NATURAL- TO FORMAL-LANGUAGE GENERATION USING TENSOR PRODUCT REPRESENTATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Generating formal-language represented by relational tuples, such as Lisp programs or mathematical expressions, from a natural-language input is an extremely challenging task because it requires to explicitly capture discrete symbolic structural information from the input to generate the output. Most state-of-the-art neural sequence models do not explicitly capture such structure information, and thus do not perform well on these tasks. In this paper we propose a new encoder-decoder model based on Tensor Product Representations (TPRs) for Natural- to Formal-language generation, called *TP-N2F*. The encoder of TP-N2F employs TPR 'binding' to encode natural-language symbolic structure in vector space and the decoder uses TPR 'unbinding' to generate a sequence of relational tuples, each consisting of a relation (or operation) and a number of arguments, in symbolic space. TP-N2F considerably outperforms LSTM-based Seq2Seq models, creating new state of the art results on two benchmarks: the MathQA dataset for math problem solving, and the AlgoList dataset for program synthesis. Ablation studies show that improvements are mainly attributed to the use of TPRs in both the encoder and decoder to explicitly capture relational structure information for symbolic reasoning.

1 INTRODUCTION

When people perform symbolic reasoning, they can easily describe the way to the conclusion step by step via relational descriptions. There is ample evidence that relational representations are important for human cognition (e.g., (Goldin-Meadow & Gentner, 2003; Forbus et al., 2017; Crouse et al., 2018; Chen & Forbus, 2018; Chen et al., 2019). Although a rapidly growing number of researchers use deep learning to solve complex symbolic-reasoning and language tasks, most existing deep learning models, including sequence models such as LSTMs, do not explicitly capture human-like relational structure information.

In this work, we propose a novel neural architecture—***TP-N2F***[1]—to solve natural- to formal-language generation tasks (N2F). In the tasks we study, math or programming problems are stated in natural language, and answers are given as programs, each program being a sequence of operator/arguments tuples—these are our relational descriptions. TP-N2F encodes natural-language symbolic structure in vector space, and decodes it as relational structures, both types of structures being embedded as **Tensor-Product Representations (TPRs)** (Smolensky, 1990). During encoding, this approach builds symbolic structures as vector-space embeddings using TPR 'binding' (Palangi et al., 2018); during decoding, it extracts symbolic constituents from structure-embedding vectors using TPR 'unbinding' (Huang et al., 2018; 2019).

As is typical of work using TPRs, it is useful to adopt three levels of description. At the highest **symbolic level**, we have the symbolic descriptions of the inputs and outputs: here, problems stated in natural language and solutions stated in a formal language of relational descriptions of program operations. At the lowest **neural-network level**, we have the activation vectors and connection-weight matrices of the model performing the task. At the intermediate level, unique to TPR approaches, the input and output symbol structures are decomposed into structural **roles**, which are filled by content-bearing symbols called **fillers**: this is the **role-structure level** spelled out in Section 2.

[1]Our code will be available at (URL)

Our contributions in this work are as follows. (i) We propose an analysis of N2F tasks at a role-structure level. (ii) We present a new Seq2Seq TP-N2F model which gives a neural-network-level implementation of a model solving the N2F task under our proposed role-structure-level description (i). To our knowledge, this is the first model to be proposed which combines both binding and unbinding features of TPRs to achieve generation tasks in deep learning. (iii) State-of-the-art performance on two recently-developed N2F tasks shows that the TP-N2F model has significant structure-learning ability on tasks requiring symbolic reasoning through program synthesis.

2 Background: Review of Tensor-Product Representation

The TPR mechanism is a method to create a vector-space embedding of complex symbolic structure. The type of a symbol structure is defined by a set of structural positions or **roles**, such as the *left-child-of-root* position in a tree, or the *second-argument* position of a particular relation. In a particular instance of a structural type, each of these roles may be occupied by a particular **filler**, which can be an atomic symbol or a substructure (e.g., the left sub-tree of a binary tree can serve as the filler of the role left-child-of-root). For now, we assume the fillers to be atomic symbols.

The TPR embedding of a symbol structure is the sum of the embeddings of all its constituents, each constituent comprising a role together with its filler. The embedding of a constituent is constructed from the embedding of a role and the embedding of the filler of that role: these are joined together by the TPR **binding** operation: the tensor (or generalized outer) product $\otimes$.

Suppose a symbolic type is defined by the roles $\{r_i\}$, and suppose that in a particular instance of that type, S, role r_i is bound by filler f_i. Then the TPR embedding of S is the order-2 tensor

$$\mathbf{T} = \sum_i \boldsymbol{f}_i \otimes \boldsymbol{r}_i = \sum_i \boldsymbol{f}_i \boldsymbol{r}_i^\top \tag{1}$$

where $\{\boldsymbol{f}_i\}$ are vector embeddings of the fillers and $\{\boldsymbol{r}_i\}$ are vector embeddings of the roles. As a simple example, consider the symbolic type string, and choose roles to be $r_1 =$ *first_element*, $r_2 =$ *second_element*, etc. Then in the specific string `S = cba`, the first role r_1 is filled by `c`, and r_2 and r_3 by `b` and `a`, respectively. Then the TPR for `S` is $\boldsymbol{c} \otimes \boldsymbol{r}_1 + \boldsymbol{b} \otimes \boldsymbol{r}_2 + \boldsymbol{a} \otimes \boldsymbol{r}_3$, where $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ are the vector embeddings of the symbols `a`, `b`, `c`, and $\boldsymbol{r}_i$ is the vector embedding of role r_i.

Define the matrix of all n_R possible role vectors to be $\boldsymbol{R} \in \mathbb{R}^{d_\mathrm{R} \times n_\mathrm{R}}$, with column i, $[\boldsymbol{R}]_{:i} = \boldsymbol{r}_i \in \mathbb{R}^{d_\mathrm{R}}$, comprising the embedding of r_i. Similarly let $\boldsymbol{F} \in \mathbb{R}^{d_\mathrm{F} \times n_\mathrm{F}}$ be the matrix of all possible filler vectors. The TPR $\mathbf{T} \in \mathbb{R}^{d_\mathrm{F} \times d_\mathrm{R}}$. $d_\mathrm{R}, n_\mathrm{R}, d_\mathrm{F}, n_\mathrm{F}$ are hyper-parameters, while $\boldsymbol{R}, \boldsymbol{F}$ are learned parameter matrices.

Choosing the tensor product as the binding operation makes it possible to recover the filler of any role in a structure S given the TPR $\mathbf{T}$ of S. This can be done with perfect precision if the embeddings of the roles are linearly independent. In that case the role matrix $\boldsymbol{R}$ is invertible: $\boldsymbol{U} = \boldsymbol{R}^{-1}$ exists such that $\boldsymbol{UR} = \boldsymbol{I}$. Now define the **unbinding** vector for role r_j, $\boldsymbol{u}_j$, to be the j^{th} column of $\boldsymbol{U}^\top$: $U^\top_{:j}$. Then, since $[\boldsymbol{I}]_{ji} = [\boldsymbol{UR}]_{ji} = \boldsymbol{U}_{j:}\boldsymbol{R}_{:i} = [\boldsymbol{U}^\top_{:j}]^\top \boldsymbol{R}_{:i} = \boldsymbol{u}_j^\top \boldsymbol{r}_i = \boldsymbol{r}_i^\top \boldsymbol{u}_j$, we have $\boldsymbol{r}_i^\top \boldsymbol{u}_j = \delta_{ji}$. This means that, to recover the filler of r_j in the structure with TPR $\mathbf{T}$, we can take its tensor inner product with $\boldsymbol{u}_j$ (or equivalently, viewing $\mathbf{T}$ as a matrix, take the matrix-vector product):

$$\mathbf{T}\boldsymbol{u}_j = \left[\sum_i \boldsymbol{f}_i \boldsymbol{r}_i^\top\right] \boldsymbol{u}_j = \sum_i \boldsymbol{f}_i \delta_{ij} = \boldsymbol{f}_j \tag{2}$$

In the architecture proposed here, we will make use of both TPR-binding using the tensor product with role vectors $\boldsymbol{r}_i$ and TPR-unbinding using the inner product with unbinding vectors $\boldsymbol{u}_j$.

3 TP-N2F Model

We propose a general TP-N2F neural-network architecture operating over TPRs to solve N2F tasks under a proposed role-structure-level description of the task. In this description, natural-language input is represented as a straightforward order-2 role structure, and formal-language relational representations of outputs are represented with a new order-3 recursive role structure proposed here. Figure 1 shows an overview diagram of the TP-N2F model.

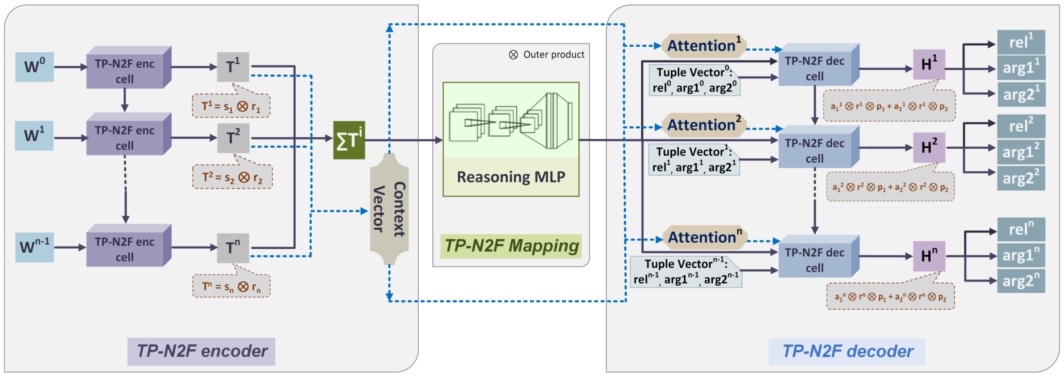

Figure 1: Overview diagram of TP-N2F.

As shown in Figure 1, while the natural language input is a sequence of words, the output is a sequence of multi-argument **relational tuples** such as $(R\ A_1\ A_2)$, a binary tuple consisting of a relation (or operation) R with its two arguments. TP-N2F contains a "TP-N2F encoder", which encodes the input natural-language sequence via TPR-binding, and a "TP-N2F decoder", which decodes relational tuples via TPR-unbinding. In the following sections, we first introduce the details of our proposed role-structure description for N2F tasks, and then present how our proposed TP-N2F model uses TPR binding- and unbinding-operations to create a neural-network implementation of this description of the N2F task.

3.1 Role-structure description of N2F tasks

In this section, we present our proposed role-structure description of N2F tasks, which specifies the form of the input natural-language symbolic expressions and the output relational representations.

3.1.1 Role-structure description for natural language

Instead of encoding each token of a sentence with a non-compositional embedding vector looked up in a learned dictionary, we use a learned role-filler decomposition to compose a tensor representation for each token from a role vector encoding the word's structural role in the sentence and a filler vector encoding the word's content. Given a sentence S with n word tokens $\{w^0, w^1, ..., w^{n-1}\}$, each word token w^t is assigned a learned role vector $\boldsymbol{r}^t$ and a learned filler vector $\boldsymbol{f}^t$ which, following the results of Palangi et al. (2018), we can hypothesize to approximately encode the grammatical role of the token and its lexical semantics, respectively. Then each word token w^t is represented by the tensor product of the role vector and the filler vector: $\mathbf{T}^t = \boldsymbol{f}^t \otimes \boldsymbol{r}^t$. In addition to the set of all its token embeddings $\{\mathbf{T}^0, \ldots, \mathbf{T}^{n-1}\}$, the sentence S as a whole is assigned a TPR equal to the sum of the TPR embeddings of all its word tokens: $\mathbf{T}_S = \sum_{t=0}^{n-1} \mathbf{T}^t$.

Using TPR for natural language has several advantages. First, natural language TPRs can be interpreted by exploring the distribution of tokens grouped by the role and symbol vectors they are assigned by a trained model (as in Palangi et al. (2018)). Second, TPRs avoid the Bag of Word (BoW) confusion (Huang et al., 2018): the BoW encoding of *Jay saw Kay* is the same as the BoW encoding of *Kay saw Jay*. However, in a TPR embedding using, say, grammatical roles, $\boldsymbol{f}_{Jay} \otimes \boldsymbol{r}_{subject} + \boldsymbol{f}_{saw} \otimes \boldsymbol{r}_{verb} + \boldsymbol{f}_{Kay} \otimes \boldsymbol{r}_{object}$ is different from $\boldsymbol{f}_{Kay} \otimes \boldsymbol{r}_{subject} + \boldsymbol{f}_{saw} \otimes \boldsymbol{r}_{verb} + \boldsymbol{f}_{Jay} \otimes \boldsymbol{r}_{object}$, because the role filled by a symbol changes with its context.

3.1.2 Role-structure description for relational representations

In this section, we propose a novel recursive role-structure description for representing symbolic relational tuples. Each relational tuple contains a relation token and multiple argument tokens. Given a binary relation R, a relational tuple can be written as $(R\ A_1\ A_2)$ where A_1, A_2 indicate two arguments of relation R. Let us adopt the two positional roles, $p_i^R = arg_i\text{-}of\text{-}R$ for $i = 1, 2$. The filler of role p_i^R is A_i. Now let us use role-structure recursively, noting that the role p_i^R can itself be decomposed into a sub-role $p_i = arg_i\text{-}of\text{-}\_$ which has a sub-filler R. Suppose that A_i, R, p_i are embedded as vectors $\boldsymbol{a}_i, \boldsymbol{r}, \boldsymbol{p}_i$. Then the TPR encoding of p_i^R is $\boldsymbol{r} \otimes \boldsymbol{p}_i$, so the TPR encoding of filler A_i bound to role p_i^R is $\boldsymbol{a}_i \otimes (\boldsymbol{r} \otimes \boldsymbol{p}_i)$. Since the tensor product is associative, we can write the TPR for the formal-language expression, the relational tuple $(R\ A_1\ A_2)$, as:

$$\mathbf{T}_F = \boldsymbol{a}_1 \otimes \boldsymbol{r} \otimes \boldsymbol{p}_1 + \boldsymbol{a}_2 \otimes \boldsymbol{r} \otimes \boldsymbol{p}_2. \tag{3}$$

Given the unbinding vectors $\boldsymbol{p}_i'$ for positional role vectors $\boldsymbol{p}_i$ and the unbinding vector $\boldsymbol{r}'$ for the vector $\boldsymbol{r}$ embedding relation R, each argument can be unbound in two steps as shown in (4)–(5).

$$\mathbf{T}_F \cdot \boldsymbol{p}_i' = [\boldsymbol{a}_1 \otimes \boldsymbol{r} \otimes \boldsymbol{p}_1 + \boldsymbol{a}_2 \otimes \boldsymbol{r} \otimes \boldsymbol{p}_2] \cdot \boldsymbol{p}_i' = \boldsymbol{a}_i \otimes \boldsymbol{r} \tag{4}$$

$$\boldsymbol{a}_i = [\boldsymbol{a}_i \otimes \boldsymbol{r}] \cdot \boldsymbol{r}_i' \tag{5}$$

Here $\cdot$ denotes the tensor inner product, which for the order-3 $\mathbf{T}_F$ and order-1 $\boldsymbol{p}_i'$ in (4) can be defined as $[\mathbf{T}_F \cdot \boldsymbol{p}_i']_{jk} = \sum_l [\mathbf{T}_F]_{jkl} [\boldsymbol{p}_i']_l$; in (5), $\cdot$ is equivalent to the matrix-vector product.

Our proposed scheme can be contrasted with the TPR scheme in which $(R\ A_1\ A_2)$ is embedded as $\boldsymbol{r} \otimes \boldsymbol{a}_1 \otimes \boldsymbol{a}_2$ (e.g., Smolensky et al. (2016); Schlag & Schmidhuber (2018)). In that scheme, an n-ary relation tuple is embedded as an order-$(n+1)$ tensor, and unbinding an argument requires knowing all the other arguments (to use their unbinding vectors). In the scheme proposed here, an n-ary relation tuple is still embedded as an order-3 tensor: there are just n terms in the sum in (3), using n position vectors $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$; unbinding simply requires knowing the unbinding vectors for these fixed position vectors.

3.1.3 The TP-N2F Scheme for Learning the input-output mapping

To generate formal relational tuples from natural language descriptions, a learning strategy for the mapping between the two structures is particularly important. As shown in (6), we formalize the learning scheme as learning a mapping function $f_{\text{mapping}}(\cdot)$, which, given a structural representation of the natural-language input, $\mathbf{T}_S$, outputs a tensor $\mathbf{T}_{FC}$ from which the structural representation $\mathbf{T}_F$ can be generated. In this paper, we use an MLP module to learn the TP-N2F mapping function. Other modules will be tested in future work.

$$\mathbf{T}_{FC} = f_{\text{mapping}}(\mathbf{T}_S) \tag{6}$$

3.2 The TP-N2F Model for Natural- to Formal-Language Generation

As shown in Figure 1, the TP-N2F model is implemented with three steps: encoding, mapping, and decoding. The encoding step is implemented by the TP-N2F natural language encoder (*TP-N2F Encoder*), which takes the sequence of word tokens as inputs, and encodes them via TPR-binding according to the TP-N2F role-structure scheme for natural language input given in Sec. 3.1.1. The mapping step is implemented by an MLP called the *Reasoning Module*, which takes the encoding produced by the TP-N2F Encoder as input. The Reasoning Module learns to map the natural-language-structure encoding of the input to a representation that will be assumed to follow the role-structure scheme for output relational-tuples specified in Sec. 3.1.2. The decoding step is implemented by the TP-N2F relational tuples decoder (*TP-N2F Decoder*), which takes the output from the reasoning MLP mapping (Sec. 3.1.3) and decodes the targeted sequence of relational tuples via TPR-unbinding. The TP-N2F Decoder utilizes an attention mechanism over the individual-word TPRs $\mathbf{T}^t$ produced by the the TP-N2F Encoder. The implementation of the TP-N2F Encoder, the TP-N2F Decoder, and the Reasoning Module are introduced in turn.

3.2.1 The TP-N2F Natural-Language Encoder

The TP-N2F encoder follows the role-structure scheme in Sec. 3.1.1 to encode each word token w^t by selecting one of n_{F} fillers and one of n_{R} roles. The fillers and roles are embedded as vectors. These embedding vectors, and the functions for selecting fillers and roles, are learned by two LSTMs, the *Filler-LSTM* and the *Role-LSTM*. (See Figure 2.)

At each time-step t, the Filler-LSTM and the Role-LSTM take a learned word-token embedding $\boldsymbol{w}^t$ as input. The hidden state of the Filler-LSTM $\boldsymbol{h}_{\text{F}}^t$ is used to compute softmax scores u_k^{F} over n_{F} filler slots, and a filler vector $\boldsymbol{f}^t = \boldsymbol{F}\boldsymbol{u}^{\text{F}}$ is learned from the softmax scores (recall $\boldsymbol{F}$ is the learned matrix of filler vectors). Similarly, a role vector is learned from the hidden state of the Role-LSTM $\boldsymbol{h}_{\text{R}}^t$. f_{F} and f_{R} indicate the functions to generate $\boldsymbol{f}^t$ and $\boldsymbol{r}^t$ from the hidden states of the two

LSTMs. The token w^t is encoded as $\mathbf{T}^t$, the tensor product of $\boldsymbol{f}^t$ and $\boldsymbol{r}^t$. $\mathbf{T}^t$ replaces the hidden vector in each LSTM and passes to the next time-step together with the LSTM cell-state vector $\boldsymbol{c}^t$: see (7)–(8). Detailed formulas are described in the Appendix. After encoding the whole sequence, the TP-N2F encoder outputs the sum of all tensor products $\sum_t \mathbf{T}^t$ to the next module.

$$\boldsymbol{h}_{\mathrm{F}}^t = f_{\mathrm{Filler-LSTM}}(\boldsymbol{w}^t, \mathbf{T}^t, \boldsymbol{c}_f^t) \quad \boldsymbol{h}_{\mathrm{R}}^t = f_{\mathrm{Role-LSTM}}(\boldsymbol{w}^t, \mathbf{T}^t, \boldsymbol{c}_r^t) \tag{7}$$

$$\mathbf{T}^t = \boldsymbol{f}^t \otimes \boldsymbol{r}^t = f_{\mathrm{F}}(\boldsymbol{h}_{\mathrm{F}}^t) \otimes f_{\mathrm{R}}(\boldsymbol{h}_{\mathrm{R}}^t) \tag{8}$$

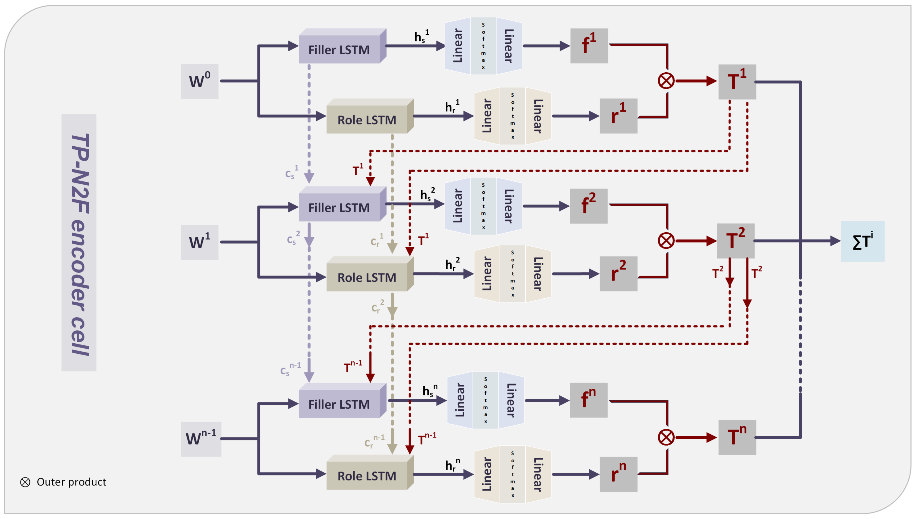

Figure 2: Implementation of TP-N2F encoder.

3.2.2 The TP-N2F Relational-Tuple Decoder

The TP-N2F decoder is an RNN that takes the output from the reasoning MLP as its initial hidden state for generating a sequence of relational tuples (see Figure 3). This decoder contains an LSTM called the *Tuple-LSTM* which feeds an *Unbinding Module*. Following Huang et al. (2018), the unbinding module is *TPR-ready*: it is designed so that *if* its input $\mathbf{H}$ is a TPR of a certain form, *then* it will unbind it correctly to produce a relation tuple. $\mathbf{H}$, which is produced by the Tuple-LSTM, is not constructed to explicitly be a TPR: rather, because the unbinding module operates on $\mathbf{H}$ as if it were a TPR, the Tuple-LSTM tends to learn a way to make $\mathbf{H}$ suitably approximate a TPR.

At each time-step t, the hidden state $\mathbf{H}^t$ of the Tuple-LSTM with attention (9) is fed as input to the unbinding module, which acts upon $\mathbf{H}^t$ as if it were the TPR of a relational tuple with m arguments possessing the role-structure described in Sec. 3.1.2: $\mathbf{H}^t \approx \sum_{i=1}^m \boldsymbol{a}_i^t \otimes \boldsymbol{r}^t \otimes \boldsymbol{p}_i$. In this paper we assume an arity of $m = 2$. (In Figure 3, the assumed hypothetical form of $\mathbf{H}^t$—as well as that of $\mathbf{B}_i^t$ below—is shown in a bubble.) Specifically, the unbinding module decodes a relational tuple from $\mathbf{H}^t$ using the two steps of TPR-unbinding given in (4)–(5). The positional unbinding vectors $\boldsymbol{p}_i'$ are learned during training and shared across all time-steps. After the first unbinding step (4)—inner product of $\mathbf{H}^t$ with $\boldsymbol{p}_i'$—we get tensors $\mathbf{B}_i^t$ (10). These are treated as the TPRs of two arguments $\boldsymbol{a}_i^t$ bound to a relation $\boldsymbol{r}^t$: $\mathbf{B}_i^t \approx \boldsymbol{a}_i^t \otimes \boldsymbol{r}^t$. A relational unbinding vector $\boldsymbol{r}'^t$ is computed by a Relation-Unbinding MLP from the sum of the $\mathbf{B}_i^t$ (11): both these vectors, by hypothesis, contain information about $\boldsymbol{r}'^t$. The inner product of each $\mathbf{B}_i^t$ with the vector $\boldsymbol{r}'^t$ yields vectors $\boldsymbol{a}_i^t$ which are treated as the embedding of relational arguments (11). Finally, symbolic values of the arguments and the relation of the generated relational-tuple are predicted by classifiers over the vectors $\boldsymbol{a}_i^t$ and $\boldsymbol{r}'^t$. (More detailed formulas are given in the Appendix.)

$$\mathbf{H}^t = Atten(f_{\mathrm{Tuple-LSTM}}(rel^{t-1}, arg_1^{t-1}, arg_2^{t-1}, \mathbf{H}^{t-1}, c^{t-1})) \tag{9}$$

$$\mathbf{B}_1^t = \mathbf{H}^t \boldsymbol{p}_1' \quad \mathbf{B}_2^t = \mathbf{H}^t \boldsymbol{p}_2' \tag{10}$$

$$\boldsymbol{r}'^t = f_{\mathrm{linear}}(\mathbf{B}_1^t + \mathbf{B}_2^t) \quad \boldsymbol{a}_1^t = \mathbf{B}_1^t \cdot \boldsymbol{r}'^t \quad \boldsymbol{a}_2^t = \mathbf{B}_2^t \cdot \boldsymbol{r}'^t \tag{11}$$

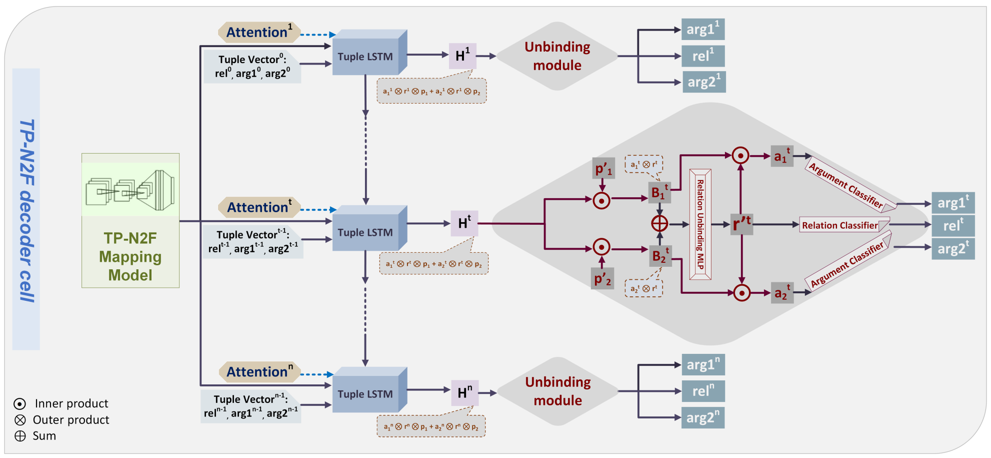

Figure 3: Implementation of TP-N2F decoder.

3.3 The Learning Strategy of the TP-N2F Model

TP-N2F is trained using back-propagation (Rumelhart et al., 1986) with the Adam optimizer (Kingma & Ba, 2017) and teacher-forcing. At each time-step, the ground-truth relational-tuple is provided as the input for the next time-step. As the TP-N2F decoder decodes a relational-tuple at each time-step, the relation token is selected only from the relation vocabulary and the argument tokens are selected only from the argument vocabulary. For an input $\mathcal{I}$ that generates N output relation-tuples, the loss is obtained by summing the cross-entropy loss $\mathcal{L}$ between the true labels L and predicted tokens for relations and arguments as shown in (12).

$$\mathcal{L}_{\mathcal{I}} = \sum_{i=0}^{N} \mathcal{L}(rel^i, L_{rel^i}) + \sum_{i=0}^{N}\sum_{j=1}^{2} \mathcal{L}(arg^i_j, L_{arg^i_j}) \tag{12}$$

4 Experiments

The proposed TP-N2F model is evaluated on two N2F tasks, generating operation-sequences to solve math problems and generating Lisp programs. In both tasks, TP-N2F achieves the state-of-the-art performance. We further analyze the behavior of the unbinding relation-vectors in the proposed model. Results of each task and analysis on unbinding relation-vectors are introduced in turn.

4.1 Generating operation-sequences to solve math problems

Given a natural-language math problem, we need to generate a sequence of operations (operators and corresponding arguments) from a set of operators and arguments to solve the given problem. Each operation is regarded as a relational-tuple by viewing the operator as relation, e.g., $(add, n1, n2)$. We test TP-N2F for this task on the MathQA dataset (Amini et al., 2019).

4.1.1 MathQA dataset

The MathQA dataset consists of about 37k math word-problems ((80/12/8)% training/dev/testing problems), each with a corresponding list of multiple-choice options and an operation-sequence program to solve the problem. An example from the dataset is presented in the Appendix.

In this task, TP-N2F is deployed to generate the operation sequence given the question. The generated operations are executed to generate the solution for the given math problem. We use the execution script from Amini et al. (2019) to execute the generated operation-sequence and compute the multi-choice accuracy for each problem. During our experiments we observed that there are about 30% noisy examples (on which the execution script fails to get the correct answer on the ground-truth program). Therefore, we report both execution accuracy (the final multiple-choice answer after running the execution engine) and operation sequence accuracy (where the generated operation sequence must match the ground-truth sequence exactly). Details about preparing the data and hyper-parameters of the TP-N2F model are described in the Appendix.

4.1.2 Results and discussion

TP-N2F is compared to a baseline provided by the seq2prog model in Amini et al. (2019), an LSTM Seq2Seq model with attention. Our model outperforms both the original seq2prog, designated SEQ2PROG-orig, and the best duplicated seq2prog after an extensive hyper-parameter search, designated SEQ2PROG-best. Table 1 presents the results. To verify the importance of the TP-N2F Encoder and Decoder, we conducted experiments to replace either the Encoder with a standard LSTM (denoted LSTM2TP) or the Decoder with a standard LSTM (denoted TP2LSTM). Both LSTM uses hidden size 100. We observe that both these TP-N2F components are important to achieve the performance gain relative to the baseline.

Table 1: Results on MathQA dataset testing set

MODEL	**Operation Accuracy(%)**	**Execution Accuracy(%)**
SEQ2PROG-orig	59.4	51.9
SEQ2PROG-best	66.97	54.0
TP2LSTM (ours)	68.84	54.61
LSTM2TP (ours)	68.21	54.61
TP-N2F (ours)	**71.89**	**55.95**

4.2 Generating program trees from natural language descriptions

Generating Lisp programs requires sensitivity to structural information because Lisp code can be regarded as tree-structured. Given a natural-language query, we need to generate code containing function calls with parameters. Each function call is a relational tuple, which has a function as the relation and parameters as arguments. We evaluate our model on the AlgoLisp dataset for this task and achieve state-of-the-art performance.

4.2.1 AlgoLisp dataset

The AlgoLisp dataset (Polosukhin & Skidanov, 2018) is a program-synthesis dataset, which has 79k/9k/10k training/dev/testing samples. Each sample contains a problem-description, a corresponding Lisp program-tree, and 10 input-output testing pairs. We parse the program tree into a sequence of commands from leaves to root and (as in MathQA) use the symbol $\#_i$ to indicate the result of the i^{th} command (generated previously by the model). A dataset sample with our parsed command sequence is presented in the Appendix.

AlgoLisp provides an execution script to run the generated program and has three evaluation metrics: accuracy of passing all test cases (Acc), accuracy of passing 50% of test cases (50p-Acc), and accuracy of generating an exactly-matched program (M-Acc). AlgoLisp has about 10% noise data (where the execution script fails to pass all test cases on the ground-truth program), so we report results both on the full test-set and the cleaned test-set (in which all noisy testing samples are removed). Details about hyper-parameters of the TP-N2F model are reported in the Appendix.

4.2.2 Results and discussion

TP-N2F is compared with an LSTM Seq2Seq with attention model and a Seq2Seq model with a pre-trained tree-decoder from the Tree2Tree autoencoder (SAPS) reported in Bednarek et al. (2019). As shown in Table 2, TP-N2F outperforms all existing models on both the full test set and the cleaned test set. Ablation experiments with TP2LSTM and LSTM2TP show that, for this task, the TP-N2F Decoder is more helpful than TP-N2F Encoder.

4.3 Interpretation of learned structure

To interpret the structure learned by the model, we extract the trained unbinding relation-vectors from the TP-N2F Decoder and reduce the dimension of vectors via Principal Component Analysis. K-means clustering results on the average vectors for each relation are presented in Figure 4. Results showed that unbinding-vectors for operators or functions with similar semantics tend to be closer

Table 2: Results of AlgoLisp dataset

	Full Testing Set			Cleaned Testing Set		
MODEL (%)	**Acc**	**50p-Acc**	**M-Acc**	**Acc**	**50p-Acc**	**M-Acc**
LSTM2LSTM+atten	67.54	70.89	75.12	76.83	78.86	75.42
TP2LSTM (ours)	72.28	77.62	79.92	77.67	80.51	76.75
LSTM2TPR (ours)	75.31	79.26	83.05	84.44	86.13	83.43
SAPSpre-VH-Att-256	83.80	87.45		92.98	94.15	
TP-N2F (ours)	**84.02**	**88.01**	**93.06**	**93.48**	**94.64**	**92.78**

with each other. For example, with 5 clusters in MathQA dataset, arithmetic operators such as *add, subtract, multiply, divide* are clustered together, and operators related to *square* or *volume* of geometry are clustered together. With 4 clusters in AlgoLisp dataset, partial/lambda functions and sort functions tend to be in one cluster, and string processing functions are clustered together. Note that there was no direct supervision to inform the model about the nature of the operations, and the TP-N2F decoder has induced this role structure with only weak supervision from natural-language-question/operation-sequence-answer pairs. More clustering results are presented in the Appendix.

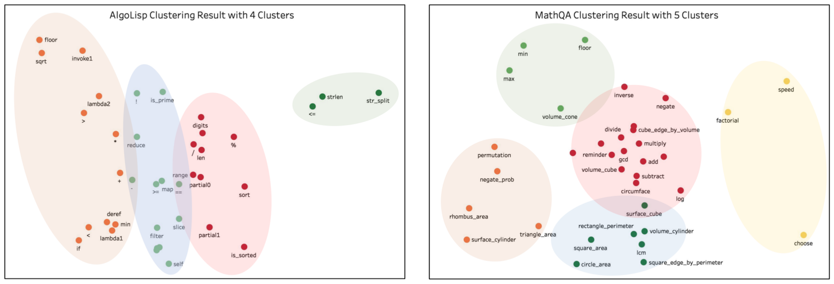

Figure 4: K-means clustering results: MathQA with 5 clusters and AlgoLisp with 4 clusters

5 Related work

TPR is a promising scheme for encoding symbolic structural information and modeling symbolic reasoning in vector-space. TPR-binding has been used for encoding and exploring grammatical structure information of natural language (Palangi et al., 2018; Huang et al., 2019). TPRs have also been used to unbind natural-language captions from images (Huang et al., 2018). Some researchers use TPRs for modeling deductive reasoning processes both on a rule-based model and deep-learning models in vector-space (Lee et al., 2016; Smolensky et al., 2016; Schlag & Schmidhuber, 2018). However, all of these works do not take advantage of combining TPR-binding and TPR-unbinding in deep learning models to learn structure representation mappings explicitly, as done in our model. Many researchers have explored generating formal-language expressions from natural-language descriptions. For example, operation-sequence generation from math problems has been formalized as a machine translation task on a word-token level and a modified LSTM Seq2Seq has been deployed on this task (Amini et al., 2019). Program-generation has been regarded as decoding tree-structure from natural language and a tree decoder has been utilized to generate program-trees (Polosukhin & Skidanov, 2018; Bednarek et al., 2019). Although researchers are paying increasing attention to N2F tasks, most of the proposed models either do not encode structural information explicitly or only fit specific tasks. Our proposed TP-N2F neural model can be applied to multiple tasks.

6 Conclusion and future work

In this work, we proposed a new scheme for neural-symbolic relational representations and a new architecture, TP-N2F, for formal-language generation from natural-language descriptions. To our knowledge, TP-N2F is the first model that combines TPRs-binding and TPRs-unbinding together in the encoder-decoder fashion. TP-N2F achieves the state-of-the-art on two instances of N2F tasks and shows significant structure-learning ability. The results show that both the TP-N2F Encoder and the TP-N2F Decoder are important for improving natural- to formal-language generation. We believe that the interpretation and symbolic structural encoding of TPRs are a promising direction for future work. We also plan to combine large-scale deep-learning models such as BERT with TP-N2F to take advantage of structure-learning for other generation tasks.

REFERENCES

Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. In *NACCL*, 2019.

Jakub Bednarek, Karol Piaskowski, and Krzysztof Krawiec. Ain't nobody got time for coding: Structure-aware program synthesis from natural language. In *arXiv.org*, 2019.

Kezhen Chen and Kenneth D. Forbus. Action recognition from skeleton data via analogical generalization over qualitative representations. In *Thirty-Second AAAI Conference*, 2018.

Kezhen Chen, Irina Rabkina, Matthew D. McLure, and Kenneth D. Forbus. Human-like sketch object recognition via analogical learning. In *Thirty-Third AAAI Conference*, volume 33, pp. 1336–1343, 2019.

Maxwell Crouse, Clifton McFate, and Kenneth D. Forbus. Learning from unannotated qa pairs to analogically disanbiguate and answer questions. In *Thirty-Second AAAI Conference*, 2018.

Kenneth.D. Forbus, Chen Liang, and Irina Rabkina. Representation and computation in cognitive models. In *Top Cognitive System*, 2017.

Susan Goldin-Meadow and Dedre Gentner. *Language in mind: Advances in the study of language and thought*. MIT Press, 2003.

Qiuyuan Huang, Paul Smolensky, Xiaodong He, Oliver Wu, and Li Deng. Tensor product generation networks for deep nlp modeling. In *NAACL*, 2018.

Qiuyuan Huang, Li Deng, Dapeng Wu, chang Liu, and Xiaodong He. Attentive tensor product learning. In *Thirty-Third AAAI Conference*, volume 33, 2019.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2017.

Moontae Lee, Xiaodong He, Wen-tau Yih, Jianfeng Gao, Li Deng, and Paul Smolensky. Reasoning in vector space: An exploratory study of question answering. In *ICLR*, 2016.

Hamid Palangi, Paul Smolensky, Xiaodong He, and Li Deng. Question-answering with grammatically-interpretable representations. In *AAAI*, 2018.

Illia Polosukhin and Alex Skidanov. Neural program search: Solving programming tasks from description and examples. In *ICLR workshop*, 2018.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In David E. Rumelhart, James L. McClelland, and the PDP Group (eds.), *Parallel distributed processing: Explorations in the microstructure of cognition*, volume 1, pp. 318–362. MIT press, Cambridge, MA, 1986.

Imanol Schlag and Jurgen Schmidhuber. Learning to reason with third order tensor products. In *Neural Information Processing Systems*, 2018.

Paul Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist networks. In *Artificial Intelligence*, volume 46, pp. 159–216, 1990.

Paul Smolensky, Moontae Lee, Xiaodong He, Wen-tau Yih, Jianfeng Gao, and Li Deng. Basic reasoning with tensor product representations. *arXiv preprint arXiv:1601.02745*, 2016.

A Appendix

A.1 Implementations of TP-N2F for experiments

In this section, we introduce the detailed implementation of TP-N2F on two datasets. We use $d_{\mathrm{R}}, n_{\mathrm{R}}, d_{\mathrm{F}}, n_{\mathrm{F}}$ to indicate the TP-N2F encoder hyper-parameters, the dimension of role vector, the number of roles, the dimension of filler vector and the number of fillers. $d_{Rel}, d_{Arg}, d_{Pos}$ indicate the TP-N2F decoder hyper-parameters, the dimension of relation vector, the dimension of argument vector, and the dimension of position vector.

In the experiment of MathQA dataset, we use $n_{\mathrm{F}} = 150$, $n_{\mathrm{R}} = 50$, $d_{\mathrm{F}} = 30$, $d_{\mathrm{R}} = 20$, $d_{Rel} = 10$, $d_{Arg} = 20$, $d_{Pos} = 5$ and we train the model 60 epochs with learning rate 0.00115. As most of the math operators in this dataset are binary, we replace all operators taking three arguments to a set of binary operators based on hand-encoded rules, and for all operators taking one argument, a padding symbol is appended.

In the experiment of AlgoLisp dataset, we use $n_{\mathrm{F}} = 150$, $n_{\mathrm{R}} = 50$, $d_{\mathrm{F}} = 30$, $d_{\mathrm{R}} = 30$, $d_{Rel} = 30$, $d_{Arg} = 20$, $d_{Pos} = 5$ and we train the model 50 epochs with learning rate 0.00115. For this dataset, most function calls take three arguments so we simply add padding symbols for those functions with arguments less than three.

A.2 Detailed formulas of TP-N2F

A.2.1 TP-N2F encoder

Filler-LSTM in TP-N2F encoder:

$$\boldsymbol{f}_f^t = \varphi(\boldsymbol{U}_{ff}\boldsymbol{w}^t + \boldsymbol{V}_{ff}\mathbf{T}^{t-1} + \boldsymbol{b}_{ff}) \tag{13}$$

$$\boldsymbol{g}_f^t = \tanh(\boldsymbol{U}_{fg}\boldsymbol{w}^t + \boldsymbol{V}_{fg}\mathbf{T}^{t-1} + \boldsymbol{b}_{fg}) \tag{14}$$

$$\boldsymbol{i}_f^t = \varphi(\boldsymbol{U}_{fi}\boldsymbol{w}^t + \boldsymbol{V}_{fi}\mathbf{T}^{t-1} + \boldsymbol{b}_{fi}) \tag{15}$$

$$\boldsymbol{o}_f^t = \varphi(\boldsymbol{U}_{fo}\boldsymbol{w}^t + \boldsymbol{V}_{fo}\mathbf{T}^{t-1} + \boldsymbol{b}_{fo}) \tag{16}$$

$$\boldsymbol{c}_f^t = \boldsymbol{f}_f^t * \boldsymbol{c}_f^{t-1} + \boldsymbol{i}_f^t * \boldsymbol{g}_f^t \tag{17}$$

$$\boldsymbol{h}_f^t = \boldsymbol{o}_f^t * \tanh(\boldsymbol{c}_f^t) \tag{18}$$

Filler vector:

$$\boldsymbol{a}_f^t = f_{sm}(f_{linear}(\boldsymbol{h}_f^t)/temperature) \tag{19}$$

$$\boldsymbol{f}^t = f_{linear}(\boldsymbol{a}_f^t) \tag{20}$$

Role-LSTM in TP-N2F encoder

$$\boldsymbol{f}_r^t = \varphi(\boldsymbol{U}_{rf}\boldsymbol{w}^t + \boldsymbol{V}_{rf}\mathbf{T}^{t-1} + \boldsymbol{b}_{rf}) \tag{21}$$

$$\boldsymbol{g}_r^t = \tanh(\boldsymbol{U}_{rg}\boldsymbol{w}^t + \boldsymbol{V}_{rg}\mathbf{T}^{t-1} + \boldsymbol{b}_{rg}) \tag{22}$$

$$\boldsymbol{i}_r^t = \varphi(\boldsymbol{U}_{ri}\boldsymbol{w}^t + \boldsymbol{V}_{ri}\mathbf{T}^{t-1} + \boldsymbol{b}_{ri}) \tag{23}$$

$$\boldsymbol{o}_r^t = \varphi(\boldsymbol{U}_{ro}\boldsymbol{w}^t + \boldsymbol{V}_{ro}\mathbf{T}^{t-1} + \boldsymbol{b}_{rO}) \tag{24}$$

$$\boldsymbol{c}_r^t = \boldsymbol{f}_r^t * \boldsymbol{c}_r^{t-1} + \boldsymbol{i}_r^t * \boldsymbol{g}_r^t \tag{25}$$

$$\boldsymbol{h}_r^t = \boldsymbol{o}_r^t * \tanh(\boldsymbol{c}_r^t) \tag{26}$$

Role vector:

$$\boldsymbol{a}_r^t = f_{sm}(f_{linear}(\boldsymbol{h}_r^t)/temperature) \tag{27}$$

$$\boldsymbol{r}^t = f_{linear}(\boldsymbol{a}_r^t) \tag{28}$$

A.2.2 TP-N2F DECODER

$$\boldsymbol{f}^t = \varphi(\boldsymbol{U}_f \boldsymbol{w}^t + \boldsymbol{V}_f \mathbf{H}^{t-1} + \boldsymbol{b}_f) \tag{29}$$

$$\boldsymbol{g}^t = tanh(\boldsymbol{U}_G \boldsymbol{w}^t + \boldsymbol{V}_G \mathbf{H}^{t-1} + \boldsymbol{b}_G) \tag{30}$$

$$\boldsymbol{i}^t = \varphi(\boldsymbol{U}_I \boldsymbol{w}^t + \boldsymbol{V}_I \mathbf{H}^{t-1} + \boldsymbol{b}_I) \tag{31}$$

$$\boldsymbol{o}^t = \varphi(\boldsymbol{U}_O \boldsymbol{w}^t + \boldsymbol{V}_O \mathbf{H}^{t-1} + \boldsymbol{b}_O) \tag{32}$$

$$\boldsymbol{c}^t = \boldsymbol{f}^t * \boldsymbol{c}^{t-1} + \boldsymbol{i}^t * \boldsymbol{g}^t \tag{33}$$

$$\boldsymbol{h}^t_{input} = \boldsymbol{o}^t * tanh(\boldsymbol{c}^t) \tag{34}$$

$$\boldsymbol{a}^t = f_{sm}(f_{score}(Context, f_{linear}(\boldsymbol{h}^t_{input}))) \tag{35}$$

$$\boldsymbol{c}^t_{weighted} = \sum_{i=0}^{n} (\boldsymbol{a}^t_i \boldsymbol{c}_i), \boldsymbol{a}^t_i \in \boldsymbol{a}^t, \boldsymbol{c}_i \in Context \tag{36}$$

$$\mathbf{H}^t = tanh(f_{linear}(concat(\boldsymbol{c}^t_{weighted}, \boldsymbol{h}^t_{input}))) \tag{37}$$

$$\mathbf{B}^t_1 = \boldsymbol{a}^t_1 \otimes \boldsymbol{r}^t = \mathbf{H}^t \boldsymbol{p}'_1 \tag{38}$$

$$\mathbf{B}^t_2 = \boldsymbol{a}^t_2 \otimes \boldsymbol{r}^t = \mathbf{H}^t \boldsymbol{p}'_2 \tag{39}$$

$$\boldsymbol{r}^t_{dual} = f_{linear}(B^t_1 + B^t_2) \tag{40}$$

$$\boldsymbol{a}^t_1 = \mathbf{B}^t_1 \boldsymbol{r}'^t \tag{41}$$

$$\boldsymbol{a}^t_2 = \mathbf{B}^t_2 \boldsymbol{r}'^t \tag{42}$$

$$Rel^t = Classifier_{rel}(\boldsymbol{r}'^t) \tag{43}$$

$$Arg1^t = Classifier_{arg}(\boldsymbol{a}^t_1) \tag{44}$$

$$Arg2^t = Classifier_{arg}(\boldsymbol{a}^t_2) \tag{45}$$

A.3 DATASET SAMPLES

A.3.1 DATA SAMPLE FROM MATHQA DATASET

Problem: The present polulation of a town is 3888. Population increase rate is 20%. Find the population of town after 1 year?
Options: a) 2500, b) 2100, c) 3500, d) 3600, e) 2700
Operations: multiply(n0,n1), divide(#0,const-100), add(n0,#1)

A.3.2 DATA SAMPLE FROM ALGOLISP DATASET

Problem: Consider an array of numbers and a number, decrements each element in the given array by the given number, what is the given array?
Program Nested List: (map a (partial1 b –))
Command-Sequence: (partial1 b –), (map a #0)

A.4 Generated programs comparison

In this section, we display some generated samples from the two dataset, where TP-N2F model generates correct programs but LSTM-Seq2Seq does not.

Question: A train running at the speed of 50 km per hour crosses a post in 4 seconds. What is the length of the train?
TP-N2F(correct):
$(multiply, n0, const1000)$ $(divide, \#0, const3600)$ $(multiply, n1, \#1)$ $(multiply, n1, \#1)$
LSTM(wrong):
$(multiply, n0, const0.2778)$ $(multiply, n1, \#0)$

Question: 20 is subtracted from 60 percent of a number, the result is 88. Find the number?
TP-N2F(correct):
$(add, n0, n2)$ $(divide, n1, const100)$ $(divide, \#0, \#1)$
LSTM(wrong):
$(add, n0, n2)$ $(divide, n1, const100)$ $(divide, \#0, \#1)$ $(multiply, \#2, n3)$ $(subtract, \#3, n0)$

Question: The population of a village is 14300. It increases annually at the rate of 15 percent. What will be its population after 2 years?
TP-N2F(correct):
$(divide, n1, const100)$ $(add, \#0, const1)$ $(power, \#1, n2)$ n$(multiply, n0, \#2)$
LSTM(wrong):
$(multiply, const4, const100)$ $(sqrt, \#0)$

Question: There are two groups of students in the sixth grade. There are 45 students in group a, and 55 students in group b. If, on a particular day, 20 percent of the students in group a forget their homework, and 40 percent of the students in group b forget their homework, then what percentage of the sixth graders forgot their homework?
TP-N2F(correct):
$(add, n0, n1)$ $(multiply, n0, n2)$ $(multiply, n1, n3)$ $(divide, \#1, const100)$ $(divide, \#2, const100)$ $(add, \#3, \#4)$ $(divide, \#5, \#0)$ $(multiply, \#6, const100)$
LSTM(wrong):
$(multiply, n0, n1)$ $(subtract, n0, n1)$ $(divide, \#0, \#1)$

Question: 1 is divided by 0.05 is equal to
TP-N2F(correct):
$(divide, n0, n1)$
LSTM(wrong):
$(divide, n0, n1)$ $(multiply, n2, \#0)$

Question: Consider a number a, compute factorial of a
TP-N2F(correct):
$(<=, arg1, 1)$ $(-, arg1, 1)$ $(self, \#1)$ $(*, \#2, arg1)$ $(if, \#0, 1, \#3)$ $(lambda1, \#4)$ $(invoke1, \#5a)$
LSTM(wrong):
$(<=, arg1, 1)$ $(-, arg1, 1)$ $(self, \#1)$ $(*, \#2, arg1)$ $(if, \#0, 1, \#3)$ $(lambda1, \#4)$ (len, a) $(invoke1, \#5, \#6)$

Question: Given an array of numbers and numbers b and c, add c to elements of the product of elements of the given array and b, what is the product of elements of the given array and b?
TP-N2F(correct):
$(partial, b, *)$ $(partial1, c, +)$ $(map, a, \#0)$ $(map, \#2, \#1)$
LSTM(wrong):
$(partial1, b, +)$ $(partial1, c, +)$ $(map, a, \#0)$ $(map, \#2, \#1)$

A.5 Unbinding relation vector clustering

We run K-means clustering on both dataset with $3, 4, 5, 6$ clusters and the results are displayed Figure 5 and Figure 6. As described before, unbinding-vectors for operators or functions with similar semantics tend to be closer to each other. For example, in MathQA dataset, arithmetic operators such as *add, subtract, multiply, divide* are clustered together at middle, and operators related to geometry such as *square* or *volume* are clustered together at bottom left. In AlgoLisp dataset, basic arithmetic functions are clustered at middle, and string processing functions are clustered at right.

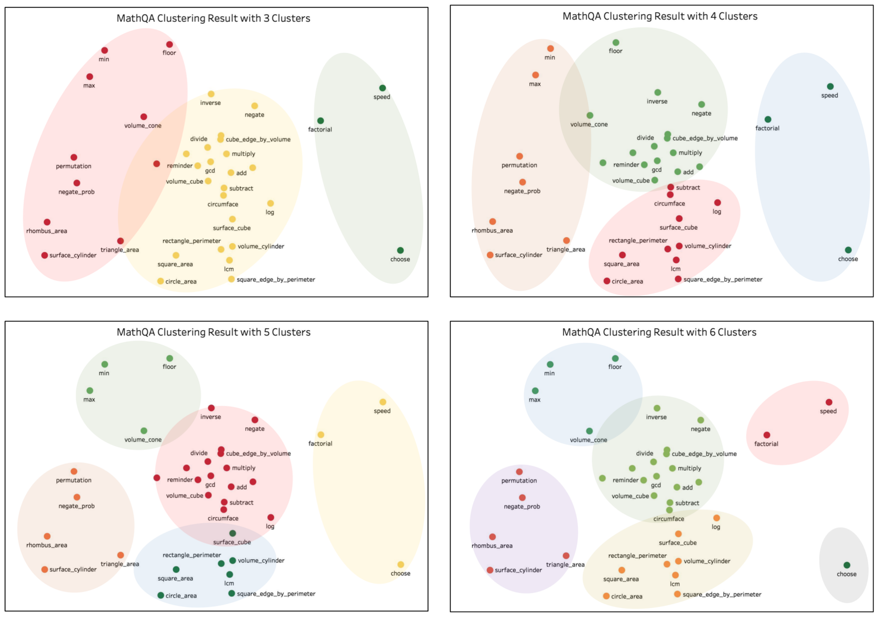

Figure 5: MathQA clustering results

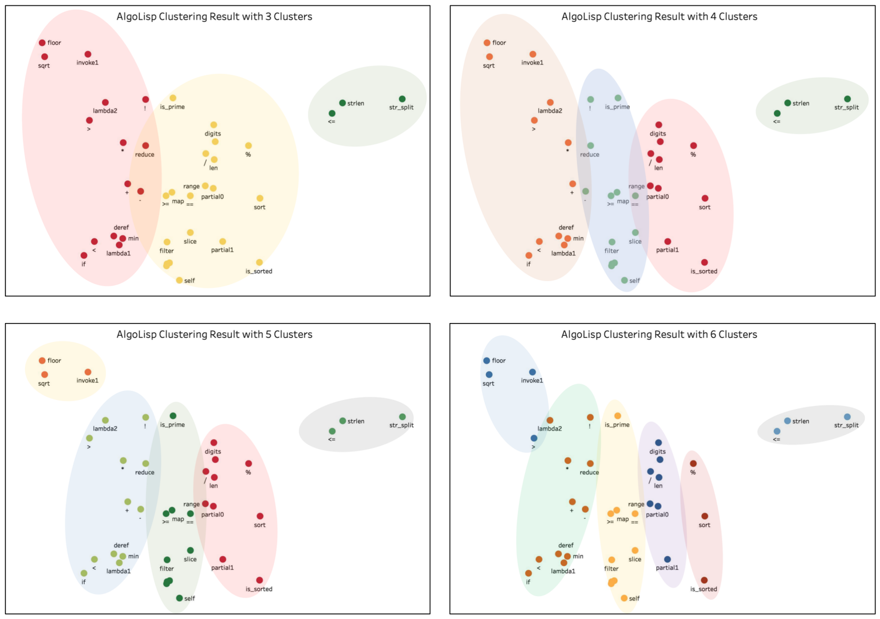

Figure 6: AlgoLisp clustering results