An Empirical Investigation of Beam-Aware Training in Supertagging

Renato Negrinho[1] **Matthew R. Gormley**[1] **Geoffrey J. Gordon**[1,2]
Carnegie Mellon University[1], MSR Montreal[2]
{negrinho,mgormley,ggordon}@cs.cmu.edu

Abstract

Structured prediction is often approached by training a locally normalized model with maximum likelihood and decoding approximately with beam search. This approach leads to mismatches as, during training, the model is not exposed to its mistakes and does not use beam search. Beam-aware training aims to address these problems, but unfortunately, it is not yet widely used due to a lack of understanding about how it impacts performance, when it is most useful, and whether it is stable. Recently, Negrinho et al. (2018) proposed a meta-algorithm that captures beam-aware training algorithms and suggests new ones, but unfortunately did not provide empirical results. In this paper, we begin an empirical investigation: we train the supertagging model of Vaswani et al. (2016) and a simpler model with instantiations of the meta-algorithm. We explore the influence of various design choices and make recommendations for choosing them. We observe that beam-aware training improves performance for both models, with large improvements for the simpler model which must effectively manage uncertainty during decoding. Our results suggest that a model must be learned with search to maximize its effectiveness.

1 Introduction

Structured prediction often relies on models that train on maximum likelihood and use beam search for approximate decoding. This procedure leads to two significant mismatches between the training and testing settings: the model is trained on oracle trajectories and therefore does not learn about its own mistakes; the model is trained without beam search and therefore does not learn how to use the beam effectively for search.

Previous algorithms have addressed one or the other of these mismatches. For example, DAgger (Ross et al., 2011) and scheduled sampling (Bengio et al., 2015) use the learned model to visit non-oracle states at training time, but do not use beam search (i.e., they keep a single hypothesis). Early update (Collins and Roark, 2004), LaSO (Daumé and Marcu, 2005), and BSO (Wiseman and Rush, 2016) are trained with beam search, but do not expose the model to beams without a gold hypothesis (i.e., they either stop or reset to beams with a gold hypothesis).

Recently, Negrinho et al. (2018) proposed a meta-algorithm that instantiates beam-aware algorithms as a result of choices for the surrogate loss (i.e., which training loss to incur at each visited beam) and data collection strategy (i.e., which beams to visit during training). A specific instantiation of their meta-algorithm addresses both mismatches by relying on an insight on how to induce training losses for beams without the gold hypothesis: *for any beam*, its lowest cost neighbor should be scored sufficiently high to be kept in the successor beam. To induce these training losses it is sufficient to be able to compute the best neighbor of any state (often called a dynamic oracle (Goldberg and Nivre, 2012)). Unfortunately, Negrinho et al. (2018) do not provide empirical results, leaving open questions such as whether instances can be trained robustly, when is beam-aware training most useful, and what is the impact on performance of the design choices.

Contributions We empirically study beam-aware algorithms instantiated through the meta-algorithm of Negrinho et al. (2018). We tackle supertagging as it is a sequence labelling task with an easy-to-compute dynamic oracle and a moderately-sized label set (approximately 1000) which may require more effective search. We examine two supertagging models (one from Vaswani et al. (2016) and a simplified version designed to be heavily

Findings of the Association for Computational Linguistics: EMNLP 2020, pages 4534–4542
November 16 - 20, 2020. ©2020 Association for Computational Linguistics

reliant on search) and train them with instantiations of the meta-algorithm. We explore how design choices influence performance, and give recommendations based on our empirical findings. For example, we find that perceptron losses perform consistently worse than margin and log losses. We observe that beam-aware training can have a large impact on performance, particularly when the model must use the beam to manage uncertainty during prediction. Code for reproducing all results in this paper is available at https://github.com/negrinho/beam_learn_supertagging.

2 Background on learning to search and beam-aware methods

For convenience, we reuse notation introduced in Negrinho et al. (2018) to describe their meta-algorithm and its components (e.g., scoring function, surrogate loss, and data collection strategy). See Figure 1 and Figure 2 for an overview of the notation. When relevant, we instantiate notation for left-to-right sequence labelling under the Hamming cost, which supertagging is a special case of.

Input and output spaces Given an input structure $x \in \mathcal{X}$, the output structure $y \in \mathcal{Y}_x$, is generated through a sequence of incremental decisions. An example $x \in \mathcal{X}$ induces a tree $G_x = (V_x, E_x)$ encoding the sequential generation of elements in $\mathcal{Y}_x$, where V_x is the set of nodes and E_x is the set of edges. The leaves of G_x correspond to elements of $\mathcal{Y}_x$ and the internal nodes correspond to incomplete outputs. For left-to-right sequence labelling, for a sequence $x \in \mathcal{X}$, each decision assigns a label to the current position of x and the nodes of tree encode labelled prefixes of x, with terminal nodes encoding complete labellings of x.

Cost functions Given a golden pair $(x, y) \in \mathcal{X} \times \mathcal{Y}$, the cost function $c_{x,y} : \mathcal{Y}_x \to \mathbb{R}$ measures how bad the prediction $\hat{y} \in \mathcal{Y}_x$ is relative to the target output structure $y \in \mathcal{Y}_x$. Using $c_{x,y} : \mathcal{Y}_x \to \mathbb{R}$, we define a cost function $c^*_{x,y} : V_x \to \mathbb{R}$ for partial outputs by assigning to each node $v \in V_x$ the cost of its best reachable complete output, i.e., $c^*_{x,y}(v) = \min_{v' \in T_v} c_{x,y}(v')$, where $T_v \subseteq \mathcal{Y}_x$ is the set of complete outputs reachable from v. For a left-to-right search space for sequence labelling, if $c_{x,y} : \mathcal{Y}_x \to \mathbb{R}$ is Hamming cost, the optimal completion cost $c^*_{x,y} : \mathcal{Y}_x \to \mathbb{R}$ is the number of mistakes in the prefix as the optimal completion matches the remaining suffix of the target output.

Dynamic oracles An oracle state is one for which the target output structure can be reached. Often optimal actions can only be computed for oracle states. Dynamic oracles compute optimal actions even for non-oracle states. Evaluations of $c^*_{x,y} : V_x \to \mathbb{R}$ for arbitrary states allows us to induce the dynamic oracle—at a state $v \in V_x$, the optimal action is to transition to the neighbor $v' \in N_v$ with the lowest completion cost. For sequence labelling, this picks the transition that assigns the correct label. For other tasks and metrics, more complex dynamic oracles may exist, e.g., in dependency parsing (Goldberg and Nivre, 2012, 2013). For notational brevity, from now on, we omit the dependency of the search spaces and cost function on $x \in \mathcal{X}$, $y \in \mathcal{Y}$, or both.

Beam search space Given a search space $G = (V, E)$, the beam search space $G_k = (V_k, E_k)$ is induced by choosing a beam size $k \in \mathbb{N}$ and a strategy for generating the successor beam out of the current beam and its neighbors. In this paper, we expand all the elements in the beam and score the neighbors simultaneously. The highest scoring k neighbors are used to form the successor beam. For $k = 1$, we recover the greedy search space G.

Beam cost functions The natural cost function $c^* : V_k \to \mathbb{R}$ for G_k is created from the element-wise cost function on G, and assigns to each beam the cost of its best element, i.e., for $b \in V_k$, $c^*(b) = \min_{v \in b} c^*(v)$. For a transition $(b, b') \in E_k$, we define the transition cost $c(b, b') = c^*(b') - c^*(b)$, where $b' \in N_b$, i.e., b' can be formed from the neighbors of the elements in b. A cost increase happens when $c(b, b') > 0$, i.e., the best complete output reachable in b is no longer reachable in b'.

Policies Policies operate in beam search space G_k and are induced through a learned scoring function $s(\cdot, \theta) : V \to \mathbb{R}$ which scores elements in the original space G. A policy $\pi : V_k \to \Delta(V_k)$, i.e., mapping states (i.e., beams) to distributions over next states. We only use deterministic policies where the successor beam is computed by sorting the neighbors in decreasing order of score and taking the top k.

Scoring function In the non-beam-aware case, the scoring function arises from the way probabilities of complete sequences are computed with the locally normalized model, namely $p(y|x, \theta) = \prod_{j=1}^{h} p(y_i|y_{1:i-1}, x, \theta)$, where we assume that all

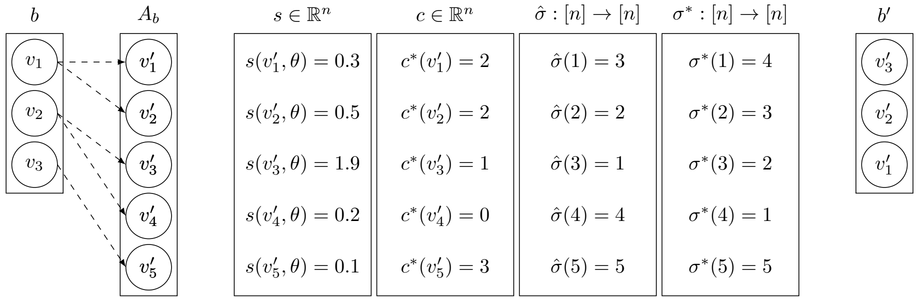

Figure 1: Beam b has neighborhood A_b, where $k = |b| = |b'| = 3$ and $n = |A_b| = 5$. Edges from elements in b to elements in A_b encode neighborhood relationships, e.g., v_3 has a single neighbor v'_5. Permutation $\hat{\sigma} : [n] \to [n]$ sorts hypotheses in decreasing order of score, and permutation $\sigma^* : [n] \to [n]$ sorts them in increasing order of cost, i.e, $v'_{\sigma^*(1)}$ is the lowest cost neighbor and $v'_{\hat{\sigma}(1)}$ is the highest scoring neighbor. The successor beam b' keeps the neighbor states in A_b with highest score according to vector s, or equivalently highest rank according to $\hat{\sigma}$.

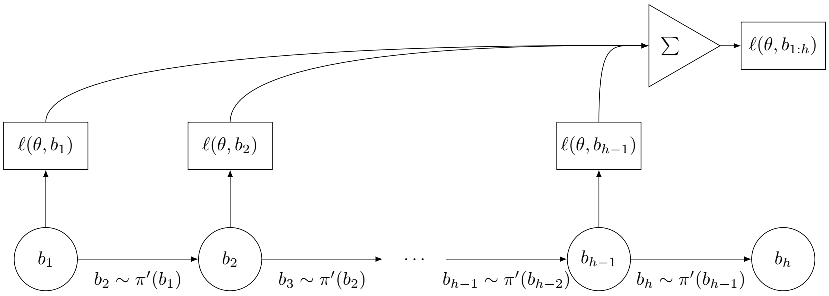

Figure 2: Sampling a trajectory through the beam search space at training time. A loss $\ell(b_i, \theta)$ is incurred at each visited beam b_i, $i \in [h-1]$, resulting in total accumulated loss $\ell(b_{1:h}, \theta)$ for beam trajectory $b_{1:h}$. The terminal beam b_h corresponds to a complete output $y(b_h) \in \mathcal{Y}$. Transitions between beams are sampled according to a data collection policy $\pi' : V_k \to \Delta(V_k)$. We consider π' induced by a scoring function $s(\cdot, \theta) : V \to \mathbb{R}$ or cost function $c^* : V \to \mathbb{R}$. Parameters θ parametrize the scoring function of the model. Losses $\ell(b_i, \theta)$ are low if the scores of the neighbors of b_i comfortably keep the lowest cost elements in the successor beam (see Section 3.2), and high otherwise. See Figure 1 for the notation to describe the surrogate loss $\ell(b_i, \theta)$ at each beam b_i.

outputs for $x \in \mathcal{X}$ have h steps. For sequence labelling, h is the length of the sentence. The resulting scoring function $s(\cdot, \theta) : V \to \mathbb{R}$ is $s(v, \theta) = \sum_{i=1}^{j} \log p(y_i | y_{1:i-1}, x, \theta)$, where $v = y_{1:j}$ and $j \in [h]$. Similarly, the scoring function that we learn in the beam-aware case is $s(v, \theta) = \sum_{i=1}^{j} \tilde{s}(v, \theta)$, where x has been omitted, $v = y_{1:j}$, and $\tilde{s}(\cdot, \theta) : V \to \mathbb{R}$ is the learned incremental scoring function. In Section 4.6, we observe that this cumulative version performs uniformly better than the non-cumulative one.

3 Meta-algorithm for learning beam search policies

We refer the reader to Negrinho et al. (2018) for a discussion of how specific choices for the meta-algorithm recover algorithms from the literature.

3.1 Data collection strategies

The data collection strategy determines which beams are visited at training time (see Figure 2). Strategies that use the learned model differ on how they compute the successor beam $b' \in N_b$ when $s(\cdot, \theta)$ leads to a beam without the gold hypothesis, i.e., $c(b, b') > 0$, where $b' = \{v_{\hat{\sigma}(1)}, \ldots, v_{\hat{\sigma}(k)}\} \subset A_b$ and $A_b = \{v_1, \ldots, v_n\} = \cup_{v \in b} N_v$. We explore several data collection strategies:

stop If the successor beam does not contain the gold hypothesis, stop collecting the trajectory. Structured perceptron training with early update (Collins and Roark, 2004) use this strategy.

reset If the successor beam does not contain the gold hypothesis, reset to a beam with only the gold hypothesis[1]. LaSO (Daumé and Marcu, 2005) use this strategy. For $k = 1$, we recover teacher forcing as only the oracle hypothesis is kept in the beam.

[1] Any beam with the gold hypothesis would be valid, e.g., the top $k-1$ according to the scores plus the gold hypothesis, which we call *reset (multiple)*

continue Ignore cost increases, always using the successor beam. DAgger (Ross et al., 2011) take this strategy, but does not use beam search. Negrinho et al. (2018) suggest this for beam-aware training but do not provide empirical results.

reset (multiple) Similar to reset, but keep $k-1$ hypothesis from the transition, i.e., $b' = \{v_{\sigma^*(1)}, v_{\hat{\sigma}(1)}, \ldots v_{\hat{\sigma}(k-1)}\}$. We might expect this data collection strategy to be closer to *continue* as a large fraction of the elements of the successor beam are induced by the learned model.

oracle Always transition to the beam induced by $\sigma^* : [n] \to [n]$, i.e., the one obtained by sorting the costs in increasing order. For $k = 1$, this recovers teacher forcing. In Section 4.4, we observe that *oracle* dramatically degrades performance due to increased exposure bias with increased k.

3.2 Surrogate losses

Surrogate losses encode that the scores produced by the model for the neighbors must score the best neighbor sufficiently high for it to be kept comfortably in the successor beam. For $k = 1$, many of these losses reduce to losses used in non-beam-aware training. Given scores $s \in \mathbb{R}^n$ and costs $c \in \mathbb{R}^n$ over neighbors in $A_b = \{v_1, \ldots, v_n\}$, we define permutations $\hat{\sigma} : [n] \to [n]$ and $\sigma^* : [n] \to [n]$ that sort the elements in A_b in decreasing order of scores and increasing order of costs, respectively, i.e., $s_{\hat{\sigma}(1)} \geq \ldots \geq s_{\hat{\sigma}(n)}$ and $c_{\sigma^*(1)} \leq \ldots \leq s_{\sigma^*(n)}$. See Figure 1 for a description of the notation used to describe surrogate losses. Our experiments compare the following surrogate losses:

perceptron (first) Penalize failing to score the best neighbor at the top of the beam (regardless of it falling out of the beam or not).

$$\ell(s, c) = \max\left(0, s_{\hat{\sigma}(1)} - s_{\sigma^*(1)}\right).$$

perceptron (last) If this loss is positive at a beam, the successor beam induced by the scores does not contain the golden hypothesis.

$$\ell(s, c) = \max\left(0, s_{\hat{\sigma}(k)} - s_{\sigma^*(1)}\right).$$

margin (last) Penalize margin violations of the best neighbor of the hypothesis in the current beam. Compares the correct neighbor $s_{\sigma^*(1)}$ with the neighbor $v_{\hat{\sigma}(k)}$ last in the beam.

$$\ell(s, c) = \max\left(0, s_{\hat{\sigma}(k)} - s_{\sigma^*(1)} + 1\right)$$

cost-sensitive margin (last) Same as *margin (last)* but weighted by the cost difference of the pair. Wiseman and Rush (2016) use this loss.

$$\ell(s, c) = c_{\hat{\sigma}(k), \sigma^*(1)} \max(0, s_{\hat{\sigma}(k)} - s_{\sigma^*(1)} + 1),$$

where $c_{\hat{\sigma}(k), \sigma^*(1)} = c_{\hat{\sigma}(k)} - c_{\sigma^*(1)}$.

log loss (neighbors) Normalizes over all elements in A_b. For beam size $k = 1$, it reduces to the usual log loss.

$$\ell(s, c) = -s_{\sigma^*(1)} + \log\left(\sum_{i=1}^{n} \exp(s_i)\right)$$

log loss (beam) Normalizes only over the top k neighbors of a beam according to the scores s.

$$\ell(s, c) = -s_{\sigma^*(1)} + \log\left(\sum_{i \in I} \exp(s_i)\right),$$

where $I = \{\sigma^*(1), \hat{\sigma}(1), \ldots, \hat{\sigma}(k)\}$. The normalization is only over the golden hypothesis $v_{\sigma^*(1)}$ and the elements included in the beam. Andor et al. (2016) use this loss.

3.3 Training

The meta-algorithm of Negrinho et al. (2018) is instantiated by choosing a surrogate loss, data collection strategy, and beam size. Training proceeds by sampling an example $(x, y) \in \mathcal{X} \times \mathcal{Y}$ from the training set. A trajectory through the beam search space G_k is collected using the chosen data collection strategy. A surrogate loss is induced at each non-terminal beam in the trajectory (see Figure 2). Parameter updates are computed based on the gradient of the sum of the losses of the visited beams.

4 Experiments

We explore different configurations of the design choices of the meta-algorithm to understand their impact on training behavior and performance.

4.1 Task details

We train our models for supertagging, a sequence labelling where accuracy is the performance metric of interest. Supertagging is a good task for exploring beam-aware training, as contrary to other sequence labelling datasets such as named-entity recognition (Tjong Kim Sang and De Meulder, 2003), chunking (Sang and Buchholz, 2000), and part-of-speech tagging (Marcus et al., 1993), has a moderate number of labels and therefore it is

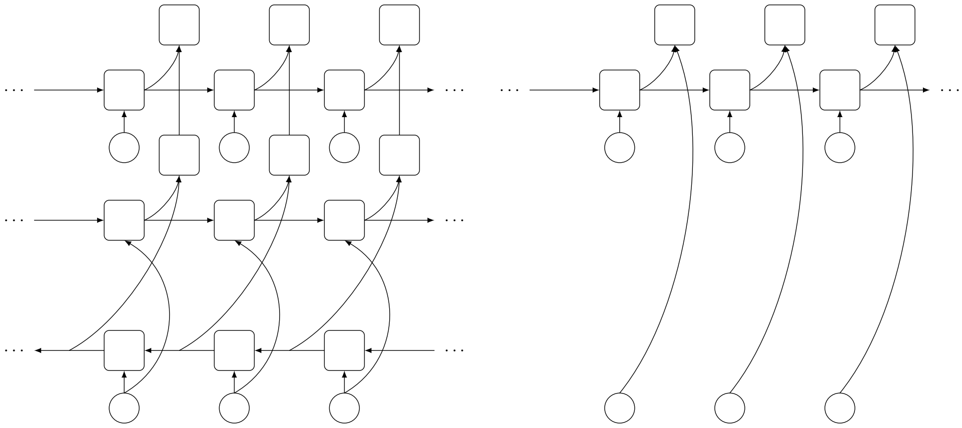

Figure 3: High-level structure of the two models used in the experiments. The model on the left is from Vaswani et al. (2016). The model on the right is a simplification of the one on the left, namely, it does not have an encoding of the complete sentence at the start of prediction.

likely to require effective search to achieve high performances. We used the standard splits for CCGBank (Hockenmaier and Steedman, 2007): the training and development sets have, respectively, 39604 and 1913 examples. Models were trained on the training set and used the development set to compute validation accuracy at the end of each epoch to keep the best model. As we are performing an empirical study, similarly to Vaswani et al. (2016), we report validation accuracies. Each configuration is ran three times with different random seeds and the mean and standard deviation are reported. We replace the words that appear at most once in the training set by `UNK`. By contrast, no tokenization was done for the training supertags.

4.2 Model details

We have implemented the model of Vaswani et al. (2016) and a simpler model designed by removing some of its components. The two main differences between our implementation and theirs are that we do not use pretrained embeddings (we train the embeddings from scratch) and we use the gold POS tags (they use only the pretrained embeddings).

Main model For the model of Vaswani et al. (2016) (see Figure 3, left), we use 64, 16, and 64 for the dimensions of the word, part-of-speech, and supertag embeddings, respectively. All LSTMs (forward, backward, and LM) have hidden dimension 256. We refer the reader to Vaswani et al. (2016) for the exact description of the model. Briefly, embeddings for the words and part-of-speech tags are concatenated and fed to a bi-directional LSTM, the outputs of both directions are then fed into a combiner (dimension-preserving linear transformations applied independently to both inputs, added together, and passed through a ReLU non-linearity). The output of the combiner and the output of the LM LSTM (which tracks the supertag prefix up to a prediction point) is then passed to another combiner that generates scores over supertags.

Simplified model We also consider a simplified model that drops the bi-LSTM encoder and the corresponding combiner (see Figure 3, right). The concatenated embeddings are fed directly into the second combiner with the LM LSTM output. Values for the hyperparameters are the same when possible. This model must leverage the beam effectively as it does not encode the sentence with a bi-LSTM. Instead, only the embeddings for the current position are available, giving a larger role to the LM LSTM over supertags. While supertagging can be tackled with a stronger model, this restriction is relevant for real-time tasks, e.g., the complete input might not be known upfront.

Training details Models are trained for 16 epochs with SGD with batch size 1 and cosine learning rate schedule (Loshchilov and Hutter, 2016), starting at 10^{-1} and ending at 10^{-5}. No weight decay or dropout was used. Training examples are shuffled after each epoch. Results are reported for the model with the best validation performance across all epochs. We use 16 epochs for all models for simplicity and fairness. This number was sufficient, e.g., we replicated Table 2 by training with 32 epochs and observed minor performance differences (see Table 6).

	1	2	4	8
oracle/reset	$93.78_{0.12}$	$93.81_{0.11}$	$93.82_{0.10}$	$93.82_{0.10}$
continue	$94.04_{0.07}$	$94.05_{0.07}$	$94.05_{0.07}$	$94.06_{0.07}$
stop	$93.86_{0.09}$	$93.90_{0.07}$	$93.90_{0.07}$	$93.91_{0.07}$
oracle/reset	$73.20_{0.31}$	$76.55_{0.24}$	$77.42_{0.27}$	$77.54_{0.22}$
continue	$81.99_{0.04}$	$82.30_{0.03}$	$82.37_{0.08}$	$82.41_{0.08}$
stop	$74.35_{0.23}$	$77.06_{0.14}$	$77.73_{0.13}$	$77.82_{0.09}$

Table 1: Development accuracies for models trained with different data collection strategies in a non-beam-aware way (i.e., $k = 1$) and decoded with beam search with varying beam size. *continue* performs best, showing the importance of exposing the model to its mistakes. Differences are larger for the simplified model.

	1	2	4	8
oracle	$94.10_{0.08}$	$92.98_{0.07}$	$91.66_{0.22}$	$85.95_{0.79}$
reset	$94.20_{0.11}$	$94.34_{0.06}$	$94.33_{0.01}$	$94.42_{0.04}$
reset (mult.)	$94.15_{0.07}$	$93.98_{0.08}$	$94.06_{0.06}$	$94.16_{0.05}$
continue	$94.15_{0.02}$	$94.35_{0.05}$	$94.37_{0.04}$	$94.33_{0.04}$
stop	$93.95_{0.09}$	$94.11_{0.05}$	$94.24_{0.07}$	$94.25_{0.06}$
oracle	$75.09_{0.17}$	$80.67_{0.40}$	$78.69_{1.27}$	$47.38_{1.79}$
reset	$75.06_{0.16}$	$87.21_{0.14}$	$91.24_{0.02}$	$92.46_{0.09}$
reset (mult.)	$75.04_{0.18}$	$86.19_{0.12}$	$90.76_{0.11}$	$92.16_{0.03}$
continue	$82.01_{0.06}$	$89.17_{0.08}$	$91.80_{0.12}$	$92.69_{0.01}$
stop	$75.08_{0.54}$	$87.16_{0.08}$	$90.98_{0.13}$	$92.18_{0.06}$

Table 2: Development accuracies for beam-aware training with varying data collection strategies.

4.3 Non-beam-aware training

We first train the models with $k = 1$ and then use beam search to decode. Crucially, the model does not train with a beam and therefore does not learn to use it effectively. We vary the data collection strategy. The results are presented in Table 1 and should be used as a reference when reading the other tables to evaluate the impact of beam-aware training. Tables are formatted such that the first and second horizontal halves contain the results for the main model and simplified model, respectively. Each position contains the mean and the standard deviation of running that configuration three times. We use this format in all tables presented.

The *continue* data collection strategy (i.e., DAgger for $k = 1$) results in better models than training on the oracle trajectories. Beam search results in small gains for these settings. In this experiment, training with oracle is the same as training with reset as the beam always contains only the oracle hypothesis. The performance differences are small for the main model but much larger for the simplified model, underscoring the importance of beam search when there is greater uncertainty about predictions. For the stronger model, the encoding of the left and right contexts with the bi-LSTM provides enough information at each position to predict greedily, i.e., without search. This difference appears consistently in all experiments, with larger gains for the weaker model.

The gains achieved by the main model by decoding with beam search post-training are very small (from 0.02 to 0.05). This suggests that training the model in a non-beam-aware fashion and then using beam search does not guarantee improvements. The model must be learned with search to improve on these results. For the simpler model, larger improvements are observed (from 0.42 to 4.34). Despite the gains with beam search for *reset* and *stop*, they are not sufficient to beat the greedy model trained on its own trajectories, yielding 81.99 for *continue* with $k = 1$ versus 77.54 for *oracle* and 77.82 for *reset*, both with $k = 8$. These results show the importance of the data collection strategy, even when the model is not trained in a beam-aware fashion. These gains are eclipsed by beam-aware training, namely, compare Table 1 with Table 2. See Figure 4 for the evolution of the validation and training accuracies with epochs.

4.4 Comparing data collection strategies

We train both models using the *log loss (neighbors)*, described in Section 3.2, and vary the data collection strategy, described in Section 3.1, and beam size. Results are presented in Table 2 Contrary to Section 4.3, these models are trained to use beam search rather than it being an artifact of approximate decoding. Beam-aware training under *oracle* worsens performance with increasing beam size (due to increasing exposure bias). During training, the model learns to pick the best neighbors for beams containing only close to optimal hypotheses, which are likely very different from the beams encountered when decoding. The results for the simplified model are similar—with increasing beam size, performance first improves but then degrades. For the main model, we observe modest but consistent improvements with larger beam sizes across all data collection strategies except *oracle*. By comparing the results with those in the first row of Table 1, we see that we improve on the model trained with maximum likelihood and decoded with beam search.

The data collection strategy has a larger impact on performance for the simplified model. *continue*

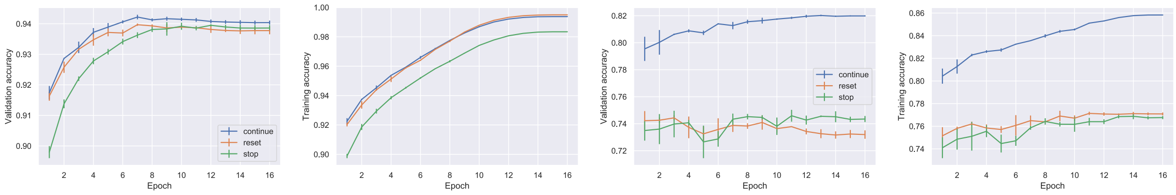

Figure 4: Validation and training accuracies for non-beam-aware training (i.e., $k = 1$) with different data collection strategies for the main (left half) and simplified (right half) models. *continue* achieves higher accuracies.

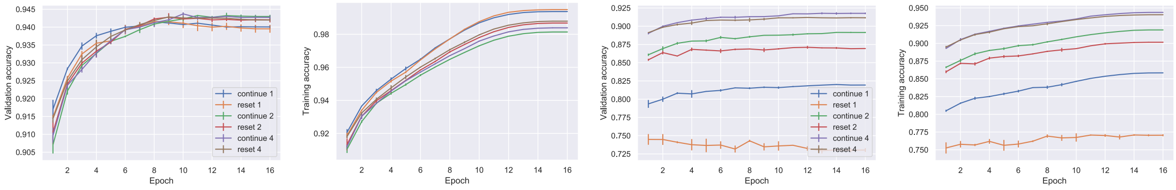

Figure 5: Validation and training accuracies for beam-aware training with different data collection strategies and beam sizes for the main (left half) and simplified (right half) models. Larger beam sizes achieve higher performances while overfitting less, and are crucial for the simplified model to achieve higher training and validation accuracies. For smaller beams *continue* performs better than *reset*. All models can be trained stably from scratch. Three runs were aggregated by showing the mean and the standard deviation for each epoch.

	1	2	4	8
percep. (first)	$92.81_{0.06}$	$93.22_{0.04}$	$93.44_{0.02}$	$93.52_{0.06}$
percep. (last)	$92.84_{0.11}$	$93.57_{0.06}$	$93.86_{0.09}$	$93.77_{0.04}$
m. (last)	$94.10_{0.07}$	$94.29_{0.07}$	$94.27_{0.03}$	$94.43_{0.04}$
cost-s. m. (last)	$93.98_{0.03}$	$94.32_{0.10}$	$94.37_{0.03}$	$94.33_{0.13}$
log loss (beam)	$92.29_{0.07}$	$92.09_{0.11}$	$94.24_{0.08}$	$94.32_{0.02}$
log loss (neig.)	$94.22_{0.00}$	$94.29_{0.03}$	$94.27_{0.06}$	$94.38_{0.01}$
percep. (first)	$77.62_{0.14}$	$86.32_{0.05}$	$89.83_{0.05}$	$91.00_{0.07}$
percep. (last)	$77.67_{0.07}$	$87.62_{0.03}$	$90.82_{0.16}$	$91.98_{0.11}$
m. (last)	$81.75_{0.04}$	$88.80_{0.02}$	$91.91_{0.05}$	$92.81_{0.05}$
cost-s. m. (last)	$81.76_{0.05}$	$88.92_{0.06}$	$91.81_{0.03}$	$92.81_{0.03}$
log loss (beam)	$77.50_{0.07}$	$88.25_{0.08}$	$91.46_{0.06}$	$92.56_{0.11}$
log loss (neig.)	$81.94_{0.02}$	$89.01_{0.10}$	$91.75_{0.03}$	$92.60_{0.03}$

Table 3: Development accuracies for the loss functions in Section 3.2.

achieves the best performance. Compare these performances with those for the simplified model in Table 1. For larger beams, the improvements achieved by beam-aware training are much larger than those achieved by non-beam-aware ones. For example, 92.69 versus 82.41 for *continue* with $k = 8$, where in the first case it is trained in a beam-aware manner ($k = 8$ for both training and decoding), while in the second case, beam search is used only during decoding ($k = 1$ during training but $k = 8$ during decoding). This shows the importance of training with beam search and exposing the model to its mistakes. Without beam-aware training, the model is unable to learn to use the beam effectively. Check Figure 5 for the evolution of the training and validation accuracies with training epoch for beam-aware training.

4.5 Comparing surrogate losses

We train both models with *continue* and vary the surrogate loss and beam size. Results are presented in Table 3.2. Perceptron losses (e.g., *perceptron (first)* and *perceptron (last)*) performed worse than their margin-based counterparts (e.g., *margin (last)* and *cost-sensitive margin (last)*). *log loss (beam)* yields poor performances for small beam sizes (e..g, $k = 1$ and $k = 2$). This is expected due to small contrastive sets (i.e., at most $k + 1$ elements are used in *log loss (beam)*). For larger beams, the results are comparable with *log loss (neighbors)*.

4.6 Additional design choices

Score accumulation The scoring function was introduced as a sum of prefix terms. A natural alternative is to produce the score for a neighbor without adding it to a running sum, i.e., $s(y_{1:j}, \theta) = \tilde{s}(y_{1:j}, \theta)$ rather than $s(y_{1:j}, \theta) = \sum_{i=1}^{j} \tilde{s}(y_{1:i}, \theta)$. Surprisingly, score accumulation performs uniformly better across all configurations. For the main model, beam-aware training degraded performance with increasing beam size. For the simplified model, beam-aware training improved on the results in Table 1, but gains were smaller than those with score accumulation. We observed that the LM LSTM failed to keep track of differences earlier in the supertag sequence, leading to similar scores over their neighbors. Accumulating the scores is a simple memory mechanism that does not require the LM LSTM to learn to propagate long-range information. This performance gap may not exist for models that access information more directly (e.g., transformers (Vaswani et al., 2017) and other attention-based models (Bahdanau et al., 2014)). See the appendix for Table 4 which compares configurations with and without score accumulation. Performance differences range from 1 to 5 absolute percentage points.

Update on all beams The meta-algorithm of Negrinho et al. (2018) suggests inducing losses on every visited beam as there is always a correct action captured by appropriately scoring the neighbors. This leads to updating the parameters on every beam. By contrast, other beam-aware work updates only on beams where the transition leads to increased cost (e.g., Daumé and Marcu (2005) and Andor et al. (2016)). We observe that always updating leads to improved performance, similar to the results in Table 3 for perceptron losses. We therefore recommend inducing losses on every visited beam. See the appendix for Table 5, which compares configurations trained with and without updating on every beam.

5 Related work

Related work uses either imitation learning (often called learning to search when applied to structured prediction) or beam-aware training. Learning to search (Daumé et al., 2009; Chang et al., 2015; Goldberg and Nivre, 2012; Bengio et al., 2015; Negrinho et al., 2018) is a popular approach for structured prediction. This literature is closely related to imitation learning (Ross and Bagnell, 2010; Ross et al., 2011; Ross and Bagnell, 2014). Ross et al. (2011) addresses exposure bias by collecting data with the learned policy at training time. Collins and Roark (2004) proposes a structured perceptron variant that trains with beam search, updating the model parameters when the correct hypothesis falls out of the beam. Huang et al. (2012) introduces a theoretical framework to analyze the convergence of early update. Zhang and Clark (2008) develops a beam-aware algorithm for dependency parsing that uses early update and dynamic oracles. Goldberg and Nivre (2012, 2013) introduce dynamic oracles for dependency parsing. Ballesteros et al. (2016) observes that exposing the model to mistakes during training improves a dependency parser. Bengio et al. (2015) makes a similar observation and present results on image captioning, constituency parsing, and speech recognition. Beam-aware training has also been used for speech recognition (Collobert et al., 2019; Baskar et al., 2019). Andor et al. (2016) proposes an early update style algorithm for learning models with a beam, but use a log loss rather than a perceptron loss as in Collins and Roark (2004). Parameters are updated when the golden hypothesis falls out of the beam or when the model terminates with the golden hypothesis in the beam. Wiseman and Rush (2016) use a similar algorithm to Andor et al. (2016) but they use a margin-based loss and reset to a beam with the golden hypothesis when it falls out of the beam. Edunov et al. (2017) use beam search to find a contrastive set to define sequence-level losses. Goyal et al. (2018, 2019) propose a beam-aware training algorithm that relies on a continuous approximation of beam search. Negrinho et al. (2018) introduces a meta-algorithm that instantiates beam-aware algorithms based on choices for beam size, surrogate loss function, and data collection strategy. They propose a DAgger-like algorithm for beam search.

6 Conclusions

Maximum likelihood training of locally normalized models with beam search decoding is the default approach for structured prediction. Unfortunately, it suffers from exposure bias and does not learn to use the beam effectively. Beam-aware training promises to address some of these issues, but is not yet widely used due to being poorly understood. In this work, we explored instantiations of the meta-algorithm of Negrinho et al. (2018) to understand how design choices affect performance. We show that beam-aware training is most useful when substantial uncertainty must be managed during prediction. We make recommendations for instantiating beam-aware algorithms based on the meta-algorithm, such as inducing losses at every beam, using log losses (rather than perceptron-style ones), and preferring the *continue* data collection strategy (or *reset* if necessary). We hope that this work provides evidence that beam-aware training can greatly impact performance and be trained stably, leading to their wider adoption.

Acknowledgements

We gratefully acknowledge support from 3M — M*Modal. This work used the Bridges system, which is supported by NSF award number ACI-

1445606, at the Pittsburgh Supercomputing Center (PSC).

References

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

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv:1409.0473*.

Miguel Ballesteros, Yoav Goldberg, Chris Dyer, and Noah A Smith. 2016. Training with exploration improves a greedy stack-LSTM parser. *arXiv:1603.03793*.

Murali Karthick Baskar, Lukáš Burget, Shinji Watanabe, Martin Karafiát, Takaaki Hori, and Jan Honza Černockỳ. 2019. Promising accurate prefix boosting for sequence-to-sequence asr. In *ICASSP*.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. *NeurIPS*.

Kai-Wei Chang, Akshay Krishnamurthy, Alekh Agarwal, Hal Daumé, and John Langford. 2015. Learning to search better than your teacher. *ICML*.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. *ACL*.

Ronan Collobert, Awni Hannun, and Gabriel Synnaeve. 2019. A fully differentiable beam search decoder. *arXiv:1902.06022*.

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

Hal Daumé and Daniel Marcu. 2005. Learning as search optimization: Approximate large margin methods for structured prediction. *ICML*.

Sergey Edunov, Myle Ott, Michael Auli, David Grangier, and Marc'Aurelio Ranzato. 2017. Classical structured prediction losses for sequence to sequence learning. *arXiv:1711.04956*.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. *COLING 2012*.

Yoav Goldberg and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *TACL*.

Kartik Goyal, Chris Dyer, and Taylor Berg-Kirkpatrick. 2019. An empirical investigation of global and local normalization for recurrent neural sequence models using a continuous relaxation to beam search. In *NAACL*.

Kartik Goyal, Graham Neubig, Chris Dyer, and Taylor Berg-Kirkpatrick. 2018. A continuous relaxation of beam search for end-to-end training of neural sequence models. *AAAI*.

Julia Hockenmaier and Mark Steedman. 2007. CCGbank: a corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*, 33(3).

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. *NAACL*.

Ilya Loshchilov and Frank Hutter. 2016. SGDR: Stochastic gradient descent with warm restarts. *arXiv:1608.03983*.

Mitchell Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*.

Renato Negrinho, Matthew Gormley, and Geoffrey J Gordon. 2018. Learning beam search policies via imitation learning. In *NeurIPS*.

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

Stéphane Ross and Drew Bagnell. 2010. Efficient reductions for imitation learning. *AISTATS*.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. *AISTATS*.

Erik F Sang and Sabine Buchholz. 2000. Introduction to the conll-2000 shared task: Chunking. *cs/0009008*.

Erik Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. *NAACL*.

Ashish Vaswani, Yonatan Bisk, Kenji Sagae, and Ryan Musa. 2016. Supertagging with LSTMs. In *ACL*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*.

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

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *EMNLP*.