HOW TO DESIGN SAMPLE AND COMPUTATIONALLY EFFICIENT VQA MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

In multi-modal reasoning tasks, such as visual question answering (VQA), there have been many modeling and training paradigms tested. Previous models propose different methods for the vision and language tasks, but which ones perform the best while being sample and computationally efficient? Based on our experiments, we find that representing the text as probabilistic programs and images as object-level scene graphs best satisfy these desiderata. We extend existing models to leverage these soft programs and scene graphs to train on question answer pairs in an end-to-end manner. Empirical results demonstrate that this differentiable end-to-end program executor is able to maintain state-of-the-art accuracy while being sample and computationally efficient.

1 INTRODUCTION

Many real-world complex tasks require both perception and reasoning (or System I and System II intelligence (Sutton & Barto, 2018)), such as VQA. What is the best way to integrate perception and reasoning components in a single model? Furthermore, how would such an integration lead to accurate models, while being sample and computationally efficient? Such questions are important to address when scaling reasoning systems to real world use cases, where empirical computation bounds must be understood in addition to the final model performance.

There is a spectrum of methods in the literature exploring different ways of integrating perception and reasoning. Nowadays, the perception is typically carried out via neural models: such as CNNs for vision, and LSTMs (Gers et al., 1999) or Transformers (Vaswani et al., 2017) for language. Depending on the representation of perception input and their reasoning interface, a method can be either more towards the neural end of the spectrum or more toward the symbolic end.

For the vision part, models can either use pixel-level or object-level symbolic representation. For the language part, models can generate either textual attention or programs, where the text is decomposed into a sequence of functions. Within the program representations, models typically operate on a selected discrete program or on probabilistic programs. The reasoning part used to produce the final answer can either use neural models, symbolic reasoning, or something in between, such as neural module networks (NMN) or soft logic blocks.

Existing works for NMN methods leverage pixel-level representations and program representations such as NMN (Hu et al., 2017), Prob-NMN (Vedantam et al., 2019), and Stack-NMN (Hu et al., 2018). Representative models that use object-level vision also leverage both neural and symbolic language and reasoning. Models that are more neural are LXMERT (Tan & Bansal, 2019) and NSM (Hudson & Manning, 2019), while those that are more symbolic are NS-VQA (Yi et al., 2018), NS-CL (Mao et al., 2019) and NGS (Li et al., 2020). A systematic comparison across these models is illustrated in Table 1 with more details in Appendix A.

Overall, neural models have more expressive power but with more parameters, while more-symbolic models have more prior structures built into them but with fewer parameters. There is an interesting bias-variance trade-off in the model design. By encoding as much bias into the model as possible, one could reduce sample requirements.

The different choices of perception and reasoning components also limit how the QA models will be trained. If both components are chosen as neural modules, then the training can be done in a very efficient end-to-end fashion. If the reasoning is carried out using more discrete operations,

	Vision		Language			Reasoning			Training	
	Pixel Att.	Object-level	Text Att.	Symbolic Progs.	Soft Progs.	Neural	Symbolic	Soft Logic	End-to-End	Sampling
LXMERT		✓	✓			✓			✓	
(Prob-)NMN	✓			✓				✓	✓	✓
Stack-NMN	✓				✓			✓	✓	
NSM		✓	✓			✓			✓	
NS-VQA		✓		✓			✓		✓	✓
NS-CL		✓		✓				✓	✓	✓
NGS		✓		✓			✓			✓

Table 1: A breakdown of VQA models by indicating which method is used with respect to their vision, language, inference, and training components. Refer to Appendix A for a detailed description of these methods.

then the perception model needs to sample discrete outputs or take discrete inputs to interface with downstream reasoning. For instance, if symbolic reasoning is used, REINFORCE (Williams, 1992) is typically used to train the perception models, which may require many samples during the optimization process. Alternatively, one can also use expensive abduction (Li et al., 2020) to manipulate the perception models outputs to provide the correct reasoning and then optimize these perception models using these pseudo-labels. Overall, more neural models will be easier to optimize, while more symbolic models will need additional expensive discrete sampling during optimization. To highlight this interesting fact, we call it the neuro-symbolic trade-off.

This neuro-symbolic trade-off also affects sample efficiency and computational efficiency. To be more sample efficient, the model needs to be less neural, yet, a more neural model can be more computationally efficient during training. Thus a method that can achieve an overall good performance in terms of both sample and computation efficiency will require systematically determining which perception and reasoning components should be used and how to integrate them. To design such a model, we first test which method within each perception and reasoning component works the most efficiently. From this neuro-symbolic trade-off exploration we can design a model that uses these most efficient components and compare its overall performance against existing models.

2 Problem Setting

Before the exploration, we formally define the different choices for the vision, language, and reasoning components. In the general VQA setting we are provided with an image I, a natural language question Q, and an answer A. We now define how these basic inputs are used in each component.

2.1 Representation for Vision

Given the image I there are two predominant visual representations: pixel and object-level attention.

Pixel Attention. Given an image one can leverage traditional deep learning architectures used for image representations and classification such as ResNets (He et al., 2016). Here the image is passed through many residual convolution layers before entering a MLP sub-network to perform a classification task. From one of these MLP linear layers, an intermediate dense image representation feature $f_I \in \mathbb{R}^D$ can be extracted, denoted by $f_I = \text{ResNet}(I)$. These features are used further down the VQA pipeline, where the downstream model computes attention over the relevant part of the feature based on the question asked.

Object-level. Another paradigm is to leverage object detection models such as Faster R-CNNs (Ren et al., 2015) to identify individual objects within images. Given objects in the image, one can

conduct more object-level or symbolic reasoning over the image, instead of reasoning through a pixel by pixel basis.

In this object-level representation, a set of object location bounding boxes (bbox) can be detected and labeled directly by using R-CNN as $O = \{(\text{bbox}_1, \text{label}_1), \cdots, (\text{bbox}_T, \text{label}_T)\} = \text{R-CNN}(I)$ for a preset number of T objects. Here $o \in O$ can be labeled as "small red shiny ball" or "large green tray" based on what is in the image.

Another approach is to factor the joint bounding box and label prediction into individual components to be handled by separate models. First the bounding boxes are extracted from the R-CNN as $\{\text{bbox}_i\}_{i=1}^{T} = \text{R-CNN}(I)$. Then these can be passed into a separate MLP network to retrieve the labels $\{\text{label}_i\}_{i=1}^{T} = \text{MLP}(\text{ResNet}(I[\text{bbox}_i]))$, where $I[\text{bbox}]$ is cropped image at that bounding box location. These can be used to define the final set of objects: $O = \{(\text{bbox}_i, \text{label}_i)_{i=1}^{T}\}$.

In such a setup, the benefit is that the R-CNN can be trained just as an object detector for a generic object class versus the background, whose annotations are easier to obtain. Furthermore, the number of supervised data the label MLP uses for training can be controlled separately. This is a useful mechanic during our model efficiency analysis where we work under the assumption that object bounding box is almost perfect, while object labeling is imperfect and expensive to annotate.

2.2 Representation for Language

The language representations operates on the natural text question Q. Some data sets also provide intermediate representations of each Q through a function program layout FP. FP represents the question as a sequence of abstract functions $\mathcal{F}$ as $FP = [\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_t]$ for $\mathcal{F}_i \in \mathcal{F}$. These function programs are used jointly with the visual representations to conduct reasoning to arrive at answer A. Details about potential realizations of $\mathcal{F}$ are described in the following reasoning representation section 2.3. Given the question Q and its representation FP we can define different approaches for representing the text.

Text Attention. Just using the embedded text tokens E a model can embed the question Q through a recurrent network to generate a final question representation h_T, where T is the maximum length sequence. Then h_T can be put through an recurrent decoder to obtain a latent function at each step c_t through an attentive combination of the hidden states $c_t = \sum_T a_t \cdot h_t$.

Symbolic Program. If we want to explicitly produce a FP for a corresponding question Q, we similarly encode the text as done for text attention. During decoding c_t is passed through a MLP to predict a valid function token. Then the most likely program is selected as $\arg\max_{FP} P(FP \mid Q)$.

Soft Program. When choosing a discrete symbolic program, the uncertainty of other function program parses is thrown out. Instead the probabilities for each program can be saved and an expected program can be computed as $\mathbb{E}[P(FP \mid Q)]$. Intuitively all the possible programs have to be considered in this scenario which can be intractable. Instead soft program methods such as Stack-NMN factor this as $\mathbb{E}[P(FP \mid Q)] = \mathbb{E}[\prod_T P(\mathcal{F}_t \mid Q)] = \prod_T \mathbb{E}[P(\mathcal{F}_t \mid Q)]$. This enables preserving a distribution of functions at each step instead of selecting a single one.

2.3 Representation for Reasoning

Given the visual and language representations, the reasoning component use these representations to derive the final answer A. Here we discuss methods that are neural, symbolic, and soft logic based.

Neural Reasoning. Reasoning can be made directly with the image feature f_I and encoded question h_T such as $A = \text{MLP}([h_T; f_I])$ in a purely neural fashion. Other approaches can leverage the object representations O. This is done by modulating the attention over which O correspond to final answer A, conditioned on h_T, as done in NSM or LCGN. LXMERT uses cross-modal attention between text embeddings E and O to predict the final answer. All these methods are more neural, but the FP can be leveraged as well to incorporate better biases through symbolic and soft programs.

Symbolic Representations. From the question we can define abstract functions $\mathcal{F}$ to generate FP as described in the previous section. Representing $\mathcal{F}$ in a symbolic form enables encoding general knowledge or certain dataset's domain specific language (DSL) into a model. This improves model

interpretability and provides better inductive biases as well. Here we further describe two classes of these functions: fine grained and coarse.

A fine grained representation of FP is sequence of n-ary predicates, functions with n arguments, composing $\mathcal{F}$. For example, given the question Q = "What is the shape of the thing left of the sphere?", a sample fine grained program can be defined as FP = [`filter_shape(sphere, O)`,`relate(left, O)`,`query_shape(O)`] Here the visual representation (O or f_I) and higher level argument concepts, such as `sphere`, are used as inputs to each function. We observe clear biases encoded into the function architecture, as given a scene graph of objects O and their relations, one could walk along this graph using FP to get the final answer A. The trade-off is that the model has to deal with more complex functions, whose input arguments and output types can vary. For example `relate_shape` and `relate` return a subset of objects, while `query_shape` returns a string. Such formulations are used by more neuro-symbolic methods such as NS-VQA and NS-CL.

Coarse function types consist of simpler predicates whose arity is fixed, typically 1, over $\mathcal{F}$. Given the previous question Q, a coarse function can be defined as FP = $[\texttt{filter}_\theta(f_I), \texttt{relate}_\theta(f_I), \texttt{query}_\theta(f_I)]$. Here less structure is required with respect to the language and visual representation where each function can be parameterized as a NMNs. These require more parameters than DSL functions but are syntactically easier to handle as they typically just operate on a fixed dimensional image feature f_I, thus implicitly encoding the function arguments.

Symbolic Reasoning. Using any coarse or fine representation type for $\mathcal{F}$, the symbolic reasoning can take place over the selected symbolic program FP. We define the high level execution of the symbolic reasoner to arrive at the answer by executing over FP as A = $\langle FP, \text{image representation} \rangle_S$. In the fine grained and coarse samples this would look like:

$$A = \langle FP, O \rangle_S = \texttt{query\_shape(relate(left, filter\_shape(sphere, O)))}$$

$$A = \langle FP, f_I \rangle_S = \texttt{query}_\theta\texttt{(relate}_\theta\texttt{(filter}_\theta\texttt{(}f_I\texttt{)))}$$

Since the structure of the reasoning is discrete, to update the visual and language model weights requires sampling based learning such as REINFORCE or abductive methods.

Soft Logic Reasoning. When conducting the symbolic reasoning we discard the uncertainty of the visual representations when generating the labels for O. Instead the probabilities for O can be saved. Here the uncertainty from the detections can be propagated during the execution in a soft manner to arrive at A. We can similarly define the soft logic reasoning as $A = \langle FP, I \rangle_{SL} = \mathbb{E}_{O \sim \text{R-CNN}(I)}[\langle FP, O \rangle_S]$. Due to the probabilistic execution, this can be optimized end-to-end.

Now that the components and their corresponding methods have been defined, we explore which methods are the most efficient for their respective task.

3 Neuro-Symbolic Trade-Off Exploration

Many deep VQA models have been developed in the literature with a range of design elements, as can be seen from Table 1. Which one of these methods is the key factor in making a deep VQA model sample efficient while at the same time achieving state-of-the-art? In this comparison using the CLEVR dataset (Johnson et al., 2017), we aim to understand which design elements individually perform the best. Based on these findings, better end-to-end models can be designed from the individual methods selected. More specifically, we will explore the sample and computational efficiency for the following components:

- Visual representations through pixel attention and object-level methods.
- Reasoning execution through neural modules, symbolic execution, and soft logic.
- Language encoding for questions through text attention, symbolic and soft programs.

For the representations of language and reasoning, we observe that these two components are tightly coupled. In language we define $\mathcal{F}$ and the corresponding FP given Q. In reasoning these functions

get executed for fine grained functions, or a network gets constructed from neural modules in the coarse case. For this reason we found it difficult to isolate the language and reasoning exploration. This motivated us to initially observe the interactions between the vision and reasoning given a fixed FP. Then by iteratively selecting the best vision and reasoning components, we explore the most efficient language and reasoning combination. Each method is also explained in more detail in Appendix D.

3.1 VISUAL REASONING

To test the visual perception and the reasoning components we break down the tests into two parts. First we first determine which visual representation is more efficient: pixel attention or object-centric. Second we find the reasoning method that best complements the selected visual representation.

Sample Efficiency

QA Data %	Pixel Att. Stack-NMN	Object-level NS-CL
1%	47.1	**96.1**
5%	66.8	**97.2**
10%	93.5	**98.8**

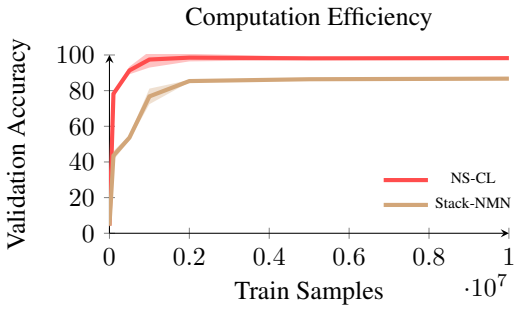

Figure 1: Sample and computational efficiency for pixel attention and object-level vision representations. The sample efficiencies experiments just use a total percentage of the available QA pairs and report the validation score after convergence. The computational efficiency is evaluated on the 10 % QA data and are run 5 times. The train samples indicate the number of examples seen by the model over time (iteration $\times$ batch size).

Pixel Attention versus Object-level. We compare pixel attention Stack-NMN and object-level NS-CL representation methods. We chose these two models as their visual representations differ but had continuous end-to-end reasoning methods given a fixed FP.

We set up this experiment by training and freezing each model's language components on CLEVR question program pairs (Q, FP) to isolate the effects of visual methods. Then the visual representation is trained from scratch using different percentages of CLEVR text question and answer (Q, A) pair data (QA Data %), where no object level supervision is provided.

The sample and computational efficiencies are illustrated in Table 1, which indicate **object-level** representations work better. Object-level detectors are able to leverage cheap object locations labels for localization and can leverage MLPs to classify objects given a dense feature $f_I \in \mathbb{R}^D$. In contrast, pixel-wise methods needs to learn the object attention maps through more parameter intensive convolutional layers over $I \in \mathbb{R}^{L \times W \times 3}$ where $D < L \times W \times 3$ channels.

Symbolic versus Soft Logic Reasoning. Since we will use fine grained object-level O information, we don't need to conduct parameter intensive pure neural reasoning over f_I. This lets us leverage of function programs FP. Similarly, given O, we don't use coarse NMN functions, which are only compatible operating over f_I. We focus on testing fine grained $\mathcal{F}$ to determine the best way to reason over the answer $A = \langle FP, I \rangle_*$, either using symbolic or soft logic.

We use NS-CL which already performs soft logic over (Q, A) pairs. To test the symbolic reasoning we replace the soft functions $\mathcal{F}$ defined in NS-CL by discrete counterparts, similar to the symbolic execution performed by NS-VQA. To train the symbolic reasoning with QA data, we test REINFORCE and abduction based optimization, denoted as NS-RL and NS-AB respectively. The results in Table 2 indicate that propagating the uncertainty of the object predictions through **soft logic** leads to gains in computational efficiency as well as final performance. Abduction has benefits over REINFORCE as it is able to selectively sample the object probabilities, which shows to be a more efficient procedure as the accuracy of the vision model increases.

Sample Efficiency

QA Data %	NS-RL	NS-AB	NS-CL
1%	47.1	59.8	**96.1**
5%	72.8	76.6	**97.2**
10%	81.3	84.2	**98.8**

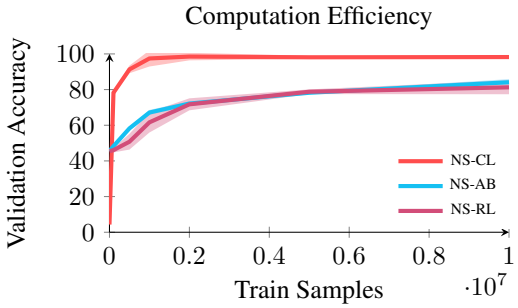

Figure 2: Sample and computational efficiency for soft versus symbolic reasoning. We take soft logic NS-CL and modify the reasoning to become symbolic. Then we optimize these symbolic methods through REINFORCE and abduction for NS-RL and NS-AB respectively.

At this point we have been testing the visual components and present the benefits of object-level representation and soft logic reasoning. Given these two methods, we now explore which language representation would be the most efficient to use.

3.2 Linguistic Reasoning

To test our language representation we want to determine the most efficient representation for Q. Taking into account the vision experiments, we find NS-CL's operation over object-level representations and executing soft logic reasoning over a fixed FP to be the most suitable approach. To build off of this, we want to understand the best representation of FP for reasoning. Therefore we look at symbolic and soft approaches. Recall the tight integration between the language representation and reasoning means that we investigate these two components in a joint fashion.

Sample Efficiency

QA Data %	Symbolic Progs. NS-CL	Soft Progs. Stack-NMN
1%	**86.5**	81.3
5%	89.2	**90.2**
10%	92.1	**94.3**

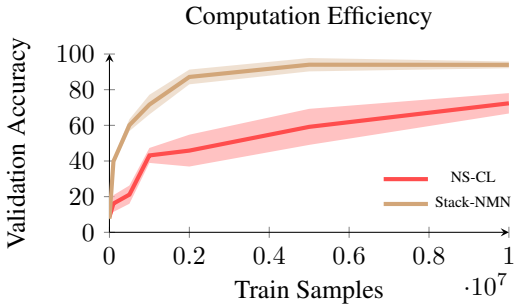

Table 2: Sample and computational efficiency for pixel attention and object-centric vision representations.

Symbolic versus Soft Execution. To test the language representation, we similarly train and freeze the visual representations to isolate the effects of the language models. These language models are then trained are also trained end-to-end on (Q, A) pairs without ground truth FP.

We compare NS-CL, which uses fine grained symbolic programs and soft logic reasoning over O to Stack-NMN which uses coarse soft programs and neural reasoning over f_I. For Stack-NMN the language and vision are trained jointly, but for NS-CL the language parser is trained disjointly from the vision. During testing the computation efficiency results, we equally divided the iterations by the curriculum learning setup described in their work. This led to more stable accuracy improvements over training on random samples as done in NS-VQA.

The results are presented in Table 2 and we generally see slower convergence than the vision trials. This is due to the fact that the program produced has to be perfect to reason correctly, while some mis-labeled objects may not lead to incorrect reasoning. Furthermore we encounter spurious predictions of incorrect FP leading to the correct answer, prolonging training.

Looking closer at the results, at 1% QA data, the symbolic representation is on top. We posit that this is due to model exhaustively sampling the program space over an end-to-end approach given limited data. However, as the amount of data increases, the **end-to-end soft programs** show better accuracies and computational efficiency. With the results from these language and vision experiments, we now have an understanding of which methods are the most efficient for VQA.

4 VISION AND LANGUAGE END-TO-END REASONING.

We iteratively experimented on different visual, language, and reasoning VQA components to determine which methods had the best accuracy and efficiency trade-offs. Based on this, we determined the following desiderata for efficient VQA methods:

- **Soft programs** to propagate language parsing uncertainty.
- **Object-level** detection models with pre-trained localization for object based reasoning.
- **Soft logic** functions to propagate the uncertainties from the vision and language branches.
- **End-to-end** differentiability to efficiently optimize the perception and reasoning jointly.

We are motivated to combine the existing Stack-NMN and NS-CL frameworks we tested to synthesize such a model. However, during our trade-off exploration we found it non-trivial to simply combine different methods across visual, language and reasoning components. We address two challenges regarding storing the intermediate computation in memory and the overall optimization when using these selected methods.

4.1 MEMORY

The first challenge is incompatibility at the reasoning level. NS-CL leverages fine grained functions that leverage objects O. The outputs of all of Stack-NMN's soft programs that operate only on f_I passed through coarse NMNs. To make such soft programs from Stack-NMN compatible with fine grained functions and object-level data, the memory storage handling the intermediate functional computation has to store object-level data as well.

We design such a memory structure by pre-assigning different object-level data modalities to different parts of a memory matrix $M \in \mathbb{R}^{T\times(A+C+1)}$. Here the first dimension T is the stack dimension used to store intermediate function outputs for future use, similar to Stack-NMN. The second $(A + C + 1)$ dimension are the rows that store the heterogeneous values while Stack-NMN only stores a D dimensional image feature f_I. The first A dimensions in the row indicate the object set attentions m_{det}, or which objects the model is paying attention to at that step. The C stores concatenated categorical value outputs possible by our vision models such as m_{color} and m_{size}. The final dimension m_{num} is reserved for some accumulated score such as `count` or a boolean bit for true or false questions. For the CLEVR specific case the object attentions $m_{det} \in \mathbb{R}^A$. The categorical values are $[m_{color}; m_{shape}; m_{texture}; m_{size}] \in \mathbb{R}^C$, and the numeric slot $m_{num} \in \mathbb{R}$.

This enables computing the reasoning softly over NS-CL like object-level predictions and Stack-NMN function program layouts as $\mathbb{E}_{O\sim\text{R-CNN}(I)}[\langle\prod_T \mathbb{E}[P(\mathcal{F}_t \mid Q)], O\rangle_S]$. After reasoning, the answers can be directly read from the memory instead of being inferred as a multi-class label. This is done by directly predicting the the question type from Q and accessing the corresponding memory location. For example if the question was asking about the color then we would return $\arg\max m_{color}$. Now that we have this fully differentiable architecture, we focus on the optimization procedure.

4.2 OPTIMIZATION

The second challenge is that Stack-NMN trains end-to-end while NS-CL iterates between REINFORCE for program parsing and end-to-end training for visual concept learning. NS-CL fixes the language model while end-to-end training vision, while we want to jointly optimizing both language and vision models. This results in a much larger optimization landscape prone to spurious predictions from end-to-end training on QA data.

To mitigate this we initially start by training the the vision MLP and language LSTM models with a small amount of the direct supervision, using between 0.1% - 1% of such data. This is done over the object (I[bbox], label) and language (Q, FP) pairs available in CLEVR. Then we train end-to-end on (Q, A) pair data where we have losses for categorical and numerical answers using, cross entropy and MSE losses respectively. Additionally, we found it useful to add the initial direct supervision losses and data when training end-to-end as well, weighted by hyperparameters α and β. We formulate this as a regularization method that prevents the models from diverging to spurious predictions when only provided with QA labels. We further provide details and demonstrate the efficacy of this regularization in Appendix C. All these terms give our overall loss as:

$$\mathcal{L}_{E2E} = \mathcal{L}_{QA\_XEnt} + \mathcal{L}_{QA\_MSE} + \alpha\mathcal{L}_{O\_XEnt} + \beta\mathcal{L}_{FP\_XEnt}$$

From these extensions over Stack-NMN and NS-CL we construct a fully Differentiable end-to-end Program executor (DePe). A more detailed description DePe's architecture and examples can be found in Appendix B.

5 Experiments

We built DePe using the desiderata over efficient VQA methods and now we test its overall end-to-end performance jointly on vision and language. First we compare it to our base NS-CL and Stack-NMN methods in terms of our desired sample and computational complexity. Then we compare DePe's accuracy against other VQA methods.

5.1 Efficiency Performance

Sample Efficiency

QA Data %	Stack-NMN	NS-CL	DePe
1%	43.6	**68.1**	65.4
5%	66.6	86.7	**87.7**
10%	80.6	**98.8**	98.0

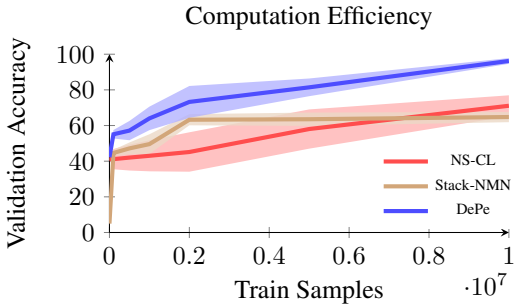

Figure 3: A comparison of efficiencies during joint training of vision and language. We test DePe with 0.1 % directly supervised labels.

We compare the efficiencies of DePe, Stack-NMN, and NS-CL in Table 3. To test the computational efficiency of NS-CL we trained the program parser till each iteration step, fixed the parser, and then trained the vision components end-to-end given the fixed parser.

Overall the results reflect the results reported during our component-wise testing. In terms of the computational efficiency, NS-CL (Mao et al., 2019) uses REINFORCE to optimize its FP parser, which requires many more iterations to converge. Stack-NMN (Hu et al., 2018) is also able to optimize end-to-end, but trains the f_I representation requiring more training samples. DePe using Stack-NMN soft programs and object-level execution from NS-CL is able to more efficiently optimize over either method alone.

In terms of sample efficiency NS-CL and DePe are comparable given enough data since the models execute similarly once NS-CL language models are fine tuned. Since DePe is trained on some directly supervised data we also attempted directly supervising Stack-NMN and NS-CL, but saw similar performance as discussed in the model detail in Appendix D.

Method	Vision	Language	Reasoning	Training	100% Data	10% Data
MAC	Pixel	Text Att.	Neural	E2E	98.9	67.3
TbD	Pixel	Symbolic Prog.	Neural	E2E	99.1	54.2
Prob-NMN	Pixel	Symbolic Prog.	Neural	E2E	97.1	88.2
Stack-NMN	Pixel	Soft Prog.	Neural	E2E	96.3	86.8
LXMERT	Object	Text Att.	Neural	E2E	97.9	66.5
LCGN	Object	Text Att.	Neural	E2E	97.9	64.8
NGS	Object	Symbolic Prog.	Symbolic	Abduction	**100.0**	87.3
NS-VQA	Object	Symbolic Prog.	Symbolic	RL	99.8	92.1
NS-CL	Object	Symbolic Prog.	Soft Logic	E2E + RL	99.2	98.8
DePe (0.1%)	Object	Soft Prog.	Soft Logic	E2E	99.0	98.0
DePe (1.0%)	Object	Soft Prog.	Soft Logic	E2E	99.5	**99.3**

Table 3: Performance of DePe and other VQA models using 100% and 10% QA pair supervision. Here we show DePe using 0.1% and 1% of the direct supervision on O and FP.

5.2 Comparative Performance

We present the accuracy comparison across different VQA models in Table 3. All models are able to achieve 96%+ given the full data set, but we are more interested in the results with lower sample complexity. At 10% data we observe that methods with continuous inference and DSL based neural modules, such DePe, NS-CL, Stack-NMN and Prob-NMN (Vedantam et al., 2019) scale better. The other methods are more on the neural side, such as TbD (Mascharka et al., 2018), MAC (Hudson & Manning, 2018), LXMERT (Tan & Bansal, 2019), and LCGN (Hu et al., 2019), require more data to converge. Methods that discretely sample, such as NS-VQA (Yi et al., 2018) or NGS (Li et al., 2020), can also achieve high accuracies given more training data, but in practice, require many iterations to converge and result in high variance results compared to continuous methods.

6 Conclusion and Future Work

In VQA there are different paradigms of modeling the visual, language, and reasoning representations. In addition to the final model performance, we are interested in understanding model efficiency in these complex perception and reasoning tasks. First we introduced the existing models with their corresponding vision, language, reasoning, and training components used. Then we formally defined these components and the common method representations used within each component. In order to determine which methods were the most sample and computationally efficient, we iteratively tested the vision, reasoning, and language components. These results showed that object-level, soft program, soft logic, and end-to-end training were important for efficient VQA. Following this we modified existing models to leverage all of these efficient methods in a joint manner. We showed that this model, DePe, was the most efficient while retaining state-of-the-art performance.

Based on these methods we look forward to testing DePe on larger real-world data sets. Since our model uses generic function programs to operate over language and vision it can be extended to different data sets with minimal modifications. Furthermore we plan to investigate how to use concepts embeddings similar to NS-CL within our memory instead of one-hot representations. We will also be interested in testing how the object-level representations work on questions involving both object and image level reasoning.

REFERENCES

Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pp. 2961–2969, 2017.

Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to reason: End-to-end module networks for visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 804–813, 2017.

Ronghang Hu, Jacob Andreas, Trevor Darrell, and Kate Saenko. Explainable neural computation via stack neural module networks. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 53–69, 2018.

Ronghang Hu, Anna Rohrbach, Trevor Darrell, and Kate Saenko. Language-conditioned graph networks for relational reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 10294–10303, 2019.

Drew Hudson and Christopher D Manning. Learning by abstraction: The neural state machine. In *Advances in Neural Information Processing Systems*, pp. 5903–5916, 2019.

Drew A Hudson and Christopher D Manning. Compositional attention networks for machine reasoning. In *International Conference on Learning Representations*, 2018.

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. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2901–2910, 2017.

Qing Li, Siyuan Huang, Yining Hong, Yixin Chen, Ying Nian Wu, and Song-Chun. Zhu. Closed loop neural-symbolic learning via integrating neural perception, grammar parsing, and symbolic reasoning. In *International Conference on Machine Learning (ICML)*, 2020.

Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=rJgMlhRctm`.

David Mascharka, Philip Tran, Ryan Soklaski, and Arjun Majumdar. Transparency by design: Closing the gap between performance and interpretability in visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4942–4950, 2018.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pp. 91–99, 2015.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Hao Tan and Mohit Bansal. Lxmert: Learning cross-modality encoder representations from transformers. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 5103–5114, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Ramakrishna Vedantam, Karan Desai, Stefan Lee, Marcus Rohrbach, Dhruv Batra, and Devi Parikh. Probabilistic neural-symbolic models for interpretable visual question answering. *arXiv preprint arXiv:1902.07864*, 2019.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *Advances in neural information processing systems*, pp. 1031–1042, 2018.

A MODEL COMPARISONS

We explore a few comparative works that contain a variety of training strategies used for VQA. Each method type handles the vision, language, reasoning and training in different fashions.

LXMERT. Reasoning over images based on questions is carried out via Transformer like deep architecture. Such neural reasoning module can be easily interfaced with neural perception modules, and optimized end-to-end with perception module jointly. Since such model incorporate few prior structures into the model, it contains lots of parameters and requires lots of question-answer pairs in order to achieve good results.

NMN methods. With Neural Module Networks (NMN), the language is turned into a discrete function program over neural modules which act on pixel level attention to answer questions. The discrete function program allows reasoning steps to be executed exactly. However, such design also makes the entire model not end-to-end differentiable. One needs to use REINFORCE to optimize the model for producing the function program.

An extension of NMN is Prob-NMN where the predicted program is supervised over a small set of samples. These ground truth programs provide a prior distribution over valid programs. This prior is used to determine future valid programs and enforce systematic program layouts.

In Stack Neural Module Network, reasoning instructions in the question are executed as a neural program over neural modules which act on pixel level attention to answer questions. This neural program execution approach produces a soft function program, where discrete reasoning structures, such as differentiable pointer and stack, are incorporated into the neural program executor. This enables Stack-NMN to maintain uncertainty over which reasoning steps are selected.

GNN Methods. In Neural State Machine (NSM) and Language-Conditioned Graph Neural Networks (LCGN), images are represented as object and relations, and graph neural networks conditioned on the language feature are used as reasoning modules. Graph neural networks are structured networks, which can represent logical classifier in graphs. Such graph neural networks and deep perception architectures are end-to-end differentiable. However, the architecture is quite generic, and requires large number of question-answer pairs to train.

NS-CL. In Neural Symbolic Concept Learner, questions are turned into a symbolic program with soft logic function, which are then executed end-to-end on object attention produced by the vision model. The soft logic makes the reasoning step end-to-end differentiable. However, the functional programs are still discrete in structure, making the entire model not end-to-end differentiable. One needs to use REINFORCE to optimize the model for producing the function program.

NGS. Neural-Grammar-Symbolic performs abduction, where both the image and language are turn into discrete object by sampling from the perception model, and symbolic program is exected to generate the answers.

In abductive learning, the symbolic reasoning step is directly interpretable, and many existing logic reasoning engine can be used. However, the model is not end-to-end differentiable. Discrete optimization is needed to generate supervision for the vision and language model to be updated.

B DIFFERENTIABLE END-TO-END PROGRAM EXECUTOR

The overall DePe architecture, as shown in Figure 4, consists of multiple sub-parts. The perception models that encode the vision and the question. The soft logic functions $\mathcal{F}$ closely follow the domain-specific language (DSL) provided by CLEVR. We implement them in a probabilistic manner based on the signature, detailed in Appendix E. The function execution results are stored in a differentiable memory that maintains a stack of these operations. As our model executes step by step, this memory is used as a buffer for the reasoning chain's probabilistic steps leading to a final answer.

B.1 PERCEPTION

Object detection. For our vision, we use object-level detection through a Mask R-CNN (He et al., 2017) to extract the corresponding bounding boxes of the objects. We then take detected objects and

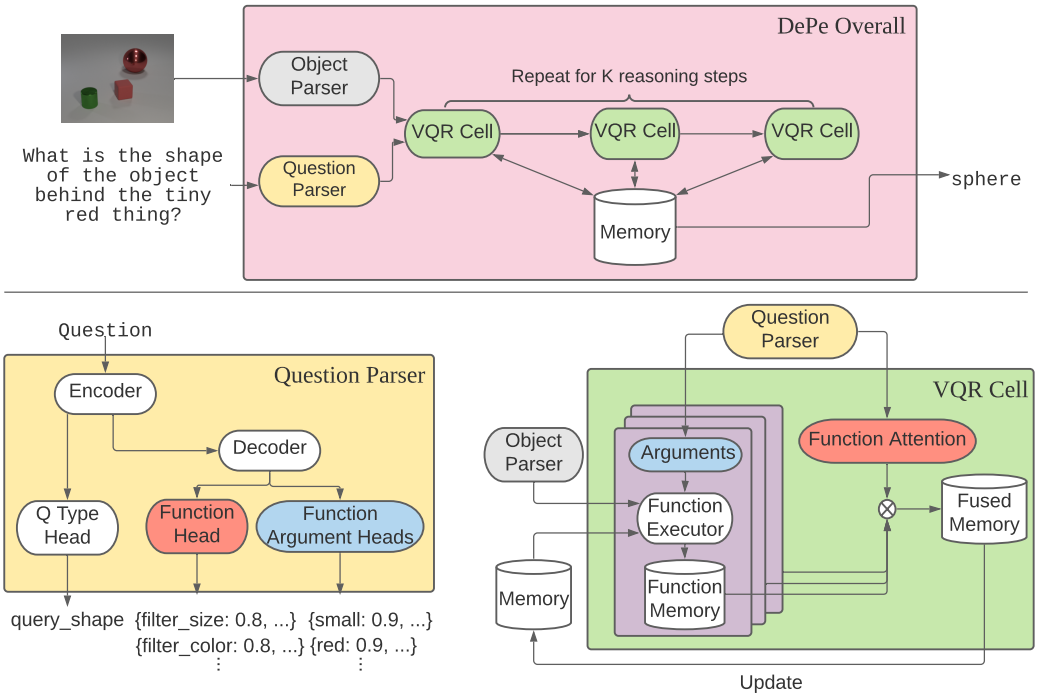

Figure 4: Our model ingests the image and extracts object-centric information. The question (textual input) is used by the question parser, which first embeds the question through an LSTM encoder. This embedding is used to predict the question type, which will be used to retrieve the answer from memory at the final step. The encoded text is passed to a decoder, which at every step predicts attention over which functions to execute and each function's arguments at the current step. The objects, arguments, and any preceding memory inputs are used by each function to execute, stored in memory belonging to each function. These function memories are weighted by functional attention and fused, which updates the original memory. The decoding and the VQR Cell execution run a fixed number of times, and then the answer is extracted from the final memory.

pass them through a pre-trained ResNet-50 model (He et al., 2016) to extract image features. These features are fed into models to predict object attributes from the image features and their bounding boxes' object relations.

Question parser. The parser takes in the text input and encodes it through a Bi-LSTM to generate the final hidden state. This hidden state is used in the program parser processes where a function model predicts the attention over which soft function is to be used at that step. Additionally, these functions may contain arguments, such as `filter_shape[cube]`, so for each class of functions, we have a model to predict a distribution over the possible attributes. The hidden state is also used to predict question type, which is used to select the final memory answer at the end of the execution.

For the vision and the language, all prediction models are small MLPs. We denote the set of trainable vision and text models parameters as θ_{vision}, and θ_{text} respectively.

B.2 Memory

In the CLEVR DSL, the functions have different types of inputs and outputs. Stack-NMN used neural modules to approximate these DSL functions, thus were able to make the image attention output of a consistent length. Since we are using the soft version of the DSL, we had to design a memory structure that could handle varying input and output modalities and sizes.

Memory block. To create a heterogeneous memory structure, we pre-assigned different modalities to different parts of a matrix. This memory matrix block $M \in \mathbb{R}^{T\times(A+C+1)}$ is used to store the intermediate results of the function computation.

Here the first dimension T is the stack dimension on which the function outputs are stored in different steps of the execution. The second $(A + C + 1)$ dimension are the rows that store the heterogeneous values. For the rows, A elements in the row indicate the object set attentions m_{det}, or which objects the model is paying attention to at that step. The C stores the categorical value outputs possible by our vision models such as m_{color} or m_{size}. For example if there are 6 possible colors, then $m_{color} \in \mathbb{R}^6$. The final dimension m_{num} is reserved for some accumulated score such as `count` or a boolean bit for true or false questions. For the CLEVR specific case the object attentions $m_{det} \in \mathbb{R}^A$. The categorical values are $[m_{color}; m_{shape}; m_{texture}; m_{size}] \in \mathbb{R}^C$, and the numeric slot $m_{det} \in \mathbb{R}$.

Stack structure. Choosing which row $t \in T$ of the memory is accessed is handled by the stack. This enables the model to accurately retrieve the previous one or two stack values as needed by the functions, instead of to predict which locations in memory to use.

Some functions may require just looking at the previous output from the VQR Cell, such as chain-like questions over a scene graph through functions such as `filter, relate, sum`. Such functions will pop from and then push to the same memory row. There are situations where functions need multiple inputs as well, such as comparison functions `attribute_equal, count_equal`. These functions will thus pop two values from the stack and will only push back one. These function signatures are summarized in Table 7 in Appendix E.

Stack manipulation. Each function will have access to the stack, which abstracts the memory management appropriately through push and pop operations. The stack memory M is initialized at random and a one-hot memory pointer $p \in \mathbb{R}^T$, starting at $p_0 = 1$. Each function returns a row m or a specific slice such as m_{color} to be updated in memory.

To push a row m onto the stack we follow the Stack-NMN convention updating the pointer as $p = \text{1d\_conv}(p, [0, 0, 1])$ and the memory row $M_i = M_i \cdot (1 - p_i) + m \cdot p_i$. Here the convolution is just moving the one hot pointer left by one in a differentiable manner. Then only the memory row that contains the one hot pointer is updated with the row m. Similarly to pop a value we perform the following retrieve the row $m = \sum_{t=1}^{T} p_i \cdot M_i$ and push back the pointer right by $p = \text{1d\_conv}(p, [1, 0, 0])$.

B.3 VQR CELL

The visual question reasoning (VQR) Cell is responsible for the step by step executions over the vision and the text to get to the answer. Compared to previous methods that executed these DSL functional programs in a discrete sequence, we generate probabilistic output over all the functions $\mathcal{F}$ at each step. With this approach, we can propagate the uncertainties of the object detections *and* the question parser for end-to-end learning. An example of this execution is visible in Figure 5.

Soft function execution. Each function has different return values and signatures. Since functions can have different input and output requirements, they require to be at different positions of the stack. Therefore each function is given its copy of the memory stack to operate over.

Once the memory outputs for each function are computed, they have to be weighted by which function our model is most likely needed for this step. This is done by using the function attention weights w computed by the question parser. These weights scale each function's memory and the pointer and then computes a weighted sum, which is then used to update the global memory M and pointer p, as shown in Algorithm 1. This global memory is then copied for all the function memories in the next Cell iteration. The cell executes for a fixed number of iterations and is set to a number such that it can cover most ground truth program lengths.

Algorithm 1: VQR Cell iteration

for $j \in [1, |\mathcal{F}|]$ **do**
  $M_j, p_j = \mathcal{F}_j(M, p)$
end
$M_{ave} = \sum_{j=1}^{|\mathcal{F}|} w_j \cdot M_j$
$p = \text{softmax}(\sum_{j=1}^{|\mathcal{F}|} w_j \cdot p_j)$
$M = M \cdot (1 - p) + M_{ave} \cdot p$

Final answer. After the last iteration of VQR Cell execution, the final answer is chosen by the memory corresponding to most probable question type. For example if we predict the question is asking about the color, then we return argmax(m_{color}). If we predict the question type is about count, then we return m_{num}. Since our memory structure is heterogeneous, taking argmax over the

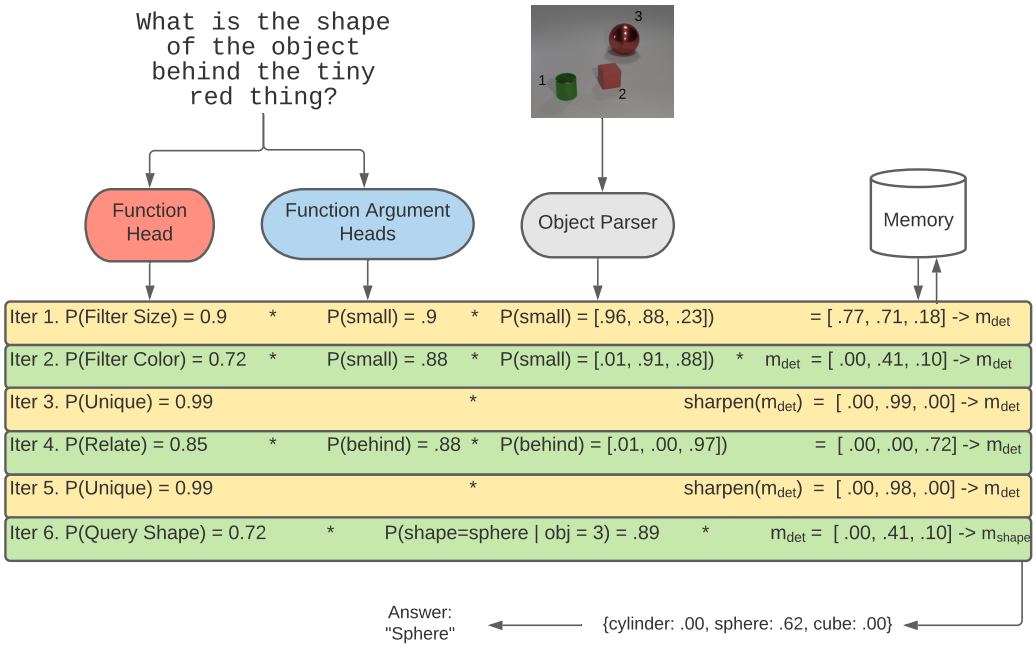

Figure 5: Here is a sample execution for the vision and text in DePe. For the simplicity of the diagram, we don't include all the probability traces and just show the ones corresponding to the max probability. For the vision, the predictions correspond to the objects detected in the scene (marked with 1, 2, 3 in the picture). For each execution, we store the multiplication in the corresponding memory row, and retrieve the final memory to answer the question.

entire row m may lead to an incorrect label due to parts of the memory storing values at different ranges, such as probabilities $\in [0, 1]$ or counts $\in \mathbb{Z}^+$.

B.4 Optimization

The objective is to minimize the overall loss based on the predictions retrieved from memory. This involves a multi-task loss as we are predicting categorical and numeric answers to questions to optimize over $\theta_{text}, \theta_{vision}$. This is done by minimizing the cross entropy loss for categorical answers and a mean squared error loss for numeric ones: $\mathcal{L} = \mathcal{L}_{QA\_XEnt} + \mathcal{L}_{QA\_MSE}$.

This optimization, particularly for the text has posed to be a difficult problem from scratch as there are many candidate functions at each step in the function program. Furthermore spurious programs can be proposed in early stages of training, corresponding to the shortest programs that answer the question, but don't align with the ground truth semantics.

To address this, models such as NS-CL, employ structural constraints when training their text parser. They employ a fixed number of concepts tokens used to parse the question and discrete templates for the parser to follow. In addition they leverage curriculum learning and switch between optimizing the parser and vision models to support both the discrete and continuous training methods.

Other models, such as NS-VQA, train on a small portion of the supervised labeled data. In this manner no such restrictions on the concepts, templates, or curriculum learning, but require labeling of ground truth programs. In this work we explore the pre-trained route. We pre-train our models with a small percentages (0.1-1%) of visual and textual data. When we use X% pre-training data, we sample X% of the training questions. Then we directly supervise our question parser and vision models on those QA programs and their corresponding vision scenes.

Additionally, we found it useful to add these direct supervision losses when training end-to-end with corresponding weights α and β. We formulate this as a regularization method that prevents the

models from diverging to spurious predictions when the training signal is only coarse QA labels.

$$\mathcal{L} = \mathcal{L}_{QA\_XEnt} + \mathcal{L}_{QA\_MSE} + \alpha\mathcal{L}_{Obj\_XEnt} + \beta\mathcal{L}_{Text\_XEnt}$$

The full definitions and learning strategies for the losses are available in Appendix C.

C Optimization Details

If given and Image I and a question Q our model makes a prediction as follows. We first make predictions of the objects and text components as:

$$\hat{y} = \text{DePe}(I, Q; \theta_{text}, \theta_{vision})$$

We look to minimize the following loss $\mathcal{L}$.

$$\mathcal{L}_{QA\_XEnt} = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{C}\mathbb{I}_{y_i \in cat} \cdot y_{ij} \log \hat{y}_j$$

$$\mathcal{L}_{QA\_MSE} = \frac{1}{N}\sum_{i}^{N}\mathbb{I}_{y_i \in num} \cdot ||y_i - \hat{y}_{num}||_2$$

$$\mathcal{L} = \mathcal{L}_{QA\_XEnt} + \mathcal{L}_{QA\_MSE}$$

We note that for stable convergence we include the pre-training data to our loss function. These are the cross entropy loss for the object attribute and relation predictions $\mathcal{L}_{Obj\_XEnt}$. These are also the cross entropy loss for the question parser predictions of the functions at each step $\mathcal{L}_{Text\_XEnt}$. This gives us our final loss function:

$$\mathcal{L} = \mathcal{L}_{QA\_XEnt} + \mathcal{L}_{QA\_MSE} + \alpha\mathcal{L}_{Obj\_XEnt} + \beta\mathcal{L}_{Text\_XEnt}$$

We include two scalar terms for the pre-training losses to balance the models behavior to find local minima within the QA losses and focusing too much on optimizing the smaller number of pre-training samples. Setting these hyperparameters is more prelevant when the ratio of pre-training to QA samples is low, since the pre-training samples provide a weak signal, but should be respected. We find that setting $\alpha = \beta = \sqrt{\%\text{ of QA data}}$, works well in practice to balance this ratio. The gains of setting these hyperparmeters can be seen in Table 4.

		0.1% pre-train			0.5% pre-train			1% pre-train		
% QA Data	$\alpha = \beta$	base	weight	Δ	base	weight	Δ	base	weight	Δ
5%	3	87.7	97.9	10.2	97.1	99.1	2.0	98.8	99.2	0.4
10%	6	92.3	98.0	5.7	96.6	99.2	2.6	99.1	99.3	0.2
100%	10	93.1	99.0	5.9	97.7	99.4	1.7	99.4	99.5	0.1

Table 4: Absolute validation accuracy gains by adding pre-training loss weights. The gains were computed against the baseline method of $\alpha = \beta = 1$.

We use Adam to optimize with $lr = 1e^{-4}, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e^{-8}$ and set our batch size to 1024.

D Experiments

Here we outline the details of the training procedure for the end-to-end training and efficiency experiments. We follow the default training procedure for the relevant work if it is not modified in this section.

D.1 LXMERT

For LXMERT we were interested in exploring if the cross modal architecture could implicitly encode our attention based reasoning process, as conveyed by the recent Transformer literature. There were a couple of approaches that we tried during our tests.

The initial tests and the results we reported were the based on the original architecture proposed in the paper as well as using the pre-trained weights the authors provided. This used 9, 5, and 5 layers of the language encoder N_L, the cross modality encoder, and the region of interest (RoI) object encoder N_R respectively.

We also test the initializing the RoI encoder from scratch since the CLEVR features are more simpler than the ones used in pre-training on real images. Additionally we tested LXMERT with fewer layers, $N_L = 4, N_X = 2, N_R = 2$, which we report in Table 5.

% QA Data	LXMERT Pre-trained	LXMERT Scratch	LXMERT 422
100	97.9	**98.0**	95.6
50	**95.3**	93.5	90.4
25	**89.2**	69.27	70.71
10	**66.5**	50.55	52.72

Table 5: LXMERT CLEVR performance on different subsets of data and configurations.

D.2 STACK-NMN

When experimenting with Stack-NMN we first trained the entire model on 100% of the CLEVR data using the ground truth layout and save it. For the visual representation experiments we take the trained model and freeze the LSTM weights while resetting the vision weights. For the language representation we freeze the CNN and NMN weights without the expert layout.

When training Stack-NMN for direct supervision, we train the model with the ground truth layout but only on 0.1% of the data. Then we take load this model and ran it on the corresponding 1, 5, and 10% QA data without the expert layout. With only 0.1%, we didn't see any sample of computational performance differences on end-to-end training.

We used the same settings as the original paper and trained all variations for 200k iterations.

D.3 NS-CL

When testing NS-CL end-to-end we require iterating between optimizing the language program parser and training the vision concept models, done in a REINFORCE and end-to-end fashion respectively. When conducting the computational complexity experiments we had to train and test the program parser and vision models at fixed number of training samples. To do this we divide up the total number of training samples uniformly into 10 curriculum lessons used in the original NS-CL. We then tune the program parser with the data belonging to the cumulative curriculum up to those number of training samples. Given this program parser we tune the vision models end-to-end over the same number of training samples. This way we have a better picture of NS-CL's behavior as a function of the training iterations.

To make NS-CL more comparative to DePe, we also use the 0.1% direct supervision data improve the NS-CL initial performance. We similarly train the program parser on the (Q, FP) pairs, thus start off at similar accuracies. Unlike DePe, NS-CL uses similarities between text and vision concepts, soit was unclear how to directly supervise the vision model given the object data. Similar to Stack-NMN we test a burn-in strategy where we train the entire model end-to-end with the QA Data from the corresponding 0.1% direct supervision then train using the 10% QA data for 150 epochs. Here we observed no significant gains from just start off with the supervised vision parser.

In the reasoning representation experiments we test NS-CL with discrete reasoning. Here we took the original soft logic $\mathcal{F}$ defined in the NS-CL paper and replaced them with symbolic logic as used in NS-VQA. Then we used REINFORCE to optimize when training on QA data in NS-RL. In NS-

AB we use abduction to find the most likely change in the object concept prediction to make the resulting reasoning provide the correct answer. We discuss this in the next section.

D.4 NS-AB ABDUCTION

Abduction involves determining which detection m_{det} changes are required to correctly answer the question given a fixed discrete function program.

At a high level these changes are done by making the most likely change that makes the output answer right. In NGS they use a tree based formulation and in our implementation we retain our stack implementation. Our stack implementation lets us test abduction proposals in a greedy efficient manner, but leads to a smaller search of the detection space leading to the ground truth answer instead of a spurious one. We describe the details of our implementation below.

D.4.1 POLICY DEFINITION

The function program $Q = \{f_i\}^T, f_i \in \mathcal{F}$ tells us to execute a sequence of T operations or functions f_i. If there are multiple reasoning branches that are aggregated, the execution trace follows a DAG dependency structure. We use a Markov decision process (MDP) formulation for the execution process:

- A policy $m^t_{det} \sim \pi_\theta(m^t_{det}|m^t_*, f_t, I)$ with vision models parameters θ_{vision} will take the operation execution from the previous stage m^t_*, the operator indicator f_t and image I, and output an action a_t. Memory from the previous stage m^t_* could be any memory slice such as $m_{det}, m_{attr}, m_{num}$. This action m^t_{det} corresponds to the object selections made for the current operation from the input image.
- The action will cause the state m^t_* to be updated to m^{t+1}_*, and the transition is described by $P(m^{t+1}_*|m^t_*, m^t_{det}, f_t)$. This is the operation execution for the current stage. This is carried by logic function f_t as described in Table 7.

After this sequence of operations, we obtain the a reward $R(A, m^{t+1}_*)$ by comparing the last selected state with the answer A, and tell us whether the answer is correct or not. If the answer is correct $R = 1$, and otheriwse 0. Then the expected correctness of answer given the uncertainty in π_θ and P is

$$\mathbb{E}[R(A, m^{t+1}_*)], \text{ where} \tag{1}$$

$$m^{t+1}_* \sim P(m^{t+1}_*|m^t_*, m^t_{det}, f_t),\ a_t \sim \pi_\theta(m^t_{det}|m^t_*, f_t, I),\ m^0_* = \emptyset,\ \forall t = 1, \ldots, T \tag{2}$$

D.4.2 OPTIMIZATION

Given m data points expressed as image I, questions Q and answer A triplets: $(I_1, Q_1, A_1), \ldots, (I_m, Q_m, A_m)$, the learning problem can be expressed as:

$$\pi^*_\theta = \arg\max_{\pi_\theta} \sum_{i=1}^{m} \mathbb{E}[R(A_i, m^{i,t+1}_*)] \tag{3}$$

Label abduction. If for a particular data point (I, Q, A) the answer according to the model is incorrect, one can try to find corrections by making minimal corrections c_i to the model detections as follows:

$$c^*_1, \ldots, c^*_T = \text{argmin}_{c_1,\ldots,c_T}\ D(m^1_{det}, \ldots, m^T_{det} \| c_1, \ldots, c_T) \tag{4}$$

$$\text{s.t. } R(A, m^{t+1}_*) = 1 \text{ makes the answer right.} \tag{5}$$

$$m^{t+1}_* \sim P(m^{t+1}_*|m^t_*, c_t, f_t), \forall t = 1, \ldots, T \tag{6}$$

where $D(\|)$ is some distance measure between the corrections c_i and the model predictions m^i_{det}. For instance, $D(\|)$ can be the negative likelihood of the correction under our model π_θ, or the edit distance.

Intuitively, we want to attempt the most likely corrections iteratively to find a solution by minimizing $D(\|)$. Sampling all possible corrections at once, can cause right answers through spurious label changes, which we want to mitigate.

Sampling methods. Due to the compositional nature of the reasoning tasks, we attempt to optimize c_i^* in a greedy fashion at each step of the program from $i = 1$ to $i = T$ instead of jointly from $i \subseteq \{1, \ldots, T\}$.

This better enforces the consistency constraint when a single c_i^* update leads to valid program executions. Valid executions mean that the predicted or abduced labels c_i^* lead to a final answer, whether right or wrong. This is opposed to making conflicting changes in all $c_i^*,\ c_j^*$ where $i \neq j$, leading to a failed program execution due to the manual abduced changes. If this greedy approach fails to find an answer, we fall back on exhaustively sampling $m_{det}^i \sim \pi_\theta$ at all program levels for a fixed number of iterations.

E MODULAR SOFT-LOGIC FUNCTIONS

The descriptions of the variables and constants used to describe memory components are listed in Table 6. The functions used are in Table 7. For notation simplicity, the function arguments are assumed to be popped from that function's memory copy or predicted from the text or vision models.

Name	Module	Description	Values
D	Hyperparameter	Number of objects	$\mathbb{Z}^+$
γ	Hyperparameter	Shift value	$\mathbb{R}$
τ	Hyperparameter	Scalar value	$\mathbb{R}$
m_{det}	Memory	Gates determining the active object detections	$\{[0, 1]\}^D$
m_{num}	Memory	Storage for numerical and boolean operations	$\mathbb{R}$
m_{attr}	Memory	Probability distribution over that attribute	$\mathbb{R}^{\lvert attr\rvert}$
p_{arg}	Text	Function argument probability distribution	$\mathbb{R}^{\lvert attr\rvert}, \mathbb{R}^{\lvert rel\rvert}$
P_{rel}	Vision	Object pairwise relation predictions	$\mathbb{R}^{D \times D \times \lvert rel\rvert}$
P_{attr}	Vision	Image attribute prediction per object	$\mathbb{R}^{D \times \lvert attr\rvert}$

Table 6: Description of functional arguments and constants.

Signature	Implementation
scene()	$m_{det} := 1$
unique(m_{det})	$m_{det} := (\frac{m_{det}}{1-m_{det}})/\text{sum}(\frac{m_{det}}{1-m_{det}})$
count(m_{det})	$m_{num} := \text{sum}(m_{det})$
exist(m_{det})	$m_{num} := \max(m_{det})$
intersect(m^1_{det}, m^2_{det})	$m_{det} := \min(m^1_{det}, m^2_{det})$
union(m^1_{det}, m^2_{det})	$m_{det} := \max(m^1_{det}, m^2_{det})$
equal_integer(m^1_{num}, m^2_{num})	$m_{num} := \text{sigmoid}((\tau - \lvert m^1_{num} - m^2_{num}\rvert)/(\gamma \cdot \tau))$
greater_than(m^1_{num}, m^2_{num})	$m_{num} := \text{sigmoid}((m^1_{num} - m^2_{num} - \gamma)/\tau)$
less_than(m^1_{num}, m^2_{num})	$m_{num} := \text{sigmoid}((m^2_{num} - m^1_{num} - \gamma)/\tau)$
relate(m_{det}, P_{rel}, p_{arg})	$m_{det} := (m_{det})^\top (P_{rel})(p_{arg})$
filter_attr(m_{det}, P_{attr}, p_{arg})	$m_{det} := \min(m_{det}, (P_{attr})(p_{arg}))$
query_attr(m_{det}, P_{attr}, p_{arg})	$m_{attr} := \min(p_{arg}, (P_{attr})^\top (m_{det}))$
same_attr(m_{det}, P_{attr})	$m_{det} := ((P_{attr})(P_{attr})^\top \odot (1 - I))(m_{det})$
equal_attr(m^1_{attr}, m^2_{attr})	$m_{num} := (m^1_{attr})^\top (m^2_{attr})$

Table 7: Implementation details for each modular function. $m_{det}, m_{num}, m_{attr}$ correspond to different parts in the memory representing attentional masks, numerical results, and attributes. p_{arg} is the distribution of functional arguments produced by the question parser, while P_{rel} and P_{attr} are relation and attribute predictions given by the perception module. Hyper-parameters $D = 50$, $\tau = 0.25$ and $\gamma = 0.5$ and attribute functions are split further by each $attr \in \{shape, color, material, size\}$.