AI/ML Project Report

Visual Question Answering : Transparency By Design

Arjit Jain 170050010
Yash Sharma 17D070059
Aditya Vavre 170050089

November 2019

1 Introduction

The task of Visual Question Answering (VQA) is that of being able to correctly answer queries about an image. There are two major approaches to designing VQA systems. Parse the question into a series of logical operations, and then perform each operation over the image features. Or, Embed both the image and the question into a feature space, and the reason over the features jointly. While both approaches have had some really exciting results, there has been some amazing work on making the VQA system explainable using the former approach. We, through this course project, explore this recent work on explainable VQA. Being able to see the network reason about a complex task like VQA, involving a basic understanding of both Visual Intelligence and Language Semantics, can surely give us an insight into what these networks are learning and how could they be made better.

1.1 CLEVR

We have used the CLEVR dataset [1] for this task. This dataset has images containing some solid shapes like spheres, cylinders, and cubes of different colors placed in a certain configuration. Questions in this data set test various aspects of visual reasoning including attribute identification, counting, comparison, spatial relationships, and logical operations. Moreover, we have ground truth functional programs, which are a series of logical operations that can be performed to solve the question, which can be used to train a network using approach 1. The dataset comprises of roughly 700,000 questions and their respective programs as described above, and 70,000 images on which these questions are asked on.

1.2 Methods

We provide a brief summary of the methods used by Johnson et al.'s Inferring and Executing Programs for Visual Reasoning [2] for visual reasoning, and David et al's Transparency by Design [3].

1.2.1 IEP

Inferring and Executing Programs for Visual Reasoning, or IEP, used two components. The first component is called the Program Engine and is used to learn functional programs from questions. The second component is called the Execution Engine which, given the functional program, create a network which implements the program and passes the input image features

through this network.
Program Engine: The output of this component is a prefix tree of the input question. This task is posed as sequence to sequence learning and a LSTM based architecture to achieve this.
Execution Engine: For each function in the functional program, a module is maintained which should learn to implement the function. To execute a program, the modules corresponding to the functions are assembled, as dictated by the program, and the image features are passed through this assembly of modules.
This paper also goes through how each of these components should be trained based on the size of the dataset. We do not provide the training details here.

1.2.2 TbD

Transparency by Design, or TbD, networks inspire from IEP. TbD uses a pretrained IEP Program Engine. The difference from IEP is in the modules. By explicitly forcing the attention mechanism to be used, TdB ensures that attended regions are used in an intuitive way. Also, TdB uses low dimensional input image features to each module compared to IEP.

1.3 Room for Improvement

While both these approaches work really well and achieve state of the art accuracy, we found that there were some parts of the network which could be improved.

- **Inefficient Training**: The program for each question is different leading to a assembly of modules being generated. This way, the standard 'batch training' is not applicable to this setting as is.
- **MultiTask Learning**: We need different modules for inferring various properties about the objects in the image, like size, shape, color etc. While we will need to have different modules for each of the possible attributes (say blue, red etc for color) due to the program, we can have some form of weight sharing between these modules performing the same task.

As a result, we perform 2 modifications -

1. From an efficiency stand point as discussed, training individual examples in a batch becomes the bottleneck. We introduce a mechanism which can parallelize training of these assembly of modules as much as possible. This parallelization mechanism can be applied to other settings, which implement such dynamic flow of modules, too. This, to the best of our knowledge did not exist before.
2. For better performance on less data, we take a multitask learning approach, specifically hard parameter sharing. This makes sense because, as motivated above, the tasks of these different module instances are inherently related, say, detecting whether the colour of the object is blue is similar to detecting whether the colour of the object is red. We've attempted to share weights for modules performing similar tasks by sharing a base network, learning a general representation, and having different last layers, specific to each task.

2 Weight Sharing and Consolidating Various Modules

The original implementation used a different instance for each program token in the vocabulary. For example, the module to query whether the given object is of color red is trained completely independent of the one to query whether the given object is of color blue.

Seeing as these tasks are related, we can benefit if we have a base network learn a general representation for related tasks, and have different layers, which use this general representation for their specific tasks. For instance, using our method, we will have one network, whose weights are shared among all modules that are queried for a color, and each color, say red, will have its own specific layers, and these layers use the output of the shared network. We have applied this approach for the attention module that was common for a large number of different program tokens and the relate module that is a big network and is common for some program tokens. The other modules do not have large enough network and are not common across a large number of program tokens for them to make a significant difference when they are weight-shared across different program tokens.

2.1 Methodology and Implementation Details

To implement the above approach we have created base modules that share weights across program tokens and derived modules that have the program token specific final layer. The derived modules use the initial layers from the base module.
There is a single base module corresponding to each task, in our case, one for attention and one for relate. There are also derived modules corresponding to each program token that take as argument, the base module for weight sharing.

3 Program parse tree

As motivated previously, the main idea is to create trees of program execution for each example in the mini batch. Then, independent computations on the same module can be batched together. Care has to be taken so that the order of computation follows the program logic. We must also achieve this in a way which does not violate the automatic differentiation (backprop) of the underlying framework.

3.1 Methodology and Implementation Details

We create level trees for each program, and each node has the name of the function/module, the input to be used, and a level. We implement the tree in such a way so that the leaf with the maximum depth has level 1. And all subsequent nodes have level greater than or equal to this depending on their position in the tree. We use these levels as time steps, and so by construction, we have that all the inputs for any node at time t (or level t) would already have been computed in the previous time step (level t-1). The outputs for each example, on each level are stored and reused so that the history of the variables is unaffected and back-propagation using auto gradient methods works as expected

4 Results

Regarding time - as the standalone process our code took roughly 25 minutes per epoch, theirs is still running taking 35 minutes. When both processes were running parallely, ours took roughly 32 minutes per epoch, and there's over an hour. Infact, running parallely, our code finished 9 epochs while's theres barely finished 4. We've trained on a subset of 128000 training samples randomly chosen from the entire dataset, same for both our optimized code and the original. Validation has been performed on 25600 randomly chosen samples. As seen in the figure - results are very much comparable for the first 10 epochs.
All our code can be found at `https://github.com/ArjitJ/tbd-nets/`

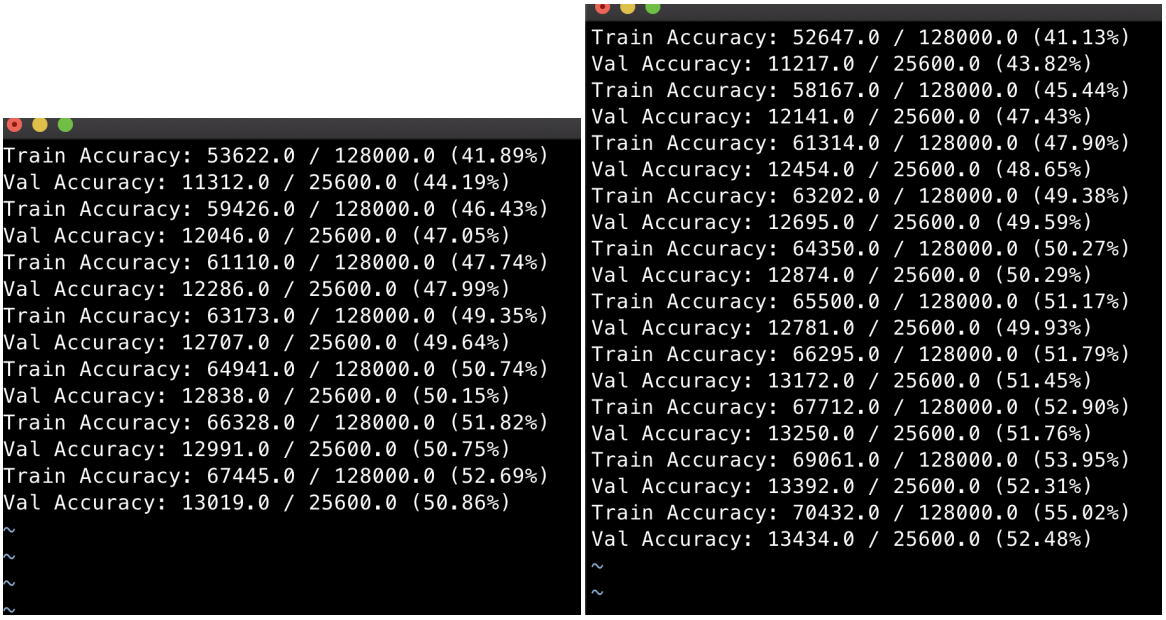

Figure 1: (a) In-progress accuracy on the original code, (b) Accuracy over 10 epochs on our optimized code

5 Conclusion

We explored some of the recent work done on explainable VQA systems. We provided two improvements to some of the current approaches based on computational efficiency and data/sample efficiency.

References

[1] Justin Johnson et al. “CLEVR: A Diagnostic Dataset for Compositional Language and Elementary Visual Reasoning”. In: *CVPR*. 2017.

[2] Justin Johnson et al. “Inferring and Executing Programs for Visual Reasoning”. In: *CoRR* abs/1705.03633 (2017). arXiv: 1705.03633. URL: http://arxiv.org/abs/1705.03633.

[3] David Mascharka et al. “Transparency by Design: Closing the Gap Between Performance and Interpretability in Visual Reasoning”. In: *CoRR* abs/1803.05268 (2018). arXiv: 1803.05268. URL: http://arxiv.org/abs/1803.05268.