$ is the mixture policy used to roll-out in Algorithm 1. With these definitions in place, we can now state our main result for Algorithm 1.

Theorem 3. *Let δ_N be as defined in Equation 4. The averaged policy $\bar{\pi}$ generated by running N steps of Algorithm 1 with a mixing parameter β satisfies*

$$\beta(J(\bar{\pi}) - J(\pi^{ref})) + (1-\beta)\sum_{t=1}^{T}\left(J(\bar{\pi}) - \min_{\pi\in\Pi}\mathbb{E}_{s\sim d_{\bar{\pi}}^t}[Q^{\bar{\pi}}(s,\pi)]\right) \le T\delta_N.$$

It might appear that the LHS of the theorem combines one term which is constant to another scaling with T. We point the reader to Lemma 1 in the appendix to see why the terms are comparable in magnitude. Note that the theorem does not assume anything about the quality of the reference policy, and it might be arbitrarily suboptimal. Assuming that Algorithm 1 uses a no-regret cost-sensitive classification algorithm (recall Definition 1), the first term in the definition of δ_N converges to

$$\ell^* = \min_{\pi\in\Pi}\frac{1}{NT}\sum_{i=1}^{N}\sum_{t=1}^{T}\mathbb{E}_{s\sim d_{\hat{\pi}_i}^t}[Q^{\pi_i^{\text{out}}}(s,\pi)].$$

This observation is formalized in the next corollary.

Corollary 1. *Suppose we use a no-regret cost-sensitive classifier in Algorithm 1. As $N \to \infty$, $\delta_N \to \delta_{class}$, where*

$$\delta_{class} = \ell^* - \frac{1}{NT}\sum_{i,t}\mathbb{E}_{s\sim d_{\hat{\pi}_i}^t}\Big[\beta\min_a Q^{\pi^{ref}}(s,a) + (1-\beta)\min_a Q^{\hat{\pi}_i}(s,a)\Big].$$

When we have $\beta = 1$, so that LOLS becomes almost identical to AGGREVATE (Ross & Bagnell, 2014), δ_{class} arises solely due to the policy class Π being restricted. For other values of $\beta \in (0,1)$, the asymptotic gap does not always vanish even if the policy class is unrestricted, since ℓ^* amounts to obtaining $\min_a Q^{\pi_i^{\text{out}}}(s,a)$ in each state. This corresponds to taking a minimum of an average rather than the average of the corresponding minimum values.

In order to avoid this asymptotic gap, it seems desirable to have regrets to reference policy and one-step deviations

controlled individually, which is equivalent to having the guarantee of Theorem 3 for all values of β in $[0, 1]$ rather than a specific one. As we show in the next section, guaranteeing a regret bound to one-step deviations when the reference policy is arbitrarily bad is rather tricky and can take an exponentially long time. Understanding structures where this can be done more tractably is an important question for future research. Nevertheless, the result of Theorem 3 has interesting consequences in several settings, some of which we discuss next.

1. The second term on the left in the theorem is always non-negative by definition, so the conclusion of Theorem 3 is at least as powerful as existing regret guarantee to reference policy when $\beta = 1$. Since the previous works in this area (Daumé III et al., 2009; Ross et al., 2011; Ross & Bagnell, 2014) have only studied regret guarantees to the reference policy, the quantity we're studying is strictly more difficult.
2. The asymptotic regret incurred by using a mixture policy for roll-out might be larger than that using the reference policy alone, when the reference policy is near-optimal. How the combination of these factors manifests in practice is empirically evaluated in Section 5.
3. When the reference policy is optimal, the first term is non-negative. Consequently, the theorem demonstrates that our algorithm competes with one-step deviations in this case. This is true irrespective of whether π^{ref} is in the policy class Π or not.
4. When the reference policy is very suboptimal, then the first term can be negative. In this case, the regret to one-step deviations can be large despite the guarantee of Theorem 3, since the first negative term allows the second term to be large while the sum stays bounded. However, when the first term is significantly negative, then the learned policy has already improved upon the reference policy substantially! This ability to improve upon a poor reference policy by using a mixture policy for rolling out is an important distinction for Algorithm 1 compared with previous approaches.

Overall, Theorem 3 shows that the learned policy is either competitive with the reference policy *and* nearly locally optimal, or improves substantially upon the reference policy.

3.3. Hardness of local optimality

In this section we demonstrate that the process of reaching a local optimum (under one-step deviations) can be exponentially slow when the initial starting policy is arbitrary. This reflects the hardness of learning to search problems when equipped with a poor reference policy, even if local rather than global optimality is considered a yardstick. We establish this lower bound for a class of algorithms substantially more powerful than LOLS. We start by defining a search space and a policy class. Our search space consists of trajectories of length T, with 2 actions available at each step of the trajectory. We use 0 and 1 to index the two actions. We consider policies whose only feature in a state is the depth of the state in the trajectory, meaning that the action taken by any policy π in a state s_t depends only on t. Consequently, each policy can be indexed by a bit string of length T. For instance, the policy $0100\ldots0$ executes action 0 in the first step of any trajectory, action 1 in the second step and 0 at all other levels. It is easily seen that two policies are one-step deviations of each other if the corresponding bit strings have a Hamming distance of 1.

To establish a lower bound, consider the following powerful algorithmic pattern. Given a current policy π, the algorithm examines the cost $J(\pi')$ for all the one-step deviations π' of π. It then chooses the policy with the smallest cost as its new learned policy. Note that access to the actual costs $J(\pi)$ makes this algorithm more powerful than existing L2S algorithms, which can only estimate costs of policies through rollouts on individual examples. Suppose this algorithm starts from an initial policy $\hat{\pi}_0$. How long does it take for the algorithm to reach a policy $\hat{\pi}_i$ which is locally optimal compared with all its one-step deviations? We next present a lower bound for algorithms of this style.

Theorem 4. *Consider any algorithm which updates policies only by moving from the current policy to a one-step deviation. Then there is a search space, a policy class and a cost function where the any such algorithm must make $\Omega(2^T)$ updates before reaching a locally optimal policy. Specifically, the lower bound also applies to Algorithm 1.*

The result shows that competing with the seemingly reasonable benchmark of one-step deviations may be very challenging from an algorithmic perspective, at least without assumptions on the search space, policy class, loss function, or starting policy. For instance, the construction used to prove Theorem 4 does not apply to Hamming loss.

4. Structured Contextual Bandit

We now show that a variant of LOLS can be run in a "structured contextual bandit" setting, where only the loss of a single structured label can be observed. As mentioned, this setting has applications to webpage layout, personalized search, and several other domains.

At each round, the learner is given an input example $\boldsymbol{x}$, makes a prediction $\hat{\boldsymbol{y}}$ and suffers structured loss $\ell(\boldsymbol{y}^*, \hat{\boldsymbol{y}})$. We assume that the structured losses lie in the interval $[0, 1]$, that the search space has depth T and that there are at most K actions available at each state. As before, the algorithm has access to a policy class Π, and also to a reference policy π^{ref}. It is important to emphasize that the reference policy does not have access to the true label, and the goal

Algorithm 2 Structured Contextual Bandit Learning

Require: Examples $\{\boldsymbol{x}_i\}_{i=1}^N$, reference policy π^{ref}, exploration probability ϵ and mixture parameter $\beta \geq 0$.

1: Initialize a policy π_0, and set $\mathcal{I} = \emptyset$.
2: **for all** $i = 1, 2, \ldots, N$ (loop over each instance) **do**
3: Obtain the example $\boldsymbol{x}_i$, set `explore` $= 1$ with probability ϵ, set $n_i = |\mathcal{I}|$.
4: **if** `explore` **then**
5: Pick random time $t \in \{0, 1, \ldots, T-1\}$.
6: Roll-in by executing $\pi_i^{\text{in}} = \hat{\pi}_{n_i}$ for t rounds and reach s_t.
7: Pick random action $a_t \in A(s_t)$; let $K = |A(s_t)|$.
8: Let $\pi_i^{\text{out}} = \pi^{\text{ref}}$ with probability β, otherwise $\hat{\pi}_{n_i}$.
9: Roll-out with π_i^{out} for $T - t - 1$ steps to evaluate

$$\hat{c}(a) = K\ell(e(a_t))\mathbf{1}[a = a_t].$$

10: Generate a feature vector $\Phi(\boldsymbol{x}_i, s_t)$.
11: $\hat{\pi}_{n_i+1} \leftarrow \text{Train}(\hat{\pi}_{n_i}, \hat{c}, \Phi(\boldsymbol{x}_i, s_t))$.
12: Augment $\mathcal{I} = \mathcal{I} \cup \{\hat{\pi}_{n_i+1}\}$
13: **else**
14: Follow the trajectory of a policy π drawn randomly from $\mathcal{I}$ to an end state e, predict the corresponding structured output $\boldsymbol{y}_{ie}$.
15: **end if**
16: **end for**

is improving on the reference policy.

Our approach is based on the ϵ-greedy algorithm which is a common strategy in partial feedback problems. Upon receiving an example $\boldsymbol{x}_i$, the algorithm randomly chooses whether to *explore or exploit* on this example. With probability $1-\epsilon$, the algorithm chooses to exploit and follows the recommendation of the current learned policy. With the remaining probability, the algorithm performs a randomized variant of the LOLS update. A detailed description is given in Algorithm 2.

We assess the algorithm's performance via a measure of regret, where the comparator is a mixture of the reference policy and the best one-step deviation. Let $\bar{\pi}_i$ be the averaged policy based on all policies in $\mathcal{I}$ at round i. $\boldsymbol{y}_{ie}$ is the predicted label in either step 9 or step 14 of Algorithm 2. The average regret is defined as:

$$\text{Regret} = \frac{1}{N}\sum_{i=1}^{N}\Big(\mathbb{E}[\ell(\boldsymbol{y}_i^*, \boldsymbol{y}_{ie})] - \beta\mathbb{E}[\ell(\boldsymbol{y}_i^*, \boldsymbol{y}_{ie_{\text{ref}}})] - (1-\beta)\sum_{t=1}^{T}\min_{\pi\in\Pi}\mathbb{E}_{s\sim d_{\bar{\pi}_i}^t}[Q^{\bar{\pi}_i}(s,\pi)]\Big)$$

Recalling our earlier definition of δ_i (4), we bound on the regret of Algorithm 2 with a proof in the appendix.

Theorem 5. *Algorithm 2 with parameter ϵ satisfies:*

$$\textit{Regret} \leq \epsilon + \frac{1}{N}\sum_{i=1}^{N}\delta_{n_i},$$

With a no-regret learning algorithm, we expect

$$\delta_i \leq \delta_{\text{class}} + cK\sqrt{\frac{\log|\Pi|}{i}}, \tag{5}$$

where $|\Pi|$ is the cardinality of the policy class. This leads to the following corollary with a proof in the appendix.

Corollary 2. *In the setup of Theorem 5, suppose further that the underlying no-regret learner satisfies (5). Then with probability at least $1 - 2/(N^5K^2T^2\log(N|\Pi|))^3$,*

$$\textit{Regret} = O\left((KT)^{2/3}\sqrt[3]{\frac{\log(N|\Pi|)}{N}} + T\delta_{\textit{class}}\right).$$

5. Experiments

This section shows that LOLS is able to improve upon a suboptimal reference policy and provides empirical evidence to support the analysis in Section 3. We conducted experiments on the following three applications.

Cost-Sensitive Multiclass classification. For each cost-sensitive multiclass sample, each choice of label has an associated cost. The search space for this task is a binary search tree. The root of the tree corresponds to the whole set of labels. We recursively split the set of labels in half, until each subset contains only one label. A trajectory through the search space is a path from root-to-leaf in this tree. The loss of the end state is defined by the cost. An optimal reference policy can lead the agent to the end state with the minimal cost. We also show results of using a bad reference policy which arbitrarily chooses an action at each state. The experiments are conducted on KDDCup 99 dataset[5] generated from a computer network intrusion detection task. The dataset contains 5 classes, $4,898,431$ training and $311,029$ test instances.

Part of speech tagging. The search space for POS tagging is left-to-right prediction. Under Hamming loss the trivial optimal reference policy simply chooses the correct part of speech for each word. We train on $38k$ sentences and test on $11k$ from the Penn Treebank (Marcus et al., 1993). One can construct suboptimal or even bad reference policies, but under Hamming loss these are all equivalent to the optimal policy because roll-outs by any fixed policy will incur exactly the same loss and the learner can immediately learn from one-step deviations.

[5] http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

roll-out → / ↓ roll-in	Reference	Mixture	Learned
Reference is optimal			
Reference	0.282	0.282	0.279
Learned	0.267	**0.266**	**0.266**
Reference is bad			
Reference	1.670	1.664	0.316
Learned	**0.266**	**0.266**	**0.266**

Table 2. The average cost on cost-sensitive classification dataset; columns are roll-out and rows are roll-in. The best result is bold. **SEARN achieves 0.281 and 0.282 when the reference policy is optimal and bad, respectively.** LOLS is Learned/Mixture and highlighted in green.

roll-out → / ↓ roll-in	Reference	Mixture	Learned
Reference is optimal			
Reference	95.58	94.12	94.10
Learned	**95.61**	94.13	94.10

Table 3. The accuracy on POS tagging; columns are roll-out and rows are roll-in. The best result is bold. **SEARN achieves 94.88.** LOLS is Learned/Mixture and highlighted in green.

Dependency parsing. A dependency parser learns to generate a tree structure describing the syntactic dependencies between words in a sentence (McDonald et al., 2005; Nivre, 2003). We implemented a hybrid transition system (Kuhlmann et al., 2011) which parses a sentence from left to right with three actions: SHIFT, REDUCELEFT and REDUCERIGHT. We used the "non-deterministic oracle" (Goldberg & Nivre, 2013) as the optimal reference policy, which leads the agent to the best end state reachable from each state. We also designed two suboptimal reference policies. A bad reference policy chooses an arbitrary legal action at each state. A suboptimal policy applies a greedy selection and chooses the action which leads to a good tree when it is obvious; otherwise, it arbitrarily chooses a legal action. (This suboptimal reference was the *default* reference policy used prior to the work on "non-deterministic oracles.") We used data from the Penn Treebank Wall Street Journal corpus: the standard data split for training (sections 02-21) and test (section 23). The loss is evaluated in UAS (unlabeled attachment score), which measures the fraction of words that pick the correct parent.

For each task and each reference policy, we compare 6 different combinations of roll-in (learned or reference) and roll-out (learned, mixture or reference) strategies. We also include SEARN in the comparison, since it has notable differences from LOLS. SEARN rolls in and out with a mixture where a different policy is drawn for each state, while LOLS draws a policy once per example. SEARN uses a batch learner, while LOLS uses online. The policy in SEARN is a mixture over the policies produced at each iteration. For LOLS, it suffices to keep just the most recent one. It is an open research question whether an analogous theoretical guarantee of Theorem 3 can be established for SEARN.

roll-out → / ↓ roll-in	Reference	Mixture	Learned
Reference is optimal			
Reference	87.2	89.7	88.2
Learned	**90.7**	90.5	86.9
Reference is suboptimal			
Reference	83.3	87.2	81.6
Learned	87.1	**90.2**	86.8
Reference is bad			
Reference	68.7	65.4	66.7
Learned	75.8	**89.4**	87.5

Table 4. The UAS score on dependency parsing data set; columns are roll-out and rows are roll-in. The best result is bold. **SEARN achieves 84.0, 81.1, and 63.4 when the reference policy is optimal, suboptimal, and bad, respectively.** LOLS is Learned/Mixture and highlighted in green.

Our implementation is based on Vowpal Wabbit[6], a machine learning system that supports online learning and L2S. For LOLS's mixture policy, we set $\beta = 0.5$. We found that LOLS is not sensitive to β, and setting β to be 0.5 works well in practice. For SEARN, we set the mixture parameter to be $1-(1-\alpha)^t$, where t is the number of rounds and $\alpha = 10^{-5}$. Unless stated otherwise all the learners take 5 passes over the data.

Tables 2, 3 and 4 show the results on cost-sensitive multiclass classification, POS tagging and dependency parsing, respectively. The empirical results qualitatively agree with the theory. Rolling in with reference is always bad. When the reference policy is **optimal**, then doing roll-outs with reference is a good idea. However, when the reference policy is **suboptimal** or **bad**, then rolling out with reference is a bad idea, and mixture rollouts perform substantially better. LOLS also significantly outperforms SEARN on all tasks.

Acknowledgements

Part of this work was carried out while Kai-Wei, Akshay and Hal were visiting Microsoft Research.

[6] http://hunch.net/~vw/

References

Abbott, H.L and Katchalski, M. On the snake in the box problem. *Journal of Combinatorial Theory, Series B*, 45 (1):13 – 24, 1988.

Cesa-Bianchi, N. and Lugosi, G. *Prediction, Learning, and Games*. Cambridge University Press, 2006.

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

Daumé III, Hal and Marcu, Daniel. Learning as search optimization: Approximate large margin methods for structured prediction. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2005.

Daumé III, Hal, Langford, John, and Marcu, Daniel. Search-based structured prediction. *Machine Learning Journal*, 2009.

Daumé III, Hal, Langford, John, and Ross, Stéphane. Efficient programmable learning to search. arXiv:1406.1837, 2014.

Doppa, Janardhan Rao, Fern, Alan, and Tadepalli, Prasad. HC-Search: A learning framework for search-based structured prediction. *Journal of Artificial Intelligence Research (JAIR)*, 50, 2014.

Goldberg, Yoav and Nivre, Joakim. Training deterministic parsers with non-deterministic oracles. *Transactions of the ACL*, 1, 2013.

Goldberg, Yoav, Sartorio, Francesco, and Satta, Giorgio. A tabular method for dynamic oracles in transition-based parsing. *Transactions of the ACL*, 2, 2014.

He, He, Daumé III, Hal, and Eisner, Jason. Imitation learning by coaching. In *Neural Information Processing Systems (NIPS)*, 2012.

Kuhlmann, Marco, Gómez-Rodríguez, Carlos, and Satta, Giorgio. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pp. 673–682. Association for Computational Linguistics, 2011.

Langford, John and Beygelzimer, Alina. Sensitive error correcting output codes. In *Learning Theory*, pp. 158–172. Springer, 2005.

Marcus, Mitch, Marcinkiewicz, Mary Ann, and Santorini, Beatrice. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2): 313–330, 1993.

McDonald, Ryan, Pereira, Fernando, Ribarov, Kiril, and Hajic, Jan. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the Joint Conference on Human Language Technology Conference and Empirical Methods in Natural Language Processing (HLT/EMNLP)*, 2005.

Nivre, Joakim. An efficient algorithm for projective dependency parsing. In *International Workshop on Parsing Technologies (IWPT)*, pp. 149–160, 2003.

Ross, Stéphane and Bagnell, J. Andrew. Efficient reductions for imitation learning. In *Proceedings of the Workshop on Artificial Intelligence and Statistics (AI-Stats)*, 2010.

Ross, Stéphane and Bagnell, J. Andrew. Reinforcement and imitation learning via interactive no-regret learning. arXiv:1406.5979, 2014.

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

Zinkevich, Martin. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2003.