Cavs: An Efficient Runtime System for Dynamic Neural Networks

Hao Zhang*

Shizhen Xu*, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng
Qirong Ho, Guangwen Yang, Eric P. Xing
Carnegie Mellon University and Petuum Inc.
* indicates equal contributions
Outline

• Deep learning and dataflow graphs
• Dynamic neural network and programming models
• Cavs: a new programming interface for dynamic NNs
A layer in a neural network is composed of a few finer computational operations, which can be represented as a forward pass through a dataflow graph.

Training the layer parameters involves deriving the gradients of its parameters -- a backward pass where the gradients flow through a backward dataflow graph representation of the layer.

Given forward dataflow graph, the backward graph can be automatically derived by auto-differentiation.
A Neural Network as a Dataflow Graph

- Define a neural network \(\sim\) assemble a dataflow graph
  - Define operations and layers: fully-connected? Convolution?
  - Define data I/O: what data to read? Where?
  - Define a loss functions: L2 loss? Softmax?
  - Define an optimization solver: SGD, Momentum, Adam, etc.
  - Connect operations, data I/O, loss functions and optimizer as a full dataflow graph, which is the representation of the neural network

Auto-differentiation Libraries (e.g. Caffe, TensorFlow) then take over
  - Automatically derive the backward graphs
  - Perform training (forward-backward passes) and apply updates
A Neural Network as a Dataflow Graph

forward
backward

Photo from TensorFlow websi
A Programming Model: Static Declaration

- Users declare a dataflow graph
- Frameworks analyze and optimize the graph
  - Automatically derive the backward graph based on autodiff
  - Incorporate some graph-level optimization if desired
- Perform training/inference iteratively

```c
/* (a) static declaration */
// all samples must share one graph
declare a static data flow graph \( \mathcal{D} \).
for \( t = 1 \rightarrow T \):
    read the \( t \)th data batch \( \{ x_i^t \}_{i=1}^K \).
    batched computation: \( \mathcal{D}(\{ x_i^t \}_{i=1}^K) \).
```

Incorporate graph-level optimization over \( \mathcal{D} \) (optionally)
Static Declaration: Advantages

- Static Declaration is the dominant choice for DL
  - Good for static workflows: define once, run for arbitrary batches/data
  - All samples compute over one graph, therefore the computation can be “by-nature” batched – by leveraging GPU and other advanced matrix-computing libs (CUDA, etc.)
  - Easy to optimize: a lot of off-the-shelf optimization techniques for dataflow graph

```c
/* (a) static declaration */
// all samples must share one graph

declare a static data flow graph D.

for t = 1 → T:
    read the tth data batch \{x_i^t\}_{i=1}^K.
    batched computation: D(\{x_i^t\}_{i=1}^K).
```

Incorporate graph-level optimization over D (optionally)
Introduction to Dynamic Neural Networks

• Deep Learning has been applied on more structured data
• The neural network computes following a data-dependent structure, in order to encode the structure information
  • Hence, The NN architecture used to handle structured data would change with the input sample
• E.g. Recurrent Neural Networks and their variants
  • Sequence RNN in machine translation, video understanding
  • Tree RNN in sentence parsing and sentiment analysis
  • GraphRNN in social network/image segmentation
Dynamic Neural Network: An Example

• 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.
Static Declaration for Dynamic Dataflow Graphs

• Can we handle **dynamic** dataflow graphs using **static** declaration?
  • **Static unroll**: preprocessing all inputs to have the same length
  • **Bucketing**: put inputs into different buckets, one bucket one NN
  • At the core of the above tricks is to pad the inputs with zeros so they have the same shape/length

• They are **very commonly adopted**, but are they good?
  • Unable to express structures beyond sequences
  • Usually result in unnecessary (extra) computation, which wastes computational resources
  • Complexity in implementation
An Extended Model: Dynamic Declaration

- **Key idea**: declare and construct a dataflow graph for each input sample
  - Move the graph declaration and construction (and optimization) from outside of the loop to inside the loop
  - Perform single instance training because it is hard to batch

/* (a) static declaration */
// all samples must share one graph
```cpp
declare a static data flow graph $\mathcal{D}$. 
for $t = 1 \rightarrow T$:
    read the $t$th data batch $\{x^t_i\}_{i=1}^K$.
    batched computation: $\mathcal{D}(\{x^t_i\}_{i=1}^K)$.
```

/* (b) dynamic declaration */
```cpp
for $t = 1 \rightarrow T$:
    read the $t$th data batch $\{x^t_i\}_{i=1}^K$.
    for $k = 1 \rightarrow K$:
        declare a data flow graph $\mathcal{D}^t_i$ for $x^t_i$.
        single-instance computation: $\mathcal{D}^t_i(x^t_i)$
```

- DL Frameworks based on dynamic declaration have gained substantial popularity in the most recent 2 years
Dynamic Declaration: Pros and Cons

• Dynamic declaration has one major advantage
  • Flexibility: it can express arbitrarily dynamically networks structures by declaring as many as dataflow graphs as the number of training data

• Dynamic declaration scarifies efficiency for flexibility

```c
/* (b) dynamic declaration */
for t = 1 → T:
    read the tth data batch \{x_{i}^{t}\}_{i=1}^{K}.
for k = 1 → K:
    declare a data flow graph D_{i}^{t} for x_{i}^{t}.
    single-instance computation: D_{i}^{t}(x_{i}^{t}).
```
Problem #1: Graph Construction Cost

- Graph construction overhead grows linearly with # of samples

```cpp
/* (a) static declaration */
// all samples must share one graph
declare a static data flow graph $D$.
for $t = 1 \rightarrow T$:
  read the $t$th data batch $\{x^t_i\}_{i=1}^K$.
batched computation: $D(\{x^t_i\}_{i=1}^K)$.
```

```cpp
/* (b) dynamic declaration */
for $t = 1 \rightarrow T$:
  read the $t$th data batch $\{x^t_i\}_{i=1}^K$.
  for $k = 1 \rightarrow K$:
    declare a data flow graph $D^t_i$ for $x^t_i$.
    single-instance computation: $D^t_i(x^t_i)$.
```
Problem #1: Graph Construction Cost

- Curve (left axis): absolute time; bar (right): percentage time
- Graph construction takes 80% of overall time in TensorFlow Fold
Problem #2: Batching will be Difficult

- No batching available any more
- Manual batching the execution of differently structured graphs is very difficult
  - Users have to write code to do batching by themselves
  - In fact, until 2017, most papers based on tree-LSTM (a typical dynamic NN) model is trained with batchsize=1

```plaintext
/* (a) static declaration */
// all samples must share one graph
declare a static data flow graph \( \mathcal{D} \).
for \( t = 1 \rightarrow T \):
  read the \( t \)th data batch \( \{ x^t_i \}_{i=1}^K \).
  batched computation: \( \mathcal{D}(\{ x^t_i \}_{i=1}^K) \).

/* (b) dynamic declaration */
for \( t = 1 \rightarrow T \):
  read the \( t \)th data batch \( \{ x^t_i \}_{i=1}^K \).
  for \( k = 1 \rightarrow K \):
    declare a data flow graph \( \mathcal{D}^t_i \) for \( x^t_i \).
    single-instance computation: \( \mathcal{D}^t_i(x^t_i) \).
```

In static declaration: batching is natural
In dynamic declaration: batching is difficult
Problem #3: Unavailable to Graph Optimizations

- In static declaration, we optimize the graph only once,
  - Graph optimization overhead is constant
  - The optimization is beneficial for all input data points
- In dynamic declaration, if we want to incorporate these optimization, we need to optimize for each declared graph
  - Linear graph optimization overhead
- As a result: the optimization might cost more than it can gain

```c
/* (a) static declaration */
// all samples must share one graph
declare a static data flow graph \( D \).
for t = 1 \rightarrow T:
    read the \( t \)th data batch \( \{ x^t_i \}_{i=1}^K \).
    batched computation: \( D(\{ x^t_i \}_{i=1}^K) \).
```

```c
/* (b) dynamic declaration */
for t = 1 \rightarrow T:
    read the \( t \)th data batch \( \{ x^t_i \}_{i=1}^K \).
    for k = 1 \rightarrow K:
        declare a data flow graph \( D^t_i \) for \( x^t_i \).
        single-instance computation: \( D^t_i(x^t_i) \).
```

Graph optimization happens here: outside of the loop
Graph optimization happens here: inside the loops!
Introducing Cavs: Design Goals

• Simple Interface, rich expressiveness
  • Keep the flexibility of dataflow graph and dynamic declaration
• At the same time, address the three aforementioned problems:
  • Minimize graph construction overhead
  • Allow for efficient computation and batching
  • (Re-)open the opportunities for graph optimization techniques
Cavs: Motivation

- Observation: Most dynamic NNs have recurrent/recursive structures
- The dynamics come from the sample-dependent structure instead of the "neural network" model itself
Cavs: A New Representation

- Cavs introduces a novel representation for dynamic NNs, and decompose a dynamic NN as two modules
  - A vertex function \( F \), which is static;
  - An input graph \( G \), which is data-dependent and dynamic;
- Hence, Cavs separates out static ML model from the data-dependent dynamics which come from input samples
Cavs: A Vertex-centric Representation

- **Programming:** think like a vertex
  - User implements a vertex function $F$, specifying how a node will interact with its neighboring nodes
  - The system reads the input graph $G$ through I/O
  - The system will compile the local vertex function and figure out the overall computing pattern of the NN over the whole graph
Cavs: Four APIs

- **Gather & Scatter** for internal data path
- **Pull & Push** for external data path
Cavs: Four APIs

• An example: expressing Tree-LSTM using the four APIs

```python
def F():
    S = gather()  # gather states of child vertices
    for k in range(N):
        c_k, h_k = split(S[k], 2)  # get hidden states c and h
        x = pull({0})  # pull the first external input x

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

    scatter(concat([c, h], 1))  # scatter c, h to parent vertices
    push(h)  # push to external connectors
```
Expressing Backpropagation

• The forward and backward passes in Cavs
  • Forward: schedule the execution of the vertex function $F$ through a batch of input graphs following the dependencies therein (e.g. from leaves to roots in trees)
  • Backward: schedule the execution of $\partial F$ through the same batch of input graphs, in a reverse order (e.g. from roots to trees)
Cavs Bypasses Graph Construction Overhead

• No repeated graph construction overhead!
  • The graph construction overhead is constant – we only need to construct $F$, which is usually a small-scale dataflow graph
  • Bypass the repeated dataflow graph construction
  • Instead, read the input graph $G$, which could be achieved by an I/O function

```c
/* (c) our proposed vertex-centric model */
declar a symbolic vertex function $F$.
for $t = 1 \rightarrow T$:
  read the $t$th data batch $\{x^t_i\}_{i=1}^K$.
  read their associated graphs $\{G^t_i\}_{i=1}^K$.
compute $F$ over $\{G^t_i\}_{i=1}^K$ with inputs $\{x^t_i\}_{i=1}^K$.
```

Declare only once $\rightarrow$ constant graph construction cost

Read through I/O, no graph construction involved any more.
Empirical Results: Graph Construction Cost

- Cavs has constant graph construction overhead
- Curve (left axis): absolute time; bar (right): percentage time
- In terms of graph construction overhead, Cavs outperforms TensorFlow-Fold and DyNet by a large margin
Cavs Enables Batched Computation

• Recall the Dynamic Declaration problem #2
• Batched computation on dynamic graphs are difficult
  • Difficult to find batching opportunities
    • Only same operations with exactly the same size of inputs/outputs can be batched
    • Need either manual batching or heavy graph analysis (NP-hard)
  • Strict requirements on memory layouts
    • For the batched computation to be efficient, their input/output need to coalesce on memory
    • How to efficiently re-arrange memory layout to guarantee continuity?
Cavs Enables Batched Computation

- Batched computation is natural and automatic in Cavs
  - Cavs transforms the backpropagation as **evaluating F at a batch of input graphs**

- Then, batched computation can be realized by a simple policy
  - Figure out a set of vertices that we are ready to evaluate F on
  - Batch the evaluation of F on this set of vertices
  - Pass the output of F to their parent vertices

- See the figure below
  - Vertices with same colors are batched evaluated.
Dynamic Batching: Memory Management Challenge

• Batched computational kernels on CPU/CPUs requires the inputs to a batched computation kernel locate continuously on memory
  • e.g. gemm kernels
  • In Dynamic Declaration, this is usually not the case due to the dynamic-varying input structures.
  • To achieve memory continuity, one has to frequently re-arrange memory layouts (memcpy) of the inputs to each batched operation.

• Cavs proposes a new data structure, DynamicTensor, to ensure memory continuity, at the same time minimize memory movement overhead

```cpp
struct DynamicTensor {
  vector<int> shape;
  int bs;
  int offset;
  void* p;
};
```
Cavs: Advanced Memory Management – Dynamic Tensor

- With dynamic tensors, Cavs designs a memory management mechanism to guarantee the coalesce of input contents of batched operations on memory.
Cavs: Improvement on Memory Management

- The improvement is significant (2x - 3x) at larger batch size, comparing to DyNet (a state-of-the-art framework for dynamic NNs).

<table>
<thead>
<tr>
<th>bs</th>
<th>Memory operations (s) (Cavs / DyNet)</th>
<th>Computation (s) (Cavs / DyNet)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Train</td>
<td>Inference</td>
</tr>
<tr>
<td>16</td>
<td>1.14 / 1.33</td>
<td>0.6 / 1.33</td>
</tr>
<tr>
<td>32</td>
<td>0.67 / 0.87</td>
<td>0.35 / 0.87</td>
</tr>
<tr>
<td>64</td>
<td>0.39 / 0.6</td>
<td>0.21 / 0.6</td>
</tr>
<tr>
<td>128</td>
<td>0.25 / 0.44</td>
<td>0.13 / 0.44</td>
</tr>
<tr>
<td>256</td>
<td>0.17 / 0.44</td>
<td>0.09 / 0.44</td>
</tr>
</tbody>
</table>
Cavs is Open to Graph Optimization

- Incorporating graph-level optimization in Cavs is the same as it in static declaration
  - Optimize the static vertex function $F$
  - $F$ will be evaluated at each vertex of the input structure
  - Optimize once, benefit elsewhere

/* (c) our proposed vertex-centric model */
declare a symbolic vertex function $\mathcal{F}$.
\begin{verbatim}
for $t = 1 \rightarrow T$:
  read the $t$th data batch $\{x_i^t\}_{i=1}^K$.
  read their associated graphs $\{G_i^t\}_{i=1}^K$.
  compute $\mathcal{F}$ over $\{G_i^t\}_{i=1}^K$ with inputs $\{x_i^t\}_{i=1}^K$.
\end{verbatim}

Graph optimization happens here: outside of the loops
Cavs Exposes Opportunities for Graph Optimization

• Cavs proposes/adopts three graph-level optimization strategies
  • Lazy batching
  • Streaming
  • Automatic kernel fusion
How Important is Graph Optimization?

- In static frameworks with static declaration, graph optimization usually yield 2 – 4x speedups depending on the graph size.
  - E.g. TensorFlow XLA, MxNet TVM, etc.
- In Cavs, we observe another 1.5x speedup with graph optimizations
Overall Performance

- Overall, Cavs is 1 – 2 orders of magnitude faster than state-of-the-art systems such as DyNet and TensorFlow-Fold on different dynamic NNs.
Cavs: Improvement on Computation

• When only comparing computation, Cavs shows maximally 5.4x/9.7x and 7.2x/2.4x speedups over Fold/DyNet on Tree-FC and Tree-LSTM, respectively.

• Setting: Tree-FC network, time/epoch (s) with varying number of tree leaves and batchsize

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

<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 (instant eval)</td>
<td>PyTorch, Chainer</td>
<td>✓</td>
<td>×</td>
<td>N/A</td>
<td>unavailable</td>
</tr>
<tr>
<td>dynamic declaration (lazy eval)</td>
<td>DyNet</td>
<td>✓</td>
<td>✓</td>
<td>high</td>
<td>not beneficial</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>
Take-home Messages

• Deep learning has moved from static architectures (CNNs) more and more to dynamic structures

• Static declaration and dynamic declaration are two mostly adopted programming models, but they both have drawbacks
  • Graph construction overhead
  • Difficulty in dynamic batching (most important!)
  • Unavailable to graph optimizations

• Cavs proposes a representation of dynamic NNs that addresses these challenges

• Dynamic neural networks is an interesting field that demands more system research, e.g. new programming models, parallelization strategies, and software frameworks
More and Thanks!

- More technical details and results in paper.
- Code will be released soon, check out at https://github.com/petuum-inc

Thanks!

Q&A