Inferring and Executing Programs for Visual Reasoning

Justin Johnson[1] Bharath Hariharan[2] Laurens van der Maaten[2]
Judy Hoffman[1] Li Fei-Fei[1] C. Lawrence Zitnick[2] Ross Girshick[2]

[1]Stanford University [2]Facebook AI Research

Abstract

Existing methods for visual reasoning attempt to directly map inputs to outputs using black-box architectures without explicitly modeling the underlying reasoning processes. As a result, these black-box models often learn to exploit biases in the data rather than learning to perform visual reasoning. Inspired by module networks, this paper proposes a model for visual reasoning that consists of a program generator *that constructs an explicit representation of the reasoning process to be performed, and an* execution engine *that executes the resulting program to produce an answer. Both the program generator and the execution engine are implemented by neural networks, and are trained using a combination of backpropagation and REINFORCE. Using the CLEVR benchmark for visual reasoning, we show that our model significantly outperforms strong baselines and generalizes better in a variety of settings.*

1. Introduction

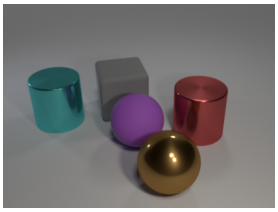
In many applications, computer-vision systems need to answer sophisticated queries by *reasoning* about the visual world (Figure 1). To deal with novel object interactions or object-attribute combinations, visual reasoning needs to be *compositional*: without ever having seen a "person touching a bike", the model should be able to understand the phrase by putting together its understanding of "person", "bike" and "touching". Such compositional reasoning is a hallmark of human intelligence, and allows people to solve a plethora of problems using a limited set of basic skills [28].

In contrast, modern approaches to visual recognition learn a mapping directly from inputs to outputs; they do not explicitly formulate and execute compositional plans. Direct input-output mapping works well for classifying images [26] and detecting objects [10] for a small, fixed set of categories. However, it fails to outperform strong baselines on tasks that require the model to understand an exponentially large space of objects, attributes, actions, and interactions, such as visual question answering (VQA) [3, 51]. Instead, models that learn direct input-output mappings tend to learn dataset biases but not reasoning [7, 18, 19].

How many chairs are at the table?

Is there a pedestrian in my lane?

Is the person with the blue hat touching the bike in the back?



Is there a matte cube that has the same size as the red metal object?

Figure 1. Compositional reasoning is a critical component needed for understanding the complex visual scenes encountered in applications such as robotic navigation, autonomous driving, and surveillance. Current models fail to do such reasoning [19].

In this paper, we argue that to successfully perform complex reasoning tasks, it might be necessary to explicitly incorporate compositional reasoning in the model structure. Specifically, we investigate a new model for visual question answering that consists of two parts: a *program generator* and an *execution engine*. The *program generator* reads the question and produces a plan or *program* for answering the question by composing functions from a function dictionary. The *execution engine* implements each function using a small neural module, and executes the resulting module network on the image to produce an answer. Both the program generator and the modules in the execution engine are neural networks with generic architectures; they can be trained separately when ground-truth programs are available, or jointly in an end-to-end fashion.

Our model builds on prior work on neural module networks that incorporate compositional reasoning [1, 2]. Prior module networks do not generalize well to new problems,

because they rely on a hand-tuned program generator based on syntactic parsing, and on hand-engineered modules. By contrast, our model does not rely on such heuristics: we only define the function vocabulary and the "universal" module architecture by hand, learning everything else.

We evaluate our model on the recently released CLEVR dataset [19], which has proven to be challenging for state-of-the-art VQA models. The CLEVR dataset contains ground-truth programs that describe the compositional reasoning required to answer the given questions. We find that with only a small amount of reasoning supervision (9000 ground truth programs which is 2% of those available), our model outperforms state-of-the-art non-compositional VQA models by ~20 percentage points on CLEVR. We also show that our model's compositional nature allows it to generalize to novel questions by composing modules in ways that are not seen during training.

Though our model works well on the algorithmically generated questions in CLEVR, the true test is whether it can answer questions asked by humans in the wild. We collect a new dataset of human-posed free-form natural language questions about CLEVR images. Many of these questions have out-of-vocabulary words and require reasoning skills that are absent from our model's repertoire. Nevertheless, when finetuned on this dataset without additional program supervision, our model learns to compose its modules in novel but intuitive ways to best answer new types of questions. The result is an interpretable mapping of free-form natural language to programs, and a ~9 point improvement in accuracy over the best competing models.

2. Related Work

Our work is related to to prior research on visual question answering, reasoning-augmented models, semantic parsers, and (neural) program-induction methods.

Visual question answering (VQA) is a popular proxy task for gauging the quality of visual reasoning systems [21, 44]. Like the CLEVR dataset, benchmark datasets for VQA typically comprise a set of questions on images with associated answers [3, 32, 40, 25, 51]; both questions and answers are generally posed in natural language. Many systems for VQA employ a very similar architecture [3, 8, 9, 31, 33, 34, 45]: they combine an RNN-based embedding of the question with a convolutional network-based embedding of an image in a classification model over possible answers. Recent work has questioned whether such systems are capable of developing visual reasoning capabilities: (1) very simple baseline models were found to perform competitively on VQA benchmarks by exploiting biases in the data [18, 50, 11] and (2) experiments on CLEVR, which was designed to control such biases, revealed that current systems do not learn to reason about spatial relationships or to learn disentangled representations [19].

Our model aims to address these problems by explicitly constructing an intermediate program that defines the reasoning process required to answer the question. We show that our model succeeds on several kinds of reasoning where other VQA models fail.

Reasoning-augmented models add components to neural network models to facilitate the development of reasoning processes in such models. For example, models such as neural Turing machines [12, 13], memory networks [41, 38], and stack-augmented recurrent networks [20] add explicit memory components to neural networks to facilitate learning of reasoning processes that involve long-term memory. While long-term memory is likely to be a crucial component of intelligence, it is not a prerequisite for reasoning, especially the kind of reasoning that is required for answering questions about images.[1] Therefore, we do not consider memory-augmented models in this study.

Module networks are an example of reasoning-augmented models that use a syntactic parse of a question to determine the architecture of the network [1, 2, 16]. The final network is composed of trained neural modules that execute the "program" produced by the parser. The main difference between our models and existing module networks is that we replace hand-designed off-the-shelf syntactic parsers [24], which perform very poorly on complex questions such as those in CLEVR [19], by a learnt program generator that can adapt to the task at hand.

Semantic parsers attempt to map natural language sentences to logical forms. Often, the goal is to answer natural language questions using a knowledge base [30]. Recent approaches to semantic parsing involve a learnt *programmer* [29]. However, the semantics of the program and the execution engine are fixed and known a priori, while we learn both the program generator and the execution engine.

Program-induction methods learn programs from input-output pairs by fitting the parameters of a neural network to predict the output that corresponds to a particular input value. Such models can take the form of a feedforward scoring function over operators in a domain-specific language that can be used to guide program search [4], or of a recurrent network that decodes a vectorial program representation into the actual program [22, 27, 35, 47, 48, 49]. The recurrent networks may incorporate compositional structure that allows them to learn new programs by combining previously learned sub-programs [36].

Our approach differs from prior work on program induction in (1) the type of input-output pairs that are used and (2) the way the domain-specific language is implemented. Prior work on neural program interpreters considers simple algorithms such as sorting of a list of integers; by contrast, we consider inputs that comprise an image and an associ-

[1] Memory is likely indispensable in more complex settings such as visual dialogues or SHRDLU [6, 43].

ated question (in natural language). Program induction approaches also assume knowledge of the low-level operators such as arithmetic operations. In contrast, we use a learnt execution engine and assume minimal prior knowledge.

3. Method

We develop a learnable compositional model for visual question answering. Our model takes as input an image x and a visual question q about the image. The model selects an answer $a \in \mathcal{A}$ to the question from a fixed set $\mathcal{A}$ of possible answers. Internally, the model predicts a program z representing the reasoning steps required to answer the question. The model then executes the predicted program on the image, producing a distribution over answers.

To this end, we organize our system into two components: a *program generator*, $z = \pi(q)$, which predicts programs from questions, and an *execution engine*, $a = \phi(x, z)$, which executes a program z on an image x to predict an answer a. Both the program generator and the execution engine are neural networks that are learned from data. In contrast to prior work [1, 2], we do not manually design heuristics for generating or executing the programs.

We present learning procedures both for settings where (some) ground-truth programs are available during training, and for settings without ground-truth programs. In practice, our models need *some* program supervision during training, but we find that the program generator requires very few of such programs in order to learn to generalize (see Figure 4).

3.1. Programs

Like all programming languages, our programs are defined by *syntax* giving rules for building valid programs, and *semantics* defining the behavior of valid programs. We focus on learning semantics for a fixed syntax. Concretely, we fix the syntax by pre-specifying a set $\mathcal{F}$ of functions f, each of which has a fixed arity $n_f \in \{1, 2\}$. Because we are interested in visual question answering, we include in the vocabulary a special constant `Scene`, which represents the visual features of the image. We represent valid programs z as *syntax trees* in which each node contains a function $f \in \mathcal{F}$, and in which each node has as many children as the arity of the function f.

3.2. Program generator

The program generator $z = \pi(q)$ predicts programs z from natural-language questions q that are represented as a sequence of words. We use a prefix traversal to serialize the syntax tree, which is a non-sequential discrete structure, into a sequence of functions. This allows us to implement the program generator using a standard LSTM sequence-to-sequence model; see [39] for details.

When decoding at test time, we simply take the argmax function at each time step. The resulting sequence of functions is converted to a syntax tree; this is straightforward since the arity of each function is known. Some generated sequences do not correspond to prefix traversals of a tree. If the sequence is too short (some functions do not have enough children) then we pad the sequence with `Scene` constants. If the sequence is too long (some functions have no parents) then unused functions are discarded.

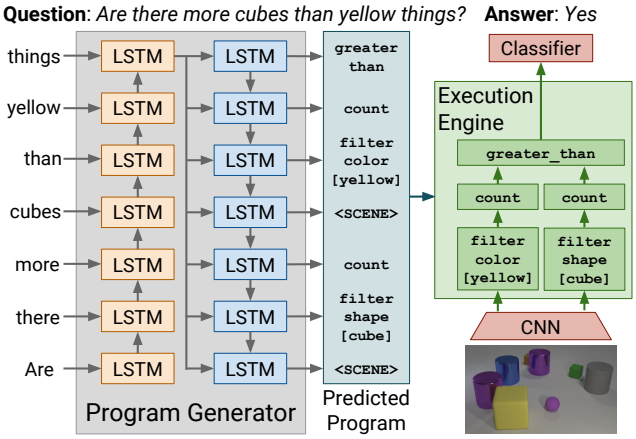


Figure 2. System overview. The **program generator** is a sequence-to-sequence model which inputs the question as a sequence of words and outputs a program as a sequence of functions, where the sequence is interpreted as a prefix traversal of the program's abstract syntax tree. The **execution engine** executes the program on the image by assembling a neural module network [2] mirroring the structure of the predicted program.

3.3. Execution engine

Given a predicted program z and and an input image x, the execution engine executes the program on the image, $a = \phi(x, z)$, to predict an answer a. The execution engine is implemented using a neural module network [2]: the program z is used to assemble a question-specific neural network that is composed from a set of modules. For each function $f \in \mathcal{F}$, the execution engine maintains a neural network module m_f. Given a program z, the execution engine creates a neural network $m(z)$ by mapping each function f to its corresponding module m_f in the order defined by the program: the outputs of the "child modules" are used as input into their corresponding "parent module".

Our modules use a generic architecture, in contrast to [2]. A module of arity n receives n features maps of shape $C \times H \times W$ and produces a feature map of shape $C \times H \times W$. Each unary module is a standard residual block [14] with two 3×3 convolutional layers. Binary modules concatenate their inputs along the channel dimension, project from $2C$ to C channels using a 1×1 convolution, and feed the result to a residual block. The `Scene` module takes visual features as input (*conv4* features from ResNet-101 [14] pretrained on ImageNet [37]) and passes these features through four

Method	Exist	Count	Compare Integer			Query				Compare				Overall
			Equal	Less	More	Size	Color	Mat.	Shape	Size	Color	Mat.	Shape	
Q-type mode	50.2	34.6	51.4	51.6	50.5	50.1	13.4	50.8	33.5	50.3	52.5	50.2	51.8	42.1
LSTM	61.8	42.5	63.0	73.2	71.7	49.9	12.2	50.8	33.2	50.5	52.5	49.7	51.8	47.0
CNN+LSTM	68.2	47.8	60.8	74.3	72.5	62.5	22.4	59.9	50.9	56.5	53.0	53.8	55.5	54.3
CNN+LSTM+SA [46]	68.4	57.5	56.8	74.9	68.2	90.1	83.3	89.8	87.6	52.1	55.5	49.7	50.9	69.8
CNN+LSTM+SA+MLP	77.9	59.7	60.3	83.7	76.7	85.4	73.1	84.5	80.7	72.3	71.2	70.1	69.7	73.2
Human† [19]	96.6	86.7	79.0	87.0	91.0	97.0	95.0	94.0	94.0	94.0	98.0	96.0	96.0	92.6
Ours-strong (700K prog.)	**97.1**	**92.7**	**98.0**	**99.0**	**98.9**	**98.8**	**98.4**	**98.1**	**97.3**	**99.8**	**98.5**	**98.9**	**98.4**	**96.9**
Ours-semi (18K prog.)	95.3	90.1	93.9	97.1	97.6	98.1	97.1	97.7	96.6	99.0	97.6	98.0	97.3	95.4
Ours-semi (9K prog.)	89.7	79.7	85.2	76.1	77.9	94.8	93.3	93.1	89.2	97.8	94.5	96.6	95.1	88.6

Table 1. Question answering accuracy (higher is better) on the CLEVR dataset for baseline models, humans, and three variants of our model. The strongly supervised variant of our model uses all 700K ground-truth programs for training, whereas the semi-supervised variants use 9K and 18K ground-truth programs, respectively. †Human performance is measured on a 5.5K subset of CLEVR questions.

convolutional layers to output a $C \times H \times W$ feature map.

Using the same architecture for all modules ensures that every valid program z corresponds to a valid neural network which inputs the visual features of the image and outputs a feature map of shape $C \times H \times W$. This final feature map is flattened and passed into a multilayer perceptron classifier that outputs a distribution over possible answers.

3.4. Training

Given a VQA dataset containing (x, q, z, a) tuples with ground truth programs z, we can train both the program generator and execution engine in a supervised manner. Specifically, we can (1) use pairs (q, z) of questions and corresponding programs to train the program generator, which amounts to training a standard sequence-to-sequence model; and (2) use triplets (x, z, a) of the image, program, and answer to train the execution engine, using backpropagation to compute the required gradients (as in [2]).

Annotating ground-truth programs for free-form natural language questions is expensive, so in practice we may have few or no ground-truth programs. To address this problem, we opt to train the program generator and execution engine jointly on (x, q, a) triples *without ground-truth programs*. However, we cannot backpropagate through the argmax operations in the program generator. Instead we replace the argmaxes with sampling and use REINFORCE [42] to estimate gradients on the outputs of the program generator; the reward for each of its outputs is the negative zero-one loss of the execution engine, with a moving-average baseline.

In practice, joint training using REINFORCE is difficult: the program generator needs to produce the right program without understanding what the functions mean, and the execution engine has to produce the right answer from programs that may not accurately implement the question asked. We propose a more practical *semi-supervised learning* approach. We first use a *small* set of ground-truth programs to train the program generator, then fix the program generator and train the execution engine using predicted programs on a large dataset of (x, q, a) triples. Finally, we use REINFORCE to jointly finetune the program generator and execution engine. Crucially, ground-truth programs are *only* used to train the initial program generator.

4. Experiments

We evaluate our model on the recent CLEVR dataset [19]. Standard VQA methods perform poorly on this dataset, showing that it is a challenging benchmark. All questions are equipped with ground-truth programs, allowing for experiments with varying amounts of supervision.

We first perform experiments using strong supervision in the form of ground-truth programs. We show that in this strongly supervised setting, the combination of program generator and execution engine works much better on CLEVR than alternative methods. Next, we show that this strong performance is maintained when a small number of ground-truth programs, which capture only a fraction of question diversity, is used for training. Finally, we evaluate the ability of our models to perform compositional generalization, as well as generalization to free-form questions posed by humans. Code reproducing the results of our experiments is available from https://github.com/facebookresearch/clevr-iep.

4.1. Baselines

Johnson *et al.* [19] tested several VQA models on CLEVR. We reproduce these models as baselines here.

Q-type mode: This baseline predicts the most frequent answer for each of the question types in CLEVR.

LSTM: Similar to [3, 33], questions are processed with learned word embeddings followed by a word-level LSTM [15]. The final LSTM hidden state is passed to a multi-layer perceptron (MLP) that predicts a distribution over answers. This method uses no image information, so it

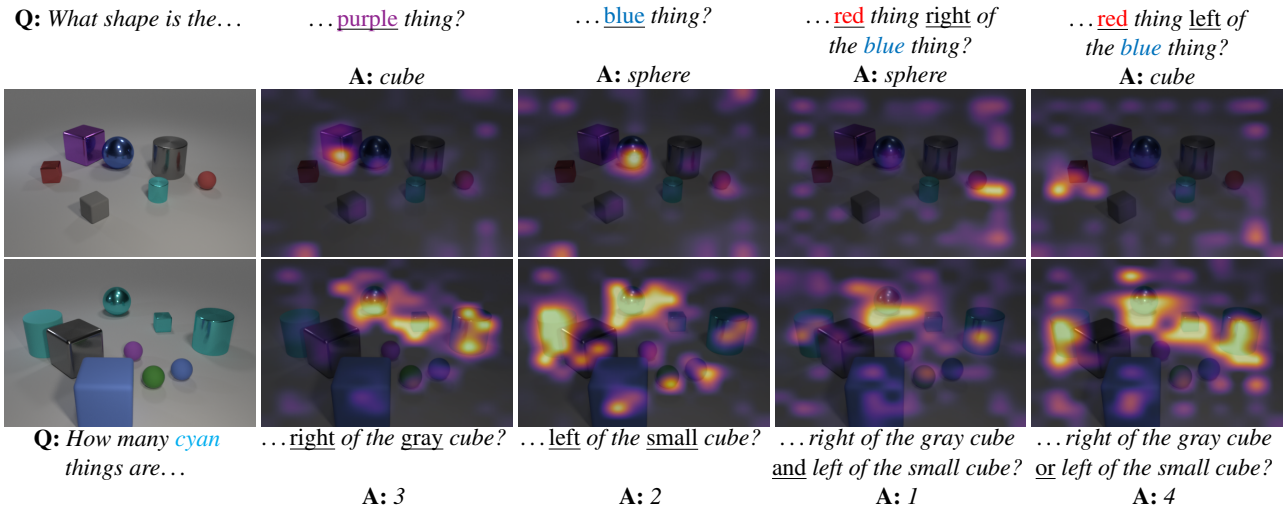


Figure 3. Visualizations of the norm of the gradient of the sum of the predicted answer scores with respect to the final feature map. From left to right, each question adds a module to the program; the new module is *underlined* in the question. The visualizations illustrate which objects the model attends to when performing the reasoning steps for question answering. Images are from the validation set.

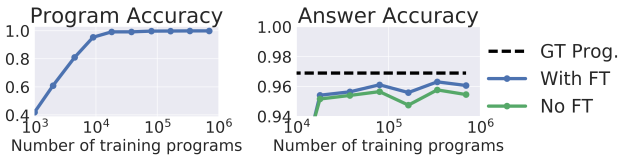


Figure 4. Accuracy of predicted programs (left) and answers (right) as we vary the number of ground-truth programs. Blue and green give accuracy before and after joint finetuning; the dashed line shows accuracy of our strongly-supervised model.

Method	Train A		Finetune B	
	A	B	A	B
LSTM	55.2	50.9	51.5	54.9
CNN+LSTM	63.7	57.0	58.3	61.1
CNN+LSTM+SA+MLP	80.3	68.7	75.7	75.8
Ours (18K prog.)	**96.6**	**73.7**	**76.1**	**92.7**

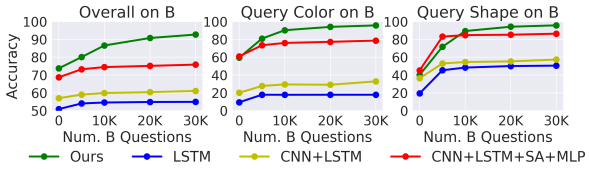


Figure 5. Question answering accuracy on the CLEVR-CoGenT dataset (higher is better). **Top:** We train models on Condition A, then test them on both Condition A and Condition B. We then finetune these models on Condition B using 3K images and 30K questions, and again test on both Conditions. Our model uses 18K programs during training on Condition A, and does not use any programs during finetuning on Condition B. **Bottom:** We investigate the effects of using different amounts of data when finetuning on Condition B. We show overall accuracy as well as accuracy on color-query and shape-query questions.

can only model question-conditional biases.

CNN+LSTM: Images and questions are encoded using convolutional network (CNN) features and final LSTM hidden states, respectively. These features are concatenated and passed to a MLP that predicts an answer distribution.

CNN+LSTM+SA [46]: Questions and images are encoded using a CNN and LSTM as above, then combined using two rounds of soft spatial attention; a linear transform of the attention output predicts the answer.

CNN+LSTM+SA+MLP: Replaces the linear transform with an MLP for better comparison with the other methods.

The models that are most similar to ours are neural module networks [1, 2]. Unfortunately, neural module networks use a hand-engineered, off-the-shelf parser to produce programs, and this parser fails[2] on the complex questions in CLEVR [19]. Therefore, we were unable to include module networks in our experiments.

4.2. Strongly and semi-supervised learning

We first experiment with a model trained using full supervision: we use the ground-truth programs for all questions in CLEVR to train both the program generator and the execution engine separately. The question answering accuracy of the resulting model on CLEVR is shown in Table 1 (Ours-strong). The results show that using strong supervision, our model can achieve near-perfect accuracy on CLEVR (even outperforming Mechanical Turk workers).

In practical scenarios, ground-truth programs are not available for all questions. We use the semi-supervised training process described in Section 3.4 to determine how many ground-truth programs are needed to match fully su-

[2]See supplemental material for example parses of CLEVR questions.

pervised models. First, the program generator is trained in a supervised manner using a small number of questions and ground-truth programs; next, the execution engine is trained on *all* CLEVR questions, using predicted rather than ground-truth programs. Finally, both components are jointly finetuned *without* ground-truth programs. Table 1 shows the accuracy of semi-supervised models trained with 9K and 18K ground-truth programs (Ours-semi).

The results show that 18K ground-truth programs are sufficient to train a model that performs almost on par with a fully supervised model (that used all 700K programs for training). This strong performance is not due to the program generator simply remembering all programs: the total number of unique programs in CLEVR is approximately 450K. This implies that after observing only a small fraction ($\leq 4\%$) of all possible programs, the model is able to understand the underlying structure of CLEVR questions and use that understanding to generalize to new questions.

Figure 4 analyzes how the accuracy of the predicted programs and the final answer vary with the number of ground-truth programs used. We measure the accuracy of the program generator by deserializing the function sequence produced by the program generator, and marking it as correct if it matches the ground-truth program exactly.[3] Our results show that with about 20K ground-truth programs, the program generator achieves near perfect accuracy, and the final answer accuracy is almost as good as strongly-supervised training. Training the execution engine using the predicted programs from the program generator instead of ground-truth programs leads to a loss of about 3 points in accuracy, but some of that loss is mitigated after joint finetuning.

4.3. What do the modules learn?

To obtain additional insight into what the modules in the execution engine have learned, we visualized the parts of the image that are being used to answer different questions; see Figure 3. Specifically, the figure displays the norm of the gradient of the sum of the predicted answer scores (softmax inputs) with respect to the final feature map. This visualization reveals several important aspects of our model.

First, it clearly attends to the correct objects even for complicated referring expressions involving spatial relationships, intersection and union of constraints, *etc.*

Second, the examples show that changing a *single* module (swapping *purple*/*blue*, *left*/*right*, *and*/*or*) results in drastic changes in both the predicted answer and model attention, demonstrating that the individual modules do in fact perform their intended functions. Modules learn specialized functions such as localization and set operations without explicit supervision of their outputs.

[3]Note that this may underestimate the true accuracy, since two different programs can be functionally equivalent.

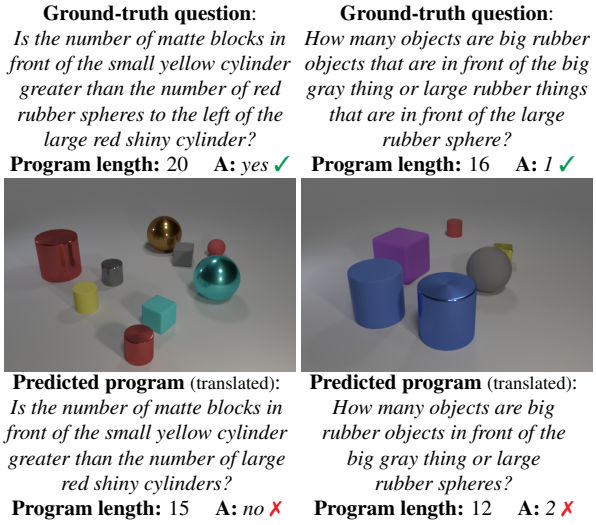


Figure 6. Examples of long questions where the program and answer were predicted incorrectly when the model was trained on short questions, but both program and answer were correctly predicted after the model was finetuned on long questions. Above each image we show the ground-truth question and its program length; below, we show a manual English translation of the predicted program and answer before finetuning on long questions.

	Train Short		Finetune Both	
Method	Short	Long	Short	Long
LSTM	46.4	48.6	46.5	49.9
CNN+LSTM	54.0	52.8	54.3	54.2
CNN+LSTM+SA+MLP	74.2	**64.3**	74.2	67.8
Ours (25K prog.)	**95.9**	55.3	**95.6**	**77.8**

Table 2. Question answering accuracy on short and long CLEVR questions. **Left columns**: Models trained only on short questions; our model uses 25K ground-truth short programs. **Right columns**: Models trained on both short and long questions. Our model is trained on short questions then finetuned on the entire dataset; no ground-truth programs are used during finetuning.

4.4. Generalizing to new attribute combinations

Johnson *et al.* [19] proposed the CLEVR-CoGenT dataset for investigating the ability of VQA models to perform compositional generalization. The dataset contains data in two different conditions: in Condition A, all cubes are gray, blue, brown, or yellow and all cylinders are red, green, purple, or cyan; in Condition B, cubes and cylinders swap color palettes. Johnson *et al.* [19] found that VQA models trained on data from Condition A performed poorly on data from Condition B, suggesting the models are not well capable of generalizing to new conditions.

We performed experiments with our model on CLEVR-CoGenT: in Figure 5, we report accuracy of the semi-supervised variant of our model trained on data from Condition A and evaluated on data from Condition B. Although the resulting model performs better than all baseline meth-

ods in Condition B, it still appears to suffer from the problems identified by [19]. A more detailed analysis of the results revealed that our model does not outperform the CNN+LSTM+SA baseline for questions about an object's shape or color. This is not surprising: if the model never sees red cubes, it has no incentive to learn that the attribute "red" refers to the color and not to the shape.

We also performed experiments in which we used a small amount of training data *without ground-truth programs* from condition B for finetuning. We varied the amount of data from condition B that is available for finetuning. As shown in Figure 5, our model learns the new attribute combinations from only ∼10K questions (∼1K images), and outperforms similarly trained baselines across the board.[4] We believe that this is because the model's compositional nature allows it to quickly learn new semantics of attributes such as "red" from little training data.

4.5. Generalizing to new question types

Our experiments in Section 4.2 showed that relatively few ground-truth programs are required to train our model effectively. Due to the large number of unique programs in CLEVR, it is impossible to capture all possible programs with a small set of ground-truth programs; however, due to the synthetic nature of CLEVR questions, it is possible that a small number of programs could cover all possible program structures. In real-world scenarios, models should be able to generalize to questions with novel program structures without observing associated ground-truth programs.

To test this, we divide CLEVR questions into two categories based on their ground-truth programs: *short* and *long*. CLEVR questions are divided into *question families*, where all questions in the same family share the same program structure. A question is *short* if its question family has a mean program length less than 16; otherwise it is *long*.[5]

We train the program generator and execution engine on short questions in a semi-supervised manner using 18K ground-truth short programs, and test the resulting model on both short and long questions. This experiment tests the ability of our model to generalize from short to long chains of reasoning. Results are shown in Table 2.

The results show that when evaluated on long questions, our model trained on short questions underperforms the CNN+LSTM+SA model trained on the same set. Presumably, this result is due to the program generator learning a bias towards short programs. Indeed, Figure 6 shows that the program generator produces programs that refer to the right objects but that are too short.

We can undo this short-program bias through joint finetuning of the program generator and execution engine on the combined set of short and long questions, *without ground-truth programs*. To pinpoint the problem of short-program bias in the program generator, we leave the execution engine fixed during finetuning; it is only used to compute REINFORCE rewards for the program generator. After finetuning, our model substantially outperforms baseline models that were trained on the entire dataset; see Table 2.

[4]Note that this finetuning hurts performance on condition *A*. Joint finetuning on both conditions will likely alleviate this issue.

[5]Partitioning at the family level rather than the question level allows for better separation of program structure between short and long questions.

Method	Train CLEVR	Train CLEVR, finetune human
LSTM	27.5	36.5
CNN+LSTM	37.7	43.2
CNN+LSTM+SA+MLP	50.4	57.6
Ours (18K prog.)	**54.0**	**66.6**

Table 3. Question answering accuracy on the CLEVR-Humans test set of four models after training on just the CLEVR dataset (**left**) and after finetuning on the CLEVR-Humans dataset (**right**).

4.6. Generalizing to human-posed questions

The fact that questions in the CLEVR benchmark were generated algorithmically may favor some approaches over others. In particular, natural language tends to be more ambiguous than algorithmically generated questions. We performed an experiment to assess the extent to which models trained on CLEVR can be finetuned to answer human questions. To this end, we collected a new dataset of natural-language questions and answers for CLEVR images.

The CLEVR-Humans Dataset. Inspired by VQA [3], workers on Amazon Mechanical Turk were asked to write questions about CLEVR images that would be *hard for a smart robot to answer*; workers were primed with questions from CLEVR and restricted to answers in CLEVR. We filtered questions by asking three workers to answer each question, and removed questions that a majority of workers could not correctly answer. We collected one question per image; after filtering, we obtained 17,817 training, 7,202 validation, and 7,145 test questions on CLEVR images. The data is available from the first author's website.

The human questions are more challenging than synthetic CLEVR questions because they exhibit more linguistic variety. Unlike existing VQA datasets, however, the CLEVR-Humans questions do not require common-sense knowledge: they focus entirely on visual reasoning abilities, which makes them a good testbed for evaluating reasoning.

Figure 7 shows some example human questions. Some questions are rewordings of synthetic CLEVR questions; others are answerable using the same basic functions as CLEVR but potentially with altered semantics for those skills. For example, people use spatial relationships "left", "right", *etc.* differently than their meanings in CLEVR questions. Finally, some questions require skills not needed for answering synthetic questions.

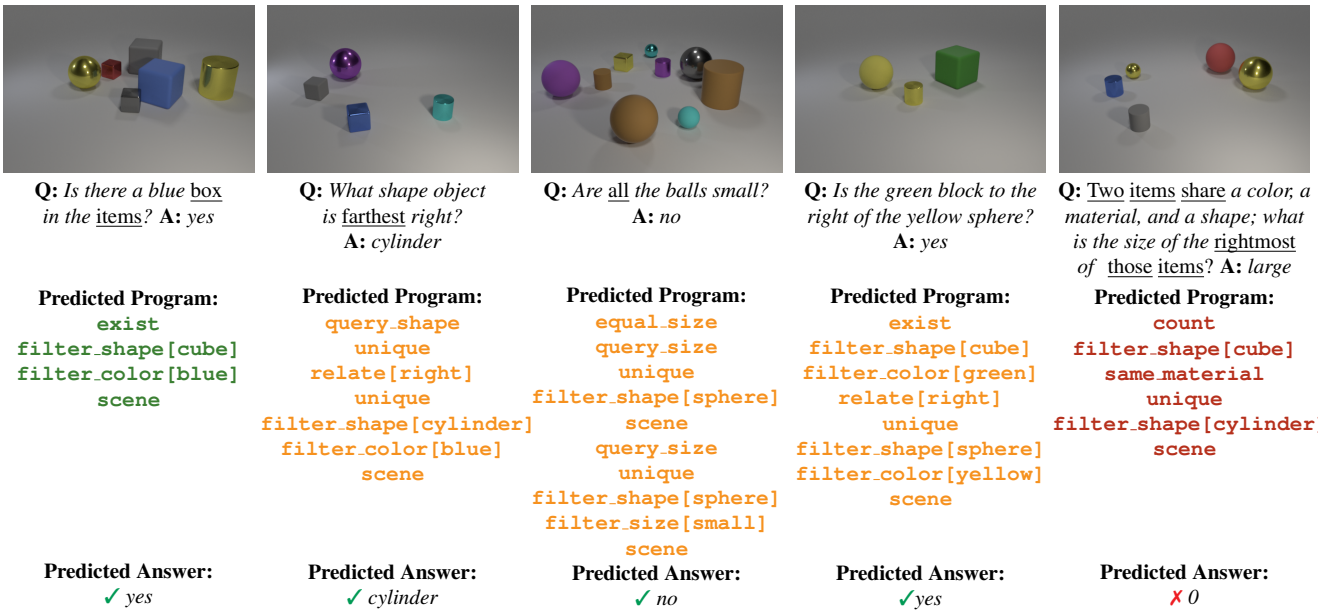


Figure 7. Examples of questions from the CLEVR-Humans dataset, along with predicted programs and answers from our model. Question words that do not appear in CLEVR questions are underlined. Some predicted programs exactly match the semantics of the question (green); some programs closely match the question semantics (yellow), and some programs appear unrelated to the question (red).

Results. We train our model on CLEVR, and then finetune *only the program generator* on the CLEVR-Humans training set to adapt it to the additional linguistic variety; we do not adapt the execution engine due to the limited quantity of data. No ground-truth programs are available during finetuning. The embeddings in the sequence-to-sequence model of question words that do not appear in CLEVR synthetic questions are initialized randomly before finetuning.

During finetuning, our model learns to reuse the reasoning skills it has already mastered in order to answer the linguistically more diverse natural-language questions. As shown in Figure 7, it learns to map novel words (*"box"*) to known modules. When human questions are not expressible using CLEVR functions, our model still learns to produce reasonable programs closely approximating the question's intent. Our model often fails on questions that cannot be reasonably approximated using our model's module inventory, such as the rightmost example in Figure 7. Quantitatively, the results in Table 3 show that our model outperforms all baselines on the CLEVR-Humans test set both with and without finetuning.

5. Discussion and Future Work

Our results show that our model is able to generalize to novel scenes and questions and can even infer programs for free-form human questions using its learned modules. Whilst these results are encouraging, there still are many questions that cannot be reasonably approximated using our fixed set of modules. For example, the question *"What color is the object with a unique shape?"* requires a model to identify unique shapes, for which no module is currently available. Adding new modules to our model is straightforward due to our generic module design, but automatically identifying and learning new modules without program supervision is still an open problem. One path forward is to design a Turing-complete set of modules; this would allow for all programs to be expressed without learning new modules. For example, by adding ternary operators (`if/then/else`) and loops (`for/do`), the question *"What color is the object with a unique shape?"* can be answered by *looping* over all shapes, counting the objects with that shape, and returning it *if* the count is one. These control-flow operators could be incorporated into our framework: for example, a loop could apply the same module to an input set and aggregate the results. We emphasize that learning such programs with limited supervision is an open research challenge, which we leave to future work.

6. Conclusion

This paper fits into a long line of work on incorporating symbolic representations into (neural) machine learning models [4, 5, 29, 36]. We have shown that explicit program representations can make it easier to compose programs to answer novel questions about images. Our generic program representation, learnable program generator and universal design for modules makes our model much more flexible than neural module networks [1, 2] and thus more easily extensible to new problems and domains.

Supplementary Material

A. Implementation Details

We will release code to reproduce our experiments. We also detail some key implementation details here.

A.1. Program Generator

In all experiments our program generator is an LSTM sequence-to-sequence model [39]. It comprises two learned recurrent neural networks: the *encoder* receives the natural-language question as a sequence of words, and summarizes the question as a fixed-length vector; the *decoder* receives this fixed-length vector as input and produces the predicted program as a sequence of functions. The encoder and decoder do not share weights.

The encoder converts the discrete words of the input question to vectors of dimension 300 using a learned word embedding layer; the resulting sequence of vectors is then processed with a two-layer LSTM using 256 hidden units per layer. The hidden state of the second LSTM layer at the final timestep is used as the input to the decoder network.

At each timestep the decoder network receives both the function from the previous timestep (or a special `<START>` token at the first timestep) and the output from the encoder network. The function is converted to a 300-dimensional vector with a learned embedding layer and concatenated with the decoder output; the resulting sequence of vectors is processed by a two-layer LSTM with 256 hidden units per layer. At each timestep the hidden state of the second LSTM layer is used to compute a distribution over all possible functions using a linear projection.

During supervised training of the program generator, we use Adam [23] with a learning rate of 5×10^{-4} and a batch size of 64; we train for a maximum of 32,000 iterations, employing early stopping based on validation set accuracy.

A.2. Execution Engine

The execution engine uses a Neural Module Network [2] to compile a custom neural network architecture based on the predicted program from the program generator. The input image is first resized to 224×224 pixels, then passed through a convolutional network to extract image features; the architecture of this network is shown in Table 4.

The predicted program takes the form of a syntax tree; the leaves of the tree are `Scene` functions which receive visual input from the convolutional network. For ground-truth programs, the root of the tree is a function corresponding to one of the question types from the CLEVR dataset [19], such as `count` or `query_shape`. For predicted programs the root of the program tree could in principle be any function, but in practice we find that trained models tend only to predict as roots those function types that appear as roots of ground-truth programs.

Layer	Output size
Input image	$3 \times 224 \times 224$
ResNet-101 [14] conv4_6	$1024 \times 14 \times 14$
Conv(3×3, $1024 \rightarrow 128$)	$128 \times 14 \times 14$
ReLU	$128 \times 14 \times 14$
Conv(3×3, $128 \rightarrow 128$)	$128 \times 14 \times 14$
ReLU	$128 \times 14 \times 14$

Table 4. Network architecture for the convolutional network used in our execution engine. The ResNet-101 model is pretrained on ImageNet [37] and remains fixed while the execution engine is trained. The output from this network is passed to modules representing `Scene` nodes in the program.

Each function in the predicted program is associated with a *module* which receives either one or two inputs; this association gives rise to a custom neural network architecture corresponding to each program. Previous implementations of Neural Module networks [1, 2] used different architectures for each module type, customizing the module architecture to the function the module was to perform. In contrast we use a generic design for our modules: each module is a small residual block [14]; the exact architectures used for our unary and binary modules are shown in Tables 5 and 6 respectively.

In initial experiments we used Batch Normalization [17] after each convolution in the modules, but we found that this prevented the model from converging. Since each image in a minibatch may have a different program, our implementation of the execution engine iterates over each program in the minibatch one by one; as a result each module is only run with a batch size of one during training, leading to poor convergence when modules contain Batch Normalization.

The output from the final module is passed to a classifier which predicts a distribution over answers; the exact architecture of the classifier is shown in Table 7.

When training the execution engine alone (using either ground-truth programs or predicted programs from a fixed program generator), we train using Adam [23] with a learning rate of 1×10^{-4} and a batch size of 64; we train for a maximum of 200,000 iterations and employ early stopping based on validation set accuracy.

A.3. Joint Training

When jointly training the program generator and execution engine, we train using Adam with a learning rate of 5×10^{-5} and a batch size of 64; we train for a maximum of 100,000 iterations, again employing early stopping based on validation set accuracy.

We use a moving average baseline to reduce the variance of gradients estimated using REINFORCE; in particular our baseline is an exponentially decaying moving average of past rewards, with a decay factor of 0.99.

Index	Layer	Output size
(1)	Previous module output	$128 \times 14 \times 14$
(2)	Conv(3×3, $128 \to 128$)	$128 \times 14 \times 14$
(3)	ReLU	$128 \times 14 \times 14$
(4)	Conv(3×3, $128 \to 128$)	$128 \times 14 \times 14$
(5)	Residual: Add (1) and (4)	$128 \times 14 \times 14$
(6)	ReLU	$128 \times 14 \times 14$

Table 5. Architecture for unary modules used in the execution engine. These modules receive the output from one other module, except for the special `Scene` module which instead receives input from the convolutional network (Table 4).

Index	Layer	Output size
(1)	Previous module output	$128 \times 14 \times 14$
(2)	Previous module output	$128 \times 14 \times 14$
(3)	Concatenate (1) and (2)	$256 \times 14 \times 14$
(4)	Conv(1×1, $256 \to 128$)	$128 \times 14 \times 14$
(5)	ReLU	$128 \times 14 \times 14$
(6)	Conv(3×3, $128 \to 128$)	$128 \times 14 \times 14$
(7)	ReLU	$128 \times 14 \times 14$
(8)	Conv(3×3, $128 \to 128$)	$128 \times 14 \times 14$
(9)	Residual: Add (5) and (8)	$128 \times 14 \times 14$
(10)	ReLU	$128 \times 14 \times 14$

Table 6. Architecture for binary modules in the execution engine. These modules receive the output from two other modules. The binary modules in our system are `intersect`, `union`, `equal_size`, `equal_color`, `equal_material`, `equal_shape`, `equal_integer`, `less_than`, and `greater_than`.

Layer	Output size
Final module output	$128 \times 14 \times 14$
Conv(1×1, $128 \to 512$)	$512 \times 14 \times 14$
ReLU	$512 \times 14 \times 14$
MaxPool(2×2, stride 2)	$512 \times 7 \times 7$
FullyConnected($512 \cdot 7 \cdot 7 \to 1024$)	1024
ReLU	1024
FullyConnected($1024 \to \|\mathcal{A}\|$)	$\|\mathcal{A}\|$

Table 7. Network architecture for the classifier used in our execution engine. The classifier receives the output from the final module and predicts a distribution over answers $\mathcal{A}$.

A.4. Baselines

We reimplement the baselines used in [19]:

LSTM. Our LSTM baseline receives the input question as a sequence of words, converts the words to 300-dimensional vectors using a learned word embedding layer, and processes the resulting sequence with a two-layer LSTM with 512 hidden units per layer. The LSTM hidden state from the second layer at the final timestep is passed to an MLP with two hidden layers of 1024 units each, with ReLU nonlinearities after each layer.

CNN+LSTM. Like the LSTM baseline, the CNN+LSTM model encodes the question using learned 300-dimensional word embeddings followed by a two-layer LSTM with 512 hidden units per layer. The image is encoded using the same CNN architecture as the execution engine, shown in Table 4. The encoded question and (flattened) image features are concatenated and passed to a two-layer MLP with two hidden layers of 1024 units each, with ReLU nonlinearities after each layer.

CNN+LSTM+SA. The question and image are encoded in exactly the same manner as the CNN+LSTM baseline. However rather than concatenating these representations, they are fed to two consecutive Stacked Attention layers [46] with a hidden dimension of 512 units; this results in a 512-dimensional vector which is fed to a linear layer to predict answer scores.

This matches the CNN+LSTM+SA model as originally described by Yang *et al*. [46]; this also matches the CNN+LSTM+SA model used in [19].

CNN+LSTM+SA+MLP. Identical to CNN+LSTM+SA; however the output of the final stacked attention module is fed to a two-layer MLP with two hidden layers of 1024 units each, with ReLU nonlinearities after each layer.

Since all other other models (LSTM, CNN+LSTM, and ours) terminate in an MLP to predict the final answer distribution, the CNN+LSTM+SA+MLP gives a more fair comparison with the other methods.

Surprisingly, the minor architectural change of replacing the linear transform with an MLP significantly improves performance on the CLEVR dataset: CNN+LSTM+SA achieves an overall accuracy of 69.8, while CNN+LSTM+SA+MLP achieves 73.2. Much of this gain comes from improved performance on comparison questions; for example on shape comparison questions CNN+LSTM+SA achieves an accuracy of 50.9 and CNN+LSTM+SA+MLP achieves 69.7.

Training. All baselines are trained using Adam with a learning rate of 5×10^{-4} with a batch size of 64 for a maximum of 360,000 iterations, employing early stopping based on validation set accuracy.

B. Neural Module Network parses

The closest method to our own is that of Andreas *et al*. [1]. Their dynamic neural module networks first perform a dependency parse of the sentence; heuristics are then used to generate a set of *layout fragments* from the dependency parse. These fragments are heuristically combined, giving a set of *candidate layouts*; the final network layout is selected from these candidates through a learned reranking step.

Unfortunately we found that the parser used in [1] for VQA questions did not perform well on the longer questions in CLEVR. In Table 8 we show random questions from the CLEVR training set together with the layout frag-

Question:	*The brown object that is the same shape as the green shiny thing is what size?*
Fragments:	`(_what _thing)`
Question:	*What material is the big purple cylinder?*
Fragments:	`(material purple);(material big);(material (and purple big))`
Question:	*How big is the cylinder that is in front of the green metal object left of the tiny shiny thing that is in front of the big red metal ball?*
Fragments:	`(_what _thing)`
Question:	*Are there any metallic cubes that are on the right side of the brown shiny thing that is behind the small metallic sphere to the right of the big cyan matte thing?*
Fragments:	`(is brown);(is cubes);(is (and brown cubes))`
Question:	*Is the number of cyan things in front of the purple matte cube greater than the number of metal cylinders left of the small metal sphere?*
Fragments:	`(is cylinder);(is cube);(is (and cylinder cube))`
Question:	*Are there more small blue spheres than tiny green things?*
Fragments:	`(is blue);(is sphere);(is (and blue sphere))`
Question:	*Are there more big green things than large purple shiny cubes?*
Fragments:	`(is cube);(is purple);(is (and cube purple))`
Question:	*What number of things are large yellow metallic balls or metallic things that are in front of the gray metallic sphere?*
Fragments:	`(number gray);(number ball);(number (and gray ball))`
Question:	*The tiny cube has what color?*
Fragments:	`(_what _thing)`
Question:	*There is a small matte cylinder; is it the same color as the tiny shiny cube that is behind the large red metallic ball?*
Fragments:	`(_what _thing)`

Table 8. Examples of random questions from the CLEVR training set, parsed using the code by Andreas *et al*. [1] for parsing questions from the VQA dataset [3]. Each parse gives a set of *layout fragments* separated by semicolons; in [1] these fragments are combined to produce *candidate layouts* for the module network. When the parser fails, it produces the default fallback fragment `(_what _thing)`.

ments computed using the parser from [1]. For many questions the parser fails, falling back to the fragment `(_what _thing)`; when this happens then the resulting module network will not respect the structure of the question at all. For questions where the parser does not fall back to the default layout, the resulting layout fragments often fail to capture key elements from the question; for example, after parsing the question *What material is the big purple cylinder?*, none of the resulting fragments mention the *cylinder*.

Acknowledgements. We thank Ranjay Krishna, Yuke Zhu, Kevin Chen, and Dhruv Batra for helpful comments and discussion. J. Johnson is partially supported by an ONR MURI grant.

References

[1] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein. Learning to compose neural networks for question answering. In *NAACL*, 2016. 1, 2, 3, 5, 8, 9, 10, 11

[2] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein. Neural module networks. In *CVPR*, 2016. 1, 2, 3, 4, 5, 8, 9

[3] S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. Zitnick, and D. Parikh. VQA: Visual question answering. In *ICCV*, 2015. 1, 2, 4, 7, 11

[4] M. Balog, A. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2017. 2, 8

[5] J. Cai, R. Shin, and D. Song. Making neural programming architectures generalize via recursion. In *ICLR*, 2017. 8

[6] A. Das, S. Kottur, K. Gupta, A. Singh, D. Yadav, J. Moura, D. Parikh, and D. Batra. Visual dialog. In *CVPR*, 2017. 2

[7] J. Devlin, S. Gupta, R. Girshick, M. Mitchell, and C. L. Zitnick. Exploring nearest neighbor approaches for image captioning. *arXiv preprint arXiv:1505.04467*, 2015. 1

[8] A. Fukui, D. H. Park, D. Yang, A. Rohrbach, T. Darrell, and M. Rohrbach. Multimodal compact bilinear pooling for visual question answering and visual grounding. In *arXiv:1606.01847*, 2016. 2

[9] Y. Gao, O. Beijbom, N. Zhang, and T. Darrell. Compact bilinear pooling. In *CVPR*, 2016. 2

[10] R. Girshick. Fast R-CNN. In *ICCV*, 2015. 1

[11] Y. Goyal, T. Khot, D. Summers-Stay, D. Batra, and D. Parikh. Making the V in VQA matter: Elevating the role of image understanding in visual question answering. In *CVPR*, 2017. 2

[12] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014. 2

[13] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwinska, S. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. Badia, K. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. . Kavukcuoglu, and D. Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016. 2

[14] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 3, 9

[15] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. 4

[16] R. Hu, M. Rohrbach, J. Andreas, T. Darrell, and K. Saenko. Modeling relationships in referential expressions with compositional modular networks. In *CVPR*, 2017. 2

[17] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015. 9

[18] A. Jabri, A. Joulin, and L. van der Maaten. Revisiting visual question answering baselines. In *ECCV*, 2016. 1, 2

[19] J. Johnson, B. Hariharan, L. van der Maaten, L. Fei-Fei, C. L. Zitnick, and R. Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. In *CVPR*, 2017. 1, 2, 4, 5, 6, 7, 9, 10

[20] A. Joulin and T. Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *NIPS*, 2015. 2

[21] K. Kafle and C. Kanan. Visual question answering: Datasets, algorithms, and future challenges. In *arXiv preprint arXiv:1610.01465*, 2016. 2

[22] Ł. Kaiser and I. Sutskever. Neural GPUs learn algorithms. In *ICLR*, 2016. 2

[23] D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015. 9

[24] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *ACL*, pages 423–430, 2003. 2

[25] R. Krishna, Y. Zhu, O. Groth, J. Johnson, K. Hata, J. Kravitz, S. Chen, Y. Kalantidis, L.-J. Li, D. A. Shamma, et al. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *IJCV*, 2017. 2

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*. 2012. 1

[27] K. Kurach, M. Andrychowicz, and I. Sutskever. Neural random-access machines. In *ICLR*, 2016. 2

[28] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 2016. 1

[29] C. Liang, J. Berant, Q. Le, K. D. Forbus, and N. Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *arXiv preprint arXiv:1611.00020*, 2016. 2, 8

[30] P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. In *ACL*, 2011. 2

[31] J. Lu, J. Yang, D. Batra, and D. Parikh. Hierarchical question-image co-attention for visual question answering. In *NIPS*, 2016. 2

[32] M. Malinowski and M. Fritz. A multi-world approach to question answering about real-world scenes based on uncertain input. In *NIPS*, 2014. 2

[33] M. Malinowski, M. Rohrbach, and M. Fritz. Ask your neurons: A neural-based approach to answering questions about images. In *ICCV*, 2015. 2, 4

[34] A. Mallya and S. Lazebnik. Learning models for actions and person-object interactions with transfer to question answering. In *ECCV*, 2016. 2

[35] A. Neelakantan, Q. V. Le, and I. Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *ICLR*, 2016. 2

[36] S. Reed and N. De Freitas. Neural programmer-interpreters. In *ICLR*, 2016. 2, 8

[37] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. ImageNet large scale visual recognition challenge. *IJCV*, 2015. 3, 9

[38] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus. End-to-end memory networks. In *NIPS*, 2015. 2

[39] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014. 3, 9

[40] M. Tapaswi, Y. Zhu, R. Stiefelhagen, A. Torralba, R. Urtasun, and S. Fidler. Movieqa: Understanding stories in movies through question-answering. In *CVPR*, 2016. 2

[41] J. Weston, S. Chopra, and A. Bordes. Memory networks. In *ICLR*, 2015. 2

[42] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(23), 1992. 4

[43] T. Winograd. *Understanding Natural Language*. Academic Press, 1972. 2

[44] Q. Wu, D. Teney, P. Wang, C. Shen, A. Dick, and A. van den Hengel. Visual question answering: A survey of methods and datasets. In *arXiv preprint arXiv:1607.05910*, 2016. 2

[45] C. Xiong, S. Merity, and R. Socher. Dynamic memory networks for visual and textual question answering. *ICML*, 2016. 2

[46] Z. Yang, X. He, J. Gao, L. Deng, and A. Smola. Stacked attention networks for image question answering. In *CVPR*, 2016. 4, 5, 10

[47] W. Zaremba, T. Mikolov, A. Joulin, and R. Fergus. Learning simple algorithms from examples. In *ICML*, 2016. 2

[48] W. Zaremba and I. Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014. 2

[49] W. Zaremba and I. Sutskever. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 2015. 2

[50] P. Zhang, Y. Goyal, D. Summers-Stay, D. Batra, and D. Parikh. Yin and yang: Balancing and answering binary visual questions. In *CVPR*, 2016. 2

[51] Y. Zhu, O. Groth, M. Bernstein, and L. Fei-Fei. Visual7W: Grounded question answering in images. In *CVPR*, 2016. 1, 2