DDRPROG: A CLEVR DIFFERENTIABLE DYNAMIC REASONING PROGRAMMER

Anonymous authors
Paper under double-blind review

ABSTRACT

We present a generic dynamic architecture that employs a problem specific differentiable forking mechanism to encode complex assumptions about the problem data structure. We adapt and apply our model to CLEVR Visual Question Answering, giving rise to the DDRprog architecture; compared to previous approaches, our model achieves higher accuracy in half as many epochs with five times fewer learnable parameters. Our model directly models underlying question logic using a recurrent controller that jointly predicts and executes functional neural modules; it explicitly forks subprocesses to handle logical branching. While FiLM and other competitive models are static architectures with less supervision, we argue that inclusion of program labels enables learning of higher level logical operations–our architecture achieves particularly high performance on questions requiring counting and integer comparison. We further demonstrate the generality of our approach though DDRstack – an application of our method to reverse Polish notation expression evaluation in which the inclusion of a stack assumption allows our approach to generalize to long expressions, significantly outperforming an LSTM with ten times as many learnable parameters.

1 INTRODUCTION AND RELATED WORKS

Deep learning is inherently data driven – visual question answering, scene recognition, language modeling, speech recognition, translation, and all other supervised tasks can by definition be expressed as: given relevant information x, predict desired quality y. While the field has attempted to model an immense set of underlying data structures with neural architectures, core convolutional and recurrent building blocks were designed with only the most general notions of spatial and temporal locality in mind. In some cases, additional information about the problem can be expressed simply as an additional loss. When hard logical assumptions are present, it is far from obvious how to do so in a manner compatible with backpropagation.

In this work, we examine two problems: visual question answering (VQA) on the CLEVR (Johnson et al. (2016)) dataset and expression evaluation in reverse Polish notation (RPN). In the former, questions seen at training time contain direct programmatic representations well modeled by a set of discrete logical operations and a stack requiring at most one recursive call. The latter is an extreme case with deep recursion requiring a full stack representation, but supervision is also available at test time. We present a generic architecture that uses a problem specific forking behavior to leverage structural information about the input data. Our approach resolves common differentibility issues and is effective on both problems: we achieve a moderate improvement over previous state-of-the-art on CLEVR and outperform a much larger baseline model on RPN.

Prior work on CLEVR VQA is largely categorized by dynamic and static approaches. IEP (Johnson et al. (2017)) and NMN (Hu et al. (2017)) both generalized the original neural module networks architecture (Andreas et al. (2015)) and used the functional annotations in CLEVR to predict a static program which is then assembled into a tree of discrete modules and executed. IEP further demonstrated success when program annotations are available for only a few percent of questions. These are most similar to our approach; we focus largely upon comparison to IEP, which performs significantly better. RN (Santoro et al. (2017)) and FiLM (Perez et al. (2017b)), the latter being the direct successor of CBN (Perez et al. (2017a)) are both static architectures which incorporate some form of implicit reasoning module in order to achieve high performance without program annotations.

In contrast, our architecture uses program annotations to explicitly model the underlying question structure and jointly executes the corresponding functional representation. As a result, our architecture performs comparably on questions requiring linear application of visual transformations, but it performs significantly better on questions requiring higher level operations such as counting and numerical comparison.

The RPN task is, to our knowledge, novel. The most comparable work we are aware of is StackRNN Joulin & Mikolov (2015)), which endows a recurrent architecture with the ability to push and pop from a stack. However, StackRNN neither learns nor assumes an explicit stack structure from the labels. While this approach is more general, StackRNN is limited to learning simpler data structures and is applied only to memorization, binary addition, and language modeling (it underperforms an LSTM in this case). In contrast, our model makes stronger structural assumptions that allow it to perform well on expression evaluation in base ten with four discrete operations.

2 DATASETS

2.1 CLEVR

CLEVR is a synthetic VQA dataset consisting of 100k images and 1 million question/answer pairs. Over 850k of these questions are unique. Images are high quality 3D Blender (Blender (2017)) renders of scenes containing geometric objects of various shapes, sizes, colors, textures, and materials. Thus the dataset is quite realistic despite being synthetic. Furthermore, the authors ran comprehensive tests to avoid exploitable biases in the data. As the objects are geometric figures, no external knowledge of natural images is required, as in earlier VQA datasets. Most importantly, CLEVR provides an additional representation of each question as a functional program. For example, "How many red spheres are there?" is represented as [*filter_red, filter_sphere, count*]. Some questions require nonlinear program structure, such as "How many objects are red or spheres", which is represented by a tree with two branches [*filter_red*] and [*filter_sphere*] followed by a binary [*union*] operation and a final [*count*]. We are unaware of any dataset with comparable properties of this size and complexity. This makes the dataset uniquely useful for evaluating dynamic architectures, which are able to directly learn and execute discrete functional program cells.

2.2 RPN

The specific expression presentation we consider is the reverse Polish notation (RPN) form [NUM]*(n+1)-[OP]*n, that is, $n+1$ numbers followed by n operations. This simplifies the problem by eliminating consideration for order of operations. Thus the task is: given a sequence of tokens corresponding to a valid expression in reverse Polish notation, evaluate the expression and produce a single real valued answer. As currently posed, this problem is incredibly complex for large n. Long expressions are highly nonlinear; without additional supervision, the problem is essentially to learn a hash function, which is intractable in general. We therefore modify the problem as such: instead of producing only the final expression evaluation, produce the sequence of answers to all n intermediate expressions in the answer labels. For example, 2 3 4 + *, the expected output would be [7, 14] because 3+4=7 and 2*7=14. We further assume the stack structure of the problem is available to the architecture should it be capable of taking advantage of such information. The problem is still sufficiently complex–note that to the model, $\{1, 3, 4, +, *\}$ would all be meaningless tokens: it must learn both the NUM and the OP tokens.

The dataset consists of 100k train, 5k validation, and 20k test expression with $n = 10$ – that is, 11 numbers followed by 10 operations. We also provide a 20k expression generalization set with $n = 30$. The label for each question contains the n solutions to each intermediate operation. During data generation, we sample NUM and OP tokens uniformly, reject expressions including division by zero, and also omit expressions that evaluate to over 100 in magnitude. The NUM tokens correspond to 0, 0.1, ..., 0.9 and the OP tokens correspond to +, -, *, /; however, architectures are not privy to this information.

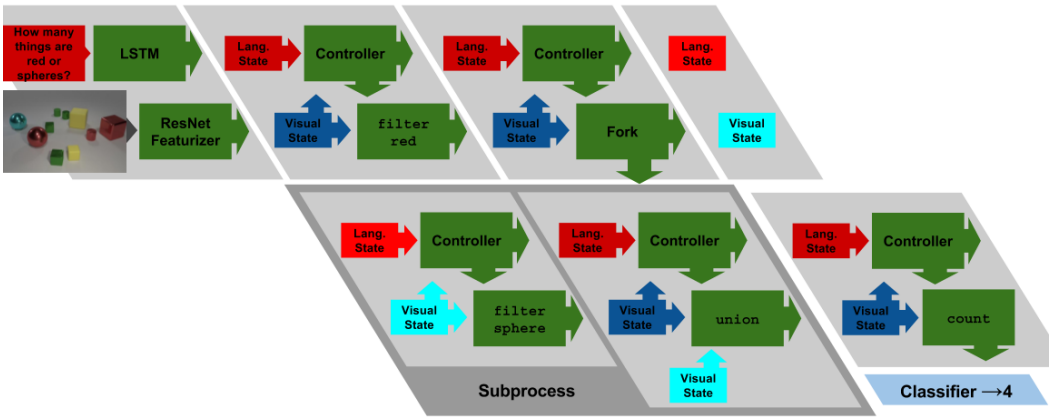

Figure 1: Visualization of the DDRprog architecture. The particular configuration shown models and answers the question "How many things are red or spheres?" by predicting $[filter_red, fork, filter_sphere, union, count]$

3 DDR ARCHITECTURE

Our architecture is conceptually problem independent, but it requires adaptation to the specific data structure of the task. The core is a recurrent controller similar to NPI (Reed & de Freitas (2015)). In both CLEVR VQA and RPN, our model maintains a discrete set of neural modules, as in IEP and NMN. However, unlike NPI, our model responds flexibly to the problem supervision. In VQA, modules are only known at training time: our CLEVR architecture, DDRprog, therefore learns to predict them jointly during inference. In the RPN task, modules correspond to operations and are also known at test time: our RPN architecture, DDRstack, therefore executes them directly. Details of the DDR variants DDRprog and DDRstack are provided below.

3.1 CLEVR VISUAL QUESTION ANSWERING: DDRPROG

Our architecture consists of the core recurrent controller and a set of neural modules, which correspond to program annotations in CLEVR. Some standard encoders are required to adapt DDR to the mixed language and visual data, as well as a small classifier. We provide pseudocode in Algorithm 1, a visual representation in Figure 1, and subnetwork details in Table 2 (see Appendix).

Algorithm 1 DDRprog. Note that **CNN** produces a flattened output and **Controller** also performs a projection and argmax over program scores to produce $programPrediction$

```
img, question ← x
stack ← Stack()
img, imgCopy ← ResNetFeaturizer(img)
langState ← LSTM(question)
for i = 1...MaxProgramLength do
    visualState ← CNN(img)
    programPrediction ← Controller(visualState, langState)
    cell ← Cells[programPrediction]
    if cell is Fork then
        stack.push(cell(img, imgCopy))
    else if cell is Binary then
        img ← cell(stack.pop(), img)
    else
        img ← cell(img)
    end if
end for
return Classifier(img)
```

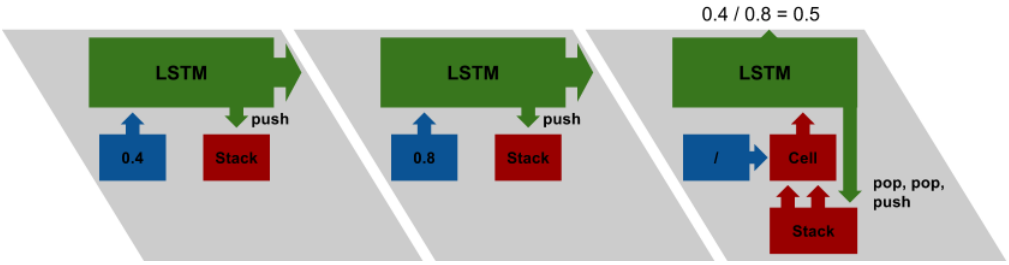

Figure 2: Visualization of the DDRstack architecture with $n = 1$. This particular configuration evaluates the [NUM][NUM][OP] formatted expression [0.4, 0.8, /], which is 0.4/0.8=0.5. NUM tokens are embedded before being passed to the LSTM. OP tokens are used as an index to select the corresponding cell. LSTM predictions at each OP token are used to predict intermediate losses (there is only one for $n = 1$).

The input data x for each sample consists of an (image, question, program) triple and an answer label y. As the program data is not available at test time, our model must learn to predict it. The network first encodes the question with an LSTM and the image with the same ResNet (He et al. (2015)) feature extractor used in IEP and FiLM. This produces a language state and a visual state. Both of these are passed to the controller–we use a recurrent highway network (RHN) (Zilly et al. (2016)), though we could have used a more familiar LSTM(Hochreiter & Schmidhuber (1997)). Note that the RHN accepts flat inputs whereas the visual state contains convolutional maps. We resolve this disparity with an additional encoding module similar in design to the IEP classifier.

At each time step, the controller predicts an index over the learnable neural modules, which use the same structure as in IEP with smaller hidden dimension. The corresponding module is applied to the visual state, which is then updated. The final module prediction is followed by a small classifier network, which uses the IEP classifier.

This architecture is sufficient on the subset of CLEVR programs that are linear. However, some CLEVR questions contain a logical branching operation (e.g. *are there more of ... than ... ?*) and cannot be answered by purely linear programs. In this case, our architecture forks a subprocess that shares the main process's language and visual states. The subprocess proceeds normally until it encounters a binary cell. As shown in Figure 1, the binary cell merges the visual states of the main process and subprocess and applies the given function. The subprocess then rejoins the main program. Note that while CLEVR programs contain at most one branch operation, our architecture is likely to generalize to more difficult questions, as the fork module could push to a deeper stack (as in our DDRstack architecture, below) or in general interface with trees, priority queues, or any other data structure to handle arbitrarily complex program traces.

The fork module must differ from a standard unary module, as it is necessary to pass the original ResNet features to the subprocess in addition to the current visual state. Consider the question: "Is the red thing larger than the blue thing?" In this case, the left program branch filters by red; it is impossible to recover the blue objects in the right branch given a red filtered image. We found that it is insufficient to pass only the original images to the subprocess, as the controller is small and has difficulty tracking the current branch. We therefore use a variant of the binary module architecture that merges the original ResNet features with the current visual state (see Algorithm 1). As the fork module is shared across all branch patterns, it is larger than the other binary modules and also one layer deeper.

3.2 Expressions in Reverse Polish Notation: DDRstack

Our architecture consists of the core recurrent controller (in this case an LSTM) and a set of 4 (one per OP) learnable binary modules, as well as an explicit stack. DDRstack can be viewed as a dynamic wrapper that augments the LSTM to include a stack assumption; it is also a direct adaptation of the analytical RPN expression evaluation algorithm to a neural architecture. We provide high level pseudocode for the model in Algorithm 2 and a visual representation in Figure 2.

Algorithm 2 DDRstack

```
tokens ← x
stack ← Stack()
state ← RandomLSTMInitialization
for all tok in tokens do
    if tok is a NUM then
        out ← Embed(tok)
    else if tok is a OP then
        arg2 ← stack.pop()
        arg1 ← stack.pop()
        out ← Cells[tok](arg1, arg2)
    end if
    out, state ← LSTMCell(out, state)
    stack.push(out)
end for
return Projection(out)
```

We train a baseline vanilla LSTM supervised with the intermediate solutions in the last n timesteps. DDRstack uses the same baseline LSTM structure as its core controller, but it processes NUM and OP tokens differently, as well as interacting with the internal stack, as outlined below. Predictions are made in the last n timesteps, as in the vanilla model (Algorithm 2 shows only the final return).

NUM: Our model embeds the token and passes it to the LSTM. It then pushes the result to the stack.

OP: Our model pops twice, calls the OP specific binary cell, and then passes the results to the LSTM. It then pushes the result to the stack. The binary cell is a simple concatenation of the arguments followed by a single fully connected layer with no nonlinearity.

4 DDRprog: CLEVR VQA Experiments and Discussion

4.1 Experiments

We train our model with Adam (Kingma & Ba (2014)) on the full CLEVR dataset, including all program annotations. The hyperparameters detailed in Table 3 (see Appendix) are minimally configured from defaults, excepting learning rate and rough network size. The network overall has 9M parameters. We exclude the ResNet feature extractor from all calculations because it is also present in the best FiLM model. Their work further demonstrated it is fairly straightforward to replace it with a from-scratch feature extractor with minimal loss in accuracy.

We pass the ground truth program labels to the model during training. Critically, the program labels are only used on the validation set for the purpose of model selection, and our final accuracy is obtained by rerunning the validation set without the ground truth programs. We train on a single GTX 1080 TI and, after 35 epochs, our model matches the previous state-of-the-art accuracy of 97.7 percent. We continue training until the 52nd epoch, dropping the learning rate to 1e-5 for the last few epochs to ensure convergence, and obtain 98.3 percent accuracy. The model predicts program cells with 99.98 percent accuracy.

4.2 Discussion

RN, FiLM, IEP, and our architecture are all fairly close in overall accuracy, but the accuracies in each category vary. From Table 1, no architecture has particular difficulty with Exist, Query, or Compare questions; the main differentiating factors are Count and Compare Integer. Though Compare Integer is the smallest class of questions and is therefore assigned less importance by the cross entropy loss, the IEP result suggests that this does not cause models to ignore this question type. We therefore consider Count and Compare Integer to be the hardest unary and binary tasks, respectively, and we assign most important to these question subsets in our analysis.

Model	Parameters	Epochs	Exist	Count	Compare Integer	Query	Compare	Overall
Q-type mode	-	-	50.2	34.6	51.1	36.9	51.2	42.1
LSTM	-	-	61.8	42.5	70.0	36.5	51.1	47.0
CNN+LSTM	-	-	68.2	47.8	70.1	48.9	54.6	54.3
CNN+LSTM+SA	-	-	68.4	57.5	67.7	87.7	52.0	69.8
CNN+LSTM+SA+MLP	-	-	77.9	59.7	75.1	80.9	70.8	73.2
Human	-	-	96.6	86.7	86.4	94.9	96.0	92.6
End-to-End NMN	-	-	85.7	68.5	84.9	89.9	88.7	83.7
IEP	41M	12	97.1	92.7	**98.7**	98.1	98.8	96.9
DDRprog	9M	52	98.8	**96.5**	98.4	99.1	**99.0**	**98.3**
RN	500k	1000	97.8	90.1	93.6	97.9	97.1	95.5
FiLM/CBN	>50M	80	**99.2**	94.5	93.8	**99.2**	**99.0**	97.6

Table 1: Accuracy on all CLEVR question types for baselines and competitive models. The Human baseline is from the original CLEVR work. SA refers to stacked spatial attention Yang et al. (2015)
.

We first compare to IEP. While our model is less than 1/4 the size of IEP (see Table 1), it closely matches binary performance (+0.2 percent on Compare, -0.3 percent on Compare Integer) and perform at least 2x better across all unary tasks (+1.7 percent on Exist, +3.8 percent on Count, + 1.0 percent on Query). We believe that our model's lack of similar gains on binary task performance can be attributed to the use of a singular fork module shared across all binary modules. The fork module is responsible for informing the right program branch of progress made in the left program branch. We have observed that this module is essential to obtaining competitive performance on binary tasks; it is likely suboptimal to use a large shared fork module as opposed to a separate smaller cell for each binary cell. However, this still requires experimental verification.

Our model surpasses RN in all categories of reasoning, achieving a 2.6x reduction in overall error. RN achieves especially impressive results for its size and lack of program labels. However, it is questionable whether the all-to-all comparison model will generalize to more logically complex questions. In particular, Count operations do not have a natural formulation as a comparison between pairs of objects, in which case our model achieves a 6.4 percent improvement. RN also struggles on the challenging Compare Integer subtask, where we achieve a 4.8 percent improvement. Furthermore, it is unclear how essential high epoch counts are to the model's performance. As detailed in Table 1, RN was trained in a distributed setting for 1000 epochs. Both our result and FiLM were obtained on single graphics cards and were only limited in number of epochs for practicality–FiLM had not fully converged, and our model was unregularized.

FiLM also surpasses RN in all categories and is the only static, no-programs architecture to surpass IEP in raw accuracy, and it does so with only slightly larger model size. FiLM achieves 1.5x relative improvement over our architecture on Exist questions, but this is offset by our 1.5x relative improvement on Count questions. FiLM does not perform particularly poorly on Count questions, but the contrast between its Compare Integer and its Exist/Query/Compare performance is quite striking–While FiLM surpasses IEP in all other categories, both IEP and our model achieve a roughly 4x improvement on Compare Integer questions (4.9 and 4.6 percent, respectively). We believe this is because it is difficult to model the more complex binary structures using only implicit branching through batch normalization parameters–many Compare (attribute) questions can be more easily resolved through a sequence of purely visual manipulations.

5 DDRSTACK: RPN EXPERIMENTS AND DISCUSSION

5.1 EXPERIMENTS

For our architecture, we use hidden dimension 32 throughout the model, resulting in only 17k parameters overall. We train with Adam using learning rate 1e-3 and obtain a test L1 error of 0.17 after 63 epochs. Using the same hidden dimension in the pure LSTM baseline (9k parameters) results in

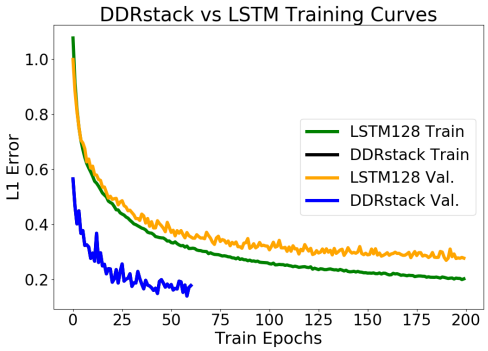

Figure 3: Training curves for DDRstack (overlapping, 17k parameters) and the LSTM128 baseline (255k parameters) on RPN10.

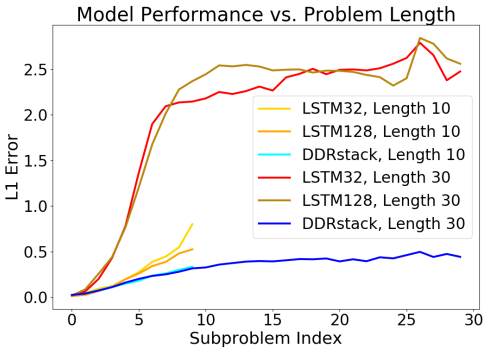

Figure 4: Generalization performance of DDRstack and the LSTM baseline to RPN30 after training on RPN10.

test error 0.28. We overcompensate for the difference in model size by increasing the hidden dimension of the LSTM to 128 (255k parameters), resulting in an only slightly lower test error of 0.24 after nearly 3000 epochs. Training curves for the larger LSTM baseline and DDRstack are shown in Figure 3.

After training both models on problems of length $n = 10$, we test both models on sequences of length $n = 10$ and $n = 30$. For a sequence of length n both models predict values not only for the entire expression but also for all n subproblems where index n corresponds to evaluating the entire sequence. For sequences of both lengths, we evaluate accuracy for the predicted answers on all subproblems. Results are shown in Figure 4.

5.2 Discussion

With a sufficient number of epochs and a large parameter budget, it is probable that an LSTM would surpass our model. We could increase our model size to match, but discounting this line of reasoning, we argue that the LSTM would still fail to learn the underlying stack structure and would therefore fail to generalize to longer expressions.

From Figure 4, both the small and large LSTM baselines approximately match our model's performance on the first 5 subproblems of the $n = 10$ dataset. From $n = 6$ to $n = 10$, the performance gap grows between our models–in particular, the small LSTM is unable to learn deep stack behavior, and performance decays sharply.

Both the small and large LSTM perform comparably on all subproblems of the $n = 30$ dataset. Somewhat confusingly, performance is far worse on the first few subproblems of this dataset than on the test set of the original task. This is not an error: recall the question formatting [NUM]*$(n + 1)$-[OP]*n. The leading subproblems do not correspond to the leading tokens of the question, but rather to a central crop. The rapid increase in error on the LSTM implies that it could not even learn this property, let alone the stack structure. Instead, it memorized all possible subproblems of length $n \in \{1, 2, 3\}$ expressions preceding the first few OP tokens. Performance quickly decays to L1 error greater than 2.0. Considering that the standard deviation of answers (discounting the first few subproblems) is approximately 6.0, this corresponds to mostly noise. In contrast, our model's explicit incorporation of the stack assumption results in a smooth generalization curve with a gradual decay in performance as problem length increases.

We briefly address a few likely concerns surrounding the above line of reasoning. First, one might argue that the comparison is invalid, as the LSTM does not incorporate any explicit knowledge of the problem structure. While this evaluation is correct, it is antithetical to the purpose of our architecture. The LSTM baseline does not incorporate this additional information because there is no obvious way to include it–the prevailing approach would be to ignore it and then argue the model's superiority on the basis that it performs well with less supervision. In contrast, our dynamic approach is a minimal augmentation that allows our model to achieve a dramatic increase in performance and generalization precisely by incorporating additional supervision. This motivation also separates

DDRstack from previous architectures that include some variant of a stack, such as StackRNN. The former makes a hard assumption about the data structure in order to solve difficult problems, whereas the latter learns to make use of a stack from scratch at the cost of constraint to simpler problems. The hardest task on which StackRNN was evaluated is binary addition; in comparison, DDRstack effectively evaluates long expressions in base 10 with 4 different operations.

6 CONCLUSION

We have introduced a novel architecture that differentiably forks subprocesses in order to encode complex assumptions about the data structure into the model. We have demonstrated that our architecture is viable when certain structural details are omitted at test time: in the case of CLEVR VQA, our architecture learns to predict the missing program annotations at test time and achieves state-of-the-art performance. When structural details are also available at test time, as in the RPN stack structure, our architecture is able to smoothly generalize whereas baseline LSTM performance immediately decays. It is our intent to continue refining the versatility of our architecture, including more accurate modeling of the fork module, as mentioned in our CLEVR VQA discussion, as well as to generalize to the semisupervised setting where some datapoints in a single problem contain more structural information than others, as in transfer learning to CLEVR Humans. It is our hope that our architecture and its design principles enable modeling of complex data structure assumptions across a wide class of problems where standard monolithic approaches would ignore such useful properties.

REFERENCES

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Deep compositional question answering with neural module networks. *CoRR*, abs/1511.02799, 2015. URL http://arxiv.org/abs/1511.02799.

Blender. *Blender - a 3D modelling and rendering package*. Blender Foundation, Blender Institute, Amsterdam, 2017. URL http://www.blender.org.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to reason: End-to-end module networks for visual question answering. *CoRR*, abs/1704.05526, 2017. URL http://arxiv.org/abs/1704.05526.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C. Lawrence Zitnick, and Ross B. Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. *CoRR*, abs/1612.06890, 2016. URL http://arxiv.org/abs/1612.06890.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Fei-Fei Li, C. Lawrence Zitnick, and Ross B. Girshick. Inferring and executing programs for visual reasoning. *CoRR*, abs/1705.03633, 2017. URL http://arxiv.org/abs/1705.03633.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *CoRR*, abs/1503.01007, 2015. URL http://arxiv.org/abs/1503.01007.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

Ethan Perez, Harm de Vries, Florian Strub, Vincent Dumoulin, and Aaron Courville. Learning visual reasoning without strong priors, 2017a.

Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. Film: Visual reasoning with a general conditioning layer, 2017b.

Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015. URL http://arxiv.org/abs/1511.06279.

Adam Santoro, David Raposo, David G. T. Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy P. Lillicrap. A simple neural network module for relational reasoning. *CoRR*, abs/1706.01427, 2017. URL http://arxiv.org/abs/1706.01427.

Zichao Yang, Xiaodong He, Jianfeng Gao, Li Deng, and Alexander J. Smola. Stacked attention networks for image question answering. *CoRR*, abs/1511.02274, 2015. URL http://arxiv.org/abs/1511.02274.

Julian G. Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *CoRR*, abs/1607.03474, 2016. URL http://arxiv.org/abs/1607.03474.

APPENDIX

Table 2: Architectural details of subnetworks in DDRprog as referenced in Figure 1 and Algorithm 1. Finegrained layer details that would only be necessary for reproducing our results are not provided, as a clean copy of our source will be released pending publication.

Subnetwork	**Details**
ResNetFeaturizer	Features from ResNet101 pretrained on ImageNet, as in IEP and FiLM
LSTM	2-Layer LSTM that encodes the question
CNN	IEP classifier variant; produces a flat visual state.
Controller	Recurrent Highway Network for language and CNN Encoded visual states
Cells	IEP set of unary and binary modules, plus our fork module and pads

Table 3: Hyperparameter details for DDRprog. Only the learning rate and model size were coarsely cross validated due to hardware limitations: hyperparameter are not optimal.

Module	**Architecture**
Hidden dimension, all convolutional modules	128
Hidden dimension, all recurrent modules	64
Question encoder depth	2
Recurrent controller depth	3
Question vocabulary embedding dimension	300
Learning rate	1e-4