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

1,2Shizhen Xu*, 1,3Hao Zhang*, 1,3Graham Neubig, 1,3Wei Dai, 1,3Qirong Ho, 2Guangwen Yang, 3Eric P. Xing
1Carnegie Mellon University, 2Tsinghua University, 3Petuum 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 \( F \) and a dynamic instance-specific graph \( G \), and performs backpropagation by scheduling the execution of \( F \) following the dependencies in \( G \). Cavs bypasses expensive graph construction and preprocessing overhead, allows for the use of static graph optimization techniques on pre-defined operations in \( F \), 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]. 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 [4]. 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 \( F \) that is only declared (by the user) and optimized once, and an input graph \( G \) 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 \( F \) following the structure of the input graph \( G \). 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 \( F \) symbolically by “thinking locally like a vertex” [42]. Cavs will perform auto-differentiation, schedule the function execution following the dependency reflected by \( G \), 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 (§7). 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).

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

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

\section{Related Work}

Table 1 gives an overview of the landscape Most existing DL frameworks, including Caffe [22], Theano [4], Tensorflow [11], 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.

\section{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 $G$, and a node in that graph as a vertex, to be distinguished from a dataflow graph $D$ 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 $G$, while Cavs will ensure the correctness and efficiency. Therefore, a vertex function $\mathcal{F}$, together with an input graph $G$, 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 $G$ as external to $(\mathcal{F}, G)$, and vice versa. Cavs allows users to connect any external static dataflow graph to a dynamic structure encoded by $(\mathcal{F}, G)$ 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 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.

**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 \( F \) and \( G \) (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 pulled by others, the content of op will be returned.

With appropriate indexing, push and pull connect a vertex inside a dynamic structure expressed by \((F,G)\) to other connectors external to \((F,G)\), 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 \( F \) it derives \( \partial F \) following the auto-differentiation rules: for each math expression such as \( s_i = \text{op}\(s_r\) in \( F \), Cavs generates a corresponded backward expression in \( \partial F \) as \( \nabla s_i = \text{grad\_op}\(\nabla s_r, s_l, s_s\) \). 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_i = \text{gather}\(\text{child\_idx}\) in \( F \), Cavs will generate a backward expression \( \text{scatter}\(\nabla s_i\) \) in \( \partial F \). 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 $F$ and launch the execution, the Cavs scheduler arranges the evaluation of $F$ over the input graphs to perform the backpropagation.

**Backpropagation.** Cavs performs backpropagation \[19\] as follows. For a sample $x_i$ with its input graph $G_i$, the scheduler starts the forward pass from the input vertices of $G_i$, and proceeds following the direction indicated by the edges in $G_i$: at each sub-step, the scheduler figures out the next activated vertex in $G_i$, and evaluates $F$ at this vertex following the symbolic programs in $F$. 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 $G$ 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 $\partial F$, which is the backward function of $F$ 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 $\{G_i\}_{i=1}^N$, 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 rely 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 on 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 \[58\], 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 $h$. 

<table>
<thead>
<tr>
<th>Model</th>
<th>CYCLON</th>
<th>TF</th>
<th>Fold</th>
<th>DyNet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fixed-LSTM</strong></td>
<td>(b = 64)</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td><strong>Var-LSTM</strong></td>
<td>(b = 64)</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td><strong>Tree-FC</strong></td>
<td>(b = 64)</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td><strong>Tree-LSTM</strong></td>
<td>(b = 64)</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
</tr>
</tbody>
</table>
We present Cavs as a vertex-centric programming interface as well as an efficient system for dynamic

We then turn to Tree-FC, a dynamic model for benchmarking. Since vanilla TensorFlow is unable to

Finally, we compare three frameworks on Tree-LSTM in Figure 2(d)(h): Cavs is 8-10x faster than

Others. Despite system advantages, we also try to investigate whether Cavs, as an interface, simplifies

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


