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

Chen Liang[*], **Jonathan Berant**[†], **Quoc Le, Kenneth D. Forbus, Ni Lao**

Northwestern University, Evanston, IL
{chenliang2013,forbus}@u.northwestern.edu
Tel-Aviv University, Tel Aviv-Yafo, Israel
joberant@cs.tau.ac.il
Google Inc., Mountain View, CA
{qvl,nlao}@google.com

Abstract

Extending the success of deep neural networks to natural language understanding and symbolic reasoning requires complex operations and external memory. Recent neural program induction approaches have attempted to address this problem, but are typically limited to differentiable memory, and consequently cannot scale beyond small synthetic tasks. In this work, we propose the Manager-Programmer-Computer framework, which integrates neural networks with *non-differentiable* memory to support *abstract*, *scalable* and *precise* operations through a friendly *neural computer interface*. Specifically, we introduce a Neural Symbolic Machine, which contains a sequence-to-sequence neural "programmer", and a non-differentiable "computer" that is a Lisp interpreter with code assist. To successfully apply REINFORCE for training, we augment it with approximate gold programs found by an iterative maximum likelihood training process. NSM is able to learn a semantic parser from weak supervision over a large knowledge base. It achieves new state-of-the-art performance on WEBQUESTIONSSP, a challenging semantic parsing dataset, with weak supervision. Compared to previous approaches, NSM is end-to-end, therefore does not rely on feature engineering or domain specific knowledge.

1 Introduction

Deep neural networks have achieved impressive performance in classification and structured prediction tasks with full supervision such as speech recognition [**?**] and machine translation [**? ? ?**]. Extending the success to natural language understanding and symbolic reasoning requires the ability to perform complex operations and make use of an external memory.

There were several recent attempts to address this problem in neural program induction [**? ? ? ? ? ? ?**], which learn programs by using a neural sequence model to control a computation component. However, the memories in these models are either low-level (such as in Neural Turing machines[**?**]), or required to be differentiable so that they can be trained by backpropagation. This makes it difficult to utilize efficient discrete memory in a traditional computer, and limits their application to small synthetic tasks.

To better utilize efficient memory and operations, we propose a Manager-Programmer-Computer (MPC) framework for neural program induction, which integrates three components (Figure 1):

[*]Work done while the author was interning at Google
[†]Work done while the author was a visiting scholar at Google

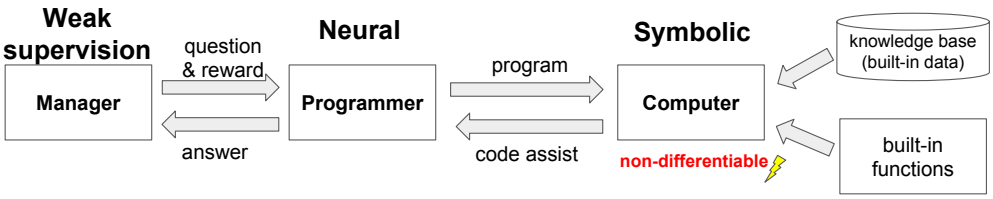

Figure 1: Manager-Programmer-Computer framework

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 (see an example task in Section 3.1).
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. (Section 2.2).
3. A **"computer"** that executes the program. It can use all the operations that can be implemented as a function in a high level programming language like Lisp. The *non-differentiable* memory enables *abstract*, *scalable* and *precise* operations, but it requires reinforcement learning. It also provides a friendly *neural computer interface* to help the "programmer" reduce the search space by detecting and eliminating invalid choices (Section 2.1).

Within the MPC framework, we introduce the Neural Symbolic Machine (NSM) and apply it to semantic parsing. NSM contains a sequence-to-sequence neural network model ("programmer") augmented with a key-variable memory to save and reuse intermediate results for compositionality, and a non-differentiable Lisp interpreter ("computer") that executes programs against a large knowledge base. As code assist, the "computer" also helps reduce the search space by checking for syntax and semantic errors.

To efficiently train NSM from weak supervision, we apply the REINFORCE algorithm [**?** **?**]. However, the REINFORCE objective is known to be very hard to optimize starting from scratch. Therefore, we augment it with approximate gold programs found by an iterative maximum likelihood training process. During training, the model always puts a reasonable amount of probability on the best programs found so far, and anchoring the model to these high-reward programs greatly speeds up the training and helps to avoid local optimum.

Compared to existing neural program induction approaches, the efficient memory 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 [**?**], NSM achieves new state-of-the-art results with weak supervision. Compared to previous work, it is end-to-end, therefore does not require any feature engineering or domain-specific knowledge.

2 Neural Symbolic Machines

Within the MPC framework, we introduce the Neural Symbolic Machine (NSM) and apply it to semantic parsing. We first introduce a non-differentiable Lisp interpreter ("computer") that executes programs against a large knowledge base, and provides code assist. Then we describe a sequence-to-sequence neural network model ("programmer") augmented with a key-variable memory to save and reuse intermediate results for compositionality. Finally, we discuss how to successfully apply REINFORCE for training by augmenting it with approximate gold programs found by an iterative maximum likelihood training process.

Before diving into details, we first define the *semantic parsing* task: 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)[3], and let $\mathcal{P}$ denote a set of relations (or properties, e.g., PLACEOFBIRTH). A knowledge base $\mathbb{K}$ is a set of assertions or triples $(e_1, p, e_2) \in \mathcal{E} \times \mathcal{P} \times \mathcal{E}$, such as (ABELINCOLN, PLACEOFBIRTH, HODGENVILLE).

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

2.1 "Computer": Lisp interpreter with code assist

Operations learned by current neural network models with differentiable memory, such as addition or sorting, do not generalize perfectly to inputs that are larger than previously observed ones [**?** **?**]. In contrast, operations implemented in ordinary programming language are *abstract*, *scalable*, and *precise*, because no matter how large the input is or whether it has been seen or not, they will be processed precisely. Based on this observation, we implement all the operations necessary for semantic parsing with ordinary non-differentiable memory, and allow the "programmer" to use them with a high level general purpose programming language.

We adopt a Lisp interpreter with predefined functions listed in 1 as the "computer". The programs that can be executed by it are equivalent to the limited subset of λ-calculus in [**?**], but easier for a sequence-to-sequence model to generate given Lisp's simple syntax. Because Lisp is a general-purpose and high level language, it is easy to extend the model with more operations, which can be implemented as new functions, and complex constructs like control flows and loops.

A program C is a list of expressions $(c_1...c_N)$. 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)". F is one of the functions in Table 1, which take as input a list of arguments of specific types, and, when executed, returns the denotation of this expression in $\mathbb{K}$, which is typically a list of entities, and saves it in a new variable. A_k is F's kth argument, which can be either a relation $p \in \mathcal{P}$ or a variable v. The variables hold the results from previous computations, which can be either a list of entities from executing an expression or an entity resolved from the natural language input.

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

Table 1: Predefined functions. v represents a variable. p represents a relation in Freebase. $\geq$ and $\leq$ are defined on numbers and datetime.

To create a better *neural computer interface*, the interpreter provides code assist by producing a list of valid tokens for the "programmer" to pick from at each step. First, a valid token should not cause a syntax error, which is usually checked by modern compilers. For example, if the previous 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, a valid token should not cause a semantic error or run-time error, which can be detected by the interpreter using the value or denotation of previous expressions. For example, given that the previously generated tokens are "(", "***Hop***", "v", the next available token is restricted to the set of relations $\{p \vert e \in v, \exists e' : (e, p, e') \in \mathbb{K}\}$ that are reachable from entities in v. By providing this *neural computer interface*, the interpreter reduces the "programmer"'s search space by orders of magnitude, and enables weakly supervised learning on a large knowledge base.

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

The "computer" implements the operations (functions) and stores the values (intermediate results) in variables, which simplifies the task for the "programmer". The "programmer" only needs to map natural language into a program, which is a sequence of tokens that references operations and values in the "computer". We use a standard sequence-to-sequence model with attention and augment it with a key-variable memory to reference the values.

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 the decoder (Figure 2). Given a sequence of words $w_1, w_2...w_m$, each word w_t is mapped to a multi-dimensional embedding q_t (see details about the embeddings 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} (see details about the embeddings in Section 3), 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 [**?**]. The tokens of the program

$a_1, a_2 ... a_n$ are generated one by one using a softmax over the vocabulary of valid tokens for each step (Section 2.1).

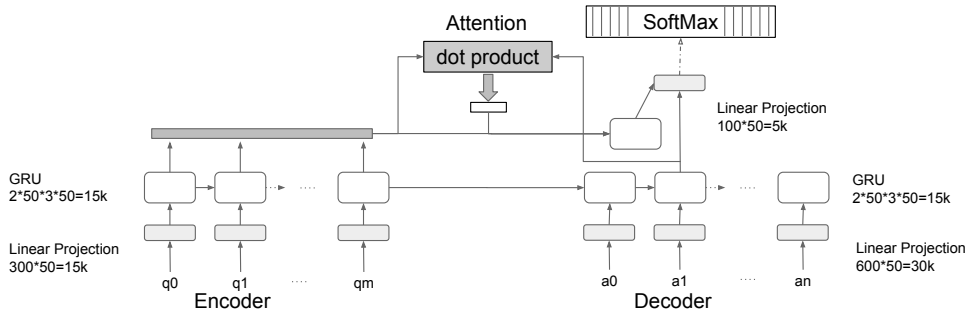

Figure 2: Seq2Seq model architecture with dot-product attention and Dropout at GRU input, output, and softmax layers.

To achieve compositionality, we augment the model with a **key-variable memory** (Figure 3). It enables the model to reference the values of intermediate results, which is saved in the "computer", when generating new expressions. Each entry in the key-variable memory has two components: a continuous multi-dimensional embedding key v_i, and a corresponding variable R_i that references a value in the "computer". Note that although the key embeddings are continuous and differentiable, the values referenced by the variables are simply the results returned by the Lisp interpreter, thus symbolic and non-differentiable. This makes it different from other memory-augmented neural networks that use continuous differentiable embeddings as the value of each memory entry [**?** **?**]. 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 "computer", and the key embedding 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 ")"), it gets executed, and the result is stored as the value of a new variable in the "computer". 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 value of the last variable is returned by the "programmer" as the answer.

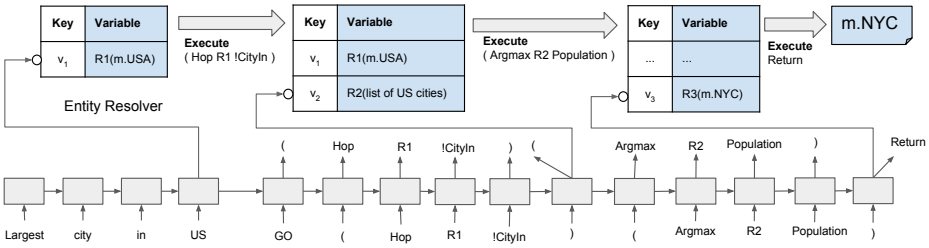

Figure 3: Semantic Parsing with NSM. The key embeddings of the key-variable memory are the output of the sequence model at certain encoding or decoding steps. For illustration purposes, we also show the values of the variables in parentheses, but the sequence model never sees these values, and only references them with the name of the variables such as “R1”. A special token “GO” indicates the start of decoding, and “RETURN” indicates the end of decoding.

2.3 Training NSM with Weak Supervision

Because the Neural Symbolic Machine uses non-differentiable memory for which backpropagation cannot be applied, we use REINFORCE[**?**] for effective training. When the reward signal is very sparse and the search space is large, it is a common practice to use supervised pretraining before policy gradient training [**?**], but it requires full supervision. In this work, we develop a method to augment REINFORCE using only weak supervision.

REINFORCE Training We can formulate NSM training as a reinforcement learning problem: given a query q, the state, action and reward at each time step $t \in \{0, 1, ..., T\}$ are (s_t, a_t, r_t). Since the environment is deterministic, the state is defined by the question q and the action sequence: $s_t = (q, a_{0:t-1})$, where $a_{0:t-1} = (a_0, ..., a_{t-1})$ is the history of actions at time t. A valid action at time t is $a_t \in A(s_t)$, where $A(s_t)$ is the set of valid tokens given by the symbolic "computer". Since each action corresponds to a token, each full history $a_{0:T}$ corresponds to a program. The reward $r_t = I[t = T] \cdot F_1(q, a_{0:T})$ is none zero only at the last step of decoding, which is the F_1 score computed using the gold answer and the answer generated by the program $a_{0:T}$. The reward of a program or action sequence is

$$R(q, a_{0:T}) = \sum_t r_t = F_1(q, a_{0:T}). \tag{1}$$

The agent's decision making procedure at each time step is characterized by a policy, $\pi(s, a, \theta) = P(a_t = a|q, a_{0:t-1}, \theta)$, where θ are the model parameters. Since the environment is deterministic, the probability of an action sequence or program $a_{0:T}$ is

$$P(a_{0:T}|q, \theta) = \prod_t P(a_t|q, a_{0:t-1}, \theta). \tag{2}$$

The policy can be optimized by policy gradient approaches such as REINFORCE [**? ?**]. The training objective is the sum of expected reward of each question

$$J^{RL}(\theta) = \sum_q \mathbb{E}_{P(a_{0:T}|q,\theta)}[R(q, a_{0:T})]. \tag{3}$$

and its gradient is computed as

$$\nabla_\theta J^{RL}(\theta) = \sum_q \sum_{a_{0:T}} P(a_{0:T}|q, \theta)[R(q, a_{0:T}) - B(q)] \nabla_\theta \log P(a_{0:T}|q, \theta), \tag{4}$$

where $B(q) = \sum_{a_{0:T}} P(a_{0:T}|q, \theta) R(q, a_{0:T})$ is a baseline that reduces the variance of the gradients without changing the optimum.

While REINFORCE training assumes a stochastic policy, we apply beam search for decoding. So different from the common practice that approximates equation 4 by sampling, we use the top k action sequences (programs) in the beam with normalized probabilities to approximate equation 4. This allows training to focus on sequences with high probability, which are on the decision boundaries, and reduces the variance of the gradient.

Empirically, REINFORCE training in our experiment converged slowly and often got stuck in local optima (see Section 3). The difficulty of REINFORCE training results from the sparse reward signal in a large search space, which caused the model probabilities for programs with non-zero rewards to be very small at the beginning. If the beam size k is small, the good programs might fall off the beam leading to zero gradients to all programs in beam. If the beam size is large, the training is very slow, and the normalized probabilities of good programs are still small, which leads to (1) near zero baselines, thus near zero gradients on "bad" programs (2) near zero gradients on good programs due to the first $P(a_{0:T}|q, \theta)$ term in equation 4.

Finding Approximate Gold Programs A solution to the problem with REINFORCE training is to add gold programs into the beam with a reasonably large probability. This is similar to the common practice of applying supervised pre-training before policy gradient training [**?**]. But since we do not have the gold programs and the search space is very large, we use an iterative process interleaving decoding with a large beam size and maximum likelihood training to efficiently find approximations to the gold programs.

The maximum likelihood (ML) training objective is

$$J^{ML}(\theta) = \sum_q \log P(a^{best}_{0:T}(q)|q, \theta) \tag{5}$$

where $a^{best}_{0:T}(q)$ is the program that achieved highest reward with shortest length on question q from all the iterations before (a question is not included at training time if we did not find any program leading to positive reward).

Training with an ML objective is fast because there is at most one program per example and the gradient is not weighted by model probability. Although decoding with large beam size is slow, we can train for multiple epochs after each decoding step. This iterative process also has a bootstrapping effect that a better model leads to better $a_{0:T}^{best}(q)$ through decoding and better $a_{0:T}^{best}(q)$ leads to a better model through training, which accelerates the search for approximate gold programs.

Even with a large beam size, some complex programs are still hard to find because of the large search space. A common solution to this problem is to use curriculum learning [**?** **?**]. The complexity of the program and the search space are controlled by two factors: (1) the set of functions used; (2) the length of the program. We apply curriculum learning by gradually increasing the program complexity (see more details in Section 3) in the search for approximate gold programs.

Nevertheless, the ML objective has drawbacks. (1) The best program $a_{0:T}^{best}(q)$ for a question could be a spurious program that accidentally produced the correct answer (e.g., answering PLACEOFBIRTH with PLACEOFDEATH relation if these two places happen to be the same), and thus does not generalize to other questions. (2) Because training lacks explicit negative examples, the model often fails to distinguish between tokens that are related to one another. For example, differentiating PARENTSOF vs. SIBLINGSOF vs. CHILDRENOF can be hard. Because of these drawbacks, we only use ML to collect the approximate gold programs to augment REINFORCE.

Augmented REINFORCE Training To overcome the difficulty of REINFORCE training, we use the approximate gold programs collected from iterative ML training to augment the programs in the beam. This is related to the common practice in reinforcement learning [**?**] to replay rare successful experiences to reduce the training variance and improve training efficiency. Our approach is also similar to recent developments [**?**] in machine translation, where ML and RL objectives are linearly combined, because anchoring the model to some high-reward outputs stabilizes the training.

In our case, we don't have access to ground-truth Lisp programs, but have access to the best Lisp programs found so far as approximations to the gold programs. So we augment the REINFORCE training by adding the approximate gold program for a question $a_{0:T}^{best}(q)$ to the final beam with probability α, and the probabilities of the original programs in the beam will be normalized to be $(1-\alpha)$. The rest of the process is the same as in standard REINFORCE (Section 2.3). In this way, the model always puts a reasonable amount of probability on the best Lisp program found so far during training.

3 Experiments and analysis

Here we show that NSM is able to learn a semantic parser from weak supervision over a large knowledge base. For evaluation we chose WEBQUESTIONSSP, a challenging semantic parsing dataset with strong baselines. Experiments show that NSM achieves new state-of-the-art performance on WEBQUESTIONSSP with weak supervision. Code assist, augmented REINFORCE, curriculum learning, and reducing overfitting are the essential elements to this result, which we will describe in detail.

3.1 Semantic Parsing and The WEBQUESTIONSSP Dataset

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

We evaluate our model NSM on the task of semantic parsing. Specifically, we used the challenging semantic parsing dataset WEBQUESTIONSSP [**?**], 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 [**?**] using Amazon Mechanical Turk and updated by annotators who are familiar with the design of Freebase [**?**]. We further separated out 620 questions in the training set as a validation set. For query pre-processing 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 [**?**] with about 94% of the gold root entities being included in the resolution results. Similar to [**?**], 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 [**?**] we use the last publicly available snapshot of Freebase KB. Since NSM training requires random access to Freebase during decoding, we preprocessed Freebase by removing predicates that are not related to world knowledge (starting with "/common/", "/type/", "/freebase/")[4], and removing all text valued predicates, which are rarely the answer. Out of all 23K relations, only 434 relations are removed during preprocessing. This results in a graph that fits in memory 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 [**?**]. 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 Freebase id of a relation and also add a projection matrix to transform them into 50 dimension. A Freebase id contains three parts: domain, type, and property. For example, the Freebase id for PARENTSOF is "/people/person/parents". "people" is the domain name, "person" is the type name and "parents" is the property name. The embedding for a relation is constructed by concatenating two vectors. The first vector is the average of the word embeddings in the domain and type name. The second vector is the average of the word embeddings in the property name. For example, if the word embedding dimension is 300, the embedding dimension for "/people/person/parents" will be 600. The first 300 dimensions are the average of the word embeddings for "people" and "person", and the second 300 dimensions are the word embedding for "parents".

Dropout rate is set to 0.5, and we see a clear tendency that larger dropout produces better performance, indicating overfitting is a major problem for learning.

3.3 Training Details

In iterative maximum likelihood training, the decoding uses beam size $k = 100$ to update the approximate gold programs and the model is trained for 20 epochs after each iteration. We use Adam optimizer [**?**] with initial learning rate 0.001 for optimization. In our experiment, this process usually converges after a few (5-8) iterations. Inspired by the staged generation process in [**?**], the curriculum learning contains two steps. We first run the iterative ML training procedure for 10 iterations with the programs constrained to only use the "Hop" function and the maximum number of expressions is 2. Then we run the iterative training procedure again, but use both "Hop" and "Equal", and the maximum number of expressions is 3. However, the relations used by the "Hop" function are restricted to those that appeared in $a_{0:T}^{best}(q)$ in the first step. We plan to make use of more functions to further improve the results in future work.

For REINFORCE training, the best hyperparameters are chosen using the validation set. We use beam size $k = 5$ for decoding, and α is set to 0.1. Because the dataset is small and some relations are only used once in the whole training set, we train the model on the entire training set for 200 iterations with the best hyperparameters. Then we train the model with learning rate decay until convergence. Learning rate is decayed as $g_t = g_0 \times \beta^{\frac{\max(0, t-t_s)}{m}}$, where $g_0 = 0.001$, $\beta = 0.5$ $m = 1000$, and t_s is the number of training steps at the end of iteration 200.

Since decoding needs to query the knowledge graph constantly, the speed bottleneck for training is decoding. We address this problem in our implementation by partitioning the dataset, and using multiple decoders in parallel to handle each partition. See figure 4 for an illustration. We use 100 decoders, which queries 50 KG servers, and one trainer. The neural network model is implemented

[4]Except that we kept "/common/topic/notable_types".

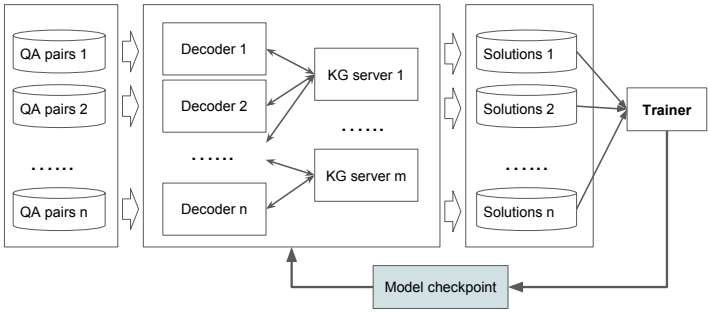

Figure 4: System Architecture. 100 decoders, 50 KG servers and 1 trainer.

in TensorFlow. Since the model is small, we didn't see a significant speedup by using GPU, so all the decoders and the trainer are using CPU only.

3.4 Results and Discussion

We evaluate performance using the offical evaluation script 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 measures the percentage of questions that are answered exactly.

Comparison to previous approaches [**? ?**] is shown in Table 2. Our model beats the state-of-the-art with weak supervision by a large margin, and is approaching the state-of-the-art with full supervision, which is a much stronger condition. Besides, our model is end-to-end, therefore does not rely on feature engineering or hand-crafted rules

Model	Avg. Prec.@1	Avg. Rec.@1	Avg. F1@1	Acc.@1
STAGG	67.3	73.1	66.8	58.8
NSM – our model	70.8	76.0	**69.0**	59.5
STAGG (full supervision)	70.9	80.3	71.7	63.9

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 or feature engineering. "@1" represents the program with the highest probability in beam. Full supervision uses human annotated programs for training.

There are four key ingredients leading to the final performance. The first one is the neural computer interface that provides code assist by checking for syntax and semantic errors. The semantic checks can be very effective for open-domain knowledge bases with large number of relations. For our task, it reduces the average number of choices from 23K per step (all relations) to less than 100 (average number of relations connected to an entity). This reduction of search space enables successful search and sequence-to-sequence training.

The second ingredient is augmented REINFORCE training. In Table 3 we use the validation set to compare augmented REINFORCE, REINFORCE, and iterative ML training. REINFORCE gets stuck at local maxima, and iterative ML training is not directly optimizing the F1 measure, so both achieve suboptimal results. In contrast, augmented REINFORCE achieves the best performance on both training and validation set.

Settings	Train Avg. F1@1	Valid Avg. F1@1
iterative ML only	68.6	60.1
REINFORCE only	55.1	47.8
Augmented REINFORCE	83.0	**67.2**

Table 3: Compare augmented REINFORCE, REINFORCE, and iterative ML.

The third ingredient is curriculum learning to find approximate gold programs. We compare the performance of the best programs found by curriculum learning and no curriculum learning in Table 4. The best programs found with curriculum learning are better than those without curriculum learning by a large margin on every metric.

Settings	Avg. Prec.@Best	Avg. Rec.@Best	Avg. F1@Best	Acc.@Best
No curriculum	79.1	91.1	78.5	67.2
Curriculum	88.6	96.1	89.5	79.8

Table 4: Comparison of curriculum learning strategies. @Best represents the program with the highest F1 score in beam.

The last important ingredient is reducing overfitting. Given the small size of the dataset, overfitting is a major problem for training neural network models. We show the contributions of different techniques in Table 5. Note that after all the techniques been applied, the model is still overfitting with training F1@1=83.0% and validation F1@1=67.2%.

Settings	Δ Avg. F1@1
-Pretrained word embeddings	-5.5
-Pretrained relation embeddings	-2.7
-Dropout on GRU input and output	-2.4
-Dropout on softmax	-1.1
-Anonymize entity tokens	-2.0

Table 5: Contributions of different techniques to reduce overfitting.

There are two main sources of errors.

1. **Search failure**: the correct program is not found during search for approximate gold programs, either because the beam size is not large enough, or because the set of functions implemented by the interpreter is insufficient. The 89.5% F1@Best (Table 4) indicates that at least 10% of queries are of this kind.
2. **Ranking failure**: the approximate gold programs are found, but are not ranked at the top. Because training error is low, this is largely due to overfitting. The 67.2% F1@1 (Table 3) indicates that around 20% of queries are of this kind.

4 Related work

Among deep learning models for program induction, Reinforcement Learning Neural Turing Machines (RL-NTMs) [**?**] are the most similar to Neural Symbolic Machines, as a non-differentiable machine is controlled by a sequence model to perform a certain task. Therefore, both of them rely on REINFORCE for effective training. The main difference between the two is the abstraction level of the programming language. RL-NTM has lower level operations such as memory address manipulation and byte reading/writing, while NSM uses a higher level programming language over a large knowledge base that includes operations such as following certain relations from certain entities, or sorting a list of entities based on a property, which is more suitable for representing semantics.

We formulate NSM training as an instance of reinforcement learning [**?**], where the task is episodic, rewards are provided at the final step, and the environment consists of the weak supervision and the symbolic machine. Much like RL-NTMs, the environment is deterministic but with a large state space, therefore function approximation is needed to learn good policies. Wiseman and Rush [**?**] proposed a max-margin approach to train a sequence-to-sequence scorer. However, their training procedure is more complicated, so we did not implement their method in this work.

NSM is very similar to Neural Programmer [**?**] and Dynamic Neural Module Network [**?**] in the sense that they are all solving the problem of semantic parsing from structured data, and therefore generate programs using very similar languages of semantics (λ-calculus [**?**]). The main difference between these approaches is how an intermediate result (the memory) is represented. Neural Programmer and Dynamic-NMN chose to represent results as vectors of weights (row selectors

and attention vectors), which enables backpropagation training and searching all possible programs in parallel. However, this strategy is not applicable to large knowledge base such as Freebase, which have around 100M entities, and 20k+ relations. Instead, NSM chose a more scalable approach, which let the "computer" save the intermediate results and references them using variable names (such as "R1" for all cities in US).

NSM is similar to the Path Ranking Algorithm (PRA) [**?**] in the sense that the semantics is encoded as a sequences of actions, and denotations are used to cut down the search space during learning. NSM is more powerful than PRA by 1) allowing more complex semantics to be composed through the use of key-variable memories. 2) controlling the search procedure with a trained deep learning model, while PRA can only sample actions with uniform probabilities. 3) allowing input questions (text) to express complex relations, and then dynamically generating action sequences. PRA can combine multiple semantic representations to produce the final predictions, which remains a future work for NSM.

5 Conclusion

In this work, we propose the Manager-Programmer-Computer framework for neural program induction. It integrates neural networks with *non-differentiable* memory to support *abstract*, *scalable* and *precise* operations through a friendly *neural computer interface*. Within this framework, we introduce the Neural Symbolic Machine, which integrates a sequence-to-sequence neural "programmer" and a Lisp interpreter with code assist. Because the interpreter is non-differentiable, we apply reinforcement learning and use approximate gold programs found by the iterative maximum likelihood training process to augment the REINFORCE training. NSM achieves new state-of-the-art results on a challenging semantic parsing dataset WEBQUESTIONSSP with weak supervision. Compared to previous approaches, it is end-to-end, therefore does not require feature engineering or domain specific knowledge.

Acknowledgements

We thank for discussions and helps from Arvind Neelakantan, Mohammad Norouzi, Tom Kwiatkowski, Eugene Brevdo, Lukasz Kaizer, Thomas Strohmann, Yonghui Wu, Zhifeng Chen, and Alexandre Lacoste.