Cavs: An Efficient Runtime System for Dynamic Neural Networks

Shizhen Xu, ***Carnegie Mellon University, Tsinghua University;*** **Hao Zhang, Graham Neubig, and Wei Dai,** ***Carnegie Mellon University, Petuum Inc.;*** **Jin Kyu Kim,** ***Carnegie Mellon University;*** **Zhijie Deng,** ***Tsinghua University;*** **Qirong Ho,** ***Petuum Inc.;*** **Guangwen Yang,** ***Tsinghua University;*** **Eric P. Xing,** ***Petuum Inc.***

https://www.usenix.org/conference/atc18/presentation/xu-shizen

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Cavs: An Efficient Runtime System for Dynamic Neural Networks

[1,2]Shizhen Xu[†], [1,3]Hao Zhang[†], [1,3]Graham Neubig, [3]Wei Dai, [1]Jin Kyu Kim, [2]Zhijie Deng,
[3]Qirong Ho, [2]Guangwen Yang, [3]Eric P. Xing

Carnegie Mellon University[1], Tsinghua University[2], Petuum Inc.[3] ([†]equal contributions)

Abstract

Recent deep learning (DL) models are moving more and more to dynamic neural network (NN) architectures, where the NN structure changes for every data sample. However, existing DL programming models are inefficient in handling dynamic network architectures because of: (1) substantial overhead caused by repeating dataflow graph construction and processing every example; (2) difficulties in batched execution of multiple samples; (3) inability to incorporate graph optimization techniques such as those used in static graphs. In this paper, we present "Cavs", a runtime system that overcomes these bottlenecks and achieves efficient training and inference of dynamic NNs. Cavs represents a dynamic NN as a static vertex function $\mathcal{F}$ and a dynamic instance-specific graph $\mathcal{G}$. It avoids the overhead of repeated graph construction by only declaring and constructing $\mathcal{F}$ once, and allows for the use of static graph optimization techniques on pre-defined operations in $\mathcal{F}$. Cavs performs training and inference by scheduling the execution of $\mathcal{F}$ following the dependencies in $\mathcal{G}$, hence naturally exposing batched execution opportunities over different samples. Experiments comparing Cavs to state-of-the-art frameworks for dynamic NNs (TensorFlow Fold, PyTorch and DyNet) demonstrate the efficacy of our approach: Cavs achieves a near one order of magnitude speedup on training of dynamic NN architectures, and ablations verify the effectiveness of our proposed design and optimizations.

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 [53, 54, 30]. One reason for its widespread adoption is the variety and quality of software toolkits, such as Caffe [23], TensorFlow [1], PyTorch [36] and DyNet [33, 34], which ease programming of DL models, and speed computation by harnessing modern computing hardware (e.g. GPUs), software libraries (e.g. CUDA, cuDNN [6]), and compute clusters [56, 57, 7].

One dominant programming paradigm, adopted by DL toolkits such as Caffe and TensorFlow, is to represent a neural network as a static dataflow graph [32, 1], where computation functions in the NN are associated with nodes in the graph, and input and output of the computation map to edges. It requires DL programmers to define the network architecture (i.e. the dataflow graph) using symbolic expressions, once before beginning execution. Then, for a given graph and data samples, the software toolkits can automatically derive the correct algorithm for training or inference, following backpropagation [21] and auto-differentiation rules. With proper optimization, the execution of these static dataflow graphs can be highly efficient; as the dataflow graph is fixed for all data, the evaluation of multiple samples through one graph can be naturally batched, leveraging the improved parallelization capability of modern hardware (e.g. GPUs). Moreover, by separating model declaration and execution, it makes it possible for the graph to be optimized once at declaration time [1], with these optimizations benefiting the efficiency of processing arbitrary input data batches at execution time.

While the dataflow graph has major efficiency advantages, its applicability highly relies on a key assumption – the graph (i.e. NN architecture) is fixed throughout the runtime. This assumption however breaks for *dynamic* NNs, where the network architectures conditionally change with every input sample, such as NNs that compute over sequences of variable lengths [22, 43], trees [45], and graphs [26].

Due to the growing interest in these sorts of dynamic models, recent years have seen an increase in the popularity of frameworks based on *dynamic declaration* [49, 33, 11], which declare a different dataflow graph per sample. While dynamic declaration is convenient to developers as it removes the restriction that computation be completely specified before training begins, it exhibits a few limitations. First, constructing a graph for every

sample results in substantial overhead, which grows linearly with the number of input instances. In fact, we find graph construction takes longer time than the computation in some frameworks (see §5.2). It also prevents the application of complex static graph optimization techniques (see §3.4). Moreover, since each sample owns a dataflow graph specifying its unique computational pattern, batching together similarly shaped computations across instances is non-trivial. Without batching, the computation is inefficient due to its lack of ability to exploit modern computational hardware. While some progress has been made in recent research [34, 27], how to automatically batch the computational operations from different graphs remains a difficult problem.

To address these challenges, we present Cavs, an efficient runtime system for dynamic NNs that exploits the recurrent and recursive nature of dynamic NNs. Instead of declaring a dataflow graph per sample, it decomposes a dynamic NN into two components: a static vertex function $\mathcal{F}$ that is only declared (by the user) and optimized once before execution, and an input-specific graph $\mathcal{G}$ obtained via I/O at runtime. Cavs inherits the flexibility of symbolic programming [1, 12, 33] for DL; it requires users to define $\mathcal{F}$ by writing symbolic expressions in the same way as in static declaration. With $\mathcal{F}$ and $\mathcal{G}$, the workflow of training or testing a dynamic NN is cast as scheduling the execution of $\mathcal{F}$ following the structure of the input graph $\mathcal{G}$. Cavs will perform auto-differentiation, schedule the execution following dependencies in $\mathcal{G}$, and guarantee efficiency and correctness.

Cavs' design allows for highly efficient computation in dynamic graphs for a number of reasons. First, it allows the vertex function only to be defined and constructed once for any type of structured data, hence avoiding the overhead of repeated dataflow graph construction. Second, as the dataflow graph encoded by the vertex function is static throughout the runtime, it can benefit from various static graph optimizations [1, 5, 12, 18](§3.4), which is not the case in the scenario of dynamic declaration (§2.2). Moreover, it naturally exposes opportunities for batched computation, i.e. we are able to parallelize the execution of $\mathcal{F}$ over multiple vertices from different input graphs (§3.2) with the support of our proposed memory management strategy (§3.3).

To evaluate Cavs' performance, we compare it to several state-of-the-art systems supporting dynamic NNs. 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 (§5.2). In a comparison with unbatched dynamic graphs in PyTorch and DyNet, two widely-used dynamic NN libraries, we verify that batching is essential for efficient processing. In a comparison with TensorFlow Fold and DyNet Auto-batching, two libraries that allow for the use of dynamic NNs with automatic operation batching, we find that Cavs' has significant performance advantages; on static graphs it performs equivalently or slightly better, and on dynamic NNs with difficult-to-batch workloads (e.g. Tree-LSTM [45] and Tree-FC [27]), Cavs demonstrates near one order of magnitude speedups across multiple dataset and hyper-parameter settings (§5.1). We further investigate the effectiveness of our design choices: Cavs benefits from not only our proposed memory management strategy, but also various optimizations on graph execution, which were originally for static dataflow graphs and not applicable in dynamic declaration.

To summarize, we make three primary contributions in this paper: (1) We propose a novel representation for dynamic NNs, based on which we design four APIs and implement the Cavs runtime system (§3.1); (2) We propose several novel strategies in Cavs for efficient training and inference of dynamic NNs: the batching policy (§3.2), a memory management mechanism to guarantee the memory coalescing (§3.3), and multiple graph execution optimizations (§3.4); (3) We compare Cavs to state-of-the-art systems for dynamic NNs (§5). We reveal the problems of existing systems, and report near 10x speedup for Cavs on various experimental settings. We also verify the effectiveness of our proposed design strategies, and quantize their contributions to the final performance.

2 Background

2.1 Dynamic Neural Networks

Successful NN models generally exhibit suitable architectures that capture the structures of the input data. For example, convolutional neural networks [24, 53], which apply fixed-structured operations to fixed-sized images, are highly effective precisely because they capture the spatial invariance common in computer vision domains [39, 44]. However, apart from images, many forms of data are structurally complex and can not be readily captured by fixed-structured NNs. Appropriately reflecting these structures in the NN design has shown effective in sentiment analysis [45], semantic similarity between sentence pairs [40], and image segmentation [26].

To see this, we will take the constituency parsing problem as an example. Sentences in natural languages are often represented by their constituency parse tree [45, 31], whose structure varies depending on the content of the sentence itself (Fig. 1(a)). Constituency parsing is an important problem in natural language processing that aims to determine the corresponding grammar type of all internal nodes given the parsing tree of a sentence. Fig. 1(b) shows an example of a network that takes into account this syntactic structure, generating representations for the sentence by traversing the parse tree bottom-up and combining the representations for each sub-tree

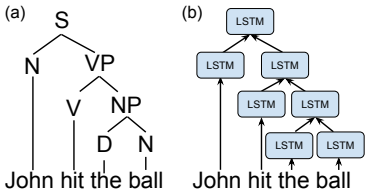

Figure 1: An example of a dynamic NN: (a) a constituency parsing tree, (b) the corresponding Tree-LSTM network. We use the following abbreviations in (a): S for sentence, N for noun, VP for verb phrase, NP for noun phrase, D for determiner, and V for verb.

using a dynamic NN called Tree Structured Long Short-term Memory (Tree-LSTM) [45]. In particular, each node of the tree maps to a LSTM function [22]. The internal computations and parameters of the LSTM function is defined in Fig. 4. At each node, it takes a variable number of inputs and returns to the parent node a vector representing the parsing semantics up to that point, until the root LSTM node returns a vector representing the semantics of the entire sentence.

The important observation is that the NN structure varies with the underlying parsing tree over *each* input sample, but the same LSTM cell is constant in shape and repeated at each internal node. Similar examples can be found for graph input [25, 26] and sequences of variable lengths [43, 2]. We refer to these NNs that exhibit different structures for different input samples as *dynamic neural networks*, in contrast to the static networks that have fixed network architecture for all samples.

2.2 Programming Dynamic NNs

There is a natural connection between NNs and directed graphs: we can map the graph nodes to the computational operations or parameters in NNs, and let the edges indicate the direction of the data being passed between the nodes. In this case, we can represent the process of training NNs as batches of data flowing through computational graphs, i.e. *dataflow graphs* [3, 1, 33].

Static declaration. As mentioned previously, *static declaration* is one dominant programming paradigm for programming NNs [3, 1, 5]. Fig 2(a) summarizes its workflow, which assumes all data samples share a fixed NN structure declared symbolically in a dataflow graph $\mathcal{D}$. Static declaration, using a single dataflow graph $\mathcal{D}$, cannot express dynamic NNs with structures changing with data samples. A primary remedy to this problem is to forgo the efficiency gains of static dataflow graphs and instead use a *dynamic declaration* framework.

Dynamic declaration. Fig 2(b) illustrates the workflow of dynamic declaration. By creating a unique dataflow graph $\mathcal{D}_k^p$ for each sample x_k^p according to its associated structure, dynamic declaration is able to express sample-dependent dataflow graphs. It however causes extra overhead on graph construction and puts constraints on runtime optimization, which usually lead to inefficient execution. Particularly, since a dataflow graph $\mathcal{D}_k^p$ needs to be constructed per sample, the overhead is linearly increasing with the number of samples, and sometimes yields downgraded performance [27] (§5.2), even for frameworks with optimized graph construction implementations [33]. Moreover, we can hardly benefit from any well-established dataflow graph optimization (§3.4). We will have to perform graph processing/optimization for each dataflow graph and every single sample; but incorporating this optimization itself has a non-negligible overhead. More importantly, as we are unable to batch the computation of different structured graphs, we note in Fig 2(b) single-instance computation $\mathcal{D}_k^p(x_k^p)$ would be very inefficient in the absence of batched computation.

Dynamic batching. To address the batching problem, some recent effort, notably TensorFlow Fold [27] and DyNet [34], propose *dynamic batching* that dynamically groups similarly shaped operations from different graphs, and batch their execution whenever possible.

Fold turns dynamic dataflow graphs into a static control flow graph to enable batched execution, but introduces a complicated functional programming-like interface and a large graph preprocessing overhead. As we will show in §5.2, the graph construction sometimes slows down the computation by 4x. DyNet proposes an auto-batching strategy that searches for batching opportunities by profiling every fine-grained operator, while this step itself has non-negligible overhead (§5.2). It is also not open to dataflow graph level optimizations.

In summary, there are three major challenges that prevent the efficient execution of dynamic neural networks: (1) non-negligible graph construction overhead; (2) difficulties in parallel execution; (3) unavailability to graph execution optimization.

2.3 Motivation

Our motivation for Cavs comes from a key property of dynamic NNs: most dynamic NNs are designed to exhibit a recursive structure; Within the recursive structure, a static computational function is being applied following the topological order over instance-specific graphs. For instance, if we denote the constituency parsing tree in §2.1 as a graph $\mathcal{G}$, where each node of the tree maps to a vertex in $\mathcal{G}$, we note the Tree-LSTM can be interpreted as follows: a computational cell function, specified in advance, is applied from leaves to the root, following the dependencies in $\mathcal{G}$. $\mathcal{G}$ might change with input samples, but the cell function itself is always static: It is parametrized by a fixed set of learnable parameters and interacts in the same way with its neighbors when applied at different vertices of $\mathcal{G}$.

These observations motivate us to decompose a dynamic NN into two parts: (1) a static computational *vertex function* $\mathcal{F}$ that needs to be declared by the program-

```
/* (a) static declaration */
// all samples must share one graph
declare a static dataflow graph 𝒟.
for p = 1 → P:
  read the pth data batch {x_k^p}_{k=1}^K.
  batched computation: 𝒟({x_k^p}_{k=1}^K).
```

```
/* (b) dynamic declaration */
for p = 1 → P:
  read the pth data batch {x_k^p}_{k=1}^K.
  for k = 1 → K:
    declare a dataflow graph 𝒟_k^p for x_k^p.
    single-instance computation: 𝒟_k^p(x_k^p).
```

```
/* (c) our proposed vertex-centric model */
declare a symbolic vertex function ℱ.
for p = 1 → P:
  read the pth data batch {x_k^p}_{k=1}^K.
  read their associated graphs {𝒢_k^p}_{k=1}^K.
  compute ℱ over {𝒢_k^p}_{k=1}^K with inputs {x_k^p}_{k=1}^K.
```

Figure 2: The workflows of (a) static declaration, (b) dynamic declaration, (c) Cavs. Notations: $\mathcal{D}$ notates both the dataflow graph itself and the computational function implied by it; p is the index of a batch while k is the index of a sample in the batch.

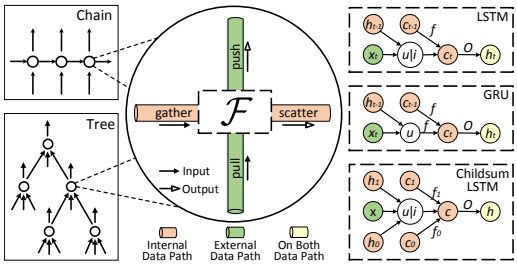

Figure 3: Cavs represents a dynamic structure as a dynamic input graph $\mathcal{G}$ (left) and a static vertex function $\mathcal{F}$ (right).

mer once before runtime; (2) a dynamic *input graph* $\mathcal{G}$ that changes with every input sample[1]. With this representation, the workflow of training a dynamic NN can be cast as scheduling the evaluation of the symbolic construct encoded by $\mathcal{F}$, following the graph dependencies of $\mathcal{G}$, as illustrated in Fig 2(c). This representation exploits the property of dynamic NNs to address the aforementioned issues in the following ways:

Minimize graph construction overhead. Cavs only requires users to declare $\mathcal{F}$ using symbolic expressions, and construct it once before execution. This bypasses repeated construction of multiple dataflow graphs, avoiding overhead. While it is still necessary to create an I/O function to read input graphs $\mathcal{G}$ for each sample, this must be done by any method, and only once before training commences, and it can be shared across samples.

Batched execution. With the proposed representation, Cavs transforms the problem of evaluating data samples $\{x_k^p\}_{k=1}^K$ (at the pth batch) on different dataflow graphs $\{\mathcal{D}_k^p\}_{k=1}$ [27, 34] into a simpler form – scheduling the execution of the vertex function $\mathcal{F}$ following the dependencies in input graphs $\{G_k^p\}_{k=1}$. For the latter problem, we can easily batch the execution of $\mathcal{F}$ on multiple vertices at runtime (§3.2), leveraging the batched computational capability of modern hardware and libraries.

Open to graph optimizations. Since the vertex function $\mathcal{F}$ encodes a dataflow graph which is static throughout runtime, it can benefit from various graph optimizations originally developed for static declaration, such as kernel fusion, streaming, and our proposed lazy batching, which are not effective in dynamic declaration.

Based on this motivation, we next describe the Cavs system. Cavs faces the following challenges in system design: (1) how to design minimal APIs in addition to the symbolic programming interface to minimize user code; (2) how to schedule the execution of $\mathcal{F}$ over multiple input graphs to enable batched computation; (3) how to manage memory to support the dynamic batching; (4) how to incorporate static graph optimization in Cavs's execution engine to exploit more parallelism.

[1]In the following text, we will distinguish the term *vertex* from *node*. We use *vertex* to denote a vertex in the input graph while *node* to denote an operator/variable in a dataflow graph. Hence, a vertex function can have many nodes as itself represents a dataflow graph.

3 Cavs Design and Optimization

3.1 Programming Interface

Conventional dataflow graph-based programming models usually entangle the computational workflow in $\mathcal{F}$ with the structure in $\mathcal{G}$, and require users to express them as a whole in a single dataflow graph. Instead, Cavs separates the static vertex function $\mathcal{F}$ from the input graph $\mathcal{G}$ (see Fig 3). While users use the same set of symbolic operators [1, 11] to assemble the computational workflow in $\mathcal{F}$, Cavs proposes four additional APIs, `gather`, `scatter`, `pull`, `push`, to specify how the messages shall be passed between connected vertices in $\mathcal{G}$:

- `gather(child_idx)`: `gather` accepts an index of a child vertex, gets its output, and returns a list of symbols that represent the output of the child.
- `scatter(op)`: `scatter` reverses `gather`. It sets the output of the current vertex as `op`. If this vertex is `gather`ed, the content of `op` will be returned.

`gather` and `scatter` are motivated by 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 (see discussion in §6).

However, in dynamic NNs, the vertex function $\mathcal{F}$ usually takes input from not only the internal vertices of $\mathcal{G}$ (internal data path in Fig 3), but also the external environment, e.g. an RNN can take inputs from a CNN feature extractor or some external I/O (external data path in Fig 3). Cavs therefore provides another two APIs to express such semantics:

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

```
def F():
  for k in range(N):
    S = gather(k) # gather states of child vertices
    c_k, h_k = split(S, 2) # get hidden states c and h
  x = pull() # pull the first external input x

  # specify the computation
  h = Σ_{k=0}^{N-1} h_k
  i = sigmoid(W^(i)× x + U^(i)× h + b^(i))
  for k in range(N):
    f_k = sigmoid(W^(f)× x + U^(f)× h_k + 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 + Σ_{k=0}^{N-1} f_k ⊗ c_k
  h = o ⊗ tanh(c)

  scatter(concat([c, h], 1)) # scatter c, h to parents
  push(h)                    # push to external connectors
```

Figure 4: The vertex function of an N-ary child-sum TreeLSTM [45] in Cavs. Within $\mathcal{F}$, users declare a computational dataflow graph using symbolic operators. The defined $\mathcal{F}$ will be evaluated on each vertex of $\mathcal{G}$ following graph dependencies.

Once $\mathcal{F}$ declared, together with an input graph $\mathcal{G}$, they encode a recursive dataflow graph structure, 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. Via `push` and `pull`, Cavs allows users to connect any external static dataflow graph to a dynamic structure encoded by $(\mathcal{F},\mathcal{G})$, to express more complex model architectures, such as the LRCN [9] (i.e. connecting a CNN to an RNN), or an encoder-decoder LSTM network [43] (i.e. connecting two different recursive structures). With these four APIs, we present in Fig 4 an example user program how the N-ary child-sum Tree-LSTM [45] can be simply expressed by using them and other mathematical operators.

Auto-differentiation. Given a vertex function $\mathcal{F}$ Cavs 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 $\nabla s_r = \texttt{grad\_op}(\nabla s_l, s_l, s_r)$ in $\partial\mathcal{F}$. For the four proposed operators, we note `scatter` is the gradient operator of `gather` in the sense that if `gather` collects inputs from child vertex written by `scatter` at the forward pass, a `scatter` needs to be performed to write the gradients 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 rules apply for `push` and `pull`.

Expressiveness. With these four APIs, Cavs can be seen as a middle ground between static and dynamic declaration. In the best case that the NN is fully recursive (e.g. most recurrent or recursive NNs), it can be represented by a single vertex function and an input graph. While in the worst case, that every sample has a unique input graph while every vertex in the graph has a unique way to interact with its neighboring vertices (i.e. the NN is dynamic but non-recursive), Cavs reduces to dynamic declaration that one has to define a vertex function for each vertex of each input graph. Fortunately, dynamic NNs in this scenario are usually avoided because of the difficulties in design, programming and learning.

3.2 Scheduling

Once $\mathcal{F}$ is defined and $\mathcal{G}$ is obtained from I/O, Cavs will perform computation by scheduling the evaluation of $\mathcal{F}$ over data samples $\{x_i\}_{i=1}^N$ and their input graphs $\{\mathcal{G}_i\}_{i=1}^N$.

Forward pass. 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 all expressions in $\mathcal{F}$ at this vertex. 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.

Backward pass. 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 evaluates $\partial\mathcal{F}$ at each vertex until all vertices have been evaluated in the backward pass.

To train a NN to convergence, the above process has to be iterated on all samples $\{x_i\}_{i=1}^N$ and their input graphs $\{\mathcal{G}_i\}_{i=1}^N$, for many epochs. We next describe our batched execution policy to speed the computation.

Batching policy. Given a data batch $\{x_k\}_{k=1}^K \subseteq \{x_i\}_{i=1}^N$ and associated graphs $\{\mathcal{G}_k\}_{k=1}^K$, this policy groups multiple vertices and performs batched evaluation of $\mathcal{F}$ in order to reduce kernel launches and exploit parallelism. Specifically, a forward pass over a batch $\{x_k\}_{k=1}^K$ are performed in multiple steps. At each step t, Cavs analyzes $\{\mathcal{G}_k\}_{k=1}^K$ at runtime and determines a set V_t that contains all activated vertices in graphs $\{\mathcal{G}_k\}_{k=1}^K$. It then evaluates $\mathcal{F}$ over these vertices by creating a *batched execution task*, with the task ID set to t[2]. The task is executed by the Cavs execution engine (§3.4). Meanwhile, the scheduler records this task by pushing V_t into a stack $\mathcal{S}$. To perform backward pass, the scheduler pops out an element V_t from $\mathcal{S}$ at each step – the execution engine will evaluate the derivative function $\partial\mathcal{F}$ over vertices in V_t, until all vertices of $\{\mathcal{G}_k\}_{k=1}^K$ are evaluated.

We note the batching policy is similar to the *dynamic batching* in Fold [27] and DyNet [33]. However, Cavs determines how to batch fully dynamically during runtime using simple breadth-first search with negligible cost (instead of analyzing full dataflow graphs before every iteration of the execution). Since batched computation requires the inputs to an expression over multiple

[2]Whenever the context is clear, we use V_t to denote both the set of vertices to be batched together, and the batched execution task itself.

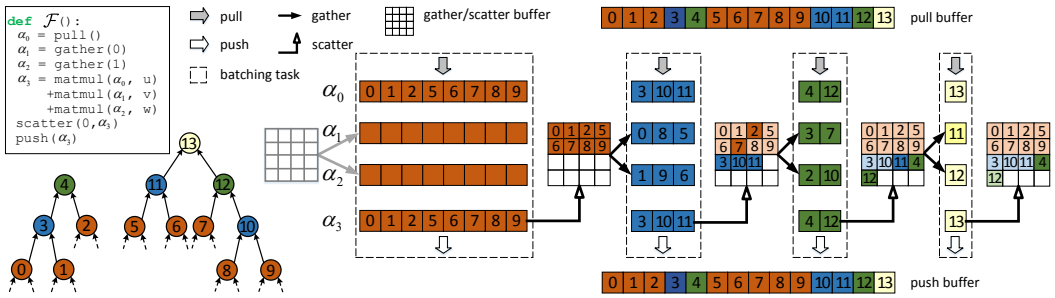

Figure 5: The memory management at the forward pass of $\mathcal{F}$ (top-left) over two input trees (bottom-left). Cavs first analyzes $\mathcal{F}$ and inputs – it creates four dynamic tensors $\{\alpha_n\}_{n=0}^{3}$, and figures out there will be four batch tasks (dash-lined boxes). Starting from the first task (orange vertices $\{0,1,2,5,6,7,8,9\}$), Cavs performs batched evaluation of each expression in $\mathcal{F}$. For example, for the pull expression $\alpha_0 = \texttt{pull}()$, it indexes the content of α_0 on all vertices from the *pull buffer* using their IDs, and copies them to α_0 continuously; for scatter and push expressions, it scatters a copy of the output (α_3) to the *gather buffer*, and pushes them to the push buffer, respectively. Cavs then proceeds to the next batching task (blue vertices). At this task, Cavs evaluates each expression of $\mathcal{F}$ once again for vertices $\{3,10,11\}$. (e.g. for a pull expression $\alpha_0 = \texttt{pull}()$, it pulls the content of α_0 from pull buffer again; for a gather expression $\alpha_2 = \texttt{gather}(1)$ at vertex 3, it gathers the output of the second child of 3, which is 1); it writes results continuously at the end of each dynamic tensor. It proceeds until all batching tasks are finished.

vertices to be placed on a continuous memory buffer, we develop a new memory management support for it.

3.3 Memory Management

In static declaration [1, 33], a symbol in the user program usually corresponds to a fixed-sized *tensor* object with a batch size dimension. While in Cavs, each batching task V_t is determined at runtime. For the batched computation to be efficient, Cavs must guarantee for each batching task, the inputs to each expression of $\mathcal{F}$ over a group of runtime-determined vertices coalescing in memory.

Cavs proposes a novel data structure *dynamic tensor* to address this challenge (Fig 6). A dynamic tensor is a wrapper of a multi-dimensional array [1, 52]. It contains four attributes: `shape`, `bs`, a pointer `p` to a chunk of memory, and `offset`. `shape` is an array of integers representing the specific shape of the tensor excluding the batch dimension. It can be inferred from the user program and set before execution. The batch size `bs` is dynamically set by the scheduler at runtime at the beginning of a batching task. To access a dynamic tensor, one moves `p` forward with the value of `offset`, and reads/writes number of elements equal to $\texttt{bs} \cdot \prod_i \texttt{shape}[i]$. Therefore, `bs` together with `offset` provide a view of the tensor, and the state of the tensor will vary based on their values. Given a vertex function $\mathcal{F}$, Cavs creates dynamic tensors $\{\alpha_n\}_{n=1}^{N}$ for each non-parameter symbol $s_n (n = 1,\ldots,N)$ in $\mathcal{F}$, and also $\{\nabla\alpha_n\}_{n=1}^{N}$ as their gradients, while it creates static tensors for model parameters.

```
struct DynamicTensor {
  vector<int> shape;
  int bs;
  int offset;
  void* p; };
```

Figure 6: Dynamic tensor.

Fig 5 illustrates how the memory is assigned during the forward pass by manipulating dynamic tensors. In particular, in a training iteration, for a batching task V_t, the scheduler sets `bs` of all $\{\alpha_n\}_{n=1}^{N}$ to $M_t = |V_t|$ (the number of vertices in V_t). The execution engine then performs batched evaluation of each expression in $\mathcal{F}$. For an expression $s_l = \texttt{op}(s_r)$[3], Cavs first accesses α_r (the dynamic tensor of the RHS symbol s_r) – it offsets α_r.`p` by α_r.`offset`, and reads a block of $M_t \prod_i \alpha_r.\texttt{shape}[i]$ elements, and presents it as a tensor with batch size M_t and other dimensions as α_r.`shape`. It then applies batched computational kernels of the operator `op` over this memory block, and writes the results to α_l (the dynamic tensor of the LHS symbol s_l) on the continuous block in between $[\alpha_l.\texttt{p} + \alpha_l.\texttt{offset}, \alpha_l.\texttt{p} + \alpha_l.\texttt{offset} + M_t \prod_i \alpha_l.\texttt{shape}[i]]$. Upon the completion of V_t, the scheduler increases `offset` of all $\{\alpha_n\}_{n=1}^{N}$ by $M_t \prod_i \alpha_n.\texttt{shape}[i]$, respectively. It then starts the next task V_{t+1}. Hence, intermediate results generated in each batching task at forward pass are stored continuously in the dynamic tensors, and their offsets are recorded.

At the entrance of $\mathcal{F}$, the vertices $\{v_m\}_{m=1}^{M_t}$ in V_t need to interact with its dependent vertices in previous V_{t-1} to `gather` their outputs as inputs (L3 of Figure 4), or `pull` inputs from the external (L5 of Figure 4). Cavs maintains memory buffers to enable this (Figure 5). It records the offsets of the dynamic tensors for each $v_m \in V_t$, and therefore during the execution of `gather` operator, the memory slices of specific children can be indexed. As shown in Figure 5, `gather` and `scatter` share the same temporary buffer for memory re-organization, but `push` and `pull` operate on external memory buffers.

Algorithm 1 summarizes the memory management during forward pass. The backward execution follows an exactly reverse order of the forward pass (§3.2), which we skip in the text. With this strategy, Cavs guarantees memory continuity for any batched computation of $\mathcal{F}$ and $\partial\mathcal{F}$. Compared to dynamic batching in DyNet, Cavs performs memory movement only at the entrance

[3]Note that the user-defined expressions can be arbitrary, e.g. with more than one argument or return values

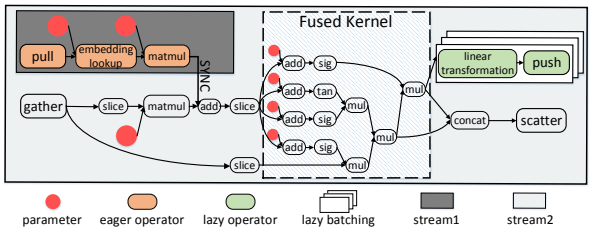

Figure 7: The dataflow graph encoded by $\mathcal{F}$ of Tree-LSTM.

and exit of $\mathcal{F}$, instead of for each expression (operator). We empirically find this significantly reduces overhead of memory operations (§5.3).

Algorithm 1 Memory management at forward pass.

function FORWARD($\{V_t\}_{t=1}^{T}, \{\alpha_n\}_{n=1}^{N}, \mathcal{F}$)
  for $t = 1 \to T$ **do**
    for $n = 1 \to N$ **do** α_n.bs $\leftarrow M_t$ **end for**
    for each expression like $s_l = \mathtt{op}(s_r)$ in $\mathcal{F}$ **do**
      if op $\in \{$gather, pull$\}$ **then**
        $C \leftarrow \prod_i \alpha_l$.shape$[i], q \leftarrow \alpha_l$.p$+\alpha_l$.offset.
        for $v_m \in V_t (m = 1 \to M_t)$ **do**
          src $\leftarrow$ IndexBuffer(op$,m$), dest $\leftarrow q + (m-1)C$.
          memcpy(dest, src, C).
        end for
      else if op $\in \{$scatter, push$\}$ **then**
        $C \leftarrow \prod_i \alpha_r$.shape$[i], q \leftarrow \alpha_r$.p$+\alpha_r$.offset.
        for $v_m \in V_t (m = 1 \to M_t)$ **do**
          dest $\leftarrow$ IndexBuffer(op$,m$), src $\leftarrow q + (m-1)C$.
          memcpy(dest, src, C).
        end for
      else
        perform batched computation: $\alpha_l =$ op_kernel(α_r).
      end if
    end for
    for $n = 1 \to N$ **do** α_n.offset$+= M_t \prod_i \alpha_n$.shape$[i]$ **end for**
  end for
end function

3.4 Optimizing Execution Engine

Since Cavs separates out a static dataflow graph encoded by $\mathcal{F}$, we can replace the original $\mathcal{F}$ with an optimized one that runs more efficiently, as long as maintaining correctness. We next described our optimization strategies.

Lazy batching and streaming[4]**.** In addition to batched execution of $\mathcal{F}$, the lazy batching and streaming explore potential parallelism for a certain group of finer-grained operators in $\mathcal{F}$ or $\partial\mathcal{F}$ called lazy and eager operators.

Definition. An operator in $\mathcal{F}$ ($\partial\mathcal{F}$) is a *lazy operator* if at the forward (backward) pass, for $\forall v \in \mathcal{G}, \forall \mathcal{G} \in \{\mathcal{G}_k\}_{k=1}^{K}$, the evaluation of $\mathcal{F}$ ($\partial\mathcal{F}$) at any parent (dependent) vertex of v does not rely on the evaluation of $\mathcal{F}$ at v. It is an *eager operator* if the evaluation at v does not rely on the evaluation of $\mathcal{F}$ ($\partial\mathcal{F}$) at any dependents (parents) of v.

Proposition. Denote $\mathcal{D}_{\mathcal{F}}$ ($\mathcal{D}_{\partial\mathcal{F}}$) as the dataflow graph encoded by $\mathcal{F}$ ($\partial\mathcal{F}$), and $g, s \in \mathcal{D}_{\mathcal{F}}$ ($\mathcal{D}_{\partial\mathcal{F}}$) as nodes of gather and scatter operator, respectively. A node that has g as its dependent and is not on any path from g to s is a lazy operator. A node that has s as its ancestor and is not on any path from g to s is an eager operator.

Fig 7 illustrates a forward dataflow graph of the vertex function of Tree-LSTM, with eager and lazy operators colored. A property of them is that their evaluation is not fully subject to the dependency reflected by the input graph $\mathcal{G}$. For instance, the pull operator in Fig 7 is eager and can be executed in prior – even before $\mathcal{F}$ has been evaluated at the vertices that gather tries to interact with; the push operator is lazy, so we can defer its execution without impacting the evaluation of $\mathcal{F}$ at parent vertices. Similarly, in $\partial\mathcal{F}$, the gradient derivation for model parameters are mostly lazy – their execution can be deferred as long as the gradients of hidden states are derived and propagated in time. Cavs leverages this property and proposes the lazy batching strategy. It defers the execution of all lazy operators in $\mathcal{F}$ and $\partial\mathcal{F}$ until all batching tasks $\{V_t\}_{t=1}^{T}$ has finished. It then performs a batched execution of these lazy operators over all vertices of $\{\mathcal{G}_k\}_{k=1}^{K}$. These operators includes, but is not limited to, the push operator that is doing memory copy, and operators for computing gradients of model parameters. Lazy batching helps exploit more parallelism and significantly reduces kernel launches. Empirically lazy batching brings 20% overall improvement (§5.3).

To leverage the exhibited parallelization opportunity between eager operators and the operators on the path from gather to scatter (Figure 7), Cavs proposes a streaming strategy that pipelines the execution of these two groups of operators. It allocates two streams, and puts the eager operators on one stream, and the rest (excluding lazy operators) on the other. Hence, independent operators in two streams run in parallel, while for those operators that depend on an eager operator, this dependency is respected by synchronization barriers (Fig 7).

Automatic kernel fusion. Given $\mathcal{F}$, before execution, Cavs will run a *fusion detector* [20] to scan its corresponded dataflow graph and report all *fuse-able* subgraphs therein, i.e. all nodes in a fuse-able subgraph can be fused as a single operator that behaves equivalently but takes less execution time (e.g. with fewer kernel launches and I/O, or faster computation). Currently, we only detect groups of directly linked elementwise operators, such as $+$, sigmoid, as shown in Fig 7, and we use a simple union-find algorithm to detect the largest possible fuse-able subgraphs. Given a fuse-able subgraph, Cavs adopts de facto automatic code generation techniques [37, 8, 38, 35] to generate lower-level kernel implementations. Replacing the original fuse-able subgraphs with fused operators during execution is beneficial in many aspects: (1) it reduces the number of kernel launches; (2) on some devices such as GPUs, kernel fu-

[4]Streaming is a borrowed terminology from CUDA programming which means executing different commands concurrently with respect to each other on different GPU streams. As Cavs' optimizations are agnostic to the low-level hardware, we use streaming interchangeably with multi-threading if the underlying computing hardware is CPU.

sion transform device memory access into faster device registers access. We empirically report another 20% improvement with automatic kernel fusion (§5.3).

4 Implementation

Cavs is implemented as a C++ library and integrable with existing DL frameworks to enhance their support for dynamic NNs. It is composed of three major layers (which is the case for most popular frameworks [3, 1, 33]): (1) a frontend that provides device-agnostic symbolic programming interface; (2) an intermediate layer that implements the core execution logic; (3) a backend with device-specific kernels for all symbolic operators.

Frontend. In addition to the four APIs, Cavs provides a macro operator `VertexFunction`. Users instantiate it by writing symbolic expressions and specifying methods to read input graphs. It encapsulates scatter/gather semantics, so users can continue using higher level APIs. To construct more complex NN architectures (e.g. encoder-decoder LSTM [43], LRCN [9])), users employ `push` and `pull` to connect multiple vertex functions, or to external structures.

Intermediate Layer. Cavs has its core runtime logic at this layer, i.e. the batching scheduler, the memory management, and the execution engine, etc.

Backend. Following practice [1, 33, 12], we implement device-specific operator kernels at this layer. Cavs has optimized implementations for the four proposed operators (`gather`, `scatter`, `pull`, `push`). Specifically, `gather` and `pull` index different slices of a tensor and puts them together continuously on memory; `scatter` and `push` by contrast splits a tensor along its batch dimension, and copy different slices to different places. Cavs implements customized memcpy kernels for there four operators, so that copying multiple slices from (or to) different places is performed within one kernel.

Distributed Execution. While Cavs's implementations are focused on improving the efficiency on a single node, they are compatible with most data-parallel distributed systems for deep learning [56, 7, 1], and can also benefit distributed execution on multiple nodes.

5 Evaluation

In this section, we evaluate Cavs on multiple NNs and datasets, obtaining the following major findings: (1) Cavs has little overhead: on static NNs, Cavs demonstrates equal performance on training and inference with other systems; On several NNs with notably difficult-to-batch structures, Cavs outperforms all existing frameworks by a large margin. (2) We confirm the graph construction overhead is substantial in both Fold [27] and dynamic declaration [33], while Cavs bypasses it by loading input graphs through I/O. (3) We verify the effectiveness of our proposed design and optimization via ablation studies, and discuss Cavs' advantages over other DL systems for dynamic dataflow graphs.

Environment. We perform all experiments in this paper on a single machine with an NVIDIA Titan X (GM200) GPU, a 16-core CPU, and CUDA v8.0 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 mainly compare Cavs to TensorFlow v1.2 [1] with XLA [18] and its variant Fold [27], PyTorch v0.3.0 [11], and DyNet v2.0 [33] with autobatching [34], as they have reported better performance than other frameworks [5, 50] on dynamic NNs. We focus on metrics for system performance, e.g. 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 [42, 43, 55]. We train it using the PTB dataset [48] 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 [27] with a single fully-connected layer as its cell function. Following the same setting in [27], we train it over synthetic samples generated by their code [47] – 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 [26, 51]. We implement the binary child-sum Tree-LSTM in [45], and train it as a sentiment classifier using Stanford sentiment treebank (SST) dataset [40]. The dataset contains 8544 training sentences, each associated with a human annotated grammar tree, and the longest one has 54 words.

5.1 Overall Performance

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` [6]: 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 [46] based on static declaration; (3) `DyNet`: we implement a 64-step

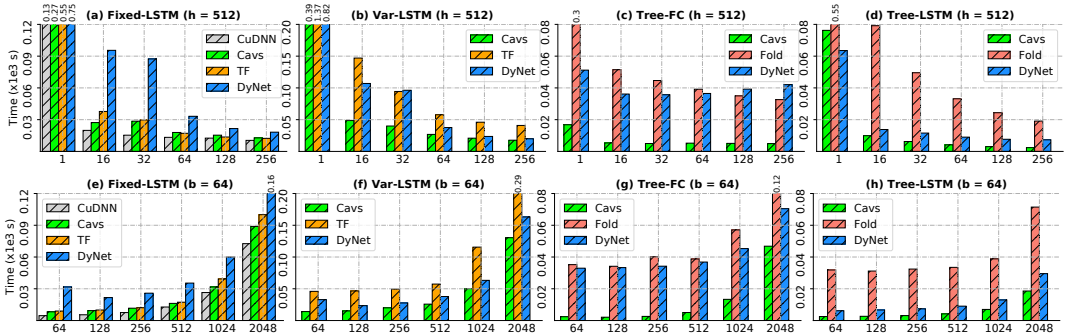

Figure 8: Comparing five systems on the averaged time to finish one epoch of training on four models: `Fixed-LSTM`, `Var-LSTM`, `Tree-FC` and `Tree-LSTM`. In (a)-(d) we fix the hidden size h and vary the batch size bs, while in (e)-(h) we fix bs and vary h.

LSTM in DyNet based on dynamic declaration – we declare a dataflow graph per sample, and train with the autobatching [34] 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 Fig 8(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 handling fully static graphs, though it is specialized for dynamic ones. We also conclude from Fig 8 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 non-batched 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 described in §6; (2) `DyNet`: an official implementation from DyNet benchmark repository based on dynamic declaration [10]; (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 8(b)(f), respectively. Although all three systems perform batched computation in different ways, `Cavs` is consistently 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 8(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 graph preprocessing/construction overhead. With a single CPU thread, `Fold` takes even more time on graph preprocessing than computation (§5.3).

Finally, we compare three frameworks on `Tree-LSTM` in Figure 8(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 54 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, which we will discuss in §5.3. We also compare Cavs with PyTorch – its per-epoch time on Tree-LSTM is 542s, 290x slower than Cavs when the batch size is 256. Compared to other systems, PyTorch cannot batch the execution of dynamic NNs.

5.2 Graph Construction and Computation

In this section, we investigate the graph construction overhead in Fold and DyNet. To batch computation of different graphs, Fold analyzes the input graphs to recognize batch-able dynamic operations, then translates them into intermediate instructions, with which, TensorFlow generates appropriate control flow graphs for evaluation – we will treat the overhead caused in both steps as

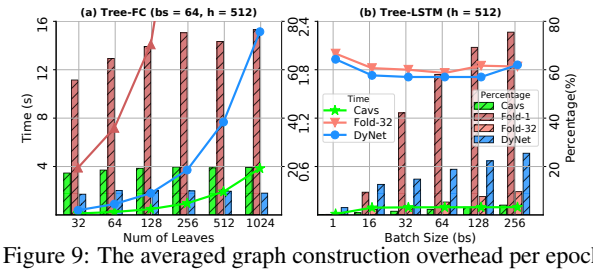

Figure 9: The averaged graph construction overhead per epoch when training (a) `Tree-FC` with different size of input graphs (b) `Tree-LSTM` with different batch size. The curves show absolute time in second (left y-axis), and the bar graphs show its percentage of the overall time (right y-axis).

Fold's graph construction overhead. DyNet, as a typical dynamic declaration framework, has to construct as many dataflow graphs as the number of samples. Though DyNet has optimized its graph construction to make it lightweight, the overhead still grows with the training set and the size of input graphs. By contrast, Cavs takes constant time to construct a small dataflow graph encoded by $\mathcal{F}$, then reads input graphs through I/O. To quantify the overhead, we separate the graph construction from computation, and visualize in Figure 9(a) how it changes with the average number of leaves (graph size) of input graphs on training `Tree-FC`, with fixed $bs = 64, h = 512$. We compare (1) `Cavs` (2) `Fold-1` which is Fold with one graph processing thread and (3) `DyNet`. We plot for one epoch, both the (averaged) absolute time for graph construction and it percentage of the overall time. Clearly we find that all three systems take increasingly more time when the size of the input graphs grows, but Cavs, which loads graphs through I/O, causes the least overhead at all settings. In terms of the relative time, `Fold` unfortunately wastes 50% at 32 leaves, and 80% when the tree has 1024 leaves, while `DyNet` and `Cavs` take only 10% and 20%, respectively.

We also wonder how the overhead is related with batch size when there is fixed computational workload. We report in Figure 9(b) the same metrics when training `Tree-LSTM` with varying bs. We add another baseline `Fold-32` with 32 threads for Fold's graph preprocessing. As `Fold-1` takes much longer time than others, we report its time at $bs = 1, 16, 32, 64, 128, 256$ here (instead of showing in Figure 9): 1.1, 7.14, 31.35, 40.1, 46.13, 48.77. Except $bs = 1$, all three systems (except `Fold-1`) take almost constant time for graph construction in one epoch, regardless of bs, while `Fold-32` and `DyNet` take similar time, but `Cavs` takes 20x less. Nevertheless, at the percentage scale, increasing bs makes this overhead more prominent, because larger batch size yields improved computational efficiency, therefore less time to finish one epoch. This, from one perspective, reflects that the graph construction is a main obstacle that grows with the number of training samples and prevents the efficient training of dynamic NNs in existing frameworks, while Cavs successfully overcomes this barrier.

# leaves	time (s)	Speedup	bs	time (s)	Speedup
32	**0.6** / 3.1 / 4.1	**5.4** / 7.1	1	76 / 550 / **62**	**7.2** / 0.8
64	**1.1** / 3.9 / 8.0	3.7 / 7.5	16	**9.8** / 69 / 12	7.0 / 1.2
128	**2** / 6.2 / 16	3.0 / 7.9	32	**6.2** / 43 / 9.9	7.0 / 1.6
256	**4** / 10.6 / 33.7	2.7 / 8.7	64	**4.1** / 29 / 7.4	**7.2** / 1.8
512	**8** / 18.5 / 70.6	2.3 / 8.9	128	**2.9** / 20.5 / 5.9	7.1 / 2.0
1024	**16** / 32 / 153	2.1 / **9.7**	256	**2.3** / 15.8 / 5.4	7.0 / **2.4**

Table 1: The averaged computation time (`Cavs`/`Fold`/`DyNet`) and the speedup (`Cavs` vs `Fold`/`DyNet`) for training one epoch on `Tree-FC` with varying size of the input trees (left part), and on `Tree-LSTM` with varying batch size (right part).

Apart from the graph construction we report in Table 1 the computation-only time. `Cavs` shows maximally 5.4x/9.7x and 7.2x/2.4x speedups over `Fold`/`DyNet` on `Tree-FC` and `Tree-LSTM`, respectively. The advantages stem from two main sources: an optimized graph execution engine, and a better-suited memory management strategy, which we investigate next.

5.3 Optimizations

Graph Execution Engine. To reveal how much each optimization in §3.4 contributes to the final performance, we disable lazy batching, fusion and streaming in Cavs and set this configuration as a baseline (speedup = 1). We then turn on one optimization at a time and record how much speedup it brings. We train `Fixed-LSTM` and `Tree-LSTM`, and report the averaged speedups one computation-only time in one epoch over the baseline configuration in Fig 10, with $bs = 64$ but varying h. Lazy batching and fusion consistently deliver nontrivial improvement – lazy batching is more beneficial with a larger h while fusion is more effective at smaller h, which are expected: lazy batching mainly parallelizes matrix-wise operations (e.g. `matmul`) commonly with $O(h^2)$ or higher complexity, while fusion mostly works on elementwise operations with $O(h)$ complexity [19].

Streaming, compared to the other strategies, is less effective on `Tree-LSTM` than on `Fixed-LSTM`, as we have found the depth of the input trees in SST exhibit high variance, i.e. some trees are much deeper than others. In this case, many batching tasks only have one vertex to be evaluated. The computation is highly fragmented and the efficiency is bounded by kernel launching latency. Lazy batching and fusion still help as they both reduce kernel launches (§3.4). Streaming, which tries to pipeline multiple kernels, can hardly yield obvious improvement.

Memory Management. Cavs' performance advantage also credits to its memory management that reduces memory movements while guarantees continuity. Quantitatively, it is difficult to compare Cavs to Fold, as Fold relies on TensorFlow where memory management is highly coupled with other system aspects. Qualitatively, we find Cavs requires less memory movement (e.g. memcpy) during dynamic batching. Built upon the `tf_while` operator, whenever Fold performs depth-

Programming Model	Frameworks	Expressiveness	Batching	Graph Construction Overhead	Graph Optimization
static declaration	Caffe, TensorFlow	×	√	low	beneficial
dynamic declaration (eager evaluation)	PyTorch, Chainer	√	×	N/A	unavailable
dynamic declaration (lazy evaluation)	DyNet	√	√	high	limited benefits
Fold	TensorFlow-Fold	√	√	high	unknown
Vertex-centric	Cavs	√	√	low	beneficial

Table 2: A side-by-side comparison of existing programming models for dynamic NNs, and their advantages and disadvantages.

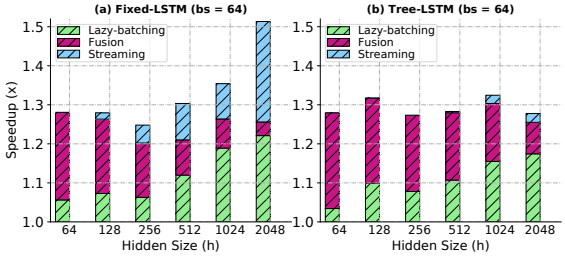

Figure 10: Improvement of each optimization strategy on execution engine over a baseline configuration (speedup = 1).

bs	Memory operations (s) (*Cavs / DyNet*)		Computation (s) (*Cavs / DyNet*)	
	Train	Inference	Train	Inference
16	**1.14** / 1.33	**0.6** / 1.33	**9.8** / 12	**2.9** / 8.53
32	**0.67** / 0.87	**0.35** / 0.87	**6.1** / 9.8	**1.9** / 5.35
64	**0.39** / 0.6	**0.21** / 0.6	**4.0** / 7.4	**1.3** / 3.48
128	**0.25** / 0.44	**0.13** / 0.44	**2.9** / 5.9	**0.97** / 2.52
256	**0.17** / 0.44	**0.09** / 0.44	**2.3** / 5.4	**0.77** / 2.58

Table 3: Breakdowns of average time per epoch on memory-related operations and computation, comparing Cavs to DyNet on training and inference of `Tree-LSTM` with varying *bs*.

based batching at depth d, it has to move all the contents of nodes in the dataflow graphs at depth $d-1$ to a desired location, as the control flow does not support cross-depth memory indexing. This results in redundant memcpy, especially when the graphs are highly skewed. By contrast, Cavs only copies contents that are necessary to the batching task. DyNet has a specialized memory management strategy for dynamic NNs. Compared to Cavs, it however suffers substantial overhead caused by repeated checks of the memory continuity – whenever DyNet wants to batch operators with same signatures, it checks whether their inputs are continuous on memory [34]. The checking overhead increases with *bs* and is more prominent on GPUs. Thanks to the simplicity of both systems, we are able to profile the memory-related overhead during both training and inference, and separate it from computation. We compare them on `TreeLSTM`, and report the breakdown time per epoch in Table 3 under different *bs*. We observe the improvement is significant (2x - 3x) at larger *bs*, especially during inference where DyNet has its continuity checks concentrated.

6 Related Work

DL programming models. In addition to §2.2, we summarize in Table 2 the major programming models and frameworks for dynamic NNs, and their pros and cons, in contrast to Cavs. Within static frameworks, there are also efforts on adapting static declaration to support sequence RNNs, such as *static unrolling* [17], *bucketing* [15] and *dynamic unrolling* [16]. The ideas are to pad zero at the end of samples so that they have the same structure (i.e. same length) for batched computation. However, they all result in unnecessary computation and can not express more complex structures than sequences. Asynchronous model-parallelism [13] enables the concurrent execution of different graphs similar to batched execution in Cavs, it however may suffer from insufficient cache re-usage and overhead by multiple kernel launches (on GPUs).

Execution optimization. A variety of developed techniques from other areas (e.g. kernel fusion, constant folding) have been adapted to speed the computation of DL dataflow graphs [1, 5, 12, 18]. Cavs separates the static vertex function from the dynamic-varying input graph, so it benefits from most of the aforementioned optimizations. We learn from these strategies and reflect them in Cavs' execution engine. We further propose lazy batching and concurrent execution to exploit more parallelism exposed by our APIs.

Graph-based systems. The vertex-centric programming model has been extensively developed in graph computing [29, 14, 4, 41]. Cavs draws insights from the GAS model [14], but is fundamentally different: `gather` and `scatter` in Cavs are fully symbolic – they allow back-propagation through them; graph computing systems compute on large natural graphs, while Cavs addresses problems that each sample has a unique graph and the training is iterative on batches of samples. In terms of system design, Cavs also faces different challenges, such as scheduling for batched execution of different graphs, guaranteeing the memory continuity. There are also some graph-based ML systems, such as GraphLab [28], but they do not handle instance-based graphs, and do not offer batching advantages for dynamic DL workloads.

7 Conclusion

We present Cavs, an efficient system for dynamic neural networks. With a novel representation, designed scheduling policy, memory management strategy, and graph execution optimizations, Cavs avoids substantial graph construction overhead, allows for batched computation over different structured graphs, and can benefit from well-established graph optimization techniques. We compare Cavs to state-of-the-art systems for dynamic NNs and report a near one order of magnitude speedup across various dynamic NN architectures and settings.

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] BAHDANAU, D., CHO, K., AND BENGIO, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[3] 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).

[4] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 1.

[5] 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).

[6] CHETLUR, S., WOOLLEY, C., VANDERMERSCH, P., COHEN, J., TRAN, J., CATANZARO, B., AND SHELHAMER, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[7] CUI, H., ZHANG, H., GANGER, G. R., GIBBONS, P. B., AND XING, E. P. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 4.

[8] DAVE, C., BAE, H., MIN, S.-J., LEE, S., EIGENMANN, R., AND MIDKIFF, S. Cetus: A source-to-source compiler infrastructure for multicores. *Computer 42*, 12 (2009).

[9] DONAHUE, J., ANNE HENDRICKS, L., GUADARRAMA, S., ROHRBACH, M., VENUGOPALAN, S., SAENKO, K., AND DARRELL, T. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 2625–2634.

[10] DYNET VARIABLE LENGTH LSTM. `https://github.com/neulab/dynet-benchmark`.

[11] FACEBOOK. `http://pytorch.org/`.

[12] FACEBOOK OPEN SOURCE. Caffe2 is a lightweight, modular, and scalable deep learning framework. `https://github.com/caffe2/caffe2`, 2017.

[13] GAUNT, A., JOHNSON, M., RIECHERT, M., TARLOW, D., TOMIOKA, R., VYTINIOTIS, D., AND WEBSTER, S. Ampnet: Asynchronous model-parallel training for dynamic neural networks. *arXiv preprint arXiv:1705.09786* (2017).

[14] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs.

[15] GOOGLE. Tensorflow bucketing. `https://www.tensorflow.org/versions/r0.12/api_docs/python/contrib.training/bucketing`.

[16] GOOGLE. Tensorflow dynamic rnn. `https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn`.

[17] GOOGLE. Tensorflow static rnn. `https://www.tensorflow.org/api_docs/python/tf/nn/static_rnn`.

[18] GOOGLE TENSORFLOW XLA. `https://www.tensorflow.org/performance/xla/`.

[19] GUSTAFSON, J. L. Reevaluating amdahl's law. *Communications of the ACM 31*, 5 (1988), 532–533.

[20] GYSI, T., OSUNA, C., FUHRER, O., BIANCO, M., AND SCHULTHESS, T. C. 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* (2015), IEEE, pp. 1–12.

[21] HINTON, G., DENG, L., YU, D., DAHL, G. E., MOHAMED, A.-R., JAITLY, N., SENIOR, A., VANHOUCKE, V., NGUYEN, P., SAINATH, T. N., ET AL. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine 29*, 6 (2012), 82–97.

[22] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[23] JIA, Y., SHELHAMER, E., DONAHUE, J., KARAYEV, S., LONG, J., GIRSHICK, R., GUADARRAMA, S., AND DARRELL, T. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093* (2014).

[24] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS* (2012).

[25] 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).

[26] LIANG, X., SHEN, X., FENG, J., LIN, L., AND YAN, S. Semantic object parsing with graph lstm. In *European Conference on Computer Vision* (2016), Springer, pp. 125–143.

[27] LOOKS, M., HERRESHOFF, M., HUTCHINS, D., AND NORVIG, P. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181* (2017).

[28] LOW, Y., GONZALEZ, J. E., KYROLA, A., BICKSON, D., GUESTRIN, C. E., AND HELLERSTEIN, J. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

[29] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[30] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[31] MITCHELL, D. C. Sentence parsing. *Handbook of psycholinguistics* (1994), 375–409.

[32] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 439–455.

[33] 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).

[34] NEUBIG, G., GOLDBERG, Y., AND DYER, C. On-the-fly operation batching in dynamic computation graphs. *arXiv preprint arXiv:1705.07860* (2017).

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

[36] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch.

[37] QUINLAN, D. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters 10*, 02n03, 215–226.

[38] RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., AND AMARASINGHE, S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices 48*, 6 (2013), 519–530.

[39] SIMONYAN, K., AND ZISSERMAN, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR* (2015).

[40] 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.

[41] SUNDARAM, N., SATISH, N., PATWARY, M. M. A., DULLOOR, S. R., ANDERSON, M. J., VADLAMUDI, S. G., DAS, D., AND DUBEY, P. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1214–1225.

[42] SUNDERMEYER, M., SCHLÜTER, R., AND NEY, H. Lstm neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association* (2012).

[43] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (2014), pp. 3104–3112.

[44] SZEGEDY, C., LIU, W., JIA, Y., SERMANET, P., REED, S., ANGUELOV, D., ERHAN, D., VANHOUCKE, V., RABINOVICH, A., ET AL. Going deeper with convolutions.

[45] 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).

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

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

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

[49] Tokui, S., Oono, K., Hido, S., and Clayton, J. 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), vol. 5.

[50] Tokui, S., Oono, K., Hido, S., and Clayton, J. 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).

[51] Vinyals, O., Kaiser, Ł., Koo, T., Petrov, S., Sutskever, I., and Hinton, G. Grammar as a foreign language. In *Advances in Neural Information Processing Systems* (2015), pp. 2773–2781.

[52] Walt, S. v. d., Colbert, S. C., and Varoquaux, G. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (2011), 22–30.

[53] Yan, Z., Zhang, H., Jagadeesh, V., DeCoste, D., Di, W., and Piramuthu, R. Hd-cnn: Hierarchical deep convolutional neural network for image classification. *ICCV* (2015).

[54] Yan, Z., Zhang, H., Wang, B., Paris, S., and Yu, Y. Automatic photo adjustment using deep neural networks. *ACM Transactions on Graphics (TOG) 35*, 2 (2016), 11.

[55] Zaremba, W., Sutskever, I., and Vinyals, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[56] Zhang, H., Hu, Z., Wei, J., Xie, P., Kim, G., Ho, Q., and Xing, E. Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216* (2015).

[57] Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., Hu, Z., Wei, J., Xie, P., and Xing, E. P. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 181–193.