SEARN in Practice

Hal Daumé III, John Langford and Daniel Marcu
me@hal3.name,jl@hunch.net,marcu@isi.edu

1 Introduction

We recently introduced an algorithm, SEARN, for solving hard structured prediction problems [DLM06]. This algorithm enjoys many nice properties: efficiency, wide applicability, theoretical justification and simplicity. However, under a desire to fit a lot of information into the original paper [DLM06], it may not be so clear how simple the technique is. This report is designed to showcase how SEARN can be applied to a wide variety of techniques and what *really* goes on behind the scenes. We will make use of three example problems, ranging from simple to complex. These are: (1) sequence labeling, (2) parsing and (3) machine translation. (These were chosen to be as widely understandable, especially in the NLP community, as possible.) In the end, we will come back to discuss SEARN for general problems.

2 Searn

The motivation behind SEARN is that in most NLP problem, one first specifies a model and features, then learns parameters on those features (usually through maximum likelihood/relative frequencies) and then attempts to apply this learned model to new data (think MT). One of the key difficulties is that when applying the model to new data, the search is rarely tractable (or computationally feasible). Even in parsing, which is "polynomial time," (typically $\mathcal{O}(n^3)$ in the length of the sentence) a plethora of pruning and beaming methods are employed in practice. Worse, in synchronous grammars, the parsing time is $\mathcal{O}(n^1 1)$, which, while polynomial, is truly impractical. In problems like MT, we are no longer in polynomial time world and are immediately in the realm of NP-hard problems.

SEARN is a method of allowing the learning to take into account the fact that we will be running a search algorithm (among other things). Actually, the result at the end of the day with SEARN is that *search* is completely removed from the problem (at test time).

The basic idea behind SEARN is simple: we view our problem (tagging, parsing, MT, etc.) as a search problem. We construct the search space as we see fit (this is an important practical issue discussed later). We then attempt to learn a classifier that will walk us through the search space in a good way. For instance, in the simple case of POS tagging, we may structure our search space so that we tag in a left-to-right manner. Then, our classifier will tell us: for a given input string and for whatever previous tag decisions we've made, what's the best tag to assign to the next word in the input. As is clear from the tagging example, we *could* train this classifier based solely on the training data.

Both in theory and in practice, this is a bad idea and leads to the label-bias problem [LMP01] and other issues. In fact, it is possible to show that, in the sequence labeling problem, even if our classifier obtains a small error rate ϵ, we could potentially do as badly as $0.5\epsilon T^2$ on the tagging problem, where T is the length of the sentence [Kää06]. The problem is essentially that errors compound. If we make a mistake on the very first label, this could potentially throw us into a part of the search space we have never been trained on and the classifier could do (essentially) arbitrarily badly.

The solution proposed by SEARN is the following. Instead of training based on the true path through the search space, train on the path that our classifier actually takes in practice. This means that the classifier will be trained based on the data that it will actually expect to see. This solves the label-bias problem but introduces a chicken-and-egg problem: how do we train such a beast? The solution—as in most chicken-and-egg problems—is to iterate.

SEARN makes use of the notion of a *policy*. A policy π is just a function $\pi(x, \hat{y}_1, \dots, \hat{y}_{t-1})$ that takes as input the true input x (eg., the sentence to tag in POS tagging) and a partial output $\hat{y}_1, \dots, \hat{y}_{t-1}$. It outputs $\hat{y}_t$. This is, as we can see, essentially just a multiclass classifier. When it is a classifier, we will write it as $h(x, \hat{y}_{1:t-1})$.

SEARN makes one requirement: an optimal policy for the training data. That is, we need a function π^* that gives us the *best* choice to make *no matter what path we've followed.* In particular, $\pi^*(x, y_{1:T}, \hat{y}_{1:t-1})$ gives the best choice for $\hat{y}_t$ given the input x and the *true output* $y_{1:T}$. In sequence labeling, this will typically be simply y_t (the best thing to do next is just to produce the correct tag), but this will not always be the case. The issue of optimal policies and how hard they are to construct is discussed later.

SEARN operates by maintaining a current policy and attempts to use to this generate new training data on which to learn a new policy (new classifier). The current policy is initialized to the optimal policy. When a new classifier is learned, we *interpolate* it with the old classifier. This gives us a convergence guarantee; otherwise the algorithm could diverge. There is a trade-off here. Our goal is to move away from optimal policy completely. To do so, we want to have a large interpolation constant. However, if we move too quickly, we can diverge. The SEARN paper [DLM06] provides an analytical value for the interpolation parameter that is guaranteed to yield convergence. However, this is too small in practice, so we use line search on development data.

The full SEARN algorithm is below:

1. Initialize the current policy to the optimal policy
2. Repeat:
   (a) Use the current policy to generate paths over all training examples
   (b) For each example, for each step in the path traversed by the current policy:
      i. Generate a multiclass example whose classes are possible decisions and whose losses are based on the loss of the current policy (see below)
   (c) Learn a new multiclass classifier on the basis of the examples
   (d) Find an interpolation constant β on development data that improves performance
   (e) Set the current policy to β times the new policy plus $1 - \beta$ times the old policy
3. Return the current policy without the optimal policy

This algorithm does the following. On the first loop through the training data, step (2a) generates paths based on the optimal policy (since the current policy is initialized to be optimal). In sequence labeling, this simply results in the observed training examples. In step (2b), it uses each position in search (each labeled word) to generate a multiclass classification example (2bi). The classes of these examples are the labels themselves. Each class also has an associated loss (discussed below); for now, think of this loss as being 0 for the correct label and 1 for all incorrect labels: a standard multiclass problem. After we have generated all multiclass examples in (2b), we run any classification algorithm over them (eg., a maximum entropy classifier). This gives us a new classifier (2c).

In step (2d), we attempt to find a value for β (between 0 and 1) so that, after executing (2e), the current policy improves. In the very first iteration, it is best to use a conservative value of β: using $\beta = 1/T^3$ is guaranteed to work, though often $1/T$ or even just 0.1 works just as well. However, on all non-first iterations of the inner loop, we use a line search method to find a good value of β. In particular, for a given β, we evaluate how well the current policy (without the optimal policy) behaves on a held-out development corpus. We find a value for β that maximizes this. In practice, β is usually quite high.

Once a value of β has been selected (either through the conservative choice, through line search, or through a deterministic choice), we perform the interpolation. Of course, it makes no sense to interpolate a deterministic function (the optimal policy) with a classifier (the new policy). What we mean by interpolation is that we keep around all options and then use a Monte-Carlo estimate of the choice. That is, in the second iteration, we will have a policy that looks like β times a classifier plus $1 - \beta$ times the optimal policy. When we ask this policy to make a decision, it flips a coin with bias β. If the coin turns up heads, it uses the classifier; if the coin turns up tails, it uses the optimal policy.

Before concluding, we need to return to the issue of selecting the losses associated with the classes when generating the multiclass examples. Formally, we want the loss associated with the choice y_t to be the *expected* regret of the current policy given that we chose y_t. (That is, it is the difference in loss based on choosing y_t as selected and choosing y_t optimally.) One simple way to compute this is to "pretend" we had made the decision y_t and to run the current policy as if we had made that choice. At the end of this run, we will have a complete output. We can compute the loss of this complete output with respect to the true output. This loss gives us the loss for the class y_t.

In practice, consistently running the current policy through the end of the example may be prohibitively expensive. An alternative choice is the *optimal approximation.* That is, instead of running the *current policy* through the end, we run the *optimal policy* through the end and then compute the loss. In many cases, we needn't actually run the policy: we will be able to compute the optimal loss in closed form. This makes the algorithm significantly faster, but potentially introduces a bias into the underlying classifier.

3 Example: Sequence Labeling

The first problem we consider is the sequence labeling problem. We will focus initially on a part-of-speech tagging problem and then discuss an NP chunking problem.

3.1 Tagging

In tagging, one observes an input sequence x of length T and produces a label sequence of length T where each element in the label sequence corresponds to an element in the input. The standard loss function for this problem is Hamming loss, which measures the number of places that the hypothesized output $\hat{y}$ differs from the true output y:

$$l^{\text{Hamming}}(y, \hat{y}) = \sum_{t=1}^{T} \mathbf{1}(y_t \neq \hat{y}_t) \tag{1}$$

There are many ways to structure the search space for this problem. The most obvious is a left-to-right greedy search. That is, we begin with an empty output and at step t we add the next label $\hat{y}_t$. Given this, the classifier we learn will have access to the input sentence x and all previous decisions $\hat{y}_1, \dots, \hat{y}_{t-1}$ when making a decision about y_t (note that SEARN does not have to make Markov assumptions). This search structure has two advantages: (1) it is easy to compute the optimal policy under this decomposition; (2) it is linguistically plausible that there is some dependence of y_t on the previous tags.

Given this choice of search space, we need to derive an optimal policy. For this case, it is trivial: $\pi^*(x, y_{1:T}, \hat{y}_{1:t-1}) = y_t$; we simply produce the next best tag. It is straightforward to observe that following this policy will always minimize our Hamming loss.

Given this decomposition, building SEARN is quite simple. In the first iteration, we simply train a classifier to produce each label y_t for each example on the basis of the *true* choices for $y_{1:t-1}$. We then choose some interpolation constant and repeat the process using the interpolated policy. Here, instead of using the true labels for $y_{1:t-1}$, we use a mixture of the true labels and the labels predicted by the classifier we learned in the first iteration (where β is the mixture parameter).

It is also worth observing that using the optimal approximation for computing the losses is trivial in this example. Since the optimal policy always chooses the training label, and since the Hamming loss decomposes completely over the individual choices, the optimal approximation for the regret for $\hat{y}_t$ will be zero if $\hat{y}_t = y_t$ and will be one otherwise.

3.2 Chunking

The chunking problem differs from the tagging problem in that it is a joint segmentation and labeling problem. Instead of mapping an input of length T to a T-many labels, we first segment the input into *chunks* and then label each chunk. A canonical example is named entity identification: we want to find all text spans where names appear (which are often longer than a single word) and label them with "Person," "Organization," "Location," etc.

The standard loss function used for chunking is the F_1 measure. Given a true chunking y, we first compute $n(y)$: the number of names in y. We then take the hypothesized chunking $\hat{y}$ and compute $n(\hat{y})$: the number of names in $\hat{y}$. Finally, we "intersect" y with $\hat{y}$ and compute $n(y \cap \hat{y})$: the number of names in the intersection. The intersection is defined to be the set of names in both y and $\hat{y}$ that have identical spans (cover exactly the same words) and identical labels. Essentially, its the number of correctly identified entities.

We first compute the precision to be $n(y \cap \hat{y})/n(\hat{y})$ (as the ratio of hypothesized names that were correct) and the recall to be $n(y \cap \hat{y})/n(y)$ (as the ration of true names that were found). The F_1 measure is the geometric mean of precision and recall: $F_1 = 2(p^{-1} + r^{-1})^{-1}$. This favors hypotheses with roughly equal precisions and recalls. The F_1 measure is favored over standard accuracy because if there are very few names in the text, simply *never* guessing a name will yield high accuracy, but very low F_1 measure.

There are at least two reasonably ways to structure search for the chunking problem: word-at-a-time or chunk-at-a-time. In word-at-a-time, search proceeds by one word per step. A decision corresponds to one of the following options: begin a new name, continue the current name (assuming that we are currently inside a name) or mark as not a name. In chunk-at-a-time, we take steps by producing an entire chunk at once. That is, in a single step, we choose either that the next word is not a name, or we choose that the next word is a name and also choose how many words long this name is.

Word-at-a-time and chunk-at-a-time behave very similarly with respect to the loss function and optimal policy. We will discuss word-at-a-time for notational convenience. The basic question is how to compute the optimal policy. We analyze three cases:

$$\pi^*(x, y_{1:T}, \hat{y}_{1:t-1}) = \begin{cases} \text{begin } X & y_t = \text{begin } X \\ \text{in } X & y_t = \text{in } X \text{ and } \hat{y}_{t-1} \in \{\text{begin } X, \text{in } X\} \\ \text{out} & \text{otherwise} \end{cases} \tag{2}$$

It is fairly straightforward to show that this policy is optimal. There is, actually, another optimal policy. For instance, if y_t is "in X" but $\hat{y}_{t-1}$ is "in Y" (for $X \neq Y$), then it is equally optimal to select $\hat{y}_t$ to be "out" or "in Y". In theory, when the optimal policy doesn't care about a particular decision, it is typically useful to randomize over the selection. That is, flip an unbiased coin and randomly choose between these options when asked to make an optimal step in this circumstance.

Like the Hamming case, we can explicitly compute the the optimal approximation loss. This is a bit more tricky, but still straightforward. We need three values: the size of the intersection n_i, the size of the truth n_t and the size of the hypothesis n_h. The size of the truth is constant, so we only care about the intersection and hypothesis. These are:

$$n_i = \begin{cases} n(\hat{y}_{1:t-1} \cap y) + 1 + n(y_{t:T}) & y_t = \text{in } X \text{ and we got the begin right} \\ n(\hat{y}_{1:t-1} \cap y) + n(y_{t:T}) & \text{otherwise} \end{cases} \tag{3}$$

$$n_h = \begin{cases} n(\hat{y}_{1:t-1}) + n(y_{t:T}) & y_t \in \{\text{begin } X, \text{out}\} \\ n(\hat{y}_{1:t-1}) + n(y_{t:T}) + 1 & \text{otherwise} \end{cases} \tag{4}$$

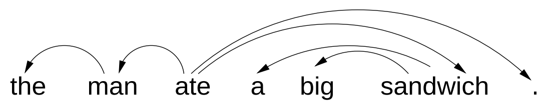
In the first equation, by "we got the begin right" it means that the most recent begin in y (before t) matches the most recent begin in $\hat{y}$ (before t). In otherwords, with respect to the current phrase, we are correct. Given these values, the optimal precision is n_i/n_h and the optimal recall is n_i/n_t yielding an easy computation for the optimal F_1.

4 Example: Parsing

In this section, we consider the problem of dependency parsing in a shift-reduce framework [SL05]. This is primarily for convenience. The extension to the constituency case is a bit more involved, but still possible. The extension to a non-shift reduce framework (i.e., to something like CKY parsing or hypergraph parsing) is at the moment a bit more unclear. I believe it possible, but it might take some more work.

The correct (unlabeled) dependency parse for the sentence "the man ate a big sandwich ." is shown in Figure 1. The standard assumption for dependency parsing is that of projectivity: essentially, none of the arcs cross. This assumption is true in most languages, but untrue, for instance, in Czech. In the shift-reduce framework, a dependency tree is built through a sequence of steps. The parser maintains an active *stack* onto which words are pushed using the *shift* action. The top two elements on the stack (the most recent two) can be combined using a reduce action. There are two reduce actions: one for each possible direction the arrow could point. The complete derivation of the tree is shown in the right of Figure 1.

It is clear from this analysis that the decision of shift/reduce-left/reduce-right could be accomplished using SEARN. The standard loss function for this problem is Hamming loss over dependencies (sometimes directed, sometimes undirected). We will consider the undirected case for simplicity.



shift(the)
shift(man)
reduce(the ← man)
shift(ate)
reduce([the man] ← ate)
shift(a)
shift(big)
shift(sandwich)
reduce(big ← sandwich)
reduce(a ← [big sandwich])
reduce([. . . ate] → [a. . .])
shift(.)
reduce([the. . . sandwich] → .)

Figure 1: (Left) The dependency tree for the sentence "the man ate a big sandwich ." (Right) The sequence of shift-reduce steps that leads to this parse structure.

Again, the key questions are defining the optimal policy and (perhaps) the optimal approximation loss. It is surprisingly easy to define the optimal policy for this problem. Note that a partial hypothesis (state in the search space) for this problem is represented by the stack, which we will denote $s_1, \dots, s_I$.

$$\pi^*(x, y, s_{1:i-1}) = \begin{cases} \text{shift} & i \le 2 \\ \text{reduce} & \text{there are no words left to shift} \\ \text{reduce} & \text{there is an arc between } s_{i-2} \text{ and } s_{i-1} \\ \text{shift} & \text{otherwise} \end{cases} \tag{5}$$

The reason this is optimal is as follows. If there should be a reduction between the two most recent words, we have to do it now because we will never have a chance again. Otherwise, any mistakes we have made so far are hopeless: we cannot recover. We might as well just shift until we have nothing left to shift and then start reducing. There are a few degrees of freedom: we should alternatively reduce until we cannot reduce any more and then start shifting. In practice, one might want to randomize these choices.

The computation of the optimal approximation loss is even easier. Any incorrectly specified arcs encountered thus far cannot be fixed, so we must accumulate error for them. Any arcs not encountered thus far can always be satisfied. So the optimal approximation Hamming loss is simply the Hamming loss up until the current step.

5 Example: Machine Translation

This is, to some people, the Holy Grail of NLP. I'm going to discuss an incredibly simple model for MT, but the extension to more complex models is really what is interesting. Specifically, we're going to sit ourselves in the world of left-to-right translation. This covers most models of word-based and phrase-based translation, though not necessarily all recent models of syntactic translation. The optimal policy question becomes the following: given a set of reference translations R, an English translation *prefix* $e_1, \dots, e_{i-1}$, what word (or phrase) should be produced next (or are we done?). This will be driven by some loss function l (such as one minus Bleu or one minus NIST or ...). It may be possible to analyze one of these losses in particular to come up with a closed form optimal policy (though I tend to doubt this would be the case for Bleu). So, to maintain generality *and* to demonstrate the SEARN is applicable even when the optimal policy is *not* available in closed form, we will take an alternative tact: search.

This is a very natural search problem. We have a search space over prefixes of translations. Actions include adding a word (or phrase) to the end of an existing translation. Our reward function is BLEU or NIST or... We want to find the best *full* output starting at some given prefix. Once we have the best full output, we simply inspect the first *decision* of that output. In order to make this search process more tractable, it is useful to restrict the search space. In fact, it is easy to verify that, for the purpose of computing the optimal policy, we only ever need to consider adding words that actually occur in a reference summary (and are not already covered). Moreover, for both BLEU and NIST, we can easily compute an admissable heuristic based on uncovered unigram counts: we can simply assume that we will get all of them correct but no corresponding bigrams or trigrams. It is likely possible to come up with better heuristics, but this one seems (intuitively) sufficient.

6 Conclusions

The efficacy of SEARN hinges on the ability to compute an optimal (or near-optimal) policy. For many problems including sequence labeling and segmentation (Section 3) and parsing (Section 4), the optimal policy is available in closed form. For other problems, such as the summarization problem described in the SEARN paper and machine translation (Section 5), the optimal policy may not be available. In such cases, the suggested approximation is to perform explicit search. One question is: can such search always be accomplished effectively. Though it is not a theorem, I believe this is always possible. The reason is that when we train SEARN based on a search-computed policy, we are essentially training SEARN to mimic whatever search algorithm we have implemented. The requirement that this search algorithm perform well seems reasonable. This is because we construct this search *knowing* the true output. If, knowing the true output, we cannot find a good solution, then it is hard to imagine that we will ever be able to find a good solution *not* knowing the true output, as is the case at test time: i.e., learning must fail.

References

[DLM06] Hal Daumé III, John Langford, and Daniel Marcu. Search-based structured prediction. Unpublished; available at `http://pub.hal3.name#daume06searn`, 2006.

[Kää06] Matti Kääriäinen. Lower bounds for reductions. Talk at the Atomic Learning Workshop (TTI-C), March 2006.

[LMP01] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2001.

[SL05] Kenji Sagae and Alon Lavie. A classifier-based parser with linear run-time complexity. In *Proceedings of the Ninth International Workshop on Parsing Technologies (IWPT 2005)*, Vancouver, Canada, 2005.