Cavs: A Vertex-centric Programming Interface for Dynamic Neural Networks

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

Abstract

Recent deep learning (DL) models have moved beyond static network architectures to dynamic ones, handling data where the network structure changes every example, such as sequences of variable lengths, trees, and graphs. Existing dataflow-based programming models for DL—both static and dynamic declaration—either cannot readily express these dynamic models, or are inefficient due to repeated dataflow graph construction and processing, and difficulties in batched execution. We present Cavs, a vertex-centric programming interface and optimized system implementation for dynamic DL models. Cavs represents dynamic network structure as a static vertex function $\mathcal{F}$ and a dynamic instance-specific graph $\mathcal{G}$, and performs backpropagation by scheduling the execution of $\mathcal{F}$ following the dependencies in $\mathcal{G}$. Cavs bypasses expensive graph construction and preprocessing overhead, allows for the use of static graph optimization techniques on pre-defined operations in $\mathcal{F}$, and naturally exposes batched execution opportunities over different graphs. Experiments comparing Cavs to two state-of-the-art frameworks for dynamic NNs (TensorFlow Fold and DyNet) demonstrate the efficacy of this approach: Cavs achieves a near one order of magnitude speedup on training of various dynamic NN architectures, and ablations demonstrate the contribution of our proposed batching and memory management strategies.

1 Introduction

Deep learning (DL), which refers to a class of neural networks (NNs) with deep architectures, is now a workhorse powering state-of-the-art results on a wide spectrum of tasks [48, 49, 27]. One reason for its widespread adoption is the variety and quality of software toolkits, such as Caffe [22], TensorFlow [1] and DyNet [29, 30], which ease programming of DL models, and speed computation by harnessing modern computing hardware (e.g. GPUs), software libraries (e.g. CUDA, cuDNN [7]), and compute clusters [51, 52, 8]. One dominant paradigm in the training of DL models, adopted by toolkits such as Caffe and TensorFlow, uses static dataflow graphs [1, 28]. These graphs represent the flow of data through computational functions, and are defined using symbolic programming [4, 1], once before beginning training or testing of the model. The training of these models is performed through auto-differentiation, in which users are only required to assemble their model architectures by connecting operators using high-level language interface (e.g. Python), after which the framework will automatically derive the correct algorithm for training [3]. With proper optimization, the execution of these static dataflow graphs can be highly efficient. Specifically, by separating model declaration and execution, it makes it possible for the graph to be further processed and optimized before runtime [1]. In addition, the evaluation of multiple data samples in a dataflow graph can be naturally batched to leverage the improved computational capability of modern hardware (e.g. GPUs), which is extremely advantageous for DL workloads [23].

* indicates equal contributions. 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

While these static dataflow graphs have major efficiency advantages, their applicability highly relies on a key assumption – the dataflow graph (i.e. NN architecture) fixed throughout the runtime. With the increasing complexity of the problems to be addressed, DL has been extended and applied on data with more complicated structures, such as sequences [20, 37], trees [38] and graphs [24], over which the NN may conditionally choose its own computation order for specific modeling needs, i.e. the structure of its dataflow graph changes over training. To better support these dynamic models, some recent frameworks [43, 29] propose to declare a dataflow graph per sample (a.k.a. *dynamic declaration*). While dynamic declaration is convenient to developers as code can basically be written in the same way as it usually is in the native programming language (e.g. Python, C++), it exhibits a few limitations. First, programmers still have to write code to explicitly assemble the dataflow graph for each input sample, which might be nontrivial for graphs with sophisticated structures. Second, as the graph construction needs to be performed repeatedly, its overhead grows linearly with the number of training instances, preventing the application of complex static graph optimization techniques (in fact, graph construction takes longer time than the computation in some frameworks [25]). Finally, since each sample owns a dataflow graph specifying its unique computational pattern, batching together similarly shaped computations across instances is non-trivial. Without batching operations, the computation is inefficient due to its lack of ability to exploit modern computational hardware, and while some progress has been made in recent research [30, 25], how to automatically batch the computational operations from different graphs remains a difficult problem.

To address these challenges, we present Cavs, a new programming interface for dynamic NNs, and a system implementation with optimization strategies tailored to it. Cavs leverages the recurrent and recursive nature of dynamic NNs. Instead of declaring a dataflow graph per sample, it alternatively decomposes a dynamic dataflow graph as two components: one static vertex function $\mathcal{F}$ that is only declared (by the user) and optimized once, and an input graph $\mathcal{G}$ that is instance-specific and not used until runtime. Thereby, the workflow of training a dynamic NN can be represented as scheduling the execution of $\mathcal{F}$ following the structure of the input graph $\mathcal{G}$. Cavs combines the best of symbolic construction of dataflow graphs for DL [1, 4] with the vertex-centric model [14] in graph computing: it only requires users to define $\mathcal{F}$ symbolically by "thinking locally like a vertex" [42]. Cavs will perform auto-differentiation, schedule the function execution following the dependency reflected by $\mathcal{G}$, and guarantee efficiency and correctness. It also inherits the flexibility of symbolic programming, i.e. users are allowed to declare multiple vertex functions to express more dynamics, or connect static dataflow graphs with dynamic ones to construct more complex NN architectures. Cavs demonstrates a few advantages over other programming models. It simplifies user programs and avoids the overhead of repeated dataflow graph construction. Moreover, this vertex-centric model naturally exposes opportunities for batched computation. Compared to dynamic declaration, as the dataflow graph encoded by the vertex function is static throughout the runtime, it can benefit from various static graph optimizations [1, 6, 13, 15], such as lazy batching, streaming, and kernel fusion, which would otherwise be less effective on the scenario of dynamic declaration because of the repeated preprocessing/optimization cost (see §2).

We implement Cavs as an additional layer pluggable to most existing DL frameworks to enhance their support for dynamic NNs. To evaluate its performance, we compare Cavs to TensorFlow Fold [25] and DyNet [29, 30], two state-of-the-art systems supporting dynamic NNs and dynamic batching. We focus our experiments on GPU training, and verify that both Fold and DyNet suffer from substantial overhead caused by repeated graph preprocessing or construction, which is bypassed by Cavs (§**??**). In terms of overall performance, on static NNs, Cavs demonstrates equivalent or slightly better performance than Fold and DyNet, while on several dynamic NNs with notably difficult-to-batch workloads (e.g. Tree-LSTM [38] and Tree-FC [25]), Cavs demonstrates near one order of magnitude speedups across various dataset and hyper-parameter settings (§4).

Model	Frameworks	Expressiveness	Batching	Graph Cons. Overhead	Graph Exec. Optimization
static declaration	Caffe, Theano, TensorFlow, MxNet	×	×	low	beneficial
dynamic declaration (instant evaluation)	PyTorch, Chainer	√	×	N/A	unavailable
dynamic declaration (lazy evaluation)	DyNet	√	√	high	not beneficial
Fold	TensorFlow-Fold	√	√	high	unknown
Vertex-centric	Cavs	√	√	low	beneficial

Table 1: The landscape of existing programming models for dynamic NNs.

2 Related Work

Table 1 gives an overview of the landscape Most existing DL frameworks, including Caffe [22], Theano [4], Tensorflow [1], MxNet [6], adopt the static declaration model in which a user declares the network architecture symbolically before the computation. When dealing with NNs with fixed structures (e.g. CNNs), they have been prove quite successful in both programmability and efficiency. However, to express dynamic NNs, a user has to declare one dataflow graph per input sample (i.e. dynamic declaration), which might not be flexible, and sometimes causes substantial graph construction overhead. In terms of performance, single-instance training is usually performed in this case, as it is not obvious to both users and developers how the computation of multiple dataflow graphs with different structures can be batched. Tensorflow Fold [25] and DyNet [30] go one step further and perform auto-batching for users. Fold proposes a by-depth batching strategy to batch same operations at the same depth of multiple (different) graphs, along with some functional programming-like APIs based on TensorFlow's control flow APIs. DyNet, on the other hand, tries to minimize its graph construction overhead, and implements both the by-depth and by-agenda batching strategies to seek for more batching opportunities during the evaluation of multiple dataflow graphs. As shown in our experiments, they are less effective than Cavs.We also note there are some "imperative" frameworks, such as PyTorch [12] and Chainer [44] that allow users to construct dynamic NNs. However, as model construction and execution are coupled, it is usually difficult to perform dynamic batching. Overall, they are still far from efficient when handling dynamic NNs.

3 Cavs Design

Our motivation comes from several key principles ML developers usually comply with to ensure the feasibility and learnability of the model during their design of dynamic NNs. We note most dynamic NNs are designed to exhibit a recursive structure (e.g. sequence RNN, Tree-RNN), or a combination of static and recursive structures (e.g. LRCN [10, 2], attention [47]), or even a combination of different recursive structures (e.g. encoder-decoder RNNs [37]). Within one such structure, a function is dynamically applied over instance-specific graphs, and every vertex of the graph usually interacts in a same way with it neighboring vertices following the function. The computational function itself, however, is usually static and parameterized by fixed learnable parameters.

This observation motivates us to design a new programming model, called Cavs, that combines the best of dataflow graphs with the vertex-centric model in graph computing. For clarity, we will use the following terminology and notation in the rest of the paper: we call the instance-specific structure associated with the input sample as an *input graph*, and notate it as $\mathcal{G}$, and a node in that graph as a *vertex*, to be distinguished from a dataflow graph $\mathcal{D}$ and the nodes (which are usually operators or variables) therein. Figure 1 illustrates the concept of this vertex-centric programming model. To describe an aforementioned dynamic structure, different from dynamic declaration, which requires users to manually declare dataflow graphs for each sample according to its associated graph, Cavs instead directly takes it as an input argument. To be aware of what computation shall be performed, Cavs requires users to implement a simple vertex function $\mathcal{F}$ by "thinking like a vertex", informing the framework how one vertex in a dynamic NN will interact with its connected vertices (if these is any). In $\mathcal{F}$, users can utilize conventional DL operators to assemble a symbolic construct that will be evaluated dynamically following the structure of $\mathcal{G}$, while Cavs will ensure the correctness and efficiency. Therefore, a vertex function $\mathcal{F}$, together with an input graph $\mathcal{G}$, implicitly encodes a recurrent dataflow graph, which maps to a subgraph of the implicit full dataflow graph of the model that may needs to be explicitly declared in traditional programming models. For convenience of notations, we will call any part of the structure that cannot be encoded by $\mathcal{F}$ and $\mathcal{G}$ as *external to* $(\mathcal{F}, \mathcal{G})$, and vice versa. Cavs allows users to connect any external static dataflow graph to a dynamic structure encoded by $(\mathcal{F}, \mathcal{G})$ to express various model architectures (e.g. connecting a CNN to an RNN), or declare multiple vertex functions for different structures, and connect them appropriately to express more complex models (e.g. an encoder-decoder LSTM network).

While it is still necessary to create an I/O function to read input graphs for each sample, this must be done in any models, and only once before training commences, which means that it can be shared across epochs or even training runs. Cavs no longer requires users to construct the full dataflow graphs for each sample by themselves. As repeated graph construction is bypassed, the overhead will also be avoided. With this vertex-centric model, Cavs transforms the problem of evaluating multiple dataflow graphs with different structures [25, 30] into a simpler form – scheduling the execution of the vertex functions following the input graphs. For the later problem, we can easily batch the execution of $\mathcal{F}$ over multiple vertices at runtime, leveraging the batching computational capability of

modern hardware. Moreover, as the vertex function itself maps to a static symbolic dataflow graph, it is open and can benefit from various graph optimization techniques originally developed for static declaration, such as kernel fusion, streaming, and our proposed lazy batching, which might not be effective in the scenario of dynamic declaration. We next describe Cavs' APIs.

3.1 Programming Interface

Besides the generic math operators used to declare the computation, Cavs exposes four symbolic APIs for users to specify how the messages shall be passed between vertices in their vertex functions: `gather`, `scatter`, `pull`, `push`.

- `gather(child_idx)`: `gather` accepts an index of the child vertices, gets the child content from gather/scatter buffer and returns a list of symbols that represent the output of these vertices.
- `scatter(op)`: `scatter` is a reverse API of `gather`, and has a symbol `op` as its input argument. Scatter will set the output of current vertex to gather/scatter buffer.

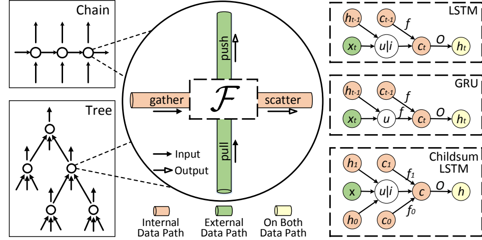

Figure 1: Cavs represents a dynamic structure as a dynamic input graph $\mathcal{G}$ (left) and a static vertex function $\mathcal{F}$ (right).

`gather` and `scatter` resemble the GAS model in graph computing [14] – both are vertex-centric APIs that help users express the overall computational patterns by thinking locally like a vertex: `gather` receives messages from dependent vertices, while `scatter` updates information to parent vertices. But note several key differences: (1) `gather` and `scatter` here are fully symbolic – `gather` allows backpropagation through it; (2) In graph computing, all nodes interact with their connected nodes in the same way following a user-specified `apply` function, while in dynamic NNs, a vertex usually interacts differently with its different child vertices, specified by the symbolic programs (between the call of `gather` and `scatter`) in the vertex function; (3) In graph computing, a vertex of a graph always interacts with other vertices of this graph, while in DL, the vertex of a dynamic NN usually takes input from not only the internal of the structure expressed by $\mathcal{F}$ and $\mathcal{G}$ (internal data path in Figure 1), but also from the external of it, e.g. a step in an RNN can take inputs from a CNN feature extractor or some external I/O (external data path in Figure 1). In this case, `gather` and `scatter` are insufficient to express such semantics. Cavs therefore provides another two APIs:

- `pull()`: `pull` grabs inputs from the external of the current dynamic structure, e.g. another NN, or some I/O.
- `push(op)`: `push` is thus the reverse of `pull` that sets the output of the current vertex as `op`. If this vertex is `pull`ed by others, the content of `op` will be returned.

With appropriate indexing, `push` and `pull` connect a vertex inside a dynamic structure expressed by $(\mathcal{F}, \mathcal{G})$ to other connectors external to $(\mathcal{F}, \mathcal{G})$, such as another dynamic structure, or another static dataflow graph.

Expressiveness. With these four APIs, Cavs can be seen as a middle ground between static and dynamic declaration: In the best case, the model can be easily represented by a single vertex function plus input graphs. While in the worse case scenario, that every sample has a unique input graph while every vertex in the graph has a unique way to interact with its neighboring vertices, Cavs reduces to dynamic declaration that one has to define a vertex function for each vertex of input graphs. However, dynamic NNs in this scenario are very rare and usually not preferred because of the difficulty of design, programming and learning.

Auto-differentiation. Cavs by nature supports auto-differentiation. Given a vertex function $\mathcal{F}$ it derives $\partial\mathcal{F}$ following the auto-differentiation rules: for each math expression such as $s_l = \texttt{op}(s_r)$ in $\mathcal{F}$, Cavs generates a corresponded backward expression in $\partial\mathcal{F}$ as $\nabla s_r = \texttt{grad_op}(\nabla s_l, s_l, s_r)$. For the four proposed operators, with the memory management strategy described above, we note `scatter` is the backward operator of `gather` in the sense that if `gather` collects inputs from `gatherBuffer` previously written by `scatter` at the forward pass, a `scatter` needs to be performed to write the gradients to the `gatherBuffer` for its dependent vertices to `gather` at the backward pass. Hence, for an expression like $s_l = \texttt{gather}(\texttt{child_idx})$ in $\mathcal{F}$, Cavs will generate a backward expression $\texttt{scatter}(\nabla s_l)$ in $\partial\mathcal{F}$. Similarly, the gradient operator of `scatter` is `gather`. The same auto-differentiation rule applies for `push` and `pull` as well.

Once users define the vertex function $\mathcal{F}$ and launch the execution, the Cavs scheduler arranges the evaluation of $\mathcal{F}$ over the input graphs to perform the backpropagation.

Backpropagation. Cavs performs backpropagation [19] as follows. For a sample x_i with its input graph $\mathcal{G}_i$, the scheduler starts the forward pass from the input vertices of $\mathcal{G}_i$, and proceeds following the direction indicated by the edges in $\mathcal{G}_i$: at each sub-step, the scheduler figures out the next activated vertex in $\mathcal{G}_i$, and evaluates $\mathcal{F}$ at this vertex following the symbolic programs in $\mathcal{F}$. It then marks this vertex as *evaluated*, and proceeds with the next activated vertex until reaching a terminal vertex (e.g. the loss function). A vertex of $\mathcal{G}$ is activated if and only if all its dependent vertices have been evaluated. The backward pass is continued right after the forward. The scheduler first resets the status of all vertices as *not evaluated*, then scans the graph in a reverse direction, starting from the ending point of the forward pass. It similarly figures out the next activated vertex, but applies another function $\partial\mathcal{F}$, which is the backward function of $\mathcal{F}$ and automatically derived by Cavs via auto-differentiation, until all vertices have been evaluated in backward. To train a NN to convergence, the above process has to be iterated by the scheduler over all samples $\{x_i\}_{i=1}^N$ and their associated graphs $\{\mathcal{G}_i\}_{i=1}^N$, for many epochs. Instead a sequential execution, Cavs designs a batching policy to perform batched computation, considering the fact that evaluating a set of same arithmetic operations together is significantly faster than the sequential evaluation of each of them.

4 Evaluation

Environment. We perform all experiments in this paper on a single machine with an NVIDIA Titan X (GM200) GPU, a 16-core (32 threads) CPU, and CUDA toolkit 8.0 and cuDNN v6 installed. As modern DL models are mostly trained using GPUs, we focus our evaluation on GPUs, but note Cavs' design and implementation do not reply on a specific type of device. We borrow the implementations of most mathematical operators from TensorFlow v1.2, while we implement the four proposed operators and other system modules by ourselves. We mainly compare Cavs to TensorFlow v1.2 [1] with XLA [15] and its variant Fold [25], as well as DyNet v2.0 [29] with autobatching [30], as they have reported better performance than other frameworks [12, 44] on dynamic NNs. We focus on metrics for system performance, e.g. the average time to scan one epoch of data. Cavs produces exactly the same numerical results with other frameworks, hence the same per-epoch convergence.

Models and dataset. We experiment on the following models with increasing difficulty to batch: (a) `Fixed-LSTM` language model (LM): a static sequence LSTM with fixed steps for language modeling [36, 37, 50]. We train it using the PTB dataset [41] that contains over 10K different words. We set the number of steps as 64, i.e. at each iteration of training, the model takes a 64-word sentence from the training corpus, and predicts the next word of each word therein. Obviously, the computation can be by nature batched easily, as each sentence has exactly the same size. (b) `Var-LSTM` LM: that accepts variable-length inputs. At each iteration the model takes a batch of natural sentences with different length from PTB, and predicts the next words; (c) `Tree-FC`: the benchmarking model used in [25] with a single fully-connected layer as its cell function. Following the same setting in [25], we train it over synthetic samples generated by their code [40] – each sample is associated with a complete binary tree with 256 leaves (therefore 511 vertices per graph); (d) `Tree-LSTM`: a family of dynamic NNs widely adopted for text analysis [24, 45]. We implement the binary child-sum Tree-LSTM model in [38], and train it as a sentiment classifier using Stanford sentiment treebank (SST) dataset [34], which contains 8544 training sentences in which the longest sentence has 54 words. Each sentence is associated with a human annotated grammar tree.

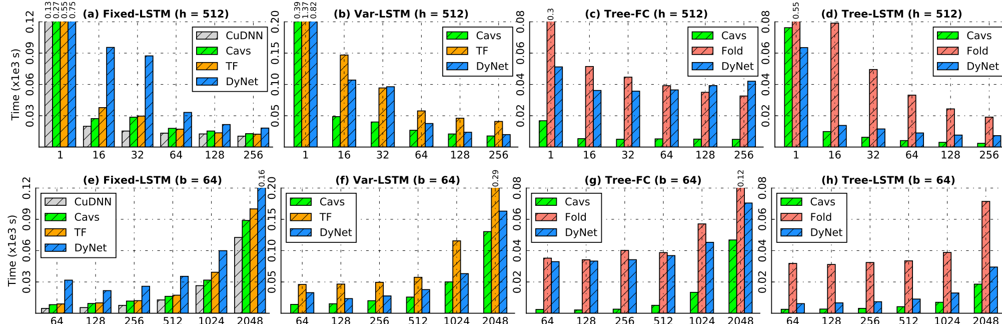

Figure 2: Comparing five systems in terms of the averaged time to finish one epoch of training (lower is better) on four models: `Fixed-LSTM`, `Var-LSTM`, `Tree-FC` and `Tree-LSTM`. In (a)-(d) we fix the hidden size h and vary the batch size bs, while in (e)-(h) we fix bs and vary h.

We first verify the viability of our design on the easiest-to-batch case: `Fixed-LSTM` language model. We compare Cavs to the following three strong baselines: (1) `CuDNN` [7]: a CuDNN-based fixed-step sequence LSTM, which is highly optimized by NVIDIA using handcrafted kernels and stands as the best performed implementation on NVIDIA GPUs; (2) `TF`: the official implementation of `Fixed-LSTM` LM in TensorFlow repository [39] based on static declaration; (3) `DyNet`: we implement a 64-step LSTM in DyNet based on dynamic declaration – we declare a dataflow graph per sample, and train with the autobatching [30] enabled; (4) `Cavs` with batching policy, and all input samples have a same input graph – a 64-node chain. We train the model to converge, and report the average time per epoch in Figure 2(a)(e), where in (a) we fix the hidden size h of the LSTM unit as 512 and vary the batch size bs, and in (e) we fix $bs = 64$ and vary h. Empirically, CuDNN performs best in all cases, but note it is highly inflexible. Cavs performs slightly better than `TF` in various settings, verifying that our system has little overhead dealing with fully static graphs, though it is specialized for dynamic ones. We also conclude from Figure 2 that batching is essential for GPU-based DL: $bs = 128$ is nearly one order of magnitude faster than $bs = 1$ regardless of used frameworks. For Cavs, the batching policy is 1.7x, 3.8x, 7.0x, 12x, 15x, 25x, 36x faster than the serial policy at $bs = 2, 4, 8, 16, 32, 64, 128$, respectively. Next, we experiment with `Var-LSTM`, the most commonly used RNN for variable-length sequences. We compare the following three implementations (CuDNN-based LSTM cannot handle variable-length inputs): (1) `TF`: an official TensorFlow implementation based on the dynamic unroll approach; (2) `DyNet`: an official implementation from DyNet benchmark repository based on dynamic declaration [11]; (3) `Cavs`: where each input sentence is associated with a chain graph that has number of vertices equal to the number of words. We vary h and bs, and report the results in Figure 2(b)(f), respectively. Although all three systems perform batched computation in different ways, `Cavs` is constantly 2-3 times faster than `TF`, and outperforms `DyNet` by a large margin. Compared to `TF`, `Cavs` saves computational resources. `TF` dynamically unrolls the LSTM unit according to the longest sentence in the current batch, but it cannot prevent unnecessary computation for those sentences that are shorter than the longest one.

We then turn to `Tree-FC`, a dynamic model for benchmarking. Since vanilla TensorFlow is unable to batch its computation, we compare `Cavs` to (1) `DyNet` and (2) `Fold`, a specialized library built upon TensorFlow for dynamic NNs, with a depth-based dynamic batching strategy. To enable the batching, it however needs to preprocess the input graphs, translate them into intermediate representations and pass them to lower-level TensorFlow control flow engine for execution. We report the results in Figure 2(c)(g) with varying bs and h, respectively. For all systems, we allocate a single CPU thread for graph preprocessing or construction. `Cavs` shows at least an order of magnitude speedups than `Fold` and `DyNet` at ($h \leq 512$). Because the size of the synthetic trees is large, one major advantage of `Cavs` over them is the alleviation of substantial graph preprocessing/construction overhead. With a single CPU thread, `Fold` takes even more time on graph preprocessing than computation.

Finally, we compare three frameworks on `Tree-LSTM` in Figure 2(d)(h): `Cavs` is 8-10x faster than `Fold`, and consistently outperforms `DyNet`. One difference in this experiment is that we allocate as many CPU threads as possible (32 on our machine) to accelerate graph preprocessing for `Fold`, otherwise it will take much longer time. Further, we note `DyNet` performs much better here than on `Tree-FC`, as the size of the input graphs in SST (maximally 52 leaves) is much smaller than the synthetic ones (256 leaves each) in `Tree-FC` experiments. We observe `DyNet` needs more time on graph construction for large input graphs, and `DyNet`'s dynamic batching is less effective on larger input graphs, as it has to perform frequent memory checks to support its dynamic batching.

Others. Despite system advantages, we also try to investigate whether Cavs, as an interface, simplifies user programs (though we do not claim as a contribution). We compare Cavs to Fold and DyNet in terms of the lines of code (LoC) needed to create a few notable dynamic NNs, including `Var-LSTM`, `Tree-LSTM`, and multi-layer sequence LSTM, with Python as the host language. If only for model declaration, Fold in general has 3.5x more LoC than Cavs, while DyNet has slightly more LoC than Cavs because of the function to repeatedly declare graphs.

5 Conclusion

We present Cavs as a vertex-centric programming interface as well as an efficient system for dynamic deep learning. Cavs represents a dynamic NN structure as static vertex functions and dynamic input graphs. It provides four novel APIs to allow users to easily program these types of NNs. With designed scheduling policy, memory management strategy, and graph execution optimizations, Cavs avoids substantial graph construction overhead suffered by dynamic declaration, and reports new state-of-the-art system performance for various notable dynamic NN architectures.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. *arXiv preprint arXiv:1605.08695*, 2016.

[2] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 39–48, 2016.

[3] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1):171–190, 2000.

[4] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian J. Goodfellow, Arnaud Bergeron, and Yoshua Bengio. Theano: Deep Learning on GPUs with Python. In *NIPSW*, 2011.

[5] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[8] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.

[9] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12), 2009.

[10] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2625–2634, 2015.

[11] DyNet Variable Length LSTM. `https://github.com/neulab/dynet-benchmark`.

[12] Facebook. `http://pytorch.org/`.

[13] Facebook Open Source. Caffe2 is a lightweight, modular, and scalable deep learning framework. `https://github.com/caffe2/caffe2`, 2017.

[14] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs.

[15] Google TensorFlow XLA. `https://www.tensorflow.org/performance/xla/`.

[16] Édouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. Efficient softmax approximation for gpus. *arXiv preprint arXiv:1609.04309*, 2016.

[17] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[18] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C Schulthess. Stella: A domain-specific tool for structured grid methods in weather and climate models. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[19] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[20] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[21] Intel Open Source Technology Center. Intel(r) math kernel library for deep neural networks (intel(r) mkl-dnn). `https://github.com/01org/mkl-dnn`, 2017.

[22] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

[24] Xiaodan Liang, Xiaohui Shen, Jiashi Feng, Liang Lin, and Shuicheng Yan. Semantic object parsing with graph lstm. In *European Conference on Computer Vision*, pages 125–143. Springer, 2016.

[25] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[26] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[27] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[28] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[29] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[30] Graham Neubig, Yoav Goldberg, and Chris Dyer. On-the-fly operation batching in dynamic computation graphs. *arXiv preprint arXiv:1705.07860*, 2017.

[31] NVIDIA. `http://docs.nvidia.com/cuda/nvrtc/index.html`.

[32] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226.

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[34] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[35] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[36] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[37] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[38] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[39] TensorFlow Fixed-sized LSTM Language Model. `https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb_word_lm.py`.

[40] TensorFlow Fold Benchmark Code. `https://github.com/tensorflow/fold/tree/master/tensorflow_fold/loom/benchmarks`.

[41] The Penn Tree Bank (PTB) Dataset. `http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz`.

[42] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[43] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, 2015.

[44] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

[45] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *Advances in Neural Information Processing Systems*, pages 2773–2781, 2015.

[46] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[47] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

[48] Zhicheng Yan, Hao Zhang, Vignesh Jagadeesh, Dennis DeCoste, Wei Di, and Robinson Piramuthu. Hd-cnn: Hierarchical deep convolutional neural network for image classification. *ICCV*, 2015.

[49] Zhicheng Yan, Hao Zhang, Baoyuan Wang, Sylvain Paris, and Yizhou Yu. Automatic photo adjustment using deep neural networks. *ACM Transactions on Graphics (TOG)*, 35(2):11, 2016.

[50] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[51] Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric Xing. Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216*, 2015.

[52] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, 2017. USENIX Association.