Cavs: A Vertex-centric Symbolic Programming Model for Dynamic Dataflow Graphs

Shizhen Xu*[1,2], Hao Zhang*[1,3], Graham Neubig[1,3], Qirong Ho[3], Guangwen Yang[2], Eric P. Xing[3]
Carnegie Mellon University[1], Tsinghua University[2], Petuum Inc.[3]

1. Introduction

Dataflow graphs have been successfully adopted in many deep learning (DL) frameworks [1, 3, 2], as they help separate model declaration from execution, and present obvious efficiency advantages. However, its applicability highly relies on a key assumption: the dataflow graph is static and fixed throughout the runtime. This condition breaks for *dynamic models* where the dataflow graph changes over training, such as neural networks (NNs) computed over graph structures [11, 12, 6]. To better support these dynamic models, some recent frameworks [4, 1, 8] propose to declare a graph per sample (a.k.a. *dynamic declaration*). While dynamic declaration is convenient to developers as it requires little change on the implementation of the native (static) framework, it exhibits a few limitations: (1) programmers have to write code to repeatedly assemble new dataflow graphs according to the input structures; (2) the overhead of graph construction grows linearly with the number of instances – as we have found, it sometimes takes longer time than the computation in some frameworks [7] (Section 3); (3) Only single-instance execution can be performed for dynamically declared graphs (instead of the batched one), in which case, the computation is very inefficient because it does not sufficiently exploit the batching advantage of modern computational hardware, while how to batch the computational operations from different graphs is not obvious.

To address these problems, we present Cavs, a new programming model for dynamic neural networks, and a system implementation with a few optimization strategies derived from this model. Cavs combines the best of symbolic construction of dataflow graphs for DL with the vertex-centric model [5] in graph computing. Instead of declaring a graph per sample, it represents a dynamic dataflow graph as two components: one static vertex function that is only declared (by the user) and optimized once, and dependency edges that are instance-specific and not visible until runtime. With this model, users will think like a vertex (complex models can be represented by multiple vertex functions) and define the vertex function symbolically while the system will schedule the function execution according to the dependency edges – it therefore simplifies user programs, and avoids the overhead of repeated dataflow graph construction. Moreover, this vertex-centric model exposes more opportunities of batching same operations during the evaluation of a batch of samples with different graphs (as we will show that the strategies in [9, 7] are only special cases of the dynamic batching strategy in Cavs). Existing optimizations such as kernel fusion from static frameworks, and memory coalescing for efficient batch evaluation, can also be easily incorporated.

We compare Cavs against Tensorflow-Fold [7], a state-of-the-art system for dynamic models, and show that Cavs is 11-23x faster than TensorFlow-Fold, and 1.6-21x faster if only considering computation.

* indicates equal contributions

[Copyright notice will appear here once 'preprint' option is removed.]

2. Cavs Design

Algorithm 1 presents the overall workflow of training a dynamic network on Cavs. We next briefly describe its four major modules: the programming interface, the vertex scheduler, the memory management and the graph optimization engine.

Programming Interface. Cavs is type of symbolic programming interface and provides conventional operators for assembling neural networks (e.g. placeholder, arithmetical, blas, etc.). Beyond this, Cavs also proposes four novel primitives for dynamic networks (but note the programing model is also compatible with non-symbolic DL frameworks [4]):

- `gather(idx)` accepts a list of indices of child nodes, and returns a list of symbols that represent their output.
- `scatter(op)` takes as input an existing symbol `op`, and scatters the output of the node `op` – if any other node tries to `gather` it.

While `gather` and `scatter` resemble the GAS model in graph computing [5], their semantics are insufficient for

Algorithm 1 Training Dynamic Networks using Cavs.

```
1: Users construct a node function $\mathcal{F}$ using symbolic operators.
2: for $t = 1 \to T$ do
3:     Get the $t$th batch of data and their graphs as $\{x_i^t, \mathcal{G}_i^t\}_{i=1}^K$
4:     Cavs schedules the execution of $\mathcal{F}$ according to the dependency edges specified by $\{\mathcal{G}_i^t\}_{i=1}^K$, with inputs $\{x_i^t\}_{i=1}^K$.
5: end for
```

Figure 1: An example user code for Sequence LSTM [11, 9].

```
def VertexFunction():
  s = gather({0})            # gather child states
  c, h = split(s)
  x = pull()                 # pull the input
  i = sigmoid(W^(i)× x + U^(i)× h + b^(i))
  f = sigmoid(W^(f)× x + U^(f)× h + b^(f))
  o = sigmoid(W^(o)× x + U^(o)× h + b^(o))
  u = tanh(W^(u)× x + U^(u)× h + b^(u))
  c = i ⊗ u + f ⊗ c
  h = o ⊗ tanh(c)
  scatter(concat([c, h], 1)) # scatter the states
  push(h)                    # push the output
```

expressing dynamic dataflow graphs, as a node in a dynamic NN interacts with not only other nodes within the graph, but also the external of the graph (e.g. a RNN receives input from external I/O or another NN). Therefore, Cavs provides two additional APIs:

- `pull()` is distinguished from `gather` in that it requests input from the external of the current graph.
- `push(op)` sets the output of the current node as `op` and passes it to its outgoing nodes that are external of the current dynamic graph.

Backpropagating through these operators is straightforward: the "gradient" operator of `gather` is exactly `scatter`, while the "gradient" of `pull` is `push`. Hence, for auto-differentiation, given an operator we simply generate its corresponding backward operator in the backward graph.

Vertex scheduling. Once users construct the vertex function and invoke the evaluation, the vertex scheduler will first assign each vertex with a unique job ID. At each step of the forward, it figures out a set of activated IDs according to the dependency edges (a vertex is *activated* if and only if the jobs of all it dependents have been done), and applies the user function over them while recording the execution order for future backward. With this design, dynamic batching [7, 9] is easy to incorporate: the scheduler will batch all groups of same computational operations incurred by the evaluation of the current set of activated vertices.

Memory management. Cavs allocates a large memory pool first, and maps a chunk of memory from the pool with each vertex to be evaluated based on their IDs. Cavs introduces *dynamic tensor* to support the aforementioned dynamic scheduling policy. The shape of a dynamic tensor is a multi-dimensional array where the first entry is the batch size, which is dynamically-varying, and the other dimensions are fixed and can be inferred from the user program. During computation, the scheduler will set the first dimension according to how many operations will be batched. As the intermediate output by forward pass need to be stored and used during backward, the dynamic tensor can be flexibly indexed from the pool, depending on the ID of their corresponding vertex. As sometimes the operations to be batched are not collocated on memory, for efficient batching (esp. on GPU), the evaluation of the Gather/Scatter operator will index a branch of (discontinuous) memory blocks based on IDs, and copy them into a dynamic tensor so that they are located continuously in memory.

Graph optimization. Cavs benefits from the symbolic design and is compatible with dataflow graph-based optimization for improved parallelism. Specifically, we note there are two levels of dependencies: the vertex evaluation needs to follow the dependency specified by the input graph, which is guaranteed by the vertex scheduler; the operations within one vertex function are subject to the dependency specified in the user program. The graph optimization thus respects the second dependency, and parallelizes any independent operations that are not subject to it for improved performance.

3. Evaluation

We compare Cavs to Fold [7] and DyNet [8] on training the `Tree-LSTM` [11] on SST [10] and `Tree-FC` [7] on synthetic trees, and report the results in Table 1. Experiments are conducted using an Nvidia TitanX (GM200) GPU. We reveal the following major findings: (1) The Graph construction overhead is significant, and sometimes takes more time than computation in Fold [7], which, however, is bypassed by Cavs' design . (2) Cavs is maximally 27x faster than Fold on `Tree-LSTM` and 8.6x faster than DyNet on `Tree-FC`.

Batch Size	Computation (s) (*Cavs/Fold/DyNet*)	Graph Construction (s) (*Cavs/Fold/Cavs*)	Speedup (vs. *Fold/DyNet*)
1	76.2 / 550 / 61.6	0.012 / 1.1 / 1.93	7.2x / 0.83x
16	9.80 / 69.0 / 12.0	0.090 / 7.14 / 1.73	7.7x / 1.4x
32	6.15 / 43.0 / 9.85	0.094 / 31.5 / 1.71	12x / 1.9x
64	4.05 / 29.0 / 7.36	0.095 / 40.1 / 1.71	17x / 2.2x
128	2.90 / 20.5 / 5.94	0.095 / 46.1 / 1.71	22x / 2.6x
256	2.27 / 15.8 / 5.44	0.096 / 48.8 / 1.86	27x / 3.1x
Tree Size	**Computation (s) (*Cavs/Fold/DyNet*)**	**Graph Construction (s) (*Cavs/Fold/Cavs*)**	**Speedup (vs. *Fold/DyNet*)**
32	0.574 / 3.10 / 4.10	0.120 / 3.90 / 0.384	10x / 6.5x
64	1.06 / 3.93 / 7.97	0.241 / 7.19 / 0.896	8.5x / 6.7x
128	2.02 / 6.15 / 16.0	0.481 / 14.1 / 1.79	8.1x / 7.1x
256	3.89 / 10.6 / 33.7	0.960 / 32.4 / 3.71	8.9x / 7.7x
512	7.96 / 18.5 / 70.6	1.92 / 46.8 / 7.68	6.6x / 7.9x
1024	15.8 / 32.4 / 153	3.84 / 106 / 15.1	7.0x / 8.6x

Table 1: The average computation and graph construction time (wall clock time in seconds per epoch) of training `Tree-LSTM` [7] with varying batch size, and `Tree-FC` [7] with varying tree size. We compare Cavs to Fold (one CPU thread for graph construction) and DyNet, and report Cavs' overall speedups over them in the last column.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. *arXiv preprint arXiv:1605.08695* (2016).

[2] Bergstra, J., Bastien, F., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., Desjardins, G., Warde-Farley, D., Goodfellow, I. J., Bergeron, A., and Bengio, Y. Theano: Deep learning on gpus with python. In *NIPSW* (2011).

[3] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[4] Facebook. Pytorch. `http://pytorch.org/`.

[5] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs.

[6] Liang, X., Hu, Z., Zhang, H., Gan, C., and Xing, E. P. Recurrent topic-transition gan for visual paragraph generation. *arXiv preprint arXiv:1703.07022* (2017).

[7] Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181* (2017).

[8] Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980* (2017).

[9] Neubig, G., Goldberg, Y., and Dyer, C. On-the-fly operation batching in dynamic computation graphs. *arXiv preprint arXiv:1705.07860* (2017).

[10] Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing* (2013), pp. 1631–1642.

[11] Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[12] Yan, Z., Zhang, H., Piramuthu, R., Jagadeesh, V., DeCoste, D., Di, W., and Yu, Y. Hd-cnn: hierarchical deep convolutional neural networks for large scale visual recognition. In *Proceedings of the IEEE International Conference on Computer Vision* (2015), pp. 2740–2748.