Neural Particle Smoothing for Sampling from Conditional Sequence Models

Chu-Cheng Lin and **Jason Eisner**
Center for Language and Speech Processing
Johns Hopkins University, Baltimore MD, 21218
{kitsing,jason}@cs.jhu.edu

Abstract

We introduce *neural particle smoothing*, a sequential Monte Carlo method for sampling annotations of an input string from a given probability model. In contrast to conventional particle filtering algorithms, we train a proposal distribution that *looks ahead* to the end of the input string by means of a right-to-left LSTM. We demonstrate that this innovation can improve the quality of the sample. To motivate our formal choices, we explain how our neural model and neural sampler can be viewed as low-dimensional but nonlinear approximations to working with HMMs over very large state spaces.

1 Introduction

Many structured prediction problems in NLP can be reduced to labeling a length-T input string $\mathbf{x}$ with a length-T sequence $\mathbf{y}$ of tags. In some cases, these tags are annotations such as syntactic parts of speech. In other cases, they represent actions that incrementally build an output structure: IOB tags build a chunking of the input (Ramshaw and Marcus, 1999), shift-reduce actions build a tree (Yamada and Matsumoto, 2003), and finite-state transducer arcs build an output string (Pereira and Riley, 1997).

One may wish to score the possible taggings using a recurrent neural network, which can learn to be sensitive to complex patterns in the training data. A globally normalized conditional probability model is particularly valuable because it quantifies uncertainty and does not suffer from label bias (Lafferty et al., 2001); also, such models often arise as the predictive conditional distribution $p(\mathbf{y} \mid \mathbf{x})$ corresponding to some well-designed generative model $p(\mathbf{x}, \mathbf{y})$ for the domain. In the neural case, however, inference in such models becomes intractable. It is hard to know what the model actually predicts and hard to compute gradients to improve its predictions.

In such intractable settings, one generally falls back on approximate inference or sampling. In the NLP community, beam search and importance sampling are common. Unfortunately, beam search considers only the approximate-top-k taggings from an exponential set (Wiseman and Rush, 2016), and importance sampling requires the construction of a good proposal distribution (Dyer et al., 2016).

In this paper we exploit the sequential structure of the tagging problem to do *sequential* importance sampling, which resembles beam search in that it constructs its proposed samples incrementally—one tag at a time, taking the actual model into account at every step. This method is known as particle filtering (Doucet and Johansen, 2009). We extend it here to take advantage of the fact that the sampler has access to the entire input string as it constructs its tagging, which allows it to look ahead or—as we will show—to use a neural network to approximate the effect of lookahead. Our resulting method is called *neural particle smoothing*.

1.1 What this paper provides

For $\mathbf{x} = x_1 \cdots x_T$, let $\mathbf{x}_{:t}$ and $\mathbf{x}_{t:}$ respectively denote the prefix $x_1 \cdots x_t$ and the suffix $x_{t+1} \cdots x_T$.

We develop *neural particle smoothing*—a sequential importance sampling method which, given a string $\mathbf{x}$, draws a sample of taggings $\mathbf{y}$ from $p_\theta(\mathbf{y} \mid \mathbf{x})$. Our method works for any conditional probability model of the quite general form[1]

$$p_\theta(\mathbf{y} \mid \mathbf{x}) \stackrel{\text{def}}{\propto} \exp G_T \tag{1}$$

where G is an *incremental stateful global scoring model* that recursively defines scores G_t of prefixes of $(\mathbf{x}, \mathbf{y})$ at all times $0 \le t \le T$:

$$G_t \stackrel{\text{def}}{=} G_{t-1} + g_\theta(\mathbf{s}_{t-1}, x_t, y_t) \quad \text{(with } G_0 \stackrel{\text{def}}{=} 0\text{)} \tag{2}$$

$$\mathbf{s}_t \stackrel{\text{def}}{=} f_\theta(\mathbf{s}_{t-1}, x_t, y_t) \quad \text{(with } \mathbf{s}_0 \text{ given)} \tag{3}$$

These quantities implicitly depend on $\mathbf{x}, \mathbf{y}$ and θ. Here $\mathbf{s}_t$ is the model's *state* after observing the pair of length-t prefixes $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$. G_t is the *score-so-far*

[1] A model may require for convenience that each input end with a special end-of-sequence symbol: that is, $x_T = \text{EOS}$.

of this prefix pair, while $G_T - G_t$ is the *score-to-go*. The state $\mathbf{s}_t$ summarizes the prefix pair in the sense that the score-to-go depends only on $\mathbf{s}_t$ and the length-$(T-t)$ suffixes $(\mathbf{x}_{t:}, \mathbf{y}_{t:})$. The *local scoring function* g_θ and *state update function* f_θ may be any functions parameterized by θ—perhaps neural networks. We assume θ is fixed and given.

This model family is expressive enough to capture any desired $p(\mathbf{y} \mid \mathbf{x})$. Why? Take any distribution $p(\mathbf{x}, \mathbf{y})$ with this desired conditionalization (e.g., the true joint distribution) and factor it as

$$\begin{aligned}\log p(\mathbf{x}, \mathbf{y}) &= \textstyle\sum_{t=1}^{T} \log p(x_t, y_t \mid \mathbf{x}_{:t-1}, \mathbf{y}_{:t-1}) \\ &= \textstyle\sum_{t=1}^{T} \underbrace{\log p(x_t, y_t \mid \mathbf{s}_{t-1})}_{\text{use as } g_\theta(\mathbf{s}_{t-1}, x_t, y_t)} = G_T \quad (4)\end{aligned}$$

by making $\mathbf{s}_t$ include as much information about $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$ as needed for (4) to hold (possibly $\mathbf{s}_t = (\mathbf{x}_{:t}, \mathbf{y}_{:t})$).[2] Then by defining g_θ as shown in (4), we get $p(\mathbf{x}, \mathbf{y}) = \exp G_T$ and thus (1) holds for each $\mathbf{x}$.

1.2 Relationship to particle filtering

Our method is spelled out in §4 (one may look now). It is a variant of the popular *particle filtering* method that tracks the state of a physical system in discrete time (Ristic et al., 2004). Our particular *proposal distribution* for y_t can be found in equations (5), (6), (25) and (26). It considers not only past observations $\mathbf{x}_{:t}$ as reflected in $\mathbf{s}_{t-1}$, but also future observations $\mathbf{x}_{t:}$, as summarized by the state $\bar{\mathbf{s}}_t$ of a right-to-left recurrent neural network $\bar{f}$ that we will train:

$$\hat{H}_t \stackrel{\text{def}}{=} h_\phi(\bar{\mathbf{s}}_{t+1}, x_{t+1}) + \hat{H}_{t+1} \quad (5)$$

$$\bar{\mathbf{s}}_t \stackrel{\text{def}}{=} \bar{f}_\phi(\bar{\mathbf{s}}_{t+1}, x_{t+1}) \quad \text{(with } \mathbf{s}_T \text{ given)} \quad (6)$$

Conditioning the distribution of y_t on future observations $\mathbf{x}_{t:}$ means that we are doing "smoothing" rather than "filtering" (in signal processing terminology). Doing so can reduce the bias and variance of our sampler. It is possible so long as $\mathbf{x}$ is provided in its entirety before the sampler runs—which is often the case in NLP.

1.3 Applications

Why sample from p_θ at all? Many NLP systems instead simply search for the *Viterbi sequence* $\mathbf{y}$ that maximizes G_T and thus maximizes $p_\theta(\mathbf{y} \mid \mathbf{x})$. If the space of states $\mathbf{s}$ is small, this can be done efficiently by dynamic programming (Viterbi, 1967); if not, then A^* may be an option (see §2). More common is to use an approximate method: beam search, or perhaps a sequential prediction policy trained with reinforcement learning. Past work has already shown how to improve these approximate search algorithms by conditioning on the future (Bahdanau et al., 2017; Wiseman and Rush, 2016).

Sampling is essentially a generalization of maximization: sampling from $\exp \frac{G_T}{\text{temperature}}$ approaches maximization as $\text{temperature} \to 0$. It is a fundamental building block for other algorithms, as it can be used to take expectations over the whole space of possible $\mathbf{y}$ values. For unfamiliar readers, Appendix E reviews how sampling is crucially used in minimum-risk decoding, supervised training, unsupervised training, imputation of missing data, pipeline decoding, and inference in graphical models.

2 Exact Sequential Sampling

To develop our method, it is useful to first consider exact samplers. Exact sampling is tractable for only some of the models allowed by §1.1. However, the form and notation of the exact algorithms in §2 will guide our development of approximations in §3.

An *exact sequential sampler* draws y_t from $p_\theta(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1})$ for each $t = 1, \ldots, T$ in sequence. Then $\mathbf{y}$ is exactly distributed as $p_\theta(\mathbf{y} \mid \mathbf{x})$.

For each given $\mathbf{x}, \mathbf{y}_{:t-1}$, observe that

$$\begin{aligned}&p_\theta(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1}) && (7)\\ &\propto p_\theta(\mathbf{y}_{:t} \mid \mathbf{x}) = \textstyle\sum_{\mathbf{y}_{t:}} p_\theta(\mathbf{y} \mid \mathbf{x}) && (8)\\ &\propto \textstyle\sum_{\mathbf{y}_{t:}} \exp G_T && (9)\\ &= \exp\,(G_t + \underbrace{\log \textstyle\sum_{\mathbf{y}_{t:}} \exp\,(G_T - G_t)}_{\text{call this } H_t}) && (10)\\ &= \exp\,(G_{t-1} + g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + H_t) && (11)\\ &\propto \exp\,(g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + H_t) && (12)\end{aligned}$$

Thus, we can easily construct the needed distribution (7) by normalizing (12) over all possible values of y_t. The challenging part of (12) is to compute H_t: as defined in (10), H_t involves a sum over exponentially many futures $\mathbf{y}_{t:}$. (See Figure 1.)

We chose the symbols G and H in homage to the A^* search algorithm (Hart et al., 1968). In that algorithm (which could be used to find the Viterbi sequence), g denotes the score-so-far of a partial solution $\mathbf{y}_{:t}$, and h denotes the optimal score-to-go. Thus, $g + h$ would be the score of the *best* sequence with prefix $\mathbf{y}_{:t}$. Analogously, our $G_t +$

[2] Furthermore, $\mathbf{s}_t$ could even depend on all of $\mathbf{x}$ (if $\mathbf{s}_0$ does), allowing direct expression of models such as stacked BiRNNs.

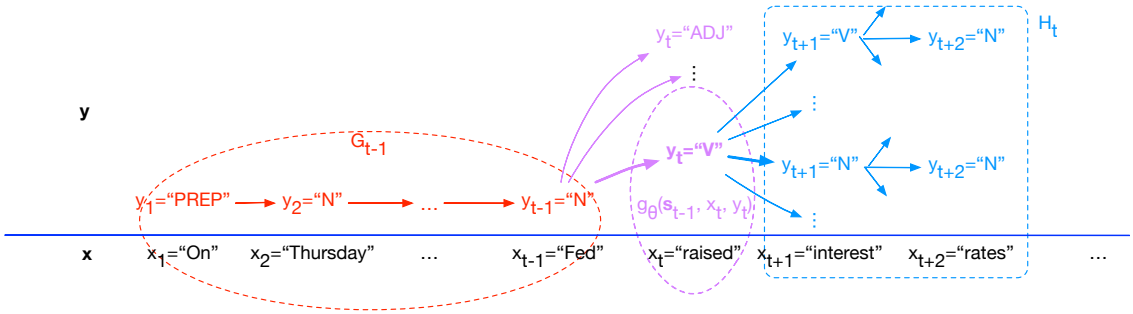


Figure 1: Sampling a single particle from a tagging model. $y_1, \ldots, y_{t-1}$ (orange) have already been chosen, with a total model score of G_{t-1}, and now the sampler is constructing a proposal distribution q (purple) from which the next tag y_t will be sampled. Each y_t is evaluated according to its contribution to G_t (namely g_θ) and its future score H_t (blue). The figure illustrates quantities used throughout the paper, beginning with exact sampling in equations (7)–(12). Our main idea (§3) is to *approximate* the H_t computation (a log-sum-exp over exponentially many sequences) when exact computation by dynamic programming is not an option. The form of our approximation uses a right-to-left recurrent neural network but is *inspired* by the exact dynamic programming algorithm.

H_t is the log of the total exponentiated scores of *all* sequences with prefix $\mathbf{y}_{:t}$. G_t and H_t might be called the *logprob-so-far* and *logprob-to-go* of $\mathbf{y}_{:t}$.

Just as A^* approximates h with a "heuristic" $\hat{h}$, the next section will approximate H_t using a neural estimate $\hat{H}_t$ (equations (5)–(6)). However, the specific form of our approximation is inspired by cases where H_t can be computed exactly. We consider those in the remainder of this section.

2.1 Exact sampling from HMMs

A *hidden Markov model* (HMM) specifies a normalized *joint* distribution $p_\theta(\mathbf{x}, \mathbf{y}) = \exp G_T$ over state sequence $\mathbf{y}$ and observation sequence $\mathbf{x}$,[3] Thus the posterior $p_\theta(\mathbf{y} \mid \mathbf{x})$ is proportional to $\exp G_T$, as required by equation (1).

The HMM specifically defines G_T by equations (2)–(3) with $\mathbf{s}_t = y_t$ and $g_\theta(\mathbf{s}_{t-1}, x_t, y_t) = \log p_\theta(y_t \mid y_{t-1}) + \log p_\theta(x_t \mid y_t)$.[4]

In this setting, H_t can be computed exactly by the *backward algorithm* (Rabiner, 1989). (Details are given in Appendix A for completeness.)

2.2 Exact sampling from OOHMMs

For sequence tagging, a weakness of (first-order) HMMs is that the model state $\mathbf{s}_t = y_t$ may contain little information: only the most recent tag y_t is remembered, so the number of possible model states $\mathbf{s}_t$ is limited by the vocabulary of output tags.

We may generalize so that the data generating process is in a latent state $u_t \in \{1, \ldots, k\}$ at each time t, and the observed y_t—along with x_t—is generated from u_t. Now k may be arbitrarily large. The model has the form

$$p_\theta(\mathbf{x}, \mathbf{y}) = \exp G_T \tag{13}$$
$$= \sum_{\mathbf{u}} \prod_{t=1}^{T} p_\theta(u_t \mid u_{t-1}) \cdot p_\theta(x_t, y_t \mid u_t)$$

This is essentially a pair HMM (Knudsen and Miyamoto, 2003) without insertions or deletions, also known as an "ϵ-free" or "same-length" probabilistic finite-state transducer. We refer to it here as an *output-output HMM* (OOHMM).[5]

Is this still an example of the general model architecture from §1.1? Yes. Since u_t is latent and evolves stochastically, it cannot be used as the state $\mathbf{s}_t$ in equations (2)–(3) or (4). However, we *can* define $\mathbf{s}_t$ to be the model's *belief state* after observing $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$. The belief state is the posterior probability distribution over the underlying state u_t of the system. That is, $\mathbf{s}_t$ deterministically keeps track of all possible states that the OOHMM might be in—just as the state of a determinized FSA keeps track of all possible states that the original nondeterministic FSA might be in.

We may compute the belief state in terms of a vector of *forward probabilities* that starts at $\boldsymbol{\alpha}_0$,

$$(\boldsymbol{\alpha}_0)_u \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } u = \text{BOS (see footnote 4)} \\ 0 & \text{if } u = \text{any other state} \end{cases} \tag{14}$$

and is updated deterministically for each $0 < t \leq T$ by the *forward algorithm* (Rabiner, 1989):

$$(\boldsymbol{\alpha}_t)_u \stackrel{\text{def}}{=} \sum_{u'=1}^{k} (\boldsymbol{\alpha}_{t-1})_{u'} \cdot p_\theta(u \mid u') \cdot p_\theta(x_t, y_t \mid u) \tag{15}$$

[3] The HMM actually specifies a distribution over a pair of infinite sequences, but here we consider the marginal distribution over just the length-T prefixes.

[4] It takes $\mathbf{s}_0 = \text{BOS}$, a beginning-of-sequence symbol, so $p_\theta(y_1 \mid \text{BOS})$ specifies the initial state distribution.

[5] This is by analogy with the *input-output HMM* (IOHMM) of Bengio and Frasconi (1996), which defines $p(\mathbf{y} \mid \mathbf{x})$ directly and conditions the transition to u_t on x_t. The OOHMM instead defines $p(\mathbf{y} \mid \mathbf{x})$ by conditionalizing (13)—which avoids the *label bias* problem (Lafferty et al., 2001) that in the IOHMM, y_t is independent of future input $\mathbf{x}_{t:}$ (given the past input $\mathbf{x}_{:t}$).

$(\boldsymbol{\alpha}_t)_u$ can be interpreted as the logprob-so-far *if* the system is in state u after observing $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$. We may express the update rule (15) by $\boldsymbol{\alpha}_t^\top = \boldsymbol{\alpha}_{t-1}^\top P$ where the matrix P depends on (x_t, y_t), namely $P_{u'u} \stackrel{\text{def}}{=} p_\theta(u \mid u') \cdot p_\theta(x_t, y_t \mid u)$.

The belief state $\mathbf{s}_t \stackrel{\text{def}}{=} [\![\boldsymbol{\alpha}_t]\!] \in \mathbb{R}^k$ simply normalizes $\boldsymbol{\alpha}_t$ into a probability vector, where $[\![\mathbf{u}]\!] \stackrel{\text{def}}{=} \mathbf{u}/(\mathbf{u}^\top \mathbf{1})$ denotes the *normalization operator*. The state update (15) now takes the form (3) as desired, with f_θ a normalized vector-matrix product:

$$\mathbf{s}_t^\top = f_\theta(\mathbf{s}_{t-1}, x_t, y_t) \stackrel{\text{def}}{=} [\![\mathbf{s}_{t-1}^\top P]\!] \tag{16}$$

As in the HMM case, we define G_t as the log of the generative prefix probability,

$$G_t \stackrel{\text{def}}{=} \log p_\theta(\mathbf{x}_{:t}, \mathbf{y}_{:t}) = \log \textstyle\sum_u (\boldsymbol{\alpha}_t)_u \tag{17}$$

which has the form (2) as desired if we put

$$\begin{aligned} g_\theta(\mathbf{s}_{t-1}, x_t, y_t) &\stackrel{\text{def}}{=} G_t - G_{t-1} \qquad (18)\\ &= \log \frac{\boldsymbol{\alpha}_{t-1}^\top P \mathbf{1}}{\boldsymbol{\alpha}_{t-1}^\top \mathbf{1}} = \log (\mathbf{s}_{t-1}^\top P \mathbf{1}) \end{aligned}$$

Again, exact sampling is possible. It suffices to compute (9). For the OOHMM, this is given by

$$\textstyle\sum_{\mathbf{y}_{t:}} \exp G_T = \boldsymbol{\alpha}_t^\top \boldsymbol{\beta}_t \tag{19}$$

where $\boldsymbol{\beta}_T \stackrel{\text{def}}{=} \mathbf{1}$ and the *backward algorithm*

$$\begin{aligned} (\boldsymbol{\beta}_t)_v &\stackrel{\text{def}}{=} p_\theta(\mathbf{x}_{t:} \mid u_t = u) \qquad (20)\\ &= \sum_{\mathbf{u}_{t:}, \mathbf{y}_{t:}} p_\theta(\mathbf{u}_{t:}, \mathbf{x}_{t:}, \mathbf{y}_{t:} \mid u_t = u)\\ &= \sum_{u'} \underbrace{p_\theta(u' \mid u) \cdot p(x_{t+1} \mid u')}_{\text{call this } \overline{P}_{uu'}} \cdot (\boldsymbol{\beta}_{t+1})_{u'} \end{aligned}$$

for $0 \le t < T$ uses dynamic programming to find the total probability of all ways to generate the future observations $\mathbf{x}_{t:}$. Note that $\boldsymbol{\alpha}_t$ is defined for a *specific prefix* $\mathbf{y}_{:t}$ (though it sums over all $\mathbf{u}_{:t}$), whereas $\boldsymbol{\beta}_t$ sums over *all suffixes* $\mathbf{y}_{t:}$ (and over all $\mathbf{u}_{t:}$), to achieve the asymmetric summation in (19).

Define $\bar{\mathbf{s}}_t \stackrel{\text{def}}{=} [\![\boldsymbol{\beta}_t]\!] \in \mathbb{R}^k$ to be a normalized version of $\boldsymbol{\beta}_t$. The $\boldsymbol{\beta}_t$ recurrence (20) can clearly be expressed in the form $\bar{\mathbf{s}}_t = [\![\overline{P} \bar{\mathbf{s}}_{t+1}]\!]$, much like (16).

2.3 The logprob-to-go for OOHMMs

Let us now work out the definition of H_t for OOHMMs (cf. equation (35) in Appendix A for HMMs). We will write it in terms of $\hat{H}_t$ from §1.2. Let us define $\hat{H}_t$ symmetrically to G_t (see (17)):

$$\hat{H}_t \stackrel{\text{def}}{=} \log \sum_u (\boldsymbol{\beta}_t)_u \;\; (= \log \mathbf{1}^\top \boldsymbol{\beta}_t) \tag{21}$$

which has the form (5) as desired if we put

$$h_\phi(\bar{\mathbf{s}}_{t+1}, x_{t+1}) \stackrel{\text{def}}{=} \hat{H}_t - \hat{H}_{t+1} = \log (\mathbf{1}^\top \overline{P} \bar{\mathbf{s}}_{t+1}) \tag{22}$$

From equations (10), (17), (19) and (21), we see

$$\begin{aligned} H_t &= \log \Big(\sum_{\mathbf{y}_{t:}} \exp G_T \Big) - G_t\\ &= \log \frac{\boldsymbol{\alpha}_t^\top \boldsymbol{\beta}_t}{(\boldsymbol{\alpha}_t^\top \mathbf{1})(\mathbf{1}^\top \boldsymbol{\beta}_t)} + \log (\mathbf{1}^\top \boldsymbol{\beta}_t)\\ &= \underbrace{\log \mathbf{s}_t^\top \bar{\mathbf{s}}_t}_{\text{call this } C_t} + \hat{H}_t \qquad (23) \end{aligned}$$

where $C_t \in \mathbb{R}$ can be regarded as evaluating the *compatibility* of the state distributions $\mathbf{s}_t$ and $\bar{\mathbf{s}}_t$.

In short, the generic strategy (12) for exact sampling says that for an OOHMM, y_t is distributed as

$$\begin{aligned} p_\theta(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1}) &\propto \exp \left(g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + H_t \right)\\ &\propto \exp \Big(\underbrace{g_\theta(\mathbf{s}_{t-1}, x_t, y_t)}_{\text{depends on } \mathbf{x}_{:t}, \mathbf{y}_{:t}} + \underbrace{C_t}_{\text{on } \mathbf{x}, \mathbf{y}_{:t}} + \underbrace{\hat{H}_t}_{\text{on } \mathbf{x}_{t:}} \Big)\\ &\propto \exp \left(g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + C_t \right) \qquad (24) \end{aligned}$$

This is equivalent to choosing y_t in proportion to (19)—but we now turn to settings where it is infeasible to compute (19) exactly. There we will use the formulation (24) but approximate C_t. For completeness, we will also consider how to approximate $\hat{H}_t$, which dropped out of the above distribution (because it was the same for all choices of y_t) but may be useful for other algorithms (see §4).

3 Neural Modeling as Approximation

3.1 Models with large state spaces

The expressivity of an OOHMM is limited by the number of states k. The state $u_t \in \{1, \ldots, k\}$ is a bottleneck between the past $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$ and the future $(\mathbf{x}_{t:}, \mathbf{y}_{t:})$, in that past and future are *conditionally independent* given u_t. Thus, the mutual information between past and future is at most $\log_2 k$ bits.

In many NLP domains, however, the past seems to carry substantial information about the future. The first half of a sentence greatly reduces the uncertainly about the second half, by providing information about topics, referents, syntax, semantics, and discourse. This suggests that an accurate HMM language model $p(\mathbf{x})$ would require *very large* k—as would a generative OOHMM model $p(\mathbf{x}, \mathbf{y})$ of *annotated* language. The situation is perhaps better for discriminative models $p(\mathbf{y} \mid \mathbf{x})$, since much of

the information for predicting $\mathbf{y}_{t:}$ might be available in $\mathbf{x}_{t:}$. Still, it is important to let $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$ contribute enough additional information about $\mathbf{y}_{t:}$: even for short strings, making k too small (giving $\leq \log_2 k$ bits) may harm prediction (Dreyer et al., 2008).

Of course, (4) says that an OOHMM can express any joint distribution for which the mutual information is finite,[6] by taking k large enough for v_{t-1} to capture the relevant info from $(\mathbf{x}_{:t-1}, \mathbf{y}_{:t-1})$.

So why not just take k to be large—say, $k = 2^{30}$ to allow 30 bits of information? Unfortunately, evaluating G_T then becomes very expensive—both computationally and statistically. As we have seen, if we define $\mathbf{s}_t$ to be the belief state $[\![\boldsymbol{\alpha}_t]\!] \in \mathbb{R}^k$, updating it at each observation (x_t, y_t) (equation (3)) requires multiplication by a $k \times k$ matrix P. This takes time $O(k^2)$, and requires enough data to learn $O(k^2)$ transition probabilities.

3.2 Neural approximation of the model

As a solution, we might hope that for the inputs $\mathbf{x}$ observed in practice, the very high-dimensional belief states $[\![\boldsymbol{\alpha}_t]\!] \in \mathbb{R}^k$ might tend to lie near a d-dimensional manifold where $d \ll k$. Then we could take $\mathbf{s}_t$ to be a vector in $\mathbb{R}^d$ that compactly encodes the approximate coordinates of $[\![\boldsymbol{\alpha}_t]\!]$ relative to the manifold: $\mathbf{s}_t = \nu([\![\boldsymbol{\alpha}_t]\!])$, where ν is the encoder.

In this new, nonlinearly warped coordinate system, the functions of $\mathbf{s}_{t-1}$ in (2)–(3) are no longer the simple, essentially linear functions given by (16) and (18). They become nonlinear functions operating on the manifold coordinates. (f_θ in (16) should now ensure that $\mathbf{s}_t^\top \approx \nu([\![(\nu^{-1}(\mathbf{s}_{t-1}))^\top P]\!])$, and g_θ in (18) should now estimate $\log\, (\nu^{-1}(\mathbf{s}_{t-1}))^\top P\mathbf{1}$.) In a sense, this is the reverse of the "kernel trick" (Boser et al., 1992) that converts a low-dimensional nonlinear function to a high-dimensional linear one.

Our hope is that $\mathbf{s}_t$ has enough dimensions $d \ll k$ to capture the useful information from the true $[\![\boldsymbol{\alpha}_t]\!]$, **and** that θ has enough dimensions $\ll k^2$ to capture most of the dynamics of equations (16) and (18). We thus proceed to fit the neural networks f_θ, g_θ directly to the data, *without ever knowing* the true k or the structure of the original operators $P \in \mathbb{R}^{k \times k}$.

We regard this as the implicit justification for various published probabilistic sequence models $p_\theta(\mathbf{y} \mid \mathbf{x})$ that incorporate neural networks. These models usually have the form of §1.1. Most simply, (f_θ, g_θ) can be instantiated as one time step in an RNN (Aharoni and Goldberg, 2017), but it is common to use enriched versions such as deep LSTMs. It is also common to have the state $\mathbf{s}_t$ contain not only a vector of manifold coordinates in $\mathbb{R}^d$ but also some unboundedly large representation of $(\mathbf{x}, \mathbf{y}_{:t})$ (cf. equation (4)), so the f_θ neural network can refer to this material with an attentional (Bahdanau et al., 2015) or stack mechanism (Dyer et al., 2015).

A few such papers have used *globally* normalized conditional models that can be viewed as approximating some OOHMM, e.g., the parsers of Dyer et al. (2016) and Andor et al. (2016). That is the case (§1.1) that particle smoothing aims to support. Most papers are *locally* normalized conditional models (e.g., Kann and Schütze, 2016; Aharoni and Goldberg, 2017); these simplify supervised training and can be viewed as approximating IOHMMs (footnote 5). For locally normalized models, $H_t = 0$ by construction, in which case particle filtering (which estimates $H_t = 0$) is just as good as particle smoothing. Particle filtering is still useful for these models, but lookahead's inability to help them is an expressive limitation (known as *label bias*) of locally normalized models. We hope the existence of particle smoothing (which learns an estimate H_t) will make it easier to adopt, train, and decode globally normalized models, as discussed in §1.3.

3.3 Neural approximation of logprob-to-go

We can adopt the same neuralization trick to approximate the OOHMM's logprob-to-go $H_t = C_t + \hat{H}_t$. We take $\bar{\mathbf{s}}_t \in \mathbb{R}^d$ on the same theory that it is a low-dimensional reparameterization of $[\![\beta_t]\!]$, and define $(\bar{f}_\phi, h_\phi)$ in equations (5)–(6) to be neural networks. Finally, we must replace the definition of C_t in (23) with another neural network c_ϕ that works on the low-dimensional approximations:[7]

$$C_t \stackrel{\text{def}}{=} c_\phi(\mathbf{s}_t, \bar{\mathbf{s}}_t) \quad \text{(except that } C_T \stackrel{\text{def}}{=} 0) \tag{25}$$

The resulting approximation to (24) (which does not actually require h_ϕ) will be denoted $q_{\theta,\phi}$:

$$q_{\theta,\phi}(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1}) \stackrel{\text{def}}{\propto} \exp\, (g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + C_t) \tag{26}$$

The neural networks in the present section are all parameterized by ϕ, and are intended to produce an estimate of the logprob-*to-go* H_t—a function of $\mathbf{x}_{t:}$, which sums over all possible $\mathbf{y}_{t:}$.

By contrast, the OOHMM-inspired neural networks suggested in §3.2 were used to specify an

[6] This is not true for the language of balanced parentheses.

[7] $C_T = 0$ is correct according to (23). Forcing this ensures $H_T = 0$, so our approximation becomes exact as of $t = T$.

actual model of the logprob-*so-far* G_t—a function of $\mathbf{x}_{:t}$ and $\mathbf{y}_{:t}$—using separate parameters θ.

Arguably ϕ has a harder modeling job than θ because it must implicitly sum over possible futures $\mathbf{y}_{t:}$. We now consider how to get corrected samples from $q_{\theta,\phi}$ even if ϕ gives poor estimates of H_t, and then how to train ϕ to improve those estimates.

4 Particle smoothing

In this paper, we assume nothing about the given model G_T except that it is given in the form of equations (1)–(3) (including the parameter vector θ).

Suppose we run the exact sampling strategy but approximate p_θ in (7) with a *proposal distribution* $q_{\theta,\phi}$ of the form in (25)–(26). Suppressing the subscripts on p and q for brevity, this means we are effectively drawing $\mathbf{y}$ not from $p(\mathbf{y} \mid \mathbf{x})$ but from

$$q(\mathbf{y} \mid \mathbf{x}) = \prod_{t=1}^{T} q(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1}) \tag{27}$$

If $C_t \approx H_t + \text{const}$ within each y_t draw, then $q \approx p$.

Normalized importance sampling corrects (mostly) for the approximation by drawing *many* sequences $\mathbf{y}^{(1)}, \ldots \mathbf{y}^{(M)}$ IID from (27) and assigning $\mathbf{y}^{(m)}$ a relative *weight* of $w^{(m)} \stackrel{\text{def}}{=} \frac{p(\mathbf{y}^{(m)} \mid \mathbf{x})}{q(\mathbf{y}^{(m)} \mid \mathbf{x})}$. This *ensemble of weighted particles* yields a distribution

$$\hat{p}(\mathbf{y}) \stackrel{\text{def}}{=} \frac{\sum_{m=1}^{M} w^{(m)} \mathbb{I}(\mathbf{y}=\mathbf{y}^{(m)})}{\sum_{m=1}^{M} w^{(m)}} \approx p(\mathbf{y} \mid \mathbf{x}) \tag{28}$$

that can be used as discussed in §1.3. To compute $w^{(m)}$ in practice, we replace the numerator $p(\mathbf{y}^{(m)} \mid \mathbf{x})$ by the unnormalized version $\exp G_T$, which gives the same $\hat{p}$. Recall that each G_T is a sum $\sum_{t=1}^{T} g_\theta(\cdots)$.

Sequential importance sampling is an equivalent implementation that makes t the *outer* loop and m the *inner* loop. It computes a *prefix ensemble*

$$Y_t \stackrel{\text{def}}{=} \{(\mathbf{y}_{:t}^{(1)}, w_t^{(1)}), \ldots, (\mathbf{y}_{:t}^{(M)}, w_t^{(M)})\} \tag{29}$$

for each $0 \le t \le T$ in sequence. Initially, $(\mathbf{y}_{:0}^{(m)}, w_0^{(m)}) = (\epsilon, \exp C_0)$ for all m. Then for $0 < t \le T$, we extend these particles in parallel:

$$\mathbf{y}_{:t}^{(m)} = \mathbf{y}_{:t-1}^{(m)} y_t^{(m)} \quad \text{(concatenation)} \tag{30}$$

$$w_t^{(m)} = w_{t-1}^{(m)} \frac{\exp\left(g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + C_t - C_{t-1}\right)}{q(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1})} \tag{31}$$

where each $y_t^{(m)}$ is drawn from (26). Each Y_t yields a distribution $\hat{p}_t$ over prefixes $\mathbf{y}_{:t}$, which estimates the distribution $p_t(\mathbf{y}_{:t}) \stackrel{\text{def}}{\propto} \exp\left(G_t + C_t\right)$. We return $\hat{p} \stackrel{\text{def}}{=} \hat{p}_T \approx p_T = p$. This gives the same $\hat{p}$ as in (28): the final $\mathbf{y}_T^{(m)}$ are the same, with the same final weights $w_T^{(m)} = \frac{\exp G_T}{q(\mathbf{y}^{(m)} \mid \mathbf{x})}$, where G_T was now summed up as $C_0 + \sum_{t=1}^{T} g_\theta(\cdots) + C_t - C_{t-1}$.

That is our basic *particle smoothing* strategy. If we use the naive approximation $C_t = 0$ everywhere, it reduces to *particle filtering*. In either case, various well-studied improvements become available, such as various resampling schemes (Douc and Cappé, 2005) and the particle cascade (Paige et al., 2014).[8]

An easy improvement is *multinomial resampling*. After computing each $\hat{p}_t$, this replaces Y_t with a set of M new draws from $\hat{p}_t$ ($\approx p_t$), each of weight 1—which tends to drop low-weight particles and duplicate high-weight ones.[9] For this to usefully focus the ensemble on good prefixes $\mathbf{y}_{:t}$, p_t should be a good approximation to the true marginal $p(\mathbf{y}_{:t} \mid \mathbf{x}) \propto \exp\left(G_t + H_t\right)$ from (10). That is why we arranged for $p_t(\mathbf{y}_{:t}) \propto \exp\left(G_t + C_t\right)$. Without C_t, we would have only $p_t(\mathbf{y}_{:t}) \propto \exp G_t$—which is fine for the traditional particle filtering setting, but in our setting it ignores future information in $\mathbf{x}_{t:}$ (which we have assumed is available) and also favors sequences $\mathbf{y}$ that happen to accumulate most of their global score G_T early rather than late (which is possible when the globally normalized model (1)–(2) is *not* factored in the generative form (4)).

5 Training the Sampler Heuristic

We now consider training the parameters ϕ of our sampler. These parameters determine the updates $\bar{f}_\phi$ in (6) and the compatibility function c_ϕ in (25). As a result, they determine the proposal distribution q used in equations (27) and (31), and thus determine the stochastic choice of $\hat{p}$ that is returned by the sampler on a given input $\mathbf{x}$.

In this paper, we simply try to tune ϕ to yield good proposals. Specifically, we try to ensure that $q_\phi(\mathbf{y} \mid \mathbf{x})$ in equation (27) is close to $p(\mathbf{y} \mid \mathbf{x})$ from equation (1). While this may not be necessary for the sampler to perform well downstream,[10] it does

[8] The particle cascade would benefit from an estimate of $\hat{H}_t$, as it (like A* search) compares particles of different lengths.

[9] While resampling mitigates the degeneracy problem, it could also reduce the diversity of particles. In our experiments in this paper, we only do multinomial resampling when the effective sample size of $\hat{p}_t$ is lower than $\frac{M}{2}$. Doucet and Johansen (2009) give a more thorough discussion on when to resample.

[10] In principle, one could attempt to train ϕ "end-to-end" on some downstream objective by using reinforcement learning or the Gumbel-softmax trick (Jang et al., 2017; Maddison et al., 2017). For example, we might try to ensure that $\hat{p}$ closely matches the model's distribution p (equation (28))—the "na-

guarantee it (assuming that the model p is correct). Specifically, we seek to minimize

$$(1-\lambda)\mathrm{KL}(p||q_\phi) + \lambda\mathrm{KL}(q_\phi||p) \quad (\text{with } \lambda \in [0,1]) \tag{32}$$

averaged over examples $\mathbf{x}$ drawn from a training set.[11] (The training set need not provide true $\mathbf{y}$'s.)

The *inclusive KL divergence* $\mathrm{KL}(p||q_\phi)$ is an expectation under p. We estimate it by replacing p with a sample $\hat{p}$, which in practice we can obtain with our sampler under the current ϕ. (The danger, then, is that $\hat{p}$ will be biased when ϕ is not yet well-trained; this can be mitigated by increasing the sample size M when drawing $\hat{p}$ for training purposes.)

Intuitively, this term tries to encourage q_ϕ in future to re-propose those $\mathbf{y}$ values that turned out to be "good" and survived into $\hat{p}$ with high weights.

The *exclusive KL divergence* $\mathrm{KL}(q_\phi||p)$ is an expectation under q_ϕ. Since we can sample from q_ϕ exactly, we can get an unbiased estimate of $\nabla_\phi \mathrm{KL}(q_\phi||p)$ with the likelihood ratio trick (Glynn, 1990).[12] (The danger is that such "REINFORCE" methods tend to suffer from very high variance.)

This term is a popular objective for variational approximation. Here, it tries to discourage q_ϕ from re-proposing "bad" $\mathbf{y}$ values that turned out to have low $\exp G_T$ relative to their proposal probability.

Our experiments balance "recall" (inclusive) and "precision" (exclusive) by taking $\lambda = \frac{1}{2}$ (which Appendix F compares to $\lambda \in \{0, 1\}$). Alas, because of our approximation to the inclusive term, neither term's gradient will "find" and directly encourage good $\mathbf{y}$ values that have never been proposed. Appendix B gives further discussion and formulas.

6 Models for the Experiments

To evaluate our methods, we needed pre-trained models p_θ. We experimented on several models. In each case, we trained a *generative* model $p_\theta(\mathbf{x}, \mathbf{y})$, so that we could try sampling from its posterior distribution $p_\theta(\mathbf{y} \mid \mathbf{x})$. This is a very common setting where particle smoothing should be able to help. Details for replication are given in Appendix C.

tural" goal of sampling. This objective can tolerate inaccurate local proposal distributions in cases where the algorithm could recover from them through resampling. Looking even farther downstream, we might merely want $\hat{p}$—which is typically used to compute expectations—to provide accurate guidance to some decision or training process (see Appendix E). This might not require fully matching the model, and might even make it desirable to deviate from an inaccurate model.

[11] Training a single approximation q_ϕ for all $\mathbf{x}$ is known as *amortized inference*.

[12] The normalizing constant of p from (1) can be ignored because the gradient of a constant is 0.

6.1 Tagging models

We can regard a tagged sentence $(\mathbf{x}, \mathbf{y})$ as a string over the "pair alphabet" $\mathcal{X} \times \mathcal{Y}$. We train an RNN language model over this "pair alphabet"—this is a neuralized OOHMM as suggested in §3.2:

$$\log p_\theta(\mathbf{x}, \mathbf{y}) = \sum_{t=1}^{T} \log p_\theta(x_t, y_t \mid \mathbf{s}_{t-1}) \tag{33}$$

This model is locally normalized, so that $\log p_\theta(\mathbf{x}, \mathbf{y})$ (as well as its gradient) is straightforward to compute for a given training pair $(\mathbf{x}, \mathbf{y})$. Joint sampling from it would also be easy (§3.2).

However, $p(\mathbf{y} \mid \mathbf{x})$ is globally renormalized (by an unknown partition function that depends on $\mathbf{x}$, namely $\exp H_0$). Conditional sampling of $\mathbf{y}$ is therefore potentially hard. Choosing y_t optimally requires knowledge of H_t, which depends on the future $\mathbf{x}_{t:}$.

As we noted in §1, many NLP tasks can be seen as tagging problems. In this paper we experiment with two such tasks: **English stressed syllable tagging**, where the stress of a syllable often depends on the number of remaining syllables,[13] providing good reason to use the *lookahead* provided by particle smoothing; and **Chinese NER**, which is a familiar textbook application and reminds the reader that our formal setup (tagging) provides enough machinery to treat other tasks (chunking).

English stressed syllable tagging This task tags a sequence of phonemes $\mathbf{x}$, which form a word, with their stress markings $\mathbf{y}$. Our training examples are the stressed words in the CMU pronunciation dictionary (Weide, 1998). We test the sampler on held-out unstressed words.

Chinese social media NER This task does named entity recognition in Chinese, by tagging the characters of a Chinese sentence in a way that marks the named entities. We use the dataset from Peng and Dredze (2015), whose tagging scheme is a variant of the BIO scheme mentioned in §1. We test the sampler on held-out sentences.

6.2 String source separation

This is an artificial task that provides a discrete analogue of speech source separation (Zibulevsky and Pearlmutter, 2001). The generative model is that J strings (possibly of different lengths) are generated

[13] English, like many other languages, assigns stress from right to left (Hayes, 1995).

IID from an RNN language model, and are then combined into a single string $\mathbf{x}$ according to a random *interleaving* string $\mathbf{y}$.[14] The posterior $p(\mathbf{y} \mid \mathbf{x})$ predicts the interleaving string, which suffices to reconstruct the original strings. The interleaving string is selected from the uniform distribution over all possible interleavings (given the J strings' lengths). For example, with $J = 2$, a possible generative story is that we first sample two strings Foo and Bar from an RNN language model. We then draw an interleaving string 112122 from the aforementioned uniform distribution, and interleave the J strings deterministically to get FoBoar.

$p(\mathbf{x}, \mathbf{y})$ is proportional to the product of the probabilities of the J strings. The only parameters of p_θ, then, are the parameters of the RNN language model, which we train on clean (non-interleaved) samples from a corpus. We test the sampler on random interleavings of held-out samples.

The state $\mathbf{s}$ (which is provided as an input to c_θ in (25)) is the concatenation of the J states of the language model as it independently generates the J strings, and $g_\theta(\mathbf{s}_{t-1}, x_t, y_t)$ is the log-probability of generating x_t as the next character of the $y_t{}^{\text{th}}$ string, given that string's language model state within $\mathbf{s}_{t-1}$. As a special case, $\mathbf{x}_T = \text{EOS}$ (see footnote 1), and $g_\theta(\mathbf{s}_{T-1}, \text{EOS}, \text{EOS})$ is the total log-probability of termination in all J language model states.

String source separation has good reason for lookahead: appending character "o" to a reconstructed string "␣gh" is only advisable if "s" and "t" are coming up soon to make "ghost." It also illustrates a powerful application setting—posterior inference under a generative model. This task conveniently allowed us to construct the generative model from a pre-trained language model. Our constructed generative model illustrates that the state $\mathbf{s}$ and transition function f can reflect interesting problem-specific structure.

CMU Pronunciation dictionary The CMU pronunciation dictionary (already used above) provides sequences of phonemes. Here we use words no longer than 5 phonemes. We interleave the (unstressed) phonemes of $J = 5$ words.

Penn Treebank The PTB corpus (Marcus et al., 1993) provides English sentences, from which we use only the sentences of length ≤ 8. We interleave the words of $J = 2$ sentences.

[14] We formally describe the generative process in Appendix G.

7 Experiments

In our experiments, we are given a pre-trained scoring model p_θ, and we train the parameters ϕ of a particle smoothing algorithm.[15]

We now show that our proposed neural particle smoothing sampler does better than the particle filtering sampler. To define "better," we evaluate samplers on the *offset KL divergence* from the true posterior.

7.1 Evaluation metrics

Given $\mathbf{x}$, the "natural" goal of conditional sampling is for the sample distribution $\hat{p}(\mathbf{y})$ to approximate the true distribution $p_\theta(\mathbf{y} \mid \mathbf{x}) = \exp G_T / \exp H_0$ from (1). We will therefore report—averaged over all held-out test examples $\mathbf{x}$—the KL divergence

$$\begin{aligned}\text{KL}(\hat{p}||p) = \mathbb{E}_{\mathbf{y}\sim\hat{p}}\left[\log \hat{p}(\mathbf{y})\right] \\ - \left(\mathbb{E}_{\mathbf{y}\sim\hat{p}}\left[\log \tilde{p}(\mathbf{y} \mid \mathbf{x})\right] - \log Z(\mathbf{x})\right),\end{aligned} \tag{34}$$

where $\tilde{p}(\mathbf{y} \mid \mathbf{x})$ denotes the *unnormalized* distribution given by $\exp G_T$ in (2), and $Z(\mathbf{x})$ denotes its normalizing constant, $\exp H_0 = \sum_{\mathbf{y}} \tilde{p}(\mathbf{y} \mid \mathbf{x})$.

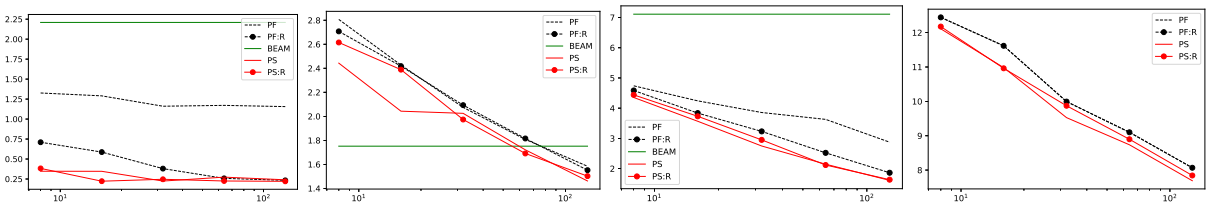
As we are unable to compute $\log Z(\mathbf{x})$ in practice, we replace it with an estimate $z(\mathbf{x})$ to obtain an *offset KL divergence*. This change of constant does not change the measured difference between two samplers, $\text{KL}(\hat{p}_1||p) - \text{KL}(\hat{p}_2||p)$. Nonetheless, we try to use a reasonable estimate so that the reported KL divergence is interpretable in an absolute sense. Specifically, we take $z(\mathbf{x}) = \log \sum_{\mathbf{y}\in\mathcal{Y}} \tilde{p}(\mathbf{y} \mid \mathbf{x}) \leq \log Z$, where $\mathcal{Y}$ is the full set of distinct particles $\mathbf{y}$ that we ever drew for input $\mathbf{x}$, including samples from the beam search models, while constructing the experimental results graph.[16] Thus, the offset KL divergence is a "best effort" lower bound on the true exclusive KL divergence $\text{KL}(\hat{p}||p)$.

7.2 Results

In all experiments we compute the offset KL divergence for both the particle filtering samplers and the particle smoothing samplers, for varying ensemble sizes M. We also compare against a beam search baseline that keeps the highest-scoring M particles at each step (scored by $\exp G_t$ with no lookahead). The results are in Figures 2a–2d.

[15] For the details of the training procedures and the specific neural architectures in our models, see Appendices C and D.

[16] Thus, $\mathcal{Y}$ was collected across all samplings, iterations,and ensemble sizes M, in an attempt to make the summation over $\mathcal{Y}$ as complete as possible. For good measure, we added some extra particles: whenever we drew M particles via particle smoothing, we drew an additional $2M$ particles by particle filtering and added them to $\mathcal{Y}$.



(a) tagging: stressed syllables (b) tagging: Chinese NER (c) source separation: PTB (d) source separation: CMUdict

Figure 2: Offset KL divergences for the tasks in §§ 6.1 and 6.2. The logarithmic x-axis is the size of particles M ($8 \leq M \leq 128$). The y-axis is the offset KL divergence described in §7.1 (in bits per sequence). The smoothing samplers offer considerable speedup: for example, in Figure 2a, the non-resampled smoothing sampler achieves comparable offset KL divergences with only $1/4$ as many particles as its filtering counterparts. Abbreviations in the legend: PF=particle filtering. PS=particle smoothing. BEAM=beam search. ':R' suffixes indicate resampled variants. For readability, beam search results are omitted from Figure 2d, but appear in Figure 3 of the appendices.

Given a fixed ensemble size, we see the smoothing sampler consistently performs better than the filtering counterpart. It often achieves comparable performance at a fraction of the ensemble size.

Beam search on the other hand falls behind on three tasks: stress prediction and the two source separation tasks. It does perform better than the stochastic methods on the Chinese NER task, but only at small beam sizes. Varying the beam size barely affects performance at all, across all tasks. This suggests that beam search is unable to explore the hypothesis space well.

We experiment with resampling for both the particle filtering sampler and our smoothing sampler. In source separation and stressed syllable prediction, where the right context contains critical information about how viable a particle is, resampling helps particle filtering *almost* catch up to particle smoothing. Particle smoothing itself is not further improved by resampling, presumably because its effective sample size is high. The goal of resampling is to kill off low-weight particles (which were overproposed) and reallocate their resources to higher-weight ones. But with particle smoothing, there are fewer low-weight particles, so the benefit of resampling may be outweighted by its cost (namely, increased variance).

8 Related Work

Much previous work has employed sequential importance sampling for approximate inference of intractable distributions (e.g., Thrun, 2000; Andrews et al., 2017). Some of this work learns adaptive proposal distributions in this setting (e.g. Gu et al., 2015; Paige and Wood, 2016). The key difference in our work is that we consider future inputs, which is impossible in online decision settings such as robotics. Klaas et al. (2006) did do particle smoothing, like us, but they did not learn adaptive proposal distributions.

Just as we use a right-to-left RNN to guide *posterior sampling* of a left-to-right generative model, Krishnan et al. (2017) employed a right-to-left RNN to guide *posterior marginal inference* in the same sort of model. Serdyuk et al. (2018) used a right-to-left RNN to regularize training of such a model.

9 Conclusion

We have described neural particle smoothing, a sequential Monte Carlo method for approximate sampling from the posterior of incremental neural scoring models. Sequential importance sampling has arguably been underused in the natural language processing community. It is quite a plausible strategy for dealing with rich, globally normalized probability models such as neural models—particularly if a good sequential proposal distribution can be found. Our contribution is a neural proposal distribution, which goes beyond particle filtering in that it uses a right-to-left recurrent neural network to "look ahead" to future symbols of $\mathbf{x}$ when proposing each symbol y_t. The form of our distribution is well-motivated.

There are many possible extensions to the work in this paper. For example, we can learn the generative model and proposal distribution jointly; we can also infuse them with hand-crafted structure, or use more deeply stacked architectures; and we can try training the proposal distribution end-to-end (footnote 10). Another possible extension would be to allow each step of q to propose a *sequence* of actions, effectively making the tagset size ∞. This extension relaxes our $|\mathbf{y}| = |\mathbf{x}|$ restriction from §1 and would allow us to do general sequence-to-sequence transduction.

Acknowledgements

This work has been generously supported by a Google Faculty Research Award and by Grant No. 1718846 from the National Science Foundation.

References

Roee Aharoni and Yoav Goldberg. 2017. Morphological inflection generation with hard monotonic attention. In *ACL*.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. In *ACL*.

Nicholas Andrews, Mark Dredze, Benjamin Van Durme, and Jason Eisner. 2017. Bayesian modeling of lexical resources for low-resource settings. In *ACL*.

Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio. 2017. An actor-critic algorithm for sequence prediction. In *ICLR*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *ICLR*.

Yoshua Bengio and Paolo Frasconi. 1996. Input-output HMMs for sequence processing. *IEEE Transactions on Neural Networks*, 7(5):1231–1249.

Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. 1992. A training algorithm for optimal margin classifiers. In *COLT*.

Alexandre Bouchard-Côté, Percy Liang, Thomas Griffiths, and Dan Klein. 2007. A probabilistic approach to diachronic phonology. In *EMNLP-CoNLL*, pages 887–896.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*.

Ryan Cotterell, John Sylak-Glassman, and Christo Kirov. 2017. Neural graphical models over strings for principal parts morphological paradigm completion. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 759–765.

Randal Douc and Olivier Cappé. 2005. Comparison of resampling schemes for particle filtering. In *Image and Signal Processing and Analysis, 2005. ISPA 2005. Proceedings of the 4th International Symposium on*, pages 64–69. IEEE.

Arnaud Doucet and Adam M. Johansen. 2009. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of Nonlinear Filtering*, 12(656-704):3.

Markus Dreyer and Jason Eisner. 2009. Graphical models over multiple strings. In *EMNLP*.

Markus Dreyer, Jason R. Smith, and Jason Eisner. 2008. Latent-variable modeling of string transductions with finite-state methods. In *EMNLP*.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *ACL*.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent neural network grammars. In *HLT-NAACL*.

Jenny Rose Finkel, Christopher D. Manning, and Andrew Y. Ng. 2006. Solving the problem of cascading errors: Approximate Bayesian inference for linguistic annotation pipelines. In *EMNLP*.

Peter W. Glynn. 1990. Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM*, 33(10):75–84.

Shixiang Gu, Zoubin Ghahramani, and Richard E. Turner. 2015. Neural adaptive sequential Monte Carlo. In *NIPS*.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimal cost paths. 4(2):100–107.

Bruce Hayes. 1995. *Metrical Stress Theory: Principles and Case Studies*. University of Chicago Press.

Alexander T. Ihler and David A. McAllester. 2009. Particle belief propagation. In *AISTATS*.

Eric Jang, Shixiang Gu, and Ben Poole. 2017. Categorical reparameterization with Gumbel-softmax. In *ICLR*.

Katharina Kann and Hinrich Schütze. 2016. Single-model encoder-decoder with explicit morphological representation for reinflection. In *ACL*.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Mike Klaas, Mark Briers, Nando de Freitas, Arnaud Doucet, Simon Maskell, and Dustin Lang. 2006. Fast particle smoothing: If I had a million particles. In *ICML*.

Bjarne Knudsen and Michael M. Miyamoto. 2003. Sequence alignments and pair hidden Markov models using evolutionary history. *Journal of Molecular Biology*, 333(2):453 – 460.

Rahul G. Krishnan, Uri Shalit, and David Sontag. 2017. Structured inference networks for nonlinear state space models. In *AAAI*.

John Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*.

Thibaut Lienart, Yee Whye Teh, and Arnaud Doucet. 2015. Expectation particle belief propagation. In *NIPS*.

Roderick J. A. Little and Donald B. Rubin. 1987. *Statistical Analysis with Missing Data*. J. Wiley & Sons, New York.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. 2017. The concrete distribution: A continuous relaxation of discrete random variables. In *ICLR*.

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

Andrew McCallum, Dayne Freitag, and Fernando Pereira. 2000. Maximum entropy Markov models for information extraction and segmentation. In *Machine Learning: Proceedings of the 17th International Conference (ICML 2000)*, pages 591–598, Stanford, CA.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3.

Brooks Paige and Frank D. Wood. 2016. Inference networks for sequential Monte Carlo in graphical models. In *ICML*.

Brooks Paige, Frank D. Wood, Arnaud Doucet, and Yee Whye Teh. 2014. Asynchronous anytime sequential Monte Carlo. In *NIPS*.

Nanyun Peng and Mark Dredze. 2015. Named entity recognition for Chinese social media with jointly trained embeddings. In *EMNLP*.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global vectors for word representation. In *EMNLP*.

Fernando C. N. Pereira and Michael D. Riley. 1997. Speech recognition by composition of weighted finite automata. *Finite-State Language Processing*, page 431.

Lawrence R. Rabiner. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of IEEE*, 77(2):257–285.

Lance A. Ramshaw and Mitchell P. Marcus. 1999. Text chunking using transformation-based learning. In *Natural Language Processing Using Very Large Corpora*, pages 157–176. Springer.

Branko Ristic, Sanjeev Arulampalam, and Neil James Gordon. 2004. *Beyond the Kalman Filter: Particle Filters for Tracking Applications*. Artech House.

Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407.

Dmitriy Serdyuk, Nan Rosemary Ke, Alessandro Sordoni, Adam Trischler, Chris Pal, and Yoshua Bengio. 2018. Twin networks: Matching the future for sequence generation. In *ICLR*.

Andreas Stuhlmüller, Jacob Taylor, and Noah Goodman. 2013. Learning stochastic inverses. In *NIPS*.

Sebastian Thrun. 2000. Monte Carlo POMDPs. In *NIPS*.

Andrew J. Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13(2):260–269.

Greg C. G. Wei and Martin A. Tanner. 1990. A Monte Carlo implementation of the EM algorithm and the poor man's data augmentation algorithms. *Journal of the American Statistical Association*, 85(411):699–704.

Robert L. Weide. 1998. The CMU pronunciation dictionary, release 0.6.

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(23).

Sam Wiseman and Alexander M. Rush. 2016. Sequence-to-sequence learning as beam-search optimization. In *EMNLP*.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, volume 3, pages 195–206.

Michael Zibulevsky and Barak A. Pearlmutter. 2001. Blind source separation by sparse decomposition in a signal dictionary. *Neural Computation*, 13(4):863–882.

A The logprob-to-go for HMMs

As noted in §2.1, the logprob-to-go H_t can be computed by the backward algorithm. By the definition of H_t in equation (10),

$$\exp H_t = \sum_{\mathbf{y}_{t:}} \exp\left(G_T - G_t\right) \tag{35}$$

$$= \sum_{\mathbf{y}_{t:}} \exp \sum_{j=t+1}^{T} g_\theta(\mathbf{s}_{j-1}, x_j, y_j) \tag{36}$$

$$= \sum_{\mathbf{y}_{t:}} \prod_{j=t+1}^{T} p_\theta(y_j \mid y_{j-1}) \cdot p_\theta(x_j \mid y_j)$$

$$= (\boldsymbol{\beta}_t)_{y_t} \text{ (backward prob of } y_t \text{ at time } t)$$

where the vector $\boldsymbol{\beta}_t$ is defined by base case $(\boldsymbol{\beta}_T)_y = 1$ and for $0 \leq t < T$ by the recurrence

$$(\beta_t)_y \stackrel{\text{def}}{=} \sum_{\mathbf{y}_{t:}} p_\theta(\mathbf{x}_{t:}, \mathbf{y}_{t:} \mid y_t = y) \tag{37}$$

$$= \sum_{y'} p_\theta(y' \mid y) \cdot p_\theta(x_{t+1} \mid y') \cdot (\boldsymbol{\beta}_{t+1})_{y'}$$

The backward algorithm (20) for OOHMMs in §2.2 is a variant of this.

B Gradients for Training the Proposal Distribution

For a given $\mathbf{x}$, both forms of KL divergence achieve their minimum of 0 when $(\forall \mathbf{y})\ q_\phi(\mathbf{y} \mid \mathbf{x}) = p(\mathbf{y} \mid \mathbf{x})$. However, we are unlikely to be able to find such a ϕ; the two metrics penalize q_ϕ differently for mismatches. We simplify the notation below by writing $q_\phi(\mathbf{y})$ and $p(\mathbf{y})$, suppressing the conditioning on $\mathbf{x}$.

Inclusive KL Divergence The inclusive KL divergence has that name because it is finite only when $\text{support}(q_\phi) \supseteq \text{support}(p)$, i.e., when q_ϕ is capable of proposing any string $\mathbf{y}$ that has positive probability under p. This is required for q_ϕ to be a valid proposal distribution for importance sampling.

$$\begin{aligned} &\text{KL}\left(p||q_\phi\right) \\ &\quad = \mathbb{E}_{\mathbf{y}\sim p}\left[\log p\left(\mathbf{y}\right) - \log q_\phi(\mathbf{y})\right] \\ &\quad = \mathbb{E}_{\mathbf{y}\sim p}\left[\log p\left(\mathbf{y}\right)\right] \\ &\qquad - \mathbb{E}_{\mathbf{y}\sim p}\left[\log q_\phi\left(\mathbf{y}\right)\right] \end{aligned} \tag{38}$$

The first term $\mathbb{E}_{\mathbf{y}\sim p}\left[\log p\left(\mathbf{y}\right)\right]$ is a constant with regard to ϕ. As a result, the gradient of the above is just the gradient of the second term:

$$\nabla_\phi \text{KL}(p||q_\phi) = \nabla_\phi \underbrace{\mathbb{E}_{\mathbf{y}\sim p}\left[-\log q_\phi\left(\mathbf{y}\right)\right]}_{\text{the cross-entropy } H(p,q_\phi)}$$

We cannot directly sample from p. However, our weighted mixture $\hat{p}$ from equation (28) (obtained by sequential importance sampling) could be a good approximation:

$$\begin{aligned} \nabla_\phi \text{KL}(p||q_\phi) &\approx \nabla_\phi \mathbb{E}_{\mathbf{y}\sim\hat{p}}\left[-\log q_\phi\left(\mathbf{y}\right)\right] \\ &= \sum_{t=1}^{T} \mathbb{E}_{\hat{p}}\left[-\nabla_\phi \log q_\phi(y_t \mid y_{:t-1}, \mathbf{x})\right] \end{aligned} \tag{39}$$

Following this approximate gradient downhill has an intuitive interpretation: if a particular y_t value ends up with high relative weight in the final ensemble $\hat{p}$, then we will try to adjust q_ϕ so that it would have had a high probability of proposing that y_t value at step t in the first place.

Exclusive KL Divergence The exclusive divergence has that name because it is finite only when $\text{support}(q_\phi) \subseteq \text{support}(p)$. It is defined by

$$\begin{aligned} \text{KL}(q_\phi||p) &= \mathbb{E}_{\mathbf{y}\sim q_\phi}\left[\log q_\phi(\mathbf{y}) - \log p(\mathbf{y})\right] \\ &= \mathbb{E}_{\mathbf{y}\sim q_\phi}\left[\log q_\phi(\mathbf{y}) - \log \tilde{p}(\mathbf{y})\right] + \log Z \\ &= \sum_{\mathbf{y}} q_\phi(\mathbf{y}) \underbrace{\left[\log q_\phi(\mathbf{y}) - \log \tilde{p}(\mathbf{y})\right]}_{\text{call this } d_\phi(\mathbf{y})} + \log Z \end{aligned} \tag{40}$$

where $p(\mathbf{y}) = \frac{1}{Z}\tilde{p}(\mathbf{y})$ for $\tilde{p}(\mathbf{y}) = \exp G_T$ and $Z = \sum_{\mathbf{y}} \tilde{p}(\mathbf{y})$. With some rearrangement, we can write its gradient as an expectation that can be estimated by sampling from q_ϕ.[17] Observing that Z is constant with respect to ϕ, first write

$$\nabla_\phi \text{KL}(q_\phi||p) \tag{41}$$

$$= \sum_{\mathbf{y}} \nabla_\phi \left(q_\phi(\mathbf{y})\, d_\phi(\mathbf{y})\right) \tag{42}$$

$$\begin{aligned} &= \sum_{\mathbf{y}} \left(\nabla_\phi q_\phi(\mathbf{y})\right)\, d_\phi(\mathbf{y}) \\ &\qquad + \sum_{\mathbf{y}} \underbrace{q_\phi(\mathbf{y}) \nabla_\phi \log q_\phi(\mathbf{y})}_{=\nabla_\phi q_\phi(\mathbf{y})} \\ &= \sum_{\mathbf{y}} \left(\nabla_\phi q_\phi(\mathbf{y})\right)\, d_\phi(\mathbf{y}) \end{aligned}$$

where the last step uses the fact that $\sum_{\mathbf{y}} \nabla_\phi q_\phi(\mathbf{y}) = \nabla_\phi \sum_{\mathbf{y}} q_\phi(\mathbf{y}) = \nabla_\phi 1 = 0$. We can turn this into an expectation with a second use of Glynn (1990)'s observation that

[17] This is an extension of the REINFORCE trick (Williams, 1992), which estimates the gradient of $\mathbb{E}_{\mathbf{y}\sim q_\phi}\left[\text{reward}(\mathbf{y})\right]$ when the reward is independent of ϕ. In our case, the expectation is over a quantity that does depend on ϕ.

$\nabla_\phi q_\phi(\mathbf{y}) = q_\phi(\mathbf{y}) \nabla_\phi \log q_\phi(\mathbf{y})$ (the "likelihood ratio trick"):

$$\begin{aligned}
&\nabla_\phi \mathrm{KL}(q_\phi || p) \\
&\quad = \sum_{\mathbf{y}} q_\phi(\mathbf{y}) d_\phi(\mathbf{y}) \nabla_\phi \log q_\phi(\mathbf{y}) \\
&\quad = \mathbb{E}_{\mathbf{y} \sim q_\phi} \left[d_\phi(\mathbf{y}) \nabla_\phi \log q_\phi(\mathbf{y}) \right] \qquad (43)
\end{aligned}$$

which can, if desired, be further rewritten as

$$\begin{aligned}
&= \mathbb{E}_{\mathbf{y} \sim q_\phi} \left[d_\phi(\mathbf{y}) \nabla_\phi d_\phi(\mathbf{y}) \right] \\
&= \mathbb{E}_{\mathbf{y} \sim q_\phi} \left[\nabla_\phi \left(\tfrac{1}{2} d_\phi(\mathbf{y})^2 \right) \right] \qquad (44)
\end{aligned}$$

If we regard $d_\phi(\mathbf{y})$ as a signed error (in the log domain) in trying to fit q_ϕ to $\tilde{p}$, then the above gradient of KL can be interpreted as the gradient of the mean squared error (divided by 2).[18]

We would get the same gradient for any rescaled version of the unnormalized distribution $\tilde{p}$, but the formula for obtaining that gradient would be different. In particular, if we rewrite the above derivation but add a constant b to both $\log \tilde{p}(\mathbf{y})$ and $\log Z$ throughout (equivalent to adding b to G_T), we will get the slightly generalized expectation formulas

$$\mathbb{E}_{\mathbf{y} \sim q_\phi} \left[(d_\phi(\mathbf{y}) - b) \nabla_\phi \log q_\phi(\mathbf{y}) \right] \qquad (45)$$

$$\mathbb{E}_{\mathbf{y} \sim q_\phi} \left[\nabla_\phi \left(\tfrac{1}{2} \left(d_\phi(\mathbf{y}) - b \right)^2 \right) \right] \qquad (46)$$

in place of equations (43) and (44). By choosing an appropriate "baseline" b, we can reduce the variance of the sampling-based estimate of these expectations. This is similar to the use of a baseline in the REINFORCE algorithm (Williams, 1992). In this work we choose b using an exponential moving average of past $\mathbb{E}\left[d_\phi(\mathbf{y})\right]$ values: at the end of each training minibatch, we update $b \leftarrow 0.1 \cdot b + 0.9 \cdot \bar{d}$, where $\bar{d}$ is the mean of the estimated $\mathbb{E}_{\mathbf{y} \sim q_\phi(\cdot|\mathbf{x})}\left[d_\phi(\mathbf{y})\right]$ values for all examples $\mathbf{x}$ in the minibatch.

C Implementation Details

We implement all RNNs in this paper as GRU networks (Cho et al., 2014) with $d = 32$ hidden units (state space $\mathbb{R}^{32}$). Each of our models (§6) always specifies the logprob-so-far in equations (2) and (3) using a 1-layer left-to-right GRU,[19] while the corresponding proposal distribution (§3.3) always specifies the state $\bar{\mathbf{s}}_t$ in (6) using a 2-layer right-to-left GRU, and specifies the compatibility function C_t in (23) using a 4-layer feedforward ReLU network.[20] For the Chinese social media NER task (§6.1), we use the Chinese character embeddings provided by Peng and Dredze (2015), while for the source separation tasks (§6.2), we use the 50-dimensional GloVe word embeddings (Pennington et al., 2014). In other cases, we train embeddings along with the rest of the network. We optimize with the Adam optimizer using the default parameters (Kingma and Ba, 2015) and L_2 regularization coefficient of 10^{-5}.

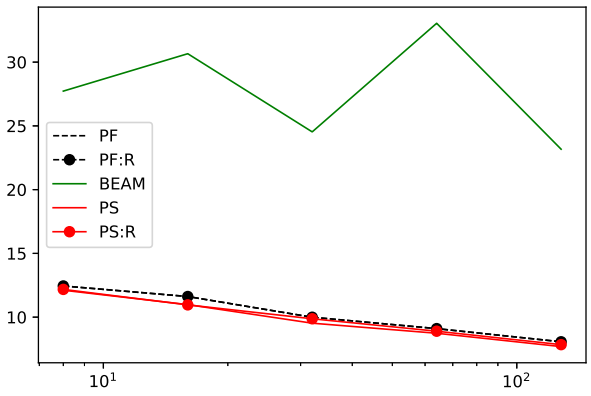


Figure 3: Offset KL divergence for the source separation task on phoneme sequences.

D Training Procedures

In all our experiments, we train the incremental scoring models (the tagging and source separation models described in §6.1 and §6.2, respectively) on the training dataset T. We do early stopping, using perplexity on a held-out development set D_1 to choose the number of epochs to train (maximum of 3).

Having obtained these model parameters θ, we train our proposal distributions $q_{\theta,\phi}$ on T, keeping θ fixed and only tuning ϕ. Again we use early stopping, using the KL divergence from §7.1 on a separate development set D_2 to choose the number of epochs to train (maximum of 20 for the two tagging tasks and source separation on the PTB dataset, and maximum of 50 for source separation on the phoneme sequence dataset). We then evaluate q_{θ^*,ϕ^*} on the test dataset E.

[Appendices E–G appear in the supplementary material file.]

[18] We thank Hongyuan Mei, Tim Vieira, and Sanjeev Khudanpur for insightful discussions on this derivation.

[19] For the tagging task described in §6.1, $g_\theta(\mathbf{s}_{t-1}, x_t, y_t) \stackrel{\text{def}}{=} \log p_\theta(x_t, y_t \mid \mathbf{s}_{t-1})$, where the GRU state $\mathbf{s}_{t-1}$ is used to define a softmax distribution over possible (x_t, y_t) pairs in the same manner as an RNN language model (Mikolov et al., 2010). Likewise, for the source separation task (§6.2), the source language models described in Appendix G are GRU-based RNN language models.

[20] As input to C_t, we actually provide not only $\mathbf{s}_t, \bar{\mathbf{s}}_t$ but also the states $f_\theta(\mathbf{s}_{t-1}, x_t, y)$ (including $\mathbf{s}_t$) that could have been reached for *each* possible value y of y_t. We have to compute these anyway while constructing the proposal distribution, and we find that it helps performance to include them.

E Applications of Sampling

In this paper, we evaluate our sampling algorithms "intrinsically" by how well a sample approximates the model distribution p_θ—rather than "extrinsically" by using the samples in some larger method.

That said, §1.3 did list some larger methods that make use of sampling. We review them here for the interested reader.

Minimum-risk decoding seeks the output

$$\operatorname*{argmin}_{\mathbf{z}} \sum_{\mathbf{y}} p_\theta(\mathbf{y} \mid \mathbf{x}) \cdot \text{loss}(\mathbf{z} \mid \mathbf{y}) \tag{47}$$

In the special case where $\text{loss}(\mathbf{z} \mid \mathbf{y})$ simply asks whether $\mathbf{z} \neq \mathbf{y}$, this simply returns the "Viterbi" sequence $\mathbf{y}$ that maximises $p_\theta(\mathbf{y} \mid \mathbf{x})$. However, it may give a different answer if the loss function gives partial credit (when $\mathbf{z} \approx \mathbf{y}$), or if the space of outputs $\mathbf{z}$ is simply coarser than the space of taggings $\mathbf{y}$—for example, if there are many action sequences $\mathbf{y}$ that could build the same output structure $\mathbf{z}$. In these cases, the optimal $\mathbf{z}$ may win due to the combined support of many suboptimal $\mathbf{y}$ values, and so finding the optimal $\mathbf{y}$ (the Viterbi sequence) is not enough to determine the optimal $\mathbf{z}$.

The risk objective (47) is a expensive expectation under the distribution $p_\theta(\mathbf{y} \mid \mathbf{x})$. To approximate it, one can replace $p_\theta(\mathbf{y} \mid \mathbf{x})$ with an approximation $\hat{p}(\mathbf{y})$ that has small support so that the summation is efficient. Particle smoothing returns such a $\hat{p}$—a non-uniform distribution (28) over M particles. Since those particles are randomly drawn, $\hat{p}$ is itself stochastic, but $\mathbb{E}\left[\hat{p}(\mathbf{y})\right] \approx p_\theta(\mathbf{y} \mid \mathbf{x})$, with the approximation improving with the quality of the proposal distribution (which is the focus of this paper) and with M.

In *supervised* training of the model (1) by maximizing conditional log-likelihood, the gradient of $\log p(\mathbf{y}^* \mid \mathbf{x})$ on a single training example $(\mathbf{x}, \mathbf{y}^*)$ is $\nabla_\theta \log p_\theta(\mathbf{y}^* \mid \mathbf{x}) = \nabla_\theta G_T^* - \sum_{\mathbf{y}} p_\theta(\mathbf{y} \mid \mathbf{x}) \cdot \nabla_\theta G_T$. The sum is again an expectation that can be estimated by using $\hat{p}$. Since $\mathbb{E}\left[\hat{p}(\mathbf{y})\right] \approx p_\theta(\mathbf{y} \mid \mathbf{x})$, this yields a stochastic estimate of the gradient that can be used in the stochastic gradient ascent algorithm (Robbins and Monro, 1951).[21]

In *unsupervised or semi-supervised training* of a generative model $p_\theta(\mathbf{x}, \mathbf{y})$, one has some training examples where $\mathbf{y}^*$ is unobserved or observed incompletely (e.g., perhaps only $\mathbf{z}$ is observed). The Monte Carlo EM algorithm for estimating θ (Wei and Tanner, 1990) replaces the missing $\mathbf{y}^*$ with samples from $p_\theta(\mathbf{y} \mid \mathbf{x}, \text{partial observation})$ (this is the Monte Carlo "E step"). This *multiple imputation* procedure has other uses as well in statistical analysis with missing data (Little and Rubin, 1987).

Modular architectures provide another use for sampling. If $p_\theta(\mathbf{y} \mid \mathbf{x})$ is just one stage in an NLP annotation *pipeline*, Finkel et al. (2006) recommend passing a diverse sample of $\mathbf{y}$ values on to the next stage, where they can be further annotated and rescored or rejected. More generally, in a *graphical model* that relates multiple strings (Bouchard-Côté et al., 2007; Dreyer and Eisner, 2009; Cotterell et al., 2017), inference could be performed by particle belief propagation (Ihler and McAllester, 2009; Lienart et al., 2015), or with the help of stochastic-inverse proposal distributions (Stuhlmüller et al., 2013). These methods call conditional sampling as a subroutine.

[21] Notice that the gradient takes this "difficult" form only because the model is globally normalized. If we were training a locally normalized conditional model (McCallum et al., 2000), or a locally normalized joint model like equation (4), then sampling methods would not be needed, because the gradient of the (conditional or joint) log-likelihood would decompose into T "easy" summands that each involve an expectation over the small set of y_t values for some t, rather than over the exponentially larger set of strings $\mathbf{y}$. However, this simplification goes away outside the fully supervised case, as the next paragraph discusses.

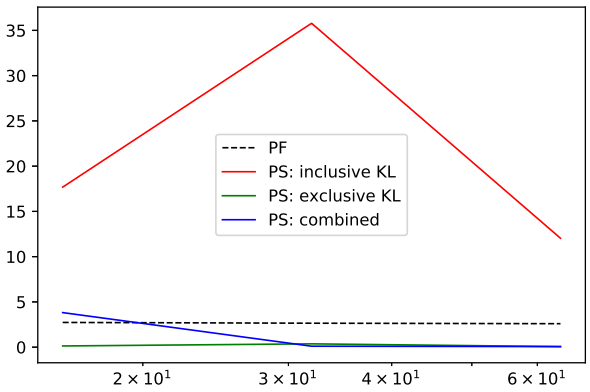


Figure 4: Offset KL divergence on the *last char* task: a pathological case where a naive particle filtering sampler does really horribly, and an ill-trained smoothing sampler even worse. The logarithmic x-axis is the particle size used to train the sampler. At test time we evaluate with the same particle size ($M = 32$).

F Effect of different objective functions on lookahead optimization

§5 discussed inclusive and exclusive KL divergences, and gave our rationale for optimizing an interpolation of the two. Here we study the effect of the interpolation weight. We train the lookahead sampler, and the joint language model, on a toy problem called "last char," where $\mathbf{y}$ is a deterministic function of $\mathbf{x}$: either a lowercased version of $\mathbf{x}$, or an identical copy of $\mathbf{x}$, depending on whether the last character of $\mathbf{x}$ is `0` or `1`. Note that this problem requires lookahead.

We obtain our $\mathbf{x}$ sequences by taking the phoneme sequence data from the stressed syllable tagging task and flipping a fair coin to decide whether to append `0` or `1` to each sequence. Thus, the dataset may include $(\mathbf{x}, \mathbf{y})$ pairs such as (`K AU CH 0`, `k au ch 1`) or (`K AU CH 1`, `K AU CH 1`), but not (`K AU CH 1`, `k au ch 1`).

We treat this as a tagging problem, and treat it with our tagging model in §6.1. Results are in Figure 4. We see that optimizing for $\mathrm{KL}(\hat{p}||q)$ at a low particle size gives much worse performance than other methods. On the other hand, the objective function $\mathrm{KL}(q||p)$ achieves constantly good performance. The middle ground $\frac{\mathrm{KL}(\hat{p}||q)+\mathrm{KL}(q||p)}{2}$ improves when the particle size increases, and achieves better results than $\mathrm{KL}(q||p)$ at larger particle sizes.

G Generative process for source separation

Given an alphabet Σ, J strings $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(J)} \in \Sigma^*$ are independently sampled from the respective distributions $p^{(1)}, \ldots p^{(J)}$ over Σ^* (possibly all the same distribution $p^{(1)} = \cdots = p^{(J)}$). These source strings are then combined into a single observed string $\mathbf{x}$, of length $K = \sum_j K_j$, according to an **interleaving string** $\mathbf{y}$, also of length K. For example, $\mathbf{y} = 1132123$ means to take two characters from $\mathbf{x}^{(1)}$, then a character from $\mathbf{x}^{(3)}$, then a character from $\mathbf{x}^{(2)}$, etc. Formally speaking, $\mathbf{y}$ is an element of the mix language $\mathcal{Y}_{\mathbf{x}} = \textsc{mix}(1^{k_1}, 2^{k_2}, \ldots, j^{k_j})$, and we construct $\mathbf{x}$ by specifying the character $x_k \in \Sigma$ to be $x^{(y_k)}_{|\{i \le k : y_i = y_k\}|}$. We assume that $\mathbf{y}$ is drawn from some distribution over $\mathcal{Y}_{\mathbf{x}}$. The source separation problem is to recover the interleaving string $\mathbf{y}$ from the interleaved string $\mathbf{x}$.

We assume that each source model $p^{(j)}(\mathbf{x}^{(j)})$ is an RNN language model—that is, a locally normalized state machine that successively generates each character of $\mathbf{x}^{(j)}$ given its left context. Thus, each source model is in some state $\mathbf{s}^{(j)}_t$ after generating the prefix $\mathbf{x}^{(j)}_{:t}$. In the remainder of this paragraph, we suppress the superscript $^{(j)}$ for simplicity. The model now stochastically generates character x_{t+1} with probability $p(x_{t+1} \mid \mathbf{s}_t)$, and from $\mathbf{s}_t$ and this x_{t+1} it deterministically computes its new state $\mathbf{s}_{t+1}$. If x_{t+1} is a special "end-of-sequence" character EOS, we return $\mathbf{x} = \mathbf{x}_{:t}$.

Given only $\mathbf{x}$ of length T, we see that $\mathbf{y}$ could be any element of $\{1, 2, \ldots, J\}^T$. We can write the posterior probability of a given $\mathbf{y}$ (by Bayes' Theorem) as

$$p(\mathbf{y} \mid \mathbf{x}) \propto p(\mathbf{y}) \prod_{j=1}^{J} p^{(j)}\left(\mathbf{x}^{(j)}\right) \quad (48)$$

where (for this given $\mathbf{y}$) $\mathbf{x}^{(j)}$ denotes the subsequence of $\mathbf{x}$ at indices k such that $y_k = j$. In our experiments, we assume that y was drawn uniformly from $\mathcal{Y}_{\mathbf{x}}$, so $p(\mathbf{y})$ is constant and can be ignored. In general, the set of possible interleavings $\mathcal{Y}_{\mathbf{x}}$ is so large that computing the constant of proportionality (partition function) for a given $\mathbf{x}$ becomes prohibitive.