VQA-NPI: Implementation of NPI in Visual Question Answering

Pegah Hozhabrierdi
CIS700 Final Project - Novel Application
Department of Computer Science
Syracuse University

Abstract

A recently proposed network, NS-VQA (6), achieves the state-of-the-art accuracy for visual question answering task on CLEVR dataset. NS-VQA consist of two learning steps (scene representation and question parser) with an execution step which uses pre-written Python scripts for running the symbolic programs. In this project, I add a third learning step to the network and study the pros and cons of using a neural program interpreter (NPI) for its symbolic program execution. The results of my experiments show that although NPI will reduce the burden of hard-coding the 39 modules in NS-VQA, it needs strong supervision on the execution traces. For NPI to achieve or surpass the NS-VQA performance, accuracy of 100 percent is needed which was not obtained during my experiments.

1 Introduction

The idea of this study is derived from one of the most recent works on visual question answering (VQA) by Yi et al. (6). In their work, they combine three stages of deep visual representation, deep language representation, and symbolic program execution to find the answer of a question about a particular image (the steps are shown in figure . The element of "learning" is the core of the first two stages while the latter stage only benefits from pre-written Python script for each symbolic module that run sequentially without any learning involved. The existence of symbolic programs as a set of modules in association with a scene (the deep representation of the image), is similar to applications for which Reed et al. designed their Neural Program Interpreter (NPI) (5). So, can we use NPI to replace the execution step in VQA architecture? The current study is focused on answering this question. To the best of my knowledge, no existing work has used the combination of NPI and VQA together.
The NPI architecture is similar to that explained in (5) with fine-tuning of the hyper-parameters. To set up an NPI environment for fulfilling the VQA task, I needed to [1] build the training/testing data, [2] build embedding of the scene (deep representation of the image by VQA), and [3] tune the NPI network. The data was collected using the pre-trained VQA network on CLEVR dataset (4). CLEVR dataset is chosen due to the fact that NS-VQA network (6) has achieved the highest state-of-the-art accuracy on it (accuracy of 99.8 percent on 270 programs). Three repositories (VQA (1), CLEVR (2), and NPI (3)) are used in a direct or indirect manner for this project. More details on the dataset and architecture can be found in Section 2. The source code for this project is available on GitHub[1]

2 Methods

In this section, the architecture of VQA-NPI and the experimental set-up is discussed in details.

[1] `https://github.com/Pegayus/vqa-npi`

2.1 Architecture

VQA-NPI has been trained separately from NS-VQA network which categorizes this study under *New Application* track. Although NS-VQA network is not the focus of this work, the knowledge about its architecture is necessary to understand the purpose of VQA-NPI and NPI data/environment creation. Figure 1 shows NS-VQA architecture and how NPI can be implemented inside. Blue box will be replaced by NPI with two inputs: the embedding of the scene representation (called environment embedding) and execution traces of the set of modules. To build these inputs for NPI training/testing, I used the pre-trained NS-VQA image and question parser on CLEVR-min dataset to obtain (c) and (e) in Figure 1.

NPI Environment. VQA has 39 possible modules for each program [2] from which two modules *unit* and *intersect* are performed between two scenes. As NPI deals with a single environment representation of fixed size, I removed programs with these two modules from the dataset. The other 37 modules can be modeled as 10 higher (composite) and lower (atomic) NPI modules shown in Table 1. Each module can have from zero to two arguments and the possible options for each argument are also included in Table 1 [3]. Figure 2 shows example of how higher order modules are defined in terms of atomic modules. To perform *DELETE_ROW*, a valid bit is added to NS-VQA scene representation that shows whether the object exists (valid = 1) in table or not (valid = 0). VQA-NPI needs all programs, arguments, and termination terms to be encoded. Termination encoding is a single bit (0 for 'False' and 1 for 'True') and one-hot encoding is used for the other two as explained below.

Environment Embedding. The output of scene parser in Figure 1 is a table in the form of a list of dictionaries (one dictionary per object). The environment embedding is the combination of the scene + pointers (called environment), arguments and termination embedding. The scene embedding is a 10x20 array; 10 is the maximum number of objects in CLEVR dataset (4) and 20 is the respective one-hot encoding for all object attributes except for positions (positions are numbers and are kept as their own embedding) [4]. Pointers embedding is a 6x10 array (6 pointers with possible position from 0 to 9) and are appended to the scene embedding.

Execution Traces. The execution traces are of the form

((FILTER, 7), [0,1], False)

with 7 as the program ID for FILTER, 0 and 1 corresponding to *COLOR* and *BROWN* respectively, and *FALSE* for termination.

NPI Architecture. The NPI architecture is the one used by Reed et al. (5) and implemented in (3) for simple addition task with minor parameter modifications. The full architecture is shown in Figure 3. There are three different losses associated with VQA-NPI: program loss, arguments loss, and termination loss. These losses are calculated using cross entropy on the output of the last softmax layer. The learning rule is to minimize the loss function L defined as

$$L = 0.5 * (T_l + P_l) + A_l \tag{1}$$

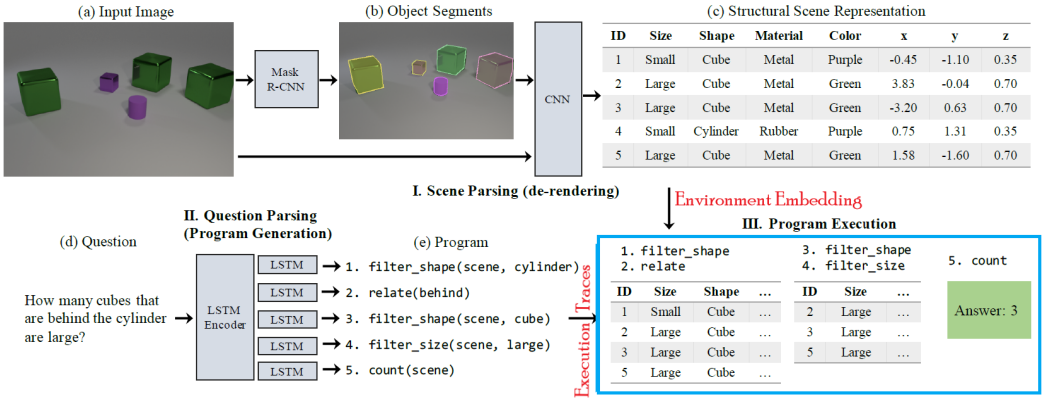
with T_l , P_l, and A_l as termination loss, program loss and argument loss respectively (0.5 is a hyper parameter that has worked well for addition task in (3) and can be tuned). I used 256 hidden layers, 0 regularization and learning rate of 0.0001. For encoding the environment, I used MLP with three dense layers according to Reed et al. (5).

Training. The Adam optimizer has been used for training with no drop-outs. The training and testing data are relatively big (8K for training and 2K+ for testing) and are accessible in the GitHub repo. However, NPI training is a slow procedure and using all the data on normal PC will not be practical. For the sake of preliminary experiments in the project, I worked with only the training set and a training size of 100 questions. For each experiment, the 100 questions are picked according to their category which is one of the 'query', 'count' and 'exist'. More details are discussed in Section 3.

[2] see the complete list of modules in `https://github.com/kexinyi/ns-vqa/blob/master/reason/executors/clevr_executor.py`

[3] In this implementation, I removed *RELATE* and *SAME* as the input to modules *relate* and *same* in NS-VQA do not depend on the scene alone and require the additional input of an object. To resolve this incompatibility within NPI architecture, I decided to develop my initial experiments without these two functions.

[4] x, y, z: one bit each (3), color: 8 bits, material: 2 bits, shape: 3 bits, size: 2 bits, valid: 2 bits



ID	Size	Shape	Material	Color	x	y	z
1	Small	Cube	Metal	Purple	-0.45	-1.10	0.35
2	Large	Cube	Metal	Green	3.83	-0.04	0.70
3	Large	Cube	Metal	Green	-3.20	0.63	0.70
4	Small	Cylinder	Rubber	Purple	0.75	1.31	0.35
5	Large	Cube	Metal	Green	1.58	-1.60	0.70

Figure 1: VQA architecture (6). Blue box will be replaced by NPI with two inputs: the embedding of the scene representation (called environment embedding) and execution traces of the set of modules.

Module	Arg 1	Arg 2
COUNT	[]	[]
COMPARE	[EQ, NEQ, GT, LT]	[]
MOVE_PTR	[ROW, X, Y, Z, COLOR, MATERIAL, SHAPE, SIZE]	[DOWN, RESET]
DELETE_ROW	[]	[]
QUERY	[X, Y, Z, COLOR, MATERIAL, SHAPE, SIZE]	[]
UNIQUE	[]	[]
FILTER	[COLOR, MATERIAL, SHPAE, SIZE]	[BLUE, BROWN, CYAN, GRAY, GREEN, PURPLE, RED, YELLOW, RUBBER, METAL, CUBE, CYLINDER, SPHERE, LARGE, SMALL]
EXIT	[]	[]
RELATE	[BEHIND, FRONT, LEFT, RIGHT]	[]
SAME	[COLOR, MATERIAL, SHPAE, SIZE]	[]

Table 1: NPI modules. Those in black are the atomic modules and those in blue are the composites which can be written in terms of the atomic modules. Each module can have zero, one or two arguments from the possible lists shown above. For example: FILTER[COLOR, BLUE] is a valid action.

2.2 Experimental Task

The problem that VQA-NPI tries to solve is to successfully find the sequence of programs that generate the answer to the VQA task. Specifically, if we have the symbolic execution trace of a certain query about a certain image, VQA-NPI is expected to execute the program successfully and give the correct output. Figure 2 depicts an example of what VQA-NPI is expected to do for a simple program of one execution trace FILTER[SIZE,LARGE]. The output of VQA-NPI will be the scene (on the right side of the figure) and the final answer to VQA task will be extracted from that scene. As a result, the most important task for VQA-NPI is to predict the next program and termination probability correctly.
The data that I have created for this task is a set of 10K+ questions (available in the repository) from 15K images [5]. Each question has the execution trace built from NS-VQA and the modules defined in Table 1. For VQA-NPI, the *type* of question is more important than the length of the execution trace. The learning curve of NPI for different types of questions is more deviated than that of different lengths of the trace for the same type of question. I have three types of questions in my dataset: query, count, and exist. For each type, there is a set of 100 questions under the name training_{type} in the repository. Each of these sets is divided into 80 question for training and 20 questions for testing.

3 Results

In this section, the results of experiments are reported. It is worth noting that many of these experiments are preliminary and can be improved in further trials.

[5]The original data I created had more than 140K questions and I picked this 10K from them based on the question types 'query', 'count', and 'exist'.

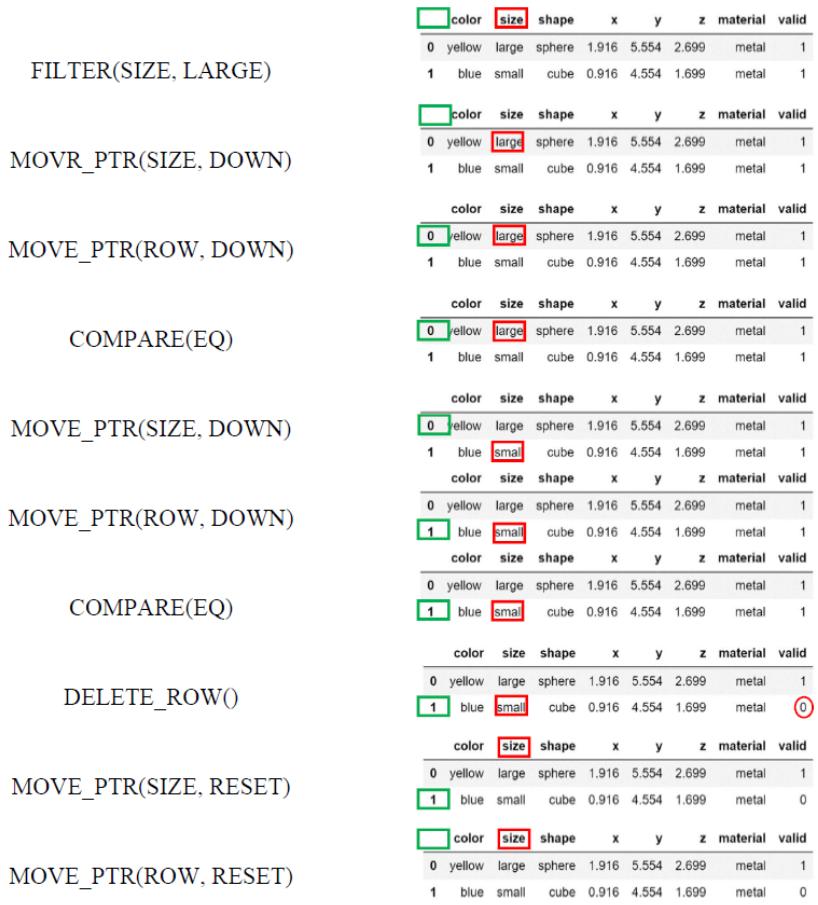


Figure 2: FILTER[SIZE, LARGE] procedure in terms of atomic modules. The figure on the right is the resulting scene of the corresponding module on the left. As indicated, only MOVE_PTR and DELETE_ROW change the scene. The red box shows size pointer and green box shows row pointer. The rest of the pointers (color, shape, positions and material) remain unchanged at the top.

3.1 General Results

As stated in previous section, I have chosen the generalization of the data according to the *type* than length of the execution trace. After splitting the data into categories'cou'exist', VQA-NPI was trained on one category (e.g. 'query') and tested on unseen data from the same category and the other two categories (generalization). The learning curves of these three experiments together with their losses are reported in Figures 4, 5, and 6. As NPI gives three different accuracies for program, termination and arguments, I also have reported three different curves for each training. As observed, all categories of questions generalize rather well for predicting the next program but they lack the termination prediction. The curious observation is that in all cases, the argument learning is poor at training stage while it can predict rather well in all testing samples.

3.2 Hyperparameter Tuning

I chose number of hidden layers (from 256 to 128), learning rate (from 0.0001 to 0.001), regularization term (from 0 to 0.01), number of dense layers in NPI encoding (from three to four) ,and weight of program and termination loss (from 0.5 in equation 1 to 0.2) as hyperparameters to change. Results of each are reported in Figure 7. Only program accuracy is shown in this report for brevity. Complete list of figures is available in the GitHub repository [6]. For all five experiments, the training has been done on 'query' and testing results are reported for the three different test sets (unseen from 'query',

[6] `https://github.com/Pegayus/vqa-npi/tree/master/hyper`

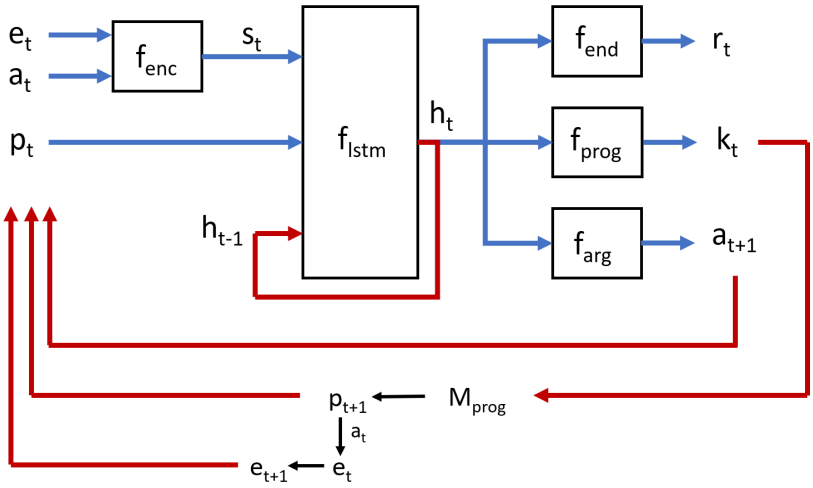


Figure 3: NPI architecture. {e, a, p, s, h, r, k} represent {environment, arguments, program, state, LSTM hidden state, probability of termination, program key embedding} at time t respectively.

'count', and 'exist'). As inferred from the figures, the number of hidden layers for LSTM can be safely reduced to 128 with minimum effect on the program accuracy. The same is true for reducing the number of dense layers in VQA-NPI environment encoding. On the other hand, increasing the learning rate from 0.0001 to 0.001 seems to degrade the performance of program prediction. The minimal increase in regularization term from 0 to 0.01 affects only the first steps of the program learning curve.

3.3 Ablation Study

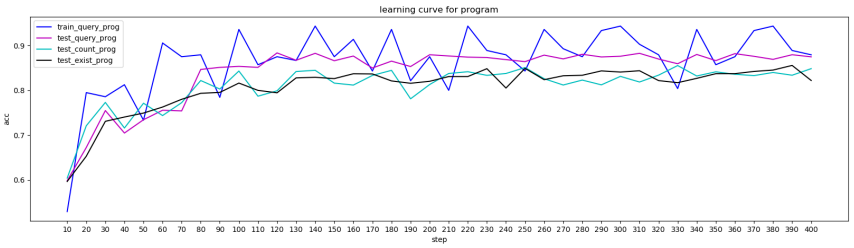
In this part, the given input to and expected output from the NPI is swapped based on some probability that is defined by a threshold (addition of noise to data). If the threshold is 0.7, then there is a 30 percent change that noise will be added to the current trace. The model has been trained previously on 80 samples (questions) with original architecture set-up. The ablation test is performed on 300 sample tests (training and testing are both from category 'query'). The result of the ablation test is shown in Figure 8.

4 Discussion

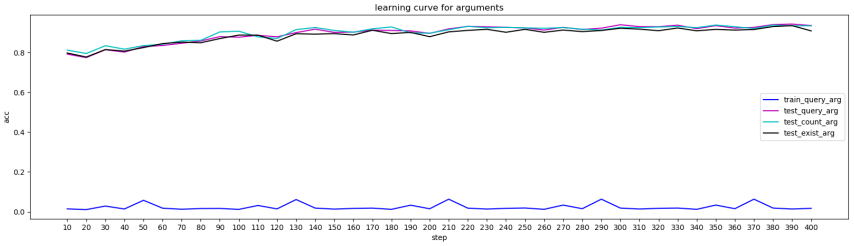
In this study, I implemented NPI for learning the VQA task by using the symbolic programs from NS-VQA network. The NPI was trained to predict the sequence of programs that will answer a certain question about a scene (image). The main results are

- VQA-NPI, as a separate module, is not sufficient to obtain NS-VQA executor accuracy. It does not achieve 100 percent accuracy needed to get the correct answer for the visual question answering. It seems to learn the programs much better than the arguments and the termination term. It's main weakness it the inability to predict the arguments. Despite rather good accuracy for termination, VQA-NPI biggest flaw is the inability to realize when to terminate a program even after the scene is empty. Any termination accuracy below 100 percent will result in a non-practical VQA-NPI for visual question answering.
- VQA-NPI is strongly dependent on the execution traces and with addition of noise on these traces (look results from ablation test) , its performance (specially in predicting the program and arguments) degrades considerably.
- VQA-NPI performance is affected the most by the changes in learning rate and rather robust against the alteration in the number of layers and other hyperparamters tested in this study.

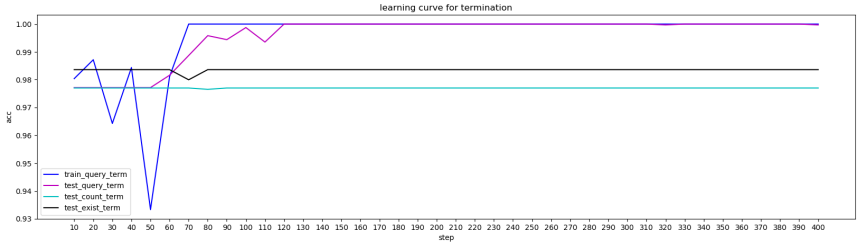
This trained network can be added to NS-VQA question parser and be trained end-to-end for *unrolling* the higher level programs (such as FILTER, EXIST, etc.). Unlike the VQA-NPI discussed here, the



(a) query program accuracy



(b) query arguments accuracy



(c) query termination accuracy

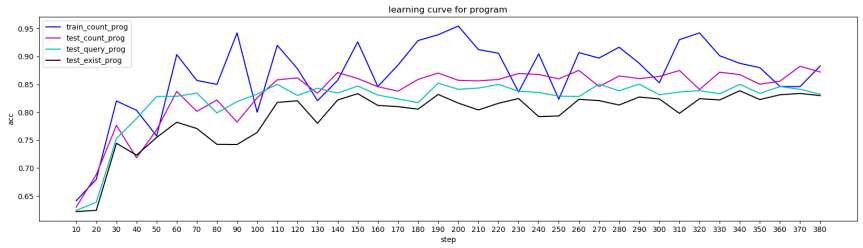
Figure 4: training, testing, and generalized testing for query.

NPI sequenced with NS-VQA question parser only needs to predict the next module if it is in the process of unrolling a higher level function. NS-VQA seq2seq parser provides the sequential list of modules and NPI will need to handle those that are not hard coded (higher level functions). Compared to the original executor in NS-VQA, NPI will need less hard-coding. However, NPI suffers from shortcomings that make it less practical than NS-VQA executor:

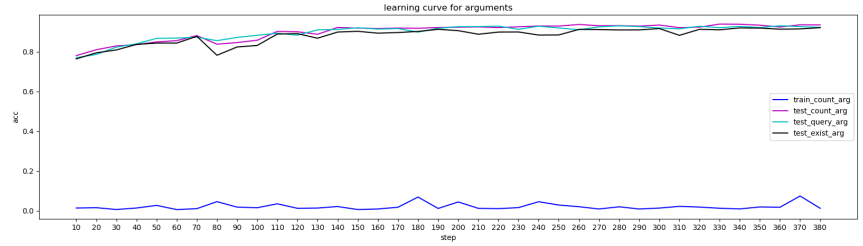
- NPI is heavily dependent on execution traces. Moreover, Generating these traces for training NPI is time consuming and less efficient than direct hard-coding.

- NPI will add another level of error into the NS-VQA system. Even if the question parser finds the right sequence of symbolic programs, NPI, with an accuracy of less than 90 percent in the majority of experiments, is very likely to predict wrong modules and ultimately giving a wrong answer.

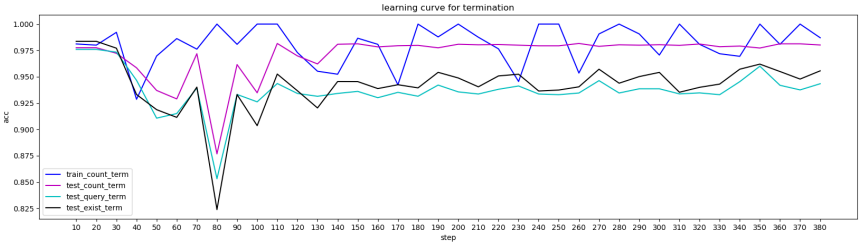
In the possible extension of this work, I will be focusing on attaching VQA-NPI into NS-VQA and train end-to-end. As a matter of fact, I tried to implement the modified NPI that will work directly with outputs from question parser. The minor differences in how NPI will be trained (predicting only the higher-level functions and evaluation the lower-level ones), makes a huge difference in the result; the program/termination/argument accuracy remains below 30 percent. It is interesting to see the results of training the NPI as done in this study and then re-train it as a modified version end-to-end with NS-VQA



(a) count program accuracy



(b) count arguments accuracy



(c) count termination accuracy

Figure 5: training, testing, and generalized testing for count.

References

[1] `https://github.com/kexinyi/ns-vqa`.

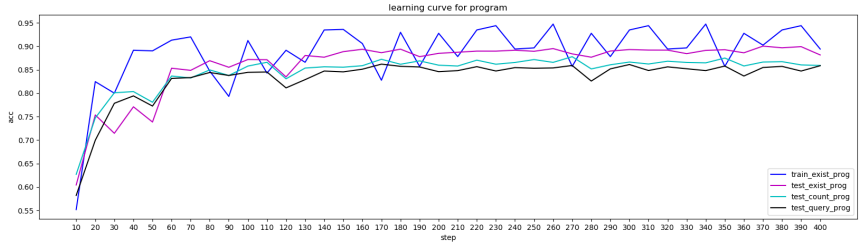
[2] `https://github.com/facebookresearch/clevr-dataset-gen`.

[3] `https://github.com/siddk/npi`.

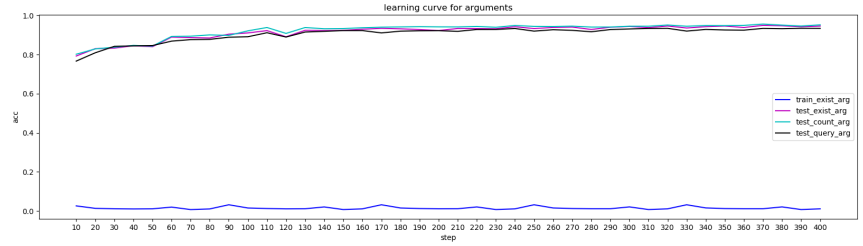
[4] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. `https://arxiv.org/abs/1612.06890/`, 2017.

[5] Scott Reed and Nando De Freitas. Neural programmer-interpreters. `https://arxiv.org/abs/1511.06279/`, 2015.

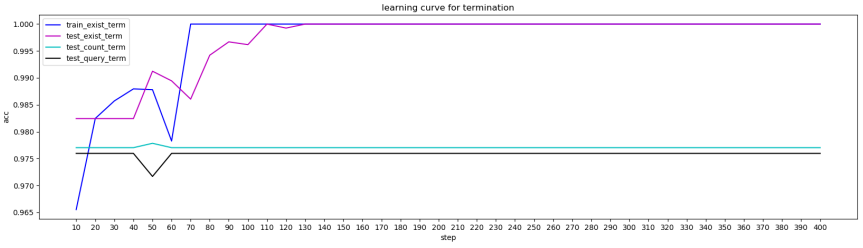
[6] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. `https://arxiv.org/abs/1810.02338/`, 2018.



(a) exist program accuracy

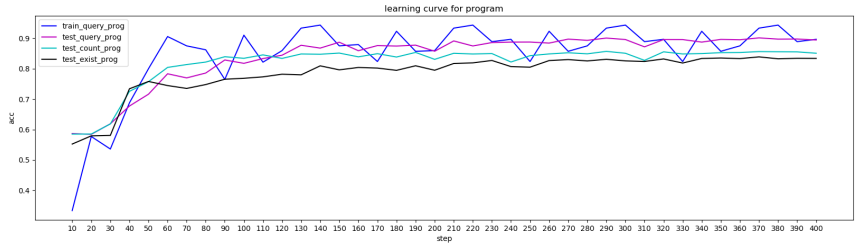


(b) exist arguments accuracy

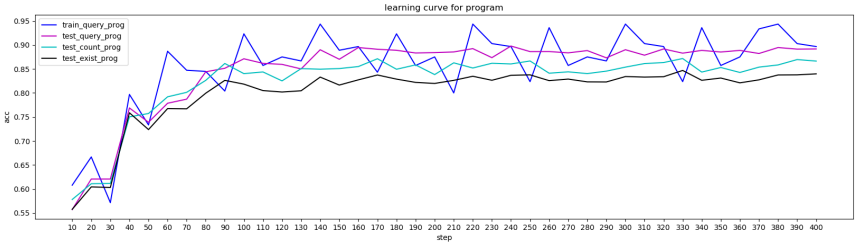


(c) exist termination accuracy

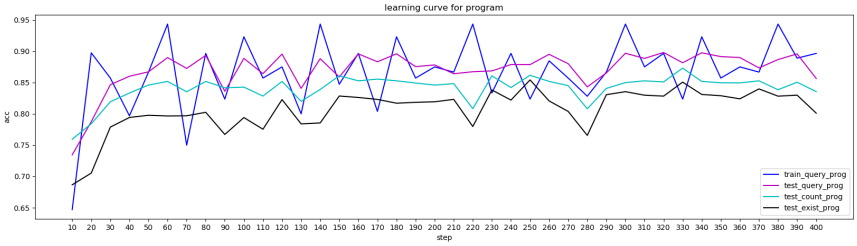
Figure 6: training, testing, and generalized testing for exist.



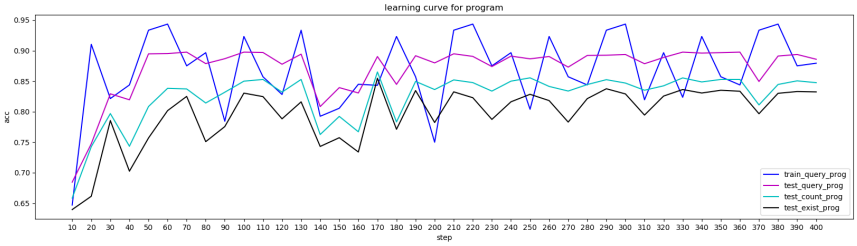
(a) layers = 128



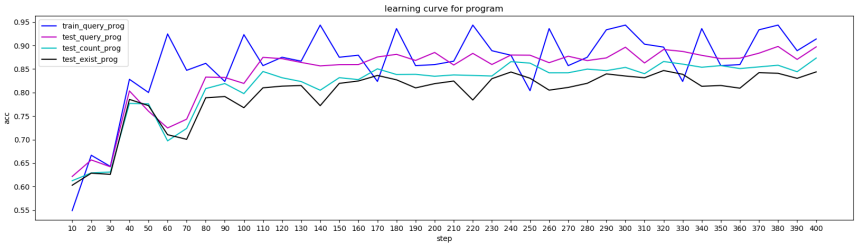
(b) VQA encoding with two dense layers



(c) learning rate = 0.001

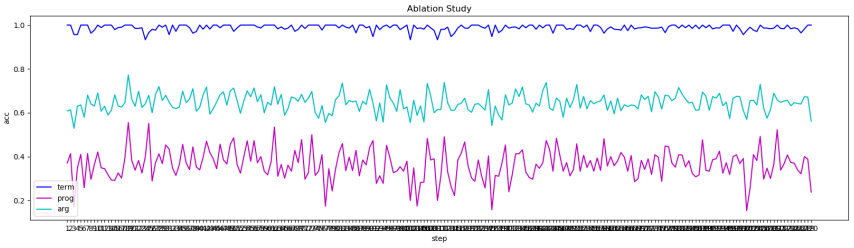


(d) program and termination weight = 0.2

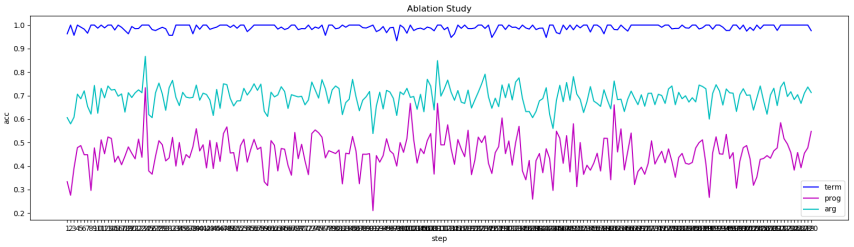


(e) regularization term = 0.01

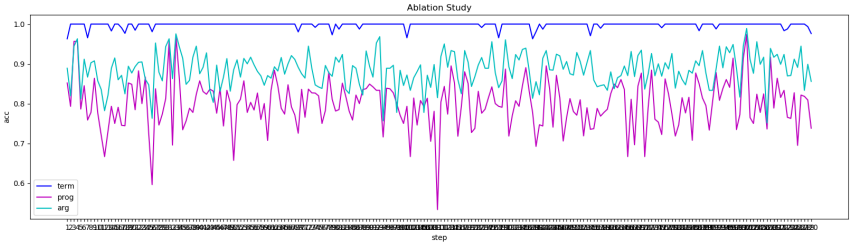
Figure 7: Program accuracy for five hyperparameter tuning.



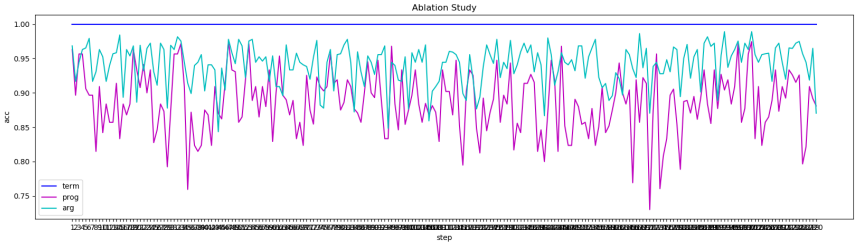
(a) threshold = 0.4



(b) threshold = 0.5



(c) threshold = 0.9



(d) threshold = 1

Figure 8: Ablation study with four different thresholds for adding noise to data