Universal Dependency Parsing with a General Transition-Based DAG Parser

Daniel Hershcovich[1,2] **Omri Abend**[2] **Ari Rappoport**[2]
[1]The Edmond and Lily Safra Center for Brain Sciences
[2]School of Computer Science and Engineering
Hebrew University of Jerusalem
{danielh,oabend,arir}@cs.huji.ac.il

Abstract

This paper presents our experiments with applying TUPA to the CoNLL 2018 UD shared task. TUPA is a general neural transition-based DAG parser, which we use to present the first experiments on recovering enhanced dependencies as part of the general parsing task. TUPA was designed for parsing UCCA, a cross-linguistic semantic annotation scheme, exhibiting reentrancy, discontinuity and non-terminal nodes. By converting UD trees and graphs to a UCCA-like DAG format, we train TUPA almost without modification on the UD parsing task. The generic nature of our approach lends itself naturally to multitask learning. Our code is available at https://github.com/CoNLL-UD-2018/HUJI.

1 Introduction

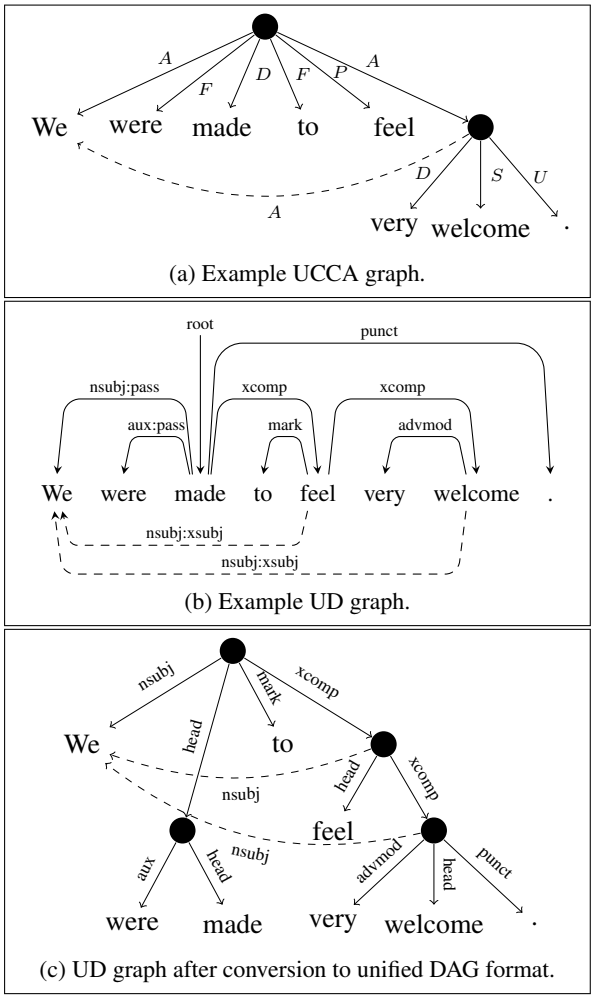
In this paper, we present the HUJI submission to the CoNLL 2018 shared task on Universal Dependency parsing (Zeman et al., 2018). We focus only on parsing, using the baseline system, UDPipe 1.2 (Straka et al., 2016; Straka and Straková, 2017) for tokenization, sentence splitting, part-of-speech tagging and morphological tagging.

Our system is based on TUPA (Hershcovich et al., 2017, 2018, see §3), a transition-based UCCA parser. UCCA (Universal Conceptual Cognitive Annotation; Abend and Rappoport, 2013) is a cross-linguistic semantic annotation scheme, representing events, participants, attributes and relations in a directed acyclic graph (DAG) structure. UCCA allows reentrancy to support argument sharing, discontinuity (corresponding to non-projectivity in dependency formalisms) and non-terminal nodes (as opposed to dependencies, which are bi-lexical). To parse Universal Dependencies (Nivre et al., 2016) using TUPA, we employ a bidirectional conversion protocol to represent UD trees and graphs in a UCCA-like unified DAG format (§2).

Enhanced dependencies. Our method treats *enhanced dependencies*[1] as part of the dependency graph, providing the first approach, to our knowledge, for supervised learning of enhanced UD parsing. Due to the scarcity of enhanced dependencies in UD treebanks, previous approaches (Schuster and Manning, 2016; Reddy et al., 2017) have attempted to recover them using language-specific rules. Our approach attempts to learn them from data: while only a few UD treebanks contain any enhanced dependencies, similar structures are an integral part of UCCA and its annotated corpora (realized as reentrancy by remote edges; see §2), and TUPA supports them as a standard feature.

As their annotation in UD is not yet widespread and standardized, enhanced dependencies are *not included* in the evaluation metrics for UD parsing, and so TUPA's ability to parse them is not reflected in the official shared task scores. However, we believe these enhancements, representing case information, elided predicates, and shared arguments due to conjunction, control, raising and relative clauses, provide richer information to downstream semantic applications, making UD better suited for text understanding. We propose an evaluation metric specific to enhanced dependencies, *enhanced LAS* (§5.1), and use it to evaluate our method.

[1]http://universaldependencies.org/u/overview/enhanced-syntax.html



(a) Example UCCA graph.

(b) Example UD graph.

(c) UD graph after conversion to unified DAG format.

Figure 1: (a) Example UCCA annotation for the sentence "We were made to feel very welcome.", containing a control verb, *made*. The dashed *A* edge is a *remote edge*. (b) Bilexical graph annotating the same sentence in UD (`reviews-077034-0002` from `UD_English-EWT`). Enhanced dependencies appear below the sentence. (c) The same UD graph, after conversion to the unified DAG format. Intermediate non-terminals and *head* edges are introduced, to get a UCCA-like structure.

2 Unified DAG Format

To apply TUPA to UD parsing, we convert UD trees and graphs into a unified DAG format (Hershcovich et al., 2018). The format consists of a rooted DAG, where the tokens are the terminal nodes.[2] Edges are labeled (but not nodes), and are divided into *primary* and *remote* edges, where the primary edges form a tree (all nodes have at most one primary parent, and the root has none). Remote edges (denoted as dashed edges in Figure 1) enable reentrancy, and thus form a DAG together with primary edges. Figure 1 shows an example UCCA graph, and a UD graph (containing two enhanced dependencies) before and after conversion. Both annotate the same sentence from the English Web Treebank (Silveira et al., 2014)[3].

Conversion protocol. To convert UD into the unified DAG format, we add a pre-terminal for each token, and attach the pre-terminals according to the original dependency edges: traversing the tree from the root down, for each head token we create a non-terminal parent with the edge label *head*, and add the node's dependents as children of the created non-terminal node (see Figure 1c). This creates a constituency-like structure, which is supported by TUPA's transition set (see §3.1).

Although the enhanced dependency graph is not necessarily a supergraph of the basic dependency tree, the graph we convert to the unified DAG format is their union: any enhanced dependnecies that are distinct from the basic dependency of a node (by having a different head or universal dependency relation) are converted to *remote edges* in the unified DAG format.

To convert graphs in the unified DAG format back into dependency graphs, we collapse all *head* edges, determining for each terminal what is the highest non-terminal headed by it, and then attaching the terminals to each other according to the edges among their headed non-terminals.

Input format. Enhanced dependencies are encoded in the 9th column of the CoNLL-U format, by an additional head index, followed by a colon and dependency relation. Multiple enhanced dependencies for the same node are separated by pipes. Figure 2 demonstrates this format. Note that if the basic dependency is repeated in the enhanced graph (`3:nsubj:pass` in the example), we do not treat it as an enhanced dependency, so that the converted graph will only contain each edge once. In addition to the UD relations defined in the basic representations, enhanced dependencies may contain the relation `ref`, used for relative clauses. In addition, they may contain more specific relation subtypes, and optionally also case information.

Language-specific extensions and case information. Dependencies may contain language-

[2] Our conversion code supports full conversion between UCCA and UD, among other representation schemes, and is publicly available at http://github.com/danielhers/semstr/tree/master/semstr/conversion.

[3] https://catalog.ldc.upenn.edu/LDC2012T13

```
1 We we PRON PRP Case=Nom|Number=Plur|Person=1|PronType=Prs 3 nsubj:pass 3:nsubj:pass|5:nsubj:xsubj|7:nsubj:xsubj _
```

Figure 2: Example line from CoNLL-U file with two enhanced dependencies: `5:nsubj:xsubj` and `7:nsubj:xsubj`.

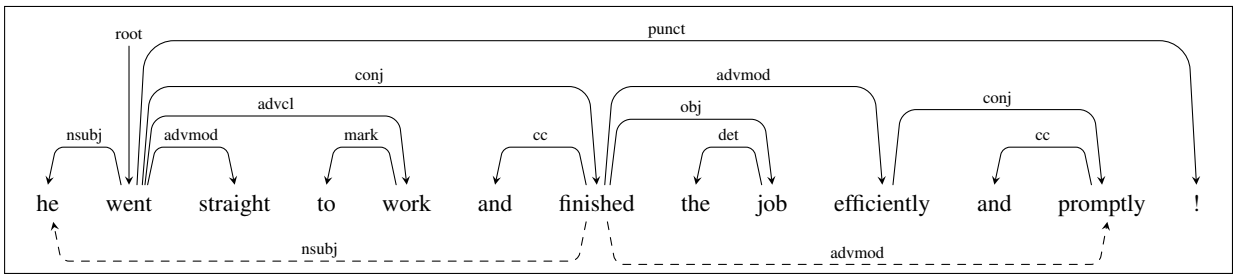


Figure 3: UD graph from `reviews-341397-0003` (`UD_English-EWT`), containing conjoined predicates and arguments.

specific relation subtypes, encoded as a suffix separated from the universal relation by a colon. These extensions are ignored by the parsing evaluation metrics, so for example, the subtyped relation `nsubj:pass` (Figure 1b) is considered the same as the universal relation `nsubj` for evaluation purposes. In the enhanced dependencies, these suffixes may also contain case information, which may be represented by the lemma of an adposition. For example, the "peace" → "earth" dependency in Figure 4 is augmented as `nmod:on` in the enhanced graph (not shown in the figure because it shares the universal relation with the basic dependency).

In the conversion process, we strip any language-specific extensions from both basic and enhanced dependencies, leaving only the universal relations. Consequently, case information that might be encoded in the enhanced dependencies is lost, and we do not handle it in our current system.

Ellipsis and null nodes. In addition to enhanced dependencies, the enhanced UD representation adds null nodes to represented elided predicates. These, too, are ignored in the standard evaluation. An example is shown in Figure 4, where an elided "wish" is represented by the node E9.1. The elided predicate's dependents are attached to its argument "peace" in the basic representation, and the argument itself is attached as an `orphan`. In the enhanced representation, all arguments are attached to the null node as if the elided predicate was present.

While UCCA supports empty nodes without surface realization in the form of *implicit units*, previous work on UCCA parsing has removed these from the graphs. We do the same for UD parsing, dropping null nodes and their associated dependencies upon conversion to the unified DAG format. We leave parsing elided predicates for future work.

Propagation of conjuncts. Enhanced dependencies contain dependencies between conjoined predicates and their arguments, and between predicates and their conjoined arguments or modifiers. While these relations can often be inferred from the basic dependencies, in many cases they require semantic knowledge to parse correctly. For example, in Figure 3, the enhanced dependencies represent the shared subject ("he") among the conjoined predicates ("went" and "finished"), and the conjoined modifiers ("efficiently" and "promptly") for the second predicate ("finished"). However, there are no enhanced dependencies between the first predicate and the second predicate's modifiers (e.g. "went" → "efficiently"), as semantically only the subject is shared and not the modifiers.

Relative clauses. Finally, enhanced graphs attach predicates of relative clauses directly to the antecedent modified by the relative clause, adding a `ref` dependency between the antecedent and the relative pronoun. An example is shown in Figure 5a. While these graphs may contain cycles ("robe" ↔ "made" in the example), they are removed upon conversion to the unified DAG format by the introduction of non-terminal nodes (see Figure 5b).

3 General Transition-based DAG Parser

We now turn to describing TUPA (Hershcovich et al., 2017, 2018), a general transition-based parser (Nivre, 2003). TUPA uses an extended set of transitions and features that supports reentrancies, discontinuities and non-terminal nodes. The parser state is composed of a buffer B of tokens

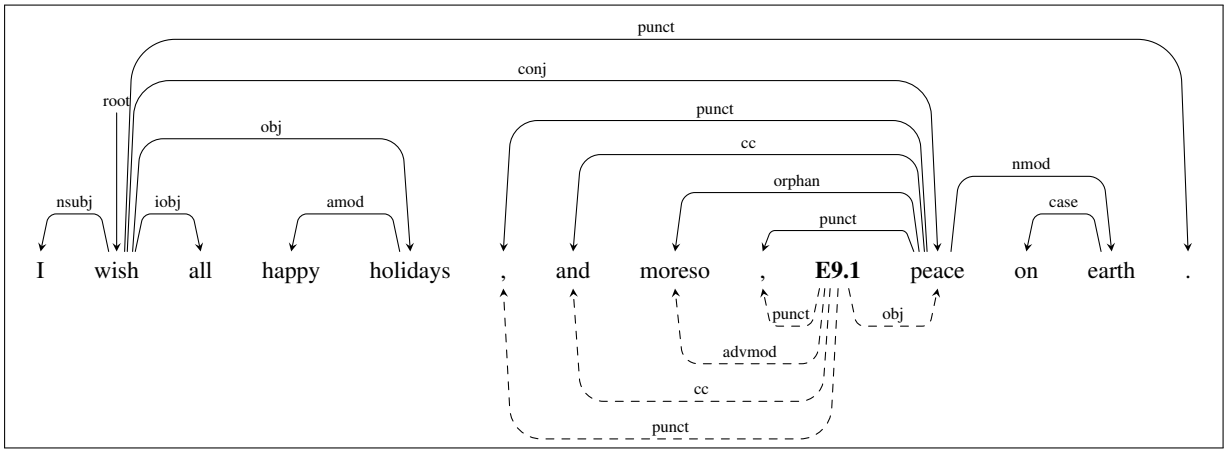


Figure 4: `newsgroup-groups.google.com_GuildWars_086f0f64ab633ab3_ENG_20041111_173500-0051` (`UD_English-EWT`), containing a null node (**E9.1**) and case information (nmod:on).

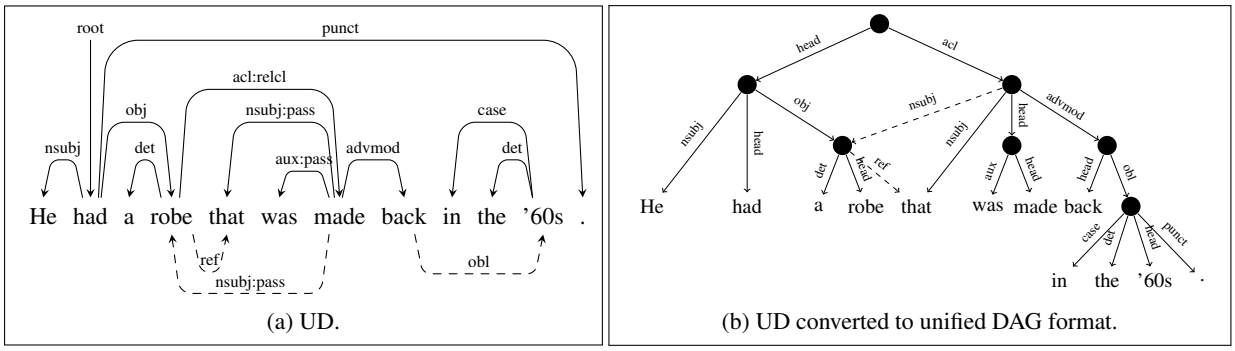


Figure 5: (a) `reviews-255261-0007` (`UD_English-EWT`), containing a relative clause, and (b) the same graph after conversion to the unified DAG format. The cycle is removed due to the non-terminal nodes introduced in the conversion.

and nodes to be processed, a stack S of nodes currently being processed, and a graph $G = (V, E, \ell)$ of constructed nodes and edges, where V is the set of *nodes*, E is the set of *edges*, and $\ell : E \rightarrow L$ is the *label* function, L being the set of possible labels. Some states are marked as *terminal*, meaning that G is the final output. A classifier is used at each step to select the next transition based on features encoding the parser's current state. During training, an oracle creates training instances for the classifier, based on gold-standard annotations.

3.1 Transition Set

Given a sequence of tokens $w_1, \ldots, w_n$, we predict a rooted graph G whose terminals are the tokens. Parsing starts with the root node on the stack, and the input tokens in the buffer.

The TUPA transition set, shown in Figure 6, includes the standard SHIFT and REDUCE operations, NODE$_X$ for creating a new non-terminal node and an X-labeled edge, LEFT-EDGE$_X$ and RIGHT-EDGE$_X$ to create a new primary X-labeled edge, LEFT-REMOTE$_X$ and RIGHT-REMOTE$_X$ to create a new remote X-labeled edge, SWAP to handle discontinuous nodes, and FINISH to mark the state as terminal.

The REMOTE$_X$ transitions are not required for parsing trees, but as we treat the problem as general DAG parsing due to the inclusion of enhanced dependencies, we include these transitions.

3.2 Transition Classifier

To predict the next transition at each step, TUPA uses a BiLSTM with feature embeddings as inputs, followed by an MLP and a softmax layer for classification. The model is illustrated in Figure 7. Inference is performed greedily, and training is done with an oracle that yields the set of all optimal transitions at a given state (those that lead to a state from which the gold graph is still reachable). Out of this set, the actual transition performed in training is the one with the highest score given by the classifier, which is trained to maximize the sum of log-likelihoods of all optimal transitions at each step.

Before Transition				Transition	After Transition					Condition
Stack	Buffer	Nodes	Edges		Stack	Buffer	Nodes	Edges	Terminal?	
S	$x \mid B$	V	E	SHIFT	$S \mid x$	B	V	E	$-$	
$S \mid x$	B	V	E	REDUCE	S	B	V	E	$-$	
$S \mid x$	B	V	E	NODE$_X$	$S \mid x$	$y \mid B$	$V \cup \{y\}$	$E \cup \{(y,x)_X\}$	$-$	$x \neq \text{root}$
$S \mid y, x$	B	V	E	LEFT-EDGE$_X$	$S \mid y, x$	B	V	$E \cup \{(x,y)_X\}$	$-$	$\begin{cases} x \notin w_{1:n}, \\ y \neq \text{root}, \\ y \not\rightsquigarrow_G x \end{cases}$
$S \mid x, y$	B	V	E	RIGHT-EDGE$_X$	$S \mid x, y$	B	V	$E \cup \{(x,y)_X\}$	$-$	
$S \mid y, x$	B	V	E	LEFT-REMOTE$_X$	$S \mid y, x$	B	V	$E \cup \{(x,y)^*_X\}$	$-$	
$S \mid x, y$	B	V	E	RIGHT-REMOTE$_X$	$S \mid x, y$	B	V	$E \cup \{(x,y)^*_X\}$	$-$	
$S \mid x, y$	B	V	E	SWAP	$S \mid y$	$x \mid B$	V	E	$-$	$\text{i}(x) < \text{i}(y)$
[root]	$\emptyset$	V	E	FINISH	$\emptyset$	$\emptyset$	V	E	$+$	

Figure 6: The transition set of TUPA. We write the stack with its top to the right and the buffer with its head to the left. $(\cdot,\cdot)_X$ denotes a primary X-labeled edge, and $(\cdot,\cdot)^*_X$ a remote X-labeled edge. $\text{i}(x)$ is the swap index (see §3.3). In addition to the specified conditions, the prospective child in an EDGE transition must not already have a primary parent.

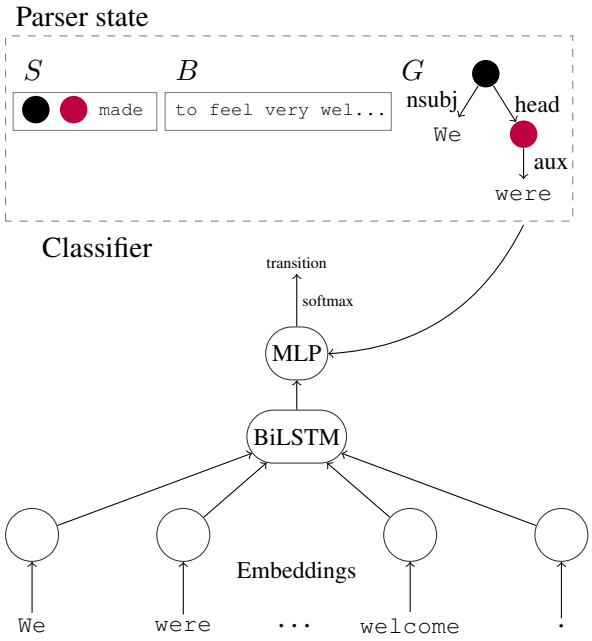


Figure 7: Illustration of the TUPA model, adapted from Hershcovich et al. (2018). Top: parser state (stack, buffer and intermediate graph). Bottom: BiLTSM architecture. Vector representation for the input tokens is computed by two layers of bidirectional LSTMs. The vectors for specific tokens are concatenated with embedding and numeric features from the parser state (for existing edge labels, number of children, etc.), and fed into the MLP for selecting the next transition.

Features. We use vector embeddings representing the words, lemmas, coarse (universal) POS tags and fine-grained POS tags, provided by UDPipe 1.2 during test. For training, we use the gold-annotated lemmas and POS tags. In addition, we use one-character prefix, three-character suffix, shape (capturing orthographic features, e.g., "Xxxx") and named entity type, provided by spaCy;[4] punctuation and gap type features (Maier and Lichte, 2016), and previously predicted edge labels and parser actions. These embeddings are initialized randomly, except for the word embeddings, which are initialized with the 250K most frequent word vectors from fastText for each language (Bojanowski et al., 2017),[5] pre-trained over Wikipedia and updated during training. We do not use word embeddings for languages without pre-trained fastText vectors (Ancient Greek, North Sami and Old French).

To the feature embeddings, we concatenate numeric features representing the node height, number of (remote) parents and children, and the ratio between the number of terminals to total number of nodes in the graph G.

Table 1 lists all feature used for the classifier. Numeric features are taken as they are, whereas categorical features are mapped to real-valued embedding vectors. For each non-terminal node, we select a *head terminal* for feature extraction, by traversing down the graph according to a priority order on edge labels (otherwise selecting the leftmost child). The priority order is:

`parataxis, conj, advcl, xcomp`

3.3 Constraints

During training and parsing, we apply constraints on the parser state to limit the possible transitions to valid ones.

A generic constraint implemented in TUPA is that stack nodes that have been swapped should not be swapped again (Hershcovich et al., 2018). To implement this constraint, we define a *swap index* for each node, assigned when the node is created. At initialization, only the root node and terminals exist. We assign the root a swap index of 0, and for each terminal, its position in the text (starting at 1). Whenever a node is created as a result

[4] http://spacy.io

[5] http://fasttext.cc

Nodes	
s_0	`wmtuepT#^$xhqyPCIEMN`
s_1	`wmtueT#^$xhyN`
s_2	`wmtueT#^$xhy`
s_3	`wmtueT#^$xhyN`
b_0	`wmtuT#^$hPCIEMN`
b_1, b_2, b_3	`wmtuT#^$`
$s_0l, s_0r, s_1l, s_1r,$ $s_0ll, s_0lr, s_0rl, s_0rr,$ $s_1ll, s_1lr, s_1rl, s_1rr$	`wme#^$`
$s_0L, s_0R, s_1L,$ s_1R, b_0L, b_0R	`wme#^$`
Edges	
$s_0 \to s_1, s_0 \to b_0,$ $s_1 \to s_0, b_0 \to s_0$	`x`
$s_0 \to b_0, b_0 \to s_0$	`e`
Past actions	
a_0, a_1	`eA`
Global	`node ratio`

Table 1: Transition classifier features.
s_i: stack node i from the top. b_i: buffer node i.
xl, xr (xL, xR): x's leftmost and rightmost children (parents). `w`: head terminal text. `m`: lemma. `u`: coarse (universal) POS tag. `t`: fine-grained POS tag. `h`: node's height. `e`: label of its first incoming edge. `p`: any separator punctuation between s_0 and s_1. `q`: count of any separator punctuation between s_0 and s_1. `x`: numeric value of gap type (Maier and Lichte, 2016). `y`: sum of gap lengths. `P`, `C`, `I`, `E`, and `M`: number of parents, children, implicit children, remote children, and remote parents. `N`: numeric value of the head terminal's named entity IOB indicator. `T`: named entity type. `#`: word shape (capturing orthographic features, e.g. "Xxxx" or "dd"). `^`: one-character prefix. `$`: three-character suffix.
$x \to y$ refers to the existing edge from x to y. `x` is an indicator feature, taking the value of 1 if the edge exists or 0 otherwise, `e` refers to the edge label, and a_i to the transition taken $i+1$ steps ago.
`A` refers to the action type (e.g. SHIFT/RIGHT-EDGE/NODE), and `e` to the edge label created by the action.
`node ratio` is the ratio between non-terminals and terminals (Hershcovich et al., 2017).

of a NODE transition, its swap index is the arithmetic mean of the swap indices of the stack top and buffer head.

In addition, we enforce UD-specific constraints, resulting from the nature of the converted DAG format: every non-terminal node must have a single outgoing `head` edge: once it has one, it may not get another, and until it does, the node may not be reduced.

4 Training details

The model is implemented using DyNet v2.0.3 (Neubig et al., 2017).[6] Unless otherwise noted, we use the default values provided by the package. We use the same hyperparameters as used in previous experiments on UCCA parsing (Hershcovich et al., 2018), without any hyperparameter tuning on UD treebanks.

Hyperparameter	Value
Pre-trained word dim.	300
Lemma dim.	200
Coarse (universal) POS tag dim.	20
Fine-grained POS tag dim.	20
Named entity dim.	3
Punctuation dim.	1
Shape dim.	3
Prefix dim.	2
Suffix dim.	3
Action dim.	3
Edge label dim.	20
MLP layers	2
MLP dimensions	50
BiLSTM layers	2
BiLSTM dimensions	500

Table 2: Hyperparameter settings.

4.1 Hyperparameters

We use dropout (Srivastava et al., 2014) between MLP layers, and recurrent dropout (Gal and Ghahramani, 2016) between BiLSTM layers, both with $p = 0.4$. We also use word, lemma, coarse- and fine-grained POS tag dropout with $\alpha = 0.2$ (Kiperwasser and Goldberg, 2016): in training, the embedding for a feature value w is replaced with a zero vector with a probability of $\frac{\alpha}{\#(w)+\alpha}$, where $\#(w)$ is the number of occurrences of w observed. In addition, we use *node dropout* (Hershcovich et al., 2018): with a probability of 0.1 at each step, all features associated with a single node in the parser state are replaced with zero vectors. For optimization we use a minibatch size of 100, decaying all weights by 10^{-5} at each update, and train with stochastic gradient descent for 50 epochs with a learning rate of 0.1, followed by AMSGrad (Sashank J. Reddi, 2018) for 250 epochs with $\alpha = 0.001$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. We found this training strategy better than using only one of the optimization methods, similar to findings by Keskar and Socher (2017). We select the epoch with the best LAS-F1 on the development set. Other hyperparameter settings are listed in Table 2.

4.2 Small Treebanks

For corpora with less than 100 training sentences, we use 750 epochs of AMSGrad instead of 250.

[6] http://dynet.io

For corpora with no development set, we use 10-fold cross-validation on the training set, each time splitting it to 80% training, 10% development and 10% validation. We perform the normal training procedure on the training and development subsets, and then select the model from the fold with the best LAS-F1 on the corresponding validation set.

4.3 Multilingual Model

For the purpose of parsing languages with no training data, we use a delexicalized multilingual model, trained on the shuffled training sets from all corpora, with no word, lemma, fine-grained tag, prefix and suffix features. We train this model for two epochs using stochastic gradient descent with a learning rate of 0.1 (we only trained this many epochs due to time constraints).

4.4 Out-of-domain Evaluation

For test treebanks without corresponding training data, but with training data in the same language, during testing we use the model trained on the largest training treebank in the same language.

5 Results

Official evaluation was done on the TIRA online platform (Potthast et al., 2014). Our system (named "HUJI") ranked 24th in the LAS-F1 ranking (with an average of 53.69 over all test treebanks), 23rd by MLAS (average of 44.6) and 21st by BLEX (average of 48.05). Since our system only performs dependency parsing and not other pipeline tasks, we henceforth focus on LAS-F1 (Nivre and Fang, 2017) for evaluation.

After the official evaluation period ended, we discovered several bugs in the conversion between the CoNLL-U format and the unified DAG format, which is used by TUPA for training and is output by it (see §2). We did not re-train TUPA on the training treebanks after fixing these bugs, but we did re-evaluate the already trained models on all test treebanks, and used the fixed code for converting their output to CoNLL-U. This yielded an unofficial average test LAS-F1 of 58.47, an improvement of 4.78 points over the official average score. In particular, for two test sets, `ar_padt` and `gl_ctg`, TUPA got a zero score in the official evaluation due to a bug with the treatment of multi-token words. These went up to 61.97 and 71.42, respectively. We also evaluated the trained TUPA models on all available development treebanks after fixing the bugs.

	TUPA (official)	**TUPA (unofficial)**	**UDPipe (baseline)**
All treebanks	53.69	58.47	65.80
Big treebanks	62.07	67.36	74.14
PUD treebanks	56.35	56.72	66.63
Small treebanks	36.74	41.19	55.01
Low-resource	8.53	12.68	17.17

Table 3: Aggregated test LAS-F1 scores for our system (TUPA) and the baseline system (UDPipe 1.2).

Table 3 presents the averaged scores on the shared task test sets, and Figure 8 the (official and unofficial) LAS-F1 scores obtained by TUPA on each of the test and development treebanks.

5.1 Evaluation on Enhanced Dependencies

Since the official evaluation ignores enhanced dependencies, we evaluate them separately using a modified version of the shared task evaluation script[7]. We calculate the *enhanced LAS*, identical to the standard LAS except that the set of dependencies in both gold and predicted graphs are the enhanced dependencies instead of the basic dependencies: ignoring null nodes and any enhanced dependency sharing a head with a basic one, we align the words in the gold graph and the system's graph as in the standard LAS, and define

$$P = \frac{\#\text{correct}}{\#\text{system}}, R = \frac{\#\text{correct}}{\#\text{gold}}, F1 = 2 \cdot \frac{P \cdot R}{P + R}.$$

Table 4 lists the enhanced LAS precision, recall and F1 score on the test treebanks with any enhanced dependencies, as well as the percentage of enhanced dependencies in each test treebank, calculated as $100 \cdot \frac{\#\text{enhanced}}{\#\text{enhanced}+\#\text{words}}$.

Just as remote edges in UCCA parsing are more challenging than primary edges (Hershcovich et al., 2017), parsing enhanced dependencies is a harder task than standard UD parsing, as the scores demonstrate. However, TUPA learns them successfully, getting as much as 56.63 enhanced LAS-F1 (on the Polish LFG test set).

5.2 Ablation Experiments

The TUPA transition classifier for some of the languages uses named entity features calculated

[7] https://github.com/CoNLL-UD-2018/HUJI/blob/master/tupa/scripts/conll18_ud_eval.py

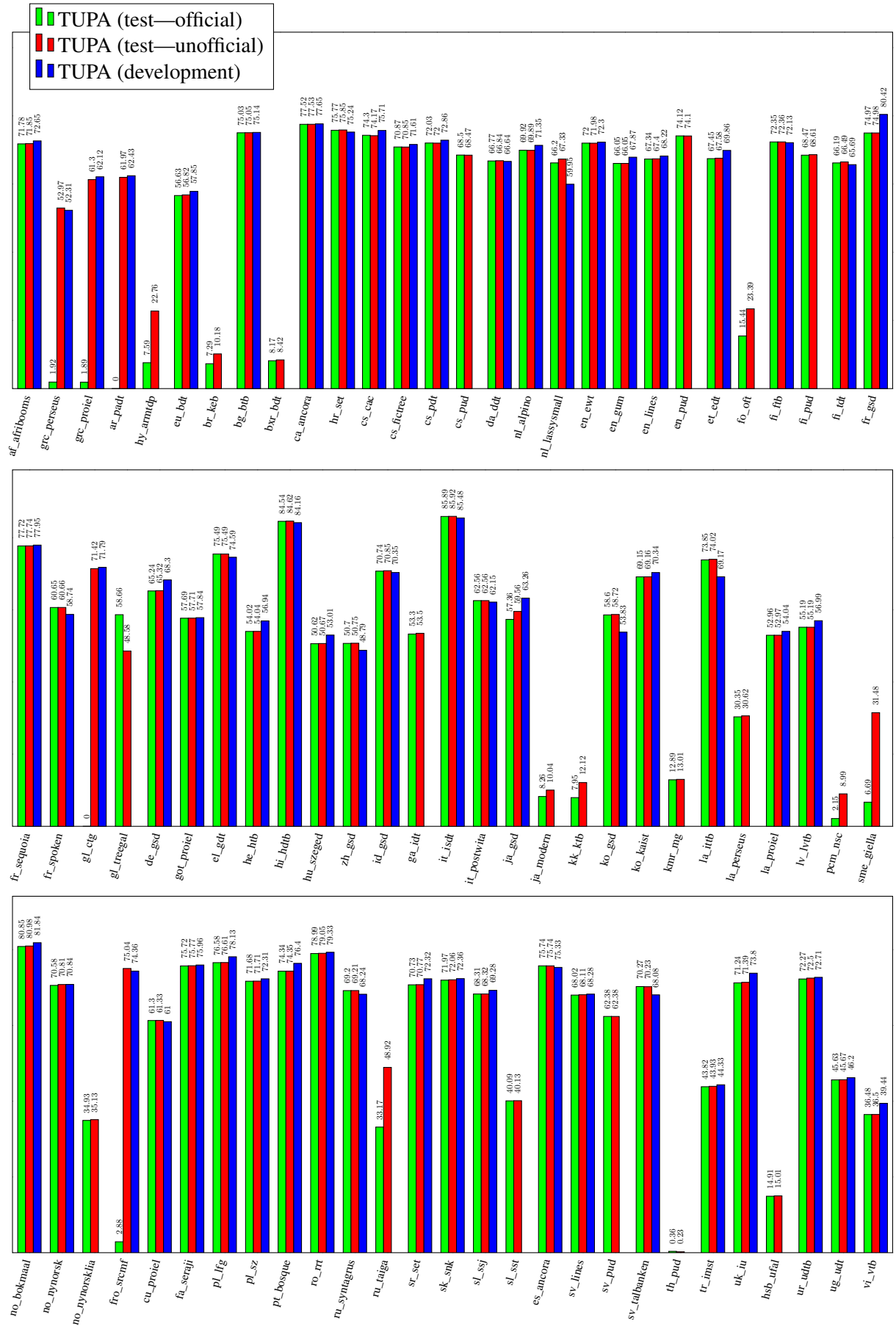


Figure 8: TUPA's LAS-F1 per treebank: official and unofficial test scores, and development scores (where available).

Treebank	Enhanced LAS P	R	F1	% Enhanced
ar_padt	28.63	16.48	20.92	5.30
cs_cac	56.13	36.62	44.32	7.57
cs_fictree	49.29	18.76	27.18	4.30
cs_pdt	50.15	27.11	35.20	4.61
nl_alpino	56.15	50.81	53.35	4.80
nl_lassysmall	49.81	51.50	50.64	4.13
en_ewt	57.66	52.58	55.00	4.36
en_pud	59.34	50.09	54.32	5.14
fi_tdt	40.73	32.03	35.86	7.34
lv_lvtb	31.87	19.01	23.81	4.12
pl_lfg	59.38	54.13	56.63	2.61
sk_snk	37.47	22.04	27.76	3.91
sv_pud	45.45	39.74	42.40	6.36
sv_talbanken	50.08	43.13	46.35	6.89

Table 4: TUPA's enhanced LAS precision, recall and F1 per test treebank with any enhanced dependencies, and percentage of enhanced dependencies in test treebank.

by spaCy.[8] For German, Spanish, Portuguese, French, Italian, Dutch and Russian, the spaCy named entity recognizer was trained on Wikipedia (Nothman et al., 2013). However, the English model was trained on OntoNotes[9], which is in fact not among the additional resources allowed by the shared task organizers. To get a fair evaluation and to quantify the contribution of the NER features, we re-trained TUPA on the English EWT (`en_ewt`) training set with the same hyperparameters as in our submitted model, just without these features. As Table 5 shows, removing the NER features (−NER) only slightly hurts the performance, by 0.26 LAS-F1 points on the test treebank, and 0.64 on the development treebank.

As further ablation experiments, we tried removing POS features, pre-trained word embeddings, and remote edges (the latter enabling TUPA to parse enhanced dependencies). Removing the POS features does hurt performance to a larger degree, by 2.85 LAS-F1 points on the test set, while removing the pre-trained word embeddings even slightly improves the performance (except the enhanced LAS on the development treebank). Removing remote edges and transitions from TUPA makes a very small improvement to LAS-F1, but of course enhanced dependencies can then no longer be produced at all.

[8] https://spacy.io/api/annotation

[9] https://catalog.ldc.upenn.edu/LDC2013T19

Model	LAS-F1 Test	Dev	Enhanced LAS-F1 Test	Dev
Original	71.98	72.30	55.00	56.12
−NER	71.72	71.66	55.59	54.83
−POS	69.13	69.38	54.17	49.29
−Embed.	72.22	72.40	56.46	54.76
−Remote	72.08	72.32	0	0

Table 5: Ablation LAS-F1 and Enhanced LAS-F1 on the English EWT development and test set. NER: named entity features. POS: part-of-speech tag features (both universal and fine-grained). Embed.: external pre-trained word embeddings (fastText). Remote: remote edges and transitions in TUPA.

6 Conclusion

We have presented the HUJI submission to the CoNLL 2018 shared task on parsing Universal Dependencies, based on TUPA, a general transition-based DAG parser. Using a simple conversion protocol to convert UD into a unified DAG format, training TUPA as-is on the UD treebanks yields results close to the UDPipe baseline for most treebanks in the standard evaluation. While other systems ignore enhanced dependencies, TUPA learns to produce them too as part of the general dependency parsing process. We believe that with hyperparameter tuning and more careful handling of cross-lingual and cross-domain parsing, TUPA can be competitive on the standard metrics too.

Furthermore, the generic nature of our parser, which supports many representation schemes, as well as domains and languages, will allow improving performance by multitask learning (cf. Hershcovich et al., 2018), which we plan to explore in future work.

Acknowledgments

This work was supported by the Israel Science Foundation (grant no. 929/17) and by the HUJI Cyber Security Research Center in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

References

Omri Abend and Ari Rappoport. 2013. Universal Conceptual Cognitive Annotation (UCCA). In *Proc. of ACL*, pages 228–238.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *TACL*, 5:135–146.

Yarin Gal and Zoubin Ghahramani. 2016. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. In D D Lee, M Sugiyama, U V Luxburg, I Guyon, and R Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 1019–1027. Curran Associates, Inc.

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2017. A transition-based directed acyclic graph parser for UCCA. In *Proc. of ACL*, pages 1127–1138.

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2018. Multitask parsing across semantic representations. In *Proc. of ACL*, pages 373–385.

Nitish Shirish Keskar and Richard Socher. 2017. Improving generalization performance by switching from Adam to SGD. *CoRR*, abs/1712.07628.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *TACL*, 4:313–327.

Wolfgang Maier and Timm Lichte. 2016. Discontinuous parsing with continuous trees. In *Proc. of Workshop on Discontinuous Structures in NLP*, pages 47–57.

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. *CoRR*, abs/1701.03980.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proc. of IWPT*, pages 149–160.

Joakim Nivre and Chiao-Ting Fang. 2017. Universal dependency evaluation. In *Proceedings of the NoDaLiDa 2017 Workshop on Universal Dependencies (UDW 2017)*, pages 86–95.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D. Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. 2016. Universal dependencies v1: A multilingual treebank collection. In *Proc. of LREC*.

Joel Nothman, Nicky Ringland, Will Radford, Tara Murphy, and James R Curran. 2013. Learning multilingual named entity recognition from wikipedia. *Artificial Intelligence*, 194:151–175.

Martin Potthast, Tim Gollub, Francisco Rangel, Paolo Rosso, Efstathios Stamatatos, and Benno Stein. 2014. Improving the reproducibility of PAN's shared tasks: Plagiarism detection, author identification, and author profiling. In *Information Access Evaluation meets Multilinguality, Multimodality, and Visualization. 5th International Conference of the CLEF Initiative (CLEF 14)*, pages 268–299, Berlin Heidelberg New York. Springer.

Siva Reddy, Oscar Täckström, Slav Petrov, Mark Steedman, and Mirella Lapata. 2017. Universal semantic parsing. pages 89–101.

Sanjiv Kumar Sashank J. Reddi, Satyen Kale. 2018. On the convergence of Adam and beyond. *ICLR*.

Sebastian Schuster and Christopher D. Manning. 2016. Enhanced English Universal Dependencies: An improved representation for natural language understanding tasks. In *Proc. of LREC*. ELRA.

Natalia Silveira, Timothy Dozat, Marie-Catherine de Marneffe, Samuel Bowman, Miriam Connor, John Bauer, and Chris Manning. 2014. A gold standard dependency corpus for English. In *Proc. of LREC*.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

Milan Straka, Jan Hajič, and Jana Straková. 2016. UDPipe: trainable pipeline for processing CoNLL-U files performing tokenization, morphological analysis, POS tagging and parsing. In *Proceedings of the 10th International Conference on Language Resources and Evaluation (LREC 2016)*, Portoro, Slovenia. European Language Resources Association.

Milan Straka and Jana Straková. 2017. Tokenizing, pos tagging, lemmatizing and parsing ud 2.0 with udpipe. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 88–99, Vancouver, Canada. Association for Computational Linguistics.

Daniel Zeman, Filip Ginter, Jan Hajič, Joakim Nivre, Martin Popel, and Milan Straka. 2018. CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, Brussels, Belgium. Association for Computational Linguistics.