Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision

Anonymized for review

Abstract

Extending the success of deep neural networks to high level tasks like natural language understanding and symbolic reasoning requires program induction and learning with weak supervision. Recent neural program induction approaches have either used primitive computation component like Turing machine or differentiable operations and memory trained by backpropagation. In this work, we propose the Manager-Programmer-Computer framework to integrate neural networks with operations and memory that are *abstract, scalable and precise* but non-differentiable, and a friendly *neural computer interface*. Specifically, we introduce the Neural Symbolic Machines. It contains a sequence-to-sequence neural "programmer" that takes in natural language input and outputs a program as a sequence of tokens, and a non-differentiable "computer" that is a Lisp interpreter with code assistance using syntax check and denotations of partial programs. This integration enables the model to effectively learn a semantic parser from weak supervision over a large knowledge base. Our model obtained new state-of-the-art performance on WEBQUESTIONSSP, a challenging semantic parsing dataset.

1 Introduction

Deep neural networks have achieved impressive performance in classification and structured prediction tasks with full supervision like speech recognition and machine translation. Extending the success to high level tasks like natural language understanding and symbolic reasoning requires the ability to perform program induction and learn with weak supervision.

There has been a lot of recent progress in neural program induction [4, 7, 9, 6, 13], which learns programs by using a neural sequence model to control a computation component. However, the computation components used by current models are either primitive like Turing machine or uses differentiable operations and memory so that it can be trained by backpropagation. This makes it difficult to utilize powerful operations, e.g. querying a large knowledge base, and efficient discrete memory similar to a real computer, thus limits the current applications to only synthetic tasks.

To better utilize powerful computation components, we propose a Manager-Programmer-Computer framework for neural program induction, which integrates three components:

1. A **"manager"** that provides weak supervision through input and a reward signal indicating how well a task is performed. Unlike full supervision, this weak supervision is much easier to obtain at large scale.
2. A **"programmer"** that takes natural language as input and generates a program that is a sequence of tokens. The programmer learns from the reward signal and must overcome the hard search problem of finding good programs. (See section 2.2 for an example).
3. A **"computer"** that executes the program. It can use all the operations / functions that can be implemented in a high level programming language like Lisp. These operations are *abstract, scalable and precise*, which is hard to achieve by neural networks. But they are also not differentiable, thus require reinforcement learning. It also provides a friendly

neural computer interface to help the programmer reduce the search space by detecting and eliminating invalid choices. (See section 2.1 for an example).

Within the MPC framework, we implemented the Neural Symbolic Machine (NSM) and applied it to semantic parsing. NSM contains a sequence-to-sequence neural network model ("programmer") augmented with a key-value memory to save and reuse intermediate results for compositionality, and a non-differentiable Lisp interpreter ("computer") that executes programs against a large knowledge base and helps reduce the search space using syntax check and the denotations of partial programs. To efficiently train NSMs with weak supervision, we adopt a hybrid training procedure by mixing maximum likelihood training and REINFORCE training.

Compared to existing neural program induction approaches, the powerful operations and friendly interface of the "computer" greatly reduce the burden of the "programmer" and enable the model to perform competitively on real applications. On the challenging semantic parsing dataset WEBQUESTIONSSP [12], NSMs obtain new state-of-the-art results with weak supervision. Compared to previous work, it does not require any feature engineering or domain-specific knowledge.

2 Neural Symbolic Machines

Now we describe in details a Neural Symbolic Machine that falls into the MPC framework, and how it is applied to learn semantic parsing from weak supervision.

Semantic parsing is defined as follows: given a knowledge base (KB) $\mathbb{K}$, and a question $q = (w_1, w_2, ..., w_k)$, produce a program or logical form z that when executed against $\mathbb{K}$ generates the right answer y. Let $\mathcal{E}$ denote a set of entities (e.g., ABELINCOLN)[1], and let $\mathcal{P}$ denote a set of properties (or relations, e.g., PLACEOFBIRTHOF). A knowledge base $\mathbb{K}$ is a set of assertions $(e_1, p, e_2) \in \mathcal{E} \times \mathcal{P} \times \mathcal{E}$, such as (HODGENVILLE, PLACEOFBIRTHOF, ABELINCOLN)).

2.1 "Computer": Lisp interpreter with code assistance

We adopt the Lisp programming language with predefined functions listed in 1 as our semantic representation, which is equivalent to the limited subset of λ-calculus in [11] but easier for a sequence-to-sequence model to generate. A program C is a list of expressions $(c_1 ... c_L)$. Each expression is either a special token "RETURN" indicating the end of the program, or a list of tokens enclosed by parentheses "(F A_0 ... A_K)", where A_k is an argument, and can be either a relation $p \in \mathcal{P}$ or a variable v that represents a set of entities or values obtained by executing a previous expression or an entity resolved from the natural language input. F is one of the functions in Table 1, which take as input a list of arguments of specific types, and, when executed, returns a new variable representing the denotation of this expression in $\mathbb{K}$.

(*Hop* v p) $\Rightarrow \{e_2 \| e_1 \in v, (e_1, p, e_2) \in \mathbb{K}\}$
(*ArgMax* v p) $\Rightarrow \{e_1 \| e_1 \in v, \exists e_2 \in \mathcal{E} : (e_1, p, e_2) \in \mathbb{K}, \forall e : (e_1, p, e) \in \mathbb{K}, e_2 \geq e\}$
(*ArgMin* v p) $\Rightarrow \{e_1 \| e_1 \in v, \exists e_2 \in \mathcal{E} : (e_1, p, e_2) \in \mathbb{K}, \forall e : (e_1, p, e) \in \mathbb{K}, e_2 \leq e\}$
(*Equal* v_1 v_2 p) $\Rightarrow \{e_1 \| e_1 \in v_1, \exists e_2 \in v_2 : (e_1, p, e_2) \in \mathbb{K}\}$

Table 1: Predefined functions. v represents a variable. p represents a predicate in Freebase.

To provide a better *neural computer interface*, the interpreter provides code assistance by producing a list of syntactically valid tokens for the programmer to pick from at each step. For example, if the previous generated token is "(", the next token must be a function, and if the previous token is *Hop*, the next token must be a variable. More importantly, the interpreter also uses the denotation of partially executed programs to restrict the choices at each step. For example, given that the previously generated tokens are "(", "*Hop*", "v", the next available token is restricted to the set of predicates $\{p | e \in v, \exists e' : (e, p, e') \in \mathbb{K}\}$ that are reachable from entities in v. By providing this "code assistance", the interpreter reduces the search space by orders of magnitude, and enables weakly supervised training on a large knowledge base.

[1]We also consider numbers (e.g., “1.33”) and date-times (e.g., “1999-1-1”) as entities.

2.2 "Programmer": key-value memory augmented Seq2Seq model

Similar to other neural program induction approaches, we use a sequence-to-sequence model for the neural "programmer", and we augment it with a key-value memory to save and reuse the intermediate results. We will go over each part below.

A typical sequence-to-sequence model consists of two RNNs, an encoder and a decoder. We used a 1-layer GRU for both the encoder and decoder. Given a sequence of words $w_1, w_2 ... w_m$, each word w_t is mapped to a multi-dimensional embedding q_t (details in Section 3). Then, the encoder reads in these embeddings and updates its hidden state step by step using: $h_{t+1} = GRU(h_t, q_t, \theta_{Encoder})$, where $\theta_{Encoder}$ are the GRU parameters. The decoder updates its hidden states u_t by $u_{t+1} = GRU(u_t, c_{t-1}, \theta_{Decoder})$, where c_{t-1} is the embedding of last step's output token a_{t-1}, and $\theta_{Decoder}$ are the GRU parameters. The last hidden state of the encoder h_T is used as the decoder's initial state. We adopt a dot-product attention similar to that of [3]. The tokens of the program $a_1, a_2 ... a_n$ are generated one by one using the attention context vector, and a softmax over the vocabulary available for each step (details in Section 2.1).

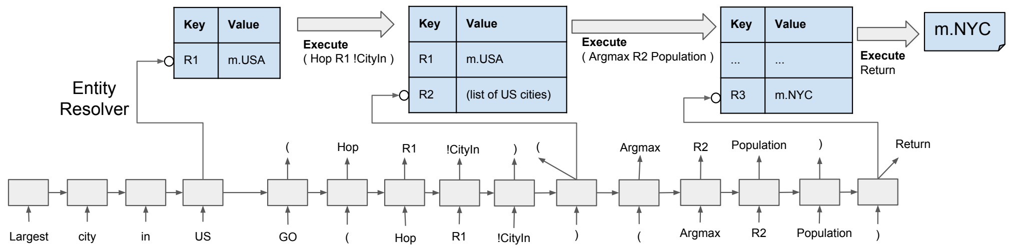

Figure 1: Semantic Parsing with NSM. The keys of the key-value memory table are the output of sequence model at certain encoding or decoding steps. For illustration purpose we show the denotations of values in the key-value tables. However, the sequence model never see these denotations, and the output tokens are just variable symbols such as "R1". A special token "GO" indicates the start of decoding, and "RETURN" indicates the end of decoding.

To achieve compositionality, we augment the model with a **key-value memory** (Figure 1). It enables the model to save the result of executing an expression and reuse it in later expressions. Each entry in the key-value memory has two components: a continuous multi-dimensional key embedding, and a corresponding symbolic value (e.g., "R1"), which represents an intermediate result in the computer, and has the form of a list of entities in the knowledge base. Note that although the key embeddings are continuous and differentiable, the value is symbolic and non-differentiable, which makes it different from other memory-augmented neural networks [10]. During encoding if a token (*"US"*) is the last token of a resolved entity (by an entity resolver), then the resolved entity id (M.USA) is saved in a new variable in the memory, and the key for this variable is the average GRU output of the tokens spanned by this entity. During decoding if an expression is completely finished (the decoder reads in ")") then it gets executed, and the result is added as a new variable in the memory. This variable is keyed by the GRU output of that step. Every time a new variable is pushed into the memory, the variable token is added to the vocabulary of the decoder. The answer returned by the programmer is the value of the last variable produced by the programmer.

2.3 Training NSM with Weak Supervision

Similar to [13], NSM uses non-differentiable operations for which back-propagation cannot be applied. So we use REINFORCE for effective training. While [13] rely on curriculum learning to manage the challenges of training with REINFORCE, we adopt a hybrid approach, which combines REINFORCE with maximum likelihood training by taking the best program (the program that achieves the highest F1 score with the shortest length) found by the "programmer" for each question as a pseudo target. The details of the hybrid training is not central to our contribution, and will be omitted here due to space limit.

3 Experiments and analysis

3.1 The WebQuestions Task

Modern semantic parsers [2], which map natural language utterances to executable logical forms, have been successfully trained over large knowledge bases from weak supervision[11], but require substantial feature engineering. Recent attempts to train an end-to-end neural network for semantic parsing [3, 5] have either used strong supervision (full logical forms), or have employed synthetic datasets.

We apply NSM to learn a semantic parser with weak supervision and no manual engineering. we used the challenging semantic parsing dataset WEBQUESTIONSSP [12], which consists of 3,098 question-answer pairs for training and 1,639 for testing. These questions were collected using Google Suggest API and the answers were originally obtained [1] using Amazon Mechanical Turk and updated by annotators who are familiar with the design of Freebase [12]. We further separate out 620 questions in the training set as validation set. For query pre-prosessing we used an in-house named entity linking system to find the entities in a question. The quality of the entity resolution is similar to that of [11] with about 94% of the gold root entities being included in the resolution results. Similar to [3], we also replaced named entity tokens with a special token "ENT". For example, the question *"who plays meg in family guy"* is changed to *"who plays ENT in ENT ENT"*.

Following [11] we use the last public available snapshot of Freebase KB. Since NSM training requires random access to Freebase during decoding, we preprocess Freebase by removing predicates that are not related to world knowledge (starting with "/common/", "/type/", "/freebase/")[2], and removing all text valued predicates, which are rarely the answer. This results in a graph with 23K relations, 82M nodes, and 417M edges.

3.2 Model details

The dimension of encoder hidden state, decoder hidden state and key embeddings are all 50. The embeddings for the functions and special tokens (e.g., "UNK", "GO") are randomly initialized by a truncated normal distribution with mean=0.0 and stddev=0.1. All the weight matrices are initialized with a uniform distribution in $[-\frac{\sqrt{3}}{d}, \frac{\sqrt{3}}{d}]$ where d is the input dimension.

For pretrained word embeddings, we used the 300 dimension GloVe word embeddings trained on 840B common crawl corpus [8]. On the encoder side, we added a projection matrix to transform the pretrained embeddings into 50 dimension. On the decoder side, we used the same GloVe word embeddings to construct the relation embeddings from words in the relation names.

3.3 Result and Discussion

We evaluate performance using the offical measures for WEBQUESTIONSSP. Because the answer to a question can contain multiple entities or values, precision, recall and F1 are computed based on the output for each individual question. The average F1 score is reported as the main evaluation metric. The accuracy@1 measures the percentage of questions that are answered exactly. The comparison with previous state-of-the-art [12, 11] is shown in Table 2. Besides the better performance, our model does not rely on domain-specific rules or feature engineering.

Model	Avg. Prec.	Avg. Rec.	Avg. F1	Acc.
STAGG	67.3	73.1	66.8	58.8
NSM – our model	69.5	76.6	**68.3**	57.4

Table 2: Comparison to previous state-of-the-art, average F1 is the main evaluation metric. Our model achieves better results without hand-crafted rules and feature engineering.

The improvement is mainly because the STAGG model [11] only allow 2 hops if the entity in the middle is a CVT, while we search all possible traces within 2 hops. We can effectively explore this much larger space because our search is guided by the neural "programmer" with the help of the *neural computer interface*.

[2]Except that we kept “/common/topic/notable_types”.

References

[1] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *EMNLP*, volume 2, page 6, 2013.

[2] Jonathan Berant and Percy Liang. Semantic parsing via paraphrasing. In *ACL (1)*, pages 1415–1425, 2014.

[3] Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Association for Computational Linguistics (ACL)*, 2016.

[4] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[5] Robin Jia and Percy Liang. Data recombination for neural semantic parsing. In *Association for Computational Linguistics (ACL)*, 2016.

[6] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[7] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834, 2015.

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

[9] Scott Reed and Nando de Freitas. Neural programmer-interpreters. In *ICLR*, 2016.

[10] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

[11] Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. Semantic parsing via staged query graph generation: Question answering with knowledge base. In *Association for Computational Linguistics (ACL)*, 2015.

[12] Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. The value of semantic parse labeling for knowledge base question answering. In *Association for Computational Linguistics (ACL)*, 2016.

[13] Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 2015.