SEARNN: Training RNNs with global-local losses

Rémi Leblond[*] **Jean-Baptiste Alayrac**[*] **Anton Osokin**
Département d'informatique de l'ENS
École normale supérieure, CNRS, INRIA, PSL Research University

Simon Lacoste-Julien
Department of CS & OR
Université de Montréal

Abstract

We propose SEARNN, a novel training algorithm for recurrent neural networks (RNNs) inspired by the "learning to search" (L2S) approach to structured prediction. RNNs have been widely successful in structured prediction applications such as machine translation or parsing, and are commonly trained using maximum likelihood estimation (MLE). Unfortunately, this training loss is not always an appropriate surrogate for the test error: by only maximizing the ground truth probability, it fails to exploit the wealth of information offered by structured losses. Further, it introduces discrepancies between training and predicting (such as exposure bias) that may hurt test performance. Instead, SEARNN leverages test-alike search space exploration to introduce global-local losses that are closer to the test error. We demonstrate improved performance over MLE on three different tasks: OCR, spelling correction and text chunking. Finally, we propose a subsampling strategy to enable SEARNN to scale to large vocabulary sizes.

1 Introduction

Recurrent neural networks (RNNs) have been recently quite successful in structured prediction applications such as machine translation (Sutskever et al., 2014), parsing (Ballesteros et al., 2016) or caption generation (Vinyals et al., 2015). These models use the same repeated *cell* (or unit) to output a sequence of tokens one by one. As each prediction takes into account all previous predictions, this cell learns to output the next token conditioned on the previous ones. The standard training loss for RNNs is derived from maximum likelihood estimation (MLE): we consider that the cell outputs a probability distribution at each step in the sequence, and we seek to maximize the probability of the ground truth.

Unfortunately, this training loss is not a particularly close surrogate to the various test errors we want to minimize. A striking example of discrepancy is that the MLE loss is close to 0/1: it makes no distinction between candidates that are close or far away from the ground truth (with respect to the structured test error), thus failing to exploit valuable information. Another example of train/test discrepancy is called *exposure* or *exploration bias* (Ranzato et al., 2016): in traditional MLE training the cell learns the conditional probability of the next token, based on the previous ground truth tokens – this is often referred to as *teacher forcing*. However, at test time the model does not have access to the ground truth, and thus feeds its own previous predictions to its next cell for prediction instead.

Improving RNN training thus appears as a relevant endeavor, which has received much attention recently. In particular, ideas coming from reinforcement learning (RL), such as the REINFORCE and ACTOR-CRITIC algorithms (Ranzato et al., 2016; Bahdanau et al., 2017), have been used to derive training losses that are more closely related to the test error that we actually want to minimize.

In order to address the issues of MLE training, we propose instead to use ideas from the structured prediction field, in particular from the "learning to search" (L2S) approach introduced by Daumé et al. (2009) and later refined by Ross and Bagnell (2014) and Chang et al. (2015) among others.

[*]Equal contribution.

Contributions. In Section 2, we review the limitations of MLE training for RNNs in details. We also clarify some related claims made in the recent literature. In Section 3, we make explicit the strong links between RNNs and the L2S approach. In Section 4, we present SEARNN, a novel training algorithm for RNNs, using ideas from L2S to derive a *global-local* loss that is much closer to the test error than MLE. We demonstrate that this novel approach leads to significant improvements on three difficult structured prediction tasks, including a spelling correction problem recently introduced in Bahdanau et al. (2017). As this algorithm is quite costly, in Section 5 we investigate scaling solutions. We propose a subsampling strategy that allows us to considerably reduce training times while maintaining improved performance compared to MLE. Finally, in Section 6 we contrast our novel approach to the related RL-inspired methods.

2 Traditional RNN training and its limitations

RNNs are a large family of neural network models aimed at representing sequential data. To do so, they produce a sequence of states $(h_1, ..., h_T)$ by recursively applying the same transformation (or *cell*) f on the sequential data: $h_t = f(h_{t-1}, y_{t-1}, x)$, with h_0 an initial state and x an optional input.

Many possible design choices fit this framework. We focus on a subset typically used for structured prediction, where we want to model the *joint probability* of a target sequence $(y_1, \ldots, y_{T_x}) \in \mathcal{A}^{T_x}$ given an input x (e.g. the *decoder* RNN in the encoder-decoder architecture (Sutskever et al., 2014; Cho et al., 2014)). Here $\mathcal{A}$ is the alphabet of output tokens and T_x is the length of the output sequence associated with input x (though T_x may take different values, in the following we drop the dependency in x and use T for simplicity). To achieve this modeling, we feed h_t through a projection layer (i.e. a linear classifier) to obtain a vector of scores s_t over all possible tokens $a \in \mathcal{A}$, and normalize these with a softmax layer (an exponential normalizer) to obtain a distribution o_t over tokens:

$$h_t = f(h_{t-1}, y_{t-1}, x)\,; \qquad s_t = \text{proj}(h_t)\,; \quad o_t = \text{softmax}(s_t) \qquad \forall\, 1 \leq t \leq T\,. \tag{1}$$

The vector o_t is interpreted as the predictive conditional distribution for the t^{th} token given by the RNN model, i.e. $p(a|y_1, \ldots, y_{t-1}, x) := o_t(a)$ for $a \in \mathcal{A}$. Multiplying the values $o_t(y_t)$ together thus yields the joint probability of the sequence y defined by the RNN (thanks to the chain rule):

$$p(y_1, ..., y_T|x) = p(y_1|x)p(y_2|y_1, x) \ldots p(y_T|y_1, ..., y_{T-1}, x) := \Pi_{t=1}^{T} o_t(y_t)\,. \tag{2}$$

As pointed in Goodfellow et al. (2016), the underlying structure of these RNNs as graphical models is thus a complete graph, and there is no conditional independence assumption to simplify the difficult prediction task of computing $\arg\max_{y \in \mathcal{Y}} p(y|x)$. In practice, one typically uses either beam search to approximate this decoding, or a sequence of *greedy* predictions $\hat{y}_t := \arg\max_{a \in \mathcal{A}} p(a|\hat{y}_1, \ldots, \hat{y}_{t-1}, x)$.

If we use the "teacher forcing" regimen, where the inputs to the RNN cell are the ground truth tokens (as opposed to its own greedy predictions), we obtain the probability of each ground truth sequence according to the RNN model. We can then use MLE to derive a loss to train the RNN. One should note here that despite the fact that the individual output probabilities are at the token level, the MLE loss involves the joint probability (computed via the chain rule) and is thus at the *sequence level*.

The limitations of MLE training. While this maximum likelihood style of training has been very successful in various applications, it suffers from several known issues, especially for structured prediction problems. The first one is called *exposure* or *exploration bias* (Ranzato et al., 2016). During training (with teacher forcing), the model learns the probabilities of the next tokens conditioned on the ground truth. But at test time, the model does not have access to the ground truth and outputs probabilities are conditioned on its own previous predictions instead. Therefore if the predictions differ from the ground truth, the model has to continue based on an exploration path it has not seen during training, which means that it is less likely to make accurate predictions. This can lead to a compounding of errors, where mistakes in prediction accumulate and prevent good performance.

The second major issue is the discrepancy between the training loss and the various test errors associated with the tasks for which RNNs are used (e.g. edit distance, F1 score...). Of course, a single surrogate is not likely be a good approximation for all these errors. One salient illustration is that MLE ignores the information contained in structured losses. It is only focusing on maximizing the probability of the ground truth. This means that it does not distinguish between a prediction that is very close to the ground truth and one that is very far away. Thus, most of the information given by a structured loss is not leveraged with this approach.

Local vs. sequence-level. Some recent papers (Ranzato et al., 2016; Wiseman and Rush, 2016) also point out the fact that since RNNs output next token predictions, their loss is local instead of sequence-level, contrary to the error we typically want to minimize. This claim seems to contradict the standard RNN analysis, which postulates that the underlying graphical model is the complete graph: that is, the RNN outputs the probability of the next tokens conditioned on all the previous predictions. Thanks to the chain rule, one recovers the probability of the whole sequence. Thus the maximum likelihood training loss is indeed a *sequence level* loss, even though we can decompose it in a product of local losses at each cell. However, if we assume that the RNN outputs are only conditioned on the last few predictions (instead of all previous ones), then we can indeed consider the MLE loss as local. In this setting the underlying graphical model obeys Markovian constraints (as in maximum entropy Markov models (MEMMs)) rather than the complete graph; this corresponds to the assumption that the information from the previous inputs is imperfectly carried through the network to the cell, preventing the model from accurately representing long-term dependencies.

Given all these limitations, exploring novel ways of training RNNs appears to be a worthy endeavor, and this field has attracted a lot of interest in the past few years. Contrary to many papers which try to adapt ideas coming from the reinforcement learning literature, we focus in this paper on the links we can draw with structured prediction, and in particular with the L2S approach.

3 Links between RNNs and learning to search

The L2S approach to structured prediction was first introduced by Daumé et al. (2009). The main idea behind it is a *learning reduction* (Beygelzimer et al., 2016): transforming a complex learning problem (structured prediction) into a simpler one that we know how to solve (multiclass classification). To achieve this, Daumé et al. (2009) propose in their SEARN algorithm to train a shared local classifier to predict each token[2] *sequentially* (conditioned on all inputs and all past decisions), thus searching greedily step by step in the big combinatorial space of structured outputs. The idea that tokens can be predicted one at a time, conditioned on their predecessors, is central to this approach.

The training procedure is iterative: at the beginning of each round, one uses the current model or policy to build an intermediate dataset to train the shared classifier on. The specificity of this new dataset is that each sample is accompanied by a cost vector containing one entry per token in the output vocabulary $\mathcal{A}$. To obtain these cost vectors, one starts by applying a *roll-in* strategy to predict all the tokens up to T, thus building one trajectory (or exploration path) per sample in the search space. Then, at each time step, one picks arbitrarily each possible token (diverging from the roll-in trajectory) and then continues predicting to finish the modified trajectory using a *roll-out* strategy. One then computes the cost of all the obtained sequences, and ends up with T vectors (one per time step) of size $|\mathcal{A}|$ (the number of possible tokens) for every sample. Figure 1 describes the same process for our SEARNN algorithm (although it uses a different classifier).

One then extracts features from the "state" at each time step t (which encompasses the full input and the previous tokens predicted up to t during the roll-in). Combining the cost vectors to these features yields the new intermediary dataset. The original problem is thus reduced to multi-class *cost-sensitive* classification, which can be further reduced to binary classification. Once the shared classifier has been fully trained on this new dataset, the policy is updated for the next round. Theoretical guarantees for various policy updating rules are provided by e.g. Daumé et al. (2009) and Chang et al. (2015).

Roll-in and roll-out strategies. The strategies used to create the intermediate datasets fulfill different roles. The *roll-in* policy controls what part of the search space the algorithm explores, while the *roll-out* policy determines how the cost of each token is computed. The main possibilities for both roll-in and roll-out are explored by Chang et al. (2015). The *reference* policy tries to pick the optimal token based on the ground truth. During the roll-in, it corresponds to picking the ground truth. For the roll-out phase, while it is easy to compute an optimal policy in some cases (e.g. for the Hamming loss where simply copying the ground truth is also optimal), it is often intractable (e.g. for BLEU score). One then uses a heuristic (in our experiments the reference policy is always to copy the ground truth for both roll-in and roll-out). The *learned policy* simply uses the current model instead, and the *mixed* policy stochastically combines both. According to Chang et al. (2015), the best combination when the reference policy is poor is to use a learned roll-in and a mixed roll-out.

[2]Note that the vocabulary used in this literature is slightly different from that of RNNs: tokens are rather referenced as actions, predictions as decisions and models as policies.

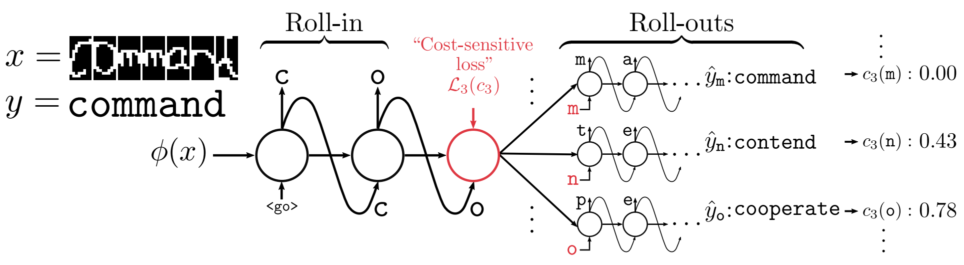

Figure 1: Illustration of the roll-in/roll-out mechanism used in SEARNN. The goal is to obtain a vector of costs for each cell of the RNN in order to define a *cost sensitive loss* to train the network. These vectors have one entry per possible token. Here, we show how to obtain the vector of costs for the red cell. First, we use a *roll-in* policy to predict until the cell of interest. We highlight here the *learned* strategy where the network passes its own prediction to the next cell. Second, we proceed to the *roll-out* phase. We feed every possible token (illustrated by the red letters) to the next cell and let the model predict the full sequence. For each token a, we obtain a predicted sequence $\hat{y}_a$. Comparing it to the ground truth sequence y yields the associated cost $c(a)$.

Links to RNNs. One can identify the following interesting similarities between a greedy approach to RNNs and L2S. Both models handle sequence labeling problems by outputting tokens recursively, conditioned on past decisions. Further, the RNN "cell" is shared at each time step and can thus also be seen as a shared local classifier that is used to make structured predictions, as in the L2S framework.

Despite this connection, many differences remain. For example, while there is a clear equivalent to the roll-in strategy in RNNs, i.e. the decision to train with or without teacher forcing (conditioning the outputs on the ground truth or instead on the previous predictions of the model), there are no roll-outs involved in standard RNN training. We thus consider next whether ideas coming from L2S could mitigate the limitations of MLE training for RNNs. In particular, one key property of L2S worth porting over to RNN training is that the former fully leverages structured losses information, contrarily to MLE as previously noted.

4 Improving RNN training with L2S

Since we are interested in leveraging structured loss information, we can try to obtain it in the same fashion as L2S. The main tool that L2S uses in order to construct a cost-sensitive dataset is the roll-out policy. In many classical structured prediction use cases, one does not need to follow through with a policy because the "cost-to-go" that the roll-out yields is either free or easily computable from the ground truth. We are however also interested in cases where this information is unavailable, and roll-outs are needed to approximate it (e.g. for machine translation). This leads to several questions: How can we integrate roll-outs in a RNN model? How do we use this additional information, i.e. what loss do we use to train the model on? How do we make it computationally tractable?

The SEARNN Algorithm. The basic idea of the SEARNN algorithm is quite simple: we borrow from L2S the idea of using a *global* loss for each *local* cell of the RNN. As in L2S, we first compute a *roll-in* trajectory, following a specific roll-in strategy. Then, at each step t of this trajectory, we compute the costs $c_t(a)$ associated with each possible token a. To do so we pick a at this step and then follow a *roll-out* strategy to finish the output sequence $\hat{y}_a$. We then compare $\hat{y}_a$ with the ground truth using the test error itself, rather than a surrogate. By repeating this for the T steps we obtain T cost vectors. We use this information to derive one *cost-sensitive* training loss for each cell, which allows us to compute an update for the parameters of the model. The full process for one cell is illustrated in Figure 1. Our losses are *global-local*, in the sense that they appear at the local level but all contain sequence-level information. Our final loss is the sum over the T local losses. We provide the pseudo-code for SEARNN in Appendix A.

Choosing a multi-class classifier. SEARNN appears quite similar to L2S, but there are a few key differences that merit more explanation. As the RNN cell can serve as a multi-class classifier, in SEARNN we could pick the cell as a (shallow) shared classifier. Instead, we pick the RNN itself, thus getting a (deep) shared classifier that also learns the features. The difference between the two options is more thoroughly detailed in Appendix C. Arbitrarily picking a token a during the roll-out phase can then be done by emulating the teacher forcing technique: if predicted tokens are fed back to the model (say if the roll-out strategy requires it), we use a for the next cell (instead of the prediction the cell would have output). We also use a in the output sequence before computing the cost.

Choosing a cost-sensitive loss. We now also explain our choice for the training loss function derived from the cost vectors. One popular possibility from L2S is go the full reduction route down to binary classification. However, this technique involves creating multiple new datasets (which is hard to implement as part of a neural network), as well as training $|\mathcal{A}|^2$ binary classifiers. We rather simply work with the multi-class classifier encoded by the RNN cell with training losses defined next.

One central idea in L2S is to learn the target tokens the model should aim for. This can be more meaningful than blindly imposing the ground truth as target, in particular when the model has deviated from the ground truth trajectory. In the specific context of RNN training, we call this approach *target learning*. It is related to the *dynamic oracle* concept introduced by Golberg and Nivre (2012). We define three losses that follow this principle. In the following, each loss is defined at the cell level. The global loss is the sum of all T losses. $s_t(a)$ refers to the score output by cell t for token a.

Log-loss (LL). Our first loss is a simple log-loss with the minimal cost token as target:

$$\mathcal{L}_t(s_t; c_t) = -\log\left(e^{s_t(a^\star)} \big/ \textstyle\sum_{i=1}^{A} e^{s_t(i)}\right) \text{ where } a^\star = \arg\min_{a\in\mathcal{A}} c(a)\,. \tag{3}$$

It is structurally similar to MLE, which is a significant advantage from an optimization perspective: as RNNs have mostly been trained using MLE, this allows us to leverage decades of previous work. Note that when the reference policy is to always copy the ground truth (which is sometimes optimal, e.g. when the test error is the Hamming loss), $a^\star$ is always the ground truth token. LL with reference roll-in and roll-out is in this case *equivalent* to MLE.

Log-loss with cost-augmented softmax (LLCAS). The log-loss approach appears to be relatively wasteful with the structured information we have access to since we are only exploiting the minimal cost value. A slight modification allows us to exploit this information more meaningfully: we add information about the full costs in the exponential, following e.g. Pletscher et al. (2010).

$$\mathcal{L}_t(s_t; c_t) = -\log\left(e^{s_t(a^\star)+c_t(a^\star)} \big/ \textstyle\sum_{i=1}^{A} e^{s_t(i)+c_t(i)}\right) \text{ where } a^\star = \arg\min_{a\in\mathcal{A}} c(a)\,. \tag{4}$$

The associated gradient update discriminates between tokens based on their costs. It leverages the structured loss information more directly and thus mitigates the 0/1 nature of MLE better.

Structured hinge loss (SHL). The LLCAS can be seen as a smooth version of the (cost-sensitive) structured hinge loss used for structured SVMs (Tsochantaridis et al., 2005), that we also consider:

$$\mathcal{L}_t(s_t; c_t) = \max_{a\in\mathcal{A}}(s_t(a) + c_t(a)) - s_t(a^\star) \text{ where } a^\star = \arg\min_{a\in\mathcal{A}} c(a)\,. \tag{5}$$

Optimization. Note that we do not need the test error to be differentiable, as our costs $c_t(a)$ are fixed when we minimize our training loss. This corresponds to defining a different loss at each round, which is the way it is done in L2S. In this case our gradient is unbiased. However, if instead we consider that we define a single loss for the whole procedure, then the costs depend on the parameters of the model and we effectively compute an approximation of the gradient. Whether it is possible not to fix the costs and to backpropagate through the roll-in and roll-out remains an open problem.

Another difference between SEARN and RNNs is that RNNs are typically trained using stochastic gradient descent, whereas SEARN is a batch method. In order to facilitate training, we decide to go for the stochastic optimization route, by selecting a random mini-batch of samples at each round (as proposed in Chang et al. (2015)). We also choose to do a single gradient step on the parameters with the associated loss (contrary to SEARN where the reduced classifier is fully trained at each round).

Expected benefits. SEARNN can improve performance because of a few key properties. First, our losses leverage the test error, leading to potentially much better surrogates than MLE.

Second, all of our training losses (even plain LL) leverage the structured information that is contained in the computed costs. This is much more satisfactory than MLE which does not exploit this information and ignores nuances between good and bad candidate predictions. Indeed, our hypothesis is that the more complex the error is, the more SEARNN can improve performance.

Third, the exploration bias we find in teacher forcing can be mitigated by using a "learned" roll-in strategy, which can be the best roll-in strategy for L2S applications according to Chang et al. (2015).

Dataset		A	T	Cost	**MLE**		**LL**			**LLCAS**		
						roll-in	learned	reference	learned	learned	reference	learned
						roll-out	mixed	learned	learned	mixed	learned	learned
OCR		26	15	Hamming	2.8		1.9	2.5	**1.8**	1.9	2.4	1.9
CoNLL		22	70	norm. Hamming	4.2		**3.7**	6.1	5.6	5.8	5.3	5.1
Spelling	0.3	43	10	edit	19.6		17.8	19.5	17.9	**17.7**	19.6	**17.7**
	0.5				43.0		37.3	43.3	37.5	**37.1**	43.3	38.2

Table 1: Comparison of the SEARNN algorithm with MLE for different cost sensitive losses and different roll-in/roll-out strategies. We provide the number of actions A and the maximum sequence length T. Note that we use 0.5 as the mixing probability when we use the mixed roll-out strategy. Greyed figures indicate unstable runs where we report the test error of the model with minimum validation error.

Fourth, the loss at each cell is *global*, in the sense that the computed costs contain information about full sequences. This may help with the classical vanishing gradients problem that is prevalent in RNN training and motivated the introduction of specialized cells such as LSTMs (Hochreiter and Schmidhuber, 1997) or GRUs (Cho et al., 2014). It may also alleviate the label bias issue that might appear if the information is not flowing perfectly through the RNN, as pointed out in Section 2.

Experiments. In order to validate these theoretical benefits, we ran SEARNN on three datasets and compared its performance against that of MLE. For fair comparison, we use the same optimization routine for all methods. We pick the one that performs best for the MLE baseline.

The first dataset is the optical character recognition (OCR) dataset introduced in Taskar et al. (2003). The task is to output English words given an input sequence of handwritten characters. We use an encoder-decoder model with GRU cells (Cho et al., 2014) of size 128. For all runs, we use SGD with constant step-size 0.5 and batch size of 64. The cost used in the SEARNN algorithm is the hamming error. Performance are reported on the test set with the total hamming error, normalized by the total number of characters.

The second experiment is run on the text-chunking CoNLL (Tjong Kim Sang and Buchholz, 2000) dataset (see Appendix B for details). We use an encoder-decoder model with 2 layers GRU cells of size 172. Our cost here is the sentence-level normalized Hamming error. On this dataset, the Adadelta optimizer with learning rate 1 worked best.

The third dataset is the Spelling dataset introduced in Bahdanau et al. (2017). The task is to recover correct text from a corrupted version. This dataset is synthetically generated from a text corpus (One Billion Word dataset): for each character, we decide with some fixed probability whether or not to replace it with a random one. The total number of tokens A is 43 (alphabet size plus a few special characters) and the maximum sequence length T is 10 (sentences from the corpus are clipped). We provide results for two sub-datasets generated with the following replacement probabilities: 0.3 and 0.5. For this task, we follow Bahdanau et al. (2017) and use the edit distance as our cost. It is defined as the edit distance between predicted sequence and the ground truth sequence over the ground truth length. We use an encoder-decoder model with GRU cells of size 100 with the attention mechanism described in (Bahdanau et al., 2017). For all runs, we use the Adam optimizer with learning rate 0.001 and batch size of 128. Results are given in Table 1.

Key takeaways. First, SEARNN outperforms MLE by a significant margin on the three different tasks and datasets, which confirms our intuition that taking structured information into account enables better performance. Second, we observed that the SHL loss was not improving results in general, while LL and LLCAS – which are structurally close to MLE – achieve better performance. This might be explained by the fact that RNN architectures and optimization techniques have been evolving for decades with MLE training in mind. Third, the best roll-in/out strategy appears to be combining a learned roll-in and a mixed roll-out, which is consistent with the claims from Chang et al. (2015). Fourth, the spelling task shows us that the harder the task is (hence the less a simplistic roll-out strategy – akin to MLE – is efficient), the stronger the improvements SEARNN makes over MLE. One should note that we get improvements even in the case where simply outputting the ground truth is the optimal policy, regardless of the current trajectory.

Dataset		MLE	LL					sLL					sLLCAS				
			uni.	stat.	pol.	bias.	top-k	uni.	stat.	pol.	bias.	top-k	uni.	stat.	pol.	bias.	top-k
OCR		2.84	1.94	**1.50**	1.96	1.84	2.13	1.82	1.91	1.86	2.25	2.69	2.03	2.33	**1.50**	2.37	1.94
Spelling	0.3	19.6	17.7	17.7	17.7	17.7	17.8	18.8	20.5	17.5	17.7	18.4	18.8	20.2	17.7	17.7	18.2
	0.5	43.0	37.0	36.7	37.1	36.6	36.6	37.4	41.4	37.7	36.9	41.7	37.6	41.7	37.0	37.8	40.5

Table 2: Comparison of the SEARNN algorithm with MLE for different datasets using the sampling approach.

5 Scaling up SEARNN

While SEARNN does provide significant improvements on the two tasks we have tested it on, it comes with a rather heavy price, since a large number of roll-outs (i.e. forward passes) have to be run in order to compute the costs. This number, $|\mathcal{A}|T$, is proportional both to the length of the sequences, and to the number of possible tokens. SEARNN is therefore not directly applicable to tasks with large output sequences or vocabulary size (such as machine translation) where computing so many forward passes becomes a computational bottleneck. Even though forward passes can be parallelized more heavily than backward ones (because they do not require maintaining activations in memory), their asymptotic cost remains in $\mathcal{O}(dT)$, where d is the number of parameters of the model.

There are a number of ways to mitigate this issue. In this paper, we focus on subsampling both the cells and the tokens when computing the costs. That is, instead of computing a cost vector for each cell, we only compute them for a subsample of all cells. Similarly, we also compute these costs only for a small portion of all possible tokens. The speedups we can expect from this strategy are large, since the total number of roll-outs is proportional to both the quantities we are decreasing.

Sampling strategies. First, we need to decide how we select the steps and tokens that we sample. We have chosen to sample steps uniformly when we do not take all of them. On the other hand, we have explored several different possibilities for token sampling. The first is indeed the uniform sampling strategy. We also tested sampling according to pre-computed, corpus-wide statistics. Finally, we tried 3 samplings using the current state of our model: stochastic current policy sampling (where we use the current state of the stochastic policy to pick at random), a biased version of current policy sampling where we boost the scores of the low-probability tokens, and finally a *top-k* strategy where we take the top k tokens according to the current policy. Note that in all strategies we always sample the ground truth action to make sure that our performance is at least as good as MLE.

Adapting our losses to sampling. Several losses require computing the costs of all possible tokens at a given step, including LL. One could still use LL by simply making the assumption that the token with minimum cost is always sampled. However this is a rather strong assumption and it means pushing down the scores of tokens that were not even sampled and hence could not compete with the others. To alleviate this issue, we replace the full softmax by a layer applied only on the tokens that were sampled (Jean et al., 2015). While the target can still only be in the sampled tokens, the unsampled tokens are left alone by the gradient update, at least for the first order dependency. This trick is even more needed for LLCAS, which otherwise requires a "reference" score for unsampled tokens, adding a difficult to tune hyperparameter. We refer to these new losses as sLL and sLLCAS.

Experiments. The main goal of these experiments is to assess whether or not combining subsampling with the SEARNN algorithm is a viable strategy. To do so we ran the method on two datasets that we used in the previous section. We decided to only focus on subsampling tokens as the vocabulary size is usually the blocking factor rather than the sequence length. Thus in the following we always sample all cells. We evaluate different sampling strategies and training losses. For all experiments, we use the learned strategy for roll-in and the mixed one for roll-out and we sample 5 tokens per cell. Finally, we use the same optimization techniques than in the previous experiment.

Key takeaways. Results are given in Table 2. The analysis of this experiment yields interesting observations. First, and perhaps most importantly, subsampling appears to be a viable strategy to obtain a large part of the improvements of SEARNN while keeping computational costs under control. Indeed, we recover a substantial part of the improvements of the full method while only sampling a fraction of all possible tokens. Second, it is difficult to decide on a best strategy for token sampling. Consequently, a mixture of several might be the best option. Third, it seems also difficult to distinguish a best performing loss. Experimentations with larger vocabulary size might be needed to better differentiate the different tokens sampling strategies and losses. Finally, this sampling technique yields a 5x speedup, therefore validating our scaling approach.

6 Discussion

RL-inspired approaches. In structured prediction tasks, we have access to ground truth trajectories, i.e. a lot more information than in traditional RL. One major direction of research has been to adapt RL techniques to leverage this additional information. The main idea is to try to optimize the expectation of the test error directly (under the stochastic policy parameterized by the RNN):

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} \mathbb{E}_{(y_1^i,..,y_T^i)\sim\pi(\theta)} r(y_1^i, .., y_T^i)\,. \tag{6}$$

Since we are taking an expectation over all possible structured outputs, the only term that depends on the parameters is the probability term (the tokens in the error term are fixed). This allows this loss function to support non-differentiable test errors, which is a key advantage. Of course, actually computing the expectation over an exponential number of possibilities is computationally intractable.

To circumvent this issue, Shen et al. (2016) subsample trajectories according to the learned policy, while Ranzato et al. (2016); Rennie et al. (2016) use the REINFORCE algorithm, which essentially approximates the expectation with a single trajectory sample. Bahdanau et al. (2017) adapt the ACTOR-CRITIC algorithm, where a second *critic* network is trained to approximate the expectation.

While all these approaches report significant improvement on various tasks, one trait they share is that they only work when initialized from a good pre-trained model. This phenomenon is often explained by the sparsity of the information contained in "sequence-level" losses. Indeed, in the case of REINFORCE, no distinction is made between the tokens that form a sequence: depending on whether the sampled trajectory is above a global baseline, all tokens are pushed up or down by the gradient update. This means good tokens are sometimes penalized and bad tokens rewarded.

In contrast, SEARNN uses "global-local" losses, with a local loss attached to each step, which contains global information since the costs are computed on full sequences. To do so, we have to "sample" more trajectories through our roll-in/roll-outs. As a result, SEARNN does not require warm-starting to achieve good experimental performance. This distinction is quite relevant, because warm-starting means initializing in a specific region of parameter space which may be hard to escape. Exploration is less constrained when starting from scratch, leading to potentially larger gains over MLE.

Reinforcement-based models also often require optimizing additional models (REINFORCE needs to learn baselines and ACTOR-CRITIC the critic model), which can introduce more complexity (e.g. target networks). SEARNN does not. This too may contribute to the warm start difference.

Finally, while minimizing the expected reward allows the RL approaches to use gradient descent even though the test error might not be differentiable, it introduces another discrepancy between training and testing. Indeed, at test time, one does not decode by sampling from the stochastic policy. Instead, one selects the best performing sequence (according to a search algorithm, such as greedy or beam search). SEARNN avoids this averse effect by computing costs using the same decoding technique as the one used at test time, so that its loss can be even closer to the test loss. The price we pay here is that we approximate the gradient by fixing the costs, although they are dependent on the parameters.

L2S-inspired approaches. Several other papers have tried using L2S-like ideas for better RNN training. Wiseman and Rush (2016) propose to include the beam search procedure in a sequence-level loss, following the "Learning A Search Optimization" approach of Daumé and Marcu (2005). Here again the sequence-level loss information is too sparse and warm starting has to be used. Ballesteros et al. (2016) use a loss that is similar to LL for parsing. However, their approach is limited to a specific task where cost-to-go are essentially free, whereas SEARNN can be used on any task. This limit also affects Sun et al. (2017), in which new gradient procedures are introduced to incorporate neural classifiers in the AGGREVATE (Ross and Bagnell, 2014) variant of L2S.

Conclusion and future work. We have described SEARNN, a novel algorithm that uses core ideas from the learning to search framework in order to alleviate the known limitations of MLE training for RNNs. By leveraging structured cost information obtained through strategic exploration, we define global-local losses. These losses give a *global* feedback related to the structured task at hand, distributed *locally* within the cells of the RNN. This alternative procedure allows us to train RNNs from scratch and to outperform MLE on three challenging structured prediction tasks. Finally we have proposed promising scaling techniques that open up the possibility of applying SEARNN on structured tasks for which the output vocabulary is very large, such as neural machine translation.

Acknowledgments

This work was partially supported by a Google Research Award and ERC grant Activia (no. 307574).

References

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

D. Bahdanau, P. Brakel, K. Xu, A. Goyal, R. Lowe, J. Pineau, A. Courville, and Y. Bengio. An actor-critic algorithm for sequence prediction. In *ICLR*, 2017.

M. Ballesteros, Y. Goldberg, C. Dyer, and N. A. Smith. Training with exploration improves a greedy stack-LSTM parser. In *EMNLP*, 2016.

A. Beygelzimer, H. Daumé, J. Langford, and P. Mineiro. Learning reductions that really work. *Proceedings of the IEEE*, 104(1):136–147, 2016.

K.-W. Chang, A. Krishnamurthy, A. Agarwal, H. Daumé, III, and J. Langford. Learning to search better than your teacher. In *ICML*, 2015.

K. Cho, B. Van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537, 2011.

H. Daumé, III and D. Marcu. Learning as search optimization: approximate large margin methods for structured prediction. In *ICML*, 2005.

H. Daumé, III, J. Langford, and D. Marcu. Search-based structured prediction. *Machine Learning*, 2009.

Y. Golberg and J. Nivre. A dynamic oracle for arc-eager dependency parsing. *COLING*, 2012.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

S. Jean, K. Cho, R. Memisevic, and Y. Bengio. On using very large target vocabulary for neural machine translation. In *ACL*, 2015.

M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. In *EMNLP*, 2015.

P. Pletscher, C. S. Ong, and J. M. Buhmann. Entropy and margin maximization for structured output learning. In *ECML PKDD*, 2010.

M. Ranzato, S. Chopra, M. Auli, and W. Zaremba. Sequence level training with recurrent neural networks. In *ICLR*, 2016.

S. Rennie, E. Marcheret, Y. Mroueh, J. Ross, and V. Goel. Self-critical sequence training for image captioning. *arXiv:1612.00563*, 2016.

S. Ross and J. A. Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv:1406.5979*, 2014.

S. Shen, Y. Cheng, Z. He, W. He, H. Wu, M. Sun, and Y. Liu. Minimum risk training for neural machine translation. *ACL*, 2016.

N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

W. Sun, A. Venkatraman, G. J. Gordon, B. Boots, and J. A. Bagnell. Deeply aggrevated: Differentiable imitation learning for sequential prediction. *arXiv:1703.01030*, 2017.

I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

B. Taskar, C. Guestrin, and D. Koller. Max-margin Markov networks. In *NIPS*, 2003.

E. F. Tjong Kim Sang and S. Buchholz. Introduction to the CoNLL-2000 shared task: Chunking. In *CoNLL*, 2000.

I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun. Large margin methods for structured and interdependent output variables. *JMLR*, 2005.

O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *CVPR*, 2015.

S. Wiseman and A. M. Rush. Sequence-to-sequence learning as beam-search optimization. In *EMNLP*, 2016.

A SEARNN algorithm.

Algorithm 1 SEARNN algorithm (for a simple encoder-decoder network)

```
Initiate the weights $\omega$ of the RNN network.
for $i$ in 1 to $N$ do
    Sample $B$ ground truth input/output structured pairs $\{(x^1, y^1), \cdots, (x^B, y^B)\}$
    # Perform the roll-in/roll-outs to get the costs. Can be heavily parallelized
    for $b$ in 1 to $B$ do
        Compute input features $\phi(x^b)$
        for $t$ in 1 to $T$ do
            # The following roll-in is actually run only once
            Run the RNN until the $t^{\text{th}}$ cell with $\phi(x^b)$ as initial state by following the roll-in strategy
            Collect the state in order to perform roll-outs
            # Roll-outs for all actions in order to collect the cost vector
            for $a$ in 1 to $A$ do
                Run the RNN from the $t^{\text{th}}$ cell to the end by enforcing action $a$ at cell $t$
                Decode to obtain the output sequence $\hat{y}_t^b(a)$
                Collect the cost $c_t^b(a)$ by comparing $\hat{y}_t^b(a)$ and $y^b$
            end for
        end for
    end for
    Obtain the losses for each cell with the collected costs
    Update the parameters of the network $\omega$ by doing a single gradient step
end for
```

B Details on CoNLL experiments

The CoNLL-2000 (Tjong Kim Sang and Buchholz, 2000) dataset[3] poses the task of text chunking, consisting in splitting an input sentence into syntactically related non-overlapping consecutive groups of words, called phrases or chunks. The input sequence x consists in tuples of words and the corresponding part-of-speech (POS) tags. The size of input vocabulary is 17,464 for words and 44 for POS tags. The output sequence y has to be of the same length and each token can take one of the 23 values representing chunks in a so-called BIO encoding. The training and test sets consist of 8,936 and 2,012 items, respectively.

The major difference from OCR consists in the fact that there are two input tokens at each time step. We use 50 dimensional embeddings for the words and 44 dimensional embeddings for POS tags. Since many words appear in the dataset very few times, we initialize the embeddings from SENNA v3.0 embeddings[4] of Collobert et al. (2011). To construct the final dictionary we lower case all the words and take the ones that have SENNA embeddings. We also separately add the words that appear in the dataset more than 10 times. Finally, we merge all the tokens representing numbers together and assign them to the special <NUM> token. The final dictionary is of size 15,222. All the out-of-dictionary words are assigned to the special <UNK> token.

For both the encoder and decoder, we use two layers of GRU cells each with memory dimension of 172. In addition, the encoder is bidirectional. To improve the decoder, we use the attention mechanism as proposed by Bahdanau et al. (2015, 2017), the input feeding mechanism as described by Luong et al. (2015, Section 3.3), and we add dropout regularization (Srivastava et al., 2014) of strength 0.3 between the two layers and on the output of the attention mechanism.

C Design decisions

Choosing a classifier: to backpropagate or not to backpropagate? In standard L2S, the classifier and the feature extractor are clearly delineated. The latter is a fixed hand-crafted transformation

[3]http://www.clips.uantwerpen.be/conll2000/chunking/

[4]http://ronan.collobert.com/senna/

applied on the input and the partial sequence that has already been predicted. One then has to pick a classifier and its convergence properties carry over to the initial problem.

In SEARNN, we choose the RNN itself as our classifier. The fixed feature extractor is reduced to the bare minimum (e.g. one-hot encoding) and the classifier performs feature learning afterwards. In this setting, the intermediate dataset is the initial state and all previous decisions $(x, y_{1:t-1})$ combined with the cost vector.[5]

An alternative way to look at RNNs, is to consider the RNN cell as a shared classifier in its own right, and the beginning of the RNN (including the previous cells) as a feature extractor. One could then pick the RNN cell (instead of the full RNN) as the SEARNN classifier, in which case the intermediate dataset would be (h_{t-1}, y_{t-1})[6] (the state at the previous step, combined with the previous decision) plus the cost vector.

While this last perspective – seeing the RNN cell as the shared classifier instead of the full RNN – is perhaps more intuitive, it actually fits the L2S framework less well. Indeed, there is no clear delineation between classifier and feature extractor as these functions are carried out by different instances of the same RNN cell (and as such share weights). This means that the feature extraction in this case is learned instead of being fixed.

This choice of classifier has a direct consequence on the optimization routine. In case we pick the RNN itself, then each loss gradient has to be fully backpropagated through the network. On the other hand, if the classifier is the cell itself, then one should not backpropagate the gradient updates.

Reference policy. The reference policy defined by Daumé et al. (2009) picks the action which "minimizes the (corresponding) cost, assuming all future decisions are made optimally", i.e. $\arg\min_{y_t} \min_{y_{t+1:T}} l(y_{1:T}, y)$.

For the roll-in phase, this policy corresponds to always picking the ground truth, since it leads to predicting the full ground truth sequence and hence the best possible loss.

For the roll-out phase, computing this policy explicitly is easy in a few select cases. However, in the general case it is not tractable. One then has to turn to heuristics, whose performance can be relatively poor. While Chang et al. (2015) tell us that overcoming a bad reference policy can be done through a careful choice of roll-in/roll-out policies, the fact remains that the better the reference policy is, the better performance will be. Choosing this heuristic well is then quite important.

The most basic heuristic is of to simply use the ground truth. Of course, one can readily see that it is not always optimal. For example, when the model skips a token and outputs the next one, a, instead, it may be more beneficial to also skip a in the roll-out phase rather than to repeat it.

Although we chose this basic heuristic in this paper, we believe that using tailored alternatives can yield better results for tasks where it is suboptimal, such as machine translation.

[5]In the encoder-decoder architecture, the decoder RNN does not receive x directly, but rather $\phi(x)$, the features extracted from the input by the encoder RNN. In this case, our SEARNN classifier includes both the encoder and the decoder RNNs.

[6]One could also add $\psi(x)$, features learned from the input through e.g. an attention mechanism.