AMR Parsing using Stack-LSTMs

Abstract

We present a transition-based AMR parser that directly generates AMR parses from plain text. We use Stack-LSTMs to represent our parser state and make decisions greedily. In our experiments, we show that our parser achieves very competitive scores on English using only AMR training data. Adding additional information, such as POS tags and dependency trees, improves the results further.

1 Introduction

Transition-based algorithms for natural language parsing are formulated as a series of decisions that read words from a buffer and incrementally combine them to form syntactic structures in a stack. Apart from dependency parsing, these models, also known as shift-reduce algorithms, have been successfully applied to tasks like phrase-structure parsing, named entity recognition, CCG parsing, joint syntactic and semantic parsing and even abstract- meaning representation parsing.

AMR parsing requires solving several natural language processing tasks; mainly named entity recognition, word sense disambiguation and joint syntactic and semantic role labeling. Given the difficulty of building an end-to-end system, most prior work is based on pipelines or heavily dependent on precalculated features.

Inspired by we present a shift- reduce algorithm that produces AMR graphs directly from plain text. presented transition-based tree-to-graph transducers that traverse a dependency tree and transforms it to an AMR graph. input is a sentence and it is therefore more similar (with a different parsing algorithm) to our approach, but their parser relies on external tools, such as dependency parsing, semantic role labeling or named entity recognition.

The input of our parser is plain text sentences and, through rich word representations, it predicts all actions (in a single algorithm) needed to generate an AMR graph representation for an input sentence; it handles the detection and annotation of named entities, word sense disambiguation and it makes connections between the nodes detected towards building a predicate argument structure. Even though the system that runs with just words is very competitive, we further improve the results incorporating POS tags and dependency trees into our model.

Stack-LSTMs have proven to be useful in tasks related to syntactic and semantic parsing and named entity recognition. In this paper, we demonstrate that they can be effectively used for AMR parsing as well.

2 Parsing Algorithm

Our parsing algorithm makes use of a STACK (that stores AMR nodes and/or words) and a BUFFER that contains the words that have yet to be processed. The parsing algorithm is inspired from the semantic actions presented by , the transition-based NER algorithm by and the arc-standard algorithm. As in the buffer starts with the root symbol at the end of the sequence. Figure 2 shows a running example. The transition inventory is the following:

- SHIFT: pops the front of the BUFFER and push it to the STACK.

- CONFIRM: calls a subroutine that predicts the AMR node corresponding to the top of the STACK. It then pops the word from the STACK and pushes the AMR node to the STACK. An example is the prediction of a propbank sense: From occurred to occur-01.
- REDUCE: pops the top of the STACK. It occurs when the word/node at the top of the stack is complete (no more actions can be applied to it). Note that it can also be applied to words that do not appear in the final output graph, and thus they are directly discarded.
- MERGE: pops the two nodes at the top of the STACK and then it merges them, it then pushes the resulting node to the top of STACK. Note that this can be applied recursively. This action serves to get multiword named entities (e.g. New York City).
- ENTITY(label): labels the node at the top of the STACK with an entity label. This action serves to label named entities, such as New York City or Madrid and it is normally run after MERGE when it is a multi-word named entity, or after SHIFT if it is a single-word named entity.
- DEPENDENT(label,node): creates a new node in the AMR graph that is dependent on the node at the top of the STACK. An example is the introduction of a negative polarity to a given node: From illegal to (legal, polarity -).
- LA(label) and RA(label): create a left/right arc with the top two nodes at the top of the

STACK. They keep both the head and the dependent in the stack to allow reentrancies (multiple incoming edges). The head is now a composition of the head and the dependent. They are enriched with the AMR label.

- SWAP: pops the two top items at the top of the STACK, pushes the second node to the front of the BUFFER, and pushes the first one back into the STACK. This action allows non-projective arcs as in but it also helps to introduce reentrancies. At oracle time, SWAP is produced when the word at the top of the stack is blocking actions that may happen between the second element at the top of the stack and any of the words in the buffer.

Figure 1 shows the parser actions and the effect on the parser state (contents of the stack, buffer) and how the graph is changed after applying the actions.

We implemented an oracle that produces the sequence of actions that leads to the gold (or close to gold) AMR graph. In order to map words in the sentences to nodes in the AMR graph we need to align them. We use the JAMR aligner provided by. It is important to mention that even though the aligner is quite accurate, it is not perfect, producing a F1 score of around 0.90. This means that most sentences have at least one alignment error which implies that our oracle is not capable of perfectly reproducing all AMR graphs. This has a direct impact on the accuracy of the parser described in the next section since it is trained on sequences of actions that are not perfect. The oracle achieves 0.895 F1 Smatch score when it is run on the development set of the LDC2014T12.

The algorithm allows a set of different constraints that varies from the basic ones (not allowing impossible actions such as SHIFT when the buffer is empty or not generating arcs when the words have not yet been CONFIRMed and thus transformed to nodes) to more complicated ones based on the propbank candidates and number of arguments. We choose to constrain the parser to the basic ones and let it learn the more complicated ones.

(r / recommend-01 :ARG1 (a / advocate-01 :ARG1 (i / it) :manner (v / vigorous)))

3 Parsing Model

In this section, we revisit Stack-LSTMs, our parsing model and our word representations.

3.1 Stack-LSTMs

The stack LSTM is an augmented LSTM that allows adding new inputs in the same way as LSTMs but it also provides a POP operation that moves a pointer to the previous element. The output vector of the LSTM will consider the stack pointer instead of the rightmost position of the sequence.

Stack$t$	Buffer$t$	Action	Stack$t+1$	Buffer$t+1$	Graph
u, S	B	SHIFT	u, S	B	–
u, S	B	CONFIRM	n, S	B	–
u, S	B	REDUCE	S	B	–
u, v, S	B	MERGE	(u, v), S	B	–
u, S	B	ENTITY(l)	(u : l), S	B	–
u, S	B	DEPENDENT(r, d)	u, S	B	r $\rightarrow$ d
u, v, S	B	RA(r)	u, v, S	B	r $\rightarrow$ v
u, v, S	B	LA(r)	u, v, S	B	r $\leftarrow$ v
u, v, S	B	SWAP	u, S	v, B	–

Table 1: Parser transitions indicating the action applied to the stack and buffer and the resulting state.

ACTION	STACK	BUFFER
INIT		It, should, be, vigorously, advocated, R
SHIFT	it	should, be, vigorously, advocated, R
CONFIRM	it	should, be, vigorously, advocated, R
SHIFT	should, it	be, vigorously, advocated, R
CONFIRM	recommend-01, it	be, vigorously, advocated, , R
SWAP	recommend-01	it, be, vigorously, advocated, R
SHIFT	it, recommend-01	be, vigorously, advocated, R
REDUCE	recommend-01	be, vigorously, advocated, R
SHIFT	be, it, recommend-01	vigorously, advocated, R
REDUCE	it, recommend-01	vigorously, advocated, R
SHIFT	vigorously, it, recommend-01	advocated, R
CONFIRM	vigorous, it, recommend-01	advocated, R
SWAP	vigorous, recommend-01	it, advocated, R
SWAP	vigorous	recommend-01, it, advocated, R
SHIFT	vigorous	recommend-01, advocated, R
SHIFT	vigorous, recommend-01	advocated, R
SHIFT	it, vigorous	recommend-01, advocated, R
CONFIRM	advocate-01, it, recommend-01, vigorous	R
LA(ARG1)	advocate-01, it, recommend-01, vigorous	R
SWAP	advocate-01, recommend-01, vigorous	it R
SHIFT	it, advocate-01, recommend-01, vigorous	R
REDUCE	advocate-01, recommend-01, vigorous	R
RA(ARG1)	advocate-01, recommend-01, vigorous	R
SWAP	advocate-01, vigorous	recommend-01, R
SHIFT	recommend01, advocate-01, vigorous	R
SHIFT	R, recommend01, advocate-01, vigorous	
LA(root)	R, recommend01, advocate-01, vigorous	
REDUCE	recommend01, advocate-01, vigorous	
REDUCE	advocate-01, vigorous	
REDUCE	vigorous	
REDUCE		

Table 2: Transition sequence for the sentence It should be vigorously advocated. R represents the root symbol

3.2 Representing the State and Making Parsing Decisions

The state of the algorithm presented in Section 2 is represented by the contents of the STACK, BUFFER and a list with the history of actions (which are encoded as Stack-LSTMs). All of this forms the vector s_t that represents the state which s calculated as follows:

$\mathrm{s}_t = \max\{0, W[s_t^t; b_t; a_t] + d\}$,

where W is a learned parameter matrix, d is a bias term and s_t^t, b_t,a_t represent the output vector of the Stack-LSTMs at time t.

Predicting the Actions: Our model then uses the vector s_t for each timestep t to compute the probability of the next action as:

$\mathrm{p(z|s}_t) = \frac{exp(g_z.s_t+q_z)}{\sum_{z' \in A} exp(g'_z.s_t+q'_z)}$,

where g_z is a column vector representing the (output) embedding of the action z, and q_z is a bias term for action z. The set A represents the actions listed in Section 2. Note that due to parsing constraints the set of possible actions may vary. The total number of actions (in the LDC2014T12 dataset) is 478; note that they include all possible labels (in the case of LA and RA) and the different dependent nodes for the DEPENDENT action.

Predicting the Nodes: When the model selects the action CONFIRM, the model needs to decide the AMR node that corresponds to the word at the top of the STACK, by using s_t, as follows:

$\mathrm{p(e|s}_t) = \frac{exp(g_e.s_t+q_e)}{\sum_{e' \in N} exp(g_{e'}.s_t+q_{e'})}$,

where N is the set of possible candidate nodes for the word at the top of the STACK. g_e is a column vector representing the (output) embedding of the node e, and q_e is a bias term for the node e. It is important to mention that this implies finding a propbank sense or a lemma. For that, we rely entirely on the AMR training set instead of using additional resources.

Given that the system runs two softmax operations, one to predict the action to take and the second one to predict the corresponding AMR node, and they both share LSTMs to make predictions, we include an additional layer with a tanh nonlinearity after s_t for each softmax.

3.3 Word Representations

We use character-based representations of words using bidirectional LSTMs . They learn representations for words that are orthographically similar. Note that they are updated with the updates to the model. demonstrated that it is possible to achieve high results in syntactic parsing and named entity recognition by just using character-based word representations (not even POS tags, in fact, in some cases the results with just character-based representations outperform those that used explicit POS tags since they provide similar vectors for words with similar/same morphosyntactic tag); in this paper we show a similar result given that both syntactic parsing and named-entity recognition play a central role in AMR parsing.

These are concatenated with pretrained word embeddings. We use a variant of the skip n-gram model with the LDC English Gigaword corpus (version 5). These embeddings encode the syntactic behavior of the words .

More formally, to represent each input token, we concatenate two vectors: a learned character-based representation ($\hat{w}^C$); and a fixed vector representation from a neural language model ($\hat{w}^{LM}$). A linear map (V) is applied to the resulting vector and passed through a component-wise ReLU,

$\mathrm{x} = \max\{0, V[\hat{w}^C; \hat{w}^{LM}] + b\}$.

where V is a learned parameter matrix, b is a bias term and w^C is the character-based learned representation for each word, $\hat{w}^{LM}$ is the pretrained word representation.

3.4 POS Tagging and Dependency Parsing

We may include preprocessed POS tags or dependency parses to incorporate more information into our model. For the POS tags we use the Stanford tagger while we use the Stack-LSTM parser trained on the English CoNLL 2009 dataset to get the dependencies.

Model	F1(Newswire)	F1(ALL)
(POS, DEP)	0.59	0.58
(POS, DEP, NER)	-	0.66
(POS, DEP, NER)	0.62	-
(POS, DEP, NER, SRL)	-	0.61
(POS, DEP, NER, SRL)	-	0.64
(POS, CCG)	0.66	-
(POS, DEP, NER)	0.70	-
(POS, DEP, NER, SRL)	0.71	0.66
(LM, NER)	-	0.61
(Wordnet, LM, NER)	-	0.66
(POS, DEP, NER)	0.63	0.59
(POS, DEP, NER, SRL)	0.70	0.66
OUR PARSER (NO PRETRAINED-NO CHARS)	0.64	0.59
OUR PARSER (NO PRETRAINED-WITH CHARS)	0.66	0.61
OUR PARSER (WITH PRETRAINED-NO CHARS)	0.66	0.62
OUR PARSER	0.68	0.63
OUR PARSER (POS)	0.68	0.63
OUR PARSER (POS, DEP)	0.69	0.64

Table 3: AMR results on the LDC2014T12 dataset; Newsire section (left) and full (right). Rows labeled with OUR-PARSER show our results. POS indicates that the system uses preprocessed POS tags, DEP indicates that it uses preprocessed dependency trees, SRL indicates that it uses preprocessed semantic roles, NER indicates that it uses preprocessed named entitites. LM indicates that it uses a LM trained on AMR data and WordNet indicates that it uses WordNet to predict the concepts. Systems marked with * are pipeline systems that require a dependency parse as input. (WITH PRETRAINED-NO CHARS) shows the results of our parser without character-based representations. (NO PRETRAINED-WITH CHARS) shows results without pretrained word embeddings. (NO PRETRAINED-NO CHARS) shows results without character-based representations and without pretrained word embeddings. The rest of our results include both pretrained embeddings and character-based representations.

POS tags: The POS tags are preprocessed and a learned representation tag is concatenated with the word representations. This is the same setting as .

Dependency Trees: We use them in the same way as POS tags by concatenating a learned representation dep of the dependency label to the parent with the word representation. Additionally, we enrich the state representation s_t, presented in Section 3.2. If the two words at the top of the STACK have a dependency between them, s_t is enriched with a learned representation that indicates that and the direction; otherwise s_t remains unchanged. s_t is calculated as follows:

$$\mathrm{s}_t = \max\{0, W[s_t^t; b_t; a_t; dep_t] + d\},$$

where dep_t is the learned vector that represents that there is an arc between the two top words at the top of the stack.

4 Experiments and Results

We use the LDC2014T12 dataset for our experiments. Table 1 shows results, including comparison with prior work that are also evaluated on the same dataset.

Our model achieves 0.68 F1 in the newswire section of the test set just by using character-based representations of words and pretrained word embeddings. All prior work uses lemmatizers, POS taggers, dependency parsers, named entity recognizers and semantic role labelers that use additional training data while we achieve competitive scores without that. reports 0.66 F1 in the full test by using WordNet for concept identification, but their performance drops to 0.61 without WordNet. It is worth noting that we achieved 0.64 in the same test set without WordNet. without SRL (via Propbank) achieves only 0.63 in the newswire test set while we achieved 0.69 without SRL (and 0.68 without dependency trees).

In order to see whether pretrained word embeddings and character-based embeddings are useful we carried out an ablation study by showing the results of our parser with and without character-based representations (replaced by standard lookup table learned embeddings) and with and without pretrained word embeddings. By looking at the results of the parser without character-based embeddings but with pretrained word embeddings we observe that the character- based representation of words are useful since they help to achieve 2 points better in the Newswire dataset and 1 point more in the full test set. The parser with character-based embeddings but without pretrained word embeddings, the parser has more difficulty to learn and only achieves 0.61 in the full test set. Finally, the model that does not use neither character-based embeddings nor pretrained word embeddings is the worst achieving only 0.59 in the full test set, note that this model has no explicity way of getting any syntactic information through the word embeddings nor a smart way to handle out of vocabulary words.

All the systems marked with * require that the input is a dependency tree, which means that they solve a transduction task between a dependency tree and an AMR graph. Even though our parser starts from plain text sentences when we incorporate more information into our model, we achieve further improvements. POS tags provide small improvements (0.6801 without POS tags vs 0.6822 for the model that runs with POS tags). Dependency trees help a bit more achieving 0.6920.

5 Conclusions and Future Work

We present a new transition-based algorithm for AMR parsing and we implement it using Stack-LSTMS and a greedy decoder. We present competitive results, without any additional resources and external tools. Just by looking at the words, we achieve 0.68 F1 (and 0.69 by preprocessing dependency trees) in the standard dataset used for evaluation.