Tackling Error Propagation through Reinforcement Learning: A Case of Greedy Dependency Parsing

Le, M.N.; Fokkens, A.S.

published in
Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics
2017

document version
Publisher's PDF, also known as Version of record

Tackling Error Propagation through Reinforcement Learning: A Case of Greedy Dependency Parsing

Minh Lê
CLTL
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
m.n.le@vu.nl

Antske Fokkens
CLTL
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
antske.fokkens@vu.nl

Abstract

Error propagation is a common problem in NLP. Reinforcement learning explores erroneous states during training and can therefore be more robust when mistakes are made early in a process. In this paper, we apply reinforcement learning to greedy dependency parsing which is known to suffer from error propagation. Reinforcement learning improves accuracy of both labeled and unlabeled dependencies of the Stanford Neural Dependency Parser, a high performance greedy parser, while maintaining its efficiency. We investigate the portion of errors which are the result of error propagation and confirm that reinforcement learning reduces the occurrence of error propagation.

1 Introduction

Error propagation is a common problem for many NLP tasks (Song et al., 2012; Quirk and Corston-Oliver, 2006; Han et al., 2013; Gildea and Palmer, 2002; Yang and Cardie, 2013). It can occur when NLP tools applied early on in a pipeline make mistakes that have negative impact on higher-level tasks further down the pipeline. It can also occur within the application of a specific task, when sequential decisions are taken and errors made early in the process affect decisions made later on.

When reinforcement learning is applied, a system actively tries out different sequences of actions. Most of these sequences will contain some errors. We hypothesize that a system trained in this manner will be more robust and less susceptible to error propagation.

We test our hypothesis by applying reinforcement learning to greedy transition-based parsers (Yamada and Matsumoto, 2003; Nivre, 2004), which have been popular because of superior efficiency and accuracy nearing state-of-the-art. They are also known to suffer from error propagation. Because they work by carrying out a sequence of actions without reconsideration, an erroneous action can exert a negative effect on all subsequent decisions. By rendering correct parses unreachable or promoting incorrect features, the first error induces the second error and so on. McDonald and Nivre (2007) argue that the observed negative correlation between parsing accuracy and sentence length indicates error propagation is at work.

We compare reinforcement learning to supervised learning on Chen and Manning (2014)'s parser. This high performance parser is available as open source. It does not make use of alternative strategies for tackling error propagation and thus provides a clean experimental setup to test our hypothesis. Reinforcement learning increased both unlabeled and labeled accuracy on the Penn TreeBank and German part of SPMRL (Seddah et al., 2014). This outcome shows that reinforcement learning has a positive effect, but does not yet prove that this is indeed the result of reduced error propagation. We therefore designed an experiment which identified which errors are the result of error propagation. We found that around 50% of avoided errors were cases of error propagation in our best arc-standard system. Considering that 27% of the original errors were caused by error propagation, this result confirms our hypothesis.

This paper provides the following contributions:

1. We introduce Approximate Policy Gradient (APG), a new algorithm that is suited for dependency parsing and other structured prediction problems.

2. We show that this algorithm improves the accuracy of a high-performance greedy parser.

Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers, pages 677–687,
Valencia, Spain, April 3-7, 2017.

3. We design an experiment for analyzing error propagation in parsing.

4. We confirm our hypothesis that reinforcement learning reduces error propagation.

To our knowledge, this paper is the first to experimentally show that reinforcement learning can reduce error propagation in NLP.

The rest of this paper is structured as follows. We discuss related work in Section 2. This is followed by a description of the parsers used in our experiments in Section 3. Section 4 outlines our experimental setup and presents our results. The error propagation experiment and its outcome are described in Section 5. Finally, we conclude and discuss future research in Section 6.

2 Related Work

In this section, we address related work on dependency parsing, including alternative approaches for reducing error propagation, and reinforcement learning.

2.1 Dependency Parsing

We use Chen and Manning (2014)'s parser as a basis for our experiments. Their parser is open-source and has served as a reference point for many recent publications (Dyer et al., 2015; Weiss et al., 2015; Alberti et al., 2015; Honnibal and Johnson, 2015, among others). They provide an efficient neural network that learns dense vector representations of words, PoS-tags and dependency labels. This small set of features makes their parser significantly more efficient than other popular parsers, such as the Malt (Nivre et al., 2007) or MST (McDonald et al., 2005) parser while obtaining higher accuracy. They acknowledge the error propagation problem of greedy parsers, but leave addressing this through (e.g.) beam search for future work.

Dyer et al. (2015) introduce an approach that uses Long Short-Term Memory (LSTM). Their parser still works incrementally and the number of required operations grows linearly with the length of the sentence, but it uses the complete buffer, stack and history of parsing decisions, giving the model access to global information. Weiss et al. (2015) introduce several improvements on Chen and Manning (2014)'s parser. Most importantly, they put a globally-trained perceptron layer instead of a softmax output layer. Their model uses smaller embeddings, rectified linear instead of cubic activation function, and two hidden layers instead of one. They furthermore apply an averaged stochastic gradient descent (ASGD) learning scheme. In addition, they apply beam search and increase training data by using unlabeled data through the tri-training approach introduced by Li et al. (2014), which leads to further improvements.

Kiperwasser and Goldberg (2016) introduce a new way to represent features using a bidirectional LSTM and improve the results of a greedy parser. Andor et al. (2016) present a mathematical proof that globally normalized models are more expressive than locally normalized counterparts and propose to use global normalization with beam search at both training and testing.

Our approach differs from all of the work mentioned above, in that it manages to improve results of Chen and Manning (2014) *without changing the architecture of the model nor the input representation*. The only substantial difference lies in the way the model is trained. In this respect, our research is most similar to training approaches using dynamic oracles (Goldberg and Nivre, 2012). Traditional static oracles can generate only one sequence of actions per sentence. A dynamic oracle gives *all* trajectories leading to the best possible result from every valid parse configuration. They can therefore be used to generate more training sequences including those containing errors. A drawback of this approach is that dynamic oracles have to be developed specifically for individual transition systems (e.g. arc-standard, arc-eager). Therefore, a large number of dynamic oracles have been developed in recent years (Goldberg and Nivre, 2012; Goldberg and Nivre, 2013; Goldberg et al., 2014; Gomez-Rodriguez et al., 2014; Björkelund and Nivre, 2015). In contrast, the reinforcement learning approach proposed in this paper is more general and can be applied to a variety of systems.

Zhang and Chan (2009) present the only study we are aware of that also uses reinforcement learning for dependency parsing. They compare their results to Nivre et al. (2006b) using the same features, but they also change the model and apply beam search. It is thus unclear to what extend their improvements are due to reinforcement learning.

Even though most approaches mentioned above improve the results reported by Chen and Manning (2014) and even more impressive results on

dependency parsing have been achieved since (notably, Andor et al. (2016)), Chen and Manning's parser provides a better baseline for our purposes. We aim at investigating the influence of reinforcement learning on error propagation and want to test this in a clean environment, where reinforcement learning does not interfere with other methods that address the same problem.

2.2 Reinforcement Learning

Reinforcement learning has been applied to several NLP tasks with success, e.g. agenda-based parsing (Jiang et al., 2012), semantic parsing (Berant and Liang, 2015) and simultaneous machine translation (Grissom II et al., 2014). To our knowledge, however, none of these studies investigated the influence of reinforcement learning on error propagation.

Learning to Search (L2S) is probably the most prominent line of research that applies reinforcement learning (more precisely, imitation learning) to NLP. Various algorithms, e.g. SEARN (Daumé III et al., 2009) and DAgger (Ross et al., 2011), have been developed sharing common high-level steps: a *roll-in* policy is executed to generate training states from which a *roll-out* policy is used to estimate the loss of certain actions. The concrete instantiation differs from one algorithm to another with choices including a referent policy (static or dynamic oracle), learned policy, or a mixture of the two. Early work in L2S focused on reducing reinforcement learning into binary classification (Daumé III et al., 2009), but newer systems favored regressors for efficiency (Chang et al., 2015, Supplementary material, Section B). Our algorithm APG is simpler than L2S in that it uses only one policy (pre-trained with standard supervised learning) and applies the existing classifier directly without reduction (the only requirement is that it is probabilistic). Nevertheless, our results demonstrate its effectiveness.

APG belongs to the family of policy gradient algorithms (Sutton et al., 1999), i.e. it maximizes the expected reward directly by following its gradient w.r.t. the parameters. The advantage of using a policy gradient algorithm in NLP is that gradient-based optimization is already widely used. REINFORCE (Williams, 1992; Ranzato et al., 2016) is a widely-used policy gradient algorithm but it is also well-known for suffering from high variance (Sutton et al., 1999).

We directly compare our approach to REINFORCE, whereas we leave a direct comparison to L2S for future work. Our experiments show that our algorithm results in lower variance and achieves better performance than REINFORCE.

Recent work addresses the approximation of reinforcement learning gradient in the context of machine translation. Shen et al. (2016)'s algorithm is roughly equivalent to the combination of an oracle and random sampling. Their approach differs from ours, because it does not retain memory across iteration as in our best-performing model (see Section 3.4).

2.3 Reinforcement and error propagation

As mentioned above, previous work that applied reinforcement learning to NLP has, to our knowledge, not shown that it improved results by reducing error propagation.

Work on identifying the impact of error propagation in parsing is rare, Ng and Curran (2015) being a notable exception. They provide a detailed error analysis for parsing and classify which kind of parsing errors are involved with error propagation. There are four main differences between their approaches and ours. First, Ng and Curran correct arcs in the tree and our algorithm corrects decisions of the parsing algorithm. Second, our approach distinguishes between cases where one erroneous action deterministically leads to multiple erroneous arcs and cases where an erroneous action leads to conditions that indirectly result in further errors (see Section 5.1 for a detailed explanation). Third, Ng and Curran's algorithm corrects all erroneous arcs that are the same type of parsing error and point out that they cannot examine the interaction between multiple errors of the same type in a sentence. Our algorithm corrects errors incrementally and therefore avoids this issue. Finally, the classification and analysis presented in Ng and Curran (2015) are more extensive and detailed than ours. Our algorithm can, however, easily be extended to perform similar analysis. Overall, Ng and Curran's approach for error analysis and ours are complementary. Combining them and applying them to various systems would form an interesting direction for future work.

3 A Reinforced Greedy Parser

This section describes the systems used in our experiments. We first describe the arc-standard al-

Step	Transition	Stack	Buffer	Arcs
0		<ROOT>	*waves hit ... Big Board*	∅
1	SHIFT	<ROOT> *waves*	*hit stocks ... Big Board*	∅
2	SHIFT	<ROOT> *waves hit*	*stocks themselves ... Big Board*	∅
3	$\text{LEFT}_{\text{nsubj}}$	<ROOT> *hit*	*stocks themselves ... Big Board*	A_1 = { *hit* $\xrightarrow{\text{nsubj}}$ *waves*}
4	SHIFT	<ROOT> *hit stocks*	*themselves on the Big Board*	A_1
5	SHIFT	<ROOT> *hit stocks themselves*	*on the Big Board*	A_1
6	$\text{RIGHT}_{\text{dep}}$	<ROOT> *hit stocks*	*on the Big Board*	A_2 = A_1∪ { *stock* $\xrightarrow{\text{dep}}$ *themselves*}
7	$\text{RIGHT}_{\text{dobj}}$	<ROOT> *hit*	*on the Big Board*	A_3 = A_2∪ { *hit* $\xrightarrow{\text{dobj}}$ *stock*}

Table 1: Parsing oracle walk-through

gorithm, because familiarity with it helps to understand our error propagation analysis. Next, we briefly point out the main differences between the arc-standard algorithm and the alternative algorithms we experimented with (arc-eager and swap-standard). We then outline the traditional and our novel machine learning approaches. The features we used are identical to those described in Chen and Manning (2014). We are not aware of research identifying the best feature for a neural parser with arc-eager or swap-standard so we use the same features for all transition systems.

3.1 Transition-Based Dependency Parsing

In an arc-standard system (Nivre, 2004), a parsing configuration consists of a triple $\langle \Sigma, \beta, A \rangle$, where Σ is a stack, β is a buffer containing the remaining input tokens and A are the dependency arcs that are created during parsing process. At initiation, the stack contains only the root symbol (Σ = [ROOT]), the buffer contains the tokens of the sentence ($\beta = [w_1, ..., w_n]$) and the set of arcs is empty ($A = \emptyset$).

The arc-standard system supports three transitions. When σ_1 is the top element and σ_2 the second element on the stack, and β_1 the first element of the buffer:[1]

LEFT$_l$ adds an arc $\sigma_1 \xrightarrow{l} \sigma_2$ to A and removes σ_2 from the stack.

RIGHT$_l$ adds an arc $\sigma_2 \xrightarrow{l} \sigma_1$ to A and removes σ_1 from the stack.

SHIFT moves β_1 to the stack.

When the buffer is empty, the stack contains only the root symbol and A contains a parse tree, the configuration is completed. For a sentence of N_w tokens, a full parse takes $2N_w$ + 1 transitions to complete (including the initiation). Figure 1 provides the gold parse tree for a (simplified) example from the Penn Treebank. The steps taken to create the dependencies between the sentence's head word *hit* and its subject and direct object are provided in Table 1.

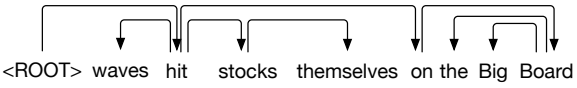

Figure 1: Correct dependencies for a simplified example from Penn TreeBank

To demonstrate that reinforcement learning can train different systems, we also carried out experiments with arc-eager (Nivre, 2003) and swap-standard (Nivre, 2009). Arc-eager is designed for incremental parsing and included in the popular MaltParser (Nivre et al., 2006a). Swap-standard is a simple and effective solution to unprojective dependency trees. Because arc-eager does not guarantee complete parse trees, we used a variation that employs an action called UNSHIFT to resume processing of tokens that would otherwise not be attached to a head (Nivre and Fernández-González, 2014).

3.2 Training with a Static Oracle

In transition-based dependency parsing, it is common to convert a dependency treebank $\mathcal{D} \ni (x, y)$ into a collection of input features $s \in \mathcal{S}$ and corresponding gold-standard actions $a \in \mathcal{A}$ for training, using a static oracle $\mathcal{O}$. In Chen and Manning (2014), a neural network works as a function mapping input features to probabilities of actions: $f_{NN} : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. The neural network is trained to minimize negative log-likelihood loss

[1] Naturally, the transitions LEFT_l and RIGHT_l can only take place if the stack contains at least two elements and SHIFT can only occur when there is at least one element on the buffer.

on the converted treebank:

$$\mathcal{L} = \sum_{(x,y)\in\mathcal{D}} \sum_{(s,a)\in\mathcal{O}(x,y)} -\log f_{NN}(s,a;\theta) \quad (1)$$

3.3 Reinforcement Learning

Following Maes et al. (2009), we view transition-based dependency parsing as a deterministic Markov Decision Process. The problem is summarized by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, r \rangle$ where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ contains all possible actions, $\mathcal{T}$ is a mapping $\mathcal{S}\times\mathcal{A}\rightarrow\mathcal{S}$ called *transition function* and $r : \mathcal{S}\times\mathcal{A}\rightarrow\mathbb{R}$ is a reward function.

A state corresponds to a configuration and is summarized into input features. Possible actions are defined for each transition system described in Section 3.1. We keep the training approach simple by using only one reward $r(\bar{y})$ at the end of each parse.

Given this framework, a stochastic policy guides our parser by mapping each state to a probabilistic distribution of actions. During training, we use function f_{NN} described in Section 3.2 as a stochastic policy. At test time, actions are chosen in a greedy fashion following existing literature. We aim at finding the policy that maximizes the expected reward (or, equivalently, minimizes the expected loss) on the training dataset:

$$\text{maximize } \eta = \sum_{(x,y)\in\mathcal{D}} \sum_{a_{1:m}\sim f} r(\bar{y}) \prod_{i=1}^{m} f_{NN}(s_i, a_i; \theta) \quad (2)$$

where $a_{1:m}$ is a sequence of actions obtained by following policy f_{NN} until termination and $s_{1:m}$ are corresponding states (with s_{m+1} being the termination state).

3.4 Approximate Policy Gradient

Gradient ascent can be used to maximize the expected reward in Equation 2. The gradient of expected reward w.r.t. parameters is (note that $\mathrm{d}z = z\mathrm{d}(\log z)$):

$$\frac{\partial\eta}{\partial\theta} = \sum_{(x,y)\in\mathcal{D}} \sum_{a_{1:m}\sim f_{NN}} r(\bar{y}) \prod_{i=1}^{m} f_{NN}(s_i, a_i) \sum_{i=1}^{m} \frac{\partial}{\partial\theta} \log f_{NN}(s_i, a_i; \theta) \quad (3)$$

Because of the exponential number of possible trajectories, calculating the gradient exactly is not possible. We propose to replace it by an approximation (hence the name *Approximate* Policy Gradient) by summing over a small subset U of trajectories. Following common practice, we also use a baseline $b(y)$ that only depends on the correct dependency tree. The parameter is then updated by following the approximate gradient:

$$\Delta\theta \propto \sum_{(x,y)\in\mathcal{D}} \sum_{a_{1:m}\in U} (r(\bar{y}) - b(y)) \prod_{i=1}^{m} f_{NN}(s_i, a_i) \sum_{i=1}^{m} \frac{\partial}{\partial\theta} \log f_{NN}(s_i, a_i; \theta) \quad (4)$$

Instead of sampling one trajectory at a time as in REINFORCE, Equation 4 has the advantage that sampling over multiple trajectories could lead to more stable training and higher performance. To achieve that goal, the choice of U is critical. We empirically evaluate three strategies:

RL-ORACLE: only includes the oracle transition sequence.

RL-RANDOM: randomly samples k distinct trajectories at each iteration. Every action is sampled according to f_{NN}, i.e. preferring trajectories for which the current policy assigns higher probability.

RL-MEMORY: samples randomly as the previous method but retains k trajectories with highest rewards across iterations in a separate memory. Trajectories are "forgotten" (removed) randomly with probability ρ before each iteration.[2]

Intuitively, trajectories that are more likely and produce higher rewards are better training examples. It follows from Equation 3 that they also bear bigger weight on the true gradient. This is the rationale behind RL-RANDOM and RL-ORACLE. For a suboptimal parser, however, these objectives sometimes work against each other. RL-MEMORY was designed to find the right balance between them. It is furthermore important that the parser is pretrained to ensure good samples. Algorithm 1 illustrates the procedure of training a dependency parser using the proposed algorithms.

[2] We assign a random number (drawn uniformly from $[0, 1]$) to each trajectory in memory and remove those assigned a number less than ρ.

$MemorySeqs \leftarrow \emptyset$;
foreach *training batch b* **do**
  foreach *sentence s* $\in$ *b* **do**
    $OracleSeq \leftarrow$ Oracle(s);
    $SystemSeqs \leftarrow$ (sample k parsing transition sequences for s);
    if *RL-Oracle* **then**
      ComputeGradients($OracleSeq$);
    else if *RL-Random* **then**
      ComputeGradients($SystemSeqs$);
    else if *RL-Memory* **then**
      $m \leftarrow MemorySeqs[s]$;
      foreach $q \in m$ **do**
        if *RandomNumber()* < ρ **then**
          Remove q from m;
        end
      end
      foreach $q \in$ *SystemSeqs* **do**
        if |*m*| < *k* **then**
          Insert q into m;
        else
          $p \leftarrow$ (sequence with smallest reward in m);
          if *reward(q)* > *reward(p)* **then**
            Replace p by q in m;
          end
        end
      end
      ComputeGradients(m);
  end
  Perform one gradient descent step;
end

Algorithm 1: Training a dependency parser with approximate policy gradient.

4 Reinforcement Learning Experiments

We first train a parser using a supervised learning procedure and then improve its performance using APG. We empirically tested that training a second time with supervised learning has little to no effect on performance.

4.1 Experimental Setup

We use PENN Treebank 3 with standard split (training, development and test set) for our experiments with arg-standard and arg-eager. Because the swap-standard parser is mainly suited for non-projective structures, which are rare in the PENN Treebank, we evaluate this parser on the German section of the SPMRL dataset. For PENN Treebank, we follow Chen and Manning's preprocessing steps. We also use their pretrained model[3] for arc-standard and train our own models in similar settings for other transition systems.

	Arc-standard		Arc-eager		Swap-standard	
	UAS	**LAS**	**UAS**	**LAS**	**UAS**	**LAS**
SL	91.3	89.4	88.3	85.8	84.3	81.3
RE	91.9	90.2	89.7	87.2	87.5	84.4
RL-O	91.8	90.2	88.9	86.5	86.8	83.9
RL-R	**92.2**	**90.6**	89.4	87.0	87.5	84.5
RL-M	**92.2**	**90.6**	**89.8**	**87.4**	**87.6**	**84.6**

Table 2: Comparing training methods on PENN Treebank (arc-standard and arc-eager) and German part of SPMRL-2014 (swap-standard).

For reinforcement learning, we use AdaGrad for optimization. We do not use dropout because we observed that it destablized the training process. The reward $r(\bar{y})$ is the number of correct labeled arcs (i.e. LAS multiplied by number of tokens).[4] The baseline is fixed to half the number of tokens (corresponding to a 0.5 LAS score). As training takes a lot of time, we tried only few values of hyperparameters on the development set and picked $k = 8$ and $\rho = 0.01$. 1,000 updates were performed (except for REINFORCE which was trained for 8,000 updates) with each training batch contains 512 randomly selected sentences. The Stanford dependency scorer[5] was used for evaluation.

4.2 Effectiveness of Reinforcement Learning

Table 2 displays the performance of different approaches to training dependency parsers. Although we used Chen and Manning (2014)'s pretrained model and Stanford open-source software, the results of our baseline are slightly worse than what is reported in their paper. This could be due to minor differences in settings and does not affect our conclusions.

Across transition systems and two languages, APG outperforms supervised learning, verifying our hypothesis. Moreover, it is not simply because the learners are exposed to more examples than their supervised counterparts. RL-ORACLE

[3]We use `PTB_Stanford_params.txt.gz` downloaded from `http://nlp.stanford.edu/software/nndep.shtml` on December 30th, 2015.

[4]Punctuation is not taken into account, following Chen and Manning (2014).

[5]Downloaded from `http://nlp.stanford.edu/software/lex-parser.shtml`.

is trained on exactly the same examples as the standard supervised learning system (SL), yet it is consistently superior. This can only be explained by the superiority of the reinforcement learning objective function compared to negative log-likelihood.

The results support our hypothesis that APG is better than REINFORCE (abbreviated as RE in Table 2) as RL-MEMORY always outperforms the classical algorithm and the other two heuristics do in two out of three cases. The usefulness of training examples that contain errors is evident through the better performance of RL-RANDOM and RL-MEMORY in comparison to RL-ORACLE.

Table 3 shows the importance of samples for RL-RANDOM. The algorithm hurts performance when only one sample is used whereas training with two or more samples improves the results. The difference cannot be explained by the total number of observed samples because one-sample training is still worse after 8,000 iterations compared to a sample size of 8 after 1,000 iterations. The benefit of added samples is twofold: increased performance and decreased variance. Because these benefits saturate quickly, we did not test sample sizes beyond 32.

	Dev		Test		Test std.	
	UAS	**LAS**	**UAS**	**LAS**	**UAS**	**LAS**
SL	91.5	89.6	91.3	89.4	-	-
RE	92.1*	90.4*	91.9*	90.2*	0.04	0.05
1	91.2*	89.1*	91.0*	88.9*	0.12	0.15
2	91.8*	90.0*	91.6*	89.9*	0.09	0.09
4	92.2*	90.5*	92.0*	90.4*	0.09	0.08
8	**92.4***	**90.8***	92.2*	**90.6***	0.03	0.05
16	**92.4**	**90.8**	92.2	**90.6**	-	-
32	**92.4**	**90.8**	**92.3**	**90.6**	-	-

Table 3: Parsing accuracy of RL-RANDOM (arc-standard) with different sample sizes compared to supervised learning (SL) and REINFORCE (RE). *: significantly different from SL with $p < 0.001$

5 Error Propagation Experiment

We hypothesized that reinforcement learning avoids error propagation. In this section, we describe our algorithm and the experiment that identifies error propagation in the arc-standard parsers.

5.1 Error Propagation

Section 3.1 explained that a transition-based parser goes through the sentence incrementally and must select a transition from [SHIFT, LEFT$_l$, RIGHT$_l$] at each step. We use the term *arc error* to refer to an erroneous arc in the resulting tree. The term *decision error* refers to a transition that leads to a loss in parsing accuracy. Decision errors in the parsing process lead to one or more arc errors in the resulting tree. There are two ways in which a single decision error may lead to multiple arc errors. First, the decision can deterministically lead to more than one arc error, because (e.g.) an erroneously formed arc also blocks other correct arcs. Second, an erroneous parse decision changes some of the features that the model uses for future decisions and these changes can lead to further (decision) errors down the road.

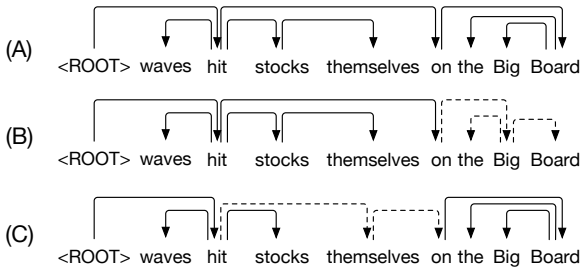

Figure 2: Three dependency graphs: gold (A), arc errors caused by one decision error (B) and arc errors caused by multiple decision errors (C).

We illustrate both cases using two incorrect derivations presented in Figure 2. The original gold tree is repeated in (A). The dependency graph in Figure 2 (B) contains three erroneous dependency arcs (indicated by dashed arrows). The first error must have occurred when the parser executed RIGHT$_{\text{amod}}$ creating the arc *Big* → *Board*. After this error, it is impossible to create the correct relations *on* → *Board* and *Board* → *the*. The wrong arcs *Big* → *the* and *on* → *Big* are thus all the result of a single decision error.

Figure 2 (C) represents the dependency graph that is actually produced by our parser.[6] It contains two erroneous arcs: *hit* → *themselves* and *themselves* → *on*. Table 4 provides a possible sequence of steps that led to this derivation, starting from the moment *stocks* was added to the stack (Step 4). The first error is introduced in Step 5', where *hit* combines with *stocks* before *stocks* has picked up its dependent *themselves*. At that point, *themselves* can no longer be combined with the right head. The proposition *on*, on the other hand, can

[6]The example is a fragment of a more complex sentence consisting of 33 tokens. The parser does provide the correct output when is analyzes this sequence in isolation.

Step	Transition	Stack	Buffer	Arcs
4	SHIFT	<ROOT> *hit stocks*	*themselves on the Big Board*	A_1
5'	$RIGHT_{dobj}$	<ROOT> *hit*	*themselves on the Big Board*	$A_2 = A_1 \cup \{hit \xrightarrow{dobj} stock\}$
6'	SHIFT	<ROOT> *hit themselves*	*on the Big Board*	A_2
7'	SHIFT	<ROOT> *hit themselves on*	*the Big Board*	A_2
...				
10'	SHIFT	<ROOT> *hit themselves on the Big Board*		A_2
11'	$LEFT_{nn}$	<ROOT> *hit themselves on the Board*		$A_3 = A_2 \cup \{Board \xrightarrow{nn} Big\}$
12'	$LEFT_{det}$	<ROOT> *hit themselves on Board*		$A_4 = A_3 \cup \{Board \xrightarrow{det} the\}$
13'	$RIGHT_{pobj}$	<ROOT> *hit themselves on*		$A_5 = A_4 \cup \{on \xrightarrow{pobj} Board\}$
14'	$RIGHT_{dep}$	<ROOT> *hit themselves*		$A_6 = A_5 \cup \{themselves \xrightarrow{dep} on\}$

Table 4: Possible parsing walk-through with error

still be combined with the correct head. This error is introduced in Step 7', where the parser moves *on* to the stack rather than creating an arc from *hit* to *themselves*.[7] There are thus two decision errors that lead to the arc errors in Figure 2 (C). The second decision error can, however, be caused indirectly by the first error. If a decision error causes additional decision errors later in the parsing process, we talk of error propagation. This cannot be known just by looking at the derivation.

5.2 Examining the impact of decision errors

We examine the impact of individual decision errors on the overall parse results in our test set by combining a dynamic oracle and a recursive function. We use a dynamic oracle based on Goldberg et al. (2014) which gives us the overall loss at any point during the derivation. The loss is equal to the minimal number of arc errors that will have been made once the parse is complete. We can thus deduce how many arc errors are deterministically caused by a given decision error.

The propagation of decision errors cannot be determined by simply examining the increase in loss during the parsing process. We use a recursive function to identify whether a particular parse suffered from this. While parsing the sentence, we register which decisions lead to an increase in loss. We then recursively reparse the sentence correcting one additional decision error during each run until the parser produces the gold. If each erroneous decision has to be corrected in order to arrive at the gold, we assume the decision errors are independent of each other. If, on the other hand, the correction of a specific decision also fixes other decisions down the road, the original parse suffers from error propagation.

	SL	RL-O	RL-R	RL-M
Total Loss	7069	6227	6042	6144
Dec. Errors	5177	4410	4345	4476
Err. Prop.	1399	1124	992	1035
New errors	411	432	403	400
Loss/error	1.37	1.41	1.39	1.37
Err. Prop. (%)	27.0	25.5	22.8	23.1

Table 5: Overview of average impact of decision errors

The results are presented in Table 5. *Total Loss* indicates the number of arc errors in the corpus, *Dec. Errors* the number of decision errors and *Err. Prop.* the number of decision errors that were the result of error propagation. This number was obtained by comparing the number of decision errors in the original parse to the number of decision errors that needed to be fixed to obtain the gold parse. If less errors had to be fixed than originally present, we counted the difference as error propagation. Note that fixing errors sometimes leads to new decision errors during the derivation. We also counted the cases where more decision errors needed to be fixed than were originally present and report them in Table 5.[8]

[7] Note that technically, *on* can still become a dependent of *hit*, but this can only happen if *on* becomes the head of *themselves* which would also be an error.

[8] We ran an alternative analysis where we counted all cases where fixing one decision error in the derivation reduced the overall number of decision errors in the parse by more than one. Under this alternative analysis, similar reductions in the proportion of error propagation were observed for reinforcement learning.

On average, decision errors deterministically lead to more than one arc error in the resulting parse tree. This remains stable across systems (around 1.4 arc errors per decision error). We furthermore observe that the proportion of decision errors that are the result of error propagation has indeed reduced for all reinforcement learning models. Among the errors avoided by APG, 35.9% were propagated errors for RL-ORACLE, 48.9% for RL-RANDOM, and 51.9% for RL-MEMORY. These percentages are all higher than the proportion of propagated errors occurring in the corpus parsed by SL (27%). This outcome confirms our hypothesis that reinforcement learning is indeed more robust for making decisions in imperfect environments and therefore reduces error propagation.

6 Conclusion

This paper introduced Approximate Policy Gradient (APG), an efficient reinforcement learning algorithm for NLP, and applied it to a high-performance greedy dependency parser. We hypothesized that reinforcement learning would be more robust against error propagation and would hence improve parsing accuracy.

To verify our hypothesis, we ran experiments applying APG to three transition systems and two languages. We furthermore introduced an experiment to investigate which portion of errors were the result of error propagation and compared the output of standard supervised machine learning to reinforcement learning. Our results showed that: (a) reinforcement learning indeed improved parsing accuracy and (b) propagated errors were over-represented in the set of avoided errors, confirming our hypothesis.

To our knowledge, this paper is the first to show experimentally that reinforcement learning can reduce error propagation in an NLP task. This result was obtained by a straight-forward implementation of reinforcement learning. Furthermore, we only applied reinforcement learning in the training phase, leaving the original efficiency of the model intact. Overall, we see the outcome of our experiments as an important first step in exploring the possibilities of reinforcement learning for tackling error propagation.

Recent research on parsing has seen impressive improvement during the last year achieving UAS around 94% (Andor et al., 2016). This improvement is partially due to other approaches that, at least in theory, address error propagation, such as beam search. Both the reinforcement learning algorithm and the error propagation study we developed can be applied to other parsing approaches. There are two (related) main questions to be addressed in future work in the domain of parsing. The first addresses whether our method is complementary to alternative approaches and could also improve the current state-of-the-art. The second question would address the impact of various approaches on error propagation and the kind of errors they manage to avoid (following Ng and Curran (2015)).

APG is general enough for other structured prediction problems. We therefore plan to investigate whether we can apply our approach to other NLP tasks such as coreference resolution or semantic role labeling and investigate if it can also reduce error propagation for these tasks.

The source code of all experiments is publicly available at `https://bitbucket.org/cltl/redep-java`.

Acknowledgments

The research for this paper was supported by the Netherlands Organisation for Scientific Research (NWO) via the Spinoza-prize Vossen projects (SPI 30-673, 2014-2019) and the VENI project *Reading between the lines* (VENI 275-89-029). Experiments were carried out on the Dutch national e-infrastructure with the support of SURF Cooperative. We would like to thank our friends and colleagues Piek Vossen, Roser Morante, Tommaso Caselli, Emiel van Miltenburg, and Ngoc Do for many useful comments and discussions. We would like to extend our thanks the anonymous reviewers for their feedback which helped improving this paper. All remaining errors are our own.

References

Chris Alberti, David Weiss, Greg Coppola, and Slav Petrov. 2015. Improved Transition-Based Parsing and Tagging with Neural Networks. In *EMNLP 2015*, pages 1354–1359. ACL.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally Normalized Transition-Based Neural Networks. *arXiv.org*, cs.CL.

Jonathan Berant and Percy Liang. 2015. Imitation Learning of Agenda-based Semantic Parsers. *TACL*, 3:545–558.

Anders Björkelund and Joakim Nivre. 2015. Non-Deterministic Oracles for Unrestricted Non-Projective Transition-Based Dependency Parsing. In *IWPT 2015*, pages 76–86. ACL.

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

Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser using Neural Networks. In *EMNLP 2014*, pages 740–750. ACL.

Hal Daumé III, John Langford, and Daniel Marcu. 2009. Search-based Structured Prediction. *Machine Learning*, 75(3):297–325, 6.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. 2015. Transition-Based Dependency Parsing with Stack Long Short-Term Memory. In *ACL 2015*, pages 334–343.

Daniel Gildea and Martha Palmer. 2002. The Necessity of Parsing for Predicate Argument Recognition. In *ACL 2002*, pages 239–246. ACL.

Yoav Goldberg and Joakim Nivre. 2012. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *COLING 2012*, pages 959–976.

Yoav Goldberg and Joakim Nivre. 2013. Training Deterministic Parsers with Non-Deterministic Oracles. In *TACL 2013*, volume 1, pages 403–414.

Yoav Goldberg, Francesco Sartorio, and Giorgio Satta. 2014. A tabular method for dynamic oracles in transition-based parsing. In *TACL 2014*, volume 2, pages 119–130.

Carlos Gomez-Rodriguez, Francesco Sartorio, and Giorgio Satta. 2014. A Polynomial-Time Dynamic Oracle for Non-Projective Dependency Parsing. In *EMNLP 2014*, pages 917–927. ACL.

Alvin C. Grissom II, Jordan Boyd-Graber, He He, John Morgan, and Hal Daume III. 2014. Don't Until the Final Verb Wait: Reinforcement Learning for Simultaneous Machine Translation. In *EMNLP 2014*, pages 1342–1352.

Dan Han, Pascual Martínez-Gómez, Yusuke Miyao, Katsuhito Sudoh, and Masaaki Nagata. 2013. Effects of parsing errors on pre-reordering performance for Chinese-to-Japanese SMT. *PACLIC 27*, pages 267–276.

Matthew Honnibal and Mark Johnson. 2015. An Improved Non-monotonic Transition System for Dependency Parsing. In *EMNLP 2015*, pages 1373–1378. ACL.

Jiarong Jiang, Adam Teichert, Hal Daumé III, and Jason Eisner. 2012. Learned Prioritization for Trading Off Accuracy and Speed. *ICML workshop on Inferning: Interactions between Inference and Learning*, (0964681):1–9.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations. *CoRR*, abs/1603.0.

Zhenghua Li, Min Zhang, and Wenliang Chen. 2014. Ambiguity-aware Ensemble Training for Semi-supervised Dependency Parsing. In *ACL 2014*, pages 457–467.

Francis Maes, Ludovic Denoyer, and Patrick Gallinari. 2009. Structured prediction with reinforcement learning. *Machine Learning*, (77):271–301.

Ryan McDonald and Joakim Nivre. 2007. Characterizing the Errors of Data-Driven Dependency Parsing Models. In *EMNLP-CoNLL 2007*.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *HLT-EMNLP 2005*, pages 523–530. Association for Computational Linguistics.

Dominick Ng and James R Curran. 2015. Identifying Cascading Errors using Constraints in Dependency Parsing. In *ACL-IJCNLP*, pages 1148–1158, Beijing. ACL.

Joakim Nivre and Daniel Fernández-González. 2014. Arc-eager Parsing with the Tree Constraint. *Computational Linguistics*, 40(2):259–267, 6.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006a. MaltParser: A data-driven parser-generator for dependency parsing. In *LREC 2006*, volume 6, pages 2216–2219.

Joakim Nivre, Johan Hall, Jens Nilsson, Gülşen Eryiğit, and Svetoslav Marinov. 2006b. Labeled pseudo-projective dependency parsing with support vector machines. In *CoNLL 2006*, pages 221–225. ACL.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Eryiğit Gülşen, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(02):95–135.

Joakim Nivre. 2003. An Efficient Algorithm for Projective Dependency Parsing. In *IWPT 2003*, pages 149–160.

Joakim Nivre. 2004. Incrementality in Deterministic Dependency Parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*.

Joakim Nivre. 2009. Non-projective Dependency Parsing in Expected Linear Time. In *ACL-IJCNLP 2009*, pages 351–359, Stroudsburg, PA, USA. ACL.

Chris Quirk and Simon Corston-Oliver. 2006. The impact of parse quality on syntactically-informed statistical machine translation. In *EMNLP 2006*, pages 62–69, Sydney, Australia. ACL.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. Sequence Level Training with Recurrent Neural Networks. *ICLR*, pages 1–15.

Stephane Ross, Geoffrey J Gordon, and J Andrew Bagnell. 2011. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. *AISTATS*, 15:627–635.

Djamé Seddah, Sandra Kübler, and Reut Tsarfaty. 2014. Introducing the SPMRL 2014 Shared Task on Parsing Morphologically-Rich Languages. In *Proceedings of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*, pages 103–109.

Shiqi Shen, Yong Cheng, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu. 2016. Minimum Risk Training for Neural Machine Translation. In *ACL 2016*, pages 1683–1692, Berlin, Germany. ACL.

Hyun-Je Song, Jeong-Woo Son, Tae-Gil Noh, Seong-Bae Park, and Sang-Jo Lee. 2012. A Cost Sensitive Part-of-Speech Tagging: Differentiating Serious Errors from Minor Errors. In *ACL 2012*, pages 1025–1034. ACL.

Richard S. Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. 1999. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *NIPS 1999*, pages 1057–1063.

David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. 2015. Structured Training for Neural Network Transition-Based Parsing. In *ACL-IJCNLP 2015*, pages 323–333. ACL.

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical Dependency Analysis with Support Vector Machines. In *Proceedings of IWPT*, pages 195–206.

Bishan Yang and Claire Cardie. 2013. Joint Inference for Fine-grained Opinion Extraction. In *ACL 2013*, pages 1640–1649. ACL.

Lidan Zhang and Kwok Ping Chan. 2009. Dependency Parsing with Energy-based Reinforcement Learning. In *IWPT 2009*, pages 234–237. ACL.