Learning Where to Sample in Structured Prediction

Tianlin Shi
Tsinghua University
tianlinshi@gmail.com

Jacob Steinhardt
Stanford University
jsteinhardt@cs.stanford.edu

Percy Liang
Stanford University
pliang@cs.stanford.edu

Abstract

In structured prediction, most inference algorithms allocate a homogeneous amount of computation to all parts of the output, which can be wasteful when different parts vary widely in terms of difficulty. In this paper, we propose a heterogeneous approach that dynamically allocates computation to the different parts. Given a pre-trained model, we tune its inference algorithm (a sampler) to increase test-time throughput. The inference algorithm is parametrized by a meta-model and trained via reinforcement learning, where actions correspond to sampling candidate parts of the output, and rewards are log-likelihood improvements. The meta-model is based on a set of domain-general meta-features capturing the progress of the sampler. We test our approach on five datasets and show that it attains the same accuracy as Gibbs sampling but is 2 to 5 times faster.

1 Introduction

For many structured prediction problems, the output contains many interdependent variables, resulting in exponentially large output spaces. These properties make exact inference intractable for models with high treewidth [Koller et al., 2007], and thus we must rely on approximations such as variational inference and Markov Chain Monte Carlo (MCMC). A key characteristic of many such approximate inference algorithms is that they iteratively modify only a local part of the output structure at a small computational cost. For example, Gibbs sampling [Brooks et al., 2011] updates the output by sampling one variable conditioned on the rest. Often, a large number of local moves is required.

One source of inefficiency in Gibbs sampling is that it dedicates a homogeneous amount of inference to each part of the output. However, in practice, the difficulty and inferential demands of each part is heterogeneous. For example, in rendering computer graphics, paths of light passing through highly reflective or glossy regions deserve more sampling [Veach, 1997]; in named-entity recognition [McCallum and Li, 2003], most tokens clearly do not contain entities and therefore should be allocated less computation. Attempts have been made to capture the nature of heterogeneity in such settings. For example, Elidan et al. [2006] schedule updates in asynchronous belief propagation based on the information residuals of the messages. Chechetka and Guestrin [2010] focus the computation of belief propagation based on the specific variables being queried. Other work has focused on building cascades of coarse-to-fine models, where simple models filter out unnecessary parts of the output and reduce the computational burden for complex models [Viola and Jones, 2001, Weiss and Taskar, 2010].

We propose a framework that constructs heterogeneous sampling algorithms using reinforcement learning (RL). We start with a collection of transition kernels, each of which proposes a modification to part of the output (in this paper, we use transition kernels derived from Gibbs sampling). At each step, our procedure chooses which transition kernel to apply based on cues from the input and the history of proposed outputs. By optimizing this procedure, we fine-tune inference to exploit the specific heterogeneity in the task of interest, thus saving overall computation at test time.

The main challenge is to find signals that consistently provide useful cues (meta-features) for directing the focus of inference across a wide variety of tasks. Moreover, it is important that these signals are cheap to compute relative to the cost of generating a sample, as otherwise the meta-features themselves become the bottleneck of the algorithm. In this paper, we provide

Appearing in Proceedings of the 18^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2015, San Diego, CA, USA. JMLR: W&CP volume 38.

general principles for constructing such meta-features, based on reasoning about *staleness* and *discord* of variables in the output. We cache these ideas out as a collection of five concrete meta-features, which empirically yield good predictions across tasks as diverse as part-of-speech (POS) tagging, named-entity recognition (NER), handwriting recognition, color inpainting, and scene decomposition.

In summary, our contributions are:

- The conceptual idea of learning to sample: we present a learning framework based on RL, and discuss meta-features that leverage heterogeneity.
- The practical value of the framework: given a pre-trained model, we can effectively optimize the test-time throughput of its Gibbs sampler.

2 Heterogeneous Sampling

Before we formalize our framework for heterogeneous sampling, let's consider a motivating example.

$\boldsymbol{x}$	I	think	now	is	the	right	time
pass 1: $\boldsymbol{y}$	PRP	VBP	RB	VBZ	DT	NN	NN
pass 2: $\boldsymbol{y}$	PRP	VBP	RB	VBZ	DT	JJ	NN

Table 1: A POS tagging example. Outputs are recorded after each sweep of Gibbs sampling. Only the ambiguous token "*right*" (NN: noun, JJ: adjective) needs more inference at the second sweep.

Suppose our task is part-of-speech (POS) tagging, where the input $\boldsymbol{x} \in \mathcal{X}$ is a sentence and the output $\boldsymbol{y} \in \mathcal{Y}$ is the sequence of POS tags for the words. An example is shown in Table 1. Suppose that the full model is a chain-structured conditional random field (CRF) with unigram potentials on each tag and bigram potentials between adjacent tags. Exact inference algorithms exist for this model, but for illustrative purposes we use cyclic Gibbs sampling, which samples from the conditional distribution of each tag in cyclic order from left to right.

The example in Table 1 shows at least two sweeps of cyclic Gibbs sampling are required, because it is hard to know whether "*right*" is an adjective or a noun until the tag for the following word "*time*" is sampled. However, the second pass wastes computation by sampling other tags that are mostly determined at the first pass. This inspires the following inference strategy:

pass 1	sample the tags for each word.
pass 2	sample the tag for "*right*".

In general, it is desirable to have the inference algorithm itself figure out which locations to sample, and at a higher level, which test instances to sample.

Algorithm 1 Template for a heterogeneous sampler

1: Initialize $\boldsymbol{y}[0] \sim P_0(\boldsymbol{y})$ for some initializing $P_0(\cdot)$.
2: **for** $t = 1$ to $T_{\mathcal{M}}(\boldsymbol{x})$ **do**
3: **select** transition kernel A_j for some $1 \leqslant j \leqslant m$
4: **sample** $\boldsymbol{y}[t] \sim A_j(\cdot \mid \boldsymbol{y}[t-1])$
5: **end for**
6: **output** $\boldsymbol{y} = \boldsymbol{y}[T]$

2.1 Framework

We now formalize the intuition from the previous example. Assume our pre-trained model specifies a distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$. On a set of test inputs $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)})$, we would like to infer the outputs $\boldsymbol{Y} = (\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n)})$ using some inference algorithm $\mathcal{M}$. To simplify notation, we will focus on a single instance $(\boldsymbol{x}, \boldsymbol{y})$, though our final algorithm considers all test instances jointly. Notice that we can reduce from the multiple-instance case to the single-instance case by just concatenating all the instances into a single instance.

We further assume that a single output $\boldsymbol{y} = (y_1, \ldots, y_m)$ is represented by m variables. For instance, in POS tagging, y_j $(j = 1, \ldots, m)$ is the part-of-speech of the j-th word in the sentence $\boldsymbol{x}$. We are given a collection of transition kernels which target the distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$. For Gibbs sampling, we have the kernels $\{A_j(\boldsymbol{y}' \mid \boldsymbol{y}) : j = 1, \ldots, m\}$, where the transition A_j samples y'_j conditioned on all other variables $\boldsymbol{y}_{\neg j}$, and leaves $\boldsymbol{y}'_{\neg j}$ equal to $\boldsymbol{y}_{\neg j}$.

Algorithm 1 describes the form of samplers we consider. A sampler generates a sequence of outputs $\boldsymbol{y}[1], \boldsymbol{y}[2], \ldots$ by iteratively selecting a variable index $j[t]$ and sampling $\boldsymbol{y}[t+1] \sim A_{j[t]}(\cdot \mid \boldsymbol{y}[t])$. Rather than applying the transition kernels in a fixed order, our samplers select the transition $A_{j[t]}$ to apply based on the input $\boldsymbol{x}$ together with the sampling history. The total number of Markov transitions $T_{\mathcal{M}}(\boldsymbol{x})$ made by $\mathcal{M}$ characterizes its computational cost on input $\boldsymbol{x}$.

How do we choose which transition kernel to apply? A natural objective is to maximize the expected log-likelihood under the model $p(\boldsymbol{y} \mid \boldsymbol{x})$ of the sampler output $\mathcal{M}(\boldsymbol{x})$:

$$\begin{aligned} \max_{\mathcal{M}} \quad & \mathbb{E}_{q_{\mathcal{M}}(\boldsymbol{y} \mid \boldsymbol{x})}[\log p(\boldsymbol{y} \mid \boldsymbol{x})] \\ \text{s.t.:} \quad & T_{\mathcal{M}}(\boldsymbol{x}) \leqslant T^*, \end{aligned} \tag{1}$$

where $q_{\mathcal{M}}(\boldsymbol{y} \mid \boldsymbol{x})$ is the probability that $\mathcal{M}$ outputs $\boldsymbol{y}$ and T^* is the computation budget. Equation (1) says that we want $q_{\mathcal{M}}$ to place as much probability mass as possible on values of $\boldsymbol{y}$ that have high probability

under p, subject to a constraint T^* on the amount of computation. Note that if $T^* = \infty$, the optimal solution would be the posterior mode.

Solving this optimization problem at test time is infeasible, so we will instead optimize $\mathcal{M}$ on a training set, and then deploy the resulting $\mathcal{M}$ at test time.

3 Reinforcement Learning of Heterogeneous Samplers

We would like to optimize the objective in (1), but searching over all samplers $\mathcal{M}$ and evaluating the expectation in (1) are both difficult. We use reinforcement learning (RL) to find an approximate solution.

Reduction to RL. Recall $\boldsymbol{y}[0], \boldsymbol{y}[1], \ldots$ is the sequence of outputs generated by our sampler, where $\boldsymbol{y}[t+1] \sim A_{j[t]}(\cdot \mid \boldsymbol{y}[t])$. To cast our problem into the RL framework, let the state $s_t = (\boldsymbol{y}[0] \ldots, \boldsymbol{y}[t], j[0], \ldots, j[t-1])$ be the entire history of samples, and the action $a_t = A_{j[t]}$ refers to the transition kernel that produces $\boldsymbol{y}[t+1]$ from $\boldsymbol{y}[t]$. We let the reward be the improvement in log-probability:

$$\mathcal{R}(s_t, a_t, s_{t+1}) = \log p(\boldsymbol{y}[t{+}1] \mid \boldsymbol{x}) - \log p(\boldsymbol{y}[t] \mid \boldsymbol{x}). \quad (2)$$

Let $\mathcal{S}$ be the space of states and $\mathcal{A}$ be the space of actions as defined above. The goal of RL is to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ to maximize the expected cumulative reward $\mathbb{E}[R_{T*}]$, where

$$R_T = \sum_{t=0}^{T-1} \mathcal{R}(s_t, a_t, s_{t+1}). \quad (3)$$

Clearly, the total reward R_T is equal to $\log p(\boldsymbol{y}[T] \mid \boldsymbol{x}) - \log p(\boldsymbol{y}[0] \mid \boldsymbol{x})$; since $\boldsymbol{y}[0]$ is independent of the particular policy, maximizing cumulative reward is equivalent to maximizing the original objective in (1). Although reward only depends on the last output $\boldsymbol{y}[t]$ of a state s_t, we will allow the policy to depend on the full history encapsulated in s_t.

Learning algorithm. Our algorithm learns an action-value function $Q(s_t, A_j)$ that predicts the value of using A_j in state s_t. Standard reinforcement learning methods, such as Q-learning [Watkins and Dayan, 1992] and SARSA [Rummery and Niranjan, 1994], do not work well for our purpose. The issue is that they attempt to learn a function $Q(s_t, A_j)$ that models the expected cumulative future reward if we take action A_j in state s_t. In our setting this cumulative reward is hard to predict for the following two reasons:

- It is difficult to estimate how far the current sample is from the global optima. As an approximation, we use $Q(s_t, A_j)$ to predict the cumulative reward over a shorter time horizon $H \ll T^*$.
- The reward over time H also depends on the context of $\boldsymbol{y}_j$. By subtracting the contextual part of the reward, we can hope to isolate the contribution of action A_j. Thus, we use $Q(s_t, A_j)$ to model the *difference* in reward from taking a transition A_j, relative to the baseline of making *no transition*.

Formally, we learn Q using sample backup [Sutton and Barto, 1998, Chapter 9.5]. We start with some state-action sequence $(s_0, a_0, s_1, a_1, \ldots, s_{T*})$ sampled from a fixed exploration policy π'. Then, for each index t $(t = 0, \ldots, T^* - 1)$ in the sequence, we generate a rollout starting from initial state $s_1^c = s_{t+1}$: for $i = 1, 2, \ldots, H$, we generate action $a_i^c = \arg\max_a Q(s_i^c, a)$ and state s_{i+1}^c using a_i^c, and define the utility due to *taking* a_t:

$$U^c = \mathcal{R}(s_t, a_t, s_{t+1}) + \sum_{i=1}^{H} \mathcal{R}(s_i^c, a_i^c, s_{i+1}^c). \quad (4)$$

Next, consider starting from state $s_1^b = s_t$ and *not taking* a_t, and letting $a_i^b = \arg\max_a Q(s_i^b, a)$. The resulting states and actions s_i^b, a_i^b define the following utility:

$$U^b = \sum_{i=1}^{H} \mathcal{R}(s_i^b, a_i^b, s_{i+1}^b). \quad (5)$$

To model the Q-function, we use a single-layer neural network with one hidden node [Tesauro, 1995]:

$$Q(s, a) = w\, \sigma(\alpha \cdot \phi(s, a)) + b, \quad (6)$$

where $\sigma(\cdot)$ is the logistic function, $\phi(s, a) \in \mathbb{R}^L$ are *meta-features* and $\alpha \in \mathbb{R}^L, w \in \mathbb{R}, b \in \mathbb{R}$ are the *meta-parameters*; we write $\boldsymbol{\theta} = (w, b, \alpha)$.

We update $\boldsymbol{\theta}$ with a temporal difference update [Sutton and Barto, 1998] based on our rollout:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\eta}_t * \boldsymbol{d}_t, \quad (7)$$

where $\boldsymbol{\eta}_t \in \mathbb{R}^{L+2}$ are step sizes, " $*$ " is element-wise multiplication and the vector $\boldsymbol{d}_t$ is

$$\boldsymbol{d}_t = \Big(U^c - U^b - Q(s_t, a_t)\Big) \nabla_{\boldsymbol{\theta}} Q(s_t, a_t). \quad (8)$$

For choosing $\boldsymbol{\eta}_t$, we use AdaGrad from Duchi et al. [2010]:

$$\boldsymbol{\eta}_t = \frac{\eta}{\sqrt{\delta + \sum_{i=0}^{t} \boldsymbol{d}_i * \boldsymbol{d}_i}}, \quad (9)$$

where η is the meta learning rate and δ is a smoothing parameter (we use $\eta = 1$ and $\delta = 10^{-4}$ in experiments).

We use the cyclic Gibbs sampler as the fixed exploration policy π' to generate the initial states for the rollouts. The entire training procedure is shown in Algorithm 2:

Algorithm 2 Learning a heterogeneous sampler.

Input Dataset $\boldsymbol{X}$, transition kernels A_j, number of epochs $\mathcal{E}$, cyclic Gibbs policy π', time horizon H.
Initialize $\boldsymbol{y} \sim P_0(\boldsymbol{y})$. Set $s_0 = (\boldsymbol{y}[0])$.
for epoch $= 1, \ldots, \mathcal{E}$ **do**
  for $t = 0, \ldots, T^* - 1$ **do**
    Get action $a_t = \pi'(s_t)$.
    Sample s_{t+1} from a_t.
    Extract $\phi(s_t, a_t)$.
    Estimate gradient using (8)
    Update meta-parameters $\boldsymbol{\theta}$ via (7).
  end for
end for

Test time. At test time, we apply Algorithm 1. In particular, we maintain a priority queue over locations j, where priorities are the Q-values.

- During the **select** step, compute $\arg\max_{A_j} Q(s_t, A_j)$ for state s_t, popping off the maximum element from the priority queue.
- After the **sample** step, re-compute the meta-features of the actions that depend on $j[t]$ and update the corresponding Q-values in the priority queue.

In terms of computational complexity, **select** takes $O(\log m)$, and re-scoring takes $O(Ln_j \log m)$, where m is the number of variables, and n_j is the number of neighbors of variable y_j. The complexity of meta-feature computation will be discussed in Section 4.

4 Meta-Features

The effectiveness of our framework hinges on the existence of good meta-features for the Q-function in (6). Our goal is to develop general meta-features that exhibit strong predictive power across many datasets. In this section, we offer a few guiding principles, which culminates in a set of five meta-feature templates.

Principles. One necessary criterion is that *computing the meta-features should be computationally cheap relative to the rest of the inference algorithm.* Without loss of generality, we assume each output variable y_j takes one of K values and has n_j neighbors. Then such meta-features would take, for example, $O(K)$ or $O(n_j)$ to compute, and they should not be computed more often than being sampled.

In order to satisfy this criterion, an idea that we have found consistently useful is *stale values*. For example, suppose we would like to use the entropy of $p(y_j \mid \boldsymbol{y}_{\neg j})$ as a meta-feature. Computing it would be just as expensive as sampling from A_j. Instead, we keep track of a *stale* version of conditional entropy. Every time we sample from A_j, we compute the entropy of y_j according to the sampling distribution and store it in memory. Then the *stale conditional entropy* of y_j is defined as the current entropy in memory. As the stale conditional entropy is a meta-feature, we leave it up to the learning algorithm to determine how much to trust it.

Reasoning about *staleness* is valuable in another way as well: we want to know how different the Markov blanket of y_j is relative to the last time it was sampled. If it is very different, this tells us two things: first, any stored quantities such as entropy that we have computed are probably out of date. More importantly, the conditional distribution $p(y_j \mid \boldsymbol{y}_{\neg j})$ is probably very different than when we last sampled from A_j, so it is probably a good idea to sample from A_j again.

In addition to staleness, another important notion is *discord*. If at least one neighbor of y_j is inconsistent with the current value of y_j, then it is probably worth sampling from A_j.

Templates. Based on the ideas of staleness and discord, we introduce the following meta-feature templates:

vary: *the number of variables in the Markov blanket of y_j that have changed since the last time we sampled from A_j.* The more neighbors of y_j that have changed, the more stale y_j could be. This meta-feature is computed as follows: upon sampling y_j, we set $\textbf{vary}(y_j) = 0$, and increment **vary** of its neighbors by 1. So the computational complexity of **vary** is only $O(n_j)$.

nb: *discord with neighbors.* We define meta-features **nb**-y-y' ($y, y' \in \{1, \ldots, K\}$) which are binary indicators to track the occurrences of each of the possible value pairs (y, y') for y_j and one of its neighbors. The intuition is that certain pairs of values are very unlikely to occur as part of a legitimate output, and thus represent a discordance that requires taking more samples; the **nb** meta-features allow us to learn these pairs from data. Although the total number of **nb** meta-features is K^2, the computational complexity is still $O(n_j)$ due to sparsity.

cond-ent: *the conditional entropy of y_j at the last*

time it was sampled. Variables with high conditional entropy have a high degree of uncertainty and can benefit from being sampled further.

However, as noted earlier, keeping track of an up-to-date conditional entropy would be too computationally expensive. We therefore use a stale version based on the last time that the variable y_j in question was sampled by A_j. The computational overhead of `cond-ent` is $O(K)$, since we can just compute the entropy based on $p(y_j \mid \boldsymbol{y}_{\neg j})$, which already needs to be computed in order to sample from A_j.

unigram-ent: Sometimes, we can fit a simpler unigram model to the training dataset, where all the variables y_j are independent (given the input). In such cases, unigram entropy of $p(y_j)$ can be used as an over-estimate of the degree of ambiguity for a given variable [Bishop, 2006]. If the unigram entropy is very low, the variable is probably not worth sampling. To capture this, we use the indicator function for the unigram entropy being below some threshold (10^{-4} in our experiments).

sp: *number of times y_j has been sampled thus far.* A potential problem with all of the above meta-features is that they might overly explore possibilities for the same variable. So we need some way to reason about the fact that at some point sampling the same variable more is unlikely to lead to improvements. We do this by keeping track of the number of times a variable has already been sampled: once a variable has been sampled too many times, it is unlikely that sampling it further will be fruitful.

5 Experiments

In this section, we provide empirical evaluation of our method, which we call `HeteroSampler`.

5.1 Datasets

To evaluated our method on five tasks: part-of-speech tagging (POS), named-entity recognition (NER), handwriting recognition, color inpainting, and scene decomposition.

The general setup is as follows. First, we use RL to learn the parameters of the meta-model on the training dataset. We then run Algorithm 1 on the test set. Unless otherwise stated, in all experiments we use $\mathcal{E} = 3$ training epochs, step size $\eta = 1$, and time horizon $H = 1$. In addition to using cyclic Gibbs to generate a base policy for training, we also compare to cyclic Gibbs at test time.

We evaluated on the following datasets:

Words	Japan	coach	Shu	Kamo	said	:	'	'	The	Syrian	own	goal	proved	lucky	for	us
Truth	B-LOC	O	B-PER	I-PER	O	O	O	O	O	B-MISC	O	O	O	O	O	O
1	I-ORG	O	B-PER	I-PER	O	O	O	O	O	B-LOC	O	O	O	O	O	O
2	B-LOC	O	B-PER	I-PER	O	O	O	O	O	B-MISC	O	O	O	O	O	O
3	B-LOC	O	B-PER	I-PER	O	O	O	O	O	B-MISC	O	O	O	O	O	O

(a) Cyclic Gibbs sampler on NER-f4

Words	Japan	coach	Shu	Kamo	said	:	'	'	The	Syrian	own	goal	proved	lucky	for	us
Truth	B-LOC	O	B-PER	I-PER	O	O	O	O	O	B-MISC	O	O	O	O	O	O
1	I-ORG	O	B-PER	I-PER	O	O	O	O	O	B-LOC	O	O	O	O	O	O
2	B-LOC	O	B-PER	I-PER	O	O	O	O	O	B-MISC	O	O	O	O	O	O
3	B-LOC	O	B-PER	I-PER	O	O	O	O	O	B-MISC	O	O	O	O	O	O

(b) `HeteroSampler` on NER-f4

Figure 1: Visualization of computational resource allocation on a test example from NER-f4. Each row is a snapshot of the sample after $k = 1, 2, 3$ sweeps. A darker color means that more cumulative samples have been taken. For `HeteroSampler`, one sweep corresponds to making m transitions, where m is the number of variables. `HeteroSampler` has learned to sample harder parts of the instance, such as ambiguous tokens.

POS/NER Tagging. The POS tagging dataset comes from the standard Wall Street Journal (WSJ) section of the Penn Treebank, and the NER tagging dataset is taken from the 2003 CoNLL Shared Task. We trained an CRF model with features between each token and its corresponding tag (i.e. features $[g(x_i), y_i]$ with feature extractor $g(\cdot)$), and higher-order features between/among tags (i.e. $[y_i, y_{i+1}, ..., y_{i+q}]$) [Liang et al., 2008]. We refer to q as the *factor size*, and call the corresponding tasks POS-fq and NER-fq. The feature extraction functions $g(\cdot)$ include prefixes and suffixes of the word (up to length 4), lowercase word, word signature (e.g. *McDiarmid* into *AA* and *AaAaaaaaa*, *banana* into *a* and *aaaaaa*) and an indicator of the word's being capitalized. The CRF is trained using AdaGrad [Duchi et al., 2010] with 5 passes over the training set, using Gibbs sampling for inference with 8 total sweeps over each instance and 5 sweeps as burn-in. The performance is evaluated via tag accuracy for POS and F1 score for NER.

Handwriting Recognition. We use the handwriting recognition dataset from Weiss and Taskar [2010]. The data were originally collected by Kassel [1995], and contain 6877 handwritten words from 150 subjects with 55 distinct words. Each instance of the dataset is a word, which is a sequence of characters. Associated with each output character is a corresponding 16×8 input binary optical image. The dataset is split into training and test set, where the training set had 6251 words and the test set had 626 words.

Similar to POS/NER tagging, the baseline algorithm is an CRF. We have a feature for each (pixel value, location, character) triple, as well as higher-order n-gram potentials between consecutive charac-

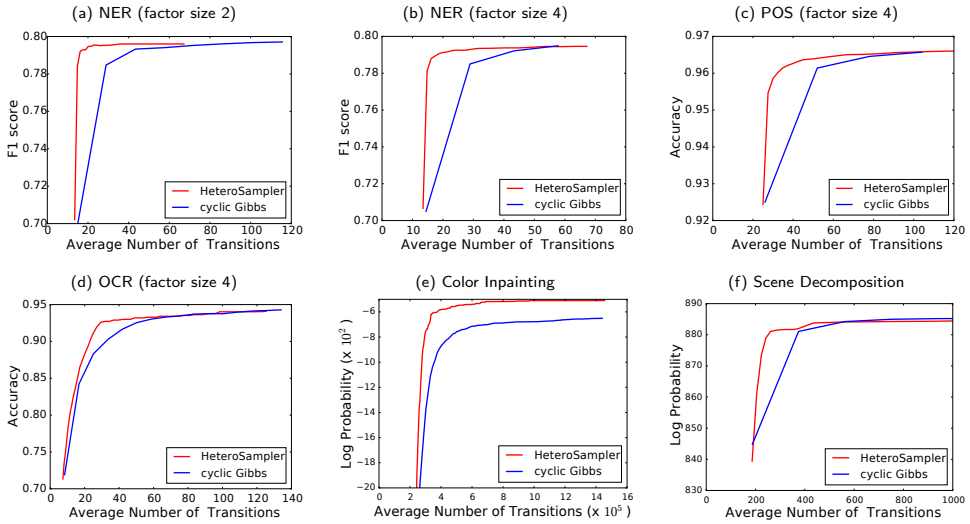


Figure 2: Accuracy vs. number of transitions across several tasks. `HeteroSampler` converges much faster than cyclic Gibbs sampling. On the color inpainting task, dynamically choosing the order of sampling also allows the algorithm to find a better local optimum.

ters. The training scheme of the full model is similar to POS/NER, except that we use 16 Gibbs sampling sweeps in total with 5 burn-in sweeps. The results are evaluated via character-wise accuracy.

Color Inpainting. The three-class color inpainting task is borrowed from Chambolle et al. [2012]. The input is a corrupted color image in a circular domain, and the target image is an equipartition of the circle using three colors. We use a pre-trained model from the OpenGM benchmark [Kappes et al., 2013]. The baseline is Gibbs sampling with 100 sweeps over the instance. There are two instances in this dataset and we use one to train the `HeteroSampler` and the other to test it. Performance is evaluated based on the log-probability of the output.

Scene Decomposition. The scene decomposition dataset is obtained from the source in Gould et al. [2009]. The goal is to segment a natural image into eight semantic categories, such as grass and sky. We use the subset of 715 images included in the OpenGM toolkit [Kappes et al., 2013], for which an existing graphical model is publicly available. The graphical model is a superpixel factor graph, and each superpixel has 773 feature dimensions. Among the 715 images, we randomly pick 358 instances for training, and the rest are used for testing. The baseline Gibbs sampling uses 16 sweeps over each image, and we train the `HeteroSampler` in the same way as Color Inpainting. Performance is again evaluated via log-probability.

5.2 Visualization of Resource Allocation

Figure 1 visualizes the allocation of sampling operations on an NER instance. While cyclic Gibbs sampling uniformly distributes its computational resources, `HeteroSampler` is able to focus on harder parts of the task such as names.

5.3 Performance under Different Budgets

To measure the performance under different budgets, we gradually increase the total number of transitions at test time for the `HeteroSampler` and the overall number of sweeps for the Gibbs baseline. The number of transitions for training are held fixed.

Figure 2 plots the performance versus average number of transitions per instance. As we see, given the same budget, `HeteroSampler` achieves equal or better performance for all tasks. In addition, `HeteroSampler` reaches the ceiling accuracy 2 to 5 times faster regardless of the problem domain. For the Color Inpainting problem, which is the most challenging of the tasks, `HeteroSampler` also achieves better end performance when it converges. This is due to the fact that, by optimizing the order of sampling, the meta-algorithm is able to find a better local optimum.

Next, we justify measuring computational cost in terms of number of samples. To do this, we measured the overhead of computing the policy relative to the cost of sampling. As long as this overhead is low, computing samples is the bottleneck in the base algorithm, and so number of samples is a reasonable measure of computational cost. Figure 3 shows how many wall-clock seconds were spent computing the `HeteroSampler` policy for each dataset. For most tasks, sampling involves computing all of the features of the full model, while computing the policy uses only a few meta-features and therefore has negligible cost. The exception is color inpainting, where the full model has only a few features.

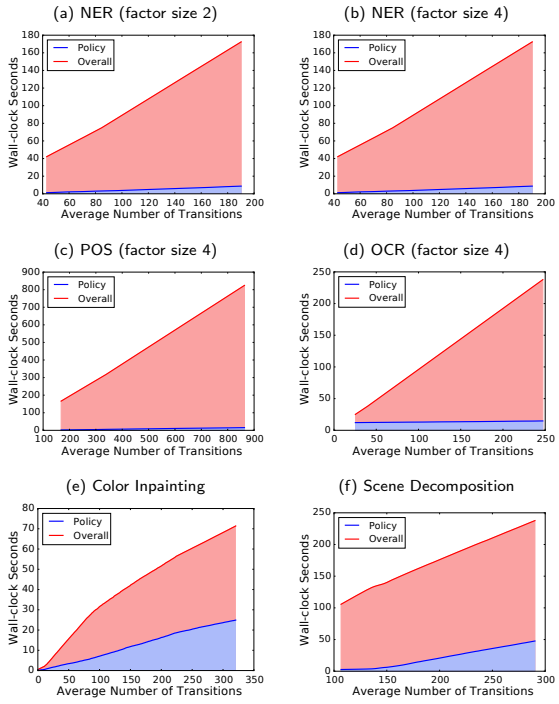
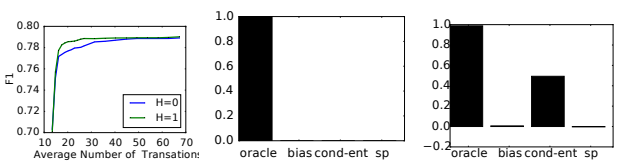


Figure 3: Wall-clock time vs. average number of transitions across different datasets. This measures the overhead of policy evaluation. The red line "sampling" shows the time spent without policy evaluation, while the blue line "policy" shows the time spent on policy evaluation.

5.4 Cumulative Rewards

We would like to verify two facts: first, training with cumulative rewards is useful relative to just using immediate reward; second, our meta-features can predict the cumulative rewards. First, Figure 4(a) shows accuracy vs. number of transitions with $H = 0$ (immediate rewards) and $H = 1$ (cumulative rewards) on `NER-f4`. As we can see, with cumulative rewards, `HeteroSampler` performs better. Table 2 provides an intuitive explanation.

To see which meta-features are contributing to the prediction of cumulative rewards, we intentionally add an `oracle` meta-feature, which is the immediate reward of sampling. Figures 4(b) and (c) visualize the weights of some meta-features for $H = 0$ and $H = 1$ respectively. As expected, when learned with immediate rewards, all weights concentrates on the `oracle` meta-feature. The learned meta-model does not encourage exploration and therefore may omit positions that have cumulative reward. When trained with cumulative rewards, the meta-model also distributes some weight to `cond-ent`, which leads to better exploration and better performance.



(a) (b) (c)

Figure 4: Effect of training with cumulative rewards. (a) Convergence curve of the inference algorithm trained with one-step look-ahead ($H = 1$) vs. immediate rewards ($H = 0$). (b) Weights of meta-features when trained with $H = 0$ plus the oracle meta-feature. (c) Weights of the meta-features when trained with $H = 1$ plus the oracle meta-feature.

input	KANSAS	CITY	AT	OAKLAND
immediate	`B-LOC`	`I-LOC`	`O`	`B-LOC`
cumulative	`B-ORG`	`I-ORG`	`O`	`B-LOC`

Table 2: Cumulative rewards are often helpful when there is high correlation between variables. In this example, "*KANSAS CITY*" is initially labeled as an location. Two coordinated actions are needed to change it to an organization and improve log-likelihood. A meta-model trained with immediate rewards would not recognize the value of sampling "*KANSAS*" alone.

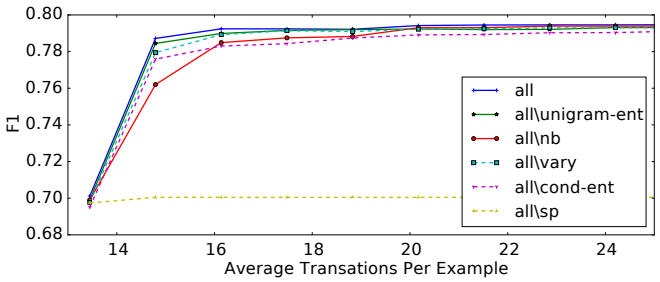
5.5 Meta-feature Ablation Analysis

To evaluate the contribute of individual meta-features and to understand their role in predicting reward, we did a meta-feature ablation analysis. With one meta-feature removed at a time, we run the meta-algorithm and produce a convergence curve, shown in Figure 5 for NER-f2 and POS-f4. All meta-features play an important role. `sp` is the most important; without it, we would repeatedly sample variables with high uncertainty.

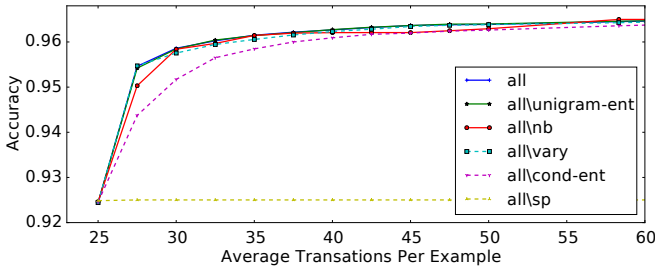
6 Related Work and Discussion

At a high-level, our approach is about fine-tuning the inference algorithm of a pre-trained model to make more effective use of computational resources at test time. Specifically, we use reinforcement learning to train a sampler that operates on variables heterogeneously based on the promise of likelihood improvements. We demonstrated substantial speed improvements on several structured prediction tasks.

The idea of treating structured prediction as a sequential decision-making problem has been explored by SEARN [Daume et al., 2009] and DAGGER [Ross et al., 2011a]. Both train a multiclass classifier to build up a structured output incrementally in a fixed order. Similar ideas have been applied in dependency parsing [Goldberg and Nivre, 2013].



(a) Meta-feature ablation on NER (factor size is 2)



(b) Meta-feature ablation on POS (factor size is 4)

Figure 5: Meta-feature ablation study on various datasets. (a) shows the convergence curves of `HeteroSampler` with one meta-feature removed on (a) NER-f2 and (b) POS-f4. F is the entire meta-feature set, and "\" denotes excluding a meta-feature. We see that each meta-feature matters for at least some of the tasks, and `sp` is the most important.

To obtain speedups, it is beneficial to learn the order in which the structured output is constructed; this flexibility is the cornerstone of our work. For example, Goldberg and Elhadad [2010] proposed an approach that learns to construct a dependency tree by adding "easy" arcs first. More generally, Jiang et al. [2012] maintains a priority queue over partially constructed hypotheses for constituency parsing and learns to choose which one to process first. While the aforementioned work builds up outputs incrementally, our heterogeneous sampler makes modifications to full outputs, which can be more flexible.

Other work also operate in the space of full outputs. For example, Doppa et al. [2014b,a] perform several steps of local search around a baseline prediction. Zhang et al. [2014] performed greedy hill-climbing from multiple random starting points for dependency parsing. Ross et al. [2011b] used DAGGER to learn message-passing inference algorithms. However, unlike our method, none of these papers deal with the issue of determining which locations are useful to operate on *without* explicitly evaluating the model score for each candidate modification. We use lightweight meta-features for this purpose.

Some methods use a fixed strategy to prioritize inference in a fixed model. For example, residual belief propagation [Elidan et al., 2006] selects the message between two variables that has changed the most from the previous iteration. In cases where we are interested in a particular query variable, Chechetka and Guestrin [2010] prioritizes messages based on importance to the query. Wick and McCallum [2011] implements the same intuition in the context of MCMC.

SampleRank [Wick et al., 2011] also performs learning in the context of sampling, but is complementary to our work: SampleRank fixes a sampling strategy and trains the underlying model, whereas we fix the underlying model and train the sampling strategy using domain-general meta-features. Wick et al. [2009] tackles the local optima problem in structured prediction by using RL to train policies that could select fruitful downward jumps, which is the same issue that our use of cumulative rewards attempts to address.

More generally, the goal of speeding up inference at test time is quite established by now. Viola and Jones [2001] used a sequence of models from simple to complex for face detection, at each successive stage pruning out unlikely locations in the image. Weiss and Taskar [2010] trained a sequence of Markov models of increasing order, each successive stage pruning out unlikely local configurations.

As feature extraction is often the performance bottleneck, it is a promising place to look for speed improvements. Weiss and Taskar [2013] used RL to train policies that adaptively determine the value of information of each feature at test time. For dependency parsing, He et al. [2013] considers a sequence of increasingly complex features, and uses DAGGER to learn which arcs to commit to before adding more features.

Our work is superficially related to the work on adaptive MCMC [Andrieu and Thoms, 2008], but the goals are quite different. Adaptive MCMC samplers attempt to preserve the stationary distribution, while our approach seeks to directly maximize log-likelihood within a fixed number of time steps.

As a final remark, in this paper, we only focused on the issue of "where to sample", but the general framework, which merely learns which transition kernels to apply, could also be applied to determine "how to sample" too by supplying a richer family of transition kernels—for example, ones based on models with different feature sets or blocked samplers. This opens up a vast set of possibilities for finer-grained adaptivity.

Acknowledgements. This work was supported by the Fannie & John Hertz Foundation for the second author and the Microsoft Research Faculty Fellowship for the third author.

References

C. Andrieu and J. Thoms. A tutorial on adaptive MCMC. *Statistics and Computing*, 18(4):343–373, 2008.

C. M. Bishop. *Pattern recognition and machine learning*. Springer New York, 2006.

S. Brooks, A. Gelman, G. Jones, and X. Meng. *Handbook of Markov Chain Monte Carlo*. CRC Press, 2011.

Antonin Chambolle, Daniel Cremers, and Thomas Pock. A convex approach to minimal partitions. *SIAM Journal on Imaging Sciences*, 5(4):1113–1158, 2012.

A. Chechetka and C. Guestrin. Focused belief propagation for query-specific inference. In *Artificial Intelligence and Statistics (AISTATS)*, 2010.

H. Daume, J. Langford, and D. Marcu. Search-based structured prediction. *Machine Learning*, 75:297–325, 2009.

J.R. Doppa, A. Fern, and P. Tadepalli. Hc-search: A learning framework for search-based structured prediction. *Journal of Artificial Intelligence Research*, 50:403–439, 2014a.

J.R. Doppa, A. Fern, and P. Tadepalli. Structured prediction via output space search. *Journal of Machine Learning Research*, 15:1317–1350, 2014b.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. In *Conference on Learning Theory (COLT)*, 2010.

G. Elidan, I. McGraw, and D. Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *Uncertainty in Artificial Intelligence (UAI)*, 2006.

Y. Goldberg and M. Elhadad. An efficient algorithm for easy-first non-directional dependency parsing. In *Association for Computational Linguistics (ACL)*, pages 742–750, 2010.

Y. Goldberg and J. Nivre. Training deterministic parsers with non-deterministic oracles. *Transactions of the Association for Computational Linguistics (TACL)*, 1, 2013.

S. Gould, R. Fulton, and D. Koller. Decomposing a scene into geometric and semantically consistent regions. In *ICCV*, 2009.

H. He, H. Daume, and J. Eisner. Dynamic feature selection for dependency parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

J. Jiang, A. Teichert, J. Eisner, and H. Daume. Learned prioritization for trading off accuracy and speed. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

J. H. Kappes, B. Andres, F. A. Hamprecht, C. Schnorr, S. Nowozin, D. Batra, S. Kim, B. X. Kausler, J. Lellmann, N. Komodakis, and C. Rother. A comparative study of modern inference techniques for discrete energy minimization problems. In *Computer Vision and Pattern Recognition (CVPR)*, 2013.

R.H. Kassel. *A comparison of approaches to on-line handwritten character recognition*. PhD thesis, Massachusetts Institute of Technology, 1995.

D. Koller, N. Friedman, L. Getoor, and B. Taskar. Graphical models in a nutshell. *Statistical Relational Learning*, page 13, 2007.

P. Liang, H. Daume, and D. Klein. Structure compilation: Trading structure for features. In *International Conference on Machine Learning (ICML)*, 2008.

A. McCallum and W. Li. for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 188–191. Association for Computational Linguistics, 2003.

S. Ross, G. Gordon, and A. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Artificial Intelligence and Statistics (AISTATS)*, 2011a.

S. Ross, D. Munoz, M. Hebert, and J. A. Bagnell. Learning message-passing inference machines for structured prediction. In *Computer Vision and Pattern Recognition (CVPR)*, pages 2737–2744, 2011b.

G.A. Rummery and M. Niranjan. *Online Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

R.S. Sutton and A.G. Barto. *Introduction to reinforcement learning*. MIT Press, 1998.

G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

E. Veach. *Robust Monte Carlo methods for light transport simulation*. PhD thesis, Stanford University, 1997.

P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition (CVPR)*, 2001.

C. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

D. Weiss and B. Taskar. Structured prediction cascades. In *Artificial Intelligence and Statistics (AISTATS)*, 2010.

D. Weiss and B. Taskar. Learning adaptive value of information for structured prediction. In *Advances in Neural Information Processing Systems (NIPS)*, 2013.

M. Wick, K. Rohanimanesh, S. Singh, and a. A. McCallum. Training factor graphs with reinforcement learning for efficient map inference. In *Advances in Neural Information Processing Systems (NIPS)*, 2009.

M. Wick, K. Rohanimanesh, and K. Bellare. Samplerank: Training factor graphs with atomic gradients. In *International Conference on Machine Learning (ICML)*, 2011.

M. L. Wick and A. McCallum. Query-aware MCMC. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2564–2572, 2011.

Y. Zhang, T. Lei, R. Barzilay, and T. Jaakkola. Greed is good if randomized: New inference for dependency parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2014.