NEURAL PROGRAMMER: INDUCING LATENT PROGRAMS WITH GRADIENT DESCENT

Arvind Neelakantan*
University of Massachusetts Amherst
arvind@cs.umass.edu

Quoc V. Le
Google Brain
qvl@google.com

Ilya Sutskever
Google Brain
ilyasu@google.com

ABSTRACT

Deep neural networks have achieved impressive supervised classification performance in many tasks including image recognition, speech recognition, and sequence to sequence learning. However, this success has not been translated to applications like question answering that may involve complex arithmetic and logic reasoning. A major limitation of these models is in their inability to learn even simple arithmetic and logic operations. For example, it has been shown that neural networks fail to learn to add two binary numbers reliably. In this work, we propose *Neural Programmer*, a neural network augmented with a small set of basic arithmetic and logic operations that can be trained end-to-end using backpropagation. Neural Programmer can call these augmented operations over several steps, thereby inducing compositional programs that are more complex than the built-in operations. The model learns from a weak supervision signal which is the result of execution of the correct program, hence it does not require expensive annotation of the correct program itself. The decisions of what operations to call, and what data segments to apply to are inferred by Neural Programmer. Such decisions, during training, are done in a differentiable fashion so that the entire network can be trained jointly by gradient descent. We find that training the model is difficult, but it can be greatly improved by adding random noise to the gradient. On a fairly complex synthetic table-comprehension dataset, traditional recurrent networks and attentional models perform poorly while Neural Programmer typically obtains nearly perfect accuracy.

1 INTRODUCTION

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 (Krizhevsky et al., 2012) and speech recognition (Hinton et al., 2012) 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 (Sutskever et al., 2014; Bahdanau et al., 2014; Luong et al., 2014), image captioning (Vinyals et al., 2015; Xu et al., 2015), conversational modeling (Shang et al., 2015; Vinyals & Le, 2015), end-to-end Q&A (Sukhbaatar et al., 2015; Peng et al., 2015; Hermann et al., 2015), and end-to-end speech recognition (Graves & Jaitly, 2014; Hannun et al., 2014; Chan et al., 2015; Bahdanau et al., 2015).

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 crisp reasoning. A major limitation of these models is in their inability to learn even simple arithmetic and logic operations. For example, Joulin & Mikolov (2015) 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 downstream applications that require complex reasoning, e.g., natural language question answering. For example, to answer the question "how many states border Texas?" (see Zettlemoyer & Collins (2005)), the algorithm has to perform an act of counting in a table which is something that a neural network is not yet good at.

*Work done during an internship at Google.

A fairly 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 (Zelle & Mooney, 1996). This problem is often formulated as mapping a natural language question to an executable query.

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 (Liang et al., 2010). 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 (Zettlemoyer & Collins, 2005) or a domain-specific grammar (Liang et al., 2011). 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. (2015) 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 model that does not require substantial human supervision and is broadly applicable across different domains, data sources and natural languages. We propose *Neural Programmer* (Figure 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.

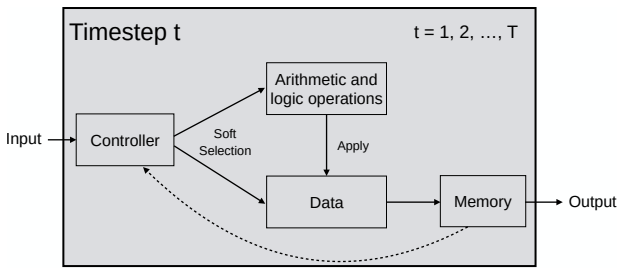

Figure 1: The architecture of Neural Programmer, a neural network augmented with arithmetic and logic operations. The controller selects the operation and the data segment. The memory stores the output of the operations applied to the data segments and the previous actions taken by the controller. The 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. This approach of using an augmented logic and arithmetic component is reminiscent of the idea of using an ALU (arithmetic and logic unit) in a conventional computer (Von Neumann, 1945). It is loosely related to the symbolic numerical processing abilities exhibited in the intraparietal sulcus (IPS) area of the brain (Piazza et al., 2004; Cantlon et al., 2006; Kucian et al., 2006; Fias et al., 2007; Dastjerdi et al., 2013). Our work is also inspired by the success of the soft attention mechanism (Bahdanau et al., 2014) and its application in learning a neural network to control an additional memory component (Graves et al., 2014; Sukhbaatar et al., 2015).

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.

We experiment with a synthetic table-comprehension dataset, consisting of questions with a wide range of difficulty levels. Examples of natural language translated queries include "print elements in column H whose field in column C is greater than 50 and field in column E is less than 20?" or "what is the difference between sum of elements in column A and number of rows in the table?". We find that LSTM recurrent networks (Hochreiter & Schmidhuber, 1997) and LSTM models with attention (Bahdanau et al., 2014) do not work well. Neural Programmer, however, can completely solve this task or achieve greater than 99% accuracy on most cases by inducing the required latent program. We find that training the model is difficult, but it can be greatly improved by injecting random Gaussian noise to the gradient (Welling & Teh, 2011; Neelakantan et al., 2016) which enhances the generalization ability of the Neural Programmer.

2 NEURAL PROGRAMMER

Even though our model is quite general, in this paper, we 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 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.

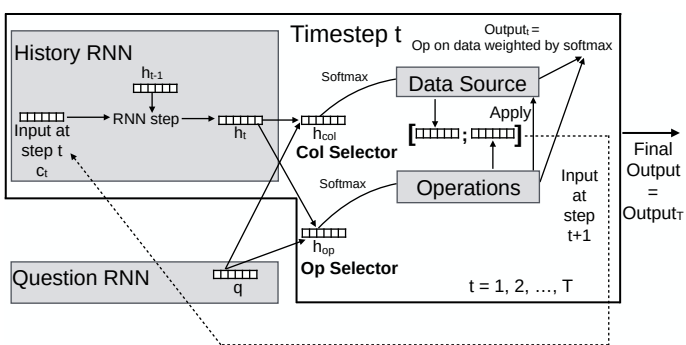

Figure 2: An implementation of Neural Programmer for the task of question answering on tables. The output of the model at time step t is obtained by applying the operations on the data segments weighted by their probabilities. The final output of the model is the output at time step T. The dotted line indicates the input to the history RNN at step t+1.

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.

2.1 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 (Werbos, 1990) parameterized by $W^{question}$ and the last hidden state of the RNN is used as the question representation (Figure 3).

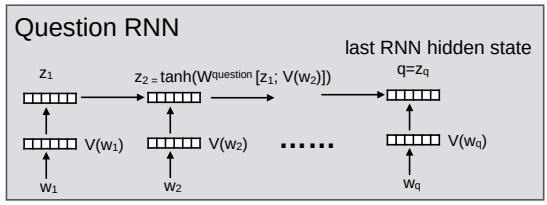

Figure 3: The question module to process the input question. $q = z_q$ denotes the question representation used by Neural Programmer.

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 2.3.

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.

2.2 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 2.4) 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).

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 2.1) 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:

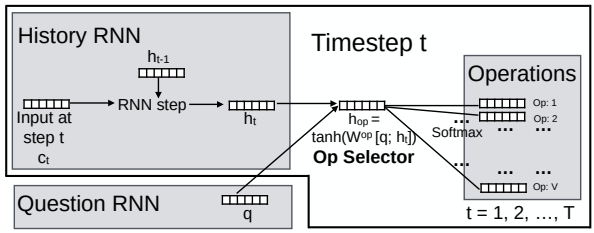

Figure 4: Operation selection at time step t where the selector assigns a probability distribution over the set of operations.

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

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 5).

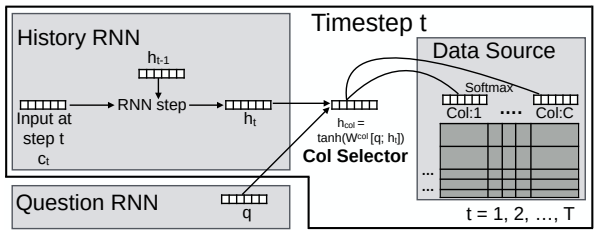

Figure 5: Data selection at time step t where the selector assigns a probability distribution over the set of columns.

2.3 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 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 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 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.

By overloading the definition of α_t^{op} and α_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 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.

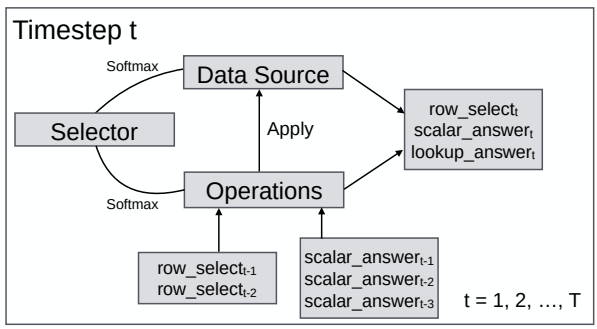

Figure 6: The output and row selector variables are obtained by applying the operations on the data segments and additively combining their outputs weighted using the probabilities assigned by the selector.

[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.

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.

2.3.1 Handling Text Entries

So far, our disscusion has been only concerned with tables that have numeric entries. In this section 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 2.2), 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:

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

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.

2.4 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 7 shows details of this component.

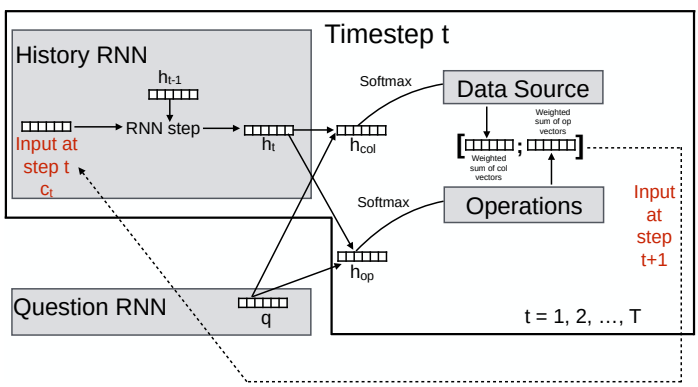

Figure 7: The history RNN which helps in remembering the previous operations and data segments selected by the model. The dotted line indicates the input to the history RNN at step t+1.

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 $[0]^d$.

2.5 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 (Huber, 1964) 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 δ 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 λ is a hyper-parameter of the model that allows to weight the two loss functions appropriately.

At inference 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 during inference.

3 EXPERIMENTS

Neural Programmer 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 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 (Pasupat & Liang, 2015) 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.

3.1 DATA

We generate question, table and answer triples using a synthetic grammar. Tables 4 and 5 (see Appendix) shows examples of question templates from the synthetic grammar for single and multiple

Algorithm 1 High-level view of Neural Programmer during its inference stage 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

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.

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.

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 6 and 7).

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 8). 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.

3.2 MODELS

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.

3.2.1 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 (Kingma & Ba, 2014). The parameters 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 (Kingma & Ba, 2014). 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 (2011), where at every step we sample a Gaussian of 0 mean and variance= curr_step$^{-0.55}$.

To prevent exploding gradients, we perform gradient clipping by scaling the gradient when the norm exceeds a threshold (Graves, 2013). The threshold value is picked from [1, 5, 50]. We tune the ϵ hyper-parameter in Adam from [1e-6, 1e-8], the Huber constant δ from [10, 25, 50] and λ (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 ϵ and gradient clipping threshold but we have to tune δ and λ for the different random seeds.

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 2: 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.

The training set consists of $50,000$ triples in all our experiments. Table 2 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 5 columns, we use a bidirectional RNN and for 10 columns we additionally perform attention (Bahdanau et al., 2014) 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 3 shows examples of how the model selects the operation and column at every time step for three test questions.

Figure 8 shows an example of the effect of adding random noise to the gradients in our experiment with 5 columns.

3.2.2 Comparison to LSTM and LSTM with Attention

We apply a three-layer sequence-to-sequence LSTM recurrent network model (Hochreiter & Schmidhuber, 1997; Sutskever et al., 2014) and LSTM model with attention (Bahdanau et al., 2014). 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

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

Table 3: 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.

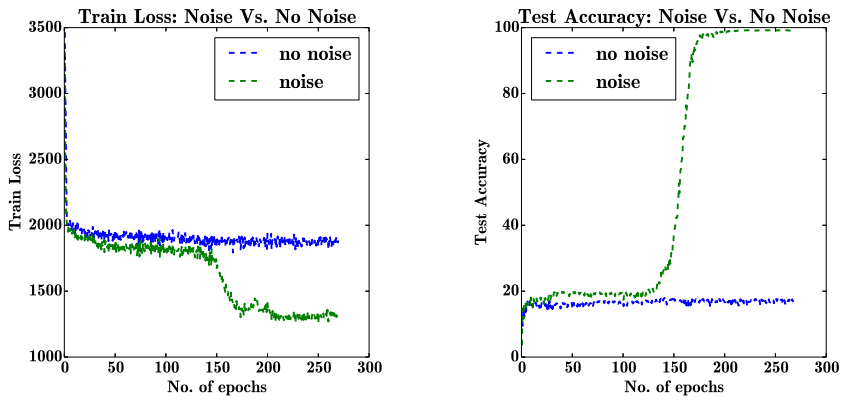

Figure 8: The effect of adding random noise to the gradients versus not adding it in our experiment with 5 columns when all hyper-parameters are the same. The models trained with noise generalizes almost always better.

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.

4 Related Work

Program induction has been studied in the context of semantic parsing (Zelle & Mooney, 1996; Zettlemoyer & Collins, 2005; Liang et al., 2011) in natural language processing. Pasupat & Liang (2015) develop a semantic parser with a hand engineered grammar for question answering on tables with natural language questions. Methods such as Piantadosi et al. (2008); Eisenstein et al. (2009); Clarke et al. (2010) learn a compositional semantic model without hand engineered compositional grammar, but still requiring a hand labeled lexical mapping of words to the operations. Poon (2013) 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. (2010) propose an hierarchical Bayesian approach to learn simple programs.

There has been some early work in using neural networks for learning context free grammar (Das et al., 1992a;b; Zeng et al., 1994) and context sensitive grammar (Steijvers, 1996; Gers & Schmidhuber, 2001) for small problems. Neelakantan et al. (2015); Lin et al. (2015) 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 (Bordes et al., 2014; Iyyer et al., 2014; Sukhbaatar et al., 2015; Peng et al., 2015; Hermann et al., 2015). While there has been lot of work in augmenting neural networks with additional memory (Das et al., 1992a; Schmidhuber, 1993; Hochreiter & Schmidhuber, 1997; Graves et al., 2014; Weston et al., 2015; Kumar et al., 2015; Joulin & Mikolov, 2015), 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, Neural Programmer-Interpreters (Reed & Freitas, 2016), a method that learns to induce programs with supervision of the entire program was proposed. This was followed by Neural Enquirer (Yin et al., 2015), which similar to our work tackles the problem of synthetic table QA. However, their method achieves perfect accuracy only when given supervision of the entire program. Later, dynamic neural module network (Andreas et al., 2016) was proposed for question answering which uses syntactic supervision in the form of dependency trees.

5 Conclusions

We develop Neural Programmer, a neural network model augmented with a small set of arithmetic and logic operations to perform complex arithmetic and logic reasoning. The model can be trained in an end-to-end fashion using backpropagation to induce programs requiring much lesser sophisticated human supervision than prior work. It is a general model for program induction broadly applicable across different domains, data sources and languages. Our experiments indicate that the model is capable of learning with delayed supervision and exhibits powerful compositionality.

Acknowledgements We sincerely thank Greg Corrado, Andrew Dai, Jeff Dean, Shixiang Gu, Andrew McCallum, and Luke Vilnis for their suggestions and the Google Brain team for the support.

References

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

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

Bahdanau, Dzmitry, Chorowski, Jan, Serdyuk, Dmitriy, Brakel, Philemon, and Bengio, Yoshua. End-to-end attention-based large vocabulary speech recognition. *arXiv preprint arxiv:1508.04395*, 2015.

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

Cantlon, Jessica F., Brannon, Elizabeth M., Carter, Elizabeth J., and Pelphrey, Kevin A. Functional imaging of numerical processing in adults and 4-y-old children. *PLoS Biology*, 2006.

Chan, William, Jaitly, Navdeep, Le, Quoc V., and Vinyals, Oriol. Listen, attend and spell. *arXiv preprint arxiv:1508.01211*, 2015.

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

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

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

Dastjerdi, Mohammad, Ozker, Muge, Foster, Brett L, Rangarajan, Vinitha, and Parvizi, Josef. Numerical processing in the human parietal cortex during experimental and natural conditions. *Nature communications*, 4, 2013.

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

Fias, Wim, Lammertyn, Jan, Caessens, Bernie, and Orban, Guy A. Processing of abstract ordinal knowledge in the horizontal segment of the intraparietal sulcus. *The Journal of Neuroscience*, 2007.

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

Graves, Alex. Generating sequences with recurrent neural networks. *arXiv preprint arxiv:1308.0850*, 2013.

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

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural Turing Machines. *arXiv preprint arxiv:1410.5401*, 2014.

Hannun, Awni Y., Case, Carl, Casper, Jared, Catanzaro, Bryan C., Diamos, Greg, Elsen, Erich, Prenger, Ryan, Satheesh, Sanjeev, Sengupta, Shubho, Coates, Adam, and Ng, Andrew Y. Deep Speech: Scaling up end-to-end speech recognition. *arXiv preprint arxiv:1412.5567*, 2014.

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

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

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

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

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

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

Kingma, Diederik P. and Ba, Jimmy. Adam: A method for stochastic optimization. *ICLR*, 2014.

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

Kucian, Karin, Loenneker, Thomas, Dietrich, Thomas, Dosch, Mengia, Martin, Ernst, and Von Aster, Michael. Impaired neural networks for approximate calculation in dyscalculic children: a functional mri study. *Behavioral and Brain Functions*, 2006.

Kumar, Ankit, Irsoy, Ozan, Su, Jonathan, Bradbury, James, English, Robert, Pierce, Brian, Ondruska, Peter, Gulrajani, Ishaan, and Socher, Richard. Ask me anything: Dynamic memory networks for natural language processing. *ArXiv*, 2015.

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

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

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

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

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

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

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

Peng, Baolin, Lu, Zhengdong, Li, Hang, and Wong, Kam-Fai. Towards neural network-based reasoning. *arXiv preprint arxiv:1508.05508*, 2015.

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

Piazza, Manuela, Izard, Veronique, Pinel, Philippe, Le Bihan, Denis, and Dehaene, Stanislas. Tuning curves for approximate numerosity in the human intraparietal sulcus. *Neuron*, 2004.

Poon, Hoifung. Grounded unsupervised semantic parsing. In *ACL*, 2013.

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

Schmidhuber, J. A self-referentialweight matrix. In *ICANN*, 1993.

Shang, Lifeng, Lu, Zhengdogn, and Li, Hang. Neural responding machine for short-text conversation. *arXiv preprint arXiv:1503.02364*, 2015.

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

Sukhbaatar, Sainbayar, Szlam, Arthur, Weston, Jason, and Fergus, Rob. End-to-end memory networks. *arXiv preprint arXiv:1503.08895*, 2015.

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

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

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

Von Neumann, John. First draft of a report on the EDVAC. Technical report, 1945.

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

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

Werbos, P. Backpropagation through time: what does it do and how to do it. In *Proceedings of IEEE*, 1990.

Weston, Jason, Chopra, Sumit, and Bordes, Antoine. Memory Networks. 2015.

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

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

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

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

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

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 4: 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.

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

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

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 7: 24 questions templates for questions of type “greater [number] sum” in the single column word variability experiment.

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 8: 10 questions templates for questions of type “[word] A sum B” in the two columns text match experiment.