Improving the Neural Dependency Parser

Chuanbo Pan[*]
Department of Computer Science
Stanford University
chuanbo@stanford.edu

Shane T. Barratt[*]
Department of Electrical Engineering
Stanford University
sbarratt@stanford.edu

Jeffrey J. Barratt[*]
Department of Computer Science
Stanford University
jbarratt@stanford.edu

Abstract

In this paper, we attempt to improve the neural dependency parser proposed by Chen and Manning. We first experiment with several neural network architecture and optimization techniques to achieve state-of-the-art performance. We then adopt the DAGGER (Dataset Aggregation) algorithm to improve the training of the neural dependency parser in the low-data regime. Finally, we experiment with training the neural dependency parser directly via reinforcement learning using Deep-Q Networks, removing the need for a supervised oracle.

1 Introduction

Dependency parsing is one of the central tasks in computational natural language processing. The result of parsing is a syntactic representation of a sentence, which can be useful for several downstream tasks such as machine translation, relation extraction and question answering. As a result, dependency parsing has garnered significant interest in the last decade.

The first data-driven approaches to dependency parsing use the arc-standard system [1] and a support vector machine to choose transitions [2]. A common non data-driven approach is using integer linear programming [3].

In 2014, neural networks took over dependency parsing. Chen and Manning [4] proposed a dependency parser that can parse more than 1000 sentences per second and achieves a $92.2\,\%$ unlabeled attachment score on the Engligh Penn Treebank (PTB) dataset. They use the arc-standard system [1] and a single-layer neural network that chooses actions based on the state of the parser. The neural network takes as input word vectors of the adjacent words on the stack/buffer, and learned dense representations of the parts of speech and dependencies. Their work is considered the state-of-the-art linear time parser.

There is a tradeoff between speed and accuracy of a parser. On one end, the work by Chen and Manning sacrifice accuracy by performing greedy parsing but their parser is linear time. On the other end of the spectrum are methods that use beam search and graph-based parsers. Andor *et al.* [5] use beam search to maintain multiple hypotheses and improve the UAS to $94.61\,\%$. Other work by Kiperwasser and Goldberg [6] and Dozat and Manning [7] use neural attention in graph-based transition parsers. Kuncoro *et al.* [8] use recurrent neural network grammars to achieve the state-of-the-art UAS of $95.8\,\%$.

[*]Equal contribution.

Le and Fokkens [9] were the first to explore using reinforcement learning to fine-tune the neural dependency parser. They found that the accuracy of Chen and Manning can be improved to $92.3\,\%$ by fine-tuning the learned policy using an approximate policy gradient algorithm. This improvement seems to be within the reported standard error, casting doubt on the validity of their claims. They also did not experiment with training the parser with reinforcement learning from scratch, which we believe is an experiment worth doing.

Acquiring labeled natural language data is a time-consuming and expensive task. Therefore, it is important that the community develops data-efficient algorithms to avoid requiring large amounts of labeled data. We will see that the performance of neural dependency parsing in the low-data regime (10–10 000 samples) can be significantly improved by using an algorithm called DAGGER [10]. This result has important implications for training neural dependency parsers on small datasets. We also believe that DAGGER can be successfully applied to other natural language processing tasks.

2 Dependency Parsing

In this section, we describe the dependency parsing paradigm, the dataset used, and common evaluation metrics. This section will also introduce notation used throughout the paper.

2.1 Transition-Based Dependency Parsing

Dependency parsing involves deriving the dependency structure of a sentence. There are many ways to do this. One way, which we focus on here, is greedy transition-based parsing.

The arc standard system [1] is central to this approach. The arc-standard transition-based parser starts with a stack $S = [\text{ROOT}]$ initially consisting only of the "ROOT" token, a buffer $B = [w_1, \ldots, w_n]$ consisting of the words of the sentence, and a set of dependencies $A = \emptyset$. At each step, the parser decides between three actions: SHIFT, Left-Arc, and Right-Arc. Assuming that $S = [\text{ROOT}, s_1, \ldots, s_{m-1}, s_m]$ and $B = [w_1, \ldots, w_n]$,

- SHIFT pops w_1 from B and appends it to S.
- Left-Arc adds (s_m, s_{m-1}) to A and pops s_{m-1} from S.
- Right-Arc adds (s_{m-1}, s_m) to A and pops s_m from S.

The parser is guaranteed to result in a dependency tree involving all words in the sentence and build *projective* dependency trees. Projective dependency trees have no crossing dependency arcs when the words are laid out in their linear order.

2.2 Neural Dependency Parsing

In order to choose between the three actions at each step, features are extracted based on the elements in the stack, buffer, and list of dependencies (details are in [2]). These features are discrete, representing words c_w, parts of speech c_{pos} (POS), and dependencies c_{dep}. The features are then fed as input to the following one-layer neural network (E_w is the embedding matrix corresponding to 50-dimensional word vectors from [11], and E_1 and E_2 are trainable 50-dimensional embedding matrices):

$$
\begin{aligned}
x &= [E_w c_w, E_1 c_{\text{pos}}, E_2 c_{\text{dep}}] \\
h &= \text{relu}(W_1 x + b_1) \\
h_d &= \text{dropout}(x, d) \\
p &= \text{softmax}(U h_d + b).
\end{aligned}
$$

To train the network, first we generate training examples $\{(c_i, t_i)\}_{i=1}^m$ from the training sentences and their parse trees using a "shortest stack" oracle. Then, the training objective is to minimize the cross-entropy loss plus a ℓ_2–regularization term

$$L(\theta) = -\sum_{i=1}^{m} \log p_{t_i} + \frac{\lambda}{2}\|\theta\|^2.$$

At test time, the parser chooses the action $\operatorname{argmax}_i p_i$ at each step.

Table 1: Performances of Different Neural Networks for Neural Dependency Parsing

Dropout Placement	Hidden Size	Number of Layers	Test UAS
No Dropout	200	2	89.36
		4	89.01
	400	2	89.62
		4	89.32
Last Layer	200	2	89.30
		4	89.19
	400	2	89.47
		4	**89.97**
Every Other Layer	200	2	89.09
		4	89.45
	400	2	89.78
		4	89.74
Every Layer	200	2	89.19
		4	89.36
	400	2	89.10
		4	89.47

2.3 Dataset and Evaluation

We conduct our experiments on the English Penn Treebank (PTB) dataset. We follow the standard splits of PTB. PTB has $39\,832$ labeled training sentences, 1700 dev sentences, and 2416 test sentences. Parsers are evaluated based on the UAS, which calculates the percentage of correct dependencies idenfied by the parser.

3 Improving Performance

The most common form of neural dependency parser simply employs imitation learning on a large dataset of sentences with known parsing schemes, as discussed in the above sections. While teams such as Chen and Manning [4] use a simple network containing only a single hidden layer in their models, not many papers have analyzed the relative benefits of different network architectures. This analysis could prove useful for future investigations into dependency parsing; choice of network architecture can vary the final test UAS score greatly.

To investigate the effects of different structures of the network, we constructed a network similar to the one described in Chen and Manning [4] with different dropout structures, hyperparameters, and sizes/numbers of hidden layers. In addition to the changed variables, we use a constant learning rate of 1×10^{-3}, a weight decay coefficient of 1×10^{-8}, and a batch size of 2048 across 10 epochs. Early stopping is employed to reduce overfitting of the training set.

We tested this base network with three different characteristics varied between 2 and 4 values for a total of 16 networks. As seen in Table 1, we were able to reach a test UAS of just under 90 with the parameters specified. The best networks tended to be deep and wide, with dropout either in just the last layer, or every other layer.

4 Dataset Aggregation

The neural dependency parser is trained using the simplest form of imitation learning, behavioral cloning. In behavioral cloning, a set of training examples labeled by an expert is gathered and the policy is trained using supervised learning. Behavioral cloning suffers from compounding errors, where at test time the algorithm makes a mistake, reaches a stack/buffer state it has not seen at train time, and continues to make mistakes throughout the parsing. A theoretical analysis shows that the number of mistakes grows quadratically in T, the number of parsing steps [10].

However, there is a simple extension to behavioral cloning called Dataset Agreggation (DAGGER) that can yield better performance and comes with theoretical guarantees. The basic premise of

Algorithm 1 DAGGER. Π is the set of neural network policies. π^* is the oracle policy. Commonly, $\beta_1 = 1$ and $\beta_i = 0$ for $i = 2, \ldots, N$.

```
1: $\mathcal{D} \leftarrow \emptyset$.
2: Initialize $\hat{\pi}_1$ to any policy in $\Pi$.
3: **for** $t = 1, \ldots, N$ **do**
4:     Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$.
5:     Sample $K$ trajectories using $\pi_i$
6:     Get dataset $\mathcal{D}_i = \{(c, \pi^*(c))\}$ of visited states by $\pi_i$ and actions given by expert.
7:     Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$.
8:     Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$.
9: **end for**
10: **return** best $\pi_i$ on validation.
```

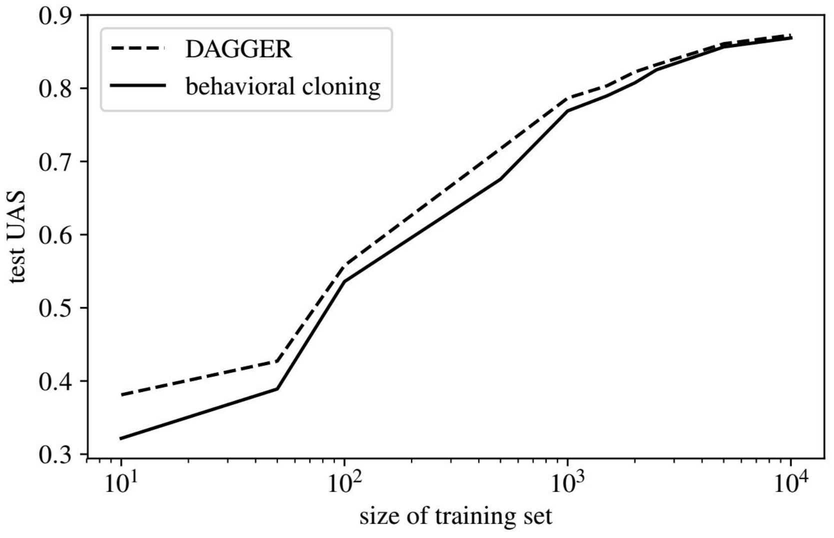


Figure 1: Performance of DAGGER algorithm vs. bsehavioral cloning.

DAGGER is to aggregate the dataset used for supervised learning with sample paths of the current policy labeled by the expert. This makes our dataset contain a set of inputs that the learned policy will likely encounter at test time, thus solving the problem of compounding error propagation. The steps required to carry out DAGGER are summarized in Algorithm 1.

We apply DAGGER to the task of training a neural dependency parser. The expert π^* is the oracle that was defined in Section 2. Instead of sampling K trajectories, we apply the parser π_i to all sentences/parses in our initial dataset. We do not reinitialize the policy every iteration and instead use the policy from the previous iteration.

We use a similar network to the one described in Chen and Manning [4]. We use two hidden layers each of size 200 with rectified linear activations, dropout after each hidden layer of .5, and a weight decay coefficient of 1×10^{-7}. Each inner training iteration involves 100 epochs with a batch size of 1024 and learning rate of 1×10^{-3} that is multiplied by 0.5 every 25 iterations. We run $N = 5$ DAGGER iterations.

To evaluate data efficiency, we varied the dataset size by taking the first n sentences in the training data. We then trained the parser using behavioral cloning and DAGGER with early stopping on the dev set, and evaluated their test UAS. Fig. 1 shows the results of this experiment. DAGGER is able to improve the test UAS by as much as 6 % with only 10 training examples. However, as the training set size increases, the benefit of DAGGER declines and suffers from decreasing marginal returns.

Algorithm 2 Deep-Q Network for Neural Dependency Parsing

```
1: Initialize replay memory $\mathcal{D}$ to capacity $k \cdot |\mathcal{D}_{sentences}|$
2: Initialize network parameters $\theta$ and target network parameters $\theta^-$
3: Pre-process sentence and known dependencies $S$
4: for $t = 1, \ldots, N$ do
5:     Shuffle and partition $S$ into $M$ equal partitions, $S_1, S_2, \ldots, S_M$
6:     for $p = 1, \ldots, M$ do
7:         Parse $S_p$ with network. With probability $\epsilon$, choose a random action when parsing
8:         Store parse results by sentence $\{(s_{p,i}, a_{p,i}, r_{p,i}, s_{p,i+1})\}_i$ in $\mathcal{D}_p$
9:         Update replay memory $\mathcal{D}$ with $\mathcal{D}_p$
10:        Sample parses $\mathcal{D}_{train}$ from $\mathcal{D}$
11:        For each $(s_i, a_i, r_i, s_{i+1})$, compute $y_i = \begin{cases} r_i & \text{If terminal } s' \\ r_i + \gamma \max_{a'} Q(s', a'|\theta^-) & \text{Otherwise} \end{cases}$
12:        Train with gradient descent to minimize $(y_i - Q(s_i, a_i|\theta))^2$
13:        After every $C$ steps, update $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$
14:    end for
15: end for
```

5 Reinforcement Learning

Reinforcement learning is a promising method to train neural dependency parsers as it is inherently a sequential decision making task. In the arc-standard system, there is a discrete set of (three) actions, and the UAS can be used as a reward signal. Furthermore, reinforcement learning does not require an oracle; all that is required is sentences and their gold parse trees. This paper focuses on applying Deep-Q learning and Deep-Q networks (DQN), a model free Q-Learning approach using deep neural networks by Mnih *et al.* [12], to training neural dependency parsers.

The original DQN application was the Arcade Learning Environment [13] and thus their algorithm as stated had to be modified for the dependency parsing "environment". Several modifications were made that can be seen in Algorithm 2. In addition, in order to stabilize training, we augmented the UAS reward with intermediate rewards. Furthermore, the final reward is the number of correctly parsed dependencies. That way, the reward scales according to the sentence length and is not unfairly decayed in longer sentences.

$$r(s, a, s') = \begin{cases} \text{\# Dependencies Correct} & \text{If terminal } s' \\ 1 & \text{If performing } a \text{ on } s \text{ produces a correct dependency} \\ 0 & \text{Otherwise} \end{cases} . \tag{1}$$

In a DQN, the neural network acts as a value function approximator. The neural network takes as input the current state and outputs a value for each action. During evaluation, the action is chosen as $a = \operatorname{argmax}_a Q(s, a)$. During training, the algorithm chooses a random action with probability ϵ to facilitate exploration. We decay ϵ over the training period.

For reinforcement learning, we again use a similar network to the one described in Section 2.2. We use one hidden layer of size 200 with a rectified linear activation, a learning rate of 5×10^{-4}, and another layer mapping from the hidden layer to a 1×3 vector representing the value for each action. We use a discount rate of $\gamma = 0.9$, an update parameter of $\tau = 0.1$, and update every $C = 200$ steps. The replay buffer parameter is initialized to $k = 8$, four new sentences are parsed during every cycle of the inner loop, and 72 sentences are sampled every time for training. We call this learning algorithm `RL-Base`. We constructed several `RL-Base` variations that are summarized in Table 2 along with their test UAS. Fig. 2 shows the dev UAS during training for the variations. These results illustrate the importance of the reward function and reward propagation.

In order to develop a better understanding of development and test set performance, we extracted several sample parses and compared them to the ground truth parses. We summarize a couple key errors our parser makes. First, Fig. 3 depicts an example where the parser misses one critical `SHIFT` operation. This one mistake can later cause a domino effect when parsing. Second, In Fig. 4, we notice how local the parses are when compared to the ground truth parses. Our `RL-Base` parser is

Table 2: Test UAS for various RL models

Name	Description	Test UAS
`RL-Base`	Base model	71.07
`RL-ExtraHidden`	Adds an extra second hidden layer before output	**71.50**
`RL-DropoutExplore`	Dropout during exploration for non-deterministic parsing	70.75
`RL-PercentUAS`	Use UAS % instead of number of correct dependencies	56.46
`RL-NoItermediate`	No intermediate rewards applied	30.66
`RL-HighDiscount`	Experiment with $\gamma = 0.999$	66.37

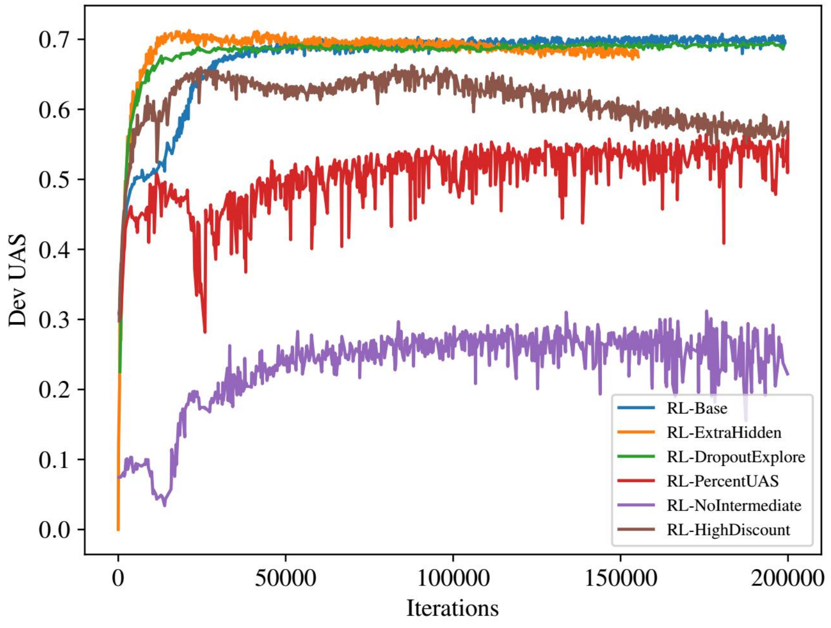


Figure 2: Dev set performance of various models during training.

capable of detecting clusters of dependencies in longer sentences, but it fails to discover dependencies that span many words. Since this is a long sentence, future rewards are most likely impacted by compounding discounts. This means that local and more immediate rewards take higher precedence.

6 Conclusion

We believed that augmenting neural dependency parsing with a reinforcement learning loss would lead to a higher UAS. While revamping the imitation learning technique using DAGGER improved the UAS in the low-data regime, we were not able to improve on the results of Chen and Manning

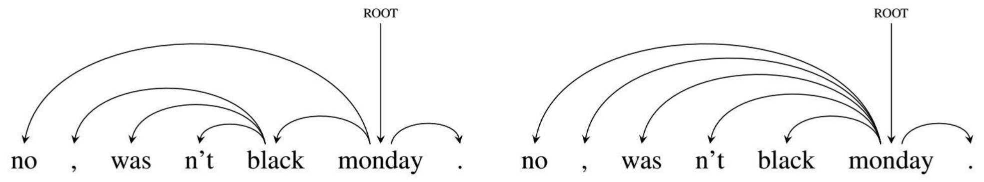
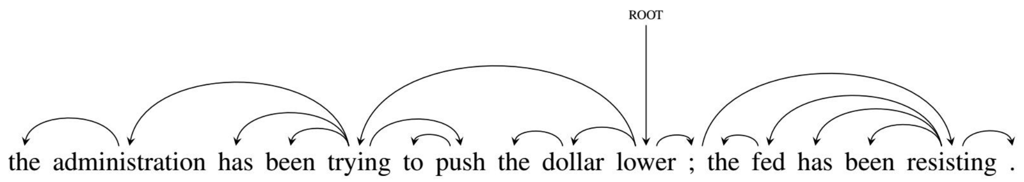


Figure 3: (Left) Parse from `RL-Base`; (right) ground truth dependencies.


ROOT
the administration has been trying to push the dollar lower ; the fed has been resisting .

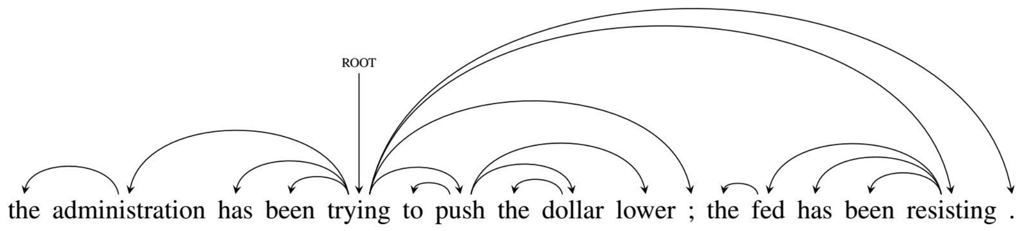

ROOT
the administration has been trying to push the dollar lower ; the fed has been resisting .

Figure 4: (Top) Parse from `RL-Base`; (bottom) ground truth dependencies.

using reinforcement learning on the full dataset. We believe that the lackluster performance of reinforcement learning could be because the DQN algorithm is ill-suited for generalization. The DQN algorithm focuses on local intermediate rewards and fails successfully backpropagate future rewards, especially for reward-less `SHIFT` operations, whereas a pure imitation learning approach with regularization and knowledge of exactly when to perform what action is able to generalize. We hope that our results are useful for future applications of reinforcement learning to statistical natural language processing.

Acknowledgments

The authors thank Richard Socher for his invaluable advice and guidance. This work was supported in part by the National Science Foundation under Grant No. 2017245257.

References

[1] Joakim Nivre. Incrementality in deterministic dependency parsing. In *Proc. of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57. Association for Computational Linguistics, 2004.

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

[3] André FT Martins, Noah A Smith, and Eric P Xing. Concise integer linear programming formulations for dependency parsing. In *Proc. 4th Intl. Joint Conf. on Natural Language Processing of the AFNLP*, pages 342–350. Association for Computational Linguistics, 2009.

[4] Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proc. 2014 Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, 2014.

[5] Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. In *Proc. 54th Ann. Meeting of the Association for Computational Linguistics*, pages 2442–2452, Aug 2016.

[6] Eliyahu Kiperwasser and Yoav Goldberg. Simple and accurate dependency parsing using bidirectional lstm feature representations. *arXiv preprint arXiv:1603.04351*, 2016.

[7] Timothy Dozat and Christopher D Manning. Deep biaffine attention for neural dependency parsing. *arXiv preprint arXiv:1611.01734*, 2016.

[8] Adhiguna Kuncoro, Miguel Ballesteros, Lingpeng Kong, Chris Dyer, Graham Neubig, and Noah A Smith. What do recurrent neural network grammars learn about syntax? pages 1249–1258, 2016.

[9] Minh Lê and Antske Fokkens. Tackling error propagation through reinforcement learning: A case of greedy dependency parsing. *arXiv preprint arXiv:1702.06794*, 2017.

[10] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proc. 14th Intl. Conf. on Artificial Intelligence and Statistics*, pages 627–635, 2011.

[11] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[13] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.