Journal of Machine Learning Research 14 (2014) 1011-1044 Submitted 6/13; Published 3/14

Structured Prediction via Output Space Search

Janardhan Rao Doppa DOPPA@EECS.OREGONSTATE.EDU
Alan Fern AFERN@EECS.OREGONSTATEEDU
Prasad Tadepalli TADEPALL@EECS.OREGONSTATEEDU
School of EECS, Oregon State University
Corvallis, OR 97331-5501, USA

Editor: S.V.N. Vishwanathan

Abstract

We consider a framework for structured prediction based on search in the space of complete structured outputs. Given a structured input, an output is produced by running a time-bounded search procedure guided by a learned cost function, and then returning the least cost output uncovered during the search. This framework can be instantiated for a wide range of search spaces and search procedures, and easily incorporates arbitrary structured-prediction loss functions. In this paper, we make two main technical contributions. First, we describe a novel approach to automatically defining an effective search space over structured outputs, which is able to leverage the availability of powerful classification learning algorithms. In particular, we define the limited-discrepancy search space and relate the quality of that space to the quality of learned classifiers. We also define a sparse version of the search space to improve the efficiency of our overall approach. Second, we give a generic cost function learning approach that is applicable to a wide range of search procedures. The key idea is to learn a cost function that attempts to mimic the behavior of conducting searches guided by the true loss function. Our experiments on six benchmark domains show that a small amount of search in limited discrepancy search space is often sufficient for significantly improving on state-of-the-art structured-prediction performance. We also demonstrate significant speed improvements for our approach using sparse search spaces with little or no loss in accuracy.

Keywords: structured prediction, state space search, imitation learning, cost function

1. Introduction

Research in machine learning has progressed from studying isolated classification tasks to the study of tasks such as information extraction and scene understanding where the inputs and outputs are structured objects. The general field of structured prediction deals with these kind of structured tasks where the goal is to learn a predictor that must produce a complex structured output given a complex structured input. A prototypical example is Part-of-Speech (POS) tagging, where the structured input is a sequence of words and structured output corresponds to the POS tags for those words. Another example is that of image scene labeling, where the structured input is an image and the structured output is a semantic labeling of the image regions.

A typical approach to structured prediction is to learn a cost function $\mathcal{C}(\mathbf{x}, \mathbf{y})$ for scoring a potential structured output $\mathbf{y}$ given a structured input $\mathbf{x}$. Given such a cost function and a new input $\mathbf{x}$, the output computation then involves solving the so-called Argmin problem:

$$\hat{\mathbf{y}} = \arg\min_{\mathbf{y}} \mathcal{C}(\mathbf{x}, \mathbf{y}).$$

For example, approaches such as Conditional Random Fields (CRFs) (Lafferty et al., 2001), Max-Margin Markov Networks (Taskar et al., 2003) and Structured SVMs (Tsochantaridis et al., 2004) represent the cost function as a linear model over template features of both $\mathbf{x}$ and $\mathbf{y}$. Unfortunately exactly solving the Argmin problem is often intractable and efficient solutions exist only in limited cases such as when the dependency structure among features forms a tree. In such cases, one might simplify the features to allow for tractable inference, which can be detrimental to prediction accuracy. Alternatively, a heuristic optimization method can be used such as loopy belief propagation or variational inference. While such methods have shown some success in practice, it can be difficult to characterize their solutions and to predict when they are likely to work well for a new problem.

In this paper, we study a new search-based approach to structured prediction based on search in the space of complete outputs. The approach involves first defining a combinatorial search space over complete structured outputs that allows for traversal of the output space. Next, given a structured input, a state-based search strategy (e.g., best-first or greedy search), guided by a learned cost function, is used to explore the space of outputs for a specified time bound. The least cost output uncovered by the search is then returned as the prediction. This approach is motivated by our observation that for a variety of structured prediction problems, if we use the true loss function of the structured prediction problem to guide the search (even non-decomposable losses), then high-quality outputs are found very quickly. This suggests that similar performance might be achieved if we could learn an appropriate cost function to guide search in place of the true loss function.

A potential advantage of our search-based approach, compared to most structured-prediction approaches (see Section 6), is that it scales gracefully with the complexity of the cost function dependency structure. In particular, the search procedure only needs to be able to efficiently evaluate the cost function at specific input-output pairs, which is generally straightforward even when the corresponding Argmin problem is intractable. Thus, we are free to increase the complexity of the cost function without considering its impact on the inference complexity. Another potential benefit of our approach is that since the search is over complete outputs, our inference is inherently an anytime procedure, meaning that it can be stopped at any time and return the best output discovered so far. This has the flexibility of allowing for the use of more or less inference time for computing outputs as dictated by the application, with the idea that more inference time may sometimes allow for higher quality outputs.

The effectiveness of our approach for a particular problem depends critically on: 1) The identification of an effective combination of search space and search strategy over structured outputs, and 2) Our ability to learn a cost function for effectively guiding the search for high quality outputs. The main contribution of our work is to provide generic solutions to these two issues and to demonstrate their empirical effectiveness.

First, we describe the limited-discrepancy search space, as a generic search space over complete outputs that can be customized to a particular problem by leveraging the power of (non-structured) classification learning algorithms. We show that the quality of this search space is directly related to the error of the learned classifiers and can be quite favorable compared to more naive search space definitions. We also define a sparse version of the search space to improve the efficiency of our approach. The sparse search spaces tries to reduce the branching factor while not hurting the quality of the search space too much.

Our second contribution is to describe a generic cost function learning algorithm that can be instantiated for a wide class of "ranking-based search strategies." The key idea is to learn a cost function that allows for imitating the search behavior of the algorithm when guided by the true loss function. We give a PAC bound for the approach in the realizable setting, showing a polynomial sample complexity for doing approximately as well as when guiding search with the true loss function.

Finally, we provide experimental results for our approach on a number of benchmark problems and show that even when using a relatively small amount of search, the performance is comparable or better than the state-of-the-art in structured prediction. We also demonstrate significant speed improvements of our approach when used with sparse search spaces.

The remainder of the paper is organized as follows. In Section 2, we introduce our problem setup and give a high-level overview of our framework. In Section 3, we define two search spaces over complete outputs in terms of a recurrent classifier, relate their quality to the accuracy of the classifier, and then, define sparse search spaces to improve the efficiency of our approach. We describe our cost function learning approach in Section 4. Section 5 presents our experimental results and finally Sections 6 and 7 discuss related work and future directions.

2. Problem Setup

A structured prediction problem specifies a space of structured inputs $\mathcal{X}$, a space of structured outputs $\mathcal{Y}$, and a non-negative *loss function* $L : \mathcal{X} \times \mathcal{Y} \times \mathcal{Y} \mapsto \Re^+$ such that $L(x, y', y)$ is the loss associated with labeling a particular input x by output y' when the true output is y. We are provided with a training set of input-output pairs $\{(x_1, y_1), \ldots, (x_N, y_N)\}$, drawn from an unknown target distribution, where y_i is the true output for input x_i. The goal is to return a function/predictor from structured inputs to outputs whose predicted outputs have low expected loss with respect to the distribution. Since our algorithms will be learning cost functions over input-output pairs we assume the availability of a *feature function* $\Phi : \mathcal{X} \times \mathcal{Y} \mapsto \Re^n$ that computes an n dimensional feature vector for any pair. Intuitively these features should provide some measure of compatibility between (parts of) the structured input and output.

We consider a framework for structured prediction based on state-based search in the space of complete structured outputs. The states of the search space are pairs of inputs and outputs (x, y), representing the possibility of predicting y as the output for x. A search space over those states is specified by two functions: 1) An *initial state function* I such that $I(x)$ returns an initial search state for any input x, and 2) A *successor function* S such that for any search state (x, y), $S((x, y))$ returns a set of successor states $\{(x, y_1), \ldots, (x, y_k)\}$, noting that each successor must involve the same input x as the parent. Section 3 will describe our approach for automatically defining and learning search spaces.

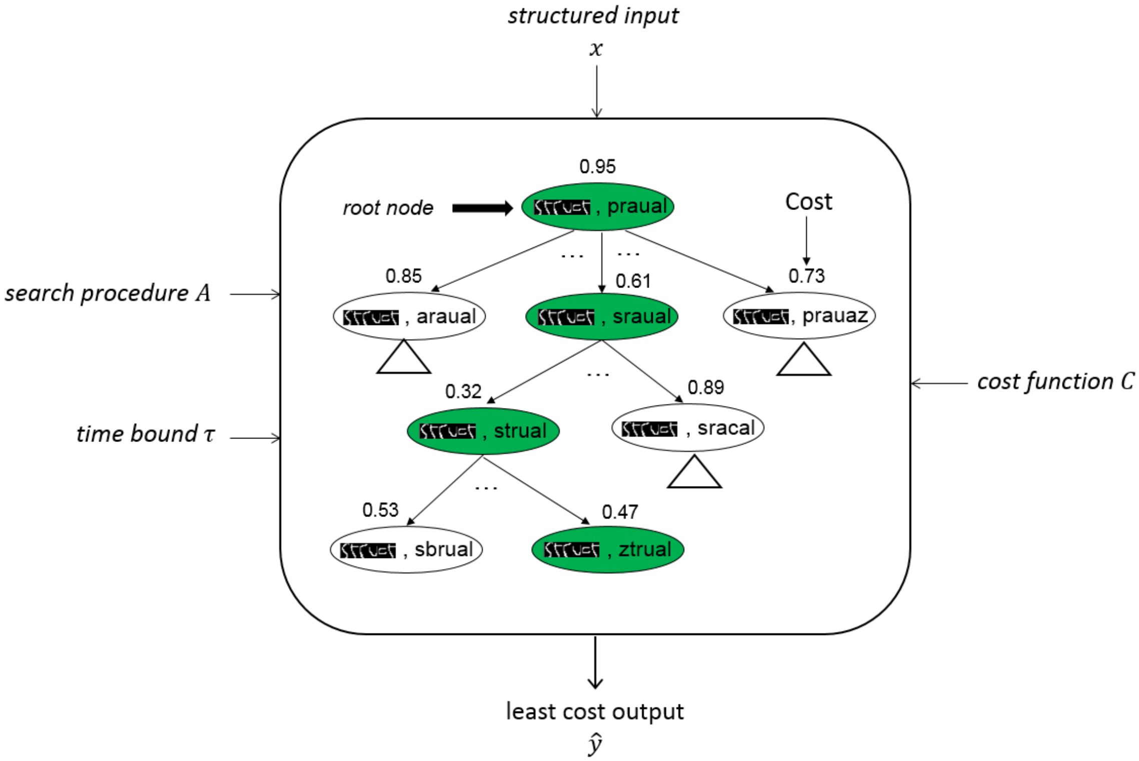


Figure 1: A high level overview of our output space search framework. Given a structured input x, we first instantiate a search space over complete outputs. Each search node in this space consists of a complete input-output pair. Next, we run a search procedure $\mathcal{A}$ (e.g., greedy search) guided by the cost function $\mathcal{C}$ for a time bound τ. The highlighted nodes correspond to the search trajectory traversed by the search procedure, in this case greedy search. We return the least cost output $\hat{y}$ that is uncovered during the search as the prediction for x.

In order to predict outputs, our framework requires two elements in addition to the search space. First, we require a cost function $\mathcal{C}$ that returns a numeric cost for any input-output pair (i.e., search state). Second, we require a search procedure $\mathcal{A}$ (e.g., greedy search or beam search) for traversing search spaces, possibly guided by the cost function. Given these elements, an input x, and a prediction time bound τ we compute an output by executing the search procedure $\mathcal{A}$ starting in the initial state $I(x)$ and guided by the cost function until the time bound is exceeded. We then return the output $\hat{y}$ corresponding to the least cost state that was uncovered during the search as the prediction for x. Figure 1 gives a high-level overview of our search-based framework for structured prediction.

The effectiveness of our search-based framework depends critically on the quality of the search space and the cost function. Ideally we would like the search space to be organized such that high quality outputs are as close as possible to the initial state, which allows the search procedure to uncover those outputs more easily. In addition, it is critical that the cost function is able to correctly score the generated outputs according to their true losses and also to provide effective guidance to the chosen search procedure. A key contribution of this work is to propose supervised learning mechanisms for producing both high-quality search spaces and cost functions, which are described in the next two sections respectively.

3. Search Spaces Over Complete Outputs

In this section we describe two search spaces over structured outputs: 1) The Flipbit space, a simple but sometimes effective baseline, and 2) The limited-discrepancy search (LDS) space, which is intended to improve on the baseline. The key trainable element of each search space is a recurrent classifier, which once trained will fully define each space. Thus, we start this section by describing recurrent classifiers and how they are learned. Next we describe how the learned recurrent classifier is used to define each of the search spaces by defining the initial state and the successor function.

3.1 Recurrent Classifiers

A recurrent classifier h constructs structured outputs based on a series of discrete decisions. This is formalized for a given structured-prediction problem by defining an appropriate *primitive search space* over the possible sequences of decisions. It is important to keep in mind the distinctions between primitive search spaces, which are used by recurrent classifiers, and the search spaces over complete outputs (e.g., flipbit and LDS) upon which our overall framework is built. A primitive search space is a 5-tuple $\langle I, A, s, f, T \rangle$, where I is a function that maps an input x to an initial search node, A is a finite set of actions (or operators), s is the successor function that maps any search node and action to a successor search node, f is a feature function from search nodes to real-valued feature vectors, and T is the terminal state predicate that maps search nodes to $\{1, 0\}$ indicating whether the node is a terminal or not. Each terminal node in the search space corresponds to a *complete structured output*, while non-terminal nodes correspond to *partial structured outputs*. Thus, the decision process for constructing an output corresponds to selecting a sequence of actions leading from the initial node to a terminal. A recurrent classifier is a function that maps nodes of the primitive search space to actions, where typically the mapping is defined in terms of a feature function $f(n)$ that returns a feature vector for any search node. Thus, given a recurrent classifier, we can produce an output for x by starting at the initial node of the primitive space and following its decisions until reaching a terminal state.

As an example, for sequence labeling problems, the initial state for a given input sequence x is a node containing x with no labeled elements. The actions correspond to the selection of individual labels, and the successor function adds the selected label in the next position. Terminal nodes correspond to fully labeled sequences and the feature function computes a feature vector based on the input and previously assigned labels. Figure 2 provides an illustration of the primitive search space for a simple handwriting recognition problem. Each search state is a pair (x, y') where x is the structured input (binary image of the handwritten word) and y' is a partial labeling of the word. The arcs in this space correspond to search steps that label the characters in the input image in a left-to-right order by extending y' in all possible ways by one element. The terminal states or leaves of

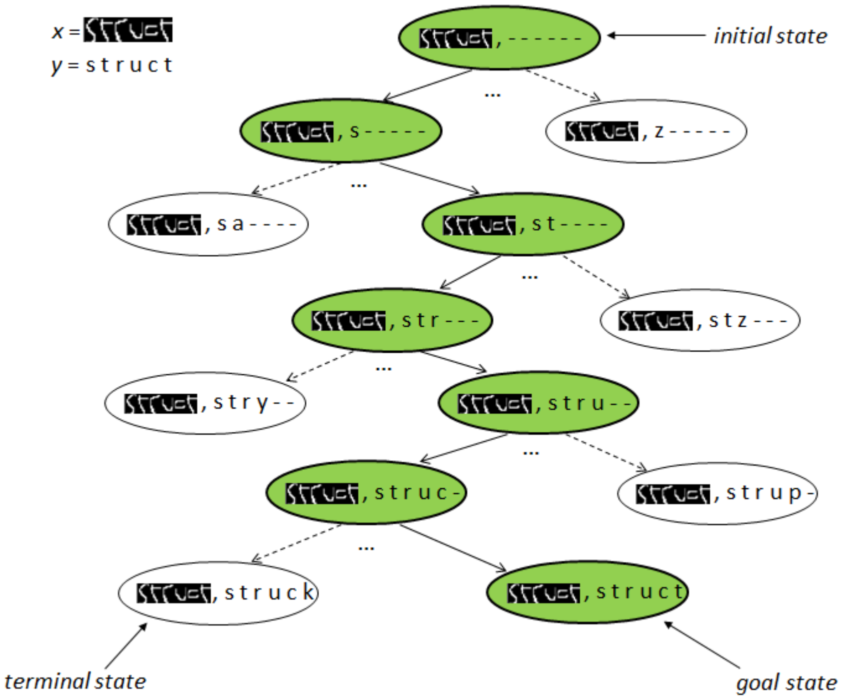


Figure 2: An example primitive search space for the handwriting recognition problem. Arcs represent labeling actions. Solid arcs correspond to the labeling actions taken by the recurrent classifier (optimal classifier in this case).

this space correspond to complete labelings of input x. The terminal state corresponding to the correct output y is labeled as *goal state*. Highlighted nodes correspond to the trajectory of the optimal recurrent classifier (i.e., a classifier that chooses correct action at every state leading to the goal state).

The most basic approach to learning a recurrent classifier is via *exact imitation* of the trajectory followed by the optimal classifier. For this, we assume that for any training input-output pair (x, y) we can efficiently find an action sequence, or *solution path*, for producing y from x. For example, the sequence of highlighted states in Figure 2 correspond to such a solution path. The exact imitation approach learns a classifier by creating a classification training example for each node n on the solution path of a structured example with feature vector $f(n)$ and label equal to the action followed by the path at n. Our experiments will use recurrent classifiers trained via exact imitation (see Algorithm 1), but more sophisticated methods such as Searn (Hal Daumé III et al., 2009) or DAgger (Ross et al., 2011) could also be used.

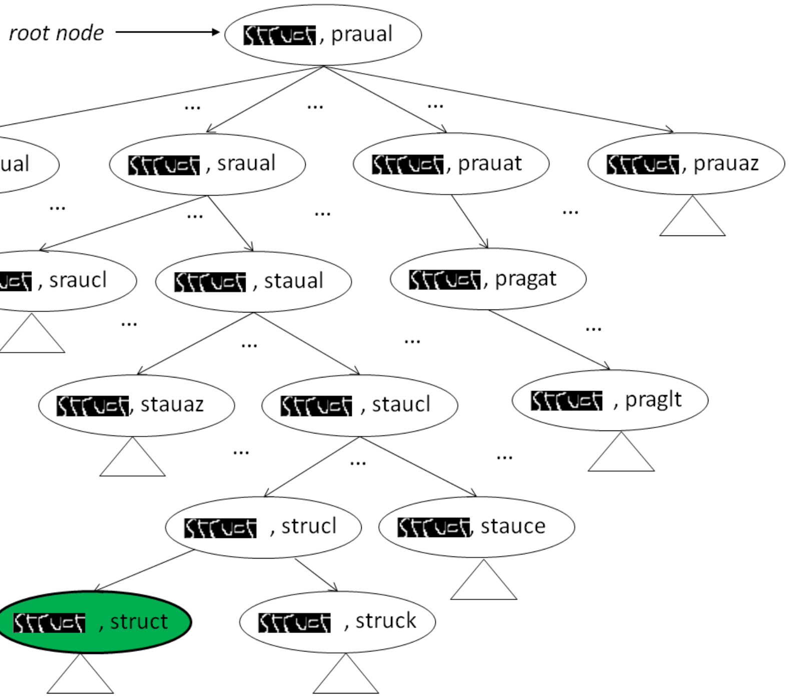


Figure 3: An example Flipbit search space for the handwriting recognition problem

Algorithm 1 Recurrent Classifier Learning via Exact Imitation

Input: $\mathcal{D}$ = Training examples

Output: h, the recurrent classifier

```
Initialize the set of classification examples $\mathcal{L} = \emptyset$
for each training example $(x, y = y_1 y_2 \cdots y_T) \in \mathcal{D}$ do
    for each search step $t = 1$ to $T$ do
        Compute features $f_n$ for search node $n = (x, y_1 \cdots y_{t-1})$
        Add classification example $(f_n, y_t)$ to $\mathcal{L}$
    end for
end for
$h$ =Classifier-Learner($\mathcal{L}$) // learn classifier from all the classification examples
return learned classifier $h$
```

3.2 Flipbit Search Space

The *Flipbit search space* is a simple baseline space over complete outputs that uses a given recurrent classifier h for bootstrapping the search. Each search state is represented by a sequence of actions in the primitive space ending in a terminal node representing a complete output. The initial search state corresponds to the actions selected by the classifier, so that $I(x)$ is equal to $(x, h(x))$, where $h(x)$ is the complete output generated by the recurrent classifier. The search steps generated by the successor function can change the value of one action at any sequence position of the parent state. In a sequence labeling problem, this corresponds to initializing to the recurrent classifier output and then searching over flips of individual labels. The flipbit space is often used by local search techniques (without the classifier initialization) and is similar to the search space underlying Gibbs Sampling.

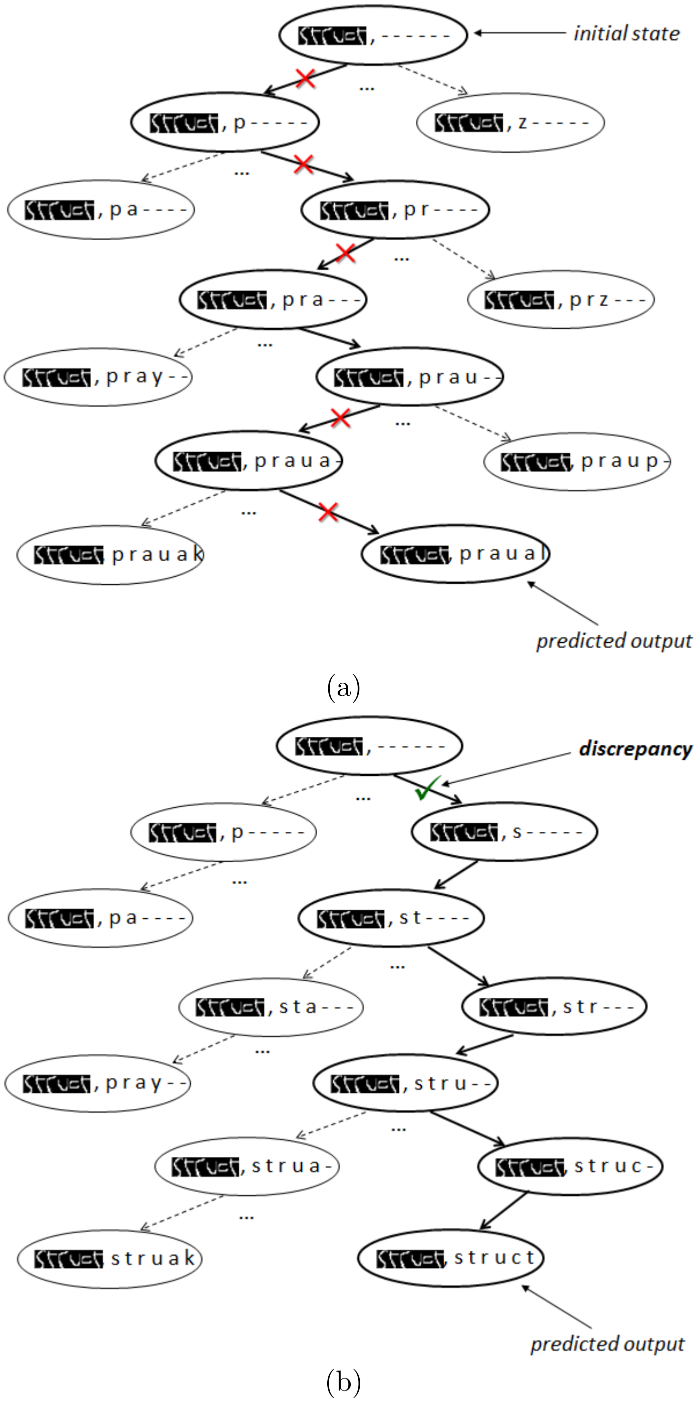
Figure 3 provides an illustration of the flipbit search space via the same handwriting recognition example that was used earlier. Each search state consists of a complete input-output pair and the complete output at every state differs from that of its parent by exactly one label. The highlighted state corresponds to the one with true output y at the smallest depth, which is equal to the number of errors in the output produced by the recurrent classifier.

3.3 Limited-Discrepancy Search Space (LDS)

Notice that the Flipbit space only uses the recurrent classifier when initializing the search. The motivation behind the Limited Discrepancy Search (LDS) space is to more aggressively exploit the recurrent classifier in order to improve the search space quality. LDS was originally introduced in the context of problem solving using heuristic search (Harvey and Ginsberg, 1995). To put LDS in context, we will describe it in terms of using a classifier for structured prediction given a primitive search space. If the learned classifier is accurate, then the number of incorrect action selections will be relatively small. However, even a small number of errors can propagate and cause poor outputs. The key idea behind LDS is to realize that if the classifier response was corrected at the small number of critical errors, then a much better output would be produced. LDS conducts a (shallow) search in the space of possible corrections in the hope of finding a solution better than the original.

More formally, given a recurrent classifier h and its selected action sequence of length T, a discrepancy is a pair (i, a) where $i \in \{1, \ldots, T\}$ is the index of a decision step and $a \in A$ is an action, which generally is different from the choice of the classifier at step i. For any set of discrepancies D we let $h[D]$ be a new classifier that selects actions identically to h, except that it returns action a at decision step i if $(i, a) \in D$. Thus, the discrepancies in D can be viewed as overriding the preferred choice of h at particular decision steps, possibly correcting errors, or introducing new errors. For a structured input x, we will let $h[D](x)$ denote the output returned by $h[D]$ for the search space conditioned on x. At one extreme, when D is empty, $h[D](x)$ simply corresponds to the output produced by the recurrent classifier. At the other extreme, when D specifies an action at each step, $h[D](x)$ is not influenced by h at all and is completely specified by the discrepancy set. In practice, when h is reasonably accurate, we will be primarily interested in small discrepancy sets relative to the size of the decision sequence. In particular, if the error rate of the classifier on individual decisions is small, then the number of corrections needed to produce a correct output will be correspondingly small. The problem is that we do not know where the corrections should be made, and thus LDS conducts a search over the discrepancy sets, usually from small to large sets.

Consider the handwriting recognition example in Figure 4. The actual output produced by the classifier for the input image is `praual`, that is, output produced by introducing zero discrepancies (see Figure 4(a)). If we introduce one discrepancy at the first position



(a)

(b)

Figure 4: Illustration of Limited Discrepancy Search: (a) Trajectory of the recurrent classifier with no discrepancies. Arcs with 'X' mark indicate incorrect actions chosen by the classifier. (b) Trajectory of the recurrent classifier with a correction (discrepancy) at the first error. A single correction allows the classifier to correct all the remaining errors.

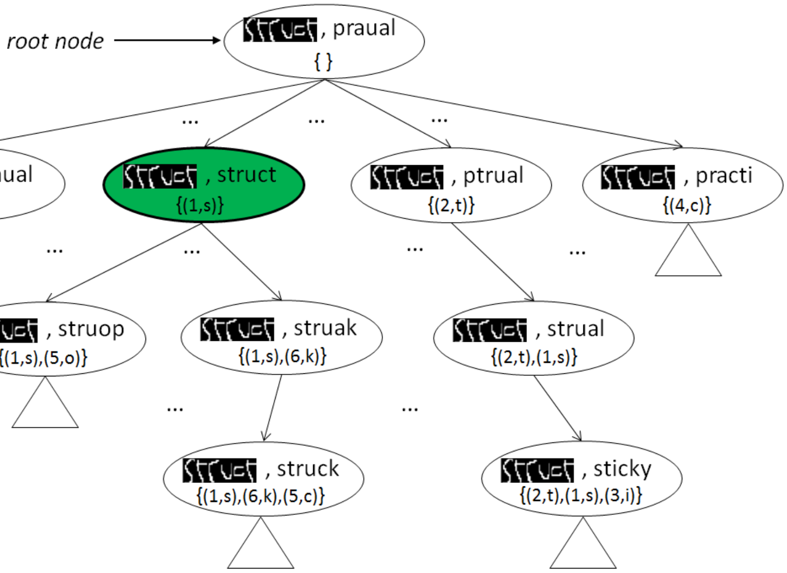


Figure 5: An example Limited Discrepancy Search (LDS) space for the handwriting recognition problem

$(1, s)$ and run the classifier for the remaining labeling, we get `struct` which corrects all the remaining mistakes (see Figure 4(b)). By introducing this single correction, the classifier automatically corrected a number of other previous mistakes, which had been introduced due to propagation of the first error. Thus only one discrepancy was required to produce the correct output even though the original output contained many more errors.

Given a recurrent classifier h, we define the corresponding limited-discrepancy search space over complete outputs S_h as follows. Each search state in the space is represented as (x, D) where x is a structured input and D is a discrepancy set. We view a state (x, D) as equivalent to the input-output state $(x, h[D](x))$. The initial state function I simply returns $(x, \emptyset)$ which corresponds to the original output of the recurrent classifier. The successor function S for a state (x, D) returns the set of states of the form (x, D'), where D' is the same as D, but with an additional discrepancy. In this way, a path through the LDS search space starts at the output generated by the recurrent classifier and traverses a sequence of outputs that differ from the original by some number of discrepancies. Given a reasonably accurate h, we expect that high-quality outputs will be generated at relatively shallow depths of this search space and hence will be generated quickly.

Figure 5 illustrates[1] the limited-discrepancy search space. Each state consists of the input x, a discrepancy set D and the output produced by running the classifier with the specified discrepancy set, that is, $h[D](x)$. The root node has an empty discrepancy set. Nodes at level one contain discrepancy sets of size one and nodes at level two contain

[1] It may not be clear from this example, but we allow over-riding the discrepancies to provide the opportunity to recover from the search errors.

discrepancy sets of size two, and so on. The highlighted state corresponds to the smallest depth state containing the true output.

3.4 Search Space Quality

Recall that in our experiments we train recurrent classifiers via exact imitation, which is an extremely simple approach compared to more elaborate methods such as Searn. We now show the desirable property that the "exact imitation accuracy" optimized by that approach is directly related to the "quality" of the LDS search space, where quality relates the expected amount of search needed to uncover the target output. More formally, given an input-output pair (x, y) we define the *LDS target depth* for an example (x, y) and recurrent classifier h to be the minimum depth of a state in the LDS space corresponding to y. Given a distribution over input-output pairs we let $d(h)$ denote the expected LDS target depth of a classifier h. Intuitively, the depth of a state in a search space is highly related to the amount of search time required to uncover the node (exponentially related for exhaustive search, and at least linearly related for more greedy search). Thus, we will use $d(h)$ as a measure of the quality of the LDS space. We now relate $d(h)$ to the classifier error rate.

For simplicity, assume that all decision sequences for the structured-prediction problem have a fixed length T and consider an input-output pair (x, y), which has a corresponding sequence of actions that generates y. Given a classifier h, we define its *exact imitation error* on (x, y) to be e/T where e is the number of mistakes h makes at nodes along the action sequence of (x, y) (i.e., how often does it disagree with the optimal classifier along the path of the optimal classifier). Further, given a distribution over input-output pairs, we let $\epsilon_{ei}(h)$ denote the expected exact imitation error with respect to examples drawn from the distribution. Note that the exact imitation training approach aims to learn a classifier that minimizes $\epsilon_{ei}(h)$. Also, let $\epsilon_r(h)$ denote the *expected recurrent error* of h, which is the expectation over randomly drawn (x, y) of the Hamming distance between the action sequence produced by h when applied to x and the true action sequence for (x, y). The error $\epsilon_r(h)$ is the actual measure of performance of h when applied to structured prediction. Recall that due to error propagation it is possible that $\epsilon_r(h)$ can be much worse than $\epsilon_{ei}(h)$, by as much as a factor of T (e.g., see Ross et al., 2011). The following proposition shows that $d(h)$ is related to $\epsilon_{ei}(h)$ rather than the potentially much larger $\epsilon_r(h)$.

Proposition 1 *For any classifier h and distribution over structured input-outputs, $d(h) = T\epsilon_{ei}(h)$.*

Proof For any example (x, y) the depth of y in S_h is equal to the number of imitation errors made by h on (x, y). To see this, simply create a discrepancy set D that contains a discrepancy at the position of each imitation error that corrects the error. This set is at a depth equal to the number of imitation errors and the classifier $h[D]$ will exactly produce the action sequence that corresponds to y. The result follows by noting that the expected number of imitation errors is equal to $T\epsilon_{ei}(h)$. ■

It is illustrative to compare this result with the Flipbit space. Let $d'(h)$ be the expected target depth in the Flipbit space of a randomly drawn (x, y). It is easy to see that $d'(h) = T\epsilon_r(h)$ since each search step can only correct a single error and the expected number of errors of the action sequence at the initial node is $T\epsilon_r(h)$. Since in practice and in theory $\epsilon_r(h)$ can be substantially larger than $\epsilon_{ei}(h)$ (by as much as a factor of T), this shows that the LDS space will often be superior to the baseline Flipbit space in terms of the expected target depth. For example, the target depth of the example LDS space in Figure 5 is one and is much smaller than the target depth of the example flipbit space in Figure 3, which is equal to five. Since this depth relates to the difficulty of search and cost-function learning, we can expect the LDS space to be advantageous when $\epsilon_r(h)$ is larger than $\epsilon_{ei}(h)$. In our experiments, we will see that this is indeed the case.

3.5 Sparse Search Spaces

In this section, we first discuss some of the scalability issues that arise in our framework due to the use of LDS and Flipbit spaces as defined above. Next, we describe how to define sparse versions of these search spaces to improve the efficiency of our approach.

In our search-based framework, the most computationally demanding part is the generation of candidate states during the search process. Given a parent state, the number of successor states for both the LDS and flipbit spaces is equal to $T \cdot (L-1)$ where T is the size of the structured output and L is the number of primitive action choices (the number of labels for sequence labeling problems). When T and/or L is large the time required for this expansion and computing the feature vector for each successor can be non-trivial. While we cannot control T, since that is dictated by the size of the input, we can consider reducing the effective size of L via pruning. Intuitively, for many sequence positions of a structured output, there will often be labels that can be easily determined to be bad by looking at the confidence of the recurrent classifier. By explicitly pruning those labels from consideration we can arrive at a sparse successor set that requires significantly less computation to generate.

More formally, our approach for defining a sparse successor function assumes that the recurrent classifier used to define the LDS and flipbit spaces produces confidence scores, rather than just simple classifications. This allows us to provide a ranking of the potential actions, or labels, at each position of the structured output. Using this ranking information it is straightforward to define sparse versions of the LDS and flipbit successor function by only considering successors corresponding to the top k labels at each sequence position. For the flipbit space this means that successors correspond to any way of changing the choice of the recurrent classifier at a sequence position to one of the top k labels at that position. For the LDS space, this means that we only consider introducing discrepancies at position i involving the top k labels at position i. In this way, the number of successors of a state in either space will be $T \cdot k$ rather than $T \cdot (L-1)$. Therefore, these sparse search spaces will lead to significant speed improvements for problems with large L (e.g., POS tagging, Handwriting recognition, and Phoneme prediction).

The potential disadvantage of using a small value of k is that accuracy could be hurt if good solution paths are pruned away due to inaccuracy of the classifier's confidence estimates. Thus, k provides a way to trade-off prediction speed versus accuracy. As we will show later in the experiments, we are generally able to find values of k that lead to significant speedups with little loss in accuracy.

4. Cost Function Learning

In this section, we describe a generic framework for cost function learning that is applicable for a wide range of search spaces and search strategies. This approach is motivated by our empirical observation that for a variety of structured prediction problems, we can uncover high quality outputs if we guide the output-space search using the true loss function as an oracle cost function to guide the search (close to zero error with both LDS and Flipbit spaces). Since the loss function depends on correct target output y^*, which is unknown at test time, we aim to learn a cost function that mimics this oracle search behavior on the training data without requiring knowledge of y^*. With an appropriate choice of hypothesis space of cost functions, good performance on the training data translates to good performance on the test data.

4.1 Cost Function Learning via Imitation Learning

Recall that in our output space search framework, the role of the cost function $\mathcal{C}$ is to evaluate the complete outputs that are uncovered by the search procedure. These evaluations may be used internally by the search procedure as a type of heuristic guidance and also used when the time bound is reached to return the least cost output that has been uncovered as the prediction. Based on our empirical observations, it is very often the case that the true loss function serves these roles very effectively, which might suggest a goal of learning a cost function that is approximately equal to the true loss function L over all possible outputs. However, this objective will often be impractical and fortunately is unnecessary. In particular, the learned cost function need not approximate the true loss function uniformly over the output space, but only needs to make the decisions that are sufficient for leading the time-bounded search to behave as if it were using the true loss function. Often this allows for a much less constrained learning problem, for example, $\mathcal{C}$ may only need to preserve the rankings among certain outputs, rather than exactly matching the values of L. The key idea behind our cost learning approach is to learn such a sufficient $\mathcal{C}$. The main assumptions made by this approach are: 1) the true loss function can provide effective guidance to the search procedure by making a series of ranking decisions, and 2) we can learn to imitate those ranking decisions sufficiently well.

Our goal now is to learn a cost function that causes the search to behave as if it were using loss function L for guiding the search and selecting the final output. We propose to formulate and solve this problem in the framework of imitation learning. In traditional imitation learning, the goal of the learner is to learn to imitate the behavior of an expert performing a sequential-decision making task in a way that generalizes to similar tasks or situations. Typically this is done by collecting a set of trajectories of the expert's behavior on a set of training tasks. Then supervised learning is used to find a policy that can replicate the decisions made on those trajectories. Often the supervised learning problem corresponds to learning a classifier or policy from states to actions and off-the-shelf tools can be used.

In our cost function learning problem, the expert corresponds to the search procedure $\mathcal{A}$ using the loss function L for a search time bound τ_{max}. The behavior that we would like to imitate is the internal behavior of this search procedure, which consists of all decisions made during the search including the final decision of which output to return. Thus, the goal of cost function learning is to learn the weights of $\mathcal{C}$ so that this behavior is replicated when it is used by the search procedure in place of L. We propose to achieve this goal by directly monitoring the expert search process on all of the structured training examples and generating the set of constraints on L that were responsible for the observed decisions. Then we attempt to learn a $\mathcal{C}$ that satisfies the constraints using an optimization procedure.

Algorithm 2 Cost Function Learning via Exact Imitation

```
Input: $\mathcal{D}$ = Training examples, $(I, S)$ = Search space definition, $L$ = Loss function, $\mathcal{A}$ = Rank-based search procedure, $\tau_{max}$ = search time bound
Output: $\mathcal{C}$, the cost function
1:  Initialize the set of ranking examples $\mathcal{R} = \emptyset$
2:  for each training example $(x, y^*) \in \mathcal{D}$ do
3:      $s_0 = I(x)$ // initial state of the search tree
4:      $M_0 = \{s_0\}$ // set of open nodes in the internal memory of the search procedure
5:      $y_{best}$ =OutputOf($s_0$) // best loss output so far
6:      for each search step $t = 1$ to $\tau_{max}$ do
7:          Select the state(s) to expand: $N_t$ =Select($\mathcal{A}, L, M_{t-1}$)
8:          Expand every state $s \in N_t$ using the successor function $S$: $C_t$ =Expand($N_t, S$)
9:          Prune states and update the internal memory state of the search procedure:
            $M_t$ =Prune($\mathcal{A}, L, M_{t-1} \cup C_t \setminus N_t$)
10:         Update the best loss output $y_{best}$ // track the best output
11:         Generate ranking examples $R_t$ to imitate this search step
12:         Add ranking examples $R_t$ to $\mathcal{R}$: $\mathcal{R} = \mathcal{R} \cup R_t$ // aggregation of training data
13:     end for
14: end for
15: $\mathcal{C}$ =Rank-Learner($\mathcal{R}$) // learn cost function from all the ranking examples
16: return learned cost function $\mathcal{C}$
```

Algorithm 2 describes our generic approach for cost function learning via exact imitation of searches conducted by the loss function. It is applicable to a wide-range of search spaces, search procedures and loss functions. The learning algorithm takes as input: 1) $\mathcal{D} = \{(x, y^*)\}$, set of training examples for a structured prediction problem (e.g., handwriting recognition); 2) $\mathcal{S}_o = (I, S)$, definition of a search space over complete outputs (e.g., LDS space), where I is the initial state function and S is the successor function; 3) L, a task loss function defined over complete outputs (e.g., hamming loss); 4) $\mathcal{A}$, a rank-based search procedure (e.g., greedy search); 5) τ_{max}, the search time bound (e.g., number of search steps).

First, it runs the search procedure $\mathcal{A}$ with the given loss function L, in the search space $\mathcal{S}_o$ instantiated for every training example (x, y^*), upto the maximum time bound τ_{max} (steps 3-10), and generates a set of pair-wise ranking examples that need to be satisfied to be able to imitate the search behavior with loss function (step 11). Second, the aggregate set of ranking examples collected over all the training examples is then given to a rank-learning algorithm (e.g., Perceptron or SVM-Rank) to learn the weights of the cost function (step 15).

The algorithmic description assumes a best-first search procedure with some level of pruning (e.g., greedy search and best-first beam search). These search procedures typically involve three key steps: 1) Selection, 2) Expansion and 3) Pruning. During selection, the search procedure selects one or more[2] open nodes from its internal memory for expansion (step 7), and expands all the selected nodes to generate the candidate set (step 8). It retains only a subset of all the open nodes after expansion in its internal memory and prunes away all the remaining ones (step 9). For example, greedy search maintains only the best node and best-first beam search with beam width b retains the best b nodes.

The most important step in our cost function learning algorithm is the generation of ranking examples to imitate the search procedure (step 11). In what follows, we first formalize ranking-based search that allows us to specify what these pairwise ranking examples are and then, give a generic description of "sufficient" pair-wise decisions to imitate the search, and illustrate them for greedy search and best-first beam search through a simple example.

4.2 Ranking-based Search

We now precisely define the notion of "guiding the search" with a loss function. If the loss function can be invoked arbitrarily by the search procedure, for example, evaluating and comparing arbitrary expressions involving the cost, then matching its performance would require the cost function to approximate it arbitrarily closely, which is quite demanding in most cases. Hence, we restrict ourselves to ranking-based search defined as follows.

Let $\mathcal{P}$ be an anytime search procedure that takes an input $x \in \mathcal{X}$, calls a cost function $\mathcal{C}$ over the pairs from $\mathcal{X} \times \mathcal{Y}$ some number of times and outputs a structured output $y_{best} \in \mathcal{Y}$. We say that $\mathcal{P}$ is a ranking-based search procedure if the results of calls to $\mathcal{C}$ are only used to compare the relative values for different pairs (x, y) and (x, y') with a fixed tie breaker. Each such comparison with tie-breaking is called a ranking decision and is characterized by the tuple (x, y, y', d), where d is a binary decision that indicates y is a better output than y' for input x. When requested, it returns the best output y_{best} encountered thus far as evaluated by the cost function.

Note that the above constraints prohibit the search procedure from being sensitive to the absolute values of the cost function for particular search states (x, y) pairs, and only consider their relative values. Many typical search strategies such as greedy search, best-first search, and beam search satisfy this property.

A *run* of a ranking-based search is a sequence $x, m_1, o_1, \ldots, m_n, o_n, y$, where x is the input to the predictor, y is the output, and m_i is the internal memory state of the predictor just before the i^{th} call to the ranking function. o_i is the i^{th} ranking decision (x, y_i, y'_i, d_i). Given a hypothesis space $\mathcal{H}$ of cost functions, the cost function learning works as follows. It runs the search procedure $\mathcal{P}$ on each training example (x, y^*) for a maximum search time bound of τ_{max} substituting the loss function $L(x, y, y^*)$ for the cost function $\mathcal{C}(x, y)$. For each run, it records the set of all ranking decisions (x, y_i, y'_i, d_i). The set of all ranking decisions from all the runs is given as input to a binary classifier, which finds a cost function $C \in \mathcal{H}$, consistent with the set of all such ranking decisions.

[2] Breadth-first beam search expands all nodes in the beam.

The ranking-based search can be viewed as a Markov Decision Process (MDP), where the internal states of the search procedure correspond to the states of the MDP, and the ranking decision is an action. The following theorem can be proved by adapting the proof of Fern et al. (2006) with minor changes, for example, no discounting, and two actions, and applies to stochastic as well as deterministic search procedures.

Theorem 2 *Let $\mathcal{H}$ be a finite class of ranking functions. For any target ranking function $r \in \mathcal{H}$, and any set of $m = \frac{1}{\epsilon} \ln \frac{|\mathcal{H}|}{\delta}$ independent runs of a rank-based search procedure $\mathcal{P}$ guided by r drawn from a target distribution over inputs, there is a $1 - \delta$ probability that every $\hat{r} \in \mathcal{H}$ that is consistent with the runs satisfies $L(\hat{r}) \leq L(r) + 2\epsilon L_{max}$, where L_{max} is the maximum possible loss of any output and $L(t)$ is the expected loss of running the search procedure $\mathcal{P}$ with the ranking function $t \in \mathcal{H}$.*

Although the theoretical result assumes that the target cost function h is in the hypothesis space, in practice this is not guaranteed as the set of generated constraints might be quite large and diverse. To help reduce the complexity of the learning problem, in practice we only learn from a smaller set of pair-wise ranking decisions that are sufficient (see below) to preserve the best output that is encountered during search at any time step.

4.3 Sufficient Pairwise Decisions

Above we noted that we only need to collect and learn to imitate the "sufficient" pairwise decisions encountered during search. We say that a set of constraints is sufficient for a structured training example (x, y^*), if any cost function that is consistent with the constraints causes the search to follow the same trajectory (sequence of states) and retains the best loss output that is encountered during search so far for input x. The precise specification of these constraints depends on the actual search procedure that is being used. For rank-based search procedures, the sufficient constraints can be categorized into three types:

1. *Selection* constraints, which ensure that the search node(s) from the internal memory state that will be expanded in the next search step is (are) ranked better than all other nodes.

2. *Pruning* constraints, which ensure that the internal memory state (set of search nodes) of the search procedure is preserved at every search step. More specifically, these constraints involve ranking every search node in the internal memory state better (lower $\mathcal{C}$-value) than those that are pruned.

3. *Anytime* constraints, which ensure that the search node corresponding to the best loss output that is encountered during search so far is ranked better than every other node that is uncovered by the search procedure.

Below, we will illustrate these constraints concretely for greedy search and best-first beam search noting that similar formulations for other rank-based search procedures is straightforward.

Greedy Search: This is the most basic rank-based search procedure. For a given input x, it traverses the search space by selecting the next state as the successor of the current

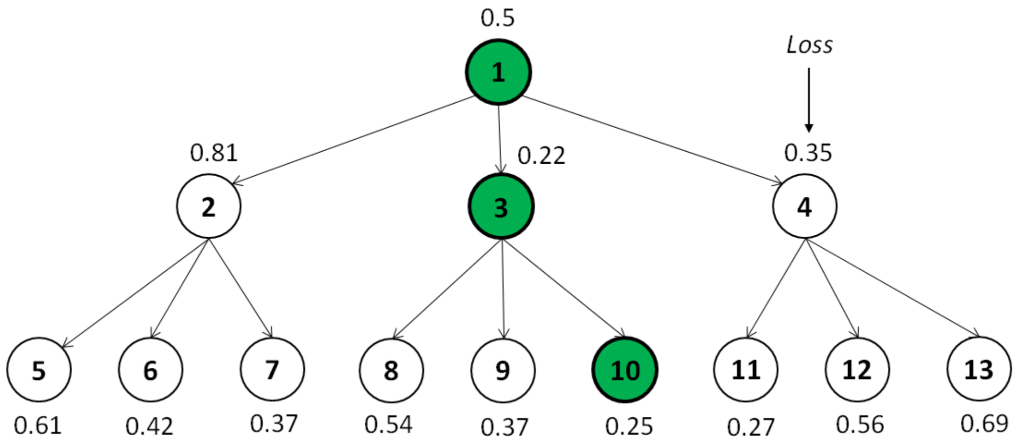


Figure 6: An example search tree that illustrates greedy search with loss function. Each node represents a complete input-output pair and can be evaluated using the loss function. The highlighted nodes correspond to the trajectory of greedy search guided by the loss function.

state that looks best according to the cost function $\mathcal{C}$ (loss function L during training). In particular, if s_i is the search state at step i, greedy search selects $s_{i+1} = \arg min_{s \in S(s_i)} \mathcal{C}(s)$, where $s_0 = I(x)$. In greedy search, the internal memory state of the search procedure at step i consists of only the best open (unexpanded) node s_i. Additionally, it keeps track of the best node s_i^{best} uncovered so far as evaluated by the cost function.

Let (x, y_i) correspond to the input-output pair associated[3] with state s_i. Since greedy search maintains only a single open node s_i in its internal memory at every search step i, there are no selection constraints. Let C_{i+1} be the candidate set after expanding state s_i, that is, $C_{i+1} = S(s_i)$. Let s_{i+1} be the best node in the candidate set C_{i+1} as evaluated by the loss function, that is, $s_{i+1} = \arg min_{s \in C_{i+1}} L(s)$. As greedy search prunes all the nodes in the candidate set other than s_{i+1}, pruning constraints need to ensure that s_{i+1} is ranked better than all the other nodes in C_{i+1}. Therefore, we include one ranking constraint for every node $(x, y) \in C_{i+1} \setminus (x, y_{i+1})$ such that $\mathcal{C}(x, y_{i+1}) < \mathcal{C}(x, y)$. As part of the anytime constraints, we introduce a constraint between (x, y_i^{best}) and (x, y_{i+1}) according to their losses. For example, if $L(x, y_{i+1}, y^*) < L(x, y_i^{best}, y^*)$, we introduce a ranking constraint such that $\mathcal{C}(x, y_{i+1}) < \mathcal{C}(x, y_i^{best})$ and vice versa.

We will now illustrate these ranking constraints through an example. Figure 6 shows an example search tree of depth two with associated losses for every search node. The highlighted nodes correspond to the trajectory of greedy search with loss function that our learner has to imitate. At first search step, $\{\mathcal{C}(3) < \mathcal{C}(2), \mathcal{C}(3) < \mathcal{C}(4)\}$, and $\{\mathcal{C}(3) < \mathcal{C}(1)\}$ are the pruning and anytime constraints respectively. Similarly, $\{\mathcal{C}(10) < \mathcal{C}(8), \mathcal{C}(10) < \mathcal{C}(9)\}$, and $\{\mathcal{C}(3) < \mathcal{C}(10)\}$ form the pruning and anytime constraints at second search step. There-

[3] We use input-output pair (x, y_i) and state s_i inter-changeably for the sake of brevity.

fore, the aggregate set of constraints needed to imitate the greedy search behavior shown in Figure 6 are:
$\{\mathcal{C}(3) < \mathcal{C}(2), \mathcal{C}(3) < \mathcal{C}(4), \mathcal{C}(3) < \mathcal{C}(1), \mathcal{C}(10) < \mathcal{C}(8), \mathcal{C}(10) < \mathcal{C}(9), \mathcal{C}(3) < \mathcal{C}(10)\}$.

Best-first Beam Search: This is a more sophisticated search procedure compared to greedy search. Best-first beam search maintains a set of b open nodes B_i in its internal memory at every search step i, where b is the beam width. Greedy search is a special case of beam search, where beam width b equals 1. For a given input x, it traverses the search space by expanding the best node s_i in the current beam B_i (i.e., $s_i = \arg min_{s \in B_i} \mathcal{C}(s)$) and computes the next beam B_{i+1} by retaining the best b open nodes after expanding s_i, where $B_0 = \{I(x)\}$.

Best-first beam search selects the best node s_i at every search step i from its beam B_i for expansion, that is, $s_i = \arg min_{s \in B_i} L(s)$. Therefore, selection constraints need to ensure that s_i is ranked better than all the other nodes in beam B_i. Therefore, we include one ranking constraint for every node $(x, y) \in B_i \setminus (x, y_i)$ such that $\mathcal{C}(x, y_i) < \mathcal{C}(x, y)$. Let C_{i+1} be the candidate set after expanding state s_i, that is, $C_{i+1} = S(s_i) \cup B_i \setminus s_i$ and let B_{i+1} be the best b nodes in the candidate set according to the loss function. As best-first beam search prunes all the nodes in the candidate set other than those in B_{i+1}, pruning constraints need to ensure that every node in B_{i+1} is ranked better than every node in $C_{i+1} \setminus B_{i+1}$. Therefore, we generate one ranking example for every pair of nodes $((x, y_b), (x, y)) \in B_{i+1} \times C_{i+1} \setminus B_{i+1}$, requiring that $\mathcal{C}(x, y_b) < \mathcal{C}(x, y)$. Similar to greedy search, as part of the anytime constraints, we introduce a ranking constraint between (x, y_i^{best}) and (x, y_{i+1}) according to their losses.

We will now illustrate the above ranking constraints for best-first beam search through the example search tree in Figure 6. Let us consider best-first beam search with beam width $b = 2$ and let (B_0, B_1, B_2) correspond to the beam trajectory of search with loss function that needs to be imitated by our learner, where $B_0 = \{1\}$, $B_1 = \{3, 4\}$ and $B_2 = \{4, 10\}$. At first search step, there are no selection constraints as B_0 contains only a single node, and $\{\mathcal{C}(3) < \mathcal{C}(2), \mathcal{C}(4) < \mathcal{C}(2)\}$ and $\{\mathcal{C}(3) < \mathcal{C}(1)\}$ are the pruning and anytime constraints respectively. Similarly, $\{\mathcal{C}(3) < \mathcal{C}(4)\}$, $\{\mathcal{C}(4) < \mathcal{C}(8), \mathcal{C}(4) < \mathcal{C}(9), \mathcal{C}(10) < \mathcal{C}(8), \mathcal{C}(10) < \mathcal{C}(9)\}$ and $\{\mathcal{C}(3) < \mathcal{C}(10)\}$ form the selection, pruning and anytime constraints at second search step.

The only thing that remains to be explained in Algorithm 2 is, how to learn a cost function $\mathcal{C}$ from the aggregate set of ranking examples $\mathcal{R}$ (step 15). Below we describe this rank learning procedure.

4.4 Rank Learner

We can use any off-the-shelf rank-learning algorithm (e.g., Perceptron, SVM-Rank) as our base learner to learn the cost function from the set of ranking examples $\mathcal{R}$. In our specific implementation we employed the online Passive-Aggressive (PA) algorithm (Crammer et al., 2006) as our base learner.[4] Training was conducted for 50 iterations in all of our experiments.

[4] In the conference version of this paper, we employed the perceptron learner and followed a training approach that slightly differs from Algorithm 2. Specifically, the ranking examples for exact imitation were generated until reaching y^*, the correct output, and after that we only generate training examples to rank y^* higher than the best cost open node(s) as evaluated by the current cost function and continue the search guided by the cost function. This particular training approach may be beneficial (results in

PA is an online large-margin algorithm, which makes several passes over the training examples $\mathcal{R}$, and updates the weights whenever it encounters a ranking error. Recall that each ranking example in $\mathcal{R}$ is of the form $\mathcal{C}(x, y_1) < \mathcal{C}(x, y_2)$, where x is a structured input with target output y^*, y_1 and y_2 are potential outputs for x such that $L(x, y_1, y^*) < L(x, y_2, y^*)$. Let $\Delta > 0$ be the difference between the losses of the two outputs involved in a ranking example. We experimented with PA variants that use margin scaling (margin scaled by Δ) and slack scaling (errors weighted by Δ) (Tsochantaridis et al., 2005). Since margin scaling performed slightly better than slack scaling, we report the results of the PA variant that employs margin scaling. Below we give the full details of the margin scaling update.

Let w_t be the current weights of the cost function. If there is a ranking error, that is, $w_t \cdot \Phi(x, y_2) - w_t \cdot \Phi(x, y_1) < \sqrt{\Delta}$, the new weights w_{t+1} that corrects the error can be obtained using the following equation.[5]

$$w_{t+1} = w_t + \tau_t(\Phi(x, y_2) - \Phi(x, y_1))$$

where the learning rate τ_t is given by

$$\tau_t = \frac{w_t \cdot \Phi(x, y_1) - w_t \cdot \Phi(x, y_2) + \sqrt{\Delta}}{\|\Phi(x, y_2) - \Phi(x, y_1)\|^2}.$$

This specific update has been previously used for cost-sensitive multiclass classification (Crammer et al., 2006) (See Equation 51) and for structured output problems (Keshet et al., 2005) (See Equation 7).

4.5 Summary of Overall Training Approach

Our search-based framework thus consists of two main learning components: 1) the search space learner, and 2) the cost function learner. We train them sequentially. First, we train the recurrent classifier as described in Section 3.1, which is used to define either the LDS or flipbit search spaces (see Section 3). Second, we train the cost function $\mathcal{C}$ to score the outputs for a given combination of search space over complete outputs S_o and a search procedure $\mathcal{A}$ as described in Algorithm 2. More specifically, for every training example (x, y^*), we run the search procedure $\mathcal{A}$ on the search space S_o instantiated for input x, using the loss function L for the specified time bound τ, and generate imitation training data (ranking examples) for the cost function learning (see Section 4.3). We give the aggregate set of imitation training data to a rank learner to train the cost function $\mathcal{C}$ as described in Section 4.4.

the conference paper are slightly better than those presented in this article and in practice, breaking ties via cost function is better than employing a random tie-breaker), but it is computationally very expensive (requires the ranking examples to be generated on-the-fly during each iteration of online training). However, this training methodology can be both beneficial and practical when applied on sparse search spaces.

[5] Crammer et al. (2006) prove bounds on the cumulative squared loss and therefore, they employ this particular margin constraint with $\sqrt{\Delta}$.

5. Empirical Results

In this section we empirically investigate our approach along several dimensions and compare it against the state-of-the-art in structured prediction.

5.1 Experimental Setup

We evaluate our approach on the following six structured prediction problems including five benchmark sequence labeling problems and a 2D image labeling problem.

- **Handwriting Recognition (HW).** The input is a sequence of binary-segmented handwritten letters and the output is the corresponding character sequence $[a-z]^{+}$. This data set contains roughly 6600 examples divided into 10 folds (Taskar et al., 2003). We consider two different variants of this task as in Hal Daumé III et al. (2009). For the `HW-Small` version of the problem,, we employ one fold for training and the remaining 9 folds for testing, and vice-versa in `HW-Large`.

- **NETtalk Stress.** This is a text-to-speech mapping problem, where the task is to assign one of the 5 stress labels to each letter of a word (Sejnowski and Rosenberg, 1987). There are 1000 training words and 1000 test words in the standard data set. We use a sliding window of size 3 for observational features.

- **NETtalk Phoneme.** This is similar to NETtalk Stress except that the task is to assign one of the 51 phoneme labels to each letter of a word.

- **Chunking.** The goal in this task is to syntactically chunk English sentences into meaningful segments. We consider the full syntactic chunking task and use the data set from the CONLL 2000 shared task,[6] which consists of 8936 sentences of training data and 2012 sentences of test data.

- **POS tagging.** We consider the tagging problem for the English language, where the goal is to assign the part-of-speech tag to each word in a sentence. The standard data from Wall Street Journal (WSJ) corpus[7] was used in our experiments.

- **Scene labeling.** This data set contains 700 images of outdoor scenes (Vogel and Schiele, 2007). Each image is divided into patches by placing a regular grid of size 10×10 over the entire image, where each patch takes one of the 9 semantic labels (*sky, water, grass, trunks, foliage, field, rocks, flowers, sand*). Simple appearance features including color, texture and position are used to represent each patch. Training was performed with 600 images, and the remaining 100 images were used for testing.

We used F_1 loss as the loss function for the chunking task and employed Hamming loss for all other tasks.

For all sequence labeling problems, the recurrent classifier labels a sequence using a left-to-right ordering and for scene labeling uses an ordering of top-left to right-bottom in a row-wise raster form. To train the recurrent classifiers, the output label of the previous

[6] CONLL task can be found at `http://www.cnts.ua.ac.be/conll2000/chunking/`.

[7] WSJ corpus can be found at `http://www.cis.upenn.edu/~treebank/`.

token is used as a feature to predict the label of the current token for all sequence labeling problems with the exception of chunking and POS tagging, where labels of the two previous tokens were used. For scene labeling, the labels of neighborhood (top and left) patches were used. In all our experiments, we train the recurrent classifier using exact imitation (see Section 3) with the Perceptron algorithm for 100 iterations with a learning rate of 1.

Unless otherwise indicated, the cost functions learned over input-output pairs are second order, meaning that they have features over neighboring label pairs and triples along with features of the structured input. For the scene labeling task, we consider pairs and triples along both horizontal and vertical directions. We trained the cost function via exact imitation as described in Section 4 using 50 iterations of Passive-Aggressive training.

5.2 Comparison to State-of-the-Art

We experimented with several instantiations of our framework. First, we consider our framework using a greedy search procedure for both the LDS and flipbit spaces, denoted by **LDS-Greedy** and **FB-Greedy**. Unless otherwise noted, in both training and testing, the greedy search was run for a number of steps equal to the length of the sequence. Using longer runs did not impact results significantly. Second, we performed experiments with best-first beam search for different beam widths and search steps, but we didn't see significant improvements over the results with greedy search. Therefore, we do not report these results. Third, to see the impact of adding additional search at test time to a greedily trained cost function, we also used the cost function learned by LDS-Greedy and FB-Greedy in the context of a best-first beam search (beam width = 100) at test time in both the LDS and flipbit spaces, denoted by **LDS-BST(greedy)** and **FB-BST(greedy)**. We also report the performance of using our trained recurrent classifier (**Recurrent**) to make predictions, which is equivalent to performing no search since both search spaces are initialized to the recurrent classifier output. Finally, we also report the exact imitation accuracy ($100 * (1 - \epsilon_{ei})$), which as described earlier (see Section 3) is the accuracy being directly optimized by the recurrent classifier and is related to the structure of the flipbit and LDS spaces.

We compare our results with other structured prediction algorithms including **CRFs** (Lafferty et al., 2001), **SVM-Struct** (Tsochantaridis et al., 2004), **SEARN** (Hal Daumé III et al., 2009) and **CASCADES** (Weiss and Taskar, 2010). For these algorithms, we report the best published results whenever available. In the remaining cases, we used publicly available code or our own implementation to generate those results. Ten percent of the training data was used to tune the hyper-parameters. CRFs were trained using SGD.[8] SVM^{hmm} was used to train SVM-Struct and the value of the parameter C was chosen from $\{10^{-4}, 10^{-3}, \cdots, 10^{3}, 10^{4}\}$ based on the validation set. Cascades were trained using the implementation[9] provided by the authors, which can be used for sequence labeling problems with Hamming loss. We present two different results for Cascades: 1) Cascades(2012) employs the version of the code at the original time this work was done, 2) Cascades(updated) employs the most recent updated[10] version of the Cascades training

[8] SGD code can be found at `http://leon.bottou.org/projects/sgd`.

[9] Cascades code can be found at `http://code.google.com/p/structured-cascades/`.

[10] Most recent based on personal communication with the author.

Algorithms	Data Sets						
	`HW-Small`	`HW-Large`	`Stress`	`Phoneme`	`Chunk`	`POS`	`Scene labeling`
a. Comparison to state-of-the-art							
$100 * (1 - \epsilon_{ei})$	73.9	83.99	77.97	77.09	88.84	92.5	**78.61**
Recurrent	65.67	74.87	72.82	73.58	88.51	92.15	56.64
LDS-Greedy	82.59	92.59	78.85	79.09	94.62	**96.93**	72.95
FB-Greedy	80.3	89.38	77.93	78.43	93.96	96.87	67.67
CRF	80.03	86.89	78.52	78.91	**94.77**	96.84	-
SVM-Struct	80.36	87.51	77.99	78.3	93.64	96.81	-
Searn	82.12^{B}	90.58^{B}	76.15	77.26	94.44^{B}	95.83	62.31
Cascades(2012)	69.62	87.95	77.18	69.77	-	96.82	-
Cascades(updated)	**86.98**	**96.78**	**79.59**	**82.44**	-	96.82	-
b. Results with Additional Search							
LDS-BST(greedy)	**83.81**$^{+}$	**93.17**$^{+}$	78.76	78.87	94.63	96.95	**74.12**$^{+}$
FB-BST(greedy)	81.19^{+}	90.21^{+}	77.61	78.32	93.98	96.91	69.23^{+}
c. Results with DAgger							
LDS-Greedy	**83.62**$^{+}$	**93.24**$^{+}$	**79.81**$^{+}$	**79.97**$^{+}$	94.61	96.91	**74.27**$^{+}$
FB-Greedy	81.28^{+}	90.45^{+}	78.96^{+}	79.23^{+}	93.94	96.89	69.63^{+}
d. Results with Third-Order Features							
LDS-Greedy	85.85^{+}	95.08^{+}	80.21^{+}	81.61^{+}	94.63	96.97	**74.71**$^{+}$
FB-Greedy	83.18^{+}	92.66^{+}	79.23^{+}	80.65^{+}	94.17^{+}	96.94	69.81^{+}
Cascades(2012)	81.87^{+}	93.76^{+}	73.48	68.98	-	96.84	-
Cascades(updated)	**89.18**$^{+}$	**97.84**$^{+}$	**80.49**$^{+}$	**82.59**$^{+}$	-	96.84	-

Table 1: Prediction accuracy results of different structured prediction algorithms and variations of our framework. A + indicates that the particular variation being considered resulted in improvement.

code, which significantly improves on the Cascades(2012). For Searn we report the best published results with a linear classifier (i.e., linear SVMs instead of Perceptron) as indicated by B in the table and otherwise ran our own implementation of Searn with optimal approximation as described in Hal Daumé III et al. (2009) and optimized the interpolation parameter β over the validation set. Note that we do not compare our results to SampleRank due to the fact that its performance is highly dependent on the hand-designed proposal distribution, which varies from one domain to another.

Table 1a shows the prediction accuracies of different algorithms ('-' indicates that we were not able to generate results for those cases as the software package was not directly applicable). Across all benchmarks we see that the most basic instantiations of our framework, LDS-Greedy and FB-Greedy, produce results that are comparable or significantly better than all the other methods excluding[11] Cascades(updated). This is particularly interesting, since these results are achieved using a relatively small amount of search and the simplest search method, and results tend to be the same or better for our other instantiations. A likely reason that we are outperforming CRFs and SVM-Struct is that we use

[11] A followup work (Doppa et al., 2014) that employs two distinct functions for guiding the search and scoring the candidate outputs generated during search performs comparably or better than Cascades(updated) across all benchmarks.

second-order features while those approaches use first-order features, since exact inference with higher order features is too costly, especially during training. As stated earlier, one of the advantages of our approach is that we can use higher-order features with negligible overhead.

Finally, the improvement in the scene labeling domain is the most significant, where Searn achieves an accuracy of 62.31 versus 72.95 for LDS-Greedy. In this domain, most prior work has considered the simpler task of classifying entire images into one of a set of discrete classes, but to the best of our knowledge no one has considered a structured prediction approach for patch classification. The only reported result for patch classification that we are aware of Vogel and Schiele (2007) obtains an accuracy of 71.7 (versus our best performance of 74.27) with non-linear SVMs trained i.i.d. on patches using more sophisticated features than ours.

5.3 Framework Variations

Adding More Search. Table 1b shows that LDS-BST(greedy) and FB-BST(greedy) are generally the same or better than LDS-Greedy and FB-Greedy, with the biggest improvements in handwriting recognition task and the challenging scene labeling ('+' indicates improvement). Results improve from 82.59 to 83.81 in HW-Small, from 92.59 to 93.17 in HW-Large and from 72.95 to 74.12 in the scene labeling task. This shows that it can be an effective strategy to train using greedy search and then insert that cost function into a more elaborate search at test time for further improvement. As noted earlier, in the domains we considered, training with a more sophisticated search procedure like beam search did not improve results over greedy search. This demonstrates the efficiency of our search spaces and can be considered as a positive result.

Exact Imitation vs. DAgger. Our experiments show that the simple exact imitation approach for cost function training performs extremely well on our problems. However, cost functions trained via exact imitation can be prone to error propagation (Kääriäinen, 2006; Ross and Bagnell, 2010). It is interesting to consider whether addressing this issue might improve results further. Therefore, we experimented with DAgger (Ross et al., 2011), a more advanced imitation training regime that addresses error propagation through on-line training and expert demonstrations. At a high-level, DAgger learns on-line from an aggregate data set collected over several iterations. The first iteration corresponds to the data produced by exact imitation of the expert. Further iterations correspond to the actions suggested by the expert on trajectories produced by a mixture of the learned policy from the previous iteration and the expert policy. This allows DAgger to learn from states visited by its possibly erroneous learned policy and correct its mistakes using expert input. In our adaptation, the "learned policy" corresponds to the decisions made by the greedy search guided by the cost function as the heuristic, and the "expert policy" corresponds to the decisions made by greedy search guided by the loss function. Ross et al. (2011) show that during the iterations of DAgger just using the learned policy without mixing the expert policy performs very well across diverse domains. Therefore, we use the same setting in our DAgger experiments.

We picked the best cost function based on a validation set after 5 iterations of DAgger, noting that no noticeable improvement was observed after 5 iterations. Table 1c shows the results of LDS-Greedy and FB-Greedy obtained by training with DAgger. We see that there are some improvements over the cost function trained with exact imitation, although the improvements are quite small ('+' indicates improvement). As we will show later, we get much more positive results for DAgger in the context of pruned search spaces.

Higher-Order Features. One of the advantages of our framework compared to other approaches for structured prediction is the ability to use more expressive feature spaces with negligible computational overhead. Table 1d shows results using third-order features (compared to second-order results in Table 1a) for LDS-Greedy, FB-Greedy and Cascades.[12] Note that it is not practical to run the other methods (e.g., CRFs and SVM-Struct) using third-order features due to the substantial increase in inference time. The results of LDS-Greedy and FB-Greedy with third-order features improve over the corresponding results with second-order features across the board ('+' indicates improvement). Finally, we note that while Cascades(updated) is able to improve performance by using third-order features, the improvement is negligible for phoneme prediction.

LDS space vs. Flipbit space. We see that generally the instances of our method that use the LDS space outperform the corresponding instances that use the Flipbit space. Interestingly, if there is a large difference between the exact imitation accuracy $1 - \epsilon_{ei}$ and the recurrent classifier accuracy (e.g., Handwriting and Scene labeling), then the LDS space is significantly better than the flip-bit space. This is particularly true in our most complex problem of scene labeling where this difference is quite large, as is the gap between LDS and Flipbit. These results show the benefit of using the LDS space and empirically confirm our observations in Section 3 that the quality of the LDS and Flipbit spaces are related to the exact imitation and recurrent error rates respectively.

5.4 Results with Sparse Search Spaces

Recall that sparse search spaces are parameterized by k, the sparsity parameter (see Section 3.5). Small values of k lead to proportionately smaller branching factors for search. We perform experiments for different values of k to evaluate the effectiveness of sparse search spaces (i.e., LDS-k and FB-k). For example, **LDS-2** and **FB-2** correspond to the configurations where k equals 2. We only report the results for $k = 2$ and $k = 4$ noting that we didn't see major improvements for larger values of k. For all these experiments, we run greedy search for a number of steps equal to the length of the sequence during both training and testing. For greedy search, the computation time can be expected to be linearly related to k since the main computational bottleneck is the generation of $T \cdot k$ successors for each node encountered during the search.

Results of Cost Function Trained on Complete Search Spaces. Table 2b gives the results of using a cost function trained on a complete (non-sparse) search space (as in the previous experiments) to make predictions via the pruned spaces. As we can see, the gap between the results of LDS-2 and FB-2 and the corresponding results obtained using complete search space (see Table 2a) is very small. This means that we get huge speed improvements during testing with only a small loss in accuracy (more details on speedup below). The accuracy loss reduces with less sparse search spaces (LDS-4 and FB-4), but comes at the expense of more computation time.

[12] Cascades code can be found at `http://code.google.com/p/structured-cascades/`.

Algorithms	Data Sets						
	`HW-Small`	`HW-Large`	`Stress`	`Phoneme`	`Chunk`	`POS`	`Scene labeling`

a. Accuracy results of training and testing on complete search space

Algorithms	`HW-Small`	`HW-Large`	`Stress`	`Phoneme`	`Chunk`	`POS`	`Scene labeling`
LDS	**82.59**	**92.59**	**78.85**	**79.09**	**94.62**	96.93	**72.95**
FB	80.3	89.38	77.93	78.43	93.96	96.87	67.67

b. Accuracy results of cost function trained on complete search space

Algorithms	`HW-Small`	`HW-Large`	`Stress`	`Phoneme`	`Chunk`	`POS`	`Scene labeling`
LDS-2	81.02	91.38	78.62	79.79	93.95	96.13	69.87
FB-2	79.47	86.95	77.82	79.23	93.18	96.08	65.43
LDS-4	**82.55**	**92.45**	**78.85**	**79.97**	**94.55**	96.77	**72.11**
FB-4	80.43	88.40	77.93	79.23	94.23	96.81	66.98

c. Accuracy results of cost function trained on sparse search space via Exact Imitation

Algorithms	`HW-Small`	`HW-Large`	`Stress`	`Phoneme`	`Chunk`	`POS`	`Scene labeling`
LDS-2	80.17	90.43	78.72	78.90	94.08	96.29	68.62
FB-2	78.95	87.61	77.57	78.80	94.11	96.45	63.45
LDS-4	**83.12**	**92.84**	**78.85**	79.46	94.55	96.51	**70.69**
FB-4	80.80	90.04	77.93	**79.59**	94.39	96.57	65.67

d. Accuracy results of cost function trained on sparse search space via DAgger

Algorithms	`HW-Small`	`HW-Large`	`Stress`	`Phoneme`	`Chunk`	`POS`	`Scene labeling`
LDS-2	82.54	92.14	79.27	80.57	94.27	96.38	71.56
FB-2	80.73	89.65	78.94	80.48	94.32	96.55	66.78
LDS-4	**85.53**	**94.14**	**79.81**	**81.23**	94.58	96.85	**73.61**
FB-4	82.87	91.75	78.96	81.26	94.56	96.89	68.71

e. Timing results (avg. time per greedy search step in milli seconds)

Algorithms	`HW-Small`	`HW-Large`	`Stress`	`Phoneme`	`Chunk`	`POS`	`Scene labeling`
LDS	40.0	40.0	1.0	23.0	421.0	695.0	2660.0
FB	20.0	19.0	1.0	10.0	134.0	170.0	1740.0
LDS-2	3.0	3.8	0.7	1.0	70.0	65.0	580.0
FB-2	2.0	2.0	0.6	0.7	27.0	17.0	350.0
LDS-4	7.0	7.2	1.3	2.0	160.0	140.0	1350.0
FB-4	3.6	3.6	1.2	1.3	60.0	35.0	790.0

Table 2: Prediction accuracy and timing results for greedy search comparing sparse and complete search spaces.

Results of Cost Function Trained on Sparse Search Spaces. It is natural to expect that performance on sparse search spaces might improve if the cost function is trained using the same sparse search space. Further, since conducting searches in the sparse spaces is computationally cheaper, learning directly in sparse spaces can be much more efficient. Table 2c shows the results of training the cost function on the sparse search spaces using exact imitation. As we can see, accuracies of LDS-2 and FB-2 slightly degrade compared to the corresponding results in Table 2b, but the results of LDS-4 and FB-4 equal or slightly improve in almost all cases except for scene labeling. These results show that we get speed improvements during both training and testing with little loss in accuracy.

Contrary to expectation, the above results show that when using the LDS-2 and FB-2 spaces, training directly on those spaces was often slightly worse than training on the complete spaces. One hypothesis for this observation is that the number and variation of states encountered during training by the exact imitation approach is much less for sparser spaces. This can possibly hurt robustness of the learned cost function. This suggests that a more sophisticated approach such as DAgger might be more effective since it effectively generates a wider diversity of states during training.

Table 2d shows the results of training with DAgger, which confirm the above hypothesis. First, DAgger significantly improves over the results obtained with exact imitation (see Table 2c) across the board. Second, the results of training (via DAgger) and testing on sparse search spaces are better than the results of training on complete search spaces and testing on sparse search spaces (see Table 2b). This agrees with the intuition that training on the search space used for testing should be superior to training on a different search space. Third, the results of LDS-4 and FB-4 with DAgger are significantly better than the results obtained by training and testing on complete search spaces (see Table 2a). This indicates that training on sparse search spaces via DAgger is very effective and gives us speed improvement with no loss in accuracy and sometimes improves accuracy compared to the complete spaces.

Inference Time and Anytime Performance. Table 2e shows the timing results (avg. time per greedy search step in secs.) of our approach during testing using sparse (LDS-k and FB-k) and complete search spaces (LDS and FB). As we can see, we get speed improvement by a factor of ten (roughly) with sparse search spaces. Note that the speedup will generally be larger for problems with larger numbers of labels L, since the number of successors decreases from $T \cdot (L-1)$ to $T \cdot k$. We would like to point out that sparse search spaces will also improve the training time of our approach (fewer ranking examples in step 11 of Algorithm 2), and can be advantageous[13] compared to standard approaches including CRFs and SVM-Struct. Further, we compare the anytime curves of configurations of our approach with sparse and complete search spaces, which show the accuracy achieved by a method versus an inference time bound at prediction time. Figure 7 shows the anytime curves for all the problems except stress prediction, where there is hardly any difference due to the small label set size (5 labels). Note that all these results are for training with DAgger.

From the anytime curves, it is clear that the configurations with sparse search spaces have a much better anytime profile compared to the ones with complete search spaces. LDS-2 and FB-2 reach the respective accuracies of LDS and FB very quickly in all the cases except for POS and Scene labeling, where there is a small loss in accuracy. However, LDS-4 and FB-4 recover the accuracy losses in those two cases. These results demonstrate that sparse search spaces would be highly effective in those situations, where there is a need to make anytime predictions.

Comparing the anytime curves of LDS and FB, we can see that LDS is comparable or better than FB in all cases other than Chunking and POS.[14] This is especially true for the handwriting recognition and scene labeling problems. In the anytime curves for scene labeling task, we can see that LDS is dominant and improves accuracy much more quickly than FB. For example, a 10 second time bound for LDS achieves the same accuracy as FB using 70 seconds. This shows the benefit of using the LDS space. In the case of Chunking and POS, there is almost no difference between the accuracy of the recurrent classifier and

[13] It is hard to do a fair comparison of wall clock times due to differences in implementations.

[14] The experimental setup only differs in the search space (LDS or FB) employed during training and testing.

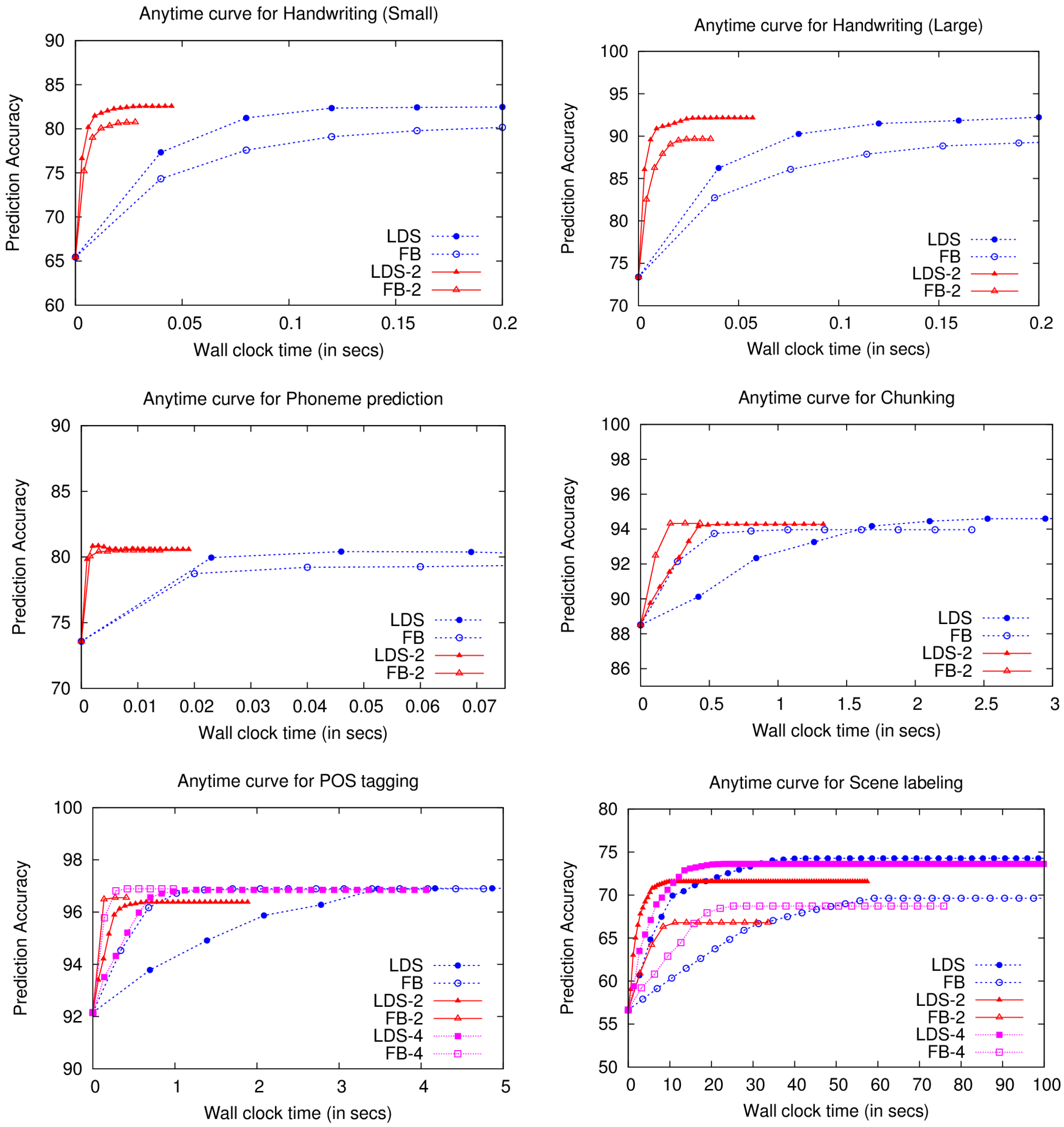


Figure 7: Anytime curves for greedy search comparing sparse and complete search spaces.

exact imitation accuracy (see Table 1a), so LDS does not provide any extra benefit over FB. Recall that there is significant additional overhead for successor generation for LDS compared to flipbit. To generate each successor, LDS must evaluate the recurrent classifier at sequence positions after discrepancies are introduced. On the other hand, the flipbit space need not evaluate the recurrent classifier during successor generation, but only uses the recurrent classifier to generate the initial state. In Chunking and POS, the additional overhead of the LDS search does not payoff in improved accuracy and the anytime curve of flipbit is accordingly better. All these findings are true for the sparser versions of LDS and FB as well.

6. Comparison to Related Work

As described earlier, the majority of structured prediction work has focused on the use of exact (when possible) or approximate inference techniques, such as loopy belief propagation and relaxation methods, for computing outputs. Learning then is focused on tuning the cost function parameters in order to optimize various objective functions, which differ among learning algorithms (Lafferty et al., 2001; Taskar et al., 2003; Tsochantaridis et al., 2004; McAllester et al., 2010). There are also approximate cost function learning approaches that do not employ any inference routine during training. For example, piece-wise training (Sutton and McCallum, 2009), Decomposed Learning (Samdani and Roth, 2012) and its special case pseudo-max training (Sontag et al., 2010) fall under this category. These training approaches are very efficient, but they still need an inference algorithm to make predictions during testing. In these cases, one could employ the Constrained Conditional Models (CCM) framework (Chang et al., 2012) with some declarative (global) constraints to make predictions using the learned cost function. The CCM framework relies on the Integer Linear Programming (ILP) inference method (Roth and tau Yih, 2005). More recent work has attempted to integrate (approximate) inference and cost function learning in a principled manner (Meshi et al., 2010; Stoyanov et al., 2011; Hazan and Urtasun, 2012; Domke, 2013). Researchers have also worked on using higher-order features for CRFs in the context of sequence labeling under the pattern sparsity assumption (Ye et al., 2009; Qian et al., 2009). However, these approaches are not applicable for the graphical models where the sparsity assumption does not hold.

An alternative approach to addressing inference complexity is cascade training (Felzenszwalb and McAllester, 2007; Weiss and Taskar, 2010; Weiss et al., 2010), where efficient inference is achieved by performing multiple runs of inference from a coarse level to a fine level of abstraction. While such approaches have shown good success, they place some restrictions on the form of the cost functions to facilitate "cascading." Another potential drawback of cascades and most other approaches is that they either ignore the loss function of a problem (e.g., by assuming Hamming loss) or require that the loss function be decomposable in a way that supports "loss augmented inference". Our approach is sensitive to the loss function and makes minimal assumptions about it, requiring only that we have a blackbox that can evaluate it for any potential output.

Our approach is partly inspired by the alternative framework of classifier-based structured prediction. These algorithms avoid directly solving the Argmin problem by assuming that structured outputs can be generated by making a series of discrete decisions. The approach then attempts to learn a *recurrent classifier* that given an input $\mathbf{x}$ is iteratively applied in order to generate the series of decisions for producing the target output $\mathbf{y}^*$. Simple training methods (e.g., Dietterich et al., 1995) have shown good success and there are some positive theoretical guarantees (Syed and Schapire, 2010; Ross and Bagnell, 2010). However, recurrent classifiers can be prone to error propagation (Kääriäinen, 2006; Ross and Bagnell, 2010). Recent work, for example, Searn (Hal Daumé III et al., 2009), SMiLe (Ross and Bagnell, 2010), and DAgger (Ross et al., 2011), attempts to address this issue using more sophisticated training techniques and have shown state-of-the-art structured-prediction results. However, all these approaches use classifiers to produce structured outputs through a single sequence of greedy decisions. Unfortunately, in many problems, some decisions are difficult to predict by a greedy classifier, but are crucial for good performance.

In contrast, our approach leverages recurrent classifiers to define good quality search spaces over complete outputs, which allows decision making by comparing multiple complete outputs and choosing the best. There are also non-greedy methods that learn a scoring function to search in the space of partial structured outputs (Hal Daumé III and Marcu, 2005; Daumé III, 2006; Xu et al., 2009; Huang et al., 2012; Yu et al., 2013). All these methods perform online training, and differ only in the way search errors are defined and how the weights are updated when errors occur. Unfortunately, training the scoring function can be difficult because it is hard to evaluate states with partial outputs and the theoretical guarantees of the learned scoring function (e.g., convergence and generalization results) rely on very strong assumptions (Xu et al., 2009).

A closely related framework to ours is the SampleRank framework (Wick et al., 2011) for structured prediction, which also learns a cost function for guiding search in the space of complete outputs. While it shares with our work the idea of explicit search in the output space, there are some significant differences. The SampleRank framework is mainly focused on Monte-Carlo search, and the underlying flipbit search space, whereas our approach can be instantiated for a wide range of search spaces (e.g., LDS space that leverages powerful recurrent classifiers) and rank-based search algorithms (e.g., greedy search, beam search and best-first search). We believe that this flexibility is important since it is well-understood in the search literature that the best search space formulation and the most appropriate search algorithm change from problem to problem. In addition, the SampleRank framework is highly dependent on a hand-designed "proposal distribution" for guiding the search or effectively defining the search space. In contrast, we describe a generic approach for constructing search spaces that is shown to be effective across a variety of domains.

Our approach is also related to Re-Ranking (Collins, 2002), which uses a generative model to propose a k-best list of outputs, which are then ranked by a separate ranking function. In contrast, rather than restricting to a generative model for producing potential outputs, our approach leverages generic search over efficient search spaces guided by a learned cost function that has minimal representational restrictions, and employs the same cost function to rank the candidate outputs. Recent work on generating multiple diverse solutions in a probabilistic framework can be considered as another way of producing candidate outputs. A representative set of approaches in this line of work are diverse M-best (Batra et al., 2012), M-best modes (Park and Ramanan, 2011; Chen et al., 2013) and Determinantal Point Processes (Kulesza and Taskar, 2012).

The general area of local search techniques applied to combinatorial optimization problems is very much relevant to our work. For example, STAGE (Boyan and Moore, 2000) learns an evaluation function over the states to improve the performance of search, where the value of a state corresponds to the performance of a local search algorithm starting from that state, and Zhang and Dietterich (1995) uses Reinforcement Learning (RL) methods to learn heuristics for job shop scheduling with the goal of minimizing the duration of the schedule. For combinatorial optimization problems, the cost function to be optimized is known a priori, where as such a function is not given for structured prediction problems. Therefore, our approach learns a cost function to score the structured outputs.

7. Summary and Future Work

We studied a general framework for structured prediction based on search in the space of complete outputs.We showed how powerful classifiers can be leveraged to define an effective search space over complete outputs, and gave a generic cost function learning approach to score the outputs for any given combination of search space and search strategy. Our experimental results showed that a very small amount of search is needed to improve upon the state-of-the-art performance, validating the effectiveness of our framework. We also addressed some of the scalability issues via a simple pruning strategy that creates sparse search spaces that are more efficient to search in.

In a follow-up work, we introduce a more general framework called $\mathcal{HC}$-Search that employs two distinct functions for guiding the search and scoring the candidate outputs generated during search (Doppa et al., 2013, 2014). $\mathcal{HC}$-Search further improves upon the results in the current paper on most benchmark tasks, and performs comparably or better than Cascades. Future work includes applying this framework to more challenging problems in natural language processing and computer vision (e.g., Coreference resolution, tracking players in sports videos, and object detection in biological images (Lam et al., 2013)), studying principled ways of using domain knowledge (e.g., nearly sound constraints) for pruning, and exploring algorithms to train for anytime performance by trading off speed and accuracy as part of the learning objective (Grubb and Bagnell, 2012; Xu et al., 2012).

Acknowledgments

The authors would like to thank the anonymous reviewers for their feedback. The first author would also like to thank Tom Dietterich for his encouragement and support throughout this work. This work was supported in part by NSF grants IIS 1219258, IIS 1018490 and in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No. FA8750-13-2-0033. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF, the DARPA, the Air Force Research Laboratory (AFRL), or the US government. Some of the material in this article was first published at ICML-2012 (Doppa et al., 2012).

References

Dhruv Batra, Payman Yadollahpour, Abner Guzmán-Rivera, and Gregory Shakhnarovich. Diverse m-best solutions in Markov random fields. In *Proceedings of European Conference on Computer Vision (ECCV)*, pages 1–16, 2012.

Justin A. Boyan and Andrew W. Moore. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research (JMLR)*, 1:77–112, 2000.

Ming-Wei Chang, Lev-Arie Ratinov, and Dan Roth. Structured learning with constrained conditional models. *Machine Learning Journal (MLJ)*, 88(3):399–431, 2012.

Chao Chen, Vladimir Kolmogorov, Yan Zhu, Dimitris Metaxas, and Christoph H. Lampert. Computing the M most probable modes of a graphical model. In *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2013.

Michael Collins. Ranking algorithms for named entity extraction: Boosting and the voted perceptron. In *Proceedings of Association of Computational Linguistics (ACL) Conference*, 2002.

Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research (JMLR)*, 7:551–585, 2006.

Hal Daumé III. *Practical Structured Learning Techniques for Natural Language Processing*. PhD thesis, University of Southern California, Los Angeles, CA, 2006.

Thomas G. Dietterich, Hermann Hild, and Ghulum Bakiri. A comparison of ID3 and backpropagation for English text-to-speech mapping. *Machine Learning Journal (MLJ)*, 18(1):51–80, 1995.

Justin Domke. Structured learning via logistic regression. In *Advances in Neural Information Processing Systems (NIPS)*, pages 647–655, 2013.

Janardhan Rao Doppa, Alan Fern, and Prasad Tadepalli. Output space search for structured prediction. In *Proceedings of International Conference on Machine Learning (ICML)*, 2012.

Janardhan Rao Doppa, Alan Fern, and Prasad Tadepalli. HC-Search: Learning heuristics and cost functions for structured prediction. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2013.

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

Pedro F. Felzenszwalb and David A. McAllester. The generalized A* architecture. *Journal of Artificial Intelligence Research (JAIR)*, 29:153–190, 2007.

Alan Fern, Sung Wook Yoon, and Robert Givan. Approximate policy iteration with a policy language bias: Solving relational Markov decision processes. *Journal of Artificial Intelligence Research (JAIR)*, 25:75–118, 2006.

Alexander Grubb and Drew Bagnell. Speedboost: Anytime prediction with uniform near-optimality. *Journal of Machine Learning Research - Proceedings Track*, 22:458–466, 2012.

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

Hal Daumé III, John Langford, and Daniel Marcu. Search-based structured prediction. *Machine Learning Journal (MLJ)*, 75(3):297–325, 2009.

William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 607–615, 1995.

Tamir Hazan and Raquel Urtasun. Efficient learning of structured predictors in general graphical models. *CoRR*, abs/1210.2346, 2012.

Liang Huang, Suphan Fayong, and Yang Guo. Structured perceptron with inexact search. In *Proceedings of Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics (HLT-NAACL)*, pages 142–151, 2012.

Matti Kääriäinen. Lower bounds for reductions. In *Atomic Learning Workshop*, 2006.

Joseph Keshet, Shai Shalev-Shwartz, Yoram Singer, and Dan Chazan. Phoneme alignment based on discriminative learning. In *Proceedings of Annual Conference of the International Speech Communication Association (Interspeech)*, pages 2961–2964, 2005.

Alex Kulesza and Ben Taskar. Determinantal point processes for machine learning. *Foundations and Trends in Machine Learning*, 5(2-3):123–286, 2012.

John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 282–289, 2001.

Michael Lam, Janardhan Rao Doppa, Xu Hu, Sinisa Todorovic, Thomas Dietterich, Abigail Reft, and Marymegan Daly. Learning to detect basal tubules of nematocysts in SEM images. In *Proceedings of ICCV Workshop on Computer Vision for Accelerated Biosciences (CVAB)*. IEEE, 2013.

David A. McAllester, Tamir Hazan, and Joseph Keshet. Direct loss minimization for structured prediction. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1594–1602, 2010.

Ofer Meshi, David Sontag, Tommi Jaakkola, and Amir Globerson. Learning efficiently with approximate inference via dual losses. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 783–790, 2010.

Dennis Park and Deva Ramanan. N-best maximal decoders for part models. In *Proccedings of IEEE International Conference on Computer Vision (ICCV)*, pages 2627–2634, 2011.

Xian Qian, Xiaoqian Jiang, Qi Zhang, Xuanjing Huang, and Lide Wu. Sparse higher order conditional random fields for improved sequence labeling. In *Proceedings of International Conference on Machine Learning (ICML)*, 2009.

Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. *Journal of Machine Learning Research - Proceedings Track*, 9:661–668, 2010.

Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. *Journal of Machine Learning Research - Proceedings Track*, 15:627–635, 2011.

Dan Roth and Wen tau Yih. Integer linear programming inference for conditional random fields. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 736–743, 2005.

Rajhans Samdani and Dan Roth. Efficient decomposed learning for structured prediction. In *Proceedings of International Conference on Machine Learning (ICML)*, 2012.

Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145–168, 1987.

David Sontag, Ofer Meshi, Tommi Jaakkola, and Amir Globerson. More data means less inference: A pseudo-max approach to structured learning. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2181–2189, 2010.

Veselin Stoyanov, Alexander Ropson, and Jason Eisner. Empirical risk minimization of graphical model parameters given approximate inference, decoding, and model structure. In *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 725–733, 2011.

Charles A. Sutton and Andrew McCallum. Piecewise training for structured prediction. *Machine Learning Journal (MLJ)*, 77(2-3):165–194, 2009.

Umar Syed and Rob Schapire. A reduction from apprenticeship learning to classification. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2253–2261, 2010.

Benjamin Taskar, Carlos Guestrin, and Daphne Koller. Max-margin Markov networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2003.

Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, and Yasemin Altun. Support vector machine learning for interdependent and structured output spaces. In *Proceedings of International Conference on Machine Learning (ICML)*, 2004.

Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun. Large margin methods for structured and interdependent output variables. *Journal of Machine Learning Research (JMLR)*, 6:1453–1484, 2005.

Julia Vogel and Bernt Schiele. Semantic modeling of natural scenes for content-based image retrieval. *International Journal of Computer Vision (IJCV)*, 72(2):133–157, 2007.

David Weiss and Benjamin Taskar. Structured prediction cascades. *Journal of Machine Learning Research - Proceedings Track*, 9:916–923, 2010.

David Weiss, Ben Sapp, and Ben Taskar. Sidestepping intractable inference with structured ensemble cascades. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2415–2423, 2010.

Michael L. Wick, Khashayar Rohanimanesh, Kedar Bellare, Aron Culotta, and Andrew McCallum. Samplerank: Training factor graphs with atomic gradients. In *Proceedings of International Conference on Machine Learning (ICML)*, 2011.

Yuehua Xu, Alan Fern, and Sung Wook Yoon. Learning linear ranking functions for beam search with application to planning. *Journal of Machine Learning Research (JMLR)*, 10: 1571–1610, 2009.

Zhixiang Xu, Kilian Weinberger, and Olivier Chapelle. The greedy miser: Learning under test-time budgets. In *Proceedings of International Conference on Machine Learning (ICML)*, 2012.

Nan Ye, Wee Sun Lee, Hai Leong Chieu, and Dan Wu. Conditional random fields with high-order features for sequence labeling. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2196–2204, 2009.

Heng Yu, Liang Huang, Haitao Mi, and Kai Zhao. Max-violation perceptron and forced decoding for scalable MT training. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1112–1123, 2013.

Wei Zhang and Thomas G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1114–1120, 1995.