# KNOWLEDGE REPRESENTATION AND REASONING WITH DEEP NEURAL NETWORKS

A Dissertation Presented

by

ARVIND NEELAKANTAN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2017

College of Information and Computer Sciences

# KNOWLEDGE REPRESENTATION AND REASONING WITH DEEP NEURAL NETWORKS

A Dissertation Presented

by

ARVIND NEELAKANTAN

Approved as to style and content by:

Andrew K. McCallum, Chair

Rajesh Bhatt, Member

Samuel R. Bowman, Member

Subhransu Maji, Member

Benjamin M. Marlin, Member

Vijay Saraswat, Member

James Allan, Chair
College of Information and Computer Sciences

# ACKNOWLEDGMENTS

First, I would like to thank my advisor Andrew McCallum for his great advice and support. His optimism and enthusiasm helped me explore ambitious research plans during my PhD. I would like to thank the committee members for suggesting thoughtful experiments to analyze and improve the methods presented in the thesis. I would like to thank members of IESL: David, Luke, Ari, Lakshmi, Emma, Pat, Nick, Craig, Rajarshi, Trapit, June, Lorraine, Anton, Limin, Mike, Ben, Alex, Sam, Jinho, Kate, and Jeevan from whom I have learned lot about research on a diverse set of topics.

I spent a considerable amount of time during my PhD at Google Brain where I had great mentors, Quoc Le and Ilya Sutskever during my first internship. I would like to thank Martin Abadi and Dario Amodei for their collaboration in my second project. I thank the Google Brain team for providing an excellent atmosphere to learn about many topics in neural networks and different tasks. Discussing research with fellow internship friends: Shane, Sam, Laurent, Georgia and others was hugely beneficial. My first internship at MSR with Ming-Wei Chang was a great learning experience. I would like to thank Michael Collins for believing in me during my MS and teaching me what it means to do research.

I would like to thank my family including my extended family and cousins for being extremely supportive and encouraging. My parents, Anand, and Priya have made many sacrifices to make sure that I get the best education and they derive immense joy from my academic progress. I hope this thesis makes them feel happy.

# ABSTRACT

## KNOWLEDGE REPRESENTATION AND REASONING WITH DEEP NEURAL NETWORKS

FEBRUARY 2017

ARVIND NEELAKANTAN

B.Tech, NATIONAL INSTITUTE OF TECHNOLOGY TRICHY

M.Sc., COLUMBIA UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Andrew K. McCallum

Knowledge representation and reasoning is one of the central challenges of artificial intelligence, and has important implications in many fields including natural language understanding and robotics. Representing knowledge with symbols, and reasoning via search and logic has been the dominant paradigm for many decades. In this work, we use deep neural networks to learn to both represent symbols and perform reasoning end-to-end from data. By learning powerful non-linear models, our approach generalizes to massive amounts of knowledge and works well with messy real-world data using minimal human effort. First, we show that recurrent neural networks with an attention mechanism achieve state-of-the-art reasoning on a large structured knowledge graph. Next, we develop Neural Programmer, a neural network augmented with discrete operations that can be learned to induce latent programs with backpropagation. We apply Neural Programmer to induce short programs on

two datasets: a synthetic dataset requiring arithmetic and logic reasoning, and a natural language question answering dataset that requires reasoning on semi-structured Wikipedia tables. We present what is to our awareness the first weakly supervised, end-to-end neural network model to induce such programs on a real-world dataset. Unlike previous learning approaches to program induction, the model does not require domain-specific grammars, rules, or annotations. Finally, we discuss methods to scale Neural Programmer training to large databases.

# TABLE OF CONTENTS

# LIST OF TABLES

**Table** **Page**

# LIST OF FIGURES

**Figure** **Page**

# CHAPTER 1

# INTRODUCTION

## 1.1 Knowledge Representation and Reasoning

Knowledge representation and reasoning (KRR) is one of the central challenges of artificial intelligence, and has important implications in many fields including natural language understanding and robotics. Knowledge representation deals with representing world knowledge in a form that can be used by computers, and reasoning involves manipulating knowledge to answer a query. An ideal knowledge representation and reasoning system should be able to generalize to large number of concepts and relationships, should work well in real-world settings, and should be broadly applicable across tasks and domains. An example KRR system are natural language interface models [148, 150] which map text questions to concepts and perform reasoning in a provided knowledge base to produce the required answer.

Representing knowledge with symbols, and reasoning via search and logic has been the dominant paradigm for many decades. The General Problem Solver [112] is one of the earliest examples of such a system where knowledge is represented as a set of rules that express permissible transformation operations, and reasoning is performed using means-end analysis, a heuristic search algorithm. Several other general purpose systems [111, 103] were built in the same spirit. The advantage of these systems are that they are precise, however, they failed to work beyond toy problems since manually programming general purpose rules are challenging and the search algorithms do not scale well to large number of rules.

The failure of general purpose systems led to the development of expert systems which are designed to work on a narrow task [16, 30, 89]. These systems require enormous human effort as they are manually programmed and not learned. Inductive Logic Programming [65, 80] methods learn hypothesis rules given background knowledge, and a set of positive and negative examples. The systems we have discussed till now do not model uncertainty which is essential in practical applications. Markov Logic Networks [100] (MLNs) handle uncertainty using weight attached to every rule. Practical applications of MLNs have been limited because inference in MLNs is not scalable to large number of rules. By restricting to a simpler model family, Probabilistic Soft Logic [55] provide a more scalable solution to handle uncertainty.

While symbolic KRR systems developed so far have had some success, they suffer from multiple drawbacks. First, since knowledge is represented using symbols it is challenging to generalize to large number of concepts and relationships since each symbol is modeled independently without sharing information between them. Next, even though some of the more modern systems have a learning component they still rely on laborious manual effort severely limiting the applicability of these systems. Finally, these systems typically only have a simple symbol grounding component, the process of mapping raw inputs to high level concepts, to map sensory data to knowledge concepts making them not suitable for dealing with messy real-world data.

## 1.2 Knowledge Representation and Reasoning with Deep Neural Networks

In this work, we use deep neural networks (DNNs) to learn to both represent symbols and perform reasoning end-to-end from data. We represent concepts and relationships using distributed (or vector) representations which naturally leads to sharing information between the symbols. By learning powerful non-linear composition models we avoid the combinatorial explosion problem faced by symbolic systems.

Finally, symbol grounding is also modeled by a DNN which can be learned from data using minimal human effort.

While DNNs have been around for many decades, the first big successes of these models happened recently in perception tasks such as speech recognition [44] and image recognition [58]. As mentioned previously, the application of DNNs to KRR is attractive, however, it is not straightforward due to the following challenges: 1) input in perception tasks are continuous data while in KRR they are discrete, and 2) reasoning typically involves programs with discrete operations or rules while perception tasks often involve *fuzzy* processing. The challenges associated with handling discrete data and discrete processing have raised questions on whether DNNs are applicable for KRR.

Symbol processing with rules and logic, and machine learning with DNNs have remained mostly parallel research areas for many decades [61]. In this thesis, we ask two big questions: 1) can we represent discrete symbols with distributed representations and learn them?, 2) can we learn neural network models to perform reasoning using these representations?. By developing new DNN models and adopting existing ones, we show that knowledge representation and symbolic reasoning can be successfully performed with DNNs. We consider two specific tasks: 1) simple reasoning on a large structured knowledge graph, and 2) reasoning that involves inducing latent programs on semi-structured database tables.

## 1.3 Reasoning in Knowledge Graphs

Recent years has seen the creation of massive knowledge bases (KBs) [10, 122, 17] that are useful in downstream tasks such as question answering and dialogue systems. These KBs contain structured information of the form relation(entity1, entity2). While these KBs contain millions of facts, they are still incomplete. A promising way to complete the KB is by performing logic reasoning in the KB [107].

Here, the paths connecting an entity pair are used to infer missing relations between them. For example, using the facts *IsBasedIn(Microsoft, Seattle)*, *StateLocatedIn(Seattle, Washington)* and *CountryLocatedIn(Washington, USA)* we can infer the fact *CountryOfHeadquarters(Microsoft, USA)*. Here we have used the path connecting the entities *Microsoft* and *USA* to infer a new fact. The logical rule used here is:

$$IsBasedIn(X,A) \wedge StateLocatedIn(A,\ B) \wedge CountryLocatedIn(B,\ Y) \Rightarrow$$
$$CountryOfHeadquarters(X,\ Y)$$

where *IsBasedIn - StateLocatedIn - CountryLocatedIn* is a path connecting the entity pair *(Microsoft, USA)* in the KB graph, and *IsBasedIn, StateLocatedIn* and *CountryLocatedIn* are the relations in the path.

Previous symbolic approaches to this task do not generalize to unseen paths as they model each path individually without sharing information across them [107, 62, 63]. We model the paths using recurrent neural networks (RNNs) and select predictive paths among the set of paths connecting the entity pair by an attention mechanism. Since modern KBs have tens of thousands of relations, there are many unseen paths at test time. Our approach can generalize to paths that are unseen during training and achieves state-of-the-art results. We show that the choice of attention mechanism has a significant effect on the performance and additionally modeling the entities in the path leads to large accuracy improvements. We discuss reasoning in knowledge graphs in detail in Chapter 2 and Chapter 3.

## 1.4 Program Induction

While the first task focused on simple reasoning on a large structured knowledge base, the next task involves more sophisticated reasoning on small semi-structured tables. Figure 1.1 shows an example table with the questions that needs to be answered. The question has to be mapped to a program, consisting of sequence of discrete operations, which when executed on the table gives the required answer. Learning discrete

| Kilometers | Name | Location | Intersecting routes |
|---|---|---|---|
| 0 | Netanya mall | Netanya | Petah Tikva Street, Raziel Road, Herzl Road |
| 0.6 | Netanya interchange | Netanya | Highway 2 |
| 0.75 | HaRakevet Road | Netanya | Entrance to Netanya Railway Station |
| 1.45 | Pinkas Street | Netanya | Pinkas Street |
| 2.2 | Deganya Road | Netanya | Deganya Road |
| 2.8 | Beit Yitzhak | Beit Yitzhak | HaSharon Road |
| 3.8 | Ganot Hadar | Ganot Hadar | Entrance to Ganot Hadar and Nordiya |
| 4.3 | HaSharon Junction ("Beit Lid") | Ganot Hadar | Highway 4 |
| 6.2 | Rabin Boulevard | Kfar Yona | Entrance to Kfar Yona |
| 7.3 | Begin Boulevard | Kfar Yona | Entrance to Kfar Yona |
| 7.7 | Ha'Atzmaut Street | Haniel | Entrance to Haniel |
| 9.3 | Yanuv junction | Yanuv | Route 5613, entrance to Yanuv |
| 10 | HaErez Way | Tnuvot industrial zone | Entrance to Tnuvot and Burgata |
| 12 | Be'erotayim junction | Hagor | Entrance to Be'erotayim and Olesh, Route 5614 toward Qalansawe |
| 13.2 | Nitzanei Oz junction | Nitzanei Oz | Route 5714, entrance to Nitzanei Oz and Yad Hana |
| 14.5 | Nitzanei Oz interchange | Nitzanei Oz | Highway 6 |
| 15 | IDF checkpoint | Tulkarm | Entrance to Nitzanei Shalom industrial zone |

Which section is the longest ?
how many portions are located in Netanya ?

Figure 1.1: An example table with questions that can be answered by reasoning on the contents of the table.

operations with DNNs is notoriously challenging [52]. Existing approaches to program induction either require the full program as supervision which is expensive to obtain or require domain-specific grammars, rules, or annotations making them not broadly applicable.

We develop Neural Programmer, a neural network augmented with discrete operations that can be learned to induce latent programs with backpropagation. The model can be trained with a weak supervision signal consisting of input and output pairs while the program to be induced is latent. First, we conduct experiments on a synthetic dataset that requires text questions to be mapped to latent programs. While

standard DNN architectures like RNNs and RNNs with attention perform poorly on this task, Neural Programmer works almost perfectly on all the different versions of the synthetic dataset. Neural Programmer and the synthetic data experiments are discussed in Chapter 4 and Chapter 5 respectively.

Next we consider semantic parsing where the task is to map natural language questions to programs or semantic parses which produce the required answer when executed on the provided table. We experiment with WikiTableQuestions dataset [92], a semantic parsing dataset that contains only 10,000 training examples. While it is generally believed that DNNs require large number of training examples to work well, Neural Programmer performs competitively to current state-of-the-art parsers on this dataset. Neural Programmer is learned end-to-end from data while existing approaches require domain-specific grammar rules. Semantic parsing with Neural Programmer is described in Chapter 6.

Finally, we discuss ideas to scale Neural Programmer by employing sparse attention and learning to halt in Chapter 7. Scaling Neural Programmer to large tables is currently problematic because: 1) the model uses soft selection to be differentiable that requires executing all built-in operations at every timestep, and 2) the model runs for a fixed number of maximum timesteps performing many wasteful operation executions when the program to be induced is of lesser complexity. We plan to experiment with the proposed ideas on the WikiTableQuestions dataset.

## 1.5 Summary of Contributions

- We show that recurrent neural networks combined with the appropriate attention mechanism achieve state-of-the-art reasoning on knowledge graphs.

- We develop Neural Programmer, a neural network augmented with discrete operations that can be learned to induce latent programs with backpropagation.

We first experiment with synthetic data where we use Neural Programmer to induce latent programs involving arithmetic and logic operations.

- We apply Neural Programmer for natural language semantic parsing achieving competitive performance using only 10,000 training examples. Unlike previous work on semantic parsing, Neural Programmer does not require domain-specific grammars, rules, or annotations.

- Overall, we develop DNN models for KRR that are learned end-to-end from data and achieve state-of-the-art or competitive performance with minimal human effort. The main components of our approach consists of RNNs to model sequences, attention mechanism as selection procedure and training by back-propagation.

## 1.6 Declaration of Previous Work

We first proposed the idea of using recurrent neural networks for path reasoning in knowledge graphs in

> Arvind Neelakantan, Benjamin Roth, and Andrew McCallum. Knowledge base completion using compositional vector space models. *Workshop on Automated Knowledge Base Construction at NIPS*, 2014

This work later appeared as a conference paper in

> Arvind Neelakantan, Benjamin Roth, and Andrew McCallum. Compositional vector space models for knowledge base completion. *ACL*, 2015

An extension of this work that explores different attention mechanisms and is additionally conditioned on the entities in the path appeared in

> Rajarshi Das, Arvind Neelakantan, David Belanger, and Andrew McCallum. Chains of reasoning over entities, relations, and text using recurrent neural networks. *EACL*, 2017

The Neural Programmer model was first presented in

> Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural Programmer: Inducing latent programs with gradient descent. *ICLR*, 2016

Application of Neural Programmer to natural language semantic parsing will appear in

> Arvind Neelakantan, Quoc V Le, Martin Abadi, Andrew McCallum, and Dario Amodei. Learning a natural language interface with neural programmer. *ICLR*, 2017

During graduate school, I contributed to other publications that are not part of the thesis

> Arvind Neelakantan, Jeevan Shankar, Alexandre Passos, and Andrew McCallum. Efficient non-parametric estimation of multiple embeddings per word in vector space. *EMNLP*, 2014

> Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *ICLR Workshop*, 2016 (First two authors contributed equally)

> Patrick Verga, Arvind Neelakantan, and Andrew McCallum. Generalizing to unseen entities and entity pairs with row-less universal schema. *EACL*, 2017 (First two authors contributed equally)

# CHAPTER 2

# KNOWLEDGE GRAPH REASONING WITH RECURRENT NEURAL NETWORKS

In this chapter, we use deep neural networks to perform logic reasoning on knowledge bases. Constructing large knowledge bases supports downstream reasoning about resolved entities and their relations, rather than the noisy textual evidence surrounding their natural language mentions. For this reason KBs have been of increasing interest in both industry and academia [10, 122, 17]. Such KBs typically contain many millions of facts, most of them relation(entity1,entity2) "triples" (also known as binary relations) such as *(Barack Obama, presidentOf, USA)* and *(Brad Pitt, marriedTo, Angelina Jolie)*.

However, even the largest KBs are woefully incomplete [76], missing many important facts, and therefore damaging their usefulness in downstream tasks. Ironically, these missing facts can frequently be inferred from other facts already in the KB, thus representing a sort of inconsistency that can be repaired by the application of an automated process. The addition of new triples by leveraging existing triples is typically known as *KB completion*.

Early work on this problem focused on learning symbolic rules. For example, Schoenmackers et al. [107] learns Horn clauses predictive of new binary relations by exhausitively exploring relational paths of increasing length, and selecting those surpassing an accuracy threshold. (A "path" is a sequence of triples in which the second entity of each triple matches the first entity of the next triple.) Lao et al. [62] introduced the Path Ranking Algorithm (PRA), which greatly improves efficiency and robustness by replacing exhaustive search with random walks, and using unique paths as

features in a per-target-relation binary classifier. A typical predictive feature learned by PRA is that *CountryOfHeadquarters(X, Y)* is implied by *IsBasedIn(X,A)* and *StateLocatedIn(A, B)* and *CountryLocatedIn(B, Y)*. Given *IsBasedIn(Microsoft, Seattle)*, *StateLocatedIn(Seattle, Washington)* and *CountryLocatedIn(Washington, USA)*, we can infer the fact *CountryOfHeadquarters(Microsoft, USA)* using the predictive feature. In later work, Lao et al. [63] greatly increase available raw material for paths by augmenting KB-schema relations with relations defined by the text connecting mentions of entities in a large corpus (also known as OpenIE relations [5]).

However, these symbolic methods can produce many millions of distinct paths, each of which is categorically distinct, treated by PRA as a distinct feature. (See Figure 2.1.) Even putting aside the OpenIE relations, this limits the applicability of these methods to modern KBs that have thousands of relation types, since the number of distinct paths increases rapidly with the number of relation types. If textually-defined OpenIE relations are included, the problem is obviously far more severe.

Better generalization can be gained by operating on embedded vector representations of relations, in which vector similarity can be interpreted as semantic similarity. For example, Bordes et al. [11] learn low-dimensional vector representations of entities and KB relations, such that vector differences between two entities should be close to the vectors associated with their relations. This approach can find relation synonyms, and thus perform a kind of one-to-one, non-path-based relation prediction for KB completion. Similarly Nickel et al. [87] and Socher et al. [116] perform KB completion by learning embeddings of relations, but based on matrices or tensors. Universal schema [101] learns to perform relation prediction cast as matrix completion (likewise using vector embeddings), but predicts textually-defined OpenIE relations as well as KB relations, and embeds entity-pairs in addition to individual entities.

Like all of the above, it also reasons about individual relations, not the evidence of a connected path of relations.

We propose an approach combining the advantages of (a) reasoning about conjunctions of relations connected in a path, and (b) generalization through vector embeddings, and (c) reasoning non-atomically and compositionally about the elements of the path, for further generalization.

Our method uses recurrent neural networks (RNNs) [137] to compose the semantics of relations in an arbitrary-length path. At each path-step it consumes both the vector embedding of the next relation, and the vector representing the path-so-far, then outputs a composed vector (representing the extended path-so-far), which will be the input to the next step. After consuming a path, the RNN should output a vector in the semantic neighborhood of the relation between the first and last entity of the path. For example, after consuming the relation vectors along the path *Melinda Gates* $\rightarrow$ *Bill Gates* $\rightarrow$ *Microsoft* $\rightarrow$ *Seattle*, our method produces a vector very close to the relation *livesIn*.

Our compositional approach allow us at test time to make predictions from paths that were unseen during training, because of the generalization provided by vector neighborhoods, and because they are composed in non-atomic fashion. This allows our model to seamlessly perform inference on many millions of paths in the KB graph. In most of our experiments, we learn a separate RNN for predicting each relation type, but alternatively, by learning a single high-capacity composition function for all relation types, our method can perform zero-shot learning—predicting new relation types for which the composition function was never explicitly trained.

Related to our work, new versions of PRA [33, 34] use pre-trained vector representations of relations to alleviate its feature explosion problem—but the core mechanism continues to be a classifier based on atomic-path features. In the 2013 work many paths are collapsed by clustering paths according to their relations' embeddings, and

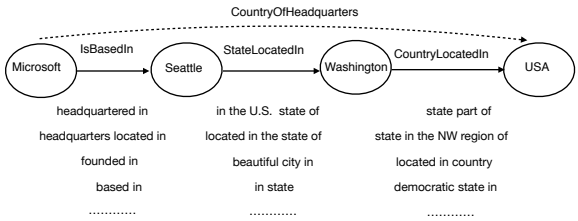



Figure 2.1: Semantically similar paths connecting entity pair (Microsoft, USA).

substituting cluster ids for the original relation types. In the 2014 work unseen paths are mapped to nearby paths seen at training time, where nearness is measured using the embeddings. Neither is able to perform zero-shot learning since there must be a classifer for each predicted relation type. Furthermore their pre-trained vectors do not have the opportunity to be tuned to the KB completion task because the two sub-tasks are completely disentangled.

An additional contribution of our work is a new large-scale data set of over 52 million triples, and its preprocessing for purposes of path-based KB completion (can be downloaded from `http://iesl.cs.umass.edu/downloads/inferencerules/release.tar.gz`). The dataset is build from the combination of Freebase [10] and Google's entity linking in ClueWeb [90]. Rather than Gardner's 1000 distinct paths per relation type, we have over 2 million. Rather than Gardner's 200 entity pairs, we use over 10k. All experimental comparisons below are performed on this new data set.

On this challenging large-scale dataset our compositional method outperforms PRA [63], and Cluster PRA [33] by 11% and 7% respectively. A further contribution of our work is a new, surprisingly strong baseline method using classifiers of path bigram features, which beats PRA and Cluster PRA, and statistically ties our compositional method. Our analysis shows that our method has substantially different strengths than the new baseline, and the combination of the two yields a 15% improvement over Gardner et al. [33]. We also show that our zero-shot model is indeed capable of predicting new unseen relation types.

## 2.1 Background

We give background on PRA which we use to obtain a set of paths connecting the entity pairs and the RNN model which we employ to model the composition function.

### 2.1.1 Path Ranking Algorithm

Since it is impractical to exhaustively obtain the set of all paths connecting an entity pair in the large KB graph, we use PRA [62] to obtain a set of paths connecting the entity pairs. Given a training set of entity pairs for a relation, PRA heuristically finds a set of paths by performing random walks from the source and target nodes keeping the most common paths. We use PRA to find millions of distinct paths per relation type. We do not use the random walk probabilities given by PRA since using it did not yield improvements in our experiments.

## 2.2 Recurrent Neural Networks for KB Completion

We propose a RNN model for KB completion that reasons on the paths connecting an entity pair to predict missing relation types. We call our model Path-RNN. Let $(e_s, e_t)$ be an entity pair and $\mathcal{S}$ denote the set of paths between them. The set $\mathcal{S}$ is obtained by performing random walks in the knowledge graph starting from $e_s$ till $e_t$. Let $\pi = \{e_s, r_1, e_1, r_2, \ldots, r_k, e_t\} \in \mathcal{S}$ denote a path between $(e_s, e_t)$. The length of a path is the number of relations in it, hence, $(\text{len}(\pi) = k)$. Let $\mathbf{y_{r_t}} \in \mathbb{R}^d$ denote the vector representation of $r_t$. The Path-RNN model combines all the *relations* in $\pi$ sequentially using a RNN with an intermediate representation $\mathbf{h_t} \in \mathbb{R}^{\mathbf{h}}$ at step $t$ given by

$$\mathbf{h_t} = f(\mathbf{W^r_{hh}} \mathbf{h_{t-1}} + \mathbf{W^r_{ih}} \mathbf{y^r_{r_t}}). \tag{2.1}$$

$\mathbf{W^r_{hh}} \in \mathbb{R}^{h \times h}$ and $\mathbf{W^r_{ih}} \in \mathbb{R}^{d \times h}$ are the parameters of the RNN. Here $r$ denotes the query relation. Path-RNN has a specialized model for predicting each query relation $r$, with separate parameters $(\mathbf{y^r_{r_t}}, \mathbf{W^r_{hh}}, \mathbf{W^r_{ih}})$ for each $r$. $f$ is the sigmoid function.

The vector representation of path $\pi$ ($\mathbf{y}_\pi$) is the last hidden state $\mathbf{h_k}$. The similarity of $\mathbf{y}_\pi$ with the query relation vector $\mathbf{y_r}$ is computed as the dot product between them:

$$\text{score}(\pi, \text{r}) = \mathbf{y}_\pi \cdot \mathbf{y_r} \tag{2.2}$$

### 2.2.1 Selection Mechanism

Unlike in previous work that use RNNs[114, 48, 47], a challenge with using them for our task is that among the set of paths connecting an entity pair, we do not observe which of the path(s) is predictive of a relation. We select the path that is closest to the relation type to be predicted in the vector space. This allows for faster training compared to marginalization. This technique has been successfully used in models other than RNNs previously [138, 82].

Let $\{s_1, s_2, \ldots, s_N\}$ be the similarity scores (Equation 2.2) for $N$ paths connecting an entity pair $(e_s, e_t)$. Path-RNN computes the probability that the entity pair $(e_s, e_t)$ participates in the query relation $(r)$ by,

$$\mathbb{P}(r|e_s, e_t) = \sigma(\max(s_1, s_2, \ldots, s_N)) \tag{2.3}$$

where $\sigma$ is the *sigmoid* function.

### 2.2.2 Model Training

We train the model using existing observed facts (triples) in the KB as positive examples and unobserved facts as negative examples. Let $\mathcal{R} = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ denote the set of all query relation types that we train for. Let $\Delta^+_\mathcal{R}, \Delta^-_\mathcal{R}$ denote the set of positive and negative triples for *all* the relation types in $\mathcal{R}$. The parameters of the model are trained to minimize the negative log-likelihood of the data.

$$L(\Theta, \Delta^+_{\mathcal{R}}, \Delta^-_{\mathcal{R}}) = -\frac{1}{M}\{ \sum_{e_s, e_t, r \in \Delta^+_{\mathcal{R}}} \log \mathbb{P}(r|e_s, e_t) + \sum_{\hat{e}_s, \hat{e}_t, \hat{r} \in \Delta^-_{\mathcal{R}}} \log(\ 1 - \mathbb{P}(\hat{r}|\hat{e}_s, \hat{e}_t))\} \tag{2.4}$$

Here $M$ is the total number of training examples and $\Theta$ denotes the set of all parameters of the model.

## 2.3 Zero-shot KB Completion

The KB completion task involves predicting facts on thousands of relations types and it is highly desirable that a method can infer facts about relation types without directly training for them. Given the vector representation of the relations, we show that our model described in the previous section is capable of predicting relational facts without explicitly training for the target (or test) relation types (zero-shot learning).

In zero-shot or zero-data learning [64, 91], some labels or classes are not available during training the model and only a description of those classes are given at prediction time. We make two modifications to the model described in the previous section, (1) learn a general RNN, and (2) fix relation vectors with pre-trained vectors, so that we can predict relations that are unseen during training. This ability of the model to generalize to unseen relations is beyond the capabilities of all previous methods for KB inference [107, 62, 33, 34].

We learn a general RNN for all relations instead of learning a separate RNN for every relation to be predicted. Here, the Path-RNN model combines all the *relations* in $\pi$ sequentially using a RNN with an intermediate representation $\mathbf{h_t} \in \mathbb{R}^{\mathbf{h}}$ at step $t$ given by

$$\mathbf{h_t} = f(\mathbf{W_{hh}}\mathbf{h_{t-1}} + \mathbf{W_{ih}}\mathbf{y_{r_t}}). \tag{2.5}$$

$\mathbf{W_{hh}} \in \mathbb{R}^{h \times h}$ and $\mathbf{W_{ih}} \in \mathbb{R}^{d \times h}$ are the parameters of the general RNN.

We initialize the vector representations of the binary relations using the representations learned in Riedel et al. [101] and do not update them during training. The relation vectors are not updated because at prediction time we would be predicting relation types which are never seen during training and hence their vectors would never get updated. We learn only the general RNN in this model. We train a single model for a set of relation types by replacing the sigmoid function with a softmax function while computing probabilities and the parameters of the RNN are learned using the available training data containing instances of few relations. The other aspects of the model remain unchanged. To predict facts whose relation types are unseen during training, we compute the vector representation of the path using the general RNN and compute the probability of the fact using the pre-trained relation vector.

## 2.4 Experiments

The hyperparameters of all the models were tuned on the same held-out development data. All the neural network models are trained for 150 iterations using 50 dimensional relation vectors, and we set the L2-regularizer and learning rate to 0.0001 and 0.1 respectively. We halved the learning rate after every 60 iterations and use mini-batches of size 20. The neural networks and the classifiers were optimized using AdaGrad [28].

### 2.4.1 Data

We ran experiments on Freebase [10] enriched with information from ClueWeb. We use the publicly available entity links to Freebase in the ClueWeb dataset [90]. Hence, we create nodes only for Freebase entities in our KB graph. We remove facts containing /type/object/type as they do not give useful predictive information for our task. We get triples from ClueWeb by considering sentences that contain two

| | |
|---|---|
| Entities | 18M |
| Freebase triples | 40M |
| ClueWeb triples | 12M |
| Relations | 25,994 |
| Relation types tested | 46 |
| Avg. paths/relation | 2.3M |
| Avg. training facts/relation | 6638 |
| Avg. positive test instances/relation | 3492 |
| Avg. negative test instances/relation | 43,160 |

Table 2.1: Statistics of our dataset.

entities linked to Freebase. We extract the phrase between the two entities and treat them as the relation types. For phrases that are of length greater than four we keep only the first and last two words. This helps us to avoid the time consuming step of dependency parsing the sentence to get the relation type. These triples are similar to facts obtained by OpenIE [5]. To reduce noise, we select relation types that occur at least 50 times. We evaluate on 46 relation types in Freebase that have the most number of instances. The methods are evaluated on a subset of facts in Freebase that were hidden during training. Table 2.1 shows important statistics of our dataset.

### 2.4.2 Predictive Paths

Table 2.2 shows predictive paths for 4 relations learned by the RNN model. The high quality of unseen paths is indicative of the fact that the RNN model is able to generalize to paths that are never seen during training.

### 2.4.3 Results

Using our dataset, we compare the performance of the following methods:

***PRA Classifier*** is the method in [63] which trains a logistic regression classifier by creating a feature for every path type.

***Cluster PRA Classifier*** is the method in [33] which replaces relation types from ClueWeb triples with their cluster membership in the KB graph before the path find-

| |
|---|
| **Relation:** /book/written_work/original_language/ *(book "x" written in language "y")*<br>**Seen paths:**<br>/book/written_work/previous_in_series → /book/written_work/author → /people/person/nationality<br>→ /people/person/nationality$^{-1}$ → /people/person/languages<br>/book/written_work/author → /people/ethnicity/people$^{-1}$ → /people/ethnicity/languages_spoken<br>**Unseen paths:**<br>"in"$^{-1}$ - "writer"$^{-1}$ → /people/person/nationality$^{-1}$ → /people/person/languages<br>/book/written_work/author → addresses → /people/person/nationality$^{-1}$ → /people/person/languages |
| **Relation:** /people/person/place_of_birth/ *(person "x" born in place "y")*<br>**Seen paths:**<br>"was,born,in" → /location/mailing_address/citytown$^{-1}$ → /location/mailing_address/state_province_region<br>"from" → /location/location/contains$^{-1}$<br>**Unseen paths:**<br>"born,in" → /location/location/contains → "near"$^{-1}$<br>"was,born,in" → commonly,known,as$^{-1}$ |
| **Relation:** /geography/river/cities/ *(river "x" flows through or borders "y")*<br>**Seen paths:**<br>"at" → /location/location/contains$^{-1}$<br>"meets,the" → /transportation/bridge/body_of_water_spanned$^{-1}$ → /location/location/contains$^{-1}$ → "in"<br>**Unseen paths:**<br>/geography/lake/outflow$^{-1}$ → /location/location/contains$^{-1}$<br>/geography/lake/outflow$^{-1}$ → /location/location/contains$^{-1}$ → "near" |
| **Relation:** /people/family/members/ *(person "y" part of family "x")*<br>**Seen paths:**<br>/royalty/monarch/royal_line$^{-1}$ → /people/person/children → /royalty/monarch/royal_line<br>→ /royalty/royal_line/monarchs_from_this_line<br>/royalty/royal_line/monarchs_from_this_line → /people/person/parents$^{-1}$<br>→ /people/person/parents$^{-1}$ → /people/person/parents$^{-1}$<br>**Unseen paths:**<br>/royalty/monarch/royal_line$^{-1}$ → "leader"$^{-1}$ → "king" → "was,married,to"$^{-1}$<br>"of,the"$^{-1}$ → "but,also,of" → "married" → "defended"$^{-1}$ |

Table 2.2: Predictive paths, according to the *RNN* model, for 4 target relations. Two examples of seen and unseen paths are shown for each target relation. Inverse relations are marked by $^{-1}$, i.e, $r(x,y) \implies r^{-1}(y,x), \forall(x,y) \in r$. Relations within quotes are OpenIE (textual) relation types.

ing step. After this step, their method proceeds in exactly the same manner as [63] training a logistic regression classifier by creating a feature for every path type. We use pre-trained relation vectors from [101] and use k-means clustering to cluster the relation types to 25 clusters as done in [33].

***Composition-Add*** uses a simple element-wise addition followed by sigmoid non-linearity as the composition function similar to [142].

***RNN-random*** is the supervised Path-RNN model with the relation vectors initialized randomly.

***RNN*** is the supervised Path-RNN model described in section 2.2 with the relation vectors initialized using the method in [101].

***PRA Classifier-b*** is our simple extension to the method in [63] which additionally uses bigrams in the path as features. We add a special *start* and *stop* symbol to the path before computing the bigram features.

***Cluster PRA Classifier-b*** is our simple extension to the method in [33] which additionally uses bigram features computed as previously described.

***RNN + PRA Classifier*** combines the predictions of *RNN* and *PRA Classifier*. We combine the predictions by assigning the score of a fact as the sum of their rank in the two models after sorting them in ascending order.

***RNN + PRA Classifier-b*** combines the predictions of *RNN* and *PRA Classifier-b* using the technique described previously.

Table 2.3 shows the results of our experiments. The method described in [34] is not included in the table since the publicly available implementation does not scale to our large dataset. First, we show that it is better to train the models using all the path types instead of using only the top $1,000$ path types as done in previous work [33, 34]. We can see that the RNN model performs significantly better than the baseline methods of [63] and [33]. The performance of the RNN model is not affected

| | train with top 1000 paths | train with all paths |
|---|---|---|
| Method | MAP | MAP |
| *PRA Classifier* | 43.46 | 51.31 |
| *Cluster PRA Classifier* | 46.26 | 53.23 |
| *Composition-Add* | 40.23 | 45.37 |
| *RNN-random* | 45.52 | 56.91 |
| *RNN* | 46.61 | 56.95 |
| *PRA Classifier-b* | 48.09 | 58.13 |
| *Cluster PRA Classifier-b* | 48.72 | 58.02 |
| *RNN + PRA Classifier* | 49.92 | 58.42 |
| *RNN + PRA Classifier-b* | 51.94 | **61.17** |

Table 2.3: Results comparing different methods on 46 types. All the methods perform better when trained using all the paths than training using the top $1,000$ paths. When training with all the paths, *RNN* performs significantly ($p < 0.005$) better than *PRA Classifier* and *Cluster PRA Classifier*. The small difference in performance between *RNN* and both *PRA Classifier-b* and *Cluster PRA Classifier-b* is not statistically significant. The best results are obtained by combining the predictions of *RNN* with *PRA Classifier-b* which performs significantly ($p < 10^{-5}$) better than both *PRA Classifier-b* and *Cluster PRA Classifier-b*.

by initialization since using random vectors and pre-trained vectors results in similar performance.

A surprising result is the impressive performance of our simple extension to the classifier approach. After the addition of bigram features, the naive PRA method is as effective as the Cluster PRA method. The small difference in performance between *RNN* and both *PRA Classifier-b* and *Cluster PRA Classifier-b* is not statistically significant. We conjecture that our method has substantially different strengths than the new baseline. While the classifier with bigram features has an ability to accurately memorize important local structure, the RNN model generalizes better to unseen paths that are very different from the paths seen is training. Empirically, combining the predictions of *RNN* and *PRA Classifier-b* achieves a statistically significant gain over *PRA Classifier-b*.

| | train with top 1000 paths | train with all paths |
|---|---|---|
| Method | MAP | MAP |
| RNN | 43.82 | 50.10 |
| zero-shot | 19.28 | 20.61 |
| Random | 7.59 | |

Table 2.4: Results comparing the zero-shot model with supervised RNN and a random baseline on 10 types. RNN is the fully supervised model described in section 2.2 while zero-shot is the model described in section 2.3. The zero-shot model without explicitly training for the target relation types achieves impressive results by performing significantly ($p < 0.05$) better than a random baseline.

#### 2.4.3.1 Zero-shot

Table 2.4 shows the results of the zero-shot model described in section 2.3 compared with the fully supervised RNN model (section 2.2) and a baseline that produces a random ordering of the test facts. We evaluate on randomly selected 10 (out of 46) relation types, hence for the fully supervised version we train 10 RNNs, one for each relation type. For evaluating the zero-shot model, we randomly split the relations into two sets of equal size and train a zero-shot model on one set and test on the other set. So, in this case we have two RNNs making predictions on relation types that they have never seen during training. As expected, the fully supervised RNN outperforms the zero-shot model by a large margin but the zero-shot model without using any direct supervision clearly performs much better than a random baseline.

## 2.5 Related Work

**KB Completion** includes methods such as [71], [143] and [7] that learn inference rules of length one. [107] learn general inference rules by considering the set of all paths in the KB and selecting paths that satisfy a certain precision threshold. Their method does not scale well to modern KBs and also depends on carefully tuned thresholds. [62] train a simple logistic regression classifier with NELL KB paths as features to perform KB completion while [33] and [34] extend it by using pre-trained

relation vectors to overcome feature sparsity. Recently, [142] learn inference rules using simple element-wise addition or multiplication as the composition function.

**Compositional Vector Space Models** have been developed to represent phrases and sentences in natural language as vectors [77, 6, 144]. Neural networks have been successfully used to learn vector representations of phrases using the vector representations of the words in that phrase. Recurrent neural networks have been used for many tasks such as language modeling [74], machine translation [124] and parsing [131]. Recursive neural networks, a more general version of the recurrent neural networks have been used for many tasks like parsing [114], sentiment classification [115, 118, 47], question answering [48] and natural language logical semantics [13]. Our overall approach is similar to RNNs with attention [3, 37] since we select a path among the set of paths connecting the entity pair to make the final prediction.

**Zero-shot or zero-data learning** was introduced in [64] for character recognition and drug discovery. [91] perform zero-shot learning for neural decoding while there has been plenty of work in this direction for image recognition [117, 31, 88].

# CHAPTER 3

# CHAINS OF REASONING OVER ENTITIES, RELATIONS, AND TEXT

We continue to improve neural network based logic reasoning in knowledge bases. We study three modeling advances: (1) Previously, our method reasons about chains of relations, but not the entities that form the nodes of the path. Now, we jointly learn and reason about relation-types, entities, and entity-types. (2) Here, we use new attention or selection mechanisms to reason about multiple paths instead of single paths. (3) The most problematic impracticality for application to KBs with broad semantics is their requirement to train a separate model for each relation-type to be predicted. In contrast, now, we train a single, high-capacity RNN that can predict all relation types. In addition to efficiency advantages, it significantly increases accuracy because the multi-task nature of the training shares strength in the common RNN parameters.

## 3.1 Modeling Approach

### 3.1.1 Shared Parameter Architecture

Path-RNN and other models such as the Path Ranking Algorithm (PRA) and its extensions [62, 63, 33, 34] makes it impractical to be used in downstream applications, since it requires training and maintaining a model for each relation type. Moreover, parameters are not shared across multiple target relation types leading to large number of parameters to be learned from the training data.

In response, now we *share* the relation type representation and the composition matrices of the RNN across all target relations enabling lesser number of parameters

for the same training data. We refer to this model as *Single-Model*. This is an example of *multi-task learning* [18] among prediction of target relation types with an underlying shared parameter architecture. The parameters are also shared in the zero-shot experiment described in the previous chapter. The RNN hidden state in (2.1) is now given by:

$$\mathbf{h_t} = f(\mathbf{W_{hh}}\mathbf{h_{t-1}} + \mathbf{W_{ih}}\mathbf{y_{r_t}}). \tag{3.1}$$

Readers should take note that the parameters here are independent of each target relation $r$.

### 3.1.2 Selection Mechanism

The Path-RNN model selects the maximum scoring path between an entity pair to make a prediction, possibly ignoring evidence from other important paths. Not only is this a waste of computation (since we have to compute a forward pass for all the paths anyway), but also the relations in all other paths do not get any gradients updates during training as the max operation returns zero gradient for all other paths except the maximum scoring one. This is especially ineffective during the initial stages of the training since the maximum probable path will be random.

Here, we introduce new methods of path selection (or attention) that takes into account multiple paths between an entity pair. Let $\{s_1, s_2, \ldots, s_N\}$ be the similarity scores (Equation 2.2) for $N$ paths connecting an entity pair $(e_s, e_t)$. The probability for entity pair $(e_s, e_t)$ to participate in relation $r$ (Equation 2.3) is now given by,

1. Top-($k$): A straightforward extension of the 'max' approach in which we average the top $k$ scoring paths. Let $\mathcal{K}$ denote the indices of top-$k$ scoring paths.

$$\mathbb{P}(r|e_s, e_t) = \sigma(\frac{1}{k}\sum_j s_j), \forall j \in \mathcal{K}$$

2. Average: Here, the final score is the average of scores of all the paths.

$$\mathbb{P}(r|e_s, e_t) = \sigma(\frac{1}{N}\sum_{i=1}^{N} s_i)$$

3. LogSumExp: In this approach the selection mechanism is a smooth approximation to the 'max' function - LogSumExp (LSE). Given a vector of scores, the LSE is calculated as

$$\text{LSE}(s_1, s_2, \ldots, s_N) = \log(\sum_i \exp(s_i))$$

and hence the probability of the triple is,

$$\mathbb{P}(r|e_1, e_2) = \sigma(\text{LSE}(s_1, s_2, \ldots, s_N))$$

The average and the LSE selection functions apply non-zero weights to *all* the paths during inference. However only a few paths between an entity pair are predictive of a query relation. LSE has another desirable property since $\frac{\partial \text{LSE}}{\partial s_i} = \frac{\exp(s_i)}{\sum_i \exp(s_i)}$. This means that during the back-propagation step, every path will receive a share of the gradient proportional to its score and hence this is a form of attention during the gradient step. In contrast, for averaging, every path will receive equal ($\frac{1}{N}$) share of the gradient. Top-($k$) (similar to max) receives sparse gradients.

### 3.1.3 Incorporating Entities

The Path-RNN model and other multi-hop relation extraction approaches (such as Guu et al. [41]) ignore the entities in the path. Consider the following paths JFK–*locatedIn*–NYC–*locatedIn*–NY and Yankee Stadium–*locatedIn*–NYC–*locatedIn*–NY. To predict the *airport_serves* relation, the Path-RNN model assigns the same scores to both the paths even though the first path should be ranked higher. This

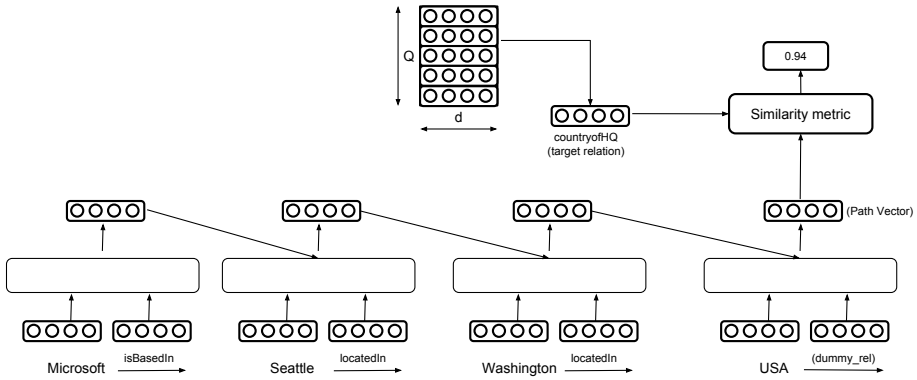



Figure 3.1: At each step, the RNN consumes both entity and relation vectors of the path. The entity representation can be obtained from its types. The path vector $\mathbf{y}_\pi$ is the last hidden state. The parameters of the RNN and relation embeddings are shared across all query relations. The dot product between the final representation of the path and the query relation gives a confidence score, with higher scores indicating that the query relation exists between the entity pair.

is because the model does not have information about the entities and just uses the relations in the path for prediction.

A straightforward way of incorporating entities is to include entity representations (along with relations) as input to the RNN. Learning separate representations of entity, however has some disadvantages. The distribution of entity occurrence is heavy tailed and hence it is hard to learn good representations of rarely occurring entities. To alleviate this problem, we use the entity types present in the KB as described below.

Most KBs have annotated types for entities and each entity can have multiple types. For example, Melinda Gates has types such as *CEO, Duke University Alumni, Philanthropist, American Citizen* etc. We obtain the entity representation by a simple addition of the entity type representations. The entity type representations are learned during training. We limit the number of entity types for an entity to 7 most frequently occurring types in the KB. Let $\mathbf{y_{e_t}} \in \mathbb{R}^m$ denote the representation of entity $e_t$, then 3.1 now becomes

$$\mathbf{h_t} = f(\mathbf{W_{hh}}\mathbf{h_{t-1}} + \mathbf{W_{ih}}\mathbf{y_{r_t}} + \mathbf{W_{eh}}\mathbf{y_{e_t}}) \tag{3.2}$$

$\mathbf{W_{eh}} \in \mathbb{R}^{m \times h}$ is the new parameter matrix for projecting the entity representation. Figure 3.1 shows our model with an example path between entities (*Microsoft, USA*) with *countryOfHQ* (country of head-quarters) as the query relation.

## 3.2 Related Work

Guu et al. [41] introduce new compositional techniques by modeling additive and multiplicative interactions between relation matrices in the path. However they model only a *single* path between an entity pair in-contrast to our ability to consider multiple paths. Toutanova et al. [126] improves upon them by additionally modeling the intermediate entities in the path and modeling multiple paths. However, in their approach they have to store scores for intermediate path length for *all* entity pairs, making it prohibitive to be used in our setting where we have more than 3M entity pairs. They also model entities as just a scalar weight whereas we learn both entity and type representations. Lastly as shown the previous chapter, non-linear composition function out-performs linear functions (as used by them) for relation extraction tasks. The performance of relation extraction methods have been improved by incorporating entity types for their candidate entities, both in sentence level [105, 113] and KB relation extraction [20], and in learning entailment rules [7].

## 3.3 Experiments

We continue to perform experiments on the same dataset as before. The dimension of the relation type representations and the RNN hidden states are $d, h = 250$ and the entity and type embeddings have $m = 50$ dimensions. We found rectifier units (ReLU) [66] to work much better than sigmoid units, even when compared to LSTMs (73.2 vs 72.4 in MAP). We use Adam [56] for optimization for all our experiments

with the default hyperparameter settings (learning rate = $1e^{-3}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e^{-8}$). Statistical significance for scores reported in Table 3.1 were done with a paired-$t$ test.

| Model | Performance (%MAP) | Selection |
|---|---|---|
| Single-Model | 68.77 | Max |
| Single-Model | 55.80 | Avg. |
| Single-Model | 68.20 | Top($k$) |
| Single-Model | **70.11** | LogSumExp |
| PRA | 64.43 | n/a |
| PRA + Bigram | 64.93 | n/a |
| Path-RNN | 65.23 | Max |
| Path-RNN | 68.43 | LogSumExp |
| Single-Model | **70.11** | LogSumExp |
| PRA + Types | 64.18 | n/a |
| Single-Model | 70.11 | LogSumExp |
| Single-Model + Entity | 71.74 | LogSumExp |
| Single-Model + Types | **73.26** | LogSumExp |
| Single-Model + Entity + Types | 72.22 | LogSumExp |

Table 3.1: The first section shows the effectiveness of LogSumExp as the score aggregation function. The next section compares performance with existing multi-hop approaches and the last section shows the performance achieved using joint reasoning with entities and types.

### 3.3.1 Effect of Selection Techniques

Section 1 of Table 3.1 shows the effect of the various selection techniques presented in section 3.1.2. *LogSumExp* gives the best results. This demonstrates the importance of considering information from all the paths. However, Avg. selection method performs the worst, which shows that it is also important to weigh the paths scores according to their values. Figure 3.2 plots the training loss w.r.t gradient update step. Due to non-zero gradient updates for all the paths, the LogSumExp selection strategy leads to faster training vs. max selection, which has sparse gradients. This is especially relevant during the early stages of training, where the argmax path is

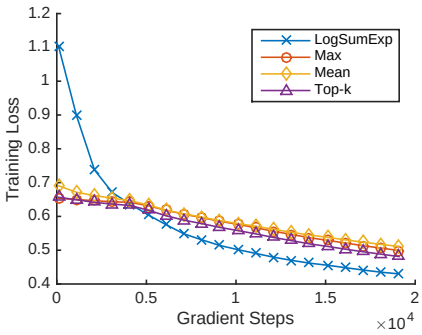



Figure 3.2: Comparison of the training loss w.r.t gradient update steps of various selection methods. The loss of LogSumExp decreases the fastest among all selection methods and hence leads to faster training.

essentially a random guess. The scores of max and LSE selection are significant with ($p < 0.02$).

### 3.3.2 Comparison with multi-hop models

We next compare the performance of the Single-Model with two other multi-hop models - Path-RNN and PRA[62]. Both of these approaches train an individual model for each query relation. We also experiment with another extension of PRA that adds bigram features (PRA + Bigram). Additionally, we run an experiment replacing the max-selection of Path-RNN with LogSumExp. The results are shown in the second section of Table 3.1. It is not surprising to see that the Single-Model, which leverages parameter sharing improves performance. It is also encouraging to see that LogSumExp makes the Path-RNN baseline stronger. The scores of Path-RNN (with LSE) and Single-Model are significant with ($p < 0.005$).

### 3.3.3 Effect of Incorporating Entities

Next, we provide quantitative results supporting our claim that modeling the entities along a KB path can improve reasoning performance. The last section of Table 3.1 lists the performance gain obtained by injecting information about entities. We achieve the best performance when we represent entities as a function of their

annotated types in Freebase (Single-Model + Types) ($p < 0.005$). In comparison, learning separate representations of entities (Single-Model + Entities) gives slightly worse performance. This is primarily because we encounter many new entities during test time, for which our model does not have a learned representation. However the relatively limited number of entity types helps us overcome the problem of representing unseen entities. We also extend PRA to include entity type information (PRA + Types), but this did not yield significant improvements.

### 3.3.4 Performance in Limited Data Regime

In constructing our dataset, we selected query relations with reasonable amounts of data. However, for many important applications we have very limited data. To simulate this common scenario, we create a new dataset by randomly selecting 23 out of 46 relations and removing all but 1% of the positive and negative triples previously used for training.

Effectively, the difference between Path-RNN and Single-Model is that Single-Model does multitask learning, since it shares parameters for different target relation types. Therefore, we expect it to outperform Path-RNN on this small dataset, since this multitask learning provides additional regularization. We also experiment with an extension of Single-Model where we introduce an additional task for multitask learning, where we seek to predict annotated types for entities. Here, parameters for the entity type embeddings are shared with the Single-Model. Supervision for this task is provided by the entity type annotation in the KB. We train with a Bayesian Personalized Ranking loss of Rendle et al. [99]. The results are listed in Table 3.2. With Single-Model there is a clear jump in performance as we expect. The additional multitask training with types gives a very small gain in performance.

The experiments described in the two chapters clearly show than RNNs with the appropriate path selection mechanism perform significantly better than sym-

| Model | Performance (%MAP) |
|---|---|
| Path-RNN | 22.06 |
| Single-Model | 63.33 |
| Single-Model + MTL | 64.81 |

Table 3.2: Model performance when trained with a small fraction of the data.

bolic methods. Additionally, they can perform zero-shot reasoning which is particularly useful when employed in large KBs. The code and data are available at https://rajarshd.github.io/ChainsofReasoning/

### 3.3.5 Human Performance

We conduct a study to measure the difference between human performance and the performance of the best method from the experiments described above. The task of labeling whether an entity pair participates in a target relation given only the paths connecting them is extremely challenging for humans. Each entity pair has many paths connecting them and the entities in the path are replaced by their types. Moreover, understanding a path requires good knowledge of the Freebase schema. So, labeling a single example takes considerable amount of time.

While we are good at labeling examples with binary labels, the methods are good at producing a ranking. To reconcile these differences we adopt the following strategy: 1) For a query relation, we sample 10 entity pairs (some are positive while others are negative examples) 2) Each example is labeled as positive or negative. Then, we measure accuracy among the labeled positive examples and this is used as the human performance on the task. 3) Using the model's predictions, we compute top-k accuracy, where k is the number of positive labels in the previous step. 4) We can use the computed top-k accuracies to compare the two systems.

Table 3.3 compares the top-k accuracy for humans vs Single-Model + Types for 4 query relations. For the first relation we use $k = 7$, $k = 4$ for the next two and $k = 5$

| Query Relation | Human Performance | Model |
|---|---|---|
| /architecture/structure/address | 0.57 | 0.71 |
| /book/written_work/original_language | 0.75 | 1.0 |
| /broadcast/content/genre | 0.75 | 1.0 |
| /people/person/religion | 0.80 | 0.80 |

Table 3.3: Top-k accuracy of humans Vs Single-Model + Types.

for the last one. Overall, I correctly label $\frac{14}{20}$ while the model gets $\frac{17}{20}$ correct. This study shows that the performance of the model is comparable to human performance.

# CHAPTER 4

# NEURAL PROGRAMMER

The KB reasoning task considered in the previous chapters involves simple reasoning on large structured data. Now, we consider a task that involves mapping inputs to latent programs to produce the required output. By inducing programs, we can now support more sophisticated reasoning. While the methods we develop our fairly general, we consider the task of mapping questions to latent programs in the context of question answering. For example we need to answer the question "how many states border Texas?" (see Zettlemoyer & Collins [150]), given a database of states and its borders. A common method for solving these problems is *program induction* where the goal is to find a program (in SQL or some high-level languages) that can correctly solve the task. An application of these models is in *semantic parsing* where the task is to build a natural language interface to a structured database [148].

A drawback of existing methods in semantic parsing is that they are difficult to train and require a great deal of human supervision. As the space over programs is non-smooth, it is difficult to apply simple gradient descent; most often, gradient descent is augmented with a complex search procedure, such as sampling [69]. To further simplify training, the algorithmic designers have to manually add more supervision signals to the models in the form of annotation of the complete program for every question [150] or a domain-specific grammar [70]. For example, designing grammars that contain rules to associate lexical items to the correct operations, e.g., the word "largest" to the operation "argmax", or to produce syntactically valid programs, e.g., disallow the program $>= dog$. The role of hand-crafted grammars is crucial in

semantic parsing yet also limits its general applicability to many different domains. In a recent work by Wang et al. [134] to build semantic parsers for 7 domains, the authors hand engineer a separate grammar for each domain.

The goal of this work is to develop a DNN model that does not require substantial human supervision and is broadly applicable across different domains, data sources and natural languages. The past few years have seen the tremendous success of deep neural networks (DNNs) in a variety of supervised classification tasks starting with image recognition [58] and speech recognition [44] where the DNNs act on a fixed-length input and output. More recently, this success has been translated into applications that involve a variable-length sequence as input and/or output such as machine translation [124, 3, 73], image captioning [132, 141], conversational modeling [108, 129], end-to-end Q&A [123, 93, 43], and end-to-end speech recognition [39, 42, 19, 4]. While these results strongly indicate that DNN models are capable of learning the fuzzy underlying patterns in the data, they have not had similar impact in applications that involve discrete or crisp reasoning. A major limitation of these models is in their inability to learn even simple arithmetic and logic operations. For example, Joulin & Mikolov [52] show that recurrent neural networks (RNNs) fail at the task of adding two binary numbers even when the result has less than 10 bits. This makes existing DNN models unsuitable for the task under consideration. For example, to answer the question "how many states border Texas?", the algorithm has to perform an act of counting in a table which is something that a neural network is not yet good at.

We propose *Neural Programmer* (Figure 4.1), a neural network augmented with a small set of basic arithmetic and logic operations that can be trained end-to-end using backpropagation. In our formulation, the neural network can run several steps using a recurrent neural network. At each step, it can select a segment in the data source and a particular operation to apply to that segment. The neural network propagates

these outputs forward at every step to form the final, more complicated output. Using the target output, we can adjust the network to select the right data segments and operations, thereby inducing the correct program. Key to our approach is that the selection process (for the data source and operations) is done in a differentiable fashion (i.e., soft selection or attention), so that the whole neural network can be trained jointly by gradient descent. At test time, we replace soft selection with hard selection.

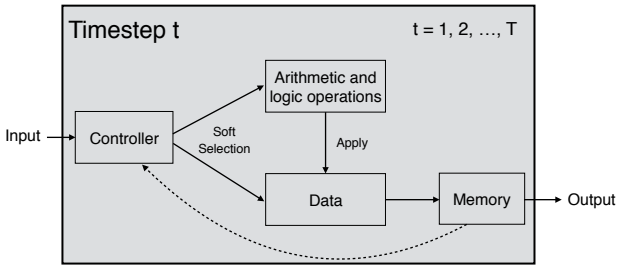



Figure 4.1: The architecture of Neural Programmer, a neural network augmented with arithmetic and logic operations. The controller selects the operation and the data segment. The memory stores the output of the operations applied to the data segments and the previous actions taken by the controller. The controller runs for several steps thereby inducing compositional programs that are more complex than the built-in operations. The dotted line indicates that the controller uses information in the memory to make decisions in the next time step.

By combining neural network with mathematical operations, we can utilize both the fuzzy pattern matching capabilities of deep networks and the crisp algorithmic power of traditional programmable computers. Our work is inspired by the success of the soft attention mechanism [3] and its application in learning a neural network to control an additional memory component [40, 123].

Neural Programmer has two attractive properties. First, it learns from a weak supervision signal which is the result of execution of the correct program. It does not require the expensive annotation of the correct program for the training examples. The human supervision effort is in the form of question, data source and answer triples. Second, Neural Programmer does not require additional rules to guide the

program search, making it a general framework. With Neural Programmer, the algorithmic designer only defines a list of basic operations which requires lesser human effort than in previous program induction techniques.

## 4.1 Model

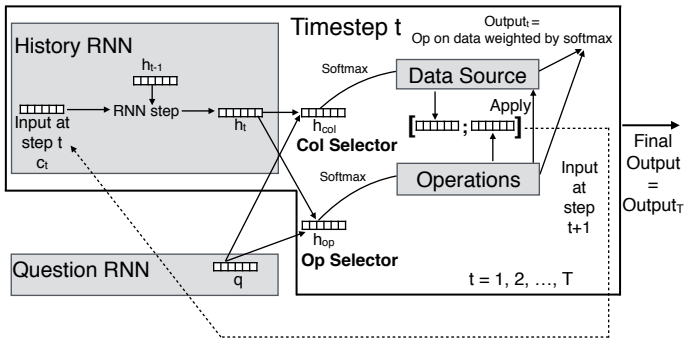
Even though our model is quite general, we first apply Neural Programmer to the task of question answering on tables, a task that has not been previously attempted by neural networks. In our implementation for this task, Neural Programmer is run for a total of $T$ time steps chosen in advance to induce compositional programs of up to $T$ operations. The model consists of four modules:

- A question Recurrent Neural Network (RNN) to process the input question,
- A selector to assign two probability distributions at every step, one over the set of operations and the other over the data segments,
- A list of operations that the model can apply and,
- A history RNN to remember the previous operations and data segments selected by the model till the current time step.

These four modules are also shown in Figure 4.2. The history RNN combined with the selector module functions as the controller in this case. Information about each component is discussed in the next sections.

Apart from the list of operations, all the other modules are learned using gradient descent on a training set consisting of triples, where each triple has a question, a data source and an answer. We assume that the data source is in the form of a table, $table \in \mathbb{R}^{M \times C}$, containing $M$ rows and $C$ columns ($M$ and $C$ can vary amongst examples). The data segments in our experiments are the columns, where each column also has a column name.

Figure 4.2: An implementation of Neural Programmer for the task of question answering on tables. The output of the model at time step $t$ is obtained by applying the operations on the data segments weighted by their probabilities. The final output of the model is the output at time step $T$. The dotted line indicates the input to the history RNN at step t+1.




### 4.1.1 Question Module

The question module converts the question tokens to a distributed representation. In the basic version of our model, we use a simple RNN [137] parameterized by $W^{question}$ and the last hidden state of the RNN is used as the question representation (Figure 4.3).

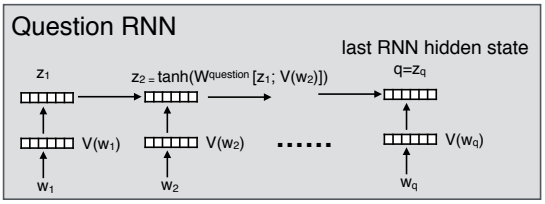



Figure 4.3: The question module to process the input question. $q = z_q$ denotes the question representation used by Neural Programmer.

Consider an input question containing $Q$ words $\{w_1, w_2, \ldots, w_Q\}$, the question module performs the following computations:

$$z_i = \tanh(W^{question}[z_{i-1}; V(w_i)]), \forall i = 1, 2, \ldots, Q$$

where $V(w_i) \in \mathbb{R}^d$ represents the embedded representation of the word $w_i$, $[a; b] \in \mathbb{R}^{2d}$ represents the concatenation of two vectors $a, b \in \mathbb{R}^d$, $W^{question} \in \mathbb{R}^{d \times 2d}$ is the recurrent matrix of the question RNN, tanh is the element-wise non-linearity function and $z_Q \in \mathbb{R}^d$ is the representation of the question. We set $z_0$ to $[0]^d$. We pre-process the question by removing numbers from it and storing the numbers in a separate list. Along with the numbers we store the word that appeared to the left of it in the question which is useful to compute the pivot values for the comparison operations described in Section 4.1.3.

For tasks that involve longer questions, we use a bidirectional RNN since we find that a simple unidirectional RNN has trouble remembering the beginning of the question. When the bidirectional RNN is used, the question representation is obtained by concatenating the last hidden states of the two-ends of the bidirectional RNNs. The question representation is denoted by $q$.

### 4.1.2 Selector

The selector produces two probability distributions at every time step $t$ ($t = 1, 2, \ldots, T$): one probablity distribution over the set of operations and another probability distribution over the set of columns. The inputs to the selector are the question representation ($q \in \mathbb{R}^d$) from the question module and the output of the history RNN (described in Section 4.1.4) at time step $t$ ($h_t \in \mathbb{R}^d$) which stores information about the operations and columns selected by the model up to the previous step.

Each operation is represented using a $d$-dimensional vector. Let the number of operations be $O$ and let $U \in \mathbb{R}^{O \times D}$ be the matrix storing the representations of the operations.

**Operation Selection** is performed by:

$$\alpha_t^{op} = softmax(U \tanh(W^{op}[q; h_t]))$$

where $W^{op} \in \mathbb{R}^{d \times 2d}$ is the parameter matrix of the operation selector that produces the probability distribution $\alpha_t^{op} \in [0,1]^O$ over the set of operations (Figure 4.4).

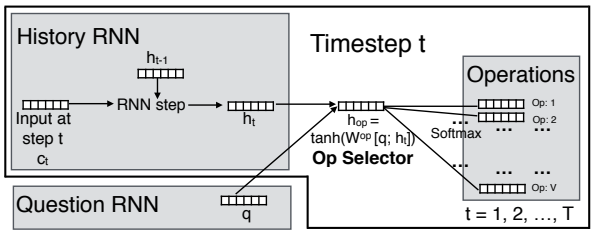



Figure 4.4: Operation selection at time step $t$ where the selector assigns a probability distribution over the set of operations.

The selector also produces a probability distribution over the columns at every time step. We obtain vector representations for the column names using the parameters in the question module (Section 4.1.1) by word embedding or an RNN phrase embedding. Let $P \in \mathbb{R}^{C \times d}$ be the matrix storing the representations of the column names.

**Data Selection** is performed by:

$$\alpha_t^{col} = softmax(P \tanh(W^{col}[q; h_t]))$$

where $W^{col} \in \mathbb{R}^{d \times 2d}$ is the parameter matrix of the column selector that produces the probability distribution $\alpha_t^{col} \in [0,1]^C$ over the set of columns (Figure 4.5).

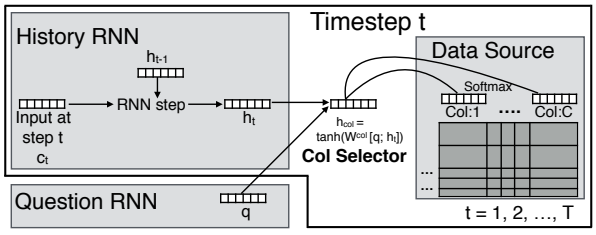



Figure 4.5: Data selection at time step $t$ where the selector assigns a probability distribution over the set of columns.

### 4.1.3 Operations

Neural Programmer currently supports two types of outputs: a) a scalar output, and b) a list of items selected from the table (i.e., table lookup).[1] The first type of output is for questions of type "Sum of elements in column C" while the second type of output is for questions of type "Print elements in column A that are greater than 50." To facilitate this, the model maintains two kinds of output variables at every step $t$, $scalar\_answer_t \in \mathbb{R}$ and $lookup\_answer_t \in [0,1]^{M \times C}$. The output $lookup\_answer_t(i,j)$ stores the probability that the element $(i,j)$ in the table is part of the output. The final output of the model is $scalar\_answer_T$ or $lookup\_answer_T$ depending on whichever of the two is updated after $T$ time steps. Apart from the two output variables, the model maintains an additional variable $row\_select_t \in [0,1]^M$ that is updated at every time step. The variables $row\_select_t[i](\forall i = 1, 2, \ldots, M)$ maintain the probability of selecting row $i$ and allows the model to dynamically select a subset of rows within a column. The output is initialized to zero while the *row_select* variable is initialized to $[1]^M$.

Key to Neural Programmer is the built-in operations, which have access to the outputs of the model at every time step before the current time step $t$, i.e., the operations have access to $(scalar\_answer_i, lookup\_answer_i), \forall i = 1, 2, \ldots, t-1$. This enables the model to build powerful compositional programs.

It is important to design the operations such that they can work with probabilistic row and column selection so that the model is differentiable. Table 4.1 shows the list of operations built into the model along with their definitions. The reset operation can be selected any number of times which when required allows the model to induce programs whose complexity is less than $T$ steps.

---

[1] It is trivial to extend the model to support general text responses by adding a decoder RNN to generate text sentences.

| Type | Operation | Definition |
|---|---|---|
| Aggregate | Sum | $sum_t[j] = \sum_{i=1}^{M} row\_select_{t-1}[i] * table[i][j], \forall j = 1, 2, \ldots, C$ |
| | Count | $count_t = \sum_{i=1}^{M} row\_select_{t-1}[i]$ |
| Arithmetic | Difference | $diff_t = scalar\_output_{t-3} - scalar\_output_{t-1}$ |
| Comparison | Greater | $g_t[i][j] = table[i][j] > pivot_g, \forall (i,j), i = 1, \ldots, M, j = 1, \ldots, C$ |
| | Lesser | $l_t[i][j] = table[i][j] < pivot_l, \forall (i,j), i = 1, \ldots, M, j = 1, \ldots, C$ |
| Logic | And | $and_t[i] = min(row\_select_{t-1}[i], row\_select_{t-2}[i]), \forall i = 1, 2, \ldots, M$ |
| | Or | $or_t[i] = max(row\_select_{t-1}[i], row\_select_{t-2}[i]), \forall i = 1, 2, \ldots, M$ |
| Assign Lookup | assign | $assign_t[i][j] = row\_select_{t-1}[i], \forall (i,j) i = 1, 2, \ldots, M, j = 1, 2, \ldots, C$ |
| Reset | Reset | $reset_t[i] = 1, \forall i = 1, 2, \ldots, M$ |

Table 4.1: List of operations along with their definitions at time step $t$, $table \in \mathbb{R}^{M \times C}$ is the data source in the form of a table and $row\_select_t \in [0,1]^M$ functions as a row selector.

While the definitions of the operations are fairly straightforward, comparison operations greater and lesser require a pivot value as input (refer Table 4.1), which appears in the question. Let $qn_1, qn_2, \ldots, qn_N$ be the numbers that appear in the question.

For every comparison operation (greater and lesser), we compute its pivot value by adding up all the numbers in the question each of them weighted with the probabilities assigned to it computed using the hidden vector at position to the left of the number,[2] and the operation's embedding vector. More precisely:

$$\beta_{op} = softmax(ZU(op))$$

$$pivot_{op} = \sum_{i=1}^{N} \beta_{op}(i) qn_i$$

where $U(op) \in \mathbb{R}^d$ is the vector representation of operation $op$ ($op \in \{\text{greater}, \text{lesser}\}$) and $Z \in \mathbb{R}^{N \times d}$ is the matrix storing the hidden vectors of the question RNN at positions to the left of the occurrence of the numbers.

[2]This choice is made to reflect the common case in English where the pivot number is usually mentioned after the operation but it is trivial to extend to use hidden vectors both in the left and the right of the number.

By overloading the definition of $\alpha_t^{op}$ and $\alpha_t^{col}$, let $\alpha_t^{op}(x)$ and $\alpha_t^{col}(j)$ denote the probability assigned by the selector to operation $x$ ($x \in$ {sum, count, difference, greater, lesser, and, or, assign, reset}) and column $j$ ($\forall j = 1, 2, \ldots, C$) at time step $t$ respectively.

Figure 4.6 show how the output and row selector variables are computed. The output and row selector variables at a step is obtained by additively combining the output of the individual operations on the different data segments weighted with their corresponding probabilities assigned by the model.

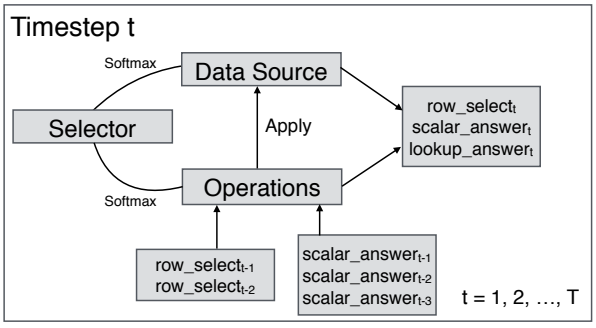



Figure 4.6: The output and row selector variables are obtained by applying the operations on the data segments and additively combining their outputs weighted using the probabilities assigned by the selector.

More formally, the output variables are given by:

$$scalar\_answer_t = \alpha_t^{op}(\text{count})\, count_t + \alpha_t^{op}(\text{difference})\, diff_t + \sum_{j=1}^{C} \alpha_t^{col}(j)\alpha_t^{op}(\text{sum})\, sum_t[j],$$

$$lookup\_answer_t[i][j] = \alpha_t^{col}(j)\alpha_t^{op}(\text{assign})\, assign_t[i][j], \forall (i,j) i = 1, 2, \ldots, M, j = 1, 2, \ldots, C$$

The row selector variable is given by:

$$row\_select_t[i] = \alpha_t^{op}(\text{and})\, and_t[i] + \alpha_t^{op}(\text{or})\, or_t[i] + \alpha_t^{op}(\text{reset})\, reset_t[i] + \sum_{j=1}^{C} \alpha_t^{col}(j)(\alpha_t^{op}(\text{greater})\, g_t[i][j] + \alpha_t^{op}(\text{lesser})\, l_t[i][j]), \forall i = 1, \ldots, M$$

It is important to note that other operations like *equal to, max, min, not* etc. can be built into this model easily.

#### 4.1.3.1 Handling Text Entries

So far, our disscusion has been only concerned with tables that have numeric entries. Now, we describe how Neural Programmer handles text entries in the input table. We assume a column can contain either numeric or text entries. An example query is "what is the sum of elements in column B whose field in column C is word:1 and field in column A is word:7?". In other words, the query is looking for text entries in the column that match specified words in the questions. To answer these queries, we add a text match operation that updates the row selector variable appropriately. In our implementation, the parameters for vector representations of the column's text entries are shared with the question module.

The text match operation uses a two-stage soft attention mechanism, back and forth from the text entries to question module. In the following, we explain its implementation in detail.

Let $TC_1, TC_2, \ldots, TC_K$ be the set of columns that each have $M$ text entries and $A \in M \times K \times d$ store the vector representations of the text entries. In the first stage, the question representation coarsely selects the appropriate text entries through the sigmoid operation. Concretely, coarse selection, $B$, is given by the sigmoid of dot product between vector representations for text entries, $A$, and question representation, $q$:

$$B[m][k] = sigmoid\left(\sum_{p=1}^{d} A[m][k][p] \cdot q[p]\right) \quad \forall(m,k) \;\; m = 1, \ldots, M, k = 1, \ldots, K$$

To obtain question-specific column representations, $D$, we use $B$ as weighting factors to compute the weighted average of the vector representations of the text entries in that column:

$$D[k][p] = \frac{1}{M} \sum_{m=1}^{M} (B[m][k] \cdot A[m][k][p]) \quad \forall (k,p) \;\; k = 1, \ldots, K, p = 1, \ldots, d$$

To allow different words in the question to be matched to the corresponding columns (e.g., match word:1 in column C and match word:7 in column A for question "what is the sum of elements in column B whose field in column C is word:1 and field in column A is word:7?'), we add the column name representations (described in Section 4.1.2), $P$, to $D$ to obtain column representations $E$. This make the representation also sensitive to the column name.

In the second stage, we use $E$ to compute an attention over the hidden states of the question RNN to get attention vector $G$ for each column of the input table. More concretely, we compute the dot product between $E$ and the hidden states of the question RNN to obtain scalar values. We then pass them through softmax to obtain weighting factors for each hidden state. $G$ is the weighted combination of the hidden states of the question RNN.

Finally, text match selection is done by:

$$text\_match[m][k] = sigmoid \left( \sum_{p=1}^{d} A[m][k][p] \cdot G[k][p] \right)$$
$$\forall (m,k) \;\; m = 1, \ldots, M, k = 1, \ldots, K$$

Without loss of generality, let the first $K$ ($K \in [0, 1, \ldots, C]$) columns out of $C$ columns of the table contain text entries while the remaining contain numeric entries. The row selector variable now is given by:

$$row\_select_t[i] = \alpha_t^{op}(\text{and})\, and_t[i] + \alpha_t^{op}(\text{or})\, or_t[i] + \alpha_t^{op}(\text{reset})\, reset_t[i] +$$
$$\sum_{j=K+1}^{C} \alpha_t^{col}(j)(\alpha_t^{op}(\text{greater})\, g_t[i][j] + \alpha_t^{op}(\text{lesser})\, l_t[i][j]) +$$
$$\sum_{j=1}^{K} \alpha_t^{col}(j)(\alpha_t^{op}(\text{text\_match})\, text\_match_t[i][j], \forall i = 1, \ldots, M$$

The two-stage mechanism is required since in our experiments we find that simply averaging the vector representations fails to make the representation of the column specific enough to the question. Unless otherwise stated, our experiments are with input tables whose entries are only numeric and in that case the model does not contain the text match operation.

### 4.1.4 History RNN

The history RNN keeps track of the previous operations and columns selected by the selector module so that the model can induce compositional programs. This information is encoded in the hidden vector of the history RNN at time step $t$, $h_t \in \mathbb{R}^d$. This helps the selector module to induce the probability distributions over the operations and columns by taking into account the previous actions selected by the model. Figure 4.7 shows details of this component.

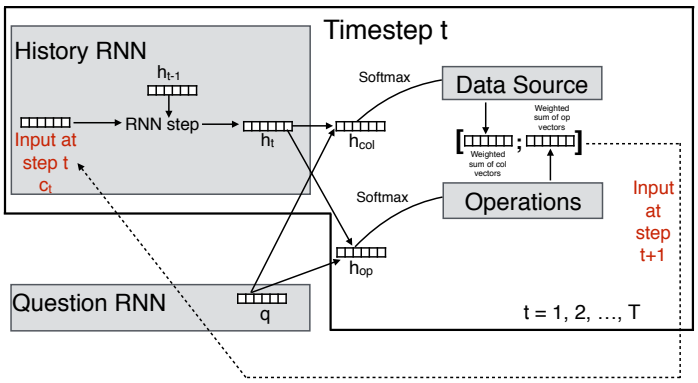



Figure 4.7: The history RNN which helps in remembering the previous operations and data segments selected by the model. The dotted line indicates the input to the history RNN at step t+1.

The input to the history RNN at time step $t$, $c_t \in \mathbb{R}^{2d}$ is obtained by concatenating the weighted representations of operations and column names with their corresponding probability distribution produced by the selector at step $t-1$. More precisely:

$$c_t = [(\alpha_{t-1}^{op})^T U; (\alpha_{t-1}^{col})^T P]$$

The hidden state of the history RNN at step $t$ is computed as:

$$h_t = \tanh(W^{history}[c_t; h_{t-1}]), \forall i = 1, 2, \ldots, Q$$

where $W^{history} \in \mathbb{R}^{d \times 3d}$ is the recurrent matrix of the history RNN, and $h_t \in \mathbb{R}^d$ is the current representation of the history. The history vector at time $t = 1$, $h_1$ is set to $[0]^d$.

## 4.2 Training Objective

The parameters of the model include the parameters of the question RNN, $W^{question}$, parameters of the history RNN, $W^{history}$, word embeddings $V(.)$, operation embeddings $U$, operation selector and column selector matrices, $W^{op}$ and $W^{col}$ respectively. During training, depending on whether the answer is a scalar or a lookup from the table we have two different loss functions.

When the answer is a scalar, we use Huber loss [46] given by:

$$L_{scalar}(scalar\_answer_T, y) = \begin{cases} \frac{1}{2}a^2, \text{if } a \leq \delta \\ \delta a - \frac{1}{2}\delta^2, \text{otherwise} \end{cases}$$

where $a = |scalar\_answer_T - y|$ is the absolute difference between the predicted and true answer, and $\delta$ is the Huber constant treated as a model hyper-parameter. In our

experiments, we find that using square loss makes training unstable while using the absolute loss makes the optimization difficult near the non-differentiable point.

When the answer is a list of items selected from the table, we convert the answer to $y \in \{0,1\}^{M \times C}$, where $y[i,j]$ indicates whether the element $(i,j)$ is part of the output. In this case we use log-loss over the set of elements in the table given by:

$$L_{lookup}(lookup\_answer_T, y) = -\frac{1}{MC}\sum_{i=1}^{M}\sum_{j=1}^{C}\Big(y[i,j]\log(lookup\_answer_T[i,j]) + (1-y[i,j])\log(1-lookup\_answer_T[i,j])\Big)$$

The training objective of the model is given by:

$$L = \frac{1}{N}\sum_{k=1}^{N}\Big([n_k == True]L_{scalar}^{(k)} + [n_k == False]\lambda L_{lookup}^{(k)}\Big)$$

where $N$ is the number of training examples, $L_{scalar}^{(k)}$ and $L_{lookup}^{(k)}$ are the scalar and lookup loss on $k^{th}$ example, $n_k$ is a boolean random variable which is set to *True* when the $k^{th}$ example's answer is a scalar and set to *False* when the answer is a lookup, and $\lambda$ is a hyper-parameter of the model that allows to weight the two loss functions appropriately.

## 4.3 Prediction Procedure

At prediction time, we replace the three softmax layers in the model with the conventional maximum (hardmax) operation and the final output of the model is either $scalar\_answer_T$ or $lookup\_answer_T$, depending on whichever among them is updated after $T$ time steps. Algorithm 1 gives a high-level view of Neural Programmer at prediction time.

**Algorithm 1** High-level view of Neural Programmer at prediction time for an input example.

**Input**: *table* $\in \mathbb{R}^{M \times C}$ and *question*
**Initialize**: $scalar\_answer_0 = 0$, $lookup\_answer_0 = 0^{M \times C}$, $row\_select_0 = 1^M$, history vector at time $t = 0$, $h_0 = 0^d$ and input to history RNN at time $t = 0$, $c_0 = 0^{2d}$
**Preprocessing**: Remove numbers from *question* and store them in a list along with the words that appear to the left of it. The tokens in the input question are $\{w_1, w_2, \ldots, w_Q\}$.
**Question Module**: Run question RNN on the preprocessed question to get question representation $q$ and list of hidden states $z_1, z_2, \ldots, z_Q$
**Pivot numbers**: $pivot_g$ and $pivot_l$ are computed using hidden states from question RNN and operation representations $U$
**for** $t = 1, 2, \ldots, T$ **do**
    Compute history vector $h_t$ by passing input $c_t$ to the history RNN
    **Operation selection** using $q$, $h_t$ and operation representations $U$
    **Data selection** on *table* using $q$, $h_t$ and column representations $V$
    Update $scalar\_answer_t$, $lookup\_answer_t$ and $row\_select_t$ using the selected operation and column
    Compute input to the history RNN at time $t + 1$, $c_{t+1}$
**end for**
**Output**: $scalar\_answer_T$ or $lookup\_answer_T$ depending on whichever of the two is updated at step $T$

## 4.4 Related Work

Program induction has been studied in the context of semantic parsing [148, 150, 70] in natural language processing. Pasupat & Liang [92] develop a semantic parser with a hand engineered grammar for question answering on tables with natural language questions. Methods such as Piantadosi et al. [94], Eisenstein et al. [29], Clarke et al. [22] learn a compositional semantic model without hand engineered compositional grammar, but still requiring a hand labeled lexical mapping of words to the operations. Poon [95] develop an unsupervised method for semantic parsing, which requires many pre-processing steps including dependency parsing and mapping from words to operations. Liang et al. [69] propose an hierarchical Bayesian approach to learn simple programs.

There has been some early work in using neural networks for learning context free grammar [25, 26, 149] and context sensitive grammar [121, 36] for small problems. Neelakantan et al. [83], Lin et al. [72] learn simple Horn clauses in a large knowledge base using RNNs. Neural networks have also been used for Q&A on datasets that do not require complicated arithmetic and logic reasoning [12, 48, 123, 93, 43]. While there has been lot of work in augmenting neural networks with additional memory [25, 106, 45, 40, 139, 59, 52], we are not aware of any other work that augments a neural network with a set of operations to enhance complex reasoning capabilities.

After our work was submitted to ArXiv, many neural network models have been developed for program induction [2, 51, 98, 147, 146], despite the notorious difficulty of handling discrete operations in neural networks [52, 54]. Most of these approaches rely on complete program as supervision [51, 98] while others [147, 146] have been tried only on synthetic tasks. Recently, Kocisky et al. [57] develop a semi-supervised semantic parsing method that uses question-program pairs as supervision.

# CHAPTER 5

# SYNTHETIC DATA EXPERIMENTS WITH NEURAL PROGRAMMER

Neural Programmer described in the previous chapter is faced with many challenges, specifically: 1) can the model learn the parameters of the different modules with delayed supervision after $T$ steps? 2) can it exhibit compositionality by generalizing to unseen questions? and 3) can the question module handle the variability and ambiguity of natural language? In our first experiments, we mainly focus on answering the first two questions using synthetic data. Our reason for using synthetic data is that it is easier to understand a new model with a synthetic dataset. We can generate the data in a large quantity, whereas the biggest real-word semantic parsing datasets we know of contains only about 14k training examples [92] which is very small by neural network standards. In one of our experiments, we introduce simple word-level variability to simulate one aspect of the difficulties in dealing with natural language input.

## 5.1 Data

We generate question, table and answer triples using a synthetic grammar. Tables 5.1 and 5.2 shows examples of question templates from the synthetic grammar for single and multiple columns respectively. The elements in the table are uniformly randomly sampled from [-100, 100] and [-200, 200] during training and test time respectively. The number of rows is sampled randomly from [30, 100] in training while during prediction the number of rows is 120. Each question in the test set is unique, i.e., it is generated from a distinct template. We use the following settings:

**Single Column:** We first perform experiments with a single column that enables 23 different question templates which can be answered using 4 time steps (Table 5.1).

| |
|---|
| sum |
| count |
| print |
| greater [number] sum |
| lesser [number] sum |
| greater [number] count |
| lesser [number] count |
| greater [number] print |
| lesser [number] print |
| greater [number1] and lesser [number2] sum |
| lesser [number1] and greater [number2] sum |
| greater [number1] or lesser [number2] sum |
| lesser [number1] or greater [number2] sum |
| greater [number1] and lesser [number2] count |
| lesser [number1] and greater [number2] count |
| greater [number1] or lesser [number2] count |
| lesser [number1] or greater [number2] count |
| greater [number1] and lesser [number2] print |
| lesser [number1] and greater [number2] print |
| greater [number1] or lesser [number2] print |
| lesser [number1] or greater [number2] print |
| sum diff count |
| count diff sum |

Table 5.1: 23 question templates for single column experiment. We have four categories of questions: 1) simple aggregation (sum, count) 2) comparison (greater, lesser) 3) logic (and, or) and, 4) arithmetic (diff). We first sample the categories uniformly randomly and each program within a category is equally likely. In the word variability experiment with 5 columns we sampled from the set of all programs uniformly randomly since greater than 90% of the test questions were unseen during training using the other procedure.

**Many Columns:** We increase the difficulty by experimenting with multiple columns (max_columns = 3, 5 or 10). During training, the number of columns is randomly sampled from (1, max_columns) and at test time every question had the maximum number of columns used during training (Table 5.2).

| |
|---|
| greater [number1] A and lesser [number2] A sum A |
| greater [number1] B and lesser [number2] B sum B |
| greater [number1] A and lesser [number2] A sum B |
| greater [number1] A and lesser [number2] B sum A |
| greater [number1] B and lesser [number2] A sum A |
| greater [number1] A and lesser [number2] B sum B |
| greater [number1] B and lesser [number2] B sum A |
| greater [number1] B and lesser [number2] B sum A |

Table 5.2: 8 question templates of type "greater [number1] and lesser [number2] sum" when there are 2 columns.

**Variability:** To simulate one aspect of the difficulties in dealing with natural language input, we consider multiple ways to refer to the same operation (Tables 5.3 and 5.4).

| | |
|---|---|
| sum | sum, total, total of, sum of |
| count | count, count of, how many |
| greater | greater, greater than, bigger, bigger than, larger, larger than |
| lesser | lesser, lesser than, smaller, smaller than, under |
| assign | print, display, show |
| difference | difference, difference between |

Table 5.3: Word variability, multiple ways to refer to the same operation.

**Text Match:** Now we consider cases where some columns in the input table contain text entries. We use a small vocabulary of 10 words and fill the column by uniformly randomly sampling from them. In our first experiment with text entries, the table always contains two columns, one with text and other with numeric entries (Table 5.5). In the next experiment, each example can have up to 3 columns containing numeric entries and up to 2 columns containing text entries during training. At test time, all the examples contain 3 columns with numeric entries and 2 columns with text entries.

| |
|---|
| greater [number] sum |
| greater [number] total |
| greater [number] total of |
| greater [number] sum of |
| greater than [number] sum |
| greater than [number] total |
| greater than [number] total of |
| greater than [number] sum of |
| bigger [number] sum |
| bigger [number] total |
| bigger [number] total of |
| bigger [number] sum of |
| bigger than [number] sum |
| bigger than [number] total |
| bigger than [number] total of |
| bigger than [number] sum of |
| larger [number] sum |
| larger [number] total |
| larger [number] total of |
| larger [number] sum of |
| larger than [number] sum |
| larger than [number] total |
| larger than [number] total of |
| larger than [number] sum of |

Table 5.4: 24 questions templates for questions of type "greater [number] sum" in the single column word variability experiment.

## 5.2 Results

In the following, we benchmark the performance of Neural Programmer on various versions of the table-comprehension dataset. We slowly increase the difficulty of the task by changing the table properties (more columns, mixed numeric and text entries) and question properties (word variability). After that we discuss a comparison between Neural Programmer, LSTM, and LSTM with Attention.

### 5.2.1 Neural Programmer

We use 4 time steps in our experiments ($T = 4$). Neural Programmer is trained with mini-batch stochastic gradient descent with Adam optimizer [56]. The parame-

| |
|---|
| word:0 A sum B |
| word:1 A sum B |
| word:2 A sum B |
| word:3 A sum B |
| word:4 A sum B |
| word:5 A sum B |
| word:6 A sum B |
| word:7 A sum B |
| word:8 A sum B |
| word:9 A sum B |

Table 5.5: 10 questions templates for questions of type "[word] A sum B" in the two columns text match experiment.

ters are initialized uniformly randomly within the range [-0.1, 0.1]. In all experiments, we set the mini-batch size to 50, dimensionality $d$ to 256, the initial learning rate and the momentum hyper-parameters of Adam to their default values [56]. We found that it is extremely useful to add random Gaussian noise to our gradients at every training step. This acts as a regularizer to the model and allows it to actively explore more programs. We use a schedule inspired from Welling & Teh [136], where at every step we sample a Gaussian of 0 mean and variance= $\text{curr\_step}^{-0.55}$.

To prevent exploding gradients, we perform gradient clipping by scaling the gradient when the norm exceeds a threshold [37]. The threshold value is picked from [1, 5, 50]. We tune the $\epsilon$ hyper-parameter in Adam from [1e-6, 1e-8], the Huber constant $\delta$ from [10, 25, 50] and $\lambda$ (weight between two losses) from [25, 50, 75, 100] using grid search. While performing experiments with multiple random restarts we find that the performance of the model is stable with respect to $\epsilon$ and gradient clipping threshold but we have to tune $\delta$ and $\lambda$ for the different random seeds.

The training set consists of $50,000$ triples in all our experiments. Table 5.6 shows the performance of Neural Programmer on synthetic data experiments. In single column experiments, the model answers all questions correctly which we manually verify by inspecting the programs induced by the model. In many columns experiments with

| Type | No. of Test Question Templates | Accuracy | % seen test |
|---|---|---|---|
| Single Column | 23 | 100.0 | 100 |
| 3 Columns | 307 | 99.02 | 100 |
| 5 Columns | 1231 | 99.11 | 98.62 |
| 10 Columns | 7900 | 99.13 | 62.44 |
| Word Variability on 1 Column | 1368 | 96.49 | 100 |
| Word Variability on 5 Columns | 24000 | 88.99 | 31.31 |
| Text Match on 2 Columns | 1125 | 99.11 | 97.42 |
| Text Match on 5 Columns | 14600 | 98.03 | 31.02 |

Table 5.6: Summary of the performance of Neural Programmer on various versions of the synthetic table-comprehension task. The prediction of the model is considered correct if it is equal to the correct answer up to the first decimal place. The last column indicates the percentage of question templates in the test set that are observed during training. The unseen question templates generate questions containing sequences of words that the model has never seen before. The model can generalize to unseen question templates which is evident in the 10-columns, word variability on 5-columns and text match on 5 columns experiments. This indicates that Neural Programmer is a powerful compositional model since solving unseen question templates requires performing a sequence of actions that it has never done during training.

5 columns, we use a bidirectional RNN and for 10 columns we additionally perform attention [3] on the question at every time step using the history vector. The model is able to generalize to unseen question templates which are a considerable fraction in our ten columns experiment. This can also be seen in the word variability experiment with 5 columns and text match experiment with 5 columns where more than two-thirds of the test set contains question templates that are unseen during training. This indicates that Neural Programmer is a powerful compositional model since solving unseen question templates requires inducing programs that do not appear during training. Almost all the errors made by the model were on questions that require the *difference operation* to be used. Table 5.7 shows examples of how the model selects the operation and column at every time step for three test questions.

Figure 5.1 shows an example of the effect of adding random noise to the gradients in our experiment with 5 columns.

| Question | t | Selected Op | Selected Column | $pivot_g$ | $pivot_l$ | Row select |
|---|---|---|---|---|---|---|
| greater 50.32 C and lesser 20.21 E sum H<br>*What is the sum of numbers in column H*<br>*whose field in column C is greater than 50.32*<br>*and field in Column E is lesser than 20.21.* | 1<br>2<br>3<br>4 | Greater<br>Lesser<br>And<br>Sum | C<br>E<br>-<br>H | 50.32 | 20.21 | $g_1$<br>$l_2$<br>$and_3$<br>$[0]^M$ |
| lesser -80.97 D or greater 12.57 B print F<br>*Print elements in column F*<br>*whose field in column D is lesser than -80.97*<br>*or field in Column B is greater than 12.57.* | 1<br>2<br>3<br>4 | Lesser<br>Greater<br>Or<br>Assign | D<br>B<br>-<br>F | 12.57 | -80.97 | $l_1$<br>$g_2$<br>$or_3$<br>$[0]^M$ |
| sum A diff count<br>*What is the difference*<br>*between sum of elements in*<br>*column A and number of rows* | 1<br>2<br>3<br>4 | Sum<br>Reset<br>Count<br>Diff | A<br>-<br>-<br>- | -1 | -1 | $[0]^M$<br>$[1]^M$<br>$[0]^M$<br>$[0]^M$ |

Table 5.7: Example outputs from the model for $T = 4$ time steps on three questions in the test set. We show the synthetically generated question along with its natural language translation. For each question, the model takes 4 steps and at each step selects an operation and a column. The pivot numbers for the comparison operations are computed before performing the 4 steps. We show the selected columns in cases during which the selected operation acts on a particular column.

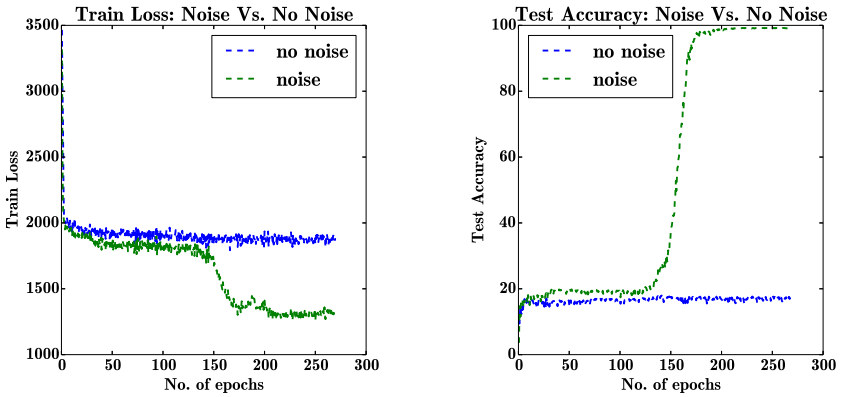



Figure 5.1: The effect of adding random noise to the gradients versus not adding it in our experiment with 5 columns when all hyper-parameters are the same. The models trained with noise generalizes almost always better.

### 5.2.2 Comparison to LSTM and LSTM with Attention

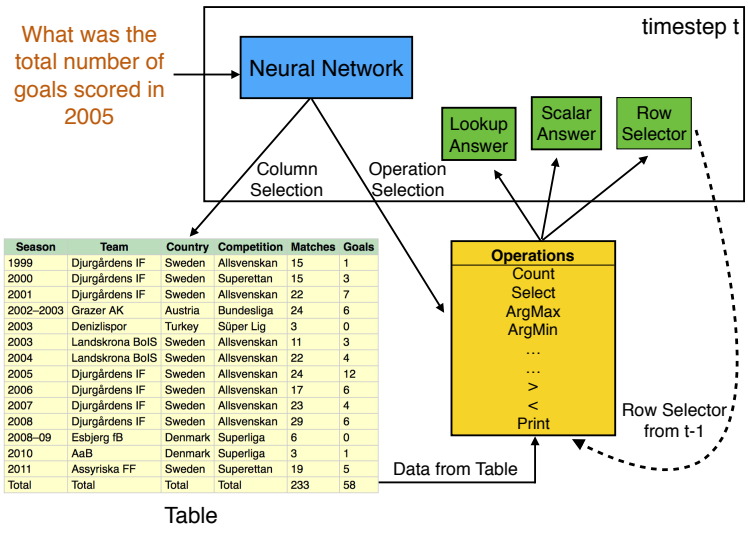
We apply a three-layer sequence-to-sequence LSTM recurrent network model [45, 124] and LSTM model with attention [3]. We explore multiple attention heads (1, 5, 10) and try two cases, placing the input table before and after the question. We consider a *simpler version* of the single column dataset with only questions that have scalar answers. The number of elements in the column is uniformly randomly sampled

from $[4, 7]$ while the elements are sampled from $[-10, 10]$. The best accuracy using these models is close to 80% in spite of relatively easier questions and supplying fresh training examples at every step. When the scale of the input numbers is changed to $[-50, 50]$ at test time, the accuracy drops to 30%.

Neural Programmer solves this task and achieves 100% accuracy using $50,000$ training examples. Since hardmax operation is used at test time, the answers (or the program induced) from Neural Programmer is invariant to the scale of numbers and the length of the input.

# CHAPTER 6

# SEMANTIC PARSING WITH NEURAL PROGRAMMER




| Season | Team | Country | Competition | Matches | Goals |
|---|---|---|---|---|---|
| 1999 | Djurgårdens IF | Sweden | Allsvenskan | 15 | 1 |
| 2000 | Djurgårdens IF | Sweden | Superettan | 15 | 3 |
| 2001 | Djurgårdens IF | Sweden | Allsvenskan | 22 | 7 |
| 2002–2003 | Grazer AK | Austria | Bundesliga | 24 | 6 |
| 2003 | Denizlispor | Turkey | Süper Lig | 3 | 0 |
| 2003 | Landskrona BoIS | Sweden | Allsvenskan | 11 | 3 |
| 2004 | Landskrona BoIS | Sweden | Allsvenskan | 22 | 4 |
| 2005 | Djurgårdens IF | Sweden | Allsvenskan | 24 | 12 |
| 2006 | Djurgårdens IF | Sweden | Allsvenskan | 17 | 6 |
| 2007 | Djurgårdens IF | Sweden | Allsvenskan | 23 | 4 |
| 2008 | Djurgårdens IF | Sweden | Allsvenskan | 29 | 6 |
| 2008–09 | Esbjerg fB | Denmark | Superliga | 6 | 0 |
| 2010 | AaB | Denmark | Superliga | 3 | 1 |
| 2011 | Assyriska FF | Sweden | Superettan | 19 | 5 |
| Total | Total | Total | Total | 233 | 58 |

Figure 6.1: Neural Programmer is a neural network augmented with a set of discrete operations. The model runs for a fixed number of time steps, selecting an operation and a column from the table at every time step. The induced program transfers information across timesteps using the *row selector* variable while the output of the model is stored in the *scalar answer* and *lookup answer* variables.

The experiments in the previous chapter are on synthetic data. Now, we consider a real-world task of learning natural language interface to databases using Neural Programmer. Databases are a pervasive way to store and access knowledge. However, it is not straightforward for users to interact with databases since it often requires programming skills and knowledge about database schemas. Overcoming this difficulty

by allowing users to communicate with databases via natural language is an active research area. The common approach to this task is by semantic parsing, which is the process of mapping natural language to symbolic representations of meaning. In this context, semantic parsing yields logical forms or programs that provide the desired response when executed on the databases [148]. Semantic parsing is a challenging problem that involves deep language understanding and reasoning with discrete operations such as counting and row selection [68].

We develop an approach to semantic parsing based on Neural Programmer. We show how to learn a natural language interface for answering questions using database tables, thus integrating differentiable operations that are typical of neural networks with the declarative knowledge contained in the tables and with discrete operations on tables and entries. For this purpose, we make several improvements and adjustments to Neural Programmer, in particular adapting its objective function to make it more broadly applicable.

As described before, Neural Programmer [84] is a neural network augmented with a set of discrete operations. It produces both a program, made up of those operations, and the result of running the program against a given table. As input, a model receives a question along with a table (Figure 6.1). The model runs for a fixed number of time steps, selecting an operation and a column from the table as the argument to the operation at each time step. During training, soft selection [3] is performed so that the model can be trained end-to-end using backpropagation. This approach allows Neural Programmer to explore the search space with better sample complexity than hard selection with the REINFORCE algorithm [140] would provide. All the parameters of the model are learned from a weak supervision signal that consists of only the final answer; the underlying program, which consists of a sequence of operations and of selected columns, is latent.

In the previous chapter, Neural Programmer is applied only on a synthetic dataset. In that dataset, when the expected answer is an entry in the given table, its position is explicitly marked in the table. However, real-world datasets certainly do not include those markers, and lead to many ambiguities (e.g., [92]). In particular, when the answer is a number that occurs literally in the table, it is not known, a priori, whether the answer should be generated by an operation or selected from the table. Similarly, when the answer is a natural language phrase that occurs in multiple positions in the table, it is not known which entry (or entries) in the table is actually responsible for the answer. We extend Neural Programmer to handle the weaker supervision signal by backpropagating through decisions that concern how the answer is generated when there is an ambiguity.

The work that is most similar to ours is that of Andreas et al. [2] on the *dynamic neural module network*. However, in their method, the neural network is only employed to search over a small set of candidate layouts provided by the syntactic parse of the question, and is trained using the REINFORCE algorithm [140]. Hence, their method cannot recover from parser errors, and it is not trivial to adapt the parser to the task at hand. Additionally, all their modules or operations are parametrized by a neural network, so it is difficult to apply their method on tasks that require discrete arithmetic operations. Finally, their experiments concern a simpler dataset that requires fewer operations, and therefore a smaller search space, than WikiTableQuestions. Concurrently to our work, Liang et al. [67] propose *neural symbolic machine*, a model very similar to Neural Programmer but trained using the REINFORCE algorithm [140]. They use only 2 discrete operations and run for a total of 3 timesteps, hence inducing programs that are much simpler than ours.

Our main experimental results concern WikiTableQuestions [92], a real-world question-answering dataset on database tables, with only 10,000 examples for weak supervision. This dataset is particularly challenging because of its small size and

the lack of strong supervision, and also because the tables provided at test time are never seen during training, so learning requires adaptation at test time to unseen column names. A state-of-the-art, traditional semantic parser that relies on pruning strategies to ease program search achieves 37.1% accuracy. Standard neural network models like sequence-to-sequence and pointer networks do not appear to be promising for this dataset, as confirmed in our experiments below, which yield single-digit accuracies. In comparison, a single Neural Programmer model using minimal text pre-processing, and trained end-to-end, achieves 34.2% accuracy. This surprising result is enabled primarily by the sample efficiency of Neural Programmer, by the enhanced objective function, and by reducing overfitting via strong regularization with dropout [120, 49, 32] and weight decay. An ensemble of 15 models, even with a trivial combination technique, achieves 37.7% accuracy.

## 6.1 Modifications to Neural Programmer

In this section we describe the modifications made to the model.

### 6.1.1 Selector

We make the following modifications to the selector module. First, in Neelakantan et al. [84], the column and operation selection are conditionally independent given the representations of the question and history RNN. We introduce a dependency by passing the weighted representation of operations with their probability distribution produced by the selector. This change make training more stable. Next, column selection is performed in Neelakantan et al. [84] using only the names of the column, however, this selection procedure is insufficient in real-world settings. For example the column selected in question 3 in Table 6.4 does not have a corresponding phrase in the question. Hence, to select a column we additionally use a boolean feature that indicates whether an entry in that column matches some phrase in the question.

Finally, we replace *tanh* activations in the selector module with $ReLU$ activations since they are easier to train. Following Neelakantan et al. [84], we employ hard selection at test time, hence, the output of the model is invariant to the size of the table and the scale of the numbers in the table.

### 6.1.2 Operations

We use 15 operations in the model that were chosen to closely match the set of operations used in the baseline model [92]. All the operations except *select* and *most frequent entry* operate only on the set of selected rows which is given by the *row selector* variable. Before the first timestep, all the rows in the table are set to be selected. The built-in operations are:

- *count* returns the number of selected rows in *row selector*.
- *select* and *most frequent entry* are operations which are computed only once for every question and output a boolean tensor with size same as the size of the input table. An entry in the output of the *select* operation is set to 1 if the entry matches some phrase in the question. The matched phrases in the question are anonymized to prevent overfitting. Similarly, for *most frequent entry*, it is set to 1 if the entry is the most frequently occurring one in its column.
- *argmax*, *argmin*, *greater than*, *less than*, *greater than or equal to*, *less than or equal to* are all operations that output a tensor with size same as the size of the input table. The semantics of these operations follow their name.
- *first*, *last*, *previous* and *next* modify the *row selector* and the semantics follow their name.
- *print* operation assigns *row selector* on the selected column of *lookup answer*.
- *reset* resets *row selector* to its initial value. This operation also serves as *no-op* when the model needs to induce programs whose complexity is less than $T$.

| Type | Operation | Definition |
|---|---|---|
| Aggregate | count | $count_t = \sum_{i=1}^{M} row\_select_{t-1}[i]$ |
| Superlative | argmax | $max_t[i][j] = \max(0.0, row\_select_{t-1}[i] - \sum_{k=1}^{M}([\Pi[i][j] < \Pi[k][j]] \times row\_select_{t-1}[k])), i = 1, \ldots, M, j = 1, \ldots, C$ |
| | argmin | $min_t[i][j] = \max(0.0, row\_select_{t-1}[i] - \sum_{k=1}^{M}([\Pi[i][j] > \Pi[k][j]] \times row\_select_{t-1}[k])), i = 1, \ldots, M, j = 1, \ldots, C$ |
| Comparison | $>$ | $g[i][j] = \Pi[i][j] > pivot_g, \forall(i,j), i = 1, \ldots, M, j = 1, \ldots, C$ |
| | $<$ | $l[i][j] = \Pi[i][j] < pivot_l, \forall(i,j), i = 1, \ldots, M, j = 1, \ldots, C$ |
| | $\geq$ | $ge[i][j] = \Pi[i][j] \geq pivot_{ge}, \forall(i,j), i = 1, \ldots, M, j = 1, \ldots, C$ |
| | $\leq$ | $le[i][j] = \Pi[i][j] \leq pivot_{le}, \forall(i,j), i = 1, \ldots, M, j = 1, \ldots, C$ |
| Table Ops | select | $s[i][j] = 1.0$ if $\Pi$[i][j] appears in question else 0.0, $\forall(i,j), i = 1, \ldots, M, j = 1, \ldots, C$ |
| | mfe | $mfe[i][j] = 1.0$ if $\Pi$[i][j] is the most common entry in column j else 0.0, $\forall(i,j), i = 1, \ldots, M, j = 1, \ldots, C$ |
| | first | $f_t[i] = \max(0.0, row\_select_{t-1}[i] - \sum_{j=1}^{i-1} row\_select_{t-1}[j]), i = 1, \ldots, M$ |
| | last | $la_t[i] = \max(0.0, row\_select_{t-1}[i] - \sum_{j=i+1}^{M} row\_select_{t-1}[j]), i = 1, \ldots, M$ |
| | previous | $p_t[i] = row\_select_{t-1}[i+1], i = 1, \ldots, M-1$ ; $p_t[M] = 0$ |
| | next | $n_t[i] = row\_select_{t-1}[i-1], i = 2, \ldots, M$ ; $n_t[1] = 0$ |
| Print | print | $lookup\ answer_t[i][j] = row\_select_{t-1}[i], \forall(i,j) i = 1, \ldots, M, j = 1, \ldots, C$ |
| Reset | reset | $r_t[i] = 1, \forall i = 1, 2, \ldots, M$ |

Table 6.1: List of all operations provided to the model along with their definitions. mfe is abbreviation for the operation *most frequent entry*. [*cond*] is 1 when *cond* is True, and 0 otherwise. Comparison operations, select, reset and most frequent entry operations are independent of the timestep while all the other operations are computed at every time step. Superlative operations and most frequent entry are computed within a column. The operations calculate the expected output with the respect to the membership probabilities given by the row selector so that they can work with probabilistic selection.

The operations are designed to work with probabilistic row and column selection so that the model is differentiable. The operations compute the expected output with the respect to the membership probabilities given by the row selector. Table 6.1 shows the list of operations built into the model along with their definitions.

### 6.1.3 Output and Row Selector

Neural programmer makes use of three variables: *row selector*, *scalar answer* and *lookup answer* which are updated at every timestep. The variable *lookup answer* stores answers that are selected from the table while *scalar answer* stores numeric answers that are not provided in the table. The induced program transfers information across

timesteps using the *row selector* variable which contains rows that are selected by the model.

Given an input table $\Pi$, containing $M$ rows and $C$ columns ($M$ and $C$ can vary across examples), the output variables at timestep $t$ are given by:

$$scalar\ answer_t = \alpha_t^{op}(count) output_t(count),$$

$$lookup\ answer_t[i][j] = \alpha_t^{col}(j)\alpha_t^{op}(print) row\ select_{t-1}[i], \forall(i,j) i = 1, 2, \ldots, M, j = 1, 2, \ldots, C$$

where $\alpha_t^{op}(op)$ and $\alpha_t^{col}(j)$ are the probabilities assigned by the selector to operation $op$ and column $j$ at timestep $t$ respectively and $output_t(count)$ is the output of the count operation at timestep $t$. $lookup\ answer_T[i][j]$ is the probability that the element $(i, j)$ in the input table is in the final answer predicted by the model.

The *row selector* is computed using the output of the remaining operations and is given by,

$$\begin{aligned}
row\ selector_t[i] = \sum_{j=1}^{C} \{ & \alpha_t^{col}(j)\alpha_t^{op}(>) g[i][j] + \alpha_t^{col}(j)\alpha_t^{op}(<) l[i][j] \\
& + \alpha_t^{col}(j)\alpha_t^{op}(\geq) ge[i][j] + \alpha_t^{col}(j)\alpha_t^{op}(\leq) le[i][j], \\
& + \alpha_t^{col}(j)\alpha_t^{op}(argmax) max_t[i][j] + \alpha_t^{col}(j)\alpha_t^{op}(argmin_t) min[i][j], \\
& + \alpha_t^{col}(j)\alpha_t^{op}(select) s[i][j] + \alpha_t^{col}(j)\alpha_t^{op}(mfe) mfe[i][j]\} \\
& + \alpha_t^{op}(previous) p_t[i] + \alpha_t^{op}(next) n_t[i] + \alpha_t^{op}(reset) r_t[i] \\
& + \alpha_t^{op}(first) f_t[i] + \alpha_t^{op}(last) la_t[i] \\
& \forall i, i = 1, 2, \ldots, M
\end{aligned}$$

where $\alpha_t^{op}(op)$ and $\alpha_t^{col}(j)$ are the probabilities assigned by the selector to operation $op$ and column $j$ at timestep $t$ respectively.

### 6.1.4 Training Objective

We modify the training objective of Neural Programmer to handle the supervision signal available in real-world settings. In previous work, the position of the answers are explicitly marked in the table when the answer is an entry from the table. However, as discussed previously, in real-world datasets (e.g., [92]) the answer is simply written down introducing two kinds of ambiguities. First, when the answer is a number and if the number is in the table, it is not known whether the loss should be computed using the *scalar answer* variable or the *lookup answer* variable. Second, when the answer is a natural language phrase and if the phrase occurs in multiple positions in the table, we again do not know which entry (or entries) in the table is actually responsible for generating the answer. We extend Neural Programmer to handle this weaker supervision signal during training by computing the loss only on the prediction that is closest to the desired response.

For scalar answers we compute the square loss:

$$L_{scalar}(scalar\ answer_T, y) = \frac{1}{2}(scalar\ answer_T - y)^2$$

where $y$ is the ground truth answer. We divide $L_{scalar}$ by the number of rows in the input table and do not backpropagate on examples for which the loss is greater than a threshold since it leads to instabilities in training.

When the answer is a list of items $y = (a_1, a_2, \ldots, a_N)$, for each element in the list $(a_i, i = 1, 2, \ldots, N)$ we compute all the entries in the table that match that element, given by $S_i = \{(r, c),\ \forall\ (r, c)\ \Pi[r][c] = a_i\}$. We tackle the ambiguity introduced when an answer item occurs at multiple entries in the table by computing the loss only on the entry which is assigned the highest probability by the model. We construct $g \in \{0, 1\}^{M \times C}$, where $g[i, j]$ indicates whether the element $(i, j)$ in the input table is

part of the output. We compute log-loss for each entry and the final loss is given by:

$$L_{lookup}(\textit{lookup answer}_T, y) = \sum_{i=1}^{N} min_{(r,c)\in S_i}(-\log(\textit{lookup answer}_T[r,c])) \\ - \frac{1}{MC}\sum_{i=1}^{M}\sum_{j=1}^{C}[g[i,j] == 0]\log(1 - \textit{lookup answer}_T[i,j])$$

where $[cond]$ is 1 when *cond* is True, and 0 otherwise.

We deal with the ambiguity that occurs when the ground truth is a number and if the number also occurs in the table, by computing the final loss as the *soft minimum* of $L_{scalar}$ and $L_{lookup}$. Otherwise, the loss for an example is $L_{scalar}$ when the ground truth is a number and $L_{lookup}$ when the ground truth matches some entries in the table. The two loss functions, $L_{scalar}$ and $L_{lookup}$ are in different scales, so we multiply $L_{lookup}$ by a constant factor which we set to 50.0 after a small exploration in our experiments.

Since we employ hard selection at test time, only one among *scalar answer* and *lookup answer* is modified at the last timestep. We use the variable that is set at the last timestep as the final output of the model.

## 6.2 Experiments

We apply Neural Programmer on the WikiTableQuestions dataset [92] and compare it to different non-neural baselines including a natural language semantic parser developed by Pasupat & Liang [92]. Further, we also report results from training the sequence-to-sequence model [124] and a modified version of the pointer networks [130]. Our model is implemented in TensorFlow [1] and the model takes approximately a day to train on a single Tesla K80 GPU. We use double-precision format to store the model parameters since the gradients become undefined values in single-precision format.

### 6.2.1 Data

We use the train, development, and test split given by Pasupat & Liang [92]. The dataset contains 11321, 2831, and 4344 examples for training, development, and testing respectively. We use their tokenization, number and date pre-processing. There are examples with answers that are neither number answers nor phrases selected from the table. We ignore these questions during training but the model is penalized during evaluation following Pasupat & Liang [92]. The tables provided in the test set are unseen at training, hence requiring the model to adapt to unseen column names at test time. We train only on examples for which the provided table has less than 100 rows since we run out of GPU memory otherwise, but consider all examples at test time.

### 6.2.2 Training Details

We use $T = 4$ timesteps in our experiments. Words and operations are represented as 256 dimensional vectors, and the hidden vectors of the question and the history RNN are also 256 dimensional. The parameters are initialized uniformly randomly within the range [-0.1, 0.1]. We train the model using the Adam optimizer [56] with mini-batches of size 20. The $\epsilon$ hyperparameter in Adam is set to 1e-6 while others are set to the default values. Since the training set is small compared to other datasets in which neural network models are usually applied, we rely on strong regularization:

- We clip the gradients to norm 1 and employ early-stopping.
- The occurrences of words that appear less than 10 times in the training set are replaced by a special unknown word token.
- We add a weight decay penalty with strength 0.0001.
- We use dropout with a keep probability of 0.8 on input and output vectors of the RNN, and selector, operation and column name representations [120].

- We use dropout with keep probability of 0.9 on the recurrent connections of the question RNN and history RNN using the technique from Gal & Ghahramani [32].

- We use word-dropout [49] with keep probability of 0.9. Here, words in the question are randomly replaced with the unknown word token while training.

We tune the dropout rates, regularization strength, and the $\epsilon$ hyperparameter using grid search on the development data, we fix the other hyperparameters after a small exploration during initial experiments.

### 6.2.3 Results

Table 6.2 shows the performance of our model in comparison to baselines from Pasupat & Liang [92]. The best result from Neural Programmer is achieved by an ensemble of 15 models. The only difference among these models is that the parameters of each model is initialized with a different random seed. We combine the models by averaging the predicted softmax distributions of the models at every timestep. While it is generally believed that neural network models require a large number of training examples compared to simpler linear models to get good performance, our model achieves competitive performance on this small dataset containing only 10,000 examples with weak supervision.

#### 6.2.3.1 Transfer Learning

We did not get better results either by using pre-trained word vectors [75] or by pre-training the question RNN with a language modeling objective [23].[1] A possible explanation is that the word vectors obtained from unsupervised learning may not be suitable to the task under consideration. For example, the learned representations of

[1]We use an RNN trained on 1 billion words [53].

| Method | Dev Accuracy | Test Accuracy |
|---|---|---|
| Baselines from Pasupat & Liang [92] | | |
| Information Retrieval System | 13.4 | 12.7 |
| Simple Semantic Parser | 23.6 | 24.3 |
| Semantic Parser | 37.0 | 37.1 |
| Neural Programmer | | |
| Neural Programmer | 34.1 | 34.2 |
| Ensemble of 15 Neural Programmer models | 37.5 | 37.7 |
| Oracle Score with 15 Neural Programmer models | 50.5 | - |

Table 6.2: Performance of Neural Programmer compared to baselines from [92]. The performance of an ensemble of 15 models is competitive to the current state-of-the-art natural language semantic parser.

words like *maximum* and *minimum* from unsupervised learning are usually close to each other but for our task it is counterproductive.

#### 6.2.3.2 Program Length Penalty

We add a term in the objective function that encourages the model to assign high probability to the reset function at each timestep. This serves as a regularization term that penalizes long programs. We find that optimizing the model with an additional term in the objective is much harder and the performance of the model does not improve.

#### 6.2.3.3 Neural Network Baselines

To understand the difficulty of the task for neural network models, we also experiment with two neural network baselines: the sequence-to-sequence model [124] and a modified version of the pointer networks [130]. The input to the sequence-to-sequence model is a concatenation of the table and the question, and the decoder produces the output one token at a time. We consider only examples whose input length is less than 400 to make the running time reasonable. The resulting dataset has $8,857$ and $1,623$ training and development examples respectively. The accuracy of the best

model on this development set after hyperparameter tuning is only 8.9%. Next, we experiment with pointer networks to select entries in the table as the final answer. We modify pointer networks to have two-attention heads: one to select the column and the other to select entries within a column. Additionally, the model performs multiple pondering steps on the table before returning the final answer. We train this model only on lookup questions, since the model does not have a decoder to generate answers. We consider only examples whose tables have less than 100 rows resulting in training and development set consisting of $7,534$ and $1,829$ examples respectively. The accuracy of the best model on this development set after hyperparameter tuning is only 4.0%. These results confirm our intuition that discrete operations are hard to learn for neural networks particularly with small data sets in real-world settings.

### 6.2.4 Analysis

| Method | Dev Accuracy |
|---|---|
| Neural Programmer | 34.1 |
| Neural Programmer - anonymization | 33.7 |
| Neural Programmer - match feature | 31.1 |
| Neural Programmer - {dropout,weight decay} | 30.3 |

Table 6.3: Model ablation studies. We find that dropout and weight decay, along with the boolean feature indicating a matched table entry for column selection have a significant effect on the performance of the model.

#### 6.2.4.1 Model Ablation

Table 6.3 shows the impact of different model design choices on the final performance. While anonymizing phrases in the question that match some table entry seems to have a small positive effect, regularization and the boolean feature indicating a matched table entry for column selection have a much larger effect on the performance.

| ID | Question | | Step 1 | Step 2 | Step 3 | Step 4 |
|---|---|---|---|---|---|---|
| 1 | what is the total number of teams? | Operation | - | - | - | count |
| | | Column | - | - | - | - |
| 2 | how many games had more than 1,500 in attendance? | Operation | - | - | >= | count |
| | | Column | - | - | attendance | - |
| 3 | what is the total number of runner-ups listed on the chart? | Operation | - | - | select | count |
| | | Column | - | - | outcome | - |
| 4 | which year held the most competitions? | Operation | - | - | mfe | print |
| | | Column | - | - | year | year |
| 5 | what opponent is listed last on the table? | Operation | last | - | last | print |
| | | Column | - | - | - | opponent |
| 6 | which section is longest?? | Operation | - | - | argmax | print |
| | | Column | - | - | kilometers | name |
| 7 | which engine(s) has the least amount of power? | Operation | - | - | argmin | print |
| | | Column | - | - | power | engine |
| 8 | what was claudia roll's time? | Operation | - | - | select | print |
| | | Column | - | - | swimmer | time |
| 9 | who had more silver medals, cuba or brazil? | Operation | argmax | select | argmax | print |
| | | Column | nation | nation | silver | nation |
| 10 | who was the next appointed director after lee p. brown? | Operation | select | next | last | print |
| | | Column | name | - | - | name |
| 11 | what team is listed previous to belgium? | Operation | select | previous | first | print |
| | | Column | team | - | - | team |

Table 6.4: A few examples of programs induced by Neural Programmer that generate the correct answer in the development set. mfe is abbreviation for the operation *most frequent entry*. The model runs for 4 timesteps selecting an operation and a column at every step. The model employs hard selection during evaluation. The column name is displayed in the table only when the operation picked at that step takes in a column as input while the operation is displayed only when it is other than the *reset* operation. Programs that choose *count* as the final operation produce a number as the final answer which is stored in *scalar answer*. For programs that select *print* as the final operation, the final answer consists of entries selected from the table which is stored in *lookup answer*.

#### 6.2.4.2 Induced Programs

Table 6.4 shows few examples of programs induced by Neural Programmer that yield the correct answer in the development set. The programs given in Table 6.4 show a few characteristics of the learned model. First, our analysis indicates that the model can adapt to unseen column names at test time. For example in Question 3, the word *outcome* occurs only 8 times in the training set and is replaced with the special unknown word token. Second, the model does not always induce the most efficient (with respect to number of operations other than the *reset* operation that are picked) program to solve a task. The last 3 questions in the table can be solved using

simpler programs. Finally, the model does not always induce the correct program to get the ground truth answer. For example, the last 2 programs will not result in the correct response for all input database tables. The programs would produce the correct response only when the *select* operation matches one entry in the table.

| Operation | Program in Table 6.4 | Amount (%) |
|---|---|---|
| Scalar Answer | | |
| Only Count | 1 | 6.5 |
| Comparison + Count | 2 | 2.1 |
| Select + Count | 3 | 22.1 |
| Scalar Answer | 1,2,3 | 30.7 |
| Lookup Answer | | |
| Most Frequent Entry + Print | 4 | 1.7 |
| First/Last + Print | 5 | 9.5 |
| Superlative + Print | 6,7 | 13.5 |
| Select + Print | 8 | 17.5 |
| Select + {first, last, previous, next, superlative} + Print | 9-11 | 27.1 |
| Lookup Answer | 4-11 | 69.3 |

Table 6.5: Statistics of the different sequence of operations among the examples answered correctly by the model in the development set. For each sequence of operations in the table, we also point to corresponding example programs in Table 6.4. Superlative operations include *argmax* and *argmin*, while comparison operations include *greater than*, *less than*, *greater than or equal to* and *less than or equal to*. The model induces a program that results in a scalar answer 30.7% of the time while the induced program is a table lookup for the remaining questions. *print* and *select* are the two most common operations used 69.3% and 66.7% of the time respectively.

#### 6.2.4.3 Contribution of Different Operations

Table 6.5 shows the contribution of the different operations. The model induces a program that results in a scalar answer 30.7% of the time while the induced program is a table lookup for the remaining questions. The two most commonly used operations by the model are *print* and *select*.

#### 6.2.4.4 Error Analysis

Now, we suggest ideas to potentially improve the performance of the model. First, the oracle performance with 15 Neural Programmer models is 50.5% on the development set while averaging achieves only 37.5% implying that there is still room for

improvement. Next, the accuracy of a single model on the training set is 53% which is about 20% higher than the accuracy in both the development set and the test set. This difference in performance indicates that the model suffers from significant overfitting even after employing strong regularization. It also suggests that the performance of the model could be greatly improved by obtaining more training data. Finally, we point out analysis in previous work [92] which shows that 21% of questions on a random set of 200 examples in the considered dataset are not answerable because of various issues such as annotation errors and tables requiring advanced normalization.

The experiments in the previous chapter and this chapter show that Neural Programmer performs significantly better than previously existing DNN models on tasks involving discrete reasoning. In this chapter, we show that Neural Programmer achieves performance comparable to a state-of-the-art traditional semantic parser even though the training set contains only 10,000 examples. The size of the training data is much smaller than other common datasets on which DNN models have been shown to work well. Moreover, unlike traditional semantic parsers, our approach does not use task specific manual rules to guide the program search. Our code is available at `https://github.com/tensorflow/models/tree/master/neural_programmer`.

#### 6.2.4.5 Effect of Strong Supervision

We study the impact of strong supervision on Neural Programmer by annotating 50 examples with the program or the semantic parse of the sentence. While the step-by-step program provides richer and unambiguous supervision, it is much more expensive to annotate programs compared to annotating only the final answer. To use the examples with strong supervision, we use a maximum likelihood objective for predicting the correct operation and column at each timestep. When using only these

50 examples, the accuracy of the model on the development set is 7.9% and 10.5% using weak and strong supervision respectively.

In another experiment, we first train the model only on strong supervision using the maximum likelihood objective for some number of training steps and then train the model with the entire training set as before with weak supervision. We find that performance of the model is same as before without any improvements. A possible explanation is that the frequently occurring programs in the annotated 50 examples occur often enough in the entire training set and the model fails to learn programs that occur infrequently even when trained with strong supervision.

#### 6.2.4.6 Adversarial Attack

To study the robustness of the model with respect to the output of the built-in operations, we introduce an adversarial attack. Here, we add an operation to the model that outputs a random number whose absolute value is always greater than a million. The accuracy of the model in this setting is 34.1% in the test set indicating that the model is indeed robust to the output of the built-in operations.

#### 6.2.4.7 Probability Calibration

Neural Programmer is trained with an objective function that pushes the model to have a peaky softmax distribution over the operations and the columns to suffer zero loss on an example. An interesting question to study is whether the probabilities given by the model are well calibrated i.e., are the probabilities low for questions the model answers incorrectly and high for questions answered correctly. We find that for close to 93% of the questions the model answers incorrectly, the probability given by the model is greater than 0.6. 73% of the incorrect questions are given a probability of less than 0.9 while 60% of the questions the model answers correctly are given a probability greater than 0.9. The threshold value of 0.9 gives the best separation between correct and incorrect questions.

Figure 6.2 shows coverage vs accuracy for different threshold values. Coverage is the fraction of examples among the ones the model answers correctly whose predicted probability is greater than the threshold while accuracy is the fraction of examples among the incorrect ones whose predicted probability is lesser than the threshold. Coverage and accuracy are never simultaneously high indicating that the probabilities predicted by the model are not well calibrated.

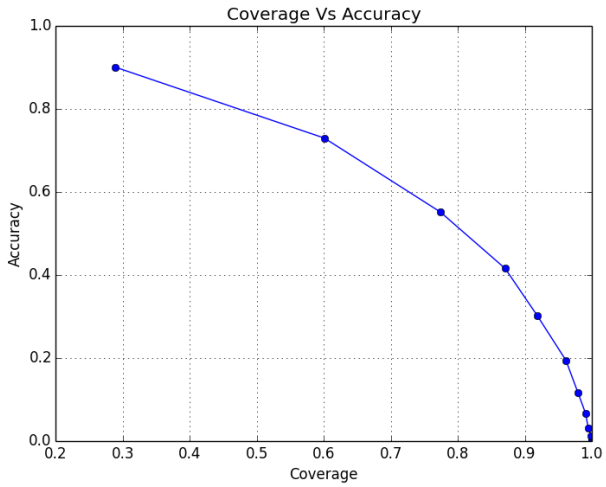



Figure 6.2: The figure shows coverage vs accuracy for different threshold values.

In another experiment, we added another term in the objective that allows the model to reject an example during training and suffer a cost for it. We try different hyperparameter settings for the cost and find that the model prefers to either reject or accept most examples. So, the model does not have a good estimate of whether it can answer a question.

# CHAPTER 7

# SCALING NEURAL PROGRAMMER

In the previous chapters, we show that Neural Programmer performs competitively to state-of-the-art approaches for learning natural language interface to databases. Moreover, Neural Programmer performs much better than other standard neural network models and is the first neural network model trained on weak supervision signals to achieve such results. However, the experiments in the last two chapters are performed only on small databases. There exists databases like Freebase [10] that contain billions of facts and millions of entities. In this chapter, we make first steps towards extending Neural Programmer to learn natural language interface to such large databases.

To enable training by backpropagation, Neural Programmer employs a soft selection (or attention) mechanism where it induces two probability distributions: one over operations and other over columns. The output of Neural Programmer at a timestep is defined as the weighted average of outputs obtained by executing every operation on every column. So, if the model has $N$ built-in operations and if the provided database table has $M$ columns, the model executes $O(M \times N)$ operations at every timestep for every example. This process is computationally expensive and makes Neural Programmer training not scale well to large databases.

In this chapter, we consider two sparse attention techniques to reduce the number of function executions at every timestep of Neural Programmer. First, we consider *k-max attention*, where we re-normalize the softmax distribution such that only the top-k elements of the softmax are non-zero. So, only the top-k most probable operations

and columns are considered. Instead of $O(M \times N)$ function executions at every timestep of the model, now we have only $O(k^2)$ function executions at every timestep of the model. By working with small values of k, we can potentially scale Neural Programmer training to large databases.

Next, we consider discrete attention where exactly one operation and one column are selected at every timestep by sampling from the softmax. Here, we need to execute only one operation at every timestep. However, sampling from the softmax is a non-differentiable operation and hence, we train the model with REINFORCE [140], a reinforcement learning algorithm [125]. While reinforcement learning methods have produced great success in game playing [78, 110, 79, 15], they rely on large number of samples which is often impractical in real-world settings. We discuss our results with the REINFORCE algorithm in the experiments section.

Liang et al. [67] use the REINFORCE algorithm to train a weakly-supervised, neural semantic parser. However, they experiment on a dataset that has several orders of magnitude smaller program search space compared to the dataset we use. While k-max attention has been used for scaling memory augmented neural networks [96] and training large mixture of experts model [109], it has not been used in the context of neural semantic parsing.

Since the available dataset for knowledge graph question answering requires only simple programs (majority of the questions require only one hop) to be induced [8], we do not have a good dataset to evaluate program induction methods for knowledge graph question answering. We continue to experiment on the WikiTableQuestions dataset used in the previous chapter. In our experiments, we find that k-max attention performs much better than training with the REINFORCE algorithm. More importantly, we find that with $k = 3$ there is a small drop in performance but requiring an order of magnitude lesser function executions at every timestep compared to the

approach described in the previous chapter. With $k = 6$, we match the performance requiring five times lesser function executions at every timestep.

## 7.1 Modifications to Neural Programmer

Continuing with the modifications to Neural Programmer, we make the following additional changes to Neural Programmer to scale training.

### 7.1.1 Selector

As discussed in Chapter 4, Neural Programmer employs a soft selection (or attention) mechanism where it induces two probability distributions: one over operations and other over columns.

**Operation Selection** is performed by:

$$\alpha_t^{op} = softmax(U \tanh(W^{op}[q; h_t]))$$

where $q \in R^d$ is the question representation, $h_t \in R^d$ is the hidden vector of the history RNN at timestep t, $W^{op} \in \mathbb{R}^{d \times 2d}$ is the parameter matrix of the operation selector that produces the probability distribution $\alpha_t^{op} \in [0, 1]^O$ over the set of operations.

The selector also produces a probability distribution over the columns at every time step. We obtain vector representations for the column names using the parameters in the question module (Section 4.1.1) by word embedding or an RNN phrase embedding. Let $P \in \mathbb{R}^{C \times d}$ be the matrix storing the representations of the column names.

**Column Selection** is performed by:

$$\alpha_t^{col} = softmax(P \tanh(W^{col}[q; h_t]))$$

where $W^{col} \in \mathbb{R}^{d \times 2d}$ is the parameter matrix of the column selector that produces the probability distribution $\alpha_t^{col} \in [0,1]^C$ over the set of columns.

Now, we consider two techniques to use a sparse selection mechanism in Neural Programmer. Both these techniques return sparse softmax distributions, $\bar{\alpha}_t^{op}$ and $\bar{\alpha}_t^{col}$.

#### 7.1.1.1 k-max Attention

We consider only the top $k$ most probable operations and columns at every timestep instead of all the operations and columns where $k$ is a model hyperparameter. The probability distribution is normalized over the top $k$ operations and columns. Only $k$ elements of $\bar{\alpha}_t^{op}$ and $\bar{\alpha}_t^{col}$ are non-zero.

$$\bar{\alpha}_t^{op}(i) = \begin{cases} \alpha_t^{op}(i), \text{if } \alpha_t^{op}(i) \in top\_k(\alpha_t^{op}) \\ 0, \text{otherwise} \end{cases}$$

$\bar{\alpha}_t^{op}$ is normalized to make the elements sum up to 1. Similarly, $\bar{\alpha}_t^{col}$ is obtained from $\alpha_t^{col}$.

#### 7.1.1.2 Discrete Attention

We consider the extreme case of sparse selection where the model selects only one operation and one column. Here, we sample an operation and a column from $\alpha_t^{op}$ and $\alpha_t^{col}$ respectively. $\bar{\alpha}_t^{op}$ and $\bar{\alpha}_t^{col}$ are obtained by converting the selected operation and column into one-hot vectors.

Sampling from a softmax distribution is a non-differntiable step and hence we can no longer train our model with standard backpropagation. We train our model with

the REINFORCE [140] algorithm where the objective is to maximize the expected reward,

$$max_{p(s_{1:T};\theta)}E(R)$$

where R is the reward function, $\theta$ are the parameters of Neural Programmer, $s_{1:T}$ is a sampled program under the current policy (or parameters), T is the number of timesteps and $p(s_{1:T};\theta)$ is the probability of the sample program with the current parameters.

Computing the approximate gradient using the REINFORCE rule requires sampling programs from the current policy which results in high variance in gradient estimation [125]. To reduce variance, we adopt the common practice of subtracting the reward with a baseline. Similarly, the model gets stuck in poor optima when training with REINFORCE and does not explore the space enough. We discuss the different reward functions, baselines and exploitation strategies we tried in the experiments section.

### 7.1.2 Output and Row Selector

As discussed previously, Neural programmer makes use of three variables: *row selector*, *scalar answer* and *lookup answer* which are updated at every timestep. The variable *lookup answer* stores answers that are selected from the table while *scalar answer* stores numeric answers that are not provided in the table. The induced program transfers information across timesteps using the *row selector* variable which contains rows that are selected by the model.

Given an input table $\Pi$, containing $M$ rows and $C$ columns ($M$ and $C$ can vary across examples) the output variables at timestep $t$ are now given by:

$$
\begin{aligned}
&\textit{scalar answer}_t = \bar{\alpha}_t^{op}(\textit{count})\textit{output}_t(\textit{count}),\\
&\textit{lookup answer}_t[i][j] = \bar{\alpha}_t^{col}(j)\bar{\alpha}_t^{op}(\textit{print})\textit{row select}_{t-1}[i],\\
&\forall(i,j) i = 1, 2, \ldots, M, j = 1, 2, \ldots, C
\end{aligned}
$$

where $\bar{\alpha}_t^{op}(op)$ and $\bar{\alpha}_t^{col}(j)$ are the probabilities assigned by the selector to operation $op$ and column $j$ at timestep $t$ respectively and $\textit{output}_t(\textit{count})$ is the output of the count operation at timestep $t$. $\textit{lookup answer}_T[i][j]$ is the probability that the element $(i, j)$ in the input table is in the final answer predicted by the model.

The *row selector* is computed using the output of the remaining operations and is now given by,

$$
\begin{aligned}
\textit{row selector}_t[i] = \sum_{j=1}^{C} \{&\bar{\alpha}_t^{col}(j)\bar{\alpha}_t^{op}(>)g[i][j] + \bar{\alpha}_t^{col}(j)\bar{\alpha}_t^{op}(<)l[i][j]\\
&+ \bar{\alpha}_t^{col}(j)\bar{\alpha}_t^{op}(\geq)ge[i][j] + \bar{\alpha}_t^{col}(j)\bar{\alpha}_t^{op}(\leq)le[i][j],\\
&+ \bar{\alpha}_t^{col}(j)\bar{\alpha}_t^{op}(argmax)max_t[i][j] + \bar{\alpha}_t^{col}(j)\bar{\alpha}_t^{op}(argmin_t)min[i][j],\\
&+ \bar{\alpha}_t^{col}(j)\bar{\alpha}_t^{op}(select)s[i][j] + \bar{\alpha}_t^{col}(j)\bar{\alpha}_t^{op}(mfe)mfe[i][j]\}\\
&+ \bar{\alpha}_t^{op}(previous)p_t[i] + \bar{\alpha}_t^{op}(next)n_t[i] + \bar{\alpha}_t^{op}(reset)r_t[i]\\
&+ \bar{\alpha}_t^{op}(first)f_t[i] + \bar{\alpha}_t^{op}(last)la_t[i]\\
&\forall i, i = 1, 2, \ldots, M
\end{aligned}
$$

where $\bar{\alpha}_t^{op}(op)$ and $\bar{\alpha}_t^{col}(j)$ are the sparse probability distributions assigned by the selector to operation $op$ and column $j$ at timestep $t$ respectively.

As can be seen from the equations above, an operations needs to be executed only if the probability assigned to that operation in a given timestep is greater than zero. Moreover, if the operation takes in a column as an argument, the operation needs to be executed only if both the probability assigned to that operation and the column is

greater than zero. By employing a sparse attention mechanism we reduce the number of operation executions.

When using the k-max attention mechanism, instead of executing $O(M \times N)$ operations at every timestep the model has to execute only $O(k^2)$ operations. By training with a small value for $k$ (in comparison to $N$ and $M$), we can scale the model to large databases. With the discrete attention mechanism, only one operation and one column is selected at every timestep and hence, only one operation needs to be executed at every timestep.

## 7.2 Related Work

Rae et al. [96] use k-max attention technique to speed up memory augmented neural networks. In memory augmented neural networks, a softmax distribution is induced for read and write operations over a set of memory slots at every timestep. This attention procedure is sped up by considering only the top-k elements of the softmax, hence reducing the number of read and write operations. In Shazeer et al. [109], k-max attention is used as a sparse gating procedure to train mixture of expert models. Here, the attention mechanism is performed over the set of experts.

Symbolic semantic parsers have been developed for freebase question answering [9, 60, 145]. The search space is cut-down either by considering only a small set of predicates [145] or entities for each input [145]. These methods are not end-to-end methods as the selection procedure is either rule-based or trained independently. Liang et al. [67] develop a neural approach for neural semantic parsing on Freebase using the REINFORCE algorithm. However, as mentioned before most of the questions in this dataset can be answered using programs of length one. Moreover, they require only 2 discrete operations to be built-in to the model for this dataset. Hence, the program search space is several orders of magnitude lesser compared to the search space in the dataset we use.

## 7.3 Experiments

We continue to perform experiments on the WikiTableQuestions dataset. Our goal is to study the trade-off between model accuracy and number of function executions. We use the same train, development and test split as used in the previous chapter. We continue to train the model with the Adam optimizer in the following experiments.

### 7.3.1 Neural Programmer with k-max Attention

We use exactly the same hyperparameter setting as given in the previous chapter. Figure 7.1 shows the test accuracy vs the value of k used in k-max attention. The number of operations in the model in this experiment is 15 while the average number of columns in the dataset is close to 10. We can see that in Table 7.1 that even for k=3, the performance of the model is close to using the entire softmax. This implies that for a small drop in performance we can execute an order of magnitude lesser number of operations at every timestep.

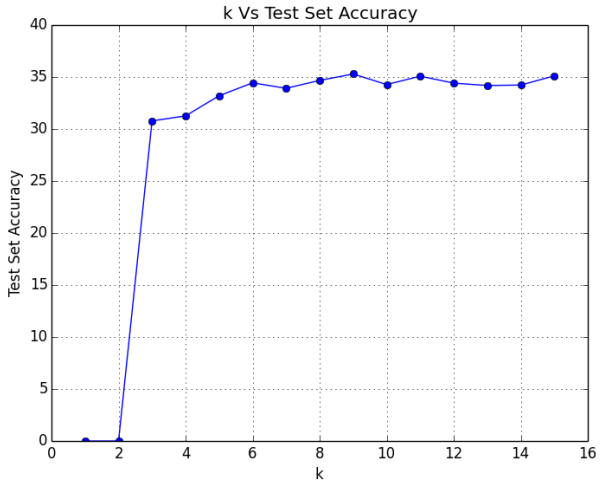



Figure 7.1: The figure shows test set accuracy vs the value of k. As can be seen from the plot, the accuracy of the model is close to running all operations on all columns even for small values of $k$.

We analyzed the programs induced by the different methods by counting the number of times the model correctly answers a lookup question or a number question

| Method | Accuracy |
|---|---|
| Full Softmax | 34.2 |
| Top-3 | 30.8 |
| Top-6 | 34.4 |

Table 7.1: Accuracy of Neural Programmer on the test set when using the full softmax, and for k=3 and k=6.

in the development set. The distribution is similar for k=6 and using the full softmax as done in the previous chapter. With k=6, the model answers 316 number questions and 644 lookup questions correctly while using the full softmax distribution results in 304 number questions and 656 lookup questions being answered correctly. With k=3, the model answers only 248 number questions correctly while answers 632 lookup questions correctly which is close to what we get with k=6 and using the full softmax. These statistics indicate that the drop in performance for k=3 is because overall it misses number questions that k=6 and using the full softmax answer correctly.

### 7.3.2 Neural Programmer with Discrete Attention

We experiment with two reward functions: 1) binary reward indicating whether the model predicts the correct answer or not, and 2) negative of the loss in the supervised setting. The model fails to learn in the first case, probably because the model has to search over millions of symbolic programs for every input question making it highly unlikely to find a program that gives a reward. Hence, the parameters of the model are not updated frequently enough. We use negative of the loss in the supervised setting as the reward function. To stabilize variance, we experiment with two baseline functions [125]: 1) average reward of the last 10,000 training steps, and 2) average reward of 20 samples obtained for that input example. We find the latter baseline to work well as it is a better approximation of the average reward for the given input especially in our case where each input has tables in different numerical range.

Finally to encourage exploration, we try two techniques: 1) adding Gaussian noise to the softmax layer and gradients, and 2) adding a term in the objective function that encourages the softmax to have high entropy [125]. We find that adding an entropy term in the objective function leads to more stable training and works better.

Even after training the model with the best reward function, baseline and exploration technique, we find the performance of the model to be extremely sensitive to the hyperparameter setting. We use a small learning rate of 1e-5 as higher learning rates lead to training instabilities and we tune a weight for the entropy term in the objective function. All the other hyperparameters are exactly the same as used in the previous chapter. After more than a week of training, the model achieves only 5% accuracy in the development set. While the success of reinforcement learning algorithms in general in real-world applications is limited, it is particularly challenging to apply in this task. Fruitful application of reinforcement learning algorithms relies on a well defined reward function [125] which is challenging in our case as the negative of the loss in the supervised setting depends on the contents of the table which varies across input examples. The different input tables have different numerical range making it difficult to design a well behaved reward function.

The experiments show that k-max attention performs much better than training with the REINFORCE algorithm. More importantly, k-max attention seems to be a promising technique for knowledge graph question answering as there is little or no performance drop for small values of k. With $k = 3$ there is a small drop in performance but requiring an order of magnitude lesser function executions at every timestep compared to the approach described in the previous chapter while with $k = 6$, we match the performance requiring five times lesser function executions at every timestep.

# CHAPTER 8

# CONCLUSIONS

In this thesis, we show that deep neural networks are a promising solution for knowledge representation and reasoning. We use existing neural network models to perform simple reasoning on large structured knowledge graphs. Our method achieves state-of-the-art results and has an attractive property to perform zero-shot prediction. In comparison to symbolic approaches, our method can generalize and reason on paths that are unseen at training. Reasoning with distributed representations on a knowledge graph is also studied in Rocktaschel et al. [104]. While our approach learns inference rules from data, their approach relies on manually supplied rules that are used as constraints the learned representations have to respect.

Next, we develop a novel neural network model, Neural Programmer to perform more sophisticated reasoning via program induction on small databases. Compared to methods that require either the full program [98, 146, 21] or trace of the program [119, 102] as supervision, our method requires only (input, output) pairs. We show that Neural Programmer can induce small latent programs on a synthetic dataset and a real-world dataset. Finally, we discuss methods to scale Neural Programmer training to large databases by reducing the number of required function executions at every timestep without sacrificing performance. Neural Programmer has been successfully applied for conversation question answering [50]. The authors report that Neural Programmer achieves 40.2% accuracy while state-of-the-art symbolic semantic parsers achieve only 33.2% accuracy. An extension of Neural Programmer to the conversation setting achieves 44.7% accuracy.

Deep neural networks have been employed in other reasoning applications such as textual entailment [14, 135], visual question answering [2] and reading comprehension [97, 133]. In textual entailment, given a pair of sentences the task is to predict whether they entail or contradict or have neutral relation between them. Visual question answering and reading comprehension are closer to the tasks considered in this thesis. Both of them are question answering tasks where in the former reasoning is required to be performed on an input image while in the latter reasoning is required to be performed on an input passage. The tasks considered in this thesis require reasoning to be performed on structured or semi-structured databases. The reasoning in visual question answering and reading comprehension tasks are more *fuzzy* compared to the reasoning involving discrete operations and rules considered in this thesis.

We have shown that representing abstract knowledge concepts with distributed representations and reasoning with powerful non-linear models leads to generalization to massive amounts of knowledge that can achieved with minimal human effort. The common theme in the methods developed in this thesis are to employ recurrent neural networks to model sequences, a differentiable attention mechanism as selection procedure and training by backpropagation.

## 8.1 Limitations

It is important to note that for program synthesis problems discrete search algorithms perform better than gradient based methods [35]. More concretely, gradient based methods like the ones developed in the thesis require more examples to induce the correct program than discrete search algorithms. The usefulness of our approach arises when the input is complicated with variabilities and ambiguities like natural language where gradient descent based approaches offer a more fruitful solution.

Neural Programmer relies on built-in operations that are manually defined. The user of the system has to provide the list of operations which could be difficult in

some cases. It would be beneficial to have a model that can add new operations and learn their definitions from data. However, this is challenging given the difficulty in learning simple discrete operations with neural networks [52, 54].

## 8.2 Future Research Directions

### 8.2.1 Intelligent Graph Exploration

The method that performs reasoning on paths in the knowledge graph discussed in Chapters 2 and 3 relies on random walks to obtain the set of paths connecting an entity pair. Even though, we construct the dataset with massive random walk exploration there could still be paths connecting the entity pair that are potentially useful for reasoning, missed by the random walk procedure. A query conditioned graph exploration mechanism might be able to collect a more useful set of paths connecting the entity pairs.

### 8.2.2 Knowledge Graph Question Answering

The datasets on which Neural Programmer has been tested contain database tables that contain a small number of columns. Hence, we could consider the set of all columns as possible input arguments and run each operation on each column individually in our experiments. However, if we consider knowledge graphs which have millions of entities where each entity could potentially be an input argument, running Neural Programmer in the naive way is computationally infeasible. In the final experiments discussed in the thesis, we can see that using only the top few elements of the softmax still performs almost as well as using all the elements of the softmax. Since the available dataset for knowledge graph question answering requires only simple programs (majority of the questions require only one hop) to be induced [8], we do not have a good dataset to evaluate program induction methods for knowledge graph question answering. One possibility is that the top-$k$ elements of the softmax (for

small values of $k$) is good enough for knowledge graph question answering and we can use the method described in the previous chapter directly.

### 8.2.3 Learning to Halt

We can extend Neural Programmer to induce programs with adaptive complexity by learning to halt. Currently, Neural Programmer always induces programs with complexity $T$. To induce programs with lesser complexity the model selects "no-op" operation for the remaining timesteps. Hence, the model performs many wasted executions especially at training time where all the operations need to be executed at every timestep. To overcome this issue, we propose to add a discrete sigmoid neuron which is used to decide whether the model continues to the next timestep or not. Adding a discrete neuron makes the model non-differentiable and we can train using the REINFORCE algorithm. Our method is inspired from the adaptive computation time technique proposed for recurrent neural networks [38]. The technical challenge here is that training the model with the REINFORCE algorithm is difficult as it performs much worse than backpropagation and is extremely sensitive to the hyperparameter setting as discussed in the discrete attention experiments before.

### 8.2.4 Longer Programs

Neural Programmer is evaluated only on datasets that require programs of maximum length four to be induced. Inducing longer programs introduces three main challenges: a) time complexity for each input example will increase linearly with respect to the length of the program, b) the number of ways the model can get the correct answer with wrong programs would be higher, and c) the optimization problem becomes harder because of vanishing and exploding gradient issues [45]. The first problem can be tackled by using sparse selection and learning to halt as discussed before. The next two problems can be tacked by using richer supervision: full program annotation [98, 146, 21] and trace-level annotations [119, 102]. An ambitious solution

for the same two problems is to use methods inspired from Feudal Reinforcement Learning [27, 128] where there are different parts of the model "thinking" at multiple timescales to circumvent vanishing and exploding gradient issues. In our case, the part of the model thinking at a longer timescale can potentially represent high-level functions consisting of sequence of operations while the part of the model thinking at a shorter timescale is responsible for predicting the individual operation. This method can be trained end-to-end just with weak supervision signal. A promising solution is to train a single model with the three kinds of supervision as each of them have their own pros and cons.

# BIBLIOGRAPHY


[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *ArXiv*, 2016.

[2] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to compose neural networks for question answering. *NAACL*, 2016.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2014.

[4] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. *ArXiv*, 2015.

[5] Michele Banko, Michael J Cafarella, Stephen Soderland, Matt Broadhead, and Oren Etzioni. Open information extraction from the web. *IJCAI*, 2007.

[6] Marco Baroni and Roberto Zamparelli. Nouns are vectors, adjectives are matrices: Representing adjective-noun constructions in semantic space. *EMNLP*, 2010.

[7] Jonathan Berant, Ido Dagan, and Jacob Goldberger. Global learning of typed entailment rules. 2011.

[8] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. *EMNLP*, 2013.

[9] Jonathan Berant, Vivek Srikumar, Pei-Chun Chen, Abby Vander Linden, Brittany Harding, Brad Huang, and Christopher D. Manning. Semantic parsing on freebase from question-answer pairs. *EMNLP*, 2013.

[10] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. 2008.

[11] Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. *NIPS*, 2013.

[12] Antoine Bordes, Sumit Chopra, and Jason Weston. Question answering with subgraph embeddings. *EMNLP*, 2014.

[13] Samuel R. Bowman, Christopher Potts, and Christopher D Manning. Recursive neural networks for learning logical semantics. *ArXiv*, 2014.

[14] Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. A large annotated corpus for learning natural language inference. *EMNLP*, 2015.

[15] Noam Brown and Tuomas Sandholm. Safe and nested subgame solving for imperfect-information games. *ArXiv*, 2017.

[16] Bruce G. Buchanan and Edward A. Feigenbaum. The stanford heuristic programming project: Goals and activities. *AAAI*, 1980.

[17] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka, and Tom M. Mitchell. Toward an architecture for never-ending language learning. *AAAI*, 2010.

[18] Rich Caruana. Multitask learning. *Machine Learning*, 1997.

[19] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. Listen, attend and spell. *ArXiv*, 2015.

[20] Kai-Wei Chang, Wen tau Yih, Bishan Yang, and Christopher Meek. Typed tensor decomposition of knowledge bases for relation extraction. *EMNLP*, 2014.

[21] Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. Learning structured natural language representations for semantic parsing. *ACL*, 2017.

[22] James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. Driving semantic parsing from the world's response. *CoNLL*, 2010.

[23] Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. *NIPS*, 2015.

[24] Rajarshi Das, Arvind Neelakantan, David Belanger, and Andrew McCallum. Chains of reasoning over entities, relations, and text using recurrent neural networks. *EACL*, 2017.

[25] Sreerupa Das, C. Lee Giles, and Guo zheng Sun. Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. *CogSci*, 1992.

[26] Sreerupa Das, C. Lee Giles, and Guo zheng Sun. Using prior knowledge in an NNPDA to learn context-free languages. *NIPS*, 1992.

[27] Peter Dayan and Geoffrey Hinton. Feudal reinforcement learning. *NIPS*, 1993.

[28] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 2011.

[29] Jacob Eisenstein, James Clarke, Dan Goldwasser, and Dan Roth. Reading to learn: Constructing features from semantic abstracts. *EMNLP*, 2009.

[30] Edward A. Feigenbaum. Expert systems: Looking back and looking ahead. *Report*, 1980.

[31] Andrea Frome, Gregory S. Corrado, Jonathon Shlens, Samy Bengio, Jeffrey Dean, Marc'Aurelio Ranzato, and Tomas Mikolov. Devise: A deep visual-semantic embedding model. *NIPS*, 2013.

[32] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. *NIPS*, 2016.

[33] Matt Gardner, Partha Pratim Talukdar, Bryan Kisiel, and Tom M. Mitchell. Improving learning and inference in a large knowledge-base using latent syntactic cues. *EMNLP*, 2013.

[34] Matt Gardner, Partha Talukdar, Jayant Krishnamurthy, and Tom Mitchell. Incorporating vector space similarity in random walk inference over knowledge bases. *EMNLP*, 2014.

[35] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *ArXiv*, 2016.

[36] Felix A. Gers and Jürgen Schmidhuber. LSTM recurrent networks learn simple context free and context sensitive languages. *IEEE Transactions on Neural Networks*, 2001.

[37] Alex Graves. Generating sequences with recurrent neural networks. *ArXiv*, 2013.

[38] Alex Graves. Adaptive computation time for recurrent neural networks. *ArXiv*, 2016.

[39] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. *ICML*, 2014.

[40] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *ArXiv*, 2014.

[41] K. Guu, J. Miller, and P. Liang. Traversing knowledge graphs in vector space. *EMNLP*, 2015.

[42] Awni Y. Hannun, Carl Case, Jared Casper, Bryan C. Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. *ArXiv*, 2014.

[43] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. *NIPS*, 2015.

[44] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

[45] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

[46] Peter Huber. Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, 1964.

[47] Ozan Irsoy and Claire Cardie. Deep recursive neural networks for compositionality in language. *NIPS*, 2014.

[48] Mohit Iyyer, Jordan L. Boyd-Graber, Leonardo Max Batista Claudino, Richard Socher, and Hal Daumé III. A neural network for factoid question answering over paragraphs. *EMNLP*, 2014.

[49] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. Deep unordered composition rivals syntactic methods for text classification. *ACL*, 2015.

[50] Mohit Iyyer, Wen tau Yih, and Ming-Wei Chang. Search-based neural structured learning for sequential question answering. *ACL*, 2017.

[51] Robin Jia and Percy Liang. Data recombination for neural semantic parsing. *ACL*, 2016.

[52] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *NIPS*, 2015.

[53] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *ArXiv*, 2016.

[54] Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *ICLR*, 2016.

[55] Angelika Kimmig, Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. *NIPS Workshop on Probabilistic Programming: Foundations and Applications*, 2012.

[56] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ArXiv*, 2014.

[57] Tomas Kocisky, Gabor Melis, Edward Grefenstette, Chris Dyer, Wang Ling, Phil Blunsom, and Karl Moritz Hermann. Semantic parsing with semi-supervised sequential autoencoders. *ArXiv*, 2016.

[58] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *NIPS*, 2012.

[59] Ankit Kumar, Ozan Irsoy, Jonathan Su, James Bradbury, Robert English, Brian Pierce, Peter Ondruska, Ishaan Gulrajani, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. *ICML*, 2016.

[60] Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke Zettlemoyer. Scaling semantic parsers with on-the-fly ontology matching. *EMNLP*, 2013.

[61] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 2016.

[62] Ni Lao, Tom Mitchell, and William W. Cohen. Random walk inference and learning in a large scale knowledge base. *EMNLP*, 2011.

[63] Ni Lao, Amarnag Subramanya, Fernando Pereira, and William W. Cohen. Reading the web with learned syntactic-semantic inference rules. *EMNLP*, 2012.

[64] Hugo Larochelle, Dumitru Erhan, and Yoshua Bengio. Zero-data learning of new tasks. *National Conference on Artificial Intelligence*, 2008.

[65] Nada Lavrac and Saso Dzeroski. Inductive logic programming: Techniques and applications. *Ellis Horwood Series in Artificial Intelligence*, 1994.

[66] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A simple way to initialize recurrent networks of rectified linear units. *ArXiv*, 2015.

[67] Chen Liang, Jonathan Berant, Quoc Le, Kenneth Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *NAMPI Workshop, NIPS*, 2016.

[68] Percy Liang. Learning executable semantic parsers for natural language understanding. *ACM*, 2016.

[69] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical Bayesian approach. *ICML*, 2010.

[70] Percy Liang, Michael I. Jordan, and Dan Klein. Learning dependency-based compositional semantics. In *ACL*, 2011.

[71] Dekang Lin and Patrick Pantel. Dirt - discovery of inference rules from text. *KDD*, 2001.

[72] Yankai Lin, Zhiyuan Liu, Huan-Bo Luan, Maosong Sun, Siwei Rao, and Song Liu. Modeling relation paths for representation learning of knowledge bases. *EMNLP*, 2015.

[73] Thang Luong, Ilya Sutskever, Quoc V. Le, Oriol Vinyals, and Wojciech Zaremba. Addressing the rare word problem in neural machine translation. *ACL*, 2014.

[74] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. *Annual Conference of the International Speech Communication Association*, 2010.

[75] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *ArXiv*, 2013.

[76] Bonan Min, Ralph Grishman, Li Wan, Chang Wang, and David Gondek. Distant supervision for relation extraction with an incomplete knowledge base. *NAACL*, 2013.

[77] Jeff Mitchell and Mirella Lapata. Vector-based models of semantic composition. *ACL*, 2008.

[78] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.

[79] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *American Association for the Advancement of Science*, 2017.

[80] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 1990.

[81] Arvind Neelakantan, Benjamin Roth, and Andrew McCallum. Knowledge base completion using compositional vector space models. *Workshop on Automated Knowledge Base Construction at NIPS*, 2014.

[82] Arvind Neelakantan, Jeevan Shankar, Alexandre Passos, and Andrew McCallum. Efficient non-parametric estimation of multiple embeddings per word in vector space. *EMNLP*, 2014.

[83] Arvind Neelakantan, Benjamin Roth, and Andrew McCallum. Compositional vector space models for knowledge base completion. *ACL*, 2015.

[84] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural Programmer: Inducing latent programs with gradient descent. *ICLR*, 2016.

[85] Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *ICLR Workshop*, 2016.

[86] Arvind Neelakantan, Quoc V Le, Martin Abadi, Andrew McCallum, and Dario Amodei. Learning a natural language interface with neural programmer. *ICLR*, 2017.

[87] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. *ICML*, 2011.

[88] Mohammad Norouzi, Tomas Mikolov, Samy Bengio, Yoram Singer, Jonathon Shlens, Andrea Frome, Greg Corrado, and Jeffrey Dean. Zero-shot learning by convex combination of semantic embeddings. *ICLR*, 2014.

[89] An Overview of the KL-ONE Knowledge Representation System. Ronald j. brachman and james g. schmolze. *Cognitive Science*, 1985.

[90] Dave Orr, Amarnag Subramanya, Evgeniy Gabrilovich, and Michael Ringgaard. 11 billion clues in 800 million documents: A web research corpus annotated with freebase concepts. *Report*, 2013.

[91] Mark Palatucci, Dean Pomerleau, Geoffrey Hinton, and Tom Mitchell. Zero-shot learning with semantic output codes. 2009.

[92] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. *ACL*, 2015.

[93] Baolin Peng, Zhengdong Lu, Hang Li, and Kam-Fai Wong. Towards neural network-based reasoning. *ArXiv*, 2015.

[94] Steven T. Piantadosi, N.D. Goodman, B.A. Ellis, and J.B. Tenenbaum. A Bayesian model of the acquisition of compositional semantics. *CogSci*, 2008.

[95] Hoifung Poon. Grounded unsupervised semantic parsing. *ACL*, 2013.

[96] Jack W. Rae, Jonathan J. Hunt, Tim Harley, Ivo Danihelka, Andrew W. Senior, Greg Wayne, Alex Graves, and Timothy P. Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. *NIPS*, 2016.

[97] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. *ArXiv*, 2016.

[98] Scott Reed and Nando De Freitas. Neural programmer-interpreters. *ICLR*, 2016.

[99] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. *UAI*, 2009.

[100] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 2006.

[101] Sebastian Riedel, Limin Yao, Andrew McCallum, and Benjamin M. Marlin. Relation extraction with matrix factorization and universal schemas. *NAACL*, 2013.

[102] Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. Programming with a differentiable forth interpreter. *ArXiv*, 2016.

[103] J.C. Robinson. A machine-oriented logic based on the resolution principle. *ACM*, 1965.

[104] Tim Rocktaschel, Sameer Singh, and Sebastian Riedel. Injecting logical background knowledge into embeddings for relation extraction. *NAACL*, 2015.

[105] Dan Roth and Wen-tau Yih. Global inference for entity and relation identification via a linear programming formulation. 2007.

[106] J. Schmidhuber. A self-referentialweight matrix. *ICANN*, 1993.

[107] Stefan Schoenmackers, Oren Etzioni, Daniel S. Weld, and Jesse Davis. Learning first-order horn clauses from web text. *EMNLP*, 2010.

[108] Lifeng Shang, Zhengdogn Lu, and Hang Li. Neural responding machine for short-text conversation. *ArXiv*, 2015.

[109] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *ICLR*, 2017.

[110] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[111] Herbert A. Simon, J.C. Shaw, and Allen Newell. Logic theorist. *Report*, 1956.

[112] Herbert A. Simon, J.C. Shaw, and Allen Newell. Report on a general problem-solving program. *International Conference on Information Processing*, 1959.

[113] Sameer Singh, Sebastian Riedel, Brian Martin, Jiaping Zheng, and Andrew McCallum. Joint inference of entities, relations, and coreference. *AKBC Workshop*, 2013.

[114] Richard Socher, Cliff Chiung-Yu Lin, Christopher D. Manning, and Andrew Y. Ng. Parsing natural scenes and natural language with recursive neural networks. *ICML*, 2011.

[115] Richard Socher, Brody Huval, Christopher D. Manning, and Andrew Y. Ng. Semantic compositionality through recursive matrix-vector spaces. *EMNLP*, 2012.

[116] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. *NIPS*, 2013.

[117] Richard Socher, Milind Ganjoo, Christopher D Manning, and Andrew Ng. Zero-shot learning through cross-modal transfer. *NIPS*, 2013.

[118] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. *EMNLP*, 2013.

[119] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[120] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 2014.

[121] Mark Steijvers. A recurrent network that performs a context-sensitive prediction task. *CogSci*, 1996.

[122] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A core of semantic knowledge. *WWW*, 2007.

[123] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. *NIPS*, 2015.

[124] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *NIPS*, 2014.

[125] Richard Sutton and Andrew Barto. Reinforcement learning: An introduction. *MIT Press*, 2012.

[126] Kristina Toutanova, Xi Victoria Lin, Scott Wen tau Yih, Hoifung Poon, and Chris Quirk. Compositional learning of embeddings for relation paths in knowledge bases and text. ACL, 2016.

[127] Patrick Verga, Arvind Neelakantan, and Andrew McCallum. Generalizing to unseen entities and entity pairs with row-less universal schema. *EACL*, 2017.

[128] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *ArXiv*, 2017.

[129] Oriol Vinyals and Quoc V. Le. A neural conversational model. *ICML DL Workshop*, 2015.

[130] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *NIPS*, 2015.

[131] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. *NIPS*, 2015.

[132] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *CVPR*, 2015.

[133] Shuohang Wang and Jing Jiang. Machine comprehension using match-lstm and answer pointer. *ArXiv*, 2016.

[134] Yushi Wang, Jonathan Berant, and Percy Liang. Building a semantic parser overnight. *ACL*, 2015.

[135] Zhiguo Wang, Wael Hamza, and Radu Florian. Bilateral multi-perspective matching for natural language sentences. *ArXiv*, 2017.

[136] Max Welling and Yee Whye Teh. Bayesian learning via stochastic gradient Langevin dynamics. *ICML*, 2011.

[137] P. Werbos. Backpropagation through time: what does it do and how to do it. *IEEE*, 1990.

[138] Jason Weston, Ron Weiss, and Hector Yee. Nonlinear latent factorization by embedding multiple user interests. *RecSys*, 2013.

[139] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory Networks. *ICLR*, 2015.

[140] Ronald Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.

[141] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *ICML*, 2015.

[142] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. *ArXiv*, 2014.

[143] Alexander Yates and Oren Etzioni. Unsupervised resolution of objects and relations on the web. *NAACL*, 2007.

[144] Ainur Yessenalina and Claire Cardie. Compositional matrix-space models for sentiment analysis. *EMNLP*, 2011.

[145] Scott Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. Semantic parsing via staged query graph generation: Question answering with knowledge base. *ACL*, 2015.

[146] Pengcheng Yin, Zhengdong Lu, Hang Li, and Ben Kao. Neural enquirer: Learning to query tables with natural language. *ArXiv*, 2015.

[147] Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. *ICML*, 2016.

[148] John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. *AAAI/IAAI*, 1996.

[149] Z. Zeng, R. Goodman, and P. Smyth. Discrete recurrent neural networks for grammatical inference. *IEEE Transactions on Neural Networks*, 1994.

[150] Luke S. Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *UAI*, 2005.