Approximate Dynamic Oracle for Dependency Parsing with Reinforcement Learning

Xiang Yu, Ngoc Thang Vu, and **Jonas Kuhn**
Institut für Maschinelle Sprachverarbeitung
Universität Stuttgart
{xiangyu,thangvu,jonas}@ims.uni-stuttgart.de

Abstract

We present a general approach with reinforcement learning (RL) to approximate dynamic oracles for transition systems where exact dynamic oracles are difficult to derive. We treat oracle parsing as a reinforcement learning problem, design the reward function inspired by the classical dynamic oracle, and use Deep Q-Learning (DQN) techniques to train the oracle with gold trees as features. The combination of a priori knowledge and data-driven methods enables an efficient dynamic oracle, which improves the parser performance over static oracles in several transition systems.

1 Introduction

Greedy transition-based dependency parsers trained with static oracles are very efficient but suffer from the error propagation problem. Goldberg and Nivre (2012, 2013) laid the foundation of dynamic oracles to train the parser with imitation learning methods to alleviate the problem. However, efficient dynamic oracles have mostly been designed for *arc-decomposable* transition systems which are usually projective. Gómez-Rodríguez et al. (2014) designed a non-projective dynamic oracle but runs in $O(n^8)$. Gómez-Rodríguez and Fernández-González (2015) proposed an efficient dynamic oracle for the non-projective Covington system (Covington, 2001; Nivre, 2008), but the system itself has quadratic worst-case complexity.

Instead of designing the oracles, Straka et al. (2015) applied the imitation learning approach (Daumé et al., 2009) by rolling out with the parser to estimate the cost of each action. Le and Fokkens (2017) took the reinforcement learning approach (Maes et al., 2009) by directly optimizing the parser towards the reward (i.e., the correct arcs) instead of the the correct action, thus no oracle is required. Both approaches circumvent the difficulty in designing the oracle cost function by using the parser to (1) explore the cost of each action, and (2) explore erroneous states to alleviate error propagation.

However, letting the parser explore for both purposes is inefficient and difficult to converge. For this reason, we propose to separate the two types of exploration: (1) the oracle explores the action space to learn the action cost with reinforcement learning, and (2) the parser explores the state space to learn from the oracle with imitation learning.

The objective of the oracle is to prevent further structure errors given a potentially erroneous state. We design the reward function to approximately reflect the number of unreachable gold arcs caused by the action, and let the model learn the actual cost from data. We use DQN (Mnih et al., 2013) with several extensions to train an Approximate Dynamic Oracle (ADO), which uses the gold tree as features and estimates the cost of each action in terms of potential attachment errors. We then use the oracle to train a parser with imitation learning methods following Goldberg and Nivre (2013).

A major difference between our ADO and the search-based or RL-based parser is that our oracle uses the gold tree as features in contrast to the lexical features of the parser, which results in a much simpler model solving a much simpler task. Furthermore, we only need to train one oracle for all treebanks, which is much more efficient.

We experiment with several transition systems, and show that training the parser with ADO performs better than training with static oracles in most cases, and on a par with the exact dynamic oracle if available. We also conduct an analysis of the oracle's robustness against error propagation for further investigation and improvement.

Our work provides an initial attempt to combine the advantages of reinforcement learning and imitation learning for structured prediction in the case of dependency parsing.

Proceedings of the Second Workshop on Universal Dependencies (UDW 2018), pages 183–191
Brussels, Belgium, November 1, 2018.

2 Approximate Dynamic Oracle

We treat oracle parsing as a deterministic Markov Decision Process (Maes et al., 2009), where a state corresponds to a parsing configuration with the gold tree known. The tokens are represented only by their positions in the stack or buffer, i.e., without lexical information. Unlike normal parsing, the initial state for the oracle can be any possible state in the entire state space, and the objective of the oracle is to minimize further structure errors, which we incorporate into the reward function.

2.1 Transition Systems and Reward Function

We define a unified reward function for the four transition systems that we experiment with: Arc-Standard (Yamada and Matsumoto, 2003; Nivre, 2004), Attardi's system with gap-degree of 1 (Attardi, 2006; Kuhlmann and Nivre, 2006), Arc-Standard with the *swap* transition (Nivre, 2009), and Arc-Hybrid (Kuhlmann et al., 2011). We denote them as STANDARD, ATTARDI, SWAP, and HYBRID, respectively. We formalize the actions in these systems in Appendix A.

The reward function approximates the *arc reachability* property as in Goldberg and Nivre (2013). Concretely, when an arc of head-dependent pair $\langle h,d\rangle$ is introduced, there are two cases of unreachable arcs: (1) if a pending token h' ($h' \neq h$) is the gold head of d, then $\langle h',d\rangle$ is unreachable; (2) if a pending token d' whose gold head is d, then $\langle d,d'\rangle$ is unreachable. If an attachment action does not immediately introduce unreachable arcs, we consider it *correct*.

The main reward for an attachment action is the negative count of immediate unreachable arcs it introduces, which sums up to the total attachment errors in the global view. We also incorporate some heuristics in the reward function, so that the *swap* action and non-projective (Attardi) attachments are slightly discouraged. Finally, we give a positive reward to a correct attachment to prevent the oracle from unnecessarily postponing attachment decisions. The exact reward values are modestly tuned in the preliminary experiments, and the reward function is defined as follows:

$$r = \begin{cases} -0.5, & \text{if action is } \textit{swap} \\ 0, & \text{if action is } \textit{shift} \\ -n, & \text{if } n \text{ unreachable arcs are introduced} \\ 0.5, & \text{if attachment is correct but non-projective} \\ 1, & \text{if attachment is correct and projective} \end{cases}$$

Although we only define the reward function for the four transition systems here, it can be easily extended for other systems by following the general principle: (1) reflect the number of unreachable arcs; (2) identify the unreachable arcs as early as possible; (3) reward correct attachment; (4) add system-specific heuristics.

Also note that the present reward function is not necessarily optimal. E.g., in the HYBRID system, a *shift* could also cause an unreachable arc, which is considered in the exact dynamic oracle by Goldberg and Nivre (2013), while the ADO can only observe the loss in later steps. We intentionally do not incorporate this knowledge into the reward function in order to demonstrate that the ADO is able to learn from delayed feedback information, which is necessary in most systems other than HYBRID. We elaborate on the comparison to the exact dynamic oracle in Section 4.

2.2 Feature Extraction and DQN Model

In contrast to the rich lexicalized features for the parser, we use a very simple feature set for the oracle. We use binary features to indicate the position of the gold head of the first 10 tokens in the stack and in the buffer. We also encode whether the gold head is already lost and whether the token has collected all its *pending* gold dependents. Additionally, we encode the 5 previous actions leading to this state, as well as all valid actions in this state.

We use the Deep Q-Network (DQN) to model the oracle, where the input is the aforementioned binary features from a state, and the output is the estimated values for each action in this state. The training objective of the basic DQN is to minimize the expected Temporal Difference (TD) loss:

$$L_{TD} = \mathbb{E}_{s,a\sim\pi}[(r + \gamma \max_{a'} Q(a'|s') - Q(a|s))^2]$$

where π is the policy given the value function Q, which assigns a score for each action, s is the current state, a is the performed action, r is the reward, γ is the discount factor, s' is the next state, and a' is the optimal action in state s'.

We apply several techniques to improve the stability of the DQN, including the averaged DQN (Anschel et al., 2016) to reduce variance, the dueling network (Wang et al., 2016) to decouple the estimation of state value and action value, and prioritized experience replay (Schaul et al., 2015) to increase the efficiency of samples.

2.3 Sampling Oracle Training Instances

Our goal is learning to handle erroneous states, so we need to sample such instances during training. Concretely, for every state in the sampling process, apart from following the ε-greedy policy (i.e., select random action with ε probability), we fork the path with certain probability by taking a valid random action to simulate the mistake by the parser. We treat each forked path as a new episode starting from the state after the forking action. Also, to increase sample efficiency, we only take the first N states in each episode, as illustrated in Figure 1.

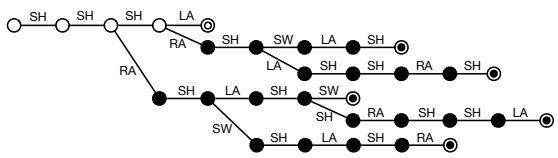


Figure 1: An illustration of the sampled episodes with $N = 5$, where light nodes represent the original path, dark nodes represent the forked paths, and double circles represent the end of the sampled path.

2.4 Cost-Sensitive Parser Training

Since the DQN estimates the cost for each action, we can thus apply cost-sensitive training with multi-margin loss (Edunov et al., 2017) for the parser instead of negative log-likelihood or hinge loss. Concretely, we enforce the margin of the parser scores between the correct action and every other action to be larger than the difference between the oracle scores of these two actions:

$$L = \sum_{a \in A} \max(0, P(a|s) - P(a^*|s) + Q(a^*|s) - Q(a|s))$$

where A is the set of valid actions, $P(\cdot|s)$ is the parser score, $Q(\cdot|s)$ is the oracle score, and a^* is the optimal action according to the oracle.

Cost-sensitive training is similar to the *non-deterministic* oracle (Goldberg and Nivre, 2013), since actions with similar oracle scores need to maintain a smaller margin, thus allowing spurious ambiguity. On the other hand, it also penalizes the actions with larger cost more heavily, thus focusing the training on the more important actions.

3 Experiments

3.1 Data and Settings

We conduct our experiments on the 55 big treebanks from the CoNLL 2017 shared task (Zeman et al., 2017), referred to by their treebank codes, e.g., `grc` for Ancient Greek. For easier replicability, we use the predicted segmentation, part-of-speech tags and morphological features by UDPipe (Straka et al., 2016), provided in the shared task, and evaluate on Labeled Attachment Score (LAS) with the official evaluation script. We also provide the parsing results by UDPipe as a baseline, which incorporates the search-based oracle for non-projective parsing (Straka et al., 2015).

We implement the parser architecture with bidirectional LSTM following Kiperwasser and Goldberg (2016) with the minimal feature set, namely three tokens in the stack and one token in the buffer. For each token, we compose character-based representations with convolutional neural networks following Yu and Vu (2017), and concatenate them with randomly initialized embeddings of the word form, universal POS tag, and morphological features. All hyperparameters of the parser and the oracle are listed in Appendix B.[1]

We compare training the parser with ADOs to static oracles in the four transition systems STANDARD, ATTARDI, SWAP, and HYBRID. Additionally, we implement the exact dynamic oracle (EDO) for HYBRID as the upper bound. For each system, we only use the portion of training data where all oracles can parse, e.g., for STANDARD and HYBRID, we only train on projective trees.

We did preliminary experiments on the ADOs in three settings: (1) $\texttt{O}_{\texttt{all}}$ is trained only on the non-projective trees from all training treebanks (ca. 133,000 sentences); (2) $\texttt{O}_{\texttt{ind}}$ is trained on the individual treebank as used for training the parser; and (3) $\texttt{O}_{\texttt{tune}}$ is based on $\texttt{O}_{\texttt{all}}$, but fine-tuned interactively during the parser training by letting the parser initiate the forked episodes. Results show that three versions have very close performance, we thus choose the simplest one $\texttt{O}_{\texttt{all}}$ to report and analyze, since in this setting only one oracle is needed for training on all treebanks.

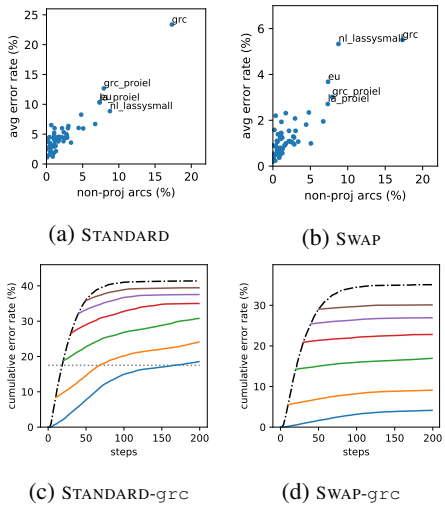
3.2 Oracle Recovery Test

Before using the oracle to train the parser, we first test the oracle's ability to control the mistakes. In this test, we use a parser trained with the static oracle to parse the development set, and starting from the parsing step 0, 10, 20, 30, 40, and 50, we let the ADO fork the path and parse until the end. We

[1] The parser and the oracle are available at `http://www.ims.uni-stuttgart.de/institut/mitarbeiter/xiangyu/index.en.html`.

use the error rate of the oracle averaged over the aforementioned starting steps as the measurement for the oracle's robustness against error propagation: the smaller the rate, the more robust the oracle. Note that we identify the errors only when the incorrect arcs are produced, but they could be already inevitable due to previous actions, which means some of the parser's mistakes are attributed to the oracle, resulting in a more conservative estimation of the oracle's recovery ability.

Figure 2a and 2b show the average error rate for each treebank and its relation to the percentage of non-projective arcs in the projective STANDARD and the non-projective SWAP systems. Generally, the error rate correlates with the percentage of the non-projective arcs. However, even in the most difficult case (i.e., `grc` with over 17% non-projective arcs), the oracle only introduces 5% errors in the non-projective system, which is much lower than the parser's error rate of over 40%. The higher error rates in the projective system is due to the fact that the number of errors is at least the number of non-projective arcs. Figure 2c and 2d show the oracles' error recovery performance in the most difficult case `grc`. The error curves of the oracles in the non-projective systems are very flat, while in the STANDARD system, the errors of the oracle starting from step 0 is only slightly higher than the number of non-projective arcs (the dotted line), which is the lower bound of errors. These results all confirm that the ADO is able to find actions to minimize further errors given any potentially erroneous state.



(a) STANDARD (b) SWAP

(c) STANDARD-`grc` (d) SWAP-`grc`

Figure 2: Results of the oracle recovery test, where (a) and (b) are average error rates across treebanks, (c) and (d) are cumulative error rates for `grc`.

3.3 Final Results

		UDPipe	STANDARD static	STANDARD ADO	ATTARDI static	ATTARDI ADO	SWAP static	SWAP ADO	HYBRID static	HYBRID ADO	HYBRID EDO
most non-proj	grc	56.04	51.49	51.94	59.02	60.21	58.59	60.87	50.40	52.46	52.74
	nl_las.	78.15	73.07	73.57	79.17	81.88	79.67	81.48	72.29	73.45	73.08
	grc_pro.	65.22	63.85	63.96	68.30	67.79	67.01	67.52	64.26	64.75	64.41
least non-proj	ja	72.21	73.03	73.07	73.12	72.99	73.16	73.26	72.91	73.39	73.34
	gl	77.31	77.19	77.47	77.34	77.63	77.28	77.39	77.27	77.50	77.49
	zh	57.40	57.99	58.56	58.69	58.86	57.83	59.19	57.99	57.83	58.57
most data	cs	82.87	84.32	84.04	84.99	84.72	84.90	84.14	84.34	84.29	84.24
	ru_syn.	86.76	88.09	87.32	88.78	88.09	88.64	87.83	88.05	88.20	88.22
	cs_cac	82.46	83.61	83.57	83.65	83.68	83.64	84.40	82.94	83.52	83.69
least data	cs_cltt	71.64	71.36	73.65	74.04	74.93	73.81	74.23	70.54	72.32	72.25
	hu	64.30	62.91	65.08	65.75	67.02	64.79	66.69	63.65	64.34	63.39
	en_par.	73.64	74.10	74.04	73.74	74.80	73.87	74.65	73.74	73.88	73.22
	AVG	73.04	73.59	73.92	74.66	74.99	74.50	75.01	73.47	73.68	73.74

Table 1: LAS on the selected test sets, where green cells mark ADO outperforming the static oracle and red cells otherwise. Average is calculated over all 55 test set.

In the final experiment, we compare the performance of the parser trained by the ADOs against the static oracle or the EDO if available. Table 1 shows the LAS of 12 representative treebanks, while the full results are shown in Appendix C. In the selection, we include treebanks with the highest percentage of non-projective arcs (`grc`, `nl_lassysmall`, `grc_proiel`), almost only projective trees (`ja`, `gl`, `zh`), the most training data (`cs`, `ru_syntagrus`, `cs_cac`), and the least training data (`cs_cltt`, `hu`, `en_partut`).

Out of the 55 treebanks, the ADO is beneficial in 41, 40, 41, and 35 treebanks for the four systems, and on average outperforms the static baseline by 0.33%, 0.33%, 0.51%, 0.21%, respectively. While considering the treebank characteristics, training with ADOs is beneficial in most cases irrespective of the projectiveness of the treebank. It works especially well for small treebanks, but not as well for very large treebanks. The reason could be that the error propagation problem is not as severe when the parsing accuracy is high, which correlates with the training data size.

In HYBRID, the benefit of the ADO and EDO is very close, they outperform the static baseline by 0.21% and 0.27%, which means that the ADO approximates the upper bound EDO quite well.

Note that we train the parsers only on projective trees in projective systems to ensure a fair comparison. However, the ADO is able to guide the parser even on non-projective trees, and the resulting parsers in STANDARD outperform the baseline by 1.24% on average (see Appendix C), almost bridging the performance gap between projective and non-projective systems.

4 Comparing to Exact Dynamic Oracle

The purpose of the ADO is to approximate the dynamic oracle in the transition systems where an exact dynamic oracle is unavailable or inefficient. However, it could demonstrate how well the approximation is when compared to the EDO, which serves as an upper bound. Therefore, we compare our ADO to the EDO (Goldberg and Nivre, 2013) in the HYBRID system.

First, we compare the reward function of the ADO (see Section 2.1) to the cost function of the EDO, which is: (1) for an attachment action that introduces an arc $\langle h, d \rangle$, the cost is the number of reachable dependents of d plus whether d is still reachable to its gold head h' ($h' \neq h$); and (2) for *shift*, the cost is the number of reachable dependents of d in the stack plus whether the gold head of d is in the stack except for the top item.

The general ideas of both oracles are very similar, namely to punish an action by the number of unreachable arcs it introduces. However, the definitions of reachability are slightly different.

Reachable arcs in the ADO are defined more loosely: as long as the head and dependent of an arc are pending in the stack or buffer, it is considered reachable, thus the reward (cost) of *shift* is always zero. However, in the HYBRID system, an arc of two tokens in the stack could be unreachable (e.g. $\langle s_0, s_1 \rangle$), thus the cost of *shift* in the EDO could be non-zero.

Note that both oracles punish each incorrect attachment exactly once, and the different definitions of reachability only affect the time when an incorrect attachment is punished, namely when the correct attachment is deemed unreachable. Generally, the ADO's reward function delays the punishment for many actions, and dealing with delayed reward signal is exactly the motivation of RL algorithms (Sutton and Barto, 1998).

The DQN model in the ADO bridges the lack of prior knowledge in the definitions of reachability by estimating not only the immediate reward of an action, but also the discounted future rewards. Take the HYBRID system for example. Although the immediate reward of a *shift* is zero, the ADO could learn a more accurate cost in its value estimation if the action eventually causes an unreachable arc. Moreover, in a system where the *exact* reachability is difficult to determine, the ADO estimates the *expected* reachability based on the training data.

We then empirically compare the behavior of the ADO with the EDO, in which we use a parser trained with the static oracle to parse the development set of a treebank, and for each state along the transition sequence produced by the parser we consult the ADO and the EDO. Since the EDO gives a set of optimal actions, we check whether the ADO's action is in the set.

On average, the ADO differs from the EDO (i.e., making suboptimal actions) only in 0.53% of all cases. Among the states where the ADO makes suboptimal actions, more than 90% has the pattern shown in Figure 3, where the gold head of s_1 is s_0 but it is already impossible to make the correct attachment for it, therefore the correct action is to make a *left-arc* to ensure that s_0 is attached correctly. However, the ADO does not realize that s_1 is already lost and estimates that a *left-arc* attachment would incur a negative reward, and is thus inclined to make a "harmless" shift, which would actually cause another lost token s_0 in the future. This type of mistakes happens about 30% of the time when this pattern occurs, and further investigation is needed to eliminate them.

Figure 3: A typical pattern where the ADO makes a mistake.

5 Conclusion

In this paper, we propose to train efficient approximate dynamic oracles with reinforcement learning methods. We tackle the problem of non-decomposable structure loss by letting the oracle learn the action loss from incremental immediate rewards, and act as a proxy for the structure loss to train the parser. We demonstrate that training with a single treebank-universal ADO generally improves the parsing performance over training with static oracle in several transition systems, we also show the ADO's comparable performance to an exact dynamic oracle.

Furthermore, the general idea in this work could be extended to other structured prediction tasks such as graph parsing, by training a better-informed oracle to transform structure costs into action costs, which gives the learning agent more accurate objective while staying in the realm of imitation learning to ensure training efficiency.

References

Oron Anschel, Nir Baram, and Nahum Shimkin. 2016. Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. *arXiv preprint arXiv:1611.01929*.

Giuseppe Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, pages 166–170. Association for Computational Linguistics.

Michael A Covington. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th annual ACM southeast conference*, pages 95–102. Citeseer.

Hal Daumé, John Langford, and Daniel Marcu. 2009. Search-based structured prediction. *Machine learning*, 75(3):297–325.

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

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. *Proceedings of COLING 2012*, pages 959–976.

Yoav Goldberg and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Transactions of the Association of Computational Linguistics*, 1:403–414.

Carlos Gómez-Rodríguez and Daniel Fernández-González. 2015. An efficient dynamic oracle for unrestricted non-projective parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, volume 2, pages 256–261.

Carlos Gómez-Rodríguez, Francesco Sartorio, and Giorgio Satta. 2014. A polynomial-time dynamic oracle for non-projective dependency parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 917–927.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional lstm feature representations. *Transactions of the Association for Computational Linguistics*, 4:313–327.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 673–682. Association for Computational Linguistics.

Marco Kuhlmann and Joakim Nivre. 2006. Mildly non-projective dependency structures. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 507–514.

Minh Le and Antske Fokkens. 2017. Tackling error propagation through reinforcement learning: A case of greedy dependency parsing. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, volume 1, pages 677–687.

Francis Maes, Ludovic Denoyer, and Patrick Gallinari. 2009. Structured prediction with reinforcement learning. *Machine learning*, 77(2-3):271.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*.

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

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*, pages 351–359. Association for Computational Linguistics.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Milan Straka, Jan Hajic, and Jana Straková. 2016. Udpipe: Trainable pipeline for processing conll-u files performing tokenization, morphological analysis, pos tagging and parsing. In *LREC*.

Milan Straka, Jan Hajic, Jana Straková, and Jan Hajic jr. 2015. Parsing universal dependency treebanks using neural networks and search-based oracle. In *International Workshop on Treebanks and Linguistic Theories (TLT14)*, pages 208–220.

Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. 2016. Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1995–2003.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, volume 3, pages 195–206. Nancy, France.

Xiang Yu and Ngoc Thang Vu. 2017. Character composition model with convolutional neural networks for dependency parsing on morphologically rich languages. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 672–678.

Daniel Zeman, Martin Popel, Milan Straka, Jan Hajic, Joakim Nivre, Filip Ginter, Juhani Luotolahti, Sampo Pyysalo, Slav Petrov, Martin Potthast, Francis Tyers, Elena Badmaeva, Memduh Gokirmak, Anna Nedoluzhko, Silvie Cinkova, Jan Hajic jr., Jaroslava Hlavacova, Václava Kettnerová, Zdenka Uresova, Jenna Kanerva, Stina Ojala, Anna Missilä, Christopher D. Manning, Sebastian Schuster, Siva Reddy, Dima Taji, Nizar Habash, Herman Leung, Marie-Catherine de Marneffe, Manuela Sanguinetti, Maria Simi, Hiroshi Kanayama, Valeria dePaiva, Kira Droganova, Héctor Martínez Alonso, Çağr Çöltekin, Umut Sulubacak, Hans Uszkoreit, Vivien Macketanz, Aljoscha Burchardt, Kim Harris, Katrin Marheinecke, Georg Rehm, Tolga Kayadelen, Mohammed Attia, Ali Elkahky, Zhuoran Yu, Emily Pitler, Saran Lertpradit, Michael Mandl, Jesse Kirchner, Hector Fernandez Alcalde, Jana Strnadová, Esha Banerjee, Ruli Manurung, Antonio Stella, Atsuko Shimada, Sookyoung Kwak, Gustavo Mendonca, Tatiana Lando, Rattima Nitisaroj, and Josie Li. 2017. Conll 2017 shared task: Multilingual parsing from raw text to universal dependencies. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–19, Vancouver, Canada. Association for Computational Linguistics.

A Transition Systems

Table 2 provides a unified view of the the actions in the four transition systems: *shift* and *right* are shared by all four systems; *left* is shared by all but the HYBRID system, which uses *left-hybrid* instead; *left-2* and *right-2* are defined only in the ATTARDI system; and *swap* is defined only in the SWAP system.

For all systems, the initial states are identical: the stack contains only the root, the buffer contains all other tokens, and the set of arcs is empty. The terminal states are also identical: the stack contains only the root, the buffer is empty, and the set of arcs is the created dependency tree.

Action	Before		After
shift	$(\sigma, j \mid \beta, A)$	$\rightarrow$	$(\sigma \mid j, \beta, A)$
left	$(\sigma \mid i \mid j, \beta, A)$	$\rightarrow$	$(\sigma \mid j, \beta, A \cup \{\langle j,i \rangle\})$
right	$(\sigma \mid i \mid j, \beta, A)$	$\rightarrow$	$(\sigma \mid i, \beta, A \cup \{\langle i,j \rangle\})$
left-2	$(\sigma \mid i \mid j \mid k, \beta, A)$	$\rightarrow$	$(\sigma \mid j \mid k, \beta, A \cup \{\langle k,i \rangle\})$
right-2	$(\sigma \mid i \mid j \mid k, \beta, A)$	$\rightarrow$	$(\sigma \mid i \mid j, \beta, A \cup \{\langle i,k \rangle\})$
left-hybrid	$(\sigma \mid i, j \mid \beta, A)$	$\rightarrow$	$(\sigma, j \mid \beta, A \cup \{\langle j,i \rangle\})$
swap	$(\sigma \mid i \mid j, \beta, A)$	$\rightarrow$	$(\sigma \mid j, i \mid \beta, A)$

Table 2: The actions defined in the four transition systems, where σ denotes the stack, β denotes the buffer, and A denotes the set of created arcs.

B Architecture and Hyperparameters

The parser takes characters, word form, universal POS tag and morphological features of each word as input. The character composition model follows Yu and Vu (2017), which takes 4 convolutional filters with width of 3, 5, 7, and 9, each filter has dimension of 32, adding to a 128-dimensional word representation. The randomly initialized word embeddings are also 128-dimensional, the POS tag and morphological features are both 32-dimensional. The concatenated word representations are then fed into a bidirectional LSTM with 128 hidden units to capture the contextual information in the sentence. The contextualized word representations of the top 3 tokens in the stack and the first token in the buffer are concatenated and fed into two layers of 256 hidden units with the ReLU activation, and the output are the scores for each action. The argmax of the scores are then further concatenated with the last hidden layer, and outputs the scores for the labels if the predicted action introduces an arc. In this way, the prediction of action and label are decoupled, and they are learned separately.

The oracle (DQN) takes the binary features described in Section 2.2 as input, which is fed into a layer of 128 hidden units. It then forks into two channels to calculate the value for the state and the actions separately, then they are aggregated as the estimated state-action value, as in Wang et al. (2016). In the DQN training, we use discount factor $\gamma = 0.9$, for the proportional prioritized experience replay, we select $\alpha = 0.9$, $\beta = 0.5$.

Both the parser and the oracle are trained with maximum 50000 mini-batches and early-stop on the development set. In every step, the parser trains on mini-batches of 10 sentences, and the oracle generates samples from 5 sentences into the replay memory, and trains on mini-batches of 1000 samples. While generating the samples for the oracle, we fork each state by a random valid action with a probability of 0.05, and we take at most 5 forked episodes for each sentence, with the maximum episode length $N = 20$.

C Full Results

The results for all 55 treebanks are shown in Table 3.

		Standard			Swap		Attardi		Hybrid		
	UDPipe	static	ADO	ADO*	static	ADO	static	ADO	static	ADO	EDO
ar	65.30	66.75	66.75	66.75	67.34	67.34	67.15	67.15	67.14	67.14	67.14
bg	83.64	84.14	84.54	84.48	84.19	84.64	83.95	84.46	84.31	84.31	84.31
ca	85.39	86.08	86.08	86.08	86.62	86.62	86.54	86.54	86.09	86.09	86.15
cs	82.87	84.32	84.32	84.47	84.99	84.99	84.90	84.90	84.34	84.34	84.34
cs_cac	82.46	83.61	83.61	84.48	83.65	83.68	83.64	84.40	82.94	83.52	83.69
cs_cltt	71.64	71.36	73.65	75.15	74.04	74.93	73.81	74.23	70.54	72.32	72.25
cu	62.76	63.81	64.16	65.94	66.47	66.92	66.37	67.09	63.85	64.39	64.69
da	73.38	73.55	74.73	75.50	74.51	75.00	74.51	75.30	73.45	73.45	74.36
de	69.11	71.93	71.94	72.87	73.26	73.26	73.03	73.03	71.72	71.80	71.96
el	79.26	79.47	79.99	80.27	80.15	80.84	79.96	80.64	79.04	79.04	79.20
en	75.84	75.99	76.37	76.76	76.37	76.65	76.32	76.63	75.82	76.31	76.19
en_lines	72.94	72.52	72.76	73.56	74.12	74.27	74.08	74.57	73.20	73.20	73.20
en_partut	73.64	74.10	74.10	74.56	73.74	74.80	73.87	74.65	73.74	73.88	73.74
es	81.47	82.79	82.79	82.79	82.76	82.76	82.49	82.49	82.85	82.85	82.85
es_ancora	83.78	84.83	84.85	85.33	85.56	85.56	85.66	85.66	85.34	85.34	85.34
et	58.79	58.77	59.34	59.47	59.10	60.84	58.93	61.12	59.11	59.11	59.11
eu	69.15	68.91	69.54	71.27	71.44	72.72	70.63	72.64	68.55	68.95	68.72
fa	79.24	79.73	79.76	80.47	80.47	80.47	79.67	79.87	80.20	80.20	80.20
fi	73.75	74.34	74.57	74.90	74.53	74.70	74.48	75.20	73.96	73.96	74.33
fi_ftb	74.03	75.58	75.86	76.10	75.93	76.03	75.18	75.18	75.41	75.41	75.42
fr	80.75	81.42	81.44	81.42	81.88	81.88	81.21	81.98	81.25	81.25	81.25
fr_sequoia	79.98	80.90	80.90	81.52	81.64	81.76	81.37	81.37	80.56	80.58	80.67
gl	77.31	77.19	77.47	77.96	77.34	77.63	77.28	77.39	77.27	77.50	77.49
got	59.81	59.81	60.61	62.05	61.26	61.97	60.94	62.53	59.09	60.78	60.14
grc	56.04	51.49	51.94	58.13	59.02	60.21	58.59	60.87	50.40	52.46	52.74
grc_proiel	65.22	63.85	63.96	67.40	68.30	68.30	67.01	67.52	64.26	64.75	64.41
he	57.23	57.95	58.30	59.01	58.02	58.72	58.18	58.41	57.82	58.02	58.06
hi	86.77	87.48	87.48	88.06	88.22	88.22	88.08	88.08	87.53	87.53	87.56
hr	77.18	77.19	77.98	78.23	78.45	78.63	77.64	77.96	77.26	77.45	77.70
hu	64.30	62.91	65.08	65.59	65.75	67.02	64.79	66.69	63.65	64.34	63.65
id	74.61	74.85	74.85	74.85	74.42	74.42	74.79	74.79	74.17	74.50	74.48
it	85.28	85.76	86.06	85.91	86.33	86.33	86.18	86.18	86.24	86.24	86.24
ja	72.21	73.03	73.07	73.06	73.12	73.12	73.16	73.26	72.91	73.39	73.34
ko	59.09	72.48	74.07	75.09	73.98	74.61	73.97	74.70	72.25	73.40	72.85
la_ittb	76.98	77.80	77.80	80.56	81.27	81.35	82.33	82.33	77.05	77.56	77.61
la_proiel	57.54	56.15	56.85	60.00	58.67	59.38	59.18	60.80	55.92	57.01	57.58
lv	59.95	60.04	61.39	61.53	60.62	60.66	60.32	60.96	59.76	60.34	60.05
nl	68.90	69.69	70.47	71.53	71.77	72.03	70.50	71.87	69.42	69.83	69.90
nl_lassysmall	78.15	73.07	73.57	78.82	79.17	81.88	79.67	81.48	72.29	73.45	73.08
no_bokmaal	83.27	84.07	84.50	84.56	84.47	84.68	84.15	84.82	83.92	83.92	84.04
no_nynorsk	81.56	82.41	82.45	83.46	82.64	82.99	82.64	82.91	81.74	82.47	82.32
pl	78.78	80.25	80.41	80.61	80.28	80.34	79.84	80.14	79.26	80.20	79.95
pt	82.11	82.33	82.33	82.83	82.49	82.73	82.92	83.07	81.77	82.03	81.93
pt_br	85.36	86.11	86.40	86.30	86.17	86.17	85.98	86.28	86.01	86.21	86.17
ro	79.88	80.06	80.21	80.45	79.96	80.37	80.41	80.41	79.59	79.66	79.73
ru	74.03	74.66	74.66	75.26	75.07	75.78	74.62	75.60	74.68	74.68	75.15
ru_syntagrus	86.76	88.09	88.09	88.09	88.78	88.78	88.64	88.64	88.05	88.20	88.22
sk	72.75	73.73	73.99	75.84	74.27	74.75	74.28	75.58	74.09	74.26	74.51
sl	81.15	81.26	81.97	83.13	82.94	83.32	82.61	83.65	81.85	81.86	81.94
sv	76.73	77.24	77.70	78.39	77.86	78.25	78.03	78.54	77.70	78.24	77.70
sv_lines	74.29	74.28	74.64	75.49	74.32	75.22	74.01	75.36	73.40	73.80	74.00
tr	53.19	53.97	54.82	55.38	54.20	55.38	54.53	55.18	54.32	54.56	54.32
ur	76.69	77.16	77.16	77.25	77.40	77.83	77.89	77.92	77.16	77.16	77.16
vi	37.47	38.32	39.09	39.10	38.38	38.85	38.50	39.68	38.36	38.67	38.80
zh	57.40	57.99	58.56	58.65	58.69	58.86	57.83	59.19	57.99	57.99	58.57
AVG	73.04	73.59	73.92	74.83	74.66	74.99	74.50	75.01	73.47	73.68	73.74

Table 3: LAS on the 55 test sets, where green cells mark ADO outperforming the static oracle and red cells for the opposite. The column ADO* indicate the parsers trained on both projective and non-projective trees. Average is calculated over all 55 test set.