TUPA at MRP 2019
A Multi-Task Baseline System
Hershcovich, Daniel; Arviv, Ofir

Published in:
Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 CoNLL

Publication date:
2020

Document version
Publisher’s PDF, also known as Version of record

Document license:
CC BY

Citation for published version (APA):
TUPA at MRP 2019: A Multi-Task Baseline System

Daniel Hershcovich* and Ofir Arviv**

*University of Copenhagen, Department of Computer Science
**Hebrew University of Jerusalem, School of Computer Science and Engineering
hershcovich@di.ku.dk, ofir.arviv@mail.huji.ac.il

Abstract

This paper describes the TUPA system submission to the shared task on Cross-Framework Meaning Representation Parsing (MRP) at the 2019 Conference for Computational Language Learning (CoNLL). TUPA provides a baseline point of comparison and is not considered in the official ranking of participating systems. While originally developed for UCCA only, TUPA has been generalized to support all MRP frameworks included in the task, and trained using multi-task learning to parse them all with a shared model. It is a transition-based parser with a BiLSTM encoder, augmented with BERT contextualized embeddings.

1 Introduction

TUPA (Transition-based UCCA/Universal Parser; Hershcovich et al., 2017) is a general transition-based parser for directed acyclic graphs (DAGs), originally designed for parsing text to graphs in the UCCA framework (Universal Conceptual Cognitive Annotation; Abend and Rappoport, 2013). It was used as the baseline system in SemEval 2019 Task 1: Cross-lingual Semantic Parsing with UCCA (Hershcovich et al., 2019b), where it was outranked by participating team submissions in all tracks (open and closed in English, German and French), but was also among the top 5 best-scoring systems in all tracks, and reached second place in the English closed tracks.

Being a general DAG parser, TUPA has been shown (Hershcovich et al., 2018a,b) to support other graph-based meaning representations and similar frameworks, including UD (Universal Dependencies; Nivre et al., 2019), which was the focus of CoNLL 2017 and 2018 Shared Tasks (Zeaman et al., 2017, 2018); AMR (Abstract Meaning Representation; Banerescu et al., 2013), targeted in SemEval 2016 and 2017 Shared Tasks (May, 2016; May and Priyadarshi, 2017); and DM (DELPH-IN MRS Bi-Lexical Dependencies; Ivanova et al., 2012), one of the target representations, among PAS and PSD (Prague Semantic Dependencies; Hajic et al., 2012; Miyao et al., 2014), in the SemEval 2014 and 2015 Shared Tasks on SDP (Semantic Dependency Parsing; Oepen et al., 2014, 2015, 2016). DM is converted from DeepBank (Flickinger et al., 2012), a corpus of hand-corrected parses from LinGO ERG (Copestake and Flickinger, 2000), an HPSG (Pollard and Sag, 1994) using Minimal Recursion Semantics (Copestake et al., 2005). EDS (Elementary Dependency Structures; Oepen and Lønning, 2006) is another framework derived from ERG, encoding English Resource Semantics in a variable-free semantic dependency graph.

The CoNLL 2019 Shared Task (Oepen et al., 2019) combines five frameworks for graph-based meaning representation: DM, PSD, EDS, UCCA and AMR. For the task, TUPA was extended to support the MRP format and frameworks, and is used as a baseline system, both as a single-task system trained separately on each framework, and as a multi-task system trained on all of them. The code is publicly available.1

2 Intermediate Graph Representation

Meaning representation graphs in the shared tasks are distributed in, and expected to be parsed to, a uniform graph interchange format, serialized as JSON Lines.2

The formalism encapsulates annotation for graphs containing nodes (corresponding either to text tokens, concepts, or logical predications), with the following components: top nodes, node

1https://github.com/danielhers/tupa/tree/mrp
2http://mrp.nlpl.eu/index.php?page=4
After graduation, John moved to New York City. 

Figure 1: Left: AMR graph, in the MRP formalism, for the sentence “After graduation, John moved to New York City.” Edge labels are shown on the edges. Node labels are shown inside the nodes, along with any node properties (in the form property=value). The text tokens are not part of the graph, and are matched to nodes by automatic alignment (anchoring). Right: converted AMR graph in the intermediate graph representation. Same as in the intermediate graph representation for all frameworks, it contains a virtual root node attached to the graph’s top node with a TOP edge, and virtual terminal nodes corresponding to text tokens, attached according to the anchoring (or, for AMR, the provided automatic alignments) with ANCHOR edges. Same as for all frameworks with node labels and properties (i.e., all but UCCA), labels and properties are replaced with placeholders corresponding to anchored tokens, where possible. The placeholder \langle ℓ \rangle corresponds to the concatenated lemmas of anchored tokens. Specifically for AMR, name operator properties (e.g., op* for New York City) are collapsed to single properties.

2.1 Roots and Anchors

TUPA supports parsing to rooted graphs with labeled edges, and with the text tokens as terminals (leaves), which is the standard format for UCCA graphs. However, MRP graphs are not given in this format, since there may be multiple roots and the text tokens are only matched to the nodes by anchoring (and not by explicit edges).

For the CoNLL 2019 Shared Task, TUPA was extended to support node labels, node properties, and edge attributes (see §3.1). Top nodes and anchoring are combined into the graph by adding a virtual root node and virtual terminal nodes, respectively, during preprocessing.

A virtual terminal node is created per token according to the tokenization predicted by UDPipe (Straka and Straková, 2017) and provided as companion data by the task organizers. All top nodes are attached as children of the virtual root with a TOP-labeled edge.

Nodes with anchoring are attached to the virtual terminals associated with the tokens whose character spans intersect with their anchoring, with ANCHOR-labeled edges. Note that anchoring is automatically determined for training in the case of AMR, using the alignments from the companion data, computed by the ISI aligner (Pourdamghani et al., 2014). There is no special treatment of non-trivial anchoring for EDS: in case a node is anchored to multiple tokens (as is the case for multi-word expressions), they are all attached with ANCHOR-labeled edges, resulting in possibly multiple parents for some virtual terminal nodes.

During inference, after TUPA returns an output graph, the virtual root and terminals are removed as postprocessing to return the final graph. Top nodes and anchoring are then inserted accordingly.

2.2 Placeholder Insertion

The number of distinct node labels and properties is very large for most frameworks, resulting in severe sparsity, as they are taken from an open vo-
<table>
<thead>
<tr>
<th>Before Transition</th>
<th>Transition</th>
<th>After Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Buffer</td>
<td>N</td>
</tr>
<tr>
<td>$S \mid x \mid B \mid V \mid E$</td>
<td>$S \mid x \mid B \mid V \mid E$</td>
<td>$S \mid x \mid B \mid V \mid E$</td>
</tr>
<tr>
<td>$S \mid x \mid B \mid V \mid E$</td>
<td>$S \mid y \mid B \mid V \cup {y}$</td>
<td>$E \mid (y, x)$</td>
</tr>
<tr>
<td>$S \mid x \mid B \mid V \mid E$</td>
<td>$S \mid y \mid B \mid V \cup {y}$</td>
<td>$E \mid (x, y)$</td>
</tr>
<tr>
<td>$S \mid x \mid B \mid V \mid E$</td>
<td>$S \mid x \mid B \mid V \mid E$</td>
<td>$E \mid \ell_{V}(x) \leftarrow X$</td>
</tr>
<tr>
<td>$S \mid y \mid B \mid V \mid E$</td>
<td>$S \mid y \mid B \mid V \mid E$</td>
<td>$E \mid \ell_{V}(y) \leftarrow X$</td>
</tr>
<tr>
<td>$S \mid x, y \mid B \mid V \mid E$</td>
<td>$S \mid x, y \mid B \mid V \mid E$</td>
<td>$E \mid \ell_{E}(x, y) \leftarrow X$</td>
</tr>
<tr>
<td>$S \mid x, y \mid B \mid V \mid E$</td>
<td>$S \mid x, y \mid B \mid V \mid E$</td>
<td>$E \mid \ell_{E}(y, x) \leftarrow X$</td>
</tr>
<tr>
<td>$S \mid B \mid V \mid E \mid (x, y)$</td>
<td>$S \mid B \mid V \mid E \mid (x, y)$</td>
<td>$E \mid \ell_{E}(x, y) \leftarrow X$</td>
</tr>
<tr>
<td>$S \mid x, y \mid B \mid V \mid E$</td>
<td>$S \mid y \mid x \mid B \mid V \mid E$</td>
<td>$E \mid \ell_{V}(x) \leftarrow X$</td>
</tr>
<tr>
<td>[root]</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Figure 2: The TUPA-MRP transition set. We write the stack with its top to the right and the buffer with its head to the left; the set of edges is also ordered with the latest edge on the right. Node, Label, Property and Attribute require that $x \neq \text{root}$; Child, Label, Property, Left-Edge and Right-Edge require that $x \notin w_{1:n}$; Attribute requires that $y \notin w_{1:n}$; Left-Edge and Right-Edge require that $y \neq \text{root}$ and that there is no directed path from $y$ to $x$; and Swap requires that $i(x) < i(y)$, where $i(x)$ is the swap index (see §3.5).

To reduce the number of unique node labels and properties, we use the (possibly automatic) anchoring and UDPipe preprocessing to introduce placeholders in the values. For example, a node labeled move-01 anchored to the token moved will be instead labeled $\langle \ell \rangle \cdot 01$, where $\langle \ell \rangle$ is a placeholder for the token’s lemma. In this way we reduce the number of node labels in the AMR training set, for example, from tens of thousands to 7,300, of which 2,000 occur only once and are treated as unknown. We use similar placeholders for the token’s surface form. Placeholders are resolved back to the full value after an output graph is produced by the parser, according to the anchoring in the graph. While nodes labels and properties sometimes have a non-trivial relationship to the text tokens, in most cases they contain the lemma or surface form, making this a simple and effective solution.

While more sophisticated alignment rules have been developed (Flanigan et al., 2014; Pourdamghani et al., 2014), such as using entity linking (Daiber et al., 2013), as employed by Bjerva et al. (2016); van Noord and Bos (2017), in this baseline system we are employing a simple strategy without relying on external, potentially non-whitelisted resources.

Named entities in AMR are expressed by name-labeled nodes, with a property for each token in the name, with keys op1, op2, etc. We instead collapse these properties to a single op property whose label is the concatenation of the name tokens, with special separator symbols. This value is in turn replaced by a placeholder, if the node is anchored and the anchored tokens match the property. Figure 1 demonstrates an AMR graph before and after the conversion to the intermediate graph representation.

3 Transition-based Meaning Representation Parser

TUPA is a transition-based parser (Nivre, 2003), constructing graphs incrementally from input tokens by applying transitions (actions) to the parser state (configuration). The parser state is composed of a buffer $B$ of tokens and nodes to be processed, a stack $S$ of nodes currently being processed, and an incrementally constructed graph $G$. Some states are marked as terminal, meaning that $G$ is the final output. The input to the parser is a sequence of tokens: $w_1, \ldots, w_n$. Parsing starts with a (virtual) root node on the stack, and the input tokens in the buffer, as (virtual) terminal nodes.

Given a gold-standard graph and a parser state, an oracle returns the set of gold transitions to apply at the next step, i.e., all transitions that preserve the reachability of the gold target graph. A classifier is trained using the oracle to select...
the next transition based on features encoding the parser’s current state, where the training objective is to maximize the sum of log-likelihoods of all gold transitions at each step. If there are multiple gold transitions, the highest-scoring one is taken in training. Inference is performed greedily: the highest-scoring transition is always taken.

Formally, the incrementally constructed graph $G$ consists of $(V, E, \ell_V, \ell_E, p, a)$, where $V$ is the set of nodes, $E$ is the sequence of directed edges, $\ell_V : V \rightarrow L_V$ is the node label function, $L_V$ being the set of possible node labels, $\ell_E : E \rightarrow L_E$ is the edge label function, $L_E$ being the set of possible edge labels, $p : V \rightarrow \mathcal{P}(P)$ is the node property function, $P$ being the set of possible node property-value pairs, and $a : E \rightarrow \mathcal{P}(A)$ is the edge attribute function, $A$ being the set of possible edge attribute-value pairs (a node may have any number of properties; an edge may have any number of attributes).

### 3.1 Transition Set

The set of possible transitions in TUPA is based on a combination of transition sets from other parsers, designed to support reentrancies (Sagae and Tsujii, 2008; Tokgöz and Eryiğit, 2015), discontinuities (Nivre, 2009; Maier, 2015; Maier and Lichte, 2016) and non-terminal nodes (Zhu et al., 2013). Beyond the original TUPA transitions (Hershcovich et al., 2017, 2018a), for the CoNLL 2019 Shared Task, transitions are added to support node labels, node properties, and edge attributes. Additionally, top nodes and node anchoring are encoded by special edges from a virtual root node and to virtual terminal nodes (corresponding to text tokens), respectively (see § 2).

The TUPA-MRP transition set is shown in Figure 2. It includes the following original TUPA transitions: the standard $\text{SHIFT}$ and $\text{REDUCE}$ operations (to move a node from the buffer to the stack and to discard a stack node, respectively), $\text{NODE}_X$ for creating a new non-terminal node and an $X$-labeled edge (so that the new node is a parent of the stack top), $\text{LEFT-EDGE}_X$ and $\text{RIGHT-EDGE}_X$ to create a new $X$-labeled edge, $\text{SWAP}$ to handle discontinuous nodes (moving the second topmost stack node back to the buffer), and $\text{FINISH}$ to mark the state as terminal.
Besides the original TUPA transitions, TUPA-MRP contains a CHILD transition to create unanchored children for existing nodes (like NODE, but the new node is a child of the stack top), a LABEL transition to select a label for an existing node (either the stack top of the second topmost stack node), a PROPERTY transition to select a property-value pair for an existing node, and an ATTRIBUTE transition to select an attribute-value pair for an existing edge (the last created edge).

The original TUPA transitions LEFT-REMOTE$_X$ and RIGHT-REMOTE$_X$, creating new remote edges (a UCCA-specific distinction), are omitted. Remote edges are encoded instead as edges with the remote attribute, and are supported by the combination of EDGE and ATTRIBUTE transitions. In contrast to the original TUPA transitions, EDGE transitions are allowed to attach multiple parents to a node.

### 3.2 Transition Classifier

To predict the next transition at each step, TUPA uses a BiLSTM module followed by an MLP and a softmax layer for classification (Kiperwasser and Goldberg, 2016). The model is illustrated in Figure 3.

The BiLSTM module (illustrated in more detail in Figure 4) is applied before the transition sequence starts, running over the input tokenized sequence. It consists of a pre-BiLSTM MLP with feature embeddings (§3.3) and pre-trained contextualized embeddings (§3.4) concatenated as inputs, followed by (multiple layers of) a bidirectional recurrent neural network (Schuster and Paliwal, 1997; Graves, 2008) with a long short-term memory cell (Hochreiter and Schmidhuber, 1997). While edge labels are combined into the identity of the transition (so that for example, LEFT-EDGED and LEFT-EDGES are separate transitions in the output), there is just one transition for each of LABEL, PROPERTY and ATTRIBUTE. After each time one of these transition is selected, an additional classifier is evoked with the set of possible values for the currently parsed framework. This hard separation is made due to the large number of node labels and properties in the MRP frameworks. Since there is only one possible edge attribute value (remote for UCCA), performing this transition always results in this value being selected.

### 3.3 Features

In both training and testing, we use vector embeddings representing the lemmas, coarse POS tags (UPOS) and fine-grained POS tags (XPOS). These feature values are provided by UDPipe as companion data by the task organizers. In addition, we use punctuation and gap type features (Maier and Lichte, 2016), and previously predicted node and edge labels, node properties, edge attributes and parser actions. These embeddings are initialized randomly (Glorot and Bengio, 2010).

To the feature embeddings, we concatenate numeric features representing the node height, number of parents and children, and the ratio between the number of terminals to total number of nodes in the graph $G$. 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, selecting the first outgoing edge each time according to alphabetical order of labels.

### 3.4 Pre-trained Contextualized Embeddings

Contextualized representation models such as BERT (Devlin et al., 2019) have recently achieved state-of-the-art results on a diverse array of downstream NLP tasks, gaining improved results compared to non-contextual representations. We use
the weighted sum of last four hidden layers of a BERT pre-trained model as extra input features.\footnote{We used the \texttt{bert-large-cased} model from \url{https://github.com/huggingface/pytorch-transformers}.}

BERT uses a wordpiece tokenizer (Wu et al., 2016), which segments all text into sub-word units, while TUPA uses the UDPipe tokenization. To maintain alignment between wordpieces and tokens, we use a summation of the outputs of BERT vectors corresponding to the wordpieces of each token as its representation.

### 3.5 Constraints

As each annotation scheme has different constraints on the allowed graph structures, we apply these constraints separately for each task. During training and parsing, the relevant constraint set rules out some of the transitions according to the parser state.

Some constraints are task-specific, others are generic. For example, in AMR, a node with an incoming \texttt{NAME} edge must have the \texttt{NAME} label. In UCCA, a node may have at most one outgoing edge with label \(\in \{\text{PROCESS, STATE}\}\).

An example of a generic constraint is that stack nodes that have been swapped should not be swapped again, to avoid infinite loops in inference. To implement this constraint, we define a \textit{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 of a \texttt{NODE} or \texttt{CHILD} transition, its swap index is the arithmetic mean of the swap indices of the stack top and buffer head. While this constraint may theoretically limit the ability to parse arbitrary graphs, in practice we find that all graphs in the shared task training set can still be reached without violating it.

### 4 Multi-Task Learning

Whereas in the single-task setting TUPA is trained separately on each framework as described above, in the multi-task setting, all frameworks share a BiLSTM for encoding the input. In addition, each framework has a framework-specific BiLSTM, private to it. Each framework has its own MLP on top of the concatenation of the shared and framework-specific BiLSTM (see Figure 3).

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lemma dim.</td>
<td>200</td>
</tr>
<tr>
<td>UPOS dim.</td>
<td>20</td>
</tr>
<tr>
<td>XPOS dim.</td>
<td>20</td>
</tr>
<tr>
<td>Dep. rel. dim.</td>
<td>10</td>
</tr>
<tr>
<td>Punct. dim.</td>
<td>1</td>
</tr>
<tr>
<td>Action dim.</td>
<td>3</td>
</tr>
<tr>
<td>Node label dim.</td>
<td>20</td>
</tr>
<tr>
<td>Node prop. dim.</td>
<td>20</td>
</tr>
<tr>
<td>Edge label dim.</td>
<td>20</td>
</tr>
<tr>
<td>Edge attrib. dim.</td>
<td>1</td>
</tr>
<tr>
<td>MLP layers</td>
<td>2</td>
</tr>
<tr>
<td>MLP dim.</td>
<td>50</td>
</tr>
<tr>
<td>Shared BiLSTM layers</td>
<td>2</td>
</tr>
<tr>
<td>Shared BiLSTM dim.</td>
<td>500</td>
</tr>
<tr>
<td>Shared pre-BiLSTM MLP layers</td>
<td>1</td>
</tr>
<tr>
<td>Shared pre-BiLSTM MLP dim.</td>
<td>300</td>
</tr>
<tr>
<td>Private BiLSTM layers</td>
<td>2</td>
</tr>
<tr>
<td>Private BiLSTM dim.</td>
<td>500</td>
</tr>
<tr>
<td>Private pre-BiLSTM MLP layers</td>
<td>1</td>
</tr>
<tr>
<td>Private pre-BiLSTM MLP dim.</td>
<td>300</td>
</tr>
</tbody>
</table>

Table 1: Hyperparameter settings.

For node labels and properties and for edge attributes (when applicable), an additional “axis” (private BiLSTM and MLP) is added per framework (e.g., AMR node labels are predicted separately and with an identical architecture to AMR transitions, except the output dimension is different). This is true for the single-task setting too, so in fact the single-task setting is multi-task over \{transitions, node labels, node properties, edge attributes\}.

### 5 Training details

The model is implemented using DyNet v2.1 (Neubig et al., 2017).\footnote{\url{http://dynet.io}} 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 (Hershcovitch et al., 2018a), without any hyperparameter tuning on the CoNLL 2019 data.

#### 5.1 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 \(\alpha = 0.2\).
Table 2: Official test MRP F-scores (in %) for TUPA (single-task and multi-task). For comparison, the highest score achieved for each framework and evaluation set is shown.

(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., 2018a): 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 \( \beta_2 = 0.999 \).

Table 1 lists other hyperparameter settings.

5.2 Official Evaluation

For the official evaluation, we did not use a development set, and trained on the full training set for as many epochs as the evaluation period allowed for. The multi-task model completed just 3 epoch of training. The single task models completed 12 epochs for DM, 22 epochs for PSD, 14 epochs for EDS, 100 epochs for UCCA (the maximum number we allowed) and 13 epochs for AMR.

Due to an oversight resulting from code re-use, in the official evaluation we used non-whitelisted resources. This is similar to the delexicalization employed by Buys and Blunsom (2017a) for AMR parsing.

5.3 Post-evaluation Training

After the evaluation period, we continued training for a longer period of time, using a slightly modified system: we used only resources whitelisted by the task organizers in the post-evaluation training, removing the constraints and placeholders based on PropBank and AMR lexicons.

In this setting, training is done over a shuffled mix of the training set for all frameworks (no special sampling is done to balance the number of instances per framework), and a development set of 500 instances per framework (see § 5.1). We select the epoch with the best average MRP F-score on a development set, selected by sampling 500 random training instances from each framework (the development instances are excluded from the training set). The large multi-task model only completed 4 training epochs in the available time, the single-task models completed 24 epochs for DM, 31 epochs for PSD, 25 epochs for EDS, 69 epochs for UCCA and 23 epochs for AMR.

6 Results

Table 2 presents the averaged scores on the test sets in the official evaluation (§ 5.2), for TUPA and for the best-performing system in each framework and evaluation set. Since non-whitelisted resources were used, the TUPA scores cannot be taken as a baseline. Furthermore, due to insufficient training time, all models but the UCCA one are underfitting, while the UCCA model is overfitting due to excessive training without early stopping (no development set was used in this setting).
Table 3: Post-evaluation test scores (in %) for TUPA (single-task and multi-task), using the MRP F-score (left), and using Native Evaluation (middle): labeled SDP F-score for DM and PSD, EDM F-score for EDS, primary labeled F-score for UCCA, and Smatch for AMR. The rightmost column (Trans./Token Ratio) shows the mean ratio between length of oracle transition sequence and sentence length, over the training set.

6.1 Post-evaluation Results

Table 3 presents the averaged scores on the test sets for the post-evaluation trained models (§5.3). Strikingly, the multi-task TUPA consistently falls behind the single-task one, for each framework separately and in the overall score. This stems from several factors, namely that the sharing strategy could be improved, but mainly since the multi-task model is probably underfitting due to insufficient training. We conclude that better efficiency and faster training is crucial for practical applicability of this approach. Perhaps a smaller multi-task model would have performed better by training on more data in the available time frame.

6.2 Diagnostic Evaluation

The rightmost column of Table 3 displays the mean ratio between length of oracle transitions sequence and sentence length by framework, over the shared task training set. Scores are clearly better as the framework has longer oracle transition sequences, perhaps because many of the transitions are “easy” as they correspond to structural elements of the graphs or properties copied from the input tokens.

6.3 Comparability with Previous Results

Previous published results of applying TUPA to UCCA parsing (Hershovich et al., 2017, 2018a, 2019b,a) used a different version of the parser, without contextualized word representations from BERT.

For comparability with previous results, we train and test an identical model to the one presented in this paper, on the SemEval 2019 Task 1 data (Hershovich et al., 2019b), which is UCCA-only, but contains tracks in English, German and French. For this experiment, we use bert-multilingual instead of bert-large-cased, and train a shared model over all three languages. A 50-dimensional learned language embedding vector is concatenated to the input. Word, lemma and XPOS features are not used. No multi-task learning with other frameworks is employed. The results are shown in Table 4. While improvement is achieved uniformly over the previous TUPA scores, even with BERT, TUPA is outperformed by the shared task winners (Jiang et al., 2019). Note that Jiang et al. (2019) also used bert-multilingual in the open tracks.

We also train and test TUPA with BERT embeddings on v1.0 of the UCCA English Web Treebank (EWT) reviews dataset (Hershovich et al., 2019a). While the EWT reviews are included in the MRP shared task UCCA data, the different format and preprocessing makes for slightly different scores, so we report the scores for comparability with previous work in Table 5. We again see pronounced improvements from incorporating pre-trained contextualized embeddings into the model.

7 Related Work

Transition-based meaning representation parsing dates back already to semantic dependency parsing work by Sagae and Tsujii (2008); Tokgöz and Eryiğit (2015), who support a DAG structure by allowing multiple parents to be created by EDGE transitions, and by Titov et al. (2009), who applied a SWAP transition (Nivre, 2008) for online reordering of nodes to support non-projectivity. Transition-based parsing was applied to AMR
Table 4: Test UCCA F-score scores (in %) on all edges, primary edges and remote edges, on the SemEval 2019 Task 1 data. The previous published TUPA scores are shown (TUPA w/o BERT), as well as scores for TUPA with BERT contextualized embeddings, TUPA (w/ BERT), averaged over three separately trained models in each setting, differing only by random seed (standard deviation < 0.03); and the scores for the best-scoring system from that shared task.

Table 5: Test UCCA F-score scores (in %) on all edges, primary edges and remote edges, on the UCCA EWT reviews data. TUPA (w/o BERT) is from (Hershcovich et al., 2019a). TUPA (w/ BERT) is averaged over three separately trained models in each setting, differing only by random seed (standard deviation < 0.03).

8 Conclusion

We have presented TUPA, a baseline system in the CoNLL 2019 shared task on Cross-Framework Meaning Representation. TUPA is a general transition-based DAG parser, which is trained with multi-task learning on multiple frameworks. Its input representation is augmented with BERT contextualized embeddings.

Acknowledgments

We are grateful for the valuable feedback from the anonymous reviewers. We would like to thank the other task organizers, Stephan Oepen, Omri Abend, Jan Hajič, Tim O’Gorman and Nianwen Xue, for valuable discussions and tips on developing the baseline systems, as well as for providing the data, evaluation metrics and information on the various frameworks.

References


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


