Fast and CLEVR Visual Reasoning Programs via Improved Topologically Batched Graph Execution

Joseph Suarez*
joseph15@stanford.edu

Clare Zhu*
clarezhu@stanford.edu

Abstract

We implement, explore, and heavily optimize the recent dynamic program generator + execution engine architecture of Johnson et al. for visual question answering (VQA) on his CLEVR dataset. The execution engine is a severe bottleneck in performance: as it is dynamically assembled per-question, it is impossible to naively parallelize over batch size. We present a variant of topological sort that both improves theoretical complexity bounds and yields large practical speedups. Our highly unoptimized implementation benefits from 2X faster training, 5.5X faster inference, and 14X faster neural cell execution, which takes less time than copying data to the GPU.

1. Introduction

The problem we address is neither VQA nor optimization of a single architecture. Our motivation is to accelerate a large class of dynamic architectures such that they become computationally comparable to their static counterparts. This cause is not motivated only by the recent successes of dynamic architectures, but by their numerous desirable properties that make them likely to retain and increase in importance in the future.

We specifically explore Johnson et al.'s recent work [13] because we believe it to be an exemplar of the potential advantages of dynamic architectures. The execution engine's modules not only yield dramatic accuracy gains over all strong baselines, but also are currently the best example of explicit modularization of knowledge. An essential aspect of intelligence is the ability to break one's understanding of the world into separate concepts without conflating unrelated ideas. Until now, it has been quicker to bombard a single (often not human-readable) network with information instead of attempting to learn distinct logical concepts. This is backward, given the success of the work of Johnson et al. Our work places his approach on equal computational footing with single network architectures.

* Denotes equal contribution

1.1. Motivation

In our earlier work, we emphasized our intentions of improving non-differentiability issues between the program generator and the execution engine. However, upon completing our implementation of the CLEVR result, we found that the vast majority of both error and computational time come from the execution engine. We thus chose to focus our efforts on computational cost for the reasons above, as well as the fact that the model already performs well above human accuracy.

1.2. Related Work

Previous notable dynamic graph results include neural module networks [3] [2] and improvements thereupon [11], which form the basis of the execution engine of Johnson et al. The difference is that latter's architecture is built on generic, minimally-engineered neural network blocks, which are more likely to generalize to a wider class of problems than the original neural module networks approach, which uses a heavily-engineered question parser and custom per-module architectures. Whereas improvement upon neural module networks constitutes improvement upon a single architecture, improvement on the CLEVR architecture is generalizable to a wide class of models under a minimal set of assumptions (see Discussion).

Additional dynamic graph results include neural Turing machines [6] and several improvements thereupon [7]. These approaches share in common with memory networks [19] the goal of augmenting architectures with queryable memory for read/write use during inference. However, these approaches are still underdeveloped, and despite several improvements in differentiability [18] and complexity of the read/write operations [14], to our knowledge, no architecture has achieved $O(1)$ memory access and efficient differentiable training. Furthermore, while such architectures are applicable in problems requiring long-term memory, visual question answering places more focus on short term memory. These works and the CLEVR result both tend towards higher level reasoning and both suffer from high computational complexity bounds, but they are otherwise largely unrelated.

Short term memory is more closely related to the concept of attention [5] and several improvements [15], all of which share the key concept of allowing a network to directly query the states of all pieces of the input data. This allows a network to focus, or "attend," to relevant portions of the input during inference. While more standard in natural language processing and machine translation, attention has been generalized to images as *spatial attention* [22]. While spatial attention was a key component of the original neural module networks work, the recent CLEVR result achieves strong performance without the need for such problem-specific, per-module engineering.

1.2.1 CLEVR

Our work is built atop the recently published CLEVR dataset [12] and subsequent results on it, both of which we examine in detail. CLEVR is a VQA dataset comprising 70K images and 700K questions/answers/programs triplets. Images are synthetic but high quality 3D renders of geometric objects with varying shapes, sizes, colors, and textures. The standard VQA task is given by (question, image) $\rightarrow$ (answer). The difference lies in the inclusion of *programs* in CLEVR, which are functional representations of the questions. CLEVR therefore allows VQA to be split between two intermediate tasks, as in Johnson et al.'s recent work: (question) $\rightarrow$ (program) and (program, image) $\rightarrow$ (answer).

One might argue that intermediate programs are unrealistic, as one is unlikely to have program annotations in large, realistic tasks. However, the advantage gained from dynamic architectures when they are available is too large to discount. Additionally, earlier visual question answering datasets, such as the eponymous VQA [4], are free-form and extremely difficult. However, this difficulty is likely not resolvable by incremental progress in network architectures without additional labeling information. This argument is corroborated by the existence of strong biases in previous VQA datasets, as discussed in the original CLEVR work, which further challenges the already low accuracies on the VQA dataset as compared to VQA on CLEVR.

CLEVR is also much more realistic than previous synthetic datasets, such as DAQUAR [16], which comprises only 8 question templates used to generate 420 unique questions. Johnson et al. took ample precautions to avoid biases in the questions without compromising realism or variety. This is illustrated by the success of his result on human written questions, as considered in his original work.

From the program generator + execution engine CLEVR result, it seems likely that one could collect a small number of annotations on realistic datasets and use these to initialize the program generator. However, we contend that one could also train the program generator on CLEVR and fine-tune

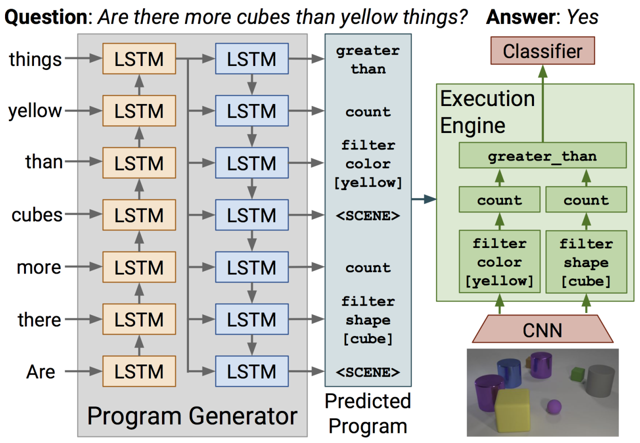

Figure 1. Architecture of the CLEVR model (Johnson et al. 2017)

jointly on the other dataset. This is likely to be a more reasonable approach, considering program annotations on synthetic datasets can be generated directly and without error. This approach of transfer learning from synthetic datasets is likely to become more viable once synthetic datasets advance and become even more realistic.

1.2.2 Visual Reasoning Programs

The CLEVR result consists of a program generator and execution engine as in Fig. 1. The program generator is a basic 2-layer word-level question encoder LSTM [10] and 2-layer word-level (in this case function-level) program decoder LSTM. The encode-decode architecture is standard in machine translation and has been used to great success [21], thus the architecture is a logical choice for the task at hand, which can be viewed as a translation from representation in natural language to functional representation. There is only one deviation from the standard encode-decode architecture: during both training and testing, the decoder receives the encoder output at every timestep instead of the ground truth label. For ease of parallelization over minibatches, each sequence is zero-padded to a uniform fixed length.

Between the program generator and the execution engine is a non-differentiable argmax. This is required in order to select the discrete program functions used in the execution engine. It also impedes end-to-end training and mandates the use of REINFORCE [20]. While this appears suboptimal, in practice almost all error and computation time come from the execution engine. For this reason we do not consider REINFORCE to be a considerable detriment to generalizability, therefore we do not consider improvements upon differentiability in the present work. (See Appendix, section A.1.)

The execution engine is less standard. As the arity (number of arguments) of each function predicted by the program

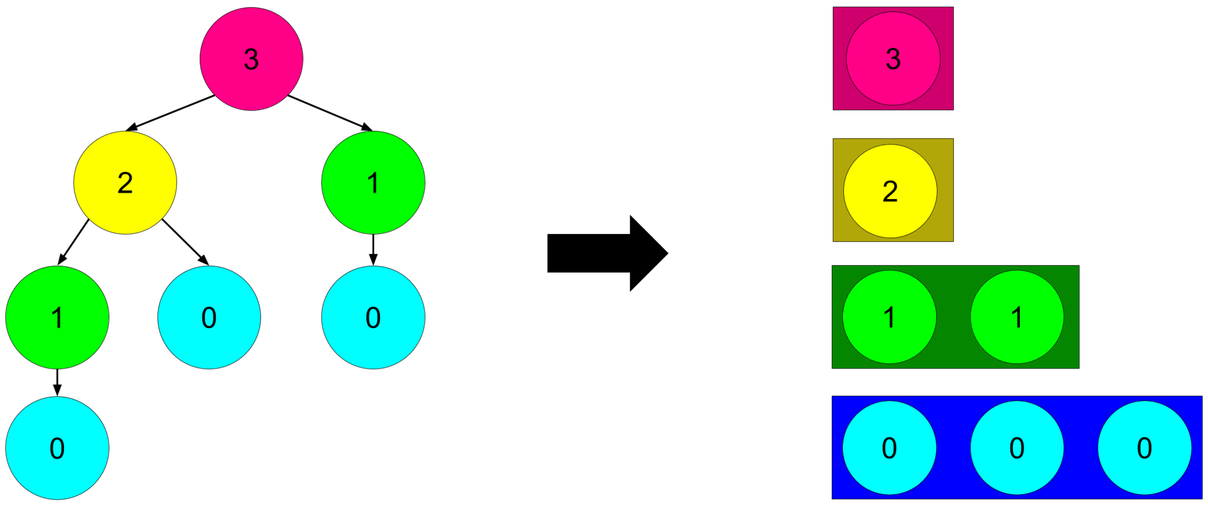

Figure 2. An example of the "coloring" mechanism used for improved topological sort.

generator is predetermined, there exists a unique mapping from the predicted vector of functions to a program tree. This is assembled via a simple depth-first search. Each function is itself a neural network, with the exception of a special *SCENE* token, which simply outputs ResNet-101 features [9] taken from an intermediate layer. This program tree is then directly executed, and the outputs are passed through a small classifier network (one convolutional and two fully connected layers) to yield a softmax distribution over answers, which is then optimized as normal via backpropagation over the cross-entropy loss.

This approach yielded an 8.6X improvement over strong baselines and a 2.2X improvement over human-level performance.

2. Methods

In the original CLEVR result, programs must be executed sequentially with an explicit loop over the examples in each minibatch. As a result, unlike static networks, the computation time of the forward pass scales linearly with the batch size. We present two variants of topological sort that remedy this issue.

To clarify the ongoing notation, programs have max length s and function vocabulary size p. The batch size is denoted by b and the max program tree depth by d.

Standard topological sort. First, consider a naive topological sort. Each program tree is first considered independently. It is sorted via an infix depth-first search. This results in a queue ordering such that each operation can be executed sequentially; no node is executed before all of its dependents. While this operation runs in time linear in the number of nodes (e.g. $O(bs)$), it is fast compared to expensive neural network operations and can be multithreaded extremely efficiently, thus we ignore this factor in our computations. (See Appendix, section A.2.)

We now have a flat representation of each program, which can be viewed as a grid of size $b \times s$. Instead of executing each program independently, we loop only over the columns and execute one full row at a time in parallel. Each node corresponds to a different element in the function vocabulary. However, for $b > p$, we need only make p expensive neural network calls instead of b. This results in $O(ps)$ execution.

Improved topological sort. In the improved variant of topological sort, we take this sorting operation one step further. Instead of flattening programs, we instead color [1] each node by the maximum distance between it and a child leaf (see Fig. 2). Nodes of the same color (across all programs) are assigned to the same pool. Each pool is executed at once in $O(p)$ neural network calls, for a total of $O(pd)$. In the case where program trees are balanced (important in the design of future datasets), this yields $O(p \log_2 s)$ execution. The program trees used in CLEVR are, unfortunately, imbalanced, thus this approach results in only a 10-25 percent improvement in performance. Note that, as d is the *maximum* depth across all programs in a minibatch, $d \gg \log_2 s$.

3. Results

We evaluate performance gains with improved topological sort vs. our implementation of the original architecture. Relevant portions of program construction/execution code are shared appropriately: our experiments are robust to any unintended inefficiency in our implementation of the original architecture.

[1]The "coloring" is purely for ease of visualization and does not refer to coloring in mathematical or graph analytic terms.

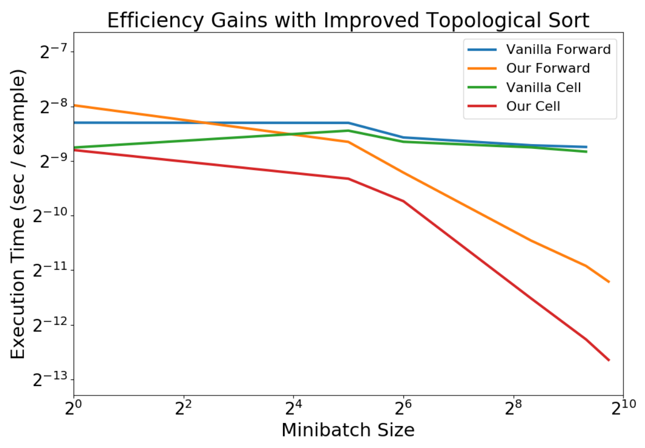

Figure 3. Log-scaled visualization of efficiency gains incurred from our improved topological sort. Vanilla denotes the implementation in Johnson et al. This makes clearer the near-linear gains in speed as the minibatch size approaches 1000.

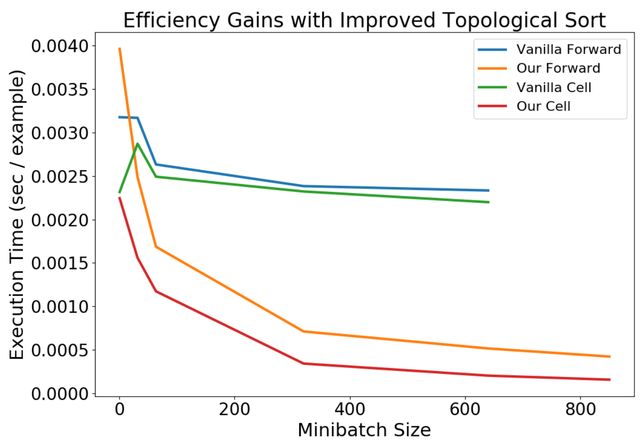

Figure 4. Linear-scaled visualization of efficiency gains. Vanilla denotes the implementation in Johnson et al. Included for standardization.

Using all memory in a single Nvidia GTX 1080Ti, we achieve 5.5X faster inference and a 2X faster backward pass (see Fig. 3, Fig. 4). It is currently unclear why gains do not better transfer to the backwards pass in the PyTorch [1] backend, as the expected gains are symmetric. However, this is not a fair comparison. Our sorting method introduces additional CPU code which accounts for over half of execution time and is not present in the original architecture. Furthermore, this CPU code is embarrassingly parallel and should be written in multithreaded C++ (Python threads do not actually yield performance gains due to the Global Interpreter Lock; we tested this extensively, see Appendix for details).

Thus, a fairer comparison is to measure neural cell execution time, in which case we experience 14X gains. There is a small amount of additional data-stacking code omitted from this computation because it can likely also be optimized; including this, gains are still well over 10X.

More importantly, scaling is linear with batch size. Doubling GPU memory yields 2X performance. Those familiar with minibatch parallelism may object that this performance gain usually drops off after a certain point (minibatch size >1000 in our experience). However, this is not likely to be an issue in our case, as there is an additional factor of the program function vocabulary size (40). With equal distribution of execution over functions, minibatch size 1000 per cell corresponds to overall minibatch of size 40000, which would require approximately 500 GB of GRAM.

Furthermore, it is possible to maintain such gains in the case of multiple GPUs (e.g. large batch size split over many devices) by assigning a different cell function to each GPU, though this would likely require a fair bit of bandwidth optimization.

4. Discussion

The recent independent result of automatic minibatching to DyNet is most similar to our work [17]. However, the DyNet minibatching optimizer relies on lazy execution in order to optimally organize data on the fly. Lazy execution is not present in alternative frameworks such as PyTorch and is often not the most desirable quality for developers to implement. While automatic batching as in DyNet is not possible in PyTorch, variants of our approach are viable. Our standard topological sort and improved topological sort approaches can therefore be viewed as a set of manual batching optimizations applicable to a large class of dynamic architectures without the need for lazy execution.

Our improved topological sort makes the following hard assumptions in order to achieve $O(p \log_2 s)$ where complexity is measured by the number of calls to expensive (e.g. neural network) functions:

1. There exists a set of p expensive modules or functions with known arities, usually neural networks.
2. The architecture, composed of modules, can be executed with batch size b such that $b \gg p$.
3. The architecture is a balanced tree with structure known a priori.

In the case where the third assumption fails because the architecture is known ahead of time but is linear or an imbalanced tree or DAG, our improved topological sort is still applicable, but with complexity $O(pd)$ where d is the maximum dependency path length. This has the same complexity as the standard topological sort but with an equivalent or more favorable constant factor.

In the case where the architecture is not specifically known ahead of time but is a DAG, it is no longer possible to use our improved topological sort. However, as the next module is always known in any architecture, it is still possible to apply our standard topological sort approach and achieve $O(pd)$ by aggregating the computations of all current modules over p. In the case of a generic graph with cycles, this approach is still applicable, but d becomes the maximum length of an unrolled graph, which is always limited in practice to avoid infinite cycles and thus infinite execution time.

5. Conclusion

We implemented the architecture described by Johnson et al. and improved parallelizability over the branches of each program tree and also over the minibatch dimension. We achieve 14X neural cell execution speed and characterize the trend of improvements as batch size varies, which yields increasing returns and becomes linear as batch size approaches 1000. We define the classes of problems for which our improved topological sort is applicable as well as the classes where it is not but standard topological sort is still feasible; in both cases, we provide complexity bounds as a function of neural network calls.

The breadth of architectures in which at least one variant of our sorting approach is applicable implies that a large class of dynamic architectures can be trained and executed as quickly and efficiently as their static counterparts.

5.1. Future Work

What remains is a problem of engineering. For the purposes of the CLEVR result, a small amount of Python multiprocessing (as opposed to multithreading) code would suffice to largely close the gap between the 5.5X forward pass improvement and the 14X cell improvement.

For longevity and for the purposes of supporting arbitrary frameworks, we would need to rewrite our topological sorter and executioner as a C++ multithreaded package callable from higher level frameworks (we focus on PyTorch). This would facilitate application of our work to novel architectures.

Though this is lower priority, for simplicity if not performance or accuracy, it is still desirable to eliminate REINFORCE from the algorithm and to reduce (or even eliminate a procedure for adapting) the need for ground truth programs. Possible approaches vary from adding an intermediate divergence term to encourage the development of distinct neural modules to imposing CycleGan [23] constraints on the network to enable the network to reconstruct the question and answer from the program and the program from the question. Briefly, these improvements comprise:

1. Eliminating the REINFORCE stage between the program generator and the execution engine.
2. Reducing the number of ground-truth programs needed for training the program generator.

A. Appendix

A.1. REINFORCE

REINFORCE is a sample-based algorithm for introducing differentiable *reward* into non-differentiable nodes in a network. This operates by considering the final reward (e.g. whether or not the final network prediction is correct) as the intermediate reward. One can then consider the difference between the reward obtained and an exponentially decaying average of past rewards. This is because it is otherwise ambiguous whether a reward is desirable. For example, getting one example correct in 100 samples incurs some positive reward, but this should still be penalized if, for example, the baseline is correct fifty percent of the time.

A.2. From 5.5X to 14X

There is a clear and seemingly significant discrepancy between the 5.5X speedup obtained in the forward pass verses the 14X speedup obtained in the neural cell execution. We consider the breakdown of operations in the forward pass in order to demonstrate our rationale for largely ignoring this discrepancy. The forward pass consists of extracting ResNet features, compiling all programs, executing all programs, and running the final classifier. In our implementation, ResNet features are extracted as a preprocessing step and are thus computationally free; this point requires a significant digression.

One might argue that ResNet is not ”free” at test time in practical circumstances because the test examples will not be known ahead of time. However, ResNet is already highly optimized and highly parallel. Furthermore, only about half of ResNet is executed before features are extracted, thus further saving computation time. Finally, quantization techniques exist for static networks such as ResNet that yield large gains in performance [8].

Test-time minibatch parallelism is often feasible in real world tasks. We do not address cases where this is not possible, as we are fundamentally interested in parallelizing execution over minibatch. Furthermore, while fast inference is more important than fast training in industry, for the purpose of conducting efficient experimentation, training time is just as essential in research. This is largely the setting and motivation for our work.

Thus, we have illustrated our rationale for optimizing dynamic tree architectures despite the inclusion of a large feature extractor. The classifier in the CLEVR result executes in a trivial fraction of the total computational time. Next to all of the performance loss from 14X to 5.5X comes from CPU code, along with a few unoptimized data stacking operations at neural cell I/O.

The main reason we largely ignore this performance gap is that the slow CPU code is embarrassingly parallel and written in highly unoptimized Python. For example, we found it far more readable and modular to first build the programs in CPU code and then traverse them in CPU *a second time* during execution. Realistically, these operations could be combined for a factor of two. More significantly, pure looping code usually benefits from at least a 10X speedup when transfered to C++. This, coupled with an additional expected 10X speedup from multithreading, ensures that GPU memory will always bottleneck computation before the CPU.

Furthermore, a larger gap is present in the backwards pass. The backwards and forwards pass are symmetric and should therefore observe similar speedups from our optimization. This is, again, a problem of engineering. It is almost certainly the case that the PyTorch backend is not properly preserving our optimizations on the forward pass, mandating the inclusion of custom backwards pass code. As this is extremely cumbersome to write, we were unable to test this properly. However, we were able to confirm that the overhead of threading does not significantly impact performance and is therefore likely to be of use in C++ (e.g. without the Global Interpreter Lock preventing true parallelization in Python).

There is the matter of a small amount of data aggregation and splitting code at the input and output of each neural module. Without significant optimization, this could cause some penalty in performance. However, speedups will still remain linear and over 10X while accounting for this factor.

A.3. Details of Theoretical Bounds

From above, it is clear why we exclude fast CPU operations from our complexity bounds. However, it remains unclear why we count only the number of neural network function calls without regard to the amount of data processed by each call–particularly to those unfamiliar with typical efficiency over minibatches. As CUDA operations are highly parallelized and have high bandwidth, the time taken to execute an optimized operation such as a matrix multiply (e.g. convolutional and fully connected layers) is almost equal for a wide range of minibatch sizes.

Depending on the size of input data and the number of parameters, it is often possible to multiply the batch size by 10 while incurring only a 2X loss in performance, which is effectively equivalent to 5X faster data processing. With tuned learning rates (or simply advanced optimizers that compute momentum updates), this almost always results in a comparable increase in the rate of convergence. As we found in prior work that for most configurations of parallelizable operations, the batchsize/time tradeoff remains extremely favorable up until at least 1000, we thus count only the number of neural network calls.

Our experimental results confirm that this hypothesis holds almost perfectly. As batch size approaches 1000, the gains in cell execution speed per example approach perfect linearity. Note that as our effective batch size is, assuming equal distribution over cell calls, $\frac{1000}{40} = 25$, it is almost certain that we will continue to experience linear gains until batch size become impractically large for single GPU memory constraints. We have already outlined a procedure for efficient parallelization over multiple cards.

Until now, we have presented our standard and improved topological sort approaches as distinct. However, until a dataset is specifically constructed to satisfy the balanced tree requirement, our improved topological sort is effectively a heuristic. It always performs at least as well as standard topological sort: the standard approach always considers exactly one module at a time, whereas the improved variant considers *at least* one cell at a time. However, in practice it is also possible to use the improved variant as a heuristic in cases where the graph structure is not known ahead of time and is arbitrary. At each step in any computational graph (e.g. neural network architecture), the next operation to be executed is known. This is clear because it would be impossible to execute an operation without knowing it.

More generally, *at least* one node is always known. It is possible that many nodes could be known at once without knowing the entire graph structure, for example, in the case where the graph is an inverted tree. In this case, one should adopt the policy of always executing all currently known nodes at once, without breaking dependencies (e.g. as is satisfied by the sort).

One might notice that this is not always optimal. While the above strategy is always equal or superior to standard topological sort, it may be possible to outperform both of them. Consider the case in where there are two programs to be executed and only two cells, 1 and 2. The first tree is, from leaf to root, labeled (1, 2). The second is labeled (2). In this case, it is better to execute (1) in the first tree and then parallelize over both (2)'s. However, our improved topological sort would first execute a (1) cell and a (2) cell simultaneously, then execute the (2) cell in the first tree.

While there are certainly gains to be made by such searches in arbitrary graphs, we do not consider them useful on CLEVR (and no other large datasets, to our knowledge, include generic program annotations) for two reasons. First, such traversals would break our guarantee that the CPU code will not account for a significant fraction of computation time. Second, CLEVR does not contain complex branching; we only observed a 10-25 percent improvement in performance from standard to improved topological sort. Many programs are perfectly linear. We thus consider such additional complications unlikely to yield large speedups.

References

[1] Pytorch. `https://github.com/pytorch/pytorch`. Accessed: 2017-06-11.

[2] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein. Learning to compose neural networks for question answering. *arXiv preprint arXiv:1601.01705*, 2016.

[3] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein. Neural module networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[4] S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. Lawrence Zitnick, and D. Parikh. Vqa: Visual question answering. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[5] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

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

[7] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[8] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

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

[11] R. Hu, J. Andreas, M. Rohrbach, T. Darrell, and K. Saenko. Learning to reason: End-to-end module networks for visual question answering. *CoRR*, abs/1704.05526, 2017.

[12] J. Johnson, B. Hariharan, L. van der Maaten, L. Fei-Fei, C. L. Zitnick, and R. B. Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. *CoRR*, abs/1612.06890, 2016.

[13] J. Johnson, B. Hariharan, L. van der Maaten, J. Hoffman, F. Li, C. L. Zitnick, and R. B. Girshick. Inferring and executing programs for visual reasoning. *CoRR*, abs/1705.03633, 2017.

[14] K. Kurach, M. Andrychowicz, and I. Sutskever. Neural random-access machines. *CoRR*, abs/1511.06392, 2015.

[15] M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[16] M. Malinowski and M. Fritz. A multi-world approach to question answering about real-world scenes based on uncertain input. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1682–1690. Curran Associates, Inc., 2014.

[17] G. Neubig, Y. Goldberg, and C. Dyer. On-the-fly operation batching in dynamic computation graphs. *CoRR*, abs/1705.07860, 2017.

[18] S. Sukhbaatar, a. szlam, J. Weston, and R. Fergus. End-to-end memory networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2440–2448. Curran Associates, Inc., 2015.

[19] J. Weston, S. Chopra, and A. Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

[20] R. J. Williams. On the use of backpropagation in associative reinforcement learning. In *IEEE 1988 International Conference on Neural Networks*, pages 263–270 vol.1, July 1988.

[21] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[22] H. Xu and K. Saenko. Ask, attend and answer: Exploring question-guided spatial attention for visual question answering. In *European Conference on Computer Vision*, pages 451–466. Springer, 2016.

[23] J. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. *CoRR*, abs/1703.10593, 2017.