Inferring and Executing Programs for Visual Reasoning Supplementary Material

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

1. Implementation Details

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

1.1. Program Generator

In all experiments our program generator is an LSTM sequence-to-sequence model [9]. 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 [7] 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.

1.2. 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 1.

Layer	Output size
Input image	$3 \times 224 \times 224$
ResNet-101 [4] conv4_6	$1024 \times 14 \times 14$
Conv(3×3, $1024 \to 128$)	$128 \times 14 \times 14$
ReLU	$128 \times 14 \times 14$
Conv(3×3, $128 \to 128$)	$128 \times 14 \times 14$
ReLU	$128 \times 14 \times 14$

Table 1. Network architecture for the convolutional network used in our execution engine. The ResNet-101 model is pretrained on ImageNet [8] and remains fixed while the execution engine is trained. The output from this network is passed to modules representing `Scene` nodes in the program.

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 [6], 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.

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 [4]; the exact architectures used for our unary and binary modules are shown in Tables 2 and 3 respectively.

In initial experiments we used Batch Normalization [5] 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 implemen-

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 2. 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 1).

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 3. 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`.

tation 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 4.

When training the execution engine alone (using either ground-truth programs or predicted programs from a fixed program generator), we train using Adam [7] 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.

1.3. 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.

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 4. 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 $\mathcal{A}$.

1.4. Baselines

We reimplement the baselines used in [6]:

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 1. 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 [10] 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*. [10]; this also matches the CNN+LSTM+SA model used in [6].

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.

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 5. 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)`.

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.

2. 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 5 we show random questions from the CLEVR training set together with the layout fragments 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*.

References

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

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

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

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 1

[5] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015. 1

[6] 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

[7] D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015. 1, 2

[8] 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. 1

[9] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014. 1

[10] Z. Yang, X. He, J. Gao, L. Deng, and A. Smola. Stacked attention networks for image question answering. In *CVPR*, 2016. 2