SEARNN: Training RNNs with global-local losses

Rémi Leblond [*1 2] Jean-Baptiste Alayrac [*1 2] Anton Osokin [1 2] Simon Lacoste-Julien [3]

Abstract

We propose SEARNN, a novel training algorithm for recurrent neural networks (RNNs) inspired by the "learning to search" (L2S) approach. RNNs have been widely successful in structured prediction tasks 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 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 3 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, e.g. machine translation (Sutskever et al., 2014). 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 its predecessors, this cell learns to output the next token conditioned on the previous ones. The standard training loss 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 train/test discrepancy is the *exposure* or *exploration bias* (Ranzato et al., 2016): in traditional MLE training the cell learns the conditional probability of the next token, based on previous ground truth tokens – this is referred to as *teacher forcing*. But at test time the model does not have access to the ground truth, and thus feeds its own previous predictions to its cell for prediction instead.

To address these issues, we propose to use ideas from the structured prediction field, in particular from the "learning to search" (L2S) approach introduced by Daumé et al. (2009).

Outline. In § 2, we detail the limitations of MLE training. In § 3, we make explicit the strong links between RNNs and L2S. In § 4, we present SEARNN, a novel training algorithm for RNNs, using ideas from L2S to derive a *global-local* loss that is closer to the test error than MLE. We show that SEARNN leads to significant improvements on 3 difficult structured prediction tasks. As this algorithm is quite costly, in § 5 we propose a subsampling strategy that allows us to considerably reduce training times while maintaining improved performance compared to MLE. Finally, in § 6 we contrast our novel approach to the related work.

2. Limitations of traditional RNN training

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: $h_t = f(h_{t-1}, y_{t-1}, x)$, given an optional input x.

We focus on a subset of this family typically used for structured prediction, where we want to model the *joint probability* of a target sequence $y_1, \ldots, y_{T_x}$ given an input x (e.g. the *decoder* RNN in the encoder-decoder architecture (Sutskever 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 add a projection layer over h_t to obtain a vector of scores s_t over all tokens $a \in \mathcal{A}$, and a softmax normalizer layer to get a distribution o_t over $\mathcal{A}$. o_t is interpreted as the predictive conditional distribution for the t^{th} token given by the RNN model, i.e. $p(a|y_{1:t-1}, x) := o_t(a)$ for $a \in \mathcal{A}$. Multiplying the values $o_t(y_t)$ together yields the joint probability of the sequence y defined by the RNN (thanks to the chain rule):

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

[*]Equal contribution [1]INRIA, Paris [2]ENS, Paris [3]UdeM, Montréal. Correspondence to: Rémi Leblond <remi.leblond@inria.fr>.

The underlying structure of these RNNs as graphical models is thus the complete graph. To decode, one uses beam search or *greedy* predictions $\hat{y}_t := \arg\max_{a \in \mathcal{A}} p(a|\hat{y}_{1:t-1}, x)$.

In the "teacher forcing" regimen, the inputs to the RNN cell are the ground truth tokens (as opposed to its own greedy predictions). We get the probability of each ground truth sequence according to the RNN model, allowing us to derive a training loss from MLE. Although the individual output probabilities are at token level, the MLE loss involves the joint probability (via chain rule) and is thus *sequence level*.

The limitations of MLE training. MLE training suffers from *exposure* or *exploration bias* (Ranzato et al., 2016). When 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 continues based on an exploration path it has not seen in training, meaning that it is less likely to make accurate predictions.

The second major issue is the discrepancy between the training loss and the various test errors (e.g. edit distance, F1 score...). Of course, a single surrogate is not likely be a good approximation for all these errors. For example, MLE ignores most of the information given by a structured loss. As it only focuses on maximizing the probability of the ground truth, it does not distinguish between a prediction that is close to the ground truth and one that is far away.

These issues motivate exploring new ways of training RNNs. This field has attracted a lot of interest in the past few years. Contrary to many papers which try to adapt ideas coming from reinforcement learning, we focus on the links we can draw with structured prediction, and in particular with L2S.

3. Links between RNNs and L2S

The main idea behind the L2S approach to structured prediction is *reduction*: transforming a complex problem into a simpler one. To achieve this, Daumé et al. (2009) propose in their SEARN algorithm to train a shared local classifier to predict each token *sequentially* (conditioned on all inputs and all past tokens), thus searching greedily step by step in a big combinatorial space. A central idea is that tokens can be predicted one at a time, conditioned on their predecessors.

The training procedure is iterative: at the beginning of each round, one uses the current model or policy to build an intermediate *cost-sensitive* dataset to train the shared classifier, where each sample is accompanied by a cost vector containing one entry for each $a \in \mathcal{A}$. To obtain these cost vectors, one first applies a *roll-in* strategy to predict all the tokens up to T, thus building one trajectory (in the search space) per sample. Then, at each time step, one picks arbitrarily each possible token (diverging from the roll-in trajectory) and continues predicting to finish the modified trajectories using a *roll-out* strategy. Computing the costs of all the obtained sequences yields T vectors (one per time step) of size A (the number of tokens) for every sample. Figure 1 describes a similar process for our SEARNN algorithm.

One then extracts features from the "state" at each time step t (i.e. the full input and the previous tokens predicted up to t during the roll-in). Combining the cost vectors to these features yields the intermediary dataset. The original problem is thus reduced to multi-class *cost-sensitive* classification. Once the shared classifier has been fully trained on this new dataset, the policy is updated for the next round.

Roll-in and roll-out strategies. These strategies 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. Alternatives for both are explored by Chang et al. (2015). The *reference* policy tries to pick the optimal action based on the ground truth. During the roll-in, it corresponds to picking the ground truth. For the roll-out, while an optimal policy is easy to compute in some cases (for the Hamming loss, simply copying the ground truth is optimal), it is often intractable. One then uses a heuristic (for us, 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. The best strategy when the reference policy is poor is to use a learned roll-in and a mixed roll-out (Chang et al., 2015).

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.

However, many differences remain. E.g., while there is a clear equivalent to the roll-in strategy in RNNs, i.e. the decision to train with or without teacher forcing, there are no roll-outs in standard RNN training. Can we use ideas coming from L2S – which leverages structured losses information – to mitigate the issues of MLE training for RNNs?

4. Improving RNN training with L2S

We can obtain structured loss information in the same fashion as L2S: through the roll-out policy. While in some structured prediction tasks the "cost-to-go" that the roll-out yields is free or easily computable, we are interested in cases where this information is unavailable, and roll-outs are needed to approximate it (e.g. machine translation). This leads to several questions: how can we integrate roll-outs in a RNN? Which training loss do we use to leverage this

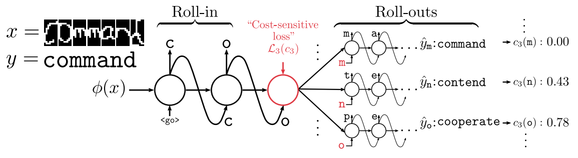
$x =$
$y =$ command
Roll-in
Roll-outs
"Cost-sensitive loss" $\mathcal{L}_3(c_3)$
$\phi(x)$
<go>
c
o
m
a
t
e
p
n
$\hat{y}_{\mathtt{m}}$:command
$\hat{y}_{\mathtt{n}}$:contend
$\hat{y}_{\mathtt{o}}$:cooperate
$c_3(\mathtt{m}) : 0.00$
$c_3(\mathtt{n}) : 0.43$
$c_3(\mathtt{o}) : 0.78$

Figure 1. Illustration of the roll-in/roll-out mechanism used in SEARNN. To define a *cost sensitive loss* to train the network we need a vector of costs for each cell of the RNN. 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 red cell. Here, we use the *learned* strategy: the network feeds its own prediction to the next cell. Second, the *roll-out* phase. We feed every possible token (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 $c(a)$.

added information? Is it computationally tractable?

The SEARNN Algorithm. We borrow from L2S the idea of using a *global* loss for each *local* cell of the RNN. 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 token a, by picking a at step t and then following a *roll-out* strategy to finish the sequence $\hat{y}_a$. $\hat{y}_a$ is compared to the ground truth using the test error itself. We use this information to derive a *cost-sensitive* training loss for each cell. The full process is illustrated in Fig 1. Our losses are *global-local*, as they appear at the local level but contain sequence-level information.

Choosing a multi-class classifier. As the RNN cell can serve as a multi-class classifier, in SEARNN we could use it as our shared classifier, contrary to L2S. Instead, we pick the RNN itself, thus getting a (deep) shared classifier that also learns the features. Arbitrarily picking a token a during the roll-out phase can be done by emulating the teacher forcing technique: when decisions are fed back, we feed a to the next cell (instead of the cell's prediction).

Choosing a cost-sensitive loss. The traditional loss in L2S is quite difficult to adapt to as a neural layer. Instead, we simply work with the multi-class classifier encoded by the RNN cell with training losses defined next.

A central idea in L2S is to learn the target tokens the model should aim for. This is more meaningful than blindly aiming for the ground truth, especially when the model deviates from the ground truth trajectory. Following this idea, we define 2 losses at the cell level (the global loss is the sum of the T losses). $s_t(a)$ is the score output by cell t for token a.

Log-loss (LL). Our first loss is a simple log-loss with the minimal cost action, $a^\star := \arg\min_{a \in \mathcal{A}} c(a)$, as target:

$$\mathcal{L}_t(s_t; c_t) = -\log\left(e^{s_t(a^\star)} \big/ \textstyle\sum_{i=1}^{A} e^{s_t(i)}\right).$$

It is structurally similar to MLE, which is an advantage from an optimization perspective as RNNs have mostly been

Dataset		MLE	*roll-in* / *roll-out*	LL			LLCAS		
				learned / mixed	reference / learned	learned / learned	learned / mixed	reference / learned	learned / learned
OCR		2.8		1.9	2.5	**1.8**	1.9	2.4	1.9
CoNLL		4.2		**3.7**	6.1	5.6	5.8	5.3	5.1
Spelling	0.3	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/out strategies.

trained using MLE. Note that when the reference policy is to always copy the ground truth (which is sometimes optimal), $a^\star$ is always the ground truth action. LL with reference roll-in and roll-out is in this case *equivalent* to MLE.

Log-loss with cost-augmented softmax (LLCAS). LL is a bit wasteful with our structured information as we are only exploiting the minimal cost value. To exploit it better, we add the full costs in the exponential, as is standard:

$$\mathcal{L}_t(s_t; c_t) = -\log\left(e^{s_t(a^\star)+c_t(a^\star)} \big/ \textstyle\sum_{i=1}^{A} e^{s_t(i)+c_t(i)}\right).$$

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

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. As RNNs are typically trained using stochastic gradient descent, we adopt it by selecting a random mini-batch of samples at each round, instead of the full dataset as in SEARN. We also do a single gradient step with the associated loss (contrary to SEARN where the reduced classifier is fully trained at each round).

Expected benefits. First, our losses leverage the test error, leading to potentially much better surrogates than MLE. Second, our training losses leverage the structured information contained in the computed costs, contrary to MLE which ignores nuances between good and bad candidates. Our hypothesis is that the more complex the error, the more SEARNN improves performance. Third, the exploration bias in teacher forcing can be mitigated by using a "learned" roll-in strategy. Fourth, the loss at each cell is *global*, as the costs are computed on full sequences. This may help with the vanishing gradients problem prevalent in RNN training.

Experiments. We ran SEARNN on an encoder-decoder model on 3 datasets and compared its performance against MLE: the OCR task from (Taskar et al., 2003), the CoNNL text chunking task (Tjong Kim Sang and Buchholz, 2000) and the spelling correction task from Bahdanau et al. (2017). For the first two, we use the Hamming error to compute costs, and report the total normalized Hamming error. The third task is to recover correct text from a corrupted version where characters are replaced by random ones with fixed probability. We provide results for two datasets generated with replacement probabilities 0.3 and 0.5. We use the edit distance as our cost. Results are given in Table 1.

Dataset		MLE	LL					LLCAS				
			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	2.03	2.33	**1.50**	2.37	1.94
Spelling	0.3	19.6	**17.7**	17.8	17.9	17.7	17.8	18.8	18.7	**17.7**	**17.7**	18.2
	0.5	43.0	37.0	36.9	37.3	36.6	**36.6**	37.6	37.7	37.0	37.8	40.5

Table 2. Performance of SEARNN using subsampling.

Key takeaways. First, SEARNN outperforms MLE significantly on all tasks, confirming the benefits of leveraging structured information. Second, 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). Third, the spelling task shows that the harder the task is, the more SEARNN improves over MLE.

5. Scaling up SEARNN

SEARNN's improvements are expensive, as AT roll-outs (forward passes) are required to compute costs. So SEARNN is not directly applicable to tasks with large vocabulary size. To mitigate this issue, we only compute cost vectors for a subsample of all tokens. We can expect a large speedup, as the total number of roll-outs is proportional to this quantity.

Sampling strategies. We explore several different possibilities. Uniform sampling, sampling according to corpus-wide statistics, and 3 samplings using the current state of our model: stochastic policy sampling (where we pick according to the current stochastic policy), a biased version of policy sampling where we boost the scores of the low-probability tokens, and a *top-k* strategy where we take the top k actions 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.

Experiments. We run the method on 2 datasets that we used in the previous section, with different samplings and training losses. We use the learned strategy for roll-in and the mixed one for roll-out and we sample 5 tokens per cell.

Key takeaways. Results are given in Table 2. First, subsampling appears to be a viable strategy as we recover all of the improvements of SEARNN while only sampling a fraction of tokens. Second, as deciding on a best sampling strategy is hard, a mixture may be the best option. Finally, this sampling technique yields a 5× speedup.

6. Comparison to RL-inspired approaches

In structured prediction tasks, we have access to ground truth trajectories, contrary to traditional RL. An active area of research is to adapt RL techniques to leverage this richer information: the idea is to optimize the expectation of the test error directly (under the RNN's stochastic policy):

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

As the only term depending on the parameters is the policy, this loss function supports non-differentiable test errors. Of course, actually computing the expectation over an exponential number of possibilities is computationally intractable. Approximating it can be done either by sampling trajectories according to the learned policy (Shen et al., 2016), by using the REINFORCE algorithm (i.e. sampling a single trajectory, see Ranzato et al. (2016)), or by training a *critic* network as in the ACTOR-CRITIC algorithm (Bahdanau et al., 2017).

While these approaches report improvement on various tasks, they only work when initialized from a good pre-trained model. This is often explained by the sparsity of the information contained in "sequence-level" losses. Indeed, in the case of REINFORCE, all tokens are pushed up or down depending on whether the trajectory is above a global baseline, without distinction. Good tokens are sometimes penalized and bad tokens rewarded.

In contrast, SEARNN uses "global-local" losses, where a local loss attached to each step is based on costs computed on full sequences. As a result, SEARNN does not require warm-starting to achieve good experimental performance. This matters 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.

Finally, the RL approach of minimizing the expected reward introduces a discrepancy between training and testing: at test time, one does not decode by sampling from the stochastic policy. Instead, one selects the best performing sequence. SEARNN avoids this averse effect by computing costs with the same decoding technique as the one used at test time, so that its loss can be even closer to the test loss.

References

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.

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

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

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

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

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.