HIT-SCIR at MRP 2019: A Unified Pipeline for Meaning Representation Parsing via Efficient Training and Effective Encoding

Wanxiang Che, Longxu Dou, Yang Xu, Yuxuan Wang, Yijia Liu, Ting Liu
Research Center for Social Computing and Information Retrieval
Harbin Institute of Technology, China
{car,lxdou,yxu,yxwang,yjliu,tliu}@ir.hit.edu.cn

Abstract

This paper describes our system (HIT-SCIR) for the CoNLL 2019 shared task: Cross-Framework Meaning Representation Parsing. We extended the basic transition-based parser with two improvements: a) **Efficient Training** by realizing stack LSTM parallel training; b) **Effective Encoding** via adopting deep contextualized word embeddings BERT (Devlin et al., 2019). Generally, we proposed a unified pipeline to meaning representation parsing, including framework-specific transition-based parsers, BERT-enhanced word representation, and post-processing. In the final evaluation, our system was ranked first according to ALL-F1 (86.2%) and especially ranked first in UCCA framework (81.67%).

1 Introduction

The goal of the CoNLL 2019 shared task (Oepen et al., 2019) is to develop a unified parsing system to process all five semantic graph banks.[1] For the first time, this task combines formally and linguistically different approaches to meaning representation in graph form in a uniform training and evaluation setup.

Recently, a lot of semantic graphbanks arise, which differ in the design of graphs (Kuhlmann and Oepen, 2016), or semantic scheme (Abend and Rappoport, 2017). More specifically, SDP (Oepen et al., 2015), including DM, PSD and PAS, treats the tokens as nodes and connect them with semantic relations; EDS (Flickinger et al., 2017) encodes MRS representations (Copestake et al., 1999) as graphs with the many-to-many relations between tokens and nodes; UCCA (Abend and Rappoport, 2013) represents semantic structures with the multi-layer framework; AMR (Banarescu et al., 2013) represents the meaning of each word using a concept graph. Koller et al. (2019) classifies these frameworks into three flavors of semantic graphs, based on the degree of alignment between the tokens and the graph nodes. In DM and PSD, nodes are sub-set of surface tokens; in EDS and UCCA, graph nodes are explicitly aligned with the tokens; in AMR, the alignments are implicit.

Most semantic parsers are only designed for one or few specific graphbanks, due to the differences in annotation schemes. For example, the currently best parser for SDP is graph-based (Dozat and Manning, 2018), which assumes dependency graphs but cannot be directly applied to UCCA, EDS, and AMR, due the existence of concept node. Hershcovich et al. (2018) parses across different semantic graphbanks (UCCA, DM, AMR), but only works well on UCCA. The system of Buys and Blunsom (2017) is a good data-driven EDS parser, but does poorly on AMR. Lindemann et al. (2019) sets a new SOTA in DM, PAS, PSD, AMR and nearly SOTA in EDS, via representing each graph with the compositional tree structure (Groschwitz et al., 2017), but they do not expand this method to UCCA. Learning from multiple flavors of meaning representation in parallel has hardly been explored, and notable exceptions include the parsers of Peng et al. (2017, 2018); Hershcovich et al. (2018).

Therefore, the main challenge in cross-framework semantic parsing task is that diverse framework differs in the mapping way between surface string and graph nodes, which incurs the incompatibility among framework-specific parsers. To address that, we propose to use transition-based parser as our basic parser, since it's more flexible to realize the mapping (node generation and alignment) compared with graph-based parser, and we improve it from the two as-

[1] See http://mrp.nlpl.eu/ for further technical details, information on how to obtain the data, and official results.

Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 CoNLL, pages 76–85

https://doi.org/10.18653/v1/K19-2007

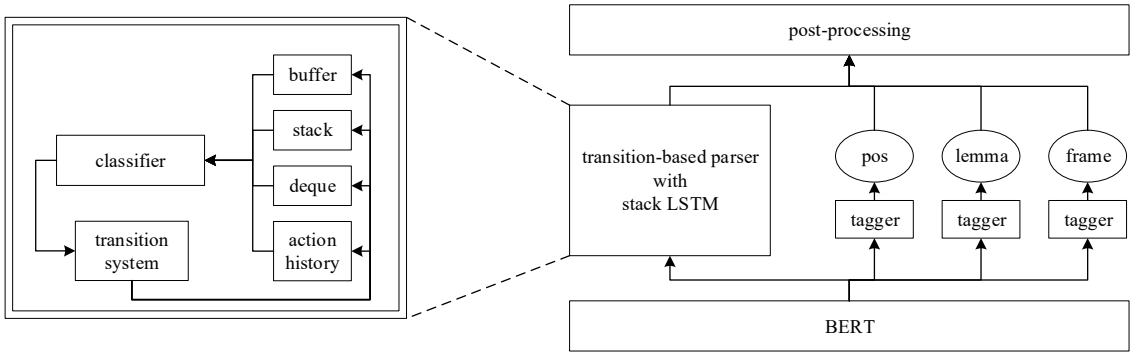

Figure 1: A unified pipeline for meaning representation parsing, including transition-based parser, BERT-enhanced word representation, and post-processing, along with the additional taggers for label the nodes with pos, frame and lemma.

pects: 1) **Efficient Training** Aligning the homogeneous operation in stack LSTM within a batch and then computing them simultaneously; 2) **Effective Encoding** Fine-tuning the parser with pre-trained BERT (Devlin et al., 2019) embedding, which enrich the context information to make accurate local decisions, and global learning for exact search. Together with the post-processing, we developed a unified pipeline for meaning representation parsing.

Our contribution can be summarised as follows:

- We proposed a unified parsing framework for cross-framework semantic parsing.
- We designed a simple but efficient method to realize stack LSTM parallel training.
- We showed that semantic parsing task benefits a lot from adopting BERT.
- Our system was ranked first in CoNLL 2019 shared task among 16 teams upon ALL-F1.

2 System Architecture

Our system architecture is shown in Figure 1. In this section, we will first introduce the transition-based parser in Section 2.1, which is the central part of our system. Then, to speed up the training of stack LSTM at transition-based parser, we propose a simple method to do batch-training in Section 2.2. And we adopt BERT to extract the contextualized word representation in Section 2.3. At last, to label the nodes with pos, frame and lemma, we use additional tagger models to predict these in Section 2.4. The framework-specific transition system is presented in Section 3 and post-processing for each framework is discussed in Section 4.

2.1 Transition-based Parser

In order to design the unified transition-based parser, we refer to the following framework-specific parsers: Wang et al. (2018b) for DM and PSD, Hershcovich et al. (2017) for UCCA, Buys and Blunsom (2017) for EDS, Liu et al. (2018) for AMR. Those parsers differ in the design of transition system to generate oracle action sequence, but similar in modeling the parsing state.

A tuple (S, L, B, E, V) is used to represent parsing state, where S is a stack holding processed words, L is a list holding words popped out of S that will be pushed back in the future, and B is a buffer holding unprocessed words. E is a set of labeled dependency arcs. V is a set of graph nodes include concept nodes and surface tokens. The initial state is $([0], [\,], [1, \cdots, n], [\,], V)$, where V only contains surface tokens since the concept nodes would be generated during parsing. And the terminal state is $([0], [\,], [\,], E, V')$. We model the S, L, B and action history with stack LSTM, which supports PUSH and POP operation. [2]

Transition classifier takes the parsing state from multiple stack LSTM models as input at once, and outputs a action that maximizes the score. The score of a transition action a on state s is calculated as

$$p(a|s) = \frac{\exp\{g_a \cdot \text{STACK LSTM}(s) + b_a\}}{\sum_{a'} \exp\{g_{a'} \cdot \text{STACK LSTM}(s) + b_{a'}\}},$$

where STACK LSTM(s) encodes the state s into a vector, g_a and b_a are embedding vector, bias vector of action a respectively. The oracle transition action sequence is obtained through transition system, proposed in in Section 3.

[2] We encourage the reader to read Dyer et al. (2015) for more details.

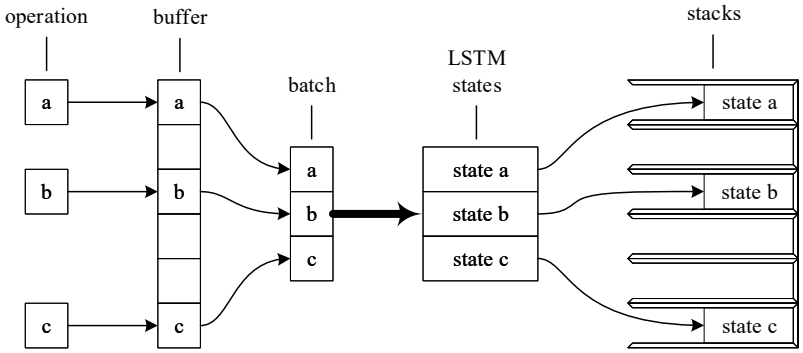

Figure 2: When some new INSERT operations come, the data to be inserted are pushed into corresponding buffers. They will be merged into a batch once batch-processing is triggered. After that, new LSTM states will be pushed to corresponding stacks.

2.2 Batch Training

Kiperwasser and Goldberg (2016) shows that batch training increases the gradient stability and speeds up the training. Delaying the backward to simulate mini-batch update is a simple way to realize batch training, but it fails to compute over data in parallel. To solve this, we propose a method of maintaining stack LSTM structure and using operation buffer.

stack LSTM The stack LSTM augments the conventional LSTM with a 'stack pointer'. And it supports the operation including: a) INSERT adds elements to the end of the sequence; b) POP moves the stack pointer to the previous element; c) QUERY returns the output vector where the stack pointer points. Among these three operation, POP and QUERY only manipulates the stack without complex computing, but INSERT performs lots of computing.

Batch Data in Operation-Level Like conventional LSTM can't form a batch inside a sequence due to the characteristics of sequential processing, stack LSTM can't either. Thus, we collect under-computed operations between different pieces of data to form a batch. In other words, we construct batch data on operation-level other than data-level in tradition. After collecting a batch of operation, we compute them simultaneously.

Operation Buffer To be more efficient, we adopt a buffer to collect operations and let it trigger the computing of those operations automatically (batch-processing), as shown in Figure 2. To ensure correctness, batch-processing will only be triggered when satisfy some conditions. More specifically, when a) operation INSERT comes and there is already an INSERT in the buffer; b) operation POP or QUERY comes. To clarify, the depth of buffer per data is 1.

2.3 BERT-Enhance Word Representation

2.3.1 Deep Contextualized Word Representations

Neural parsers often use pretrained word embeddings as their primary input, i.e. word2vec (Mikolov et al., 2013) and GloVe (Pennington et al., 2014), which assign a single static representation to each word so that they cannot capture context-dependent meaning. By contrast, deep contextualized word representations, i.e. ELMo (Peters et al., 2018) and BERT (Devlin et al., 2019), encode words with respect to the context, which have been proven to be useful for many NLP tasks, achieving state-of-the-art performance in standard Natural Language Understanding (NLU) benchmarks, such as GLUE (Wang et al., 2018a). Che et al. (2018) adopted ELMo in CoNLL 2018 shared task (Zeman et al., 2018) and achieved first prize in terms of LAS metric. (Kondratyuk and Straka, 2019) exceeds the state-of-the-art in UD with fine-tuning model with BERT.

2.3.2 BERT

We adopt BERT in our model, which uses the language-modeling objective and trained on unannotated text for getting deep contextualized embeddings. BERT differs from ELMo in that it employs a bidirectional Transformer (Vaswani et al., 2017), which benefit from learning potential dependencies between words directly. For a token w_k in sentence S, BERT splits it to several pieces and use a sequence of WordPiece embedding (Wu et al., 2016) $\mathbf{s}_{k,1}, \mathbf{s}_{k,2}, ..., \mathbf{s}_{k,piece\_num_k}$ instead of

a single token embedding. Each $\mathbf{s}_{k,i}$ is passed to an L-layered BiTransformer, which is trained with a masked language modeling objective (i.e. randomly masking a percentage of input tokens and only predicting these masked tokens).

To encode the whole sentence, we extract the first piece $\mathbf{s}_{k,1}$ of each token w_k, with applying a scalar mix on all L layers of transformer, to represent the corresponding token w_k.

2.4 Tagger

Semantic graphs in all frameworks can be broken down into 'atomic' component pieces, i.e. tuples capturing (a) top nodes, (b) node labels, (c) node properties, (d) node anchoring, (e) unlabeled edges, (f) edge labels, and (g) edge attributes. Not all tuple types apply to all frameworks, however.[3] The released dataset and evaluation is annotated by MRP, which consists of the tuple including the graph component mentioned above.

Our transition-based parser can provide the edge information, while the other node information, such as pos, frame and lemma, require us to use additional tagger models to label the sentence sequence. The tagger we adopted is directly imported from AllenNLP library, which only models the dependency between node and label (emission score), not models the dependency between labels (transition score). The details about integrating and converting system output into MRP format will be introduced in Section 4.

3 Transition Systems

Building on previous work on parsing reentrancies, discontinuities, and non-terminal nodes, we define an extended set of transitions and features that supports the conjunction of these properties. To solve cross-arc problem, we use list-based arc-eager algorithm for DM, PSD, and EDS framework as Choi and McCallum (2013); Nivre (2003, 2008); for UCCA framework, we employ SWAP operation to generate cross-arc as Hershcovich et al. (2017).[4]

3.1 DM and PSD

We follow the work of (Wang et al., 2018b) to design transition system for DM and PSD.

- LEFT-EDGE$_X$ and RIGHT-EDGE$_X$ add an arc with label X between w_j and w_i , where w_i is the top elements of stack and w_j is the top elements of buffer. They are performed only when one of w_i and w_j is the head of the other.

- SHIFT is performed when no dependency exists between w_j and any word in S other than w_i, which pushes all words in list and w_j into stack S.

- REDUCE is performed only when w_i has head and is not the head or child of any word in buffer, which pops w_i out of stack.

- PASS is performed when neither SHIFT nor REDUCE can be performed, which moves w_i to the front of list.

- FINISH pops the root node and marks the state as terminal.

3.2 UCCA

We follow the work of (Hershcovich et al., 2017) to design transition system for UCCA.

- SHIFT and REDUCE operations are the same as DM and PSD. REDUCE pops the stack, to allow removing a node once all its edges have been created.

- NODE transition creates new non-terminal nodes. For every $X \in L$, NODE$_X$ creates a new node on the buffer as a parent of the first element on the stack, with an X-labeled edge.

- LEFT-EDGE$_X$ and RIGHT-EDGE$_X$ create a new primary X-labeled edge between the first two elements on the stack, where the parent is the left or the right node, respectively.

- LEFT-REMOTE$_X$ and RIGHT-REMOTE$_X$ do not have this restriction, and the created edge is additionally marked as *remote*.

- SWAP pops the second node on the stack and adds it to the top of the buffer, as with the similarly named transition in previous work (Maier, 2015; Nivre, 2009).

- FINISH pops the root node and marks the state as terminal.

[3] For further explanation, please visit the official website:http://mrp.nlpl.eu/index.php?page=5

[4] The transition sets for each framework have been introduced with table format in supplementary material.

As a UCCA node may only have one incoming primary edge, EDGE transitions are disallowed if the child node already has an incoming primary edge. To support the prediction of multiple parents, node and edge transitions leave the stack unchanged, as in other work on transition-based dependency graph parsing (Sagae and Tsujii, 2008).

3.3 EDS

Based on the work of (Buys and Blunsom, 2017), we extended NODE-START$_L$ and NODE-END actions for generating concept node and realizing node alignment.

To clarify, w_i is the top element in stack and w_j is the top element in buffer. Moreover, w_i could only be concept node (stack and list only contain concept node), and w_j could be concept node or surface token.

- SHIFT and REDUCE operations are the same as DM and PSD.
- LEFT-EDGE$_X$ and RIGHT-EDGE$_X$ add an arc with label X between w_j and w_i. (w_j is the concept node)
- DROP pops w_j. Then push all elements in list into stack. (w_j is the surface token).
- REDUCE is performed only when w_i has head and is not the head or child of any node in buffer B, which pops w_i out of stack S.
- NODE-START$_X$ generates a new concept node with label X and set it's alignment starting from w_j. (w_j is the surface token)
- NODE-END set the alignment of w_i ending in w_j. (w_j is the surface token)
- PASS is performed when neither SHIFT nor REDUCE$_l$ can be performed, which moves w_i to the front of list .
- FINISH pops the root node and marks the state as terminal.

3.4 AMR

We extend the basic transition set to obtain the ability to generate graph nodes from the surface string, following previous work (Liu et al., 2018). There are 3 steps to parse graph nodes from the surface string in general. (a) Many concepts appear as phrases rather than single words, so we connect token spans on top of buffer to form special single tokens if needed using operation MERGE. (b) Then we use operation CONFIRM to convert a single token on buffer to a graph node(concept). In order to process entity concepts like *date-entity* better, operation ENTITY is a special form of CONFIRM which also generates property nodes of the entity concept. (c) The other concepts are not derived from surface string but previous concepts. If there is a concept node on top of buffer, operation NEW can be performed to parse this kind of concept nodes.

After solving the problem of parsing concept nodes from surface string, the basic transition set used in DM and PSD is able to predict edges between concept nodes.

- REDUCE and PASS operations are the same as DM and PSD.
- SHIFT, LEFT-EDGE$_X$ and RIGHT-EDGE$_X$ are similar to operations in DM and PSD, but they can be performed only when the top of buffer is a concept node.
- DROP operation pops the top of buffer when it is a token.
- MERGE operation connect the top two tokens in the buffer to a single token which is waiting for being converted to a concept node.
- CONFIRM$_X$ operation convert top of buffer to a concept node X if it is a token.
- ENTITY$_X$ operation does same things with CONFIRM$_X$ and then adds internal attributes of entity X, such as *year*, *month* and *day* of a *date-entity*.
- NEW$_X$ operation create a concept node labeled with X and push it to the buffer.
- FINISH pops the root node and marks the state as terminal.

4 Pre-processing and Post-processing

As discussed in 2.4, the official dataset is annotated with MRP format, while our system's input is a set of the triple (incoming arc, outgoing arc, arc label). Therefore, besides developing the transition system, we need to do: a) **Pre-processing:** Before training, we need to construct the input for our system based on MRP format graph; b) **Post-processing:** After prediction, we need to convert system's output into MRP format graph.

At first, all those framework construct triple input is basically the same, which using *directed edges*, *edge labels* and *node_id*. About *node anchor*, we directly derive the anchoring based on segmentation from companion data alignment with each sentence.[5]

While the other elements, such as *top nodes*, are a bit different among the frameworks. We will introduce these framework-specific work in the following.

4.1 DM and PSD

Node Properties Nodes in DM and PSD are labeled with lemmas and carry two additional properties that jointly determine the predicate sense, viz. pos and frame. We use two taggers to handle this problem.

Top Nodes At first, we construct an artifact node called ROOT. Then we add an edge (node, ROOT, ROOT) where the node is enumerated from top nodes.

Node Label We copy the lemmas from additional companion data and set it as node labels.

4.2 UCCA

Top Nodes There is only one top node in UCCA, which used to initialize the stack. Meanwhile, top node is the protect symbol of stack (never be popped out).

Edge Properties UCCA is the only framework with edge properties, used as a sign for remote edges. We treat remote edges the same as primary edge, except the edge label added with a special symbol, i.e. star(*).

Node Anchoring Refer to the original UCCA framework design, we link the the node in layer 0 to the surface token with edge label 'Terminal'. In post-processing, we combine surface token and layer 0 nodes via collapsing 'Terminal' edge to extract the alignment or anchor information.

4.3 EDS

Top Nodes The TOP operation will set the first concept node in buffer as top nodes.

Node Labels We train a tagger to handle this. Although there are many node labels exists, the result shows our system performs well on this.

Node Properties The only framework-specific property used on EDS nodes is carg (for constant argument), a string-valued parameter that is used with predicates(node label) like named or dofw, for proper names and the days of the week, respectively.

We write some rules to convert the surface token into properties value, such as converting million(token) to 1000000(value) when *card*(node label).

Node Anchoring We obtain alignment information through NODE_START and NODE_END operation,

4.4 AMR

Alignment There is no anchor between tokens from surface string and nodes from AMR graph. So we have to know which token aligns to which node, or we cannot train our model. Actually, finding alignment is a quite hard problem so that we could only get approximate solutions through heuristic searching. Although basic alignments have been contained in the companion data, we decide to use an enhanced rule-based aligner TAMR (Liu et al., 2018).

TAMR recalls more alignments by matching words and concepts from the view of semantic and morphological. (a) semantic match: Glove embedding represents words in some vector space. Considering a word and a concept striping off trailing number, we think them matching if their cosine similarity is small enough. (b) morphological match: Morphosemantic database in the WordNet project provides links connecting noun and verb senses, which helps match words and concepts.

Top Nodes There is exact one top node in AMR. For the convenience of processing, we add a guard element to the stack and use operation LEFT-EDGE$_{ROOT}$ between guard element and concept nodes to predict top nodes.

Node Labels Node label appears as the name of each concept which is parameter of operation ENTITY, CONFIRM and NEW.

Node Properties This is the main part of post-processing. Since our model predicts everything

[5]Organizer released pre-tokenized, PoS-tagged, and lemmatized form for training and evaluation data, besides a sequence of 'raw' sentence string. You could download the sample companion data from http://svn.nlpl.eu/mrp/2019/public/companion.tgz

System	DM	PSD	EDS	UCCA	AMR	ALL-F1
HIT-SCIR	95.08 (2)	90.55(4)	90.75(2)	**81.67 (1)**	72.94 (2)	86.2
SJTU-NICT	**95.50 (1)**	91.19 (3)	89.90 (3)	77.80 (3)	71.97 (3)	85.3
Suda-Alibaba	92.26 (7)	85.56 (8)	**91.85 (1)**	78.43 (2)	71.72 (5)	84.0
Saarland	94.69 (4)	**91.28 (1)**	89.10 (4)	67.55 (6)	66.72 (6)	81.9
Hitachi	91.02 (8)	91.21 (2)	83.74 (6)	70.36 (5)	43.86 (8)	76.0
Amazon	93.26 (6)	89.98 (5)	-	-	**73.38 (1)**	-

Table 1: The top 5 evaluation results upon cross-framework metric ALL-F1. 95.08 (2) indicates HIT-SCIR system scores 95.08 F1 in DM framework, and it ranks 2nd in DM. Amazon achieves 1st in AMR. We only list the involved results they submitted.

	DM		**PSD**		**UCCA**		**EDS**		**AMR**	
Feature	LF1	MRP	LF1	MRP	LF1	MRP	EDM	MRP	SMATCH	MRP
GloVe	87.1	87.3	74.1	73.7	56.3	87.5	82.5	88.2	64.8	65.3
BERT(base)	94.3	90.5	83.6	76.7	64.3	92.8	87.6	91.5	71.0	71.4

Table 2: HIT-SCIR parser results on MRP split dataset with GloVe or BERT as pretrained word representation. MRP stands for cross-framework evaluation metric. LF1 stands for SDP Labeled F1 (Oepen et al., 2014) in DM/PSD, UCCA Labeled Dependency F1 (Hershcovich et al., 2019) in UCCA. And EDM (Dridan and Oepen, 2011) stands for Elementary Dependency Match in EDS. SMATCH (Cai and Knight, 2013) is an evaluation metric for semantic feature structures in AMR.

as nodes and edges, we need an extra procedure to recognize which nodes should be properties in the final result. Once recognized, node along with the corresponding edge will be converted to the property of its parent node, edge label for the key, and node label for the value.

We write some rules to perform the recognizing procedure. Rules come from 2 basic facts. (a) attribute node: Numbers, URLs, and other special tokens like '-'(value of 'polarity') should be values of properties. (b) constant relation: When an edge has a label like 'value', 'quant', 'op$_x$' and so on, it is usually a key to property. We treat it as property if there is an edge of constant relation connecting to an attribute node.

5 Experiments

In this section, we will show the basic model setup including BERT fine-tuning, and results including overall evaluation, training speed. More details about training, including model selection, hyperparameters and so on, are contained in supplementary material.

5.1 Model Setup

Our work uses the AllenNLP library built for the PyTorch framework. We split parameters into two groups, i.e., BERT parameters and the other parameters (base parameters). The two parameter groups differ in learning rate. For training we use Adam (Kingma and Ba, 2015). Code for our parser and model weights are available at https://github.com/DreamerDeo/HIT-SCIR-CoNLL2019.

Fine-Tuning BERT with Parser Based on Devlin et al. (2019), fine-tuning BERT with supervised downstream task will receive the most benefit. So we choose to fine-tune BERT model together with the original parser. In our preliminary study, gradual unfreezing and slanted triangular learning rate scheduler is essential for BERT fine-tuning model. More details are discussed in supplementary material.

5.2 Results

Overall Evaluation We list the evaluation results on Table 1, which is ranked by the cross-framework metric, named ALL-F1, attached with the result of specific framework. [6] In final submission, we only use the single model for prediction. In the follow-up experiments, we get further improvement via the ensemble model. The related results is listed in supplementary material.

Training Speed To explore the effect of batch-training methods which proposed in Section 2.2

[6]Evaluation results of CoNLL 2019 shared task are available at http://bit.ly/cfmrp19.

	Parser	Feature	DM		PAS		PSD	
			id F	ood F	id F	ood F	id F	ood F
Wang et al. (2018b)	T	word2vec	89.3	83.2	91.4	87.2	76.1	73.2
Dozat and Manning (2018)	G	GloVe+char	92.7	87.8	94.0	90.6	80.5	78.6
HIT-SCIR	T	GloVe+char	86.1	79.2	89.8	85.2	72.8	68.5
AllenNLP	G	GloVe+char	91.6	86.1	93.1	89.6	77.4	73.0
HIT-SCIR	T	BERT	92.9	89.2	94.4	92.4	81.6	81.0
AllenNLP	G	BERT	94.1	90.8	94.8	92.9	80.7	79.5

Table 3: Semantic parsing accuracies (id = in domain test set; ood = out of domain test set). G and T stand for graph-based parser and transition-based parser. We adopted BERT (base+cased) model here. AllenNLP refers to the graph-based parser (Dozat and Manning, 2018) re-implemented by AllenNLP.

in training process, we conduct several experiments through adjusting the batch-size. Since we have adopted two different ways to address the cross-arc problem: list-based (DM, PSD, EDS, AMR) and SWAP operation (UCCA), we try batch-training experiments on DM and UCCA respectively. The result is shown in Figure 3. 5.3x on DM and 2.7x on UCCA speedup could be reached approximately while increasing batch size. We use GloVe pretrained embedding instead of BERT to reduce memory cost and support a larger batch size in the speed test.

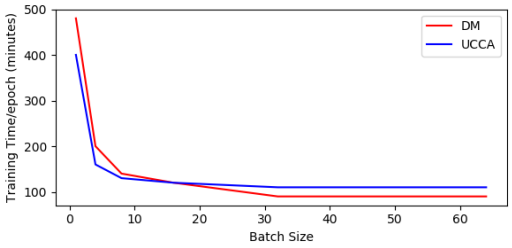

Figure 3: The training time per epoch, under different batch-size experiment setting, which indicates the efficient of batch-training methods we proposed in 2.2.

Improvement through BERT Our parser benefits a lot from BERT compared with GloVe as shown in Table 2. The improvement is more obvious in the out-of-domain evaluations, illustrating BERT's ability to transfer across domains.

6 Discussion

In recent years, graph-based parser holds the state-of-the-art in dependency parsing area due to its ability in the global decision, compared with transition-based parser. However, when we concatenated those models with BERT, we receive the similar performance, which shows that powerful representation could eliminate the gap between structure or parsing strategy.

Kulmizev et al. (2019) proposes that deep contextualized word representations are more effective at reducing errors in transition-based parsing than in graph-based parsing. Their experiments were all about dependency parsing (tree structure), and we found similar results in meaning representation parsing (graph structure), as shown in Table 3. It remains the future work to study this phenomenon with the theoretical analysis.

7 Conclusion and Future Work

Our system extends the basic transition-based parser with the following improvements: 1) adopting BERT for better word representation; 2) realizing batch-training for stack LSTM to speed up the training process. And we proposed a unified pipeline for meaning representation parsing, suitable for main stream graphbanks. In the final evaluation, we were ranked first place in CoNLL 2019 shared task according to ALL-F1 (86.2%) and especially ranked first in UCCA framework (81.67%).

Acknowledgments

We thank the reviewers for their insightful comments and the HIT-SCIR colleagues for the coordination on the machine usage. This work was supported by the National Natural Science Foundation of China (NSFC) via grant 61976072, 61632011 and 61772153.

References

Omri Abend and Ari Rappoport. 2013. Universal conceptual cognitive annotation (UCCA). In *ACL*.

Omri Abend and Ari Rappoport. 2017. The state of the art in semantic representation. In *ACL*.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *LAWID*.

Jan Buys and Phil Blunsom. 2017. Robust incremental neural semantic graph parsing. In *ACL*.

Shu Cai and Kevin Knight. 2013. Smatch: an evaluation metric for semantic feature structures. In *ACL*.

Wanxiang Che, Yijia Liu, Yuxuan Wang, Bo Zheng, and Ting Liu. 2018. Towards better UD parsing: Deep contextualized word embeddings, ensemble, and treebank concatenation. In *CoNLL*.

Jinho D. Choi and Andrew McCallum. 2013. Transition-based dependency parsing with selectional branching. In *ACL*.

Ann Copestake, Dan Flickinger, Ivan A. Sag, and Carl Pollard. 1999. Minimal recursion semantics: an introduction.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*.

Timothy Dozat and Christopher D. Manning. 2018. Simpler but more accurate semantic dependency parsing. In *ACL*.

Rebecca Dridan and Stephan Oepen. 2011. Parser evaluation using elementary dependency matching. In *ACL*.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *ACL and IJCNLP*.

Dan Flickinger, Jan Hajič, Angelina Ivanova, Marco Kuhlmann, Yusuke Miyao, Stephan Oepen, and Daniel Zeman. 2017. Open SDP 1.2.

Jonas Groschwitz, Meaghan Fowlie, Mark Johnson, and Alexander Koller. 2017. A constrained graph algebra for semantic parsing with AMRs. In *IWCS*.

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2017. A transition-based directed acyclic graph parser for UCCA. In *ACL*.

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2018. Multitask parsing across semantic representations. In *ACL*.

Daniel Hershcovich, Zohar Aizenbud, Leshem Choshen, Elior Sulem, Ari Rappoport, and Omri Abend. 2019. SemEval-2019 task 1: Cross-lingual semantic parsing with UCCA. In *IWSE*.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *TACL*.

Alexander Koller, Stephan Oepen, and Weiwei Sun. 2019. Graph-based meaning representations: Design and processing. In *ACL*.

Dan Kondratyuk and Milan Straka. 2019. 75 languages, 1 model: Parsing universal dependencies universally. In *EMNLP*.

Marco Kuhlmann and Stephan Oepen. 2016. Squibs: Towards a catalogue of linguistic graph Banks. *Computational Linguistics*.

Artur Kulmizev, Miryam de Lhoneux, Johannes Gontrum, Elena Fano, and Joakim Nivre. 2019. Deep contextualized word embeddings in transition-based and graph-based dependency parsing – a tale of two parsers revisited. In *EMNLP*.

Matthias Lindemann, Jonas Groschwitz, and Alexander Koller. 2019. Compositional semantic parsing across graphbanks. In *ACL*.

Yijia Liu, Wanxiang Che, Bo Zheng, Bing Qin, and Ting Liu. 2018. An AMR aligner tuned by transition-based parser. In *EMNLP*.

Wolfgang Maier. 2015. Discontinuous incremental shift-reduce parsing. In *ACL and IJCNLP*.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *ICLR*.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *ICPT*.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*.

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *ACL and AFNLP*.

Stephan Oepen, Omri Abend, Jan Hajič, Daniel Hershcovich, Marco Kuhlmann, Tim O'Gorman, Nianwen Xue, and Milan Straka. 2019. MRP 2019: Cross-framework Meaning Representation Parsing. In *CoNLL*.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinková, Dan Flickinger, Jan Hajič, and Zdeňka Urešová. 2015. SemEval 2015 task 18: Broad-coverage semantic dependency parsing. In *SemEval*.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Dan Flickinger, Jan Hajič, Angelina Ivanova, and Yi Zhang. 2014. SemEval 2014 task 8: Broad-coverage semantic dependency parsing. In *SemEval*.

Hao Peng, Sam Thomson, and Noah A. Smith. 2017. Deep multitask learning for semantic dependency parsing. In *ACL*.

Hao Peng, Sam Thomson, Swabha Swayamdipta, and Noah A. Smith. 2018. Learning joint semantic parsers from disjoint data. In *NAACL-HLT*.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *EMNLP*.

Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *NAACL-HLT*.

Kenji Sagae and Jun'ichi Tsujii. 2008. Shift-reduce dependency DAG parsing. In *Coling*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2018a. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *EMNLP*.

Yuxuan Wang, Wanxiang Che, Jiang Guo, and Ting Liu. 2018b. A neural transition-based approach for semantic dependency graph parsing. In *AAAI*.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, ukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*.

Daniel Zeman, Jan Hajič, Martin Popel, Martin Potthast, Milan Straka, Filip Ginter, Joakim Nivre, and Slav Petrov. 2018. CoNLL 2018 shared task: Multilingual parsing from raw text to universal dependencies. In *CoNLL*.