Microsoft
Computational Network Toolkit (CNTK)

A Tutorial Given at NIPS 2015 Workshops

Dong Yu and Xuedong Huang
Microsoft Technology & Research

Contributors So Far

- Amit Agarwal, Eldar Akchurin, Chris Basoglu, Guoguo Chen, Scott Cyphers, Jasha Droppo, Adam Eversole, Brian Guenter, Mark Hillebrand, Ryan Hoens, Xuedong Huang, Zhiheng Huang, Vladimir Ivanov, Alexey Kamenev, Philipp Kranen, Oleksii Kuchaiev, Wolfgang Manousek, Avner May, Bhaskar Mitra, Olivier Nano, Gaizka Navarro, Alexey Orlov, Marko Padmilac, Hari Parthasarathi, Baolin Peng, Alexey Reznichenko, Frank Seide, Michael L. Seltzer, Malcolm Slaney, Andreas Stolcke, Huaming Wang, Yongqiang Wang, Kaisheng Yao, Dong Yu, Yu Zhang, Geoffrey Zweig

Outline

- **Introduction (Xuedong Huang)**
- CNTK Deep Dive (Dong Yu)
- Summary/Q&A (Dong & Xuedong)

What do we know now that we did not know 40 years ago?

BY XUEDONG HUANG, JAMES BAKER, AND RAJ REDDY

A Historical Perspective of Speech Recognition

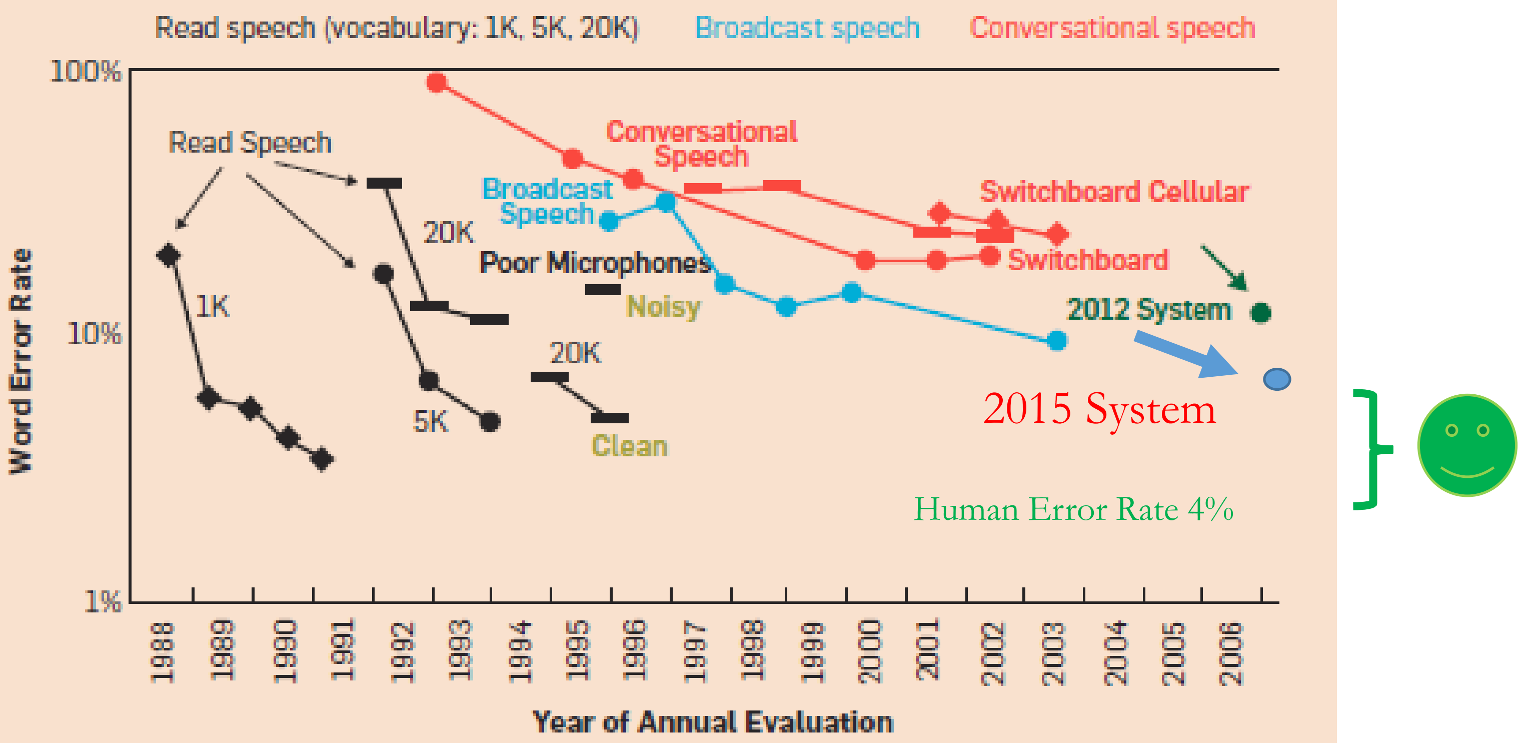
WITH THE INTRODUCTION of Apple's Siri and similar voice search services from Google and Microsoft, it is natural to wonder why it has taken so long for voice recognition technology to advance to this level. Also, we wonder, when can we expect to hear a more human-level performance? In 1976, one of the authors (Reddy) wrote a comprehensive review of the state of the art of voice recognition at that time. A non-expert

» key insights

- The insights gained from the speech recognition advances over the past 40 years are explored, originating from generations of Carnegie Mellon

94 COMMUNICATIONS OF THE ACM | JANUARY 2014 | VOL. 57 | NO. 1

Figure 1. Historical progress of speech recognition word error rate on more and more difficult tasks.[10] The latest system for the switchboard task is marked with the green dot.



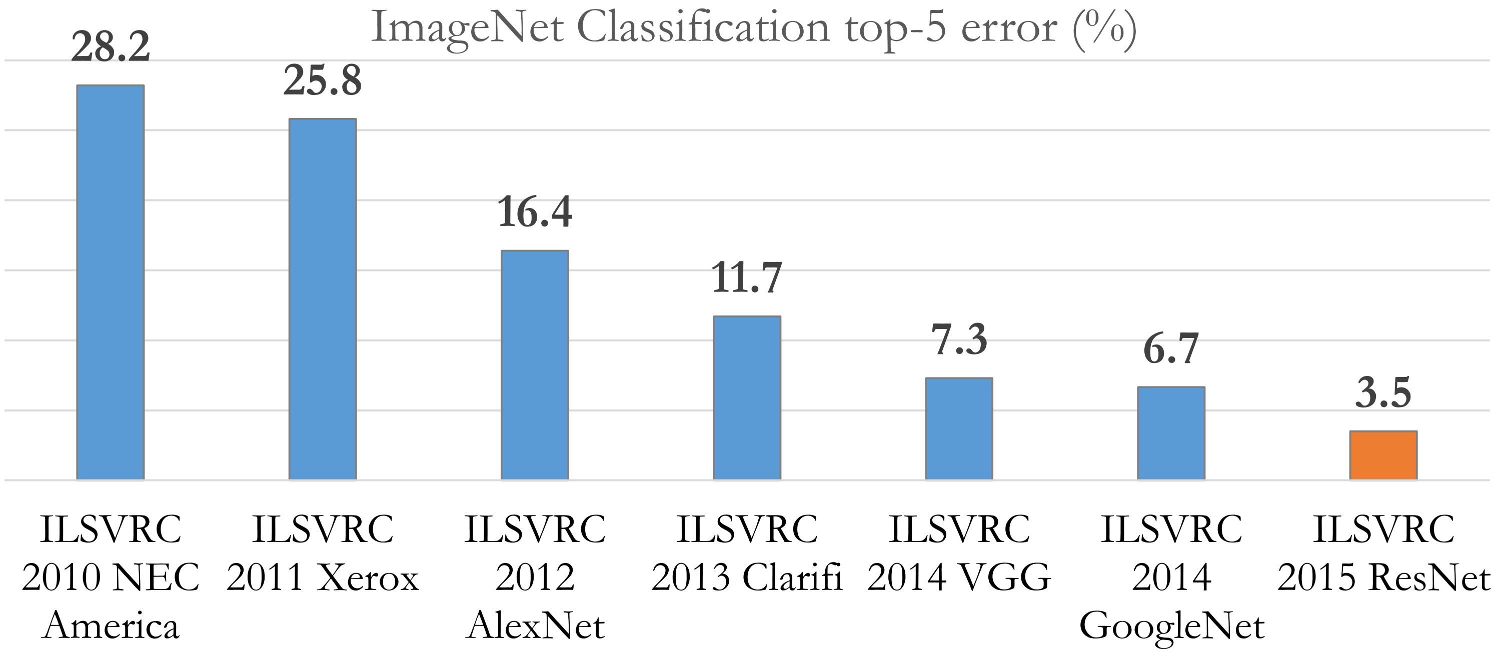
2015 System

Human Error Rate 4%

Microsoft researchers win ImageNet computer vision challenge

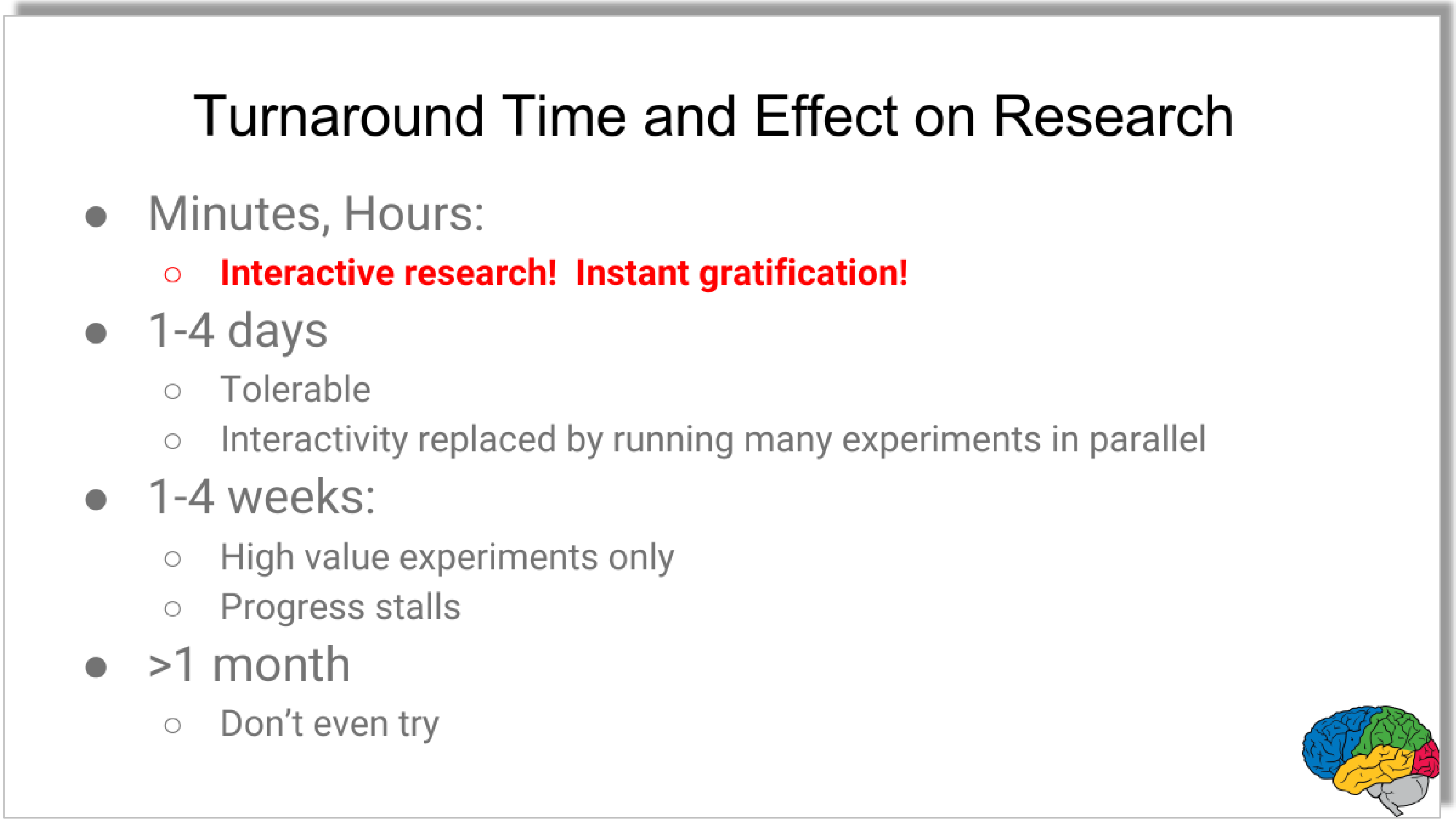
Jian Sun, a principal research manager at Microsoft Research, led the image understanding project. Photo: Craig Tuschhoff/Microsoft.

ImageNet: Microsoft 2015 ResNet



Microsoft had all **5 entries** being the 1-st places this year: ImageNet classification, ImageNet localization, ImageNet detection, COCO detection, and COCO segmentation

We Agree with Jeff Dean's View



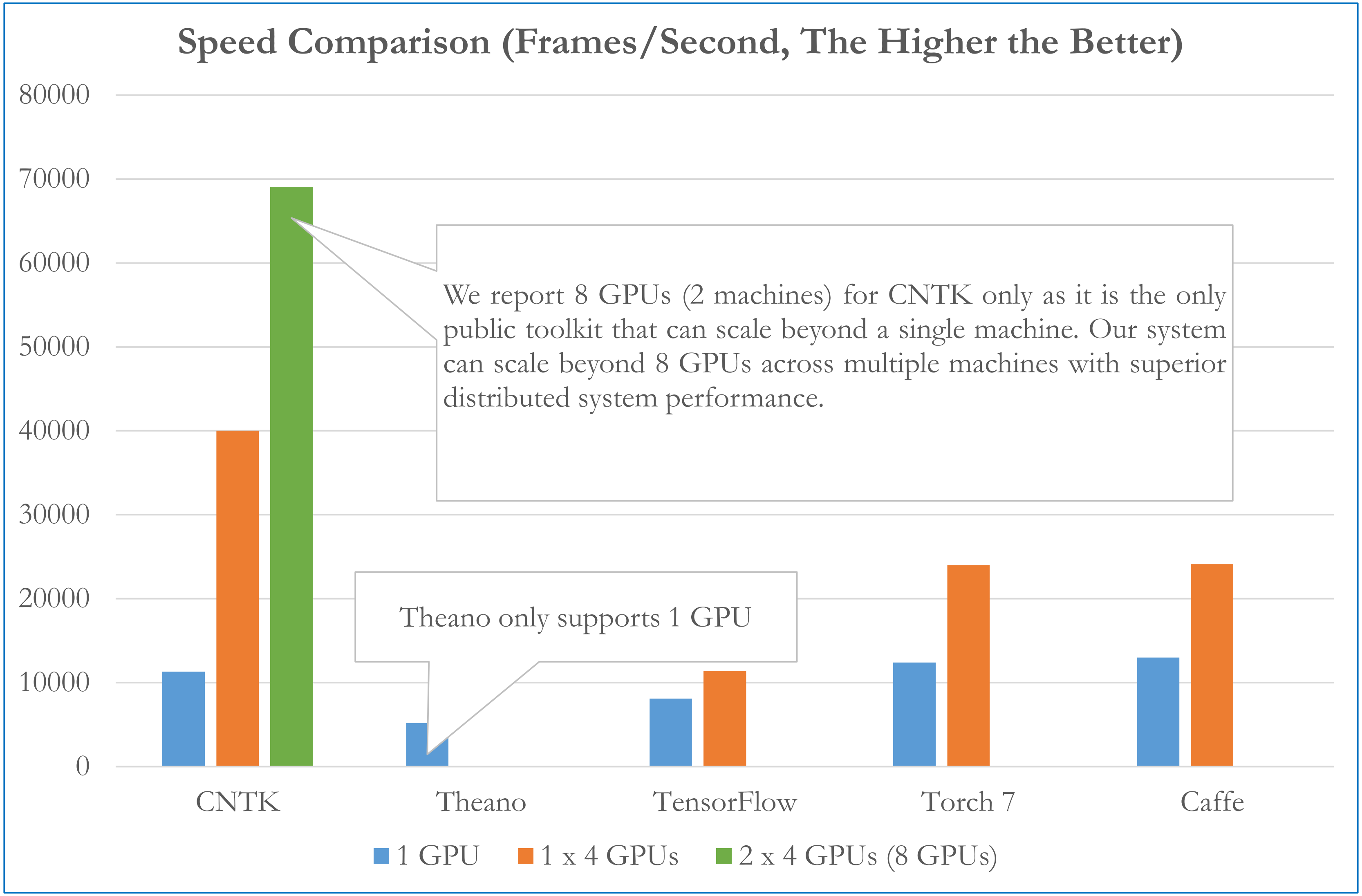
Turnaround Time and Effect on Research

- Minutes, Hours:
  - **Interactive research! Instant gratification!**
- 1-4 days
  - Tolerable
  - Interactivity replaced by running many experiments in parallel
- 1-4 weeks:
  - High value experiments only
  - Progress stalls
- >1 month
  - Don't even try

Azure GPU Lab (Project Philly) - Coming

- High performance deep learning platform on Azure
- Scalable to hundreds of NVIDIA GPUs
- Rapid, no-hassle, deep learning experimentation
- Larger models and training data sets
- Multitenant
- Fault tolerant
- Open source friendly
- CNTK optimized (cuDNN-v4 supported)
- 3rd party accessible (coming)

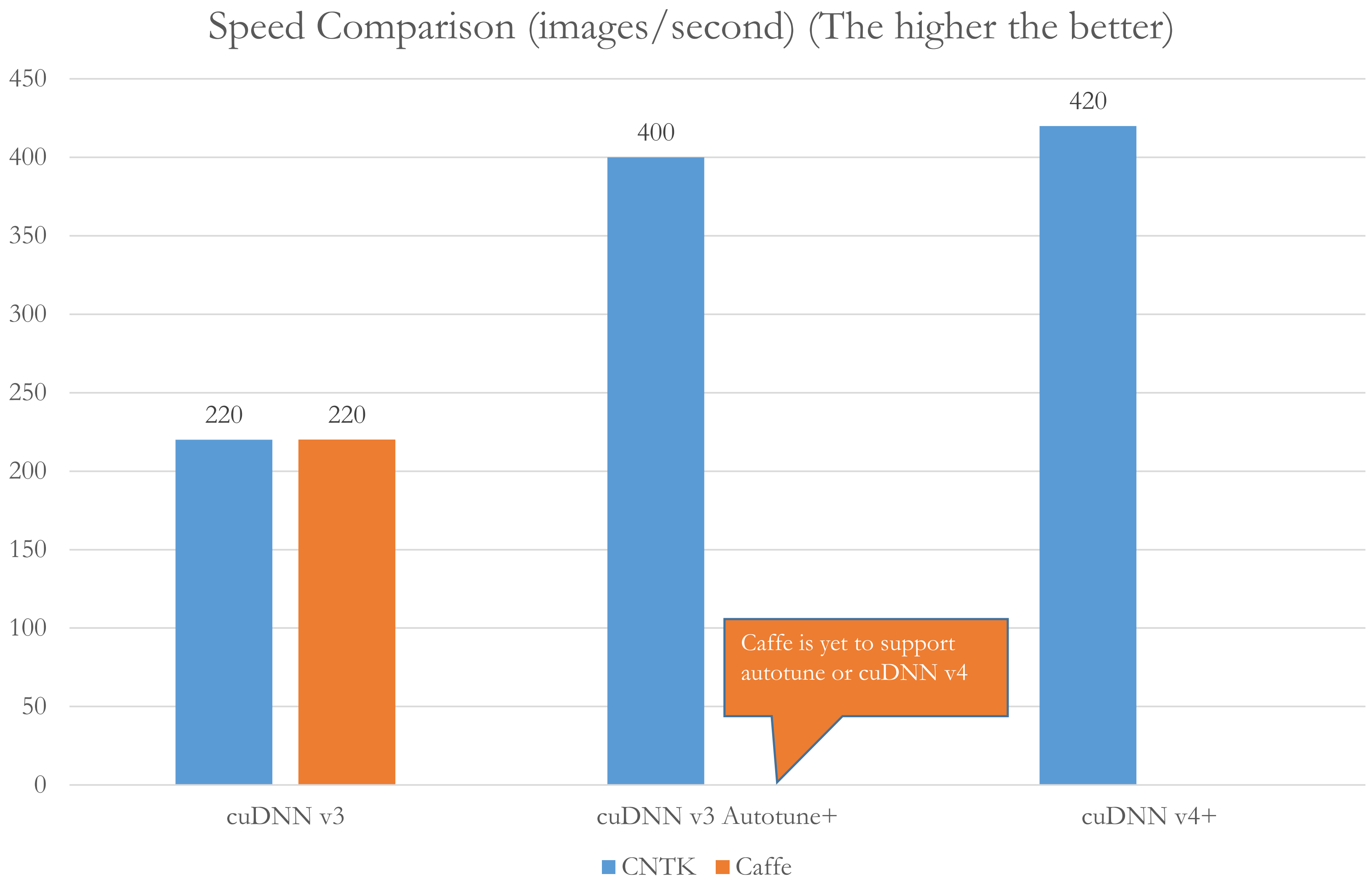
CNTK Computational Performance



We report 8 GPUs (2 machines) for CNTK only as it is the only public toolkit that can scale beyond a single machine. Our system can scale beyond 8 GPUs across multiple machines with superior distributed system performance.

Theano only supports 1 GPU

CNTK Benchmark on CNN



Caffe is yet to support autotune or cuDNN v4

Outline

- Introduction (Xuedong Huang)
- **CNTK Deep Dive (Dong Yu)**
- Summary/Q&A (Dong & Xuedong)

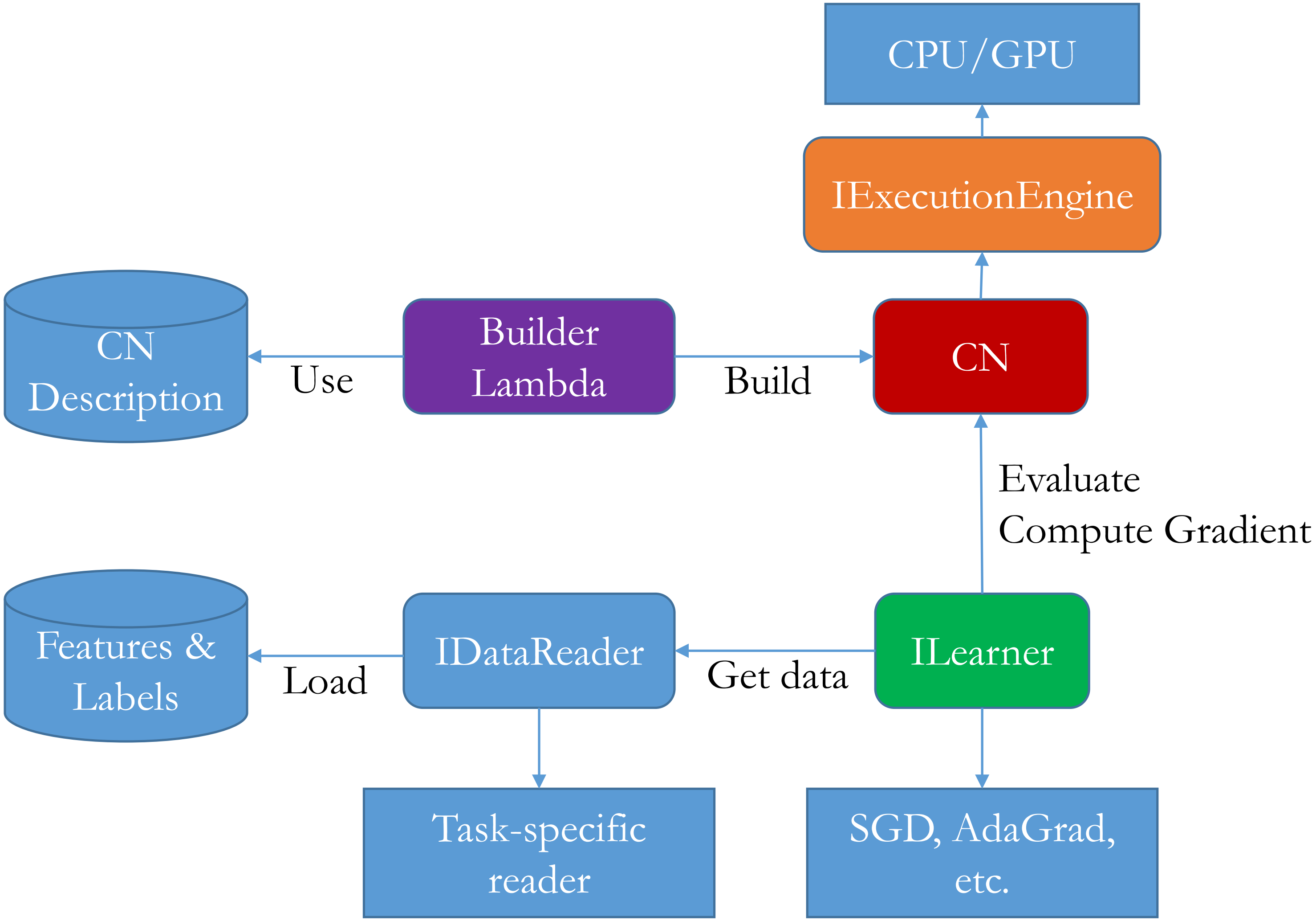
Design Goal of CNTK

- A deep learning tool that balances
  - **Efficiency**: Can train production systems as fast as possible
  - **Performance**: Can achieve state-of-the-art performance on benchmark tasks and production systems
  - **Flexibility**: Can support various tasks such as speech, image, and text, and can try out new ideas quickly
- Inspiration: Legos
  - Each brick is very simple and performs a specific function
  - Create arbitrary objects by combining many bricks
- CNTK enables the creation of existing and novel models by combining simple functions in arbitrary ways.

Functionality

- Supports
  - CPU and GPU with a **focus on GPU Cluster**
  - Windows and Linux
  - automatic numerical differentiation
  - **Efficient static and recurrent network training through batching**
  - **data parallelization within and across machines with 1-bit quantized SGD**
  - memory sharing during execution planning
- Modularized: separation of
  - computational networks
  - execution engine
  - learning algorithms
  - model description
  - data readers
- Models can be described and modified with
  - C++ code
  - Network definition language (NDL) and model editing language (MEL)
  - Brain Script (beta)
  - Python and C# (planned)

CNTK Architecture



Main Operations

- Train a model with the train command
- Evaluate a model with the eval command
- Edit models (e.g., add nodes, remove nodes, change the flag of a node) with the edit command
- Write outputs of one or more nodes in the model to files with the write command

- Finer operation can be controlled through script languages (beta)

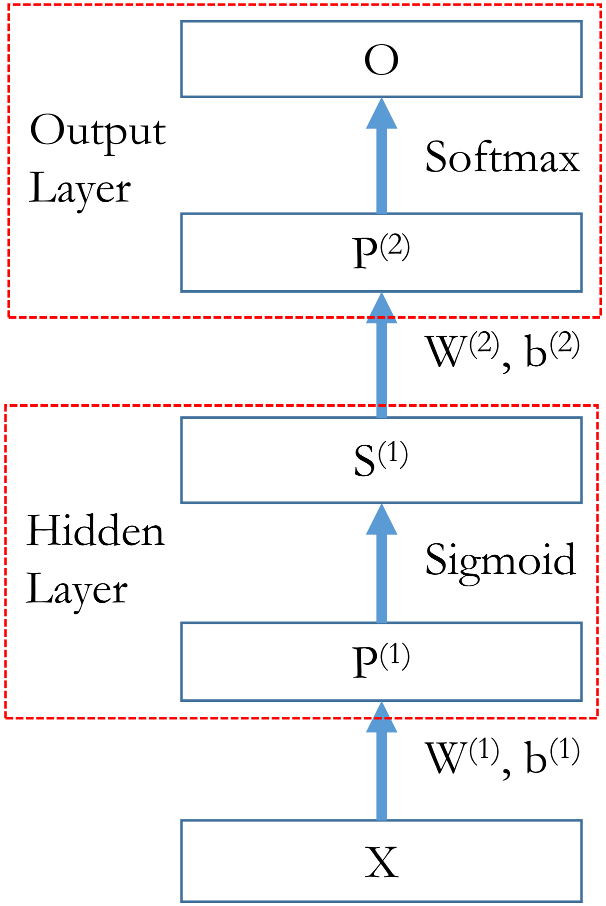
At the Heart: Computational Networks

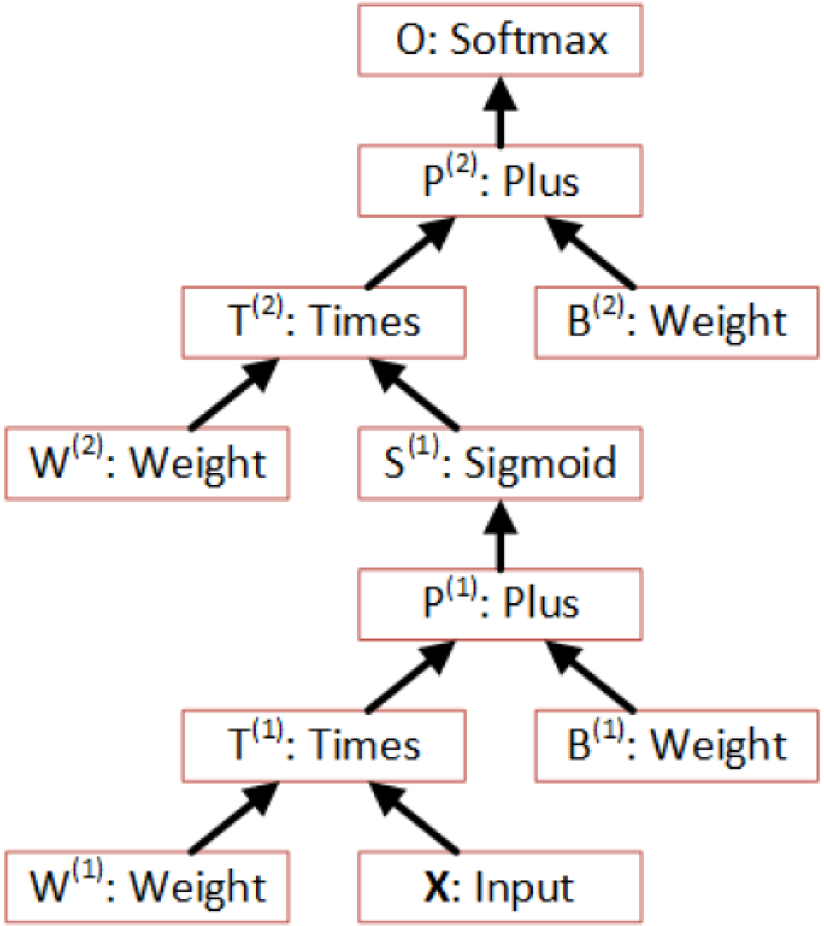
- A generalization of machine learning models that can be described as a series of computational steps.
  - E.g., DNN, CNN, RNN, LSTM, DSSM, Log-linear model
- Representation:
  - A list of computational nodes denoted as

    n = {node name : operation name}
  - The parent-children relationship describing the operands

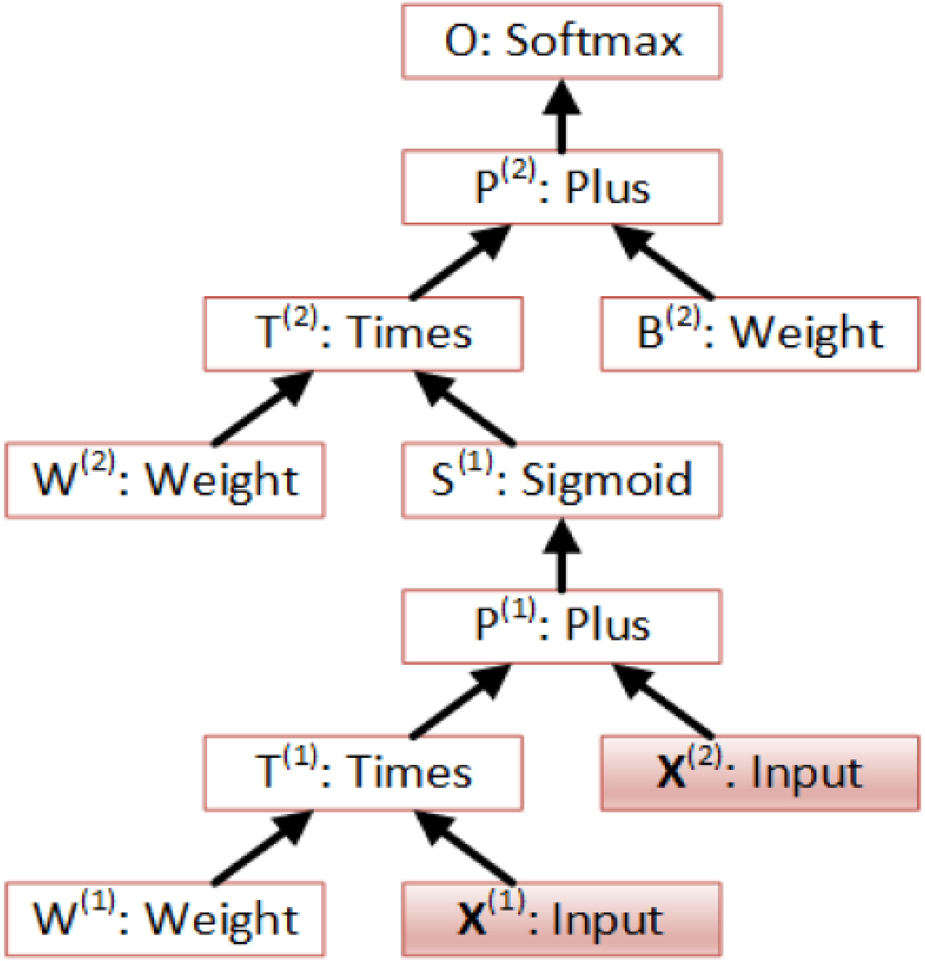
    $\{n : c_1, \cdot \cdot \cdot , c_{Kn} \}$
    - Kn is the number of children of node n. For leaf nodes Kn = 0.
    - Order of the children matters: e.g., XY is different from YX
  - Given the inputs (operands) the value of the node can be computed.
- **Can flexibly describe deep learning models.**
  - Adopted by many other popular tools as well

Example: One Hidden Layer NN

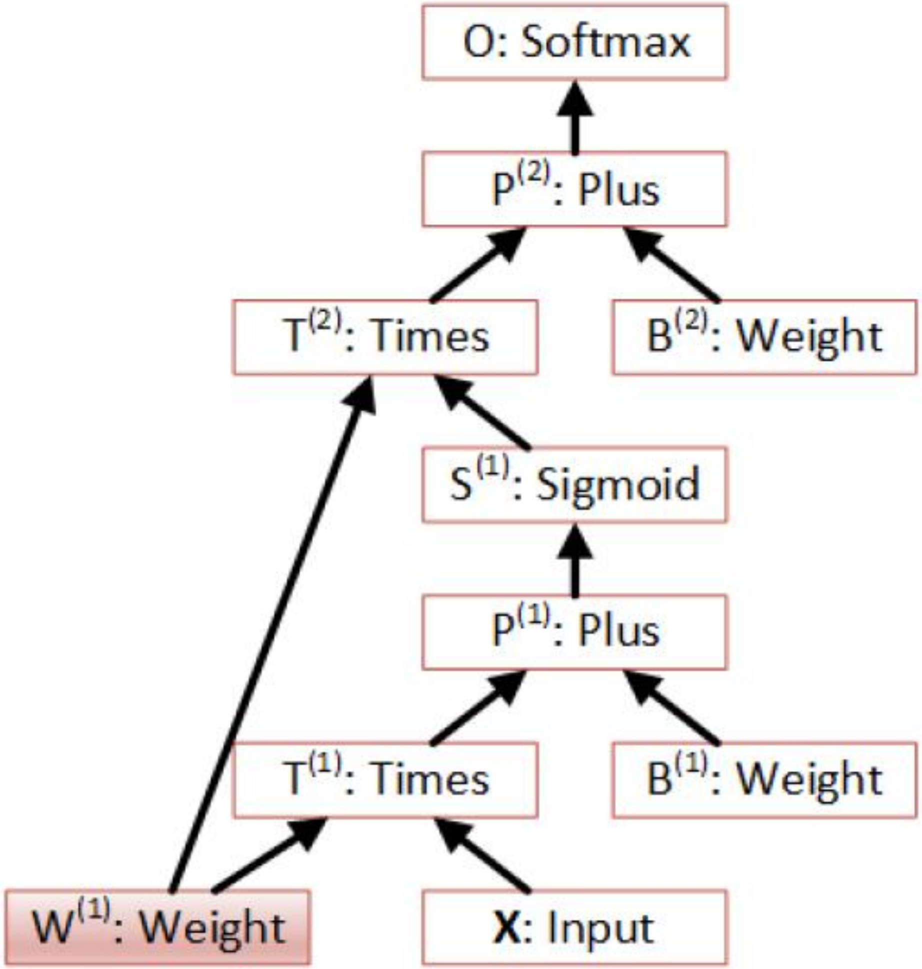




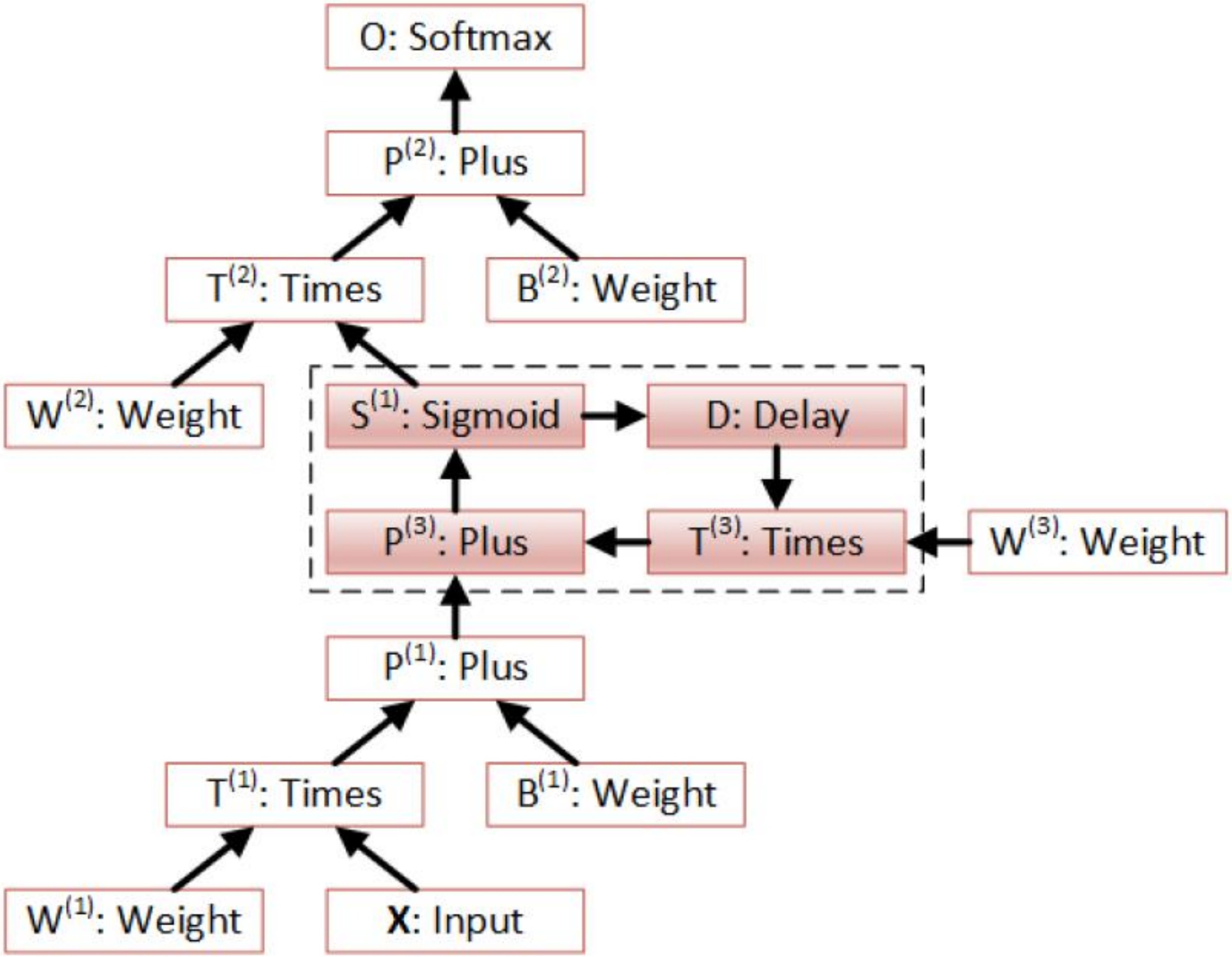
Example: CN with Multiple Inputs



Example: CN with Shared Parameters



Example: CN with Recurrence



Usage Example (with Config File)

- cntk configFile=yourConfigFile DeviceNumber=1

```
command=TIMIT_TrainNDL

TIMIT_TrainNDL=[
    action=train
    deviceId=$DeviceNumber$
    modelPath=“$your_model_path$

    NDLNetworkBuilder=[…]
    SGD=[…]
    reader=[…]
]
```

String Replacement
CPU: CPU
GPU: >=0 or auto

- You can also use Brain Script (beta) to specify the configuration or use Python and C# (in future releases) to directly instantiate related objects.

Usage Example (with Config File)

```
SGD=[
        minibatchSize=256:1024
        learningRatesPerMB=0.8:3.2*14:0.08
        momentumPerMB=0.9
]
```

First epoch

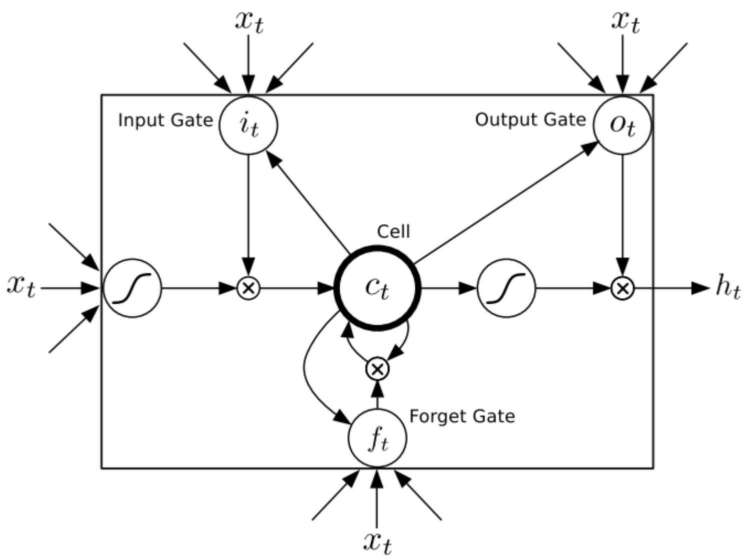
Rest epochs

Next 14 epochs

```
reader=[
   readerType="HTKMLFReader"
   randomize="auto"
   features=[
      dim=792
      scpFile="$FBankScpShort$"
   ]
   labels=[
      mlfFile="$MlfDir$\TIMIT.train.mlf"
      labelDim=183
      labelMappingFile="$MlfDir$\TIMIT.statelist"
   ]
]
```

Users can provide their own reader for their specific format

Network Definition with NDL (One Way)



$$
\begin{aligned}
\mathbf{i}_t &= \sigma\left(\mathbf{W}^{(xi)}\mathbf{x}_t + \mathbf{W}^{(hi)}\mathbf{h}_{t-1} + \mathbf{W}^{(ci)}\mathbf{c}_{t-1} + \mathbf{b}^{(i)}\right) \\
\mathbf{f}_t &= \sigma\left(\mathbf{W}^{(xf)}\mathbf{x}_t + \mathbf{W}^{(hf)}\mathbf{h}_{t-1} + \mathbf{W}^{(cf)}\mathbf{c}_{t-1} + \mathbf{b}^{(f)}\right) \\
\mathbf{c}_t &= \mathbf{f}_t \bullet \mathbf{c}_{t-1} + \mathbf{i}_t \bullet \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right) \\
\mathbf{o}_t &= \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right) \\
\mathbf{h}_t &= \mathbf{o}_t \bullet \tanh\left(\mathbf{c}_t\right),
\end{aligned}
$$

Network Definition with NDL (One Way)

```
LSTMComponent(inputDim, outputDim, inputVal) = [
   Wxo = Parameter(outputDim, inputDim)
   Wxi = Parameter(outputDim, inputDim)
   Wxf  = Parameter(outputDim, inputDim)
   Wxc = Parameter(outputDim, inputDim)

   bo = Parameter(outputDim, 1, init=fixedvalue, value=-1.0)
   bc = Parameter(outputDim, 1, init=fixedvalue, value=0.0)
   bi = Parameter(outputDim, 1, init=fixedvalue, value=-1.0)
   bf  = Parameter(outputDim, 1, init=fixedvalue, value=-1.0)

   Whi = Parameter(outputDim, outputDim)
   Wci = Parameter(outputDim , 1)
   Whf  = Parameter(outputDim, outputDim)
   Wcf  = Parameter(outputDim , 1)
   Who = Parameter(outputDim, outputDim)
   Wco = Parameter(outputDim , 1)
   Whc = Parameter(outputDim, outputDim)
```

Wrapped as a macro and can be reused

parameters

Network Definition with NDL (One Way)

```
delayH = PastValue(outputDim, output, timeStep=1)
delayC = PastValue(outputDim, ct, timeStep=1)
```

recurrent nodes

```
WxiInput = Times(Wxi, inputVal)
WhidelayHI = Times(Whi, delayH)
WcidelayCI = DiagTimes(Wci, delayC)
```

$$\mathbf{i}_t \quad = \quad \sigma\left(\mathbf{W}^{(xi)}\mathbf{x}_t + \mathbf{W}^{(hi)}\mathbf{h}_{t-1} + \mathbf{W}^{(ci)}\mathbf{c}_{t-1} + \mathbf{b}^{(i)}\right)$$

```
it = Sigmoid (Plus ( Plus (Plus (WxiInput, bi), WhidelayHI),
        WcidelayCI))
```

```
WhfdelayHF = Times(Whf, delayH)
WcfdelayCF = DiagTimes(Wcf, delayC)
Wxfinput = Times(Wxf, inputVal)
```

$$\mathbf{f}_t \quad = \quad \sigma\left(\mathbf{W}^{(xf)}\mathbf{x}_t + \mathbf{W}^{(hf)}\mathbf{h}_{t-1} + \mathbf{W}^{(cf)}\mathbf{c}_{t-1} + \mathbf{b}^{(f)}\right)$$

```
ft = Sigmoid( Plus (Plus (Plus(Wxfinput, bf), WhfdelayHF),
        WcfdelayCF))
```

Network Definition with NDL (One Way)

```
    WxcInput = Times(Wxc, inputVal)
    WhcdelayHC = Times(Whc, delayH)
    bit = ElementTimes(it, Tanh( Plus(WxcInput, Plus(WhcdelayHC, bc))))
    bft = ElementTimes(ft, delayC)

    ct = Plus(bft, bit)
```

$$\mathbf{c}_t = \mathbf{f}_t \bullet \mathbf{c}_{t-1} + \mathbf{i}_t \bullet \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

```
    Wxoinput = Times(Wxo, inputVal)
    WhodelayHO = Times(Who, delayH)
    Wcoct = DiagTimes(Wco, ct)

    ot = Sigmoid( Plus( Plus( Plus(Wxoinput, bo), WhodelayHO), Wcoct))
```

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$

```
    output = ElementTimes(ot, Tanh(ct))
]
```

$$\mathbf{h}_t = \mathbf{o}_t \bullet \tanh\left(\mathbf{c}_t\right)$$

Network Definition with NDL (Another Way)

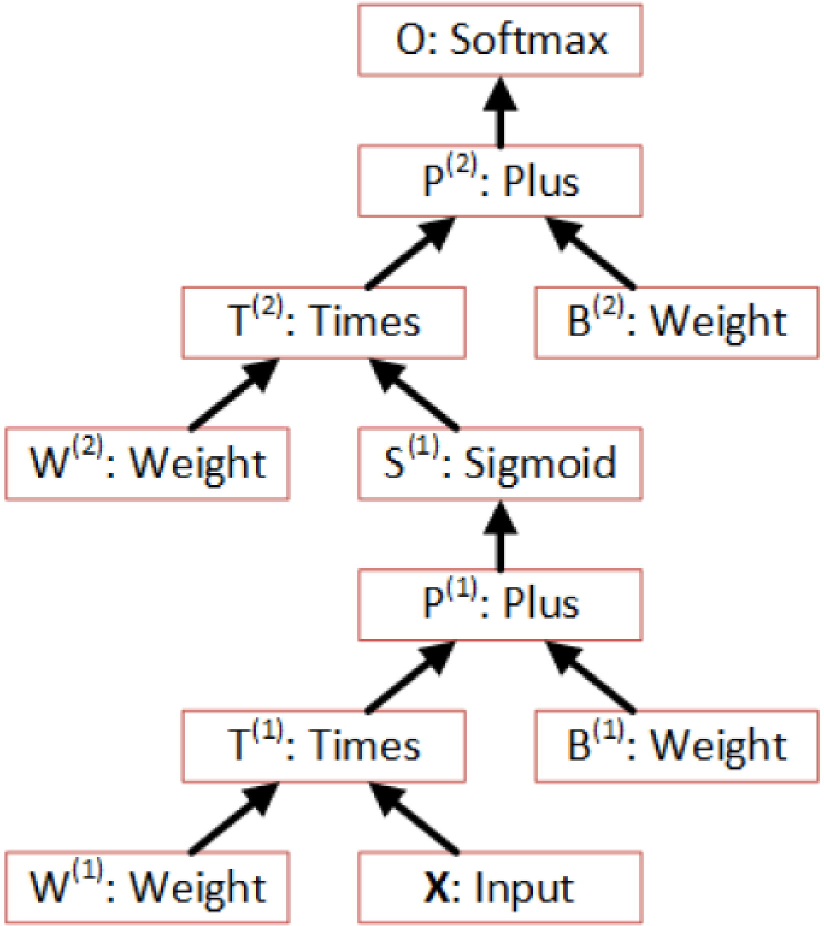
```
LSTMComponent(inputDim, outputDim, inputValue, outputDimX2, outputDimX3, outputDimX4, initScale, initBias) = [
    wx = Parameter(outputDimX4, inputDim, init="uniform", initValueScale=initScale);
    b = Parameter(outputDimX4, init="fixedValue", value=initBias);
    Wh = Parameter(outputDimX4, outputDim, init="uniform", initValueScale=initScale);
    Wci = Parameter(outputDim, init="uniform", initValueScale=initScale);
    Wcf = Parameter(outputDim, init="uniform", initValueScale=initScale);
    Wco = Parameter(outputDim, init="uniform", initValueScale=initScale);

    dh = PastValue(outputDim, output, timeStep=1);
    dc = PastValue(outputDim, ct, timeStep=1);

    wxx = Times(wx, inputValue);
    wxxpb = Plus(wxx, b);
    whh = Times(wh, dh);
    wxxpbpwhh = Plus(wxxpb,whh)

    G1 = RowSlice(0, outputDim, wxxpbpwhh)
    G2 = RowSlice(outputDim, outputDim, wxxpbpwhh)
    G3 = RowSlice(outputDimX2, outputDim, wxxpbpwhh);
    G4 = RowSlice(outputDimX3, outputDim, wxxpbpwhh);

    Wcidc = DiagTimes(Wci, dc);
    it = Sigmoid (Plus ( G1, Wcidc));
    bit = ElementTimes(it, Tanh( G2 ));
    Wcfdc = DiagTimes(Wcf, dc);
    ft = Sigmoid( Plus (G3, Wcfdc));
    bft = ElementTimes(ft, dc);
    ct = Plus(bft, bit);
    Wcoct = DiagTimes(Wco, ct);
    ot = Sigmoid( Plus( G4, Wcoct));
    output = ElementTimes(ot, Tanh(ct));
]
```

The whole LSTM with combined operations: Simpler and Faster

All the operations shown here are also available to other script languages.

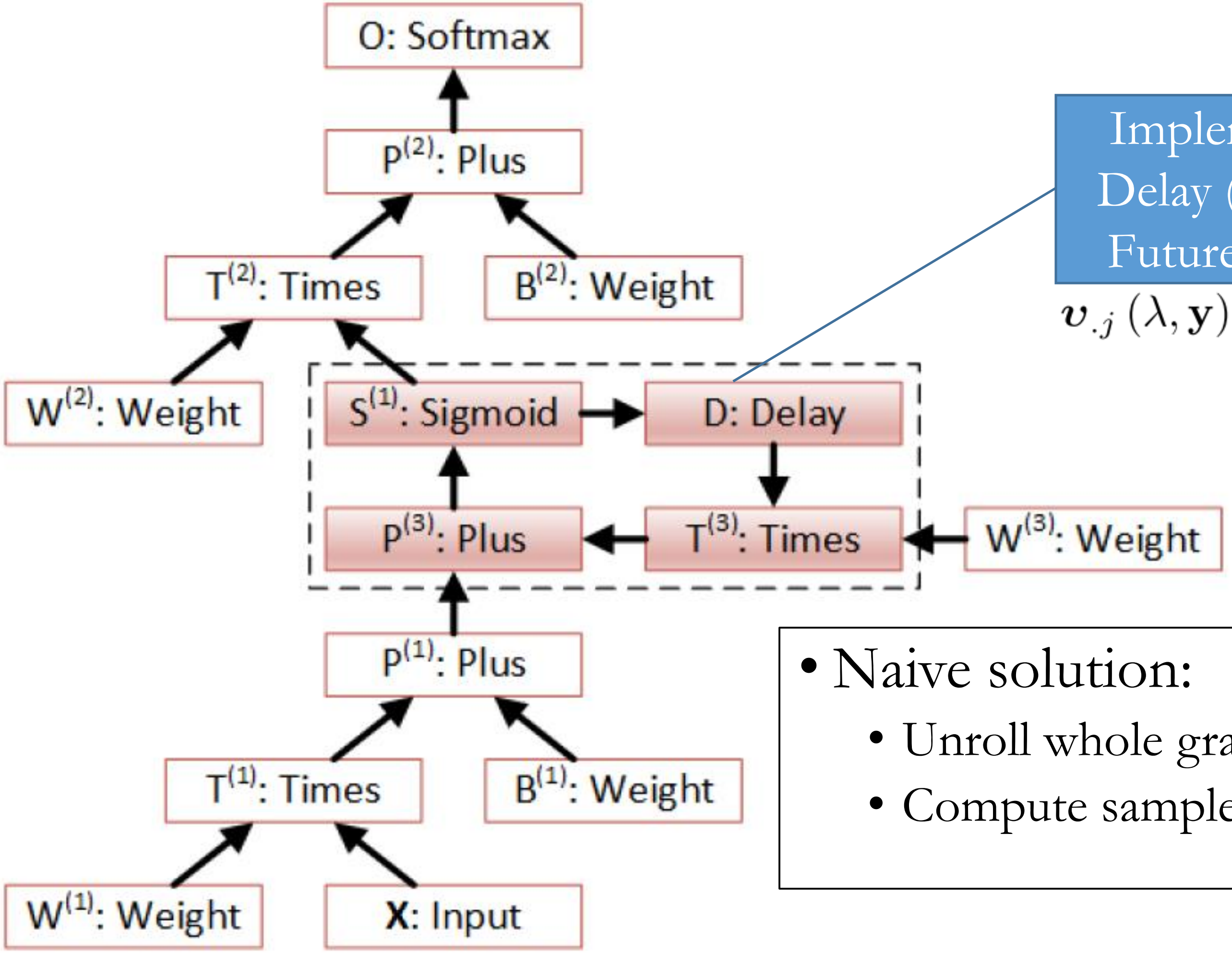
Without Loops

- Given the root node, the computation order can be determined by a depth-first traverse of the directed acyclic graph (DAG).
- Only need to run it once and cache the order
- Can **easily parallelize on the whole minibatch** to speed up computation



With Loops (Recurrent Connections)

- Very important in many interesting models



Implemented with Delay (PastValue or FutureValue) node

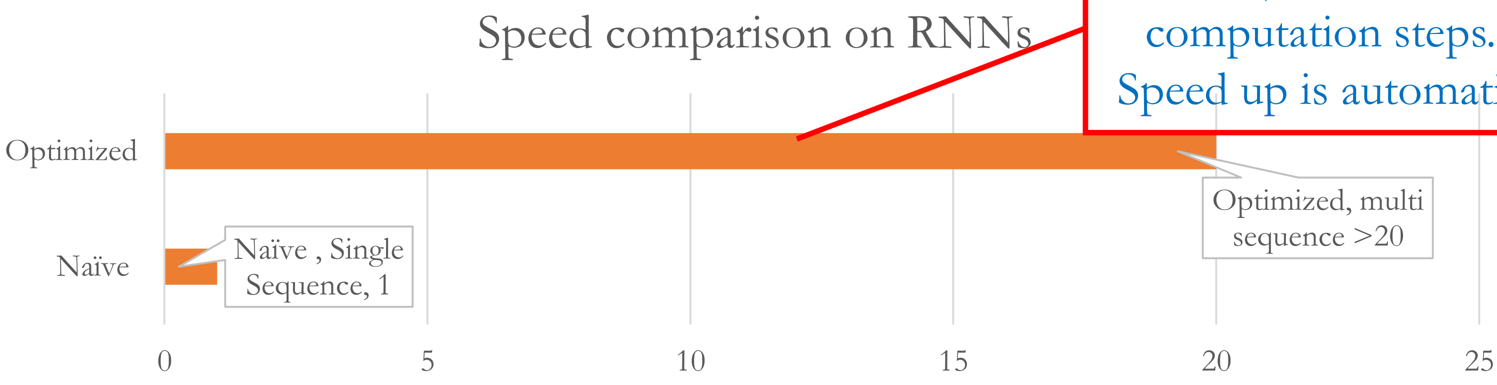
$$\boldsymbol{v}_{.j}(\lambda, \mathbf{y}) = \mathbf{y}_{.(j-\lambda)}$$

- Naive solution:
  - Unroll whole graph over time
  - Compute sample by sample

With Loops (Recurrent Connections)

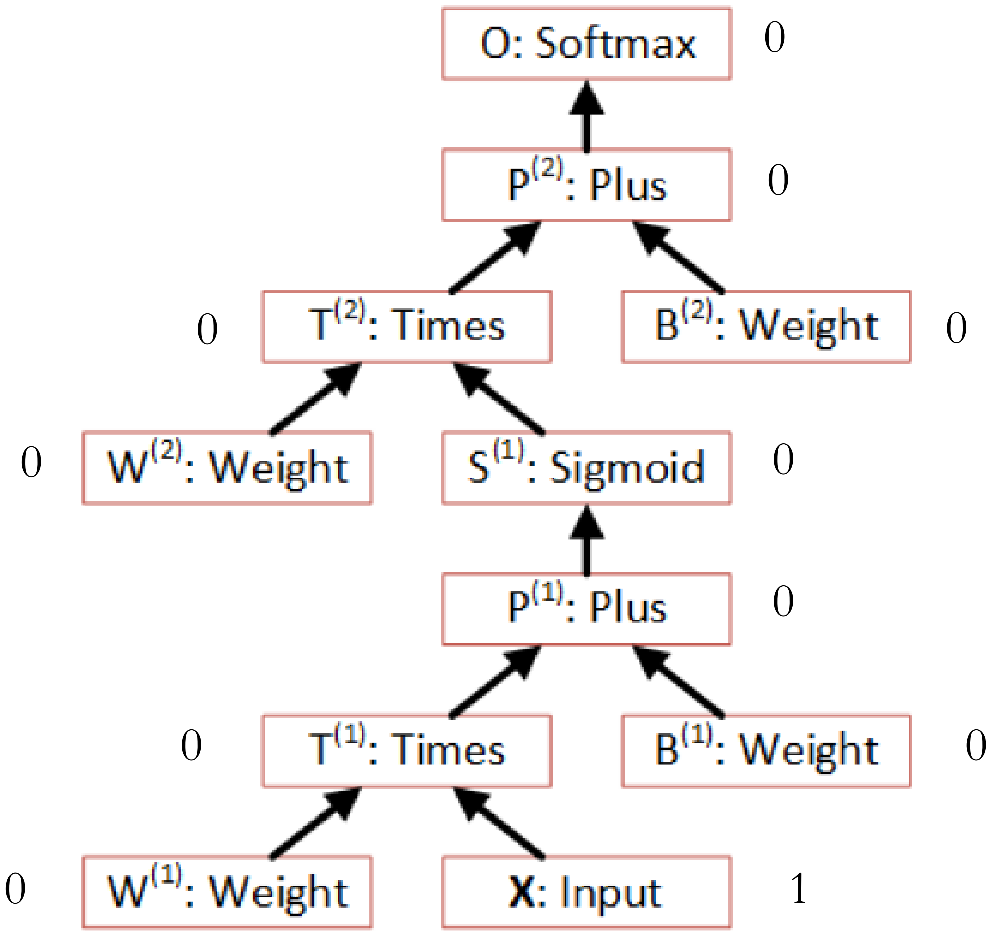
- We developed a smart algorithm to analyze the computational network so that we can
  - Find loops in arbitrary computational networks
  - Do whole minibatch computation on everything except nodes inside loops
  - Group multiple sequences with variable lengths (better convergence property than tools that only support batching of same length sequences)

Users just describe computation steps.
Speed up is automatic



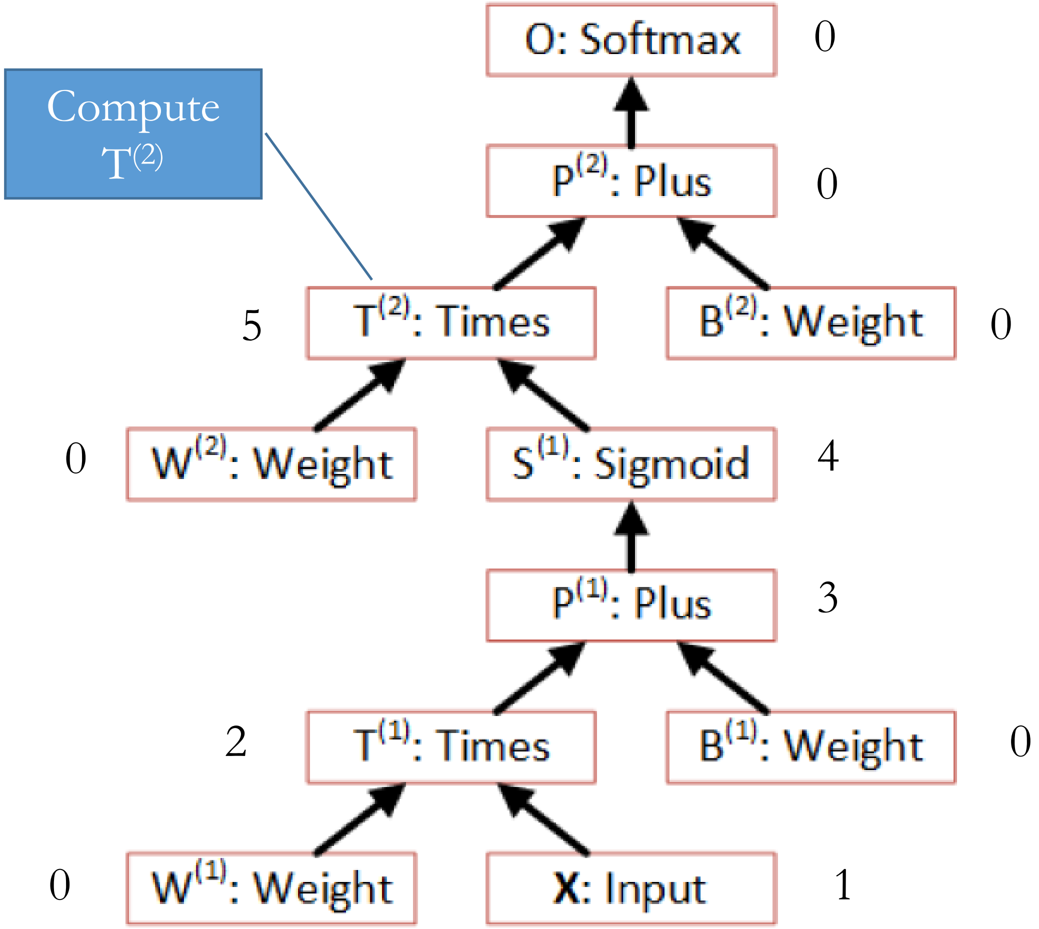
Forward Computation Efficiency

- Add time stamps to reduce duplicate computation when you need to evaluate several nodes



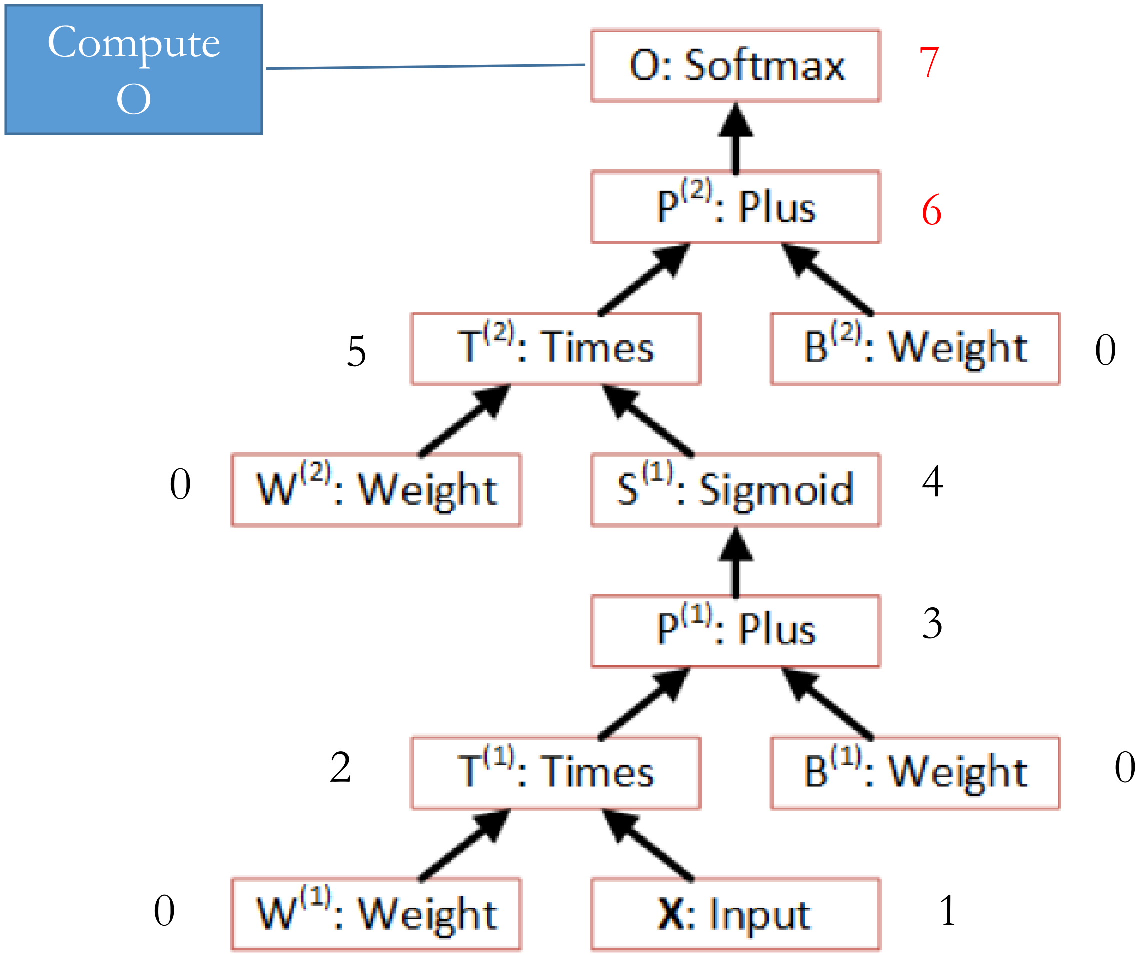
Forward Computation Efficiency

- Add time stamps to reduce duplicate computation when you need to evaluate several nodes



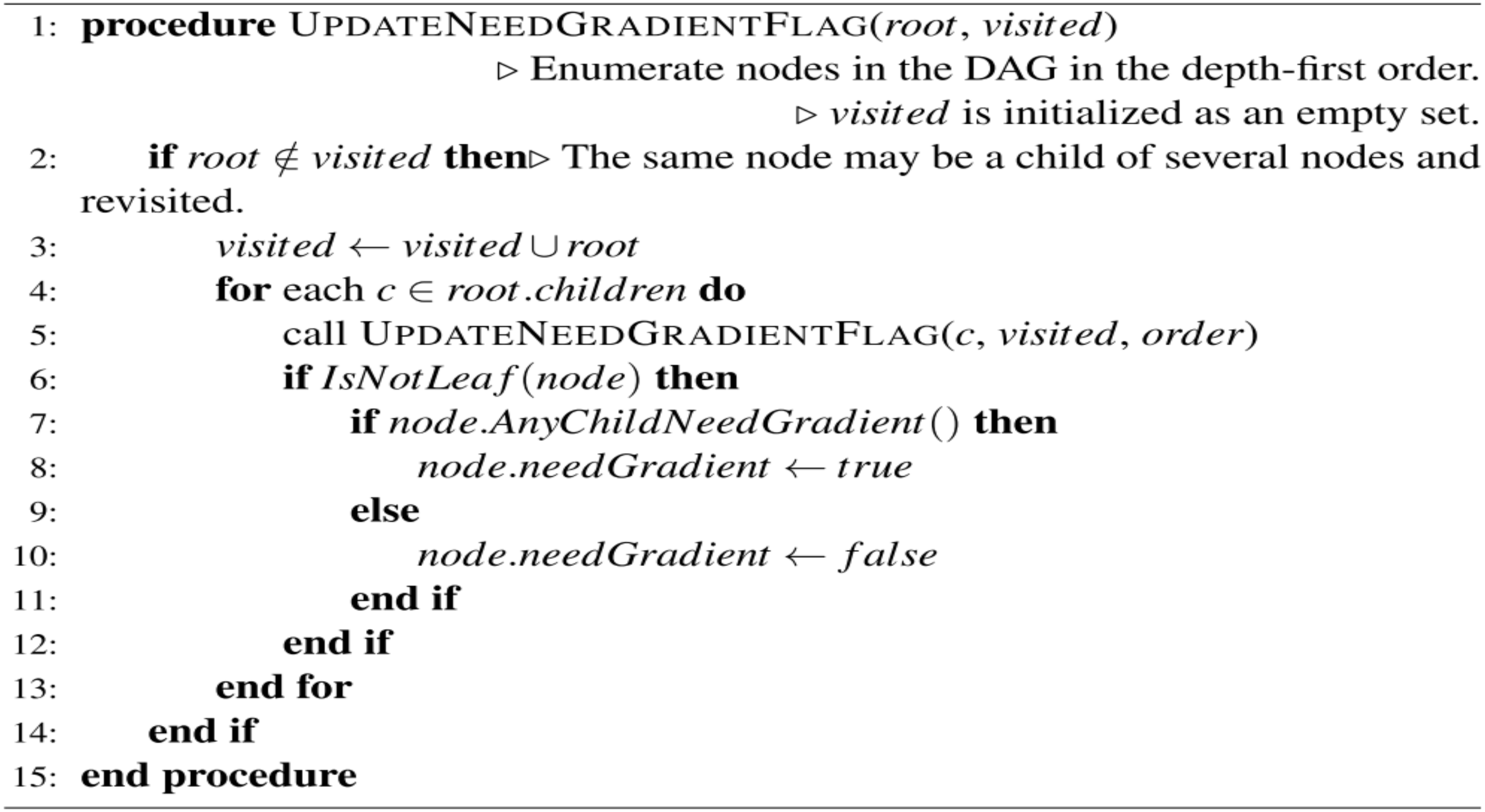
Forward Computation Efficiency

- Add time stamps to reduce duplicate computation when you need to evaluate several nodes



Gradient Computation Efficiency

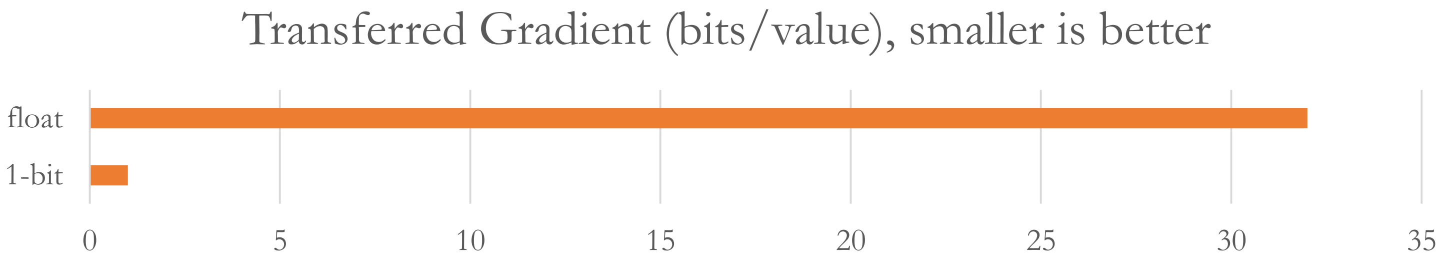
- **Gradient computation is not needed for some nodes.**
  - Set parameterUpdateRequired=false for constant leaf nodes.
    - Indicates needGradient=false for those nodes
  - Propagate the flag up the graph using the depth-first traversal.
  - Only compute the gradient when needGradient=true.



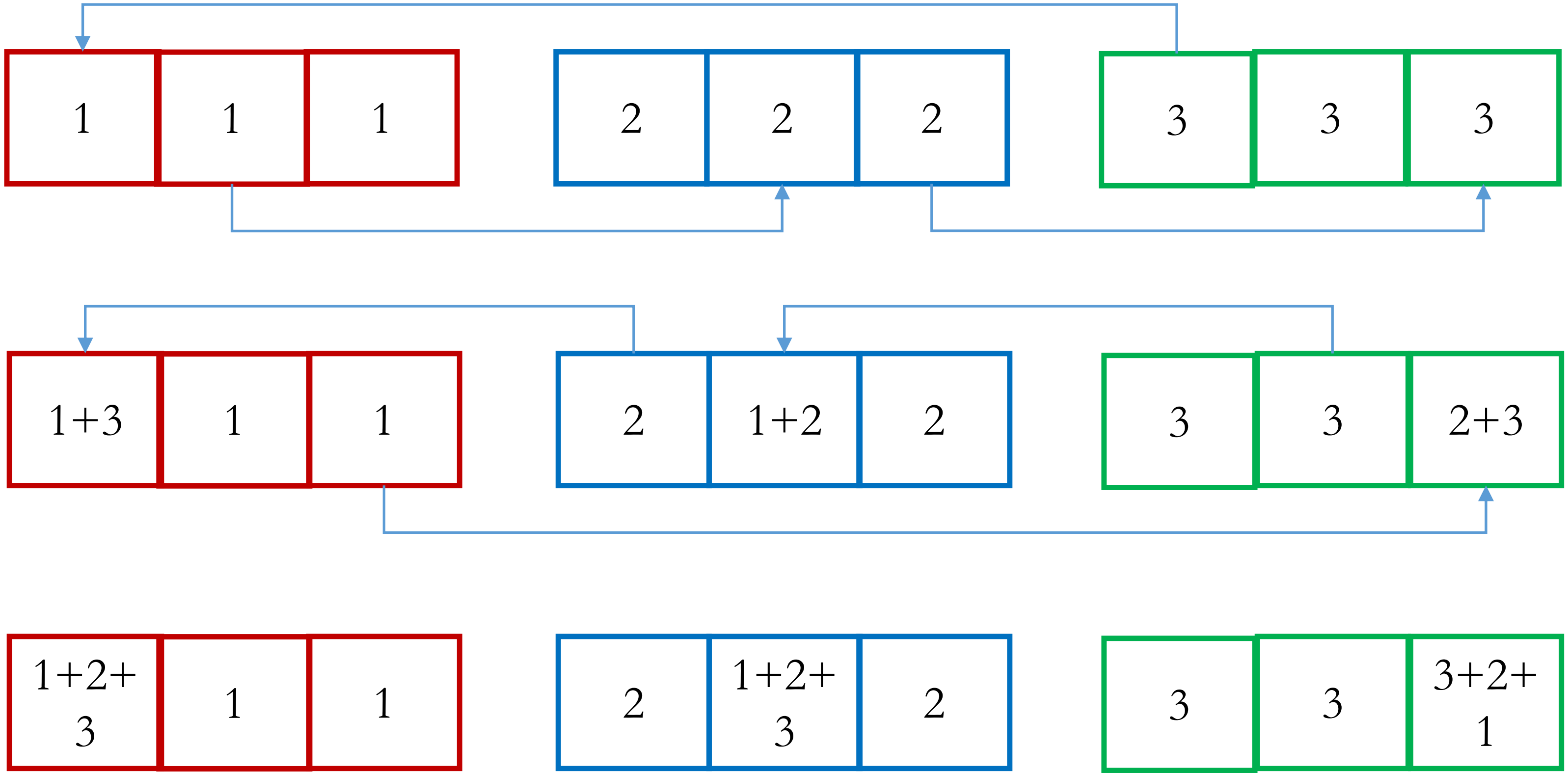
```
procedure UPDATENEEDGRADIENTFLAG($root$, $visited$)
                        ▷ Enumerate nodes in the DAG in the depth-first order.
                                        ▷ $visited$ is initialized as an empty set.
    if $root \notin visited$ then▷ The same node may be a child of several nodes and revisited.
        $visited \leftarrow visited \cup root$
        for each $c \in root.children$ do
            call UPDATENEEDGRADIENTFLAG($c$, $visited$, $order$)
            if $IsNotLeaf(node)$ then
                if $node.AnyChildNeedGradient()$ then
                    $node.needGradient \leftarrow true$
                else
                    $node.needGradient \leftarrow false$
                end if
            end if
        end for
    end if
end procedure
```

Data Parallelization: 1-Bit Quantized SGD

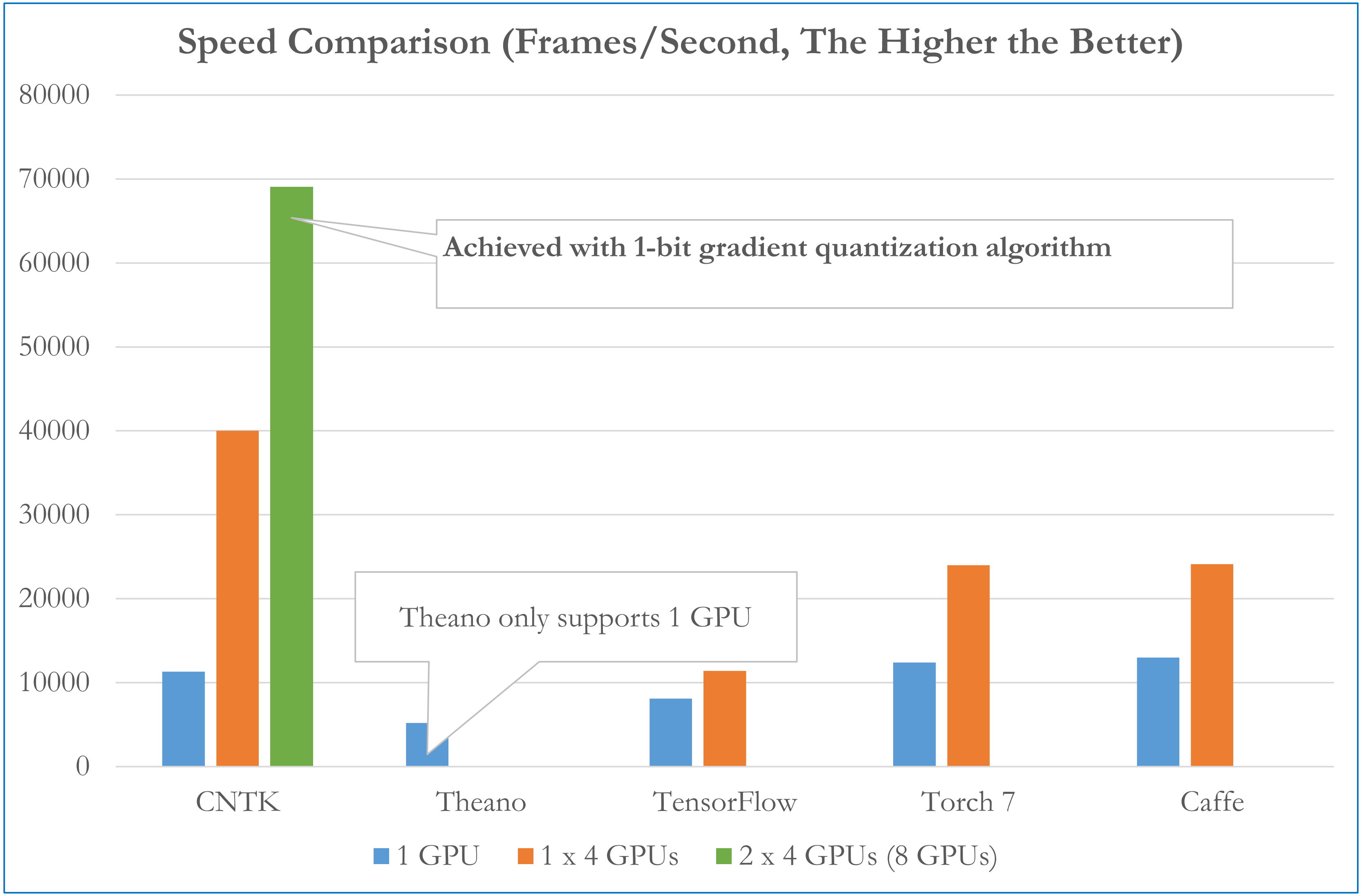
- **Bottleneck** for distributed learning: **Communication** cost
- Solution: reduce the amount of data need to be communicated by **quantizing gradients to just 1 bit**
  - It's a lot safer to quantize gradients than model parameters and outputs (gradients are small and noisy anyway)
  - Carry quantization residue to next minibatch (important)
  - Further hide communication with double-buffering: send one while processing the other
  - Use an O(1) communication scheduler to sync gradients
  - increase minibatch size to fully utilize each GPU as early as possible



O(1) Aggregation

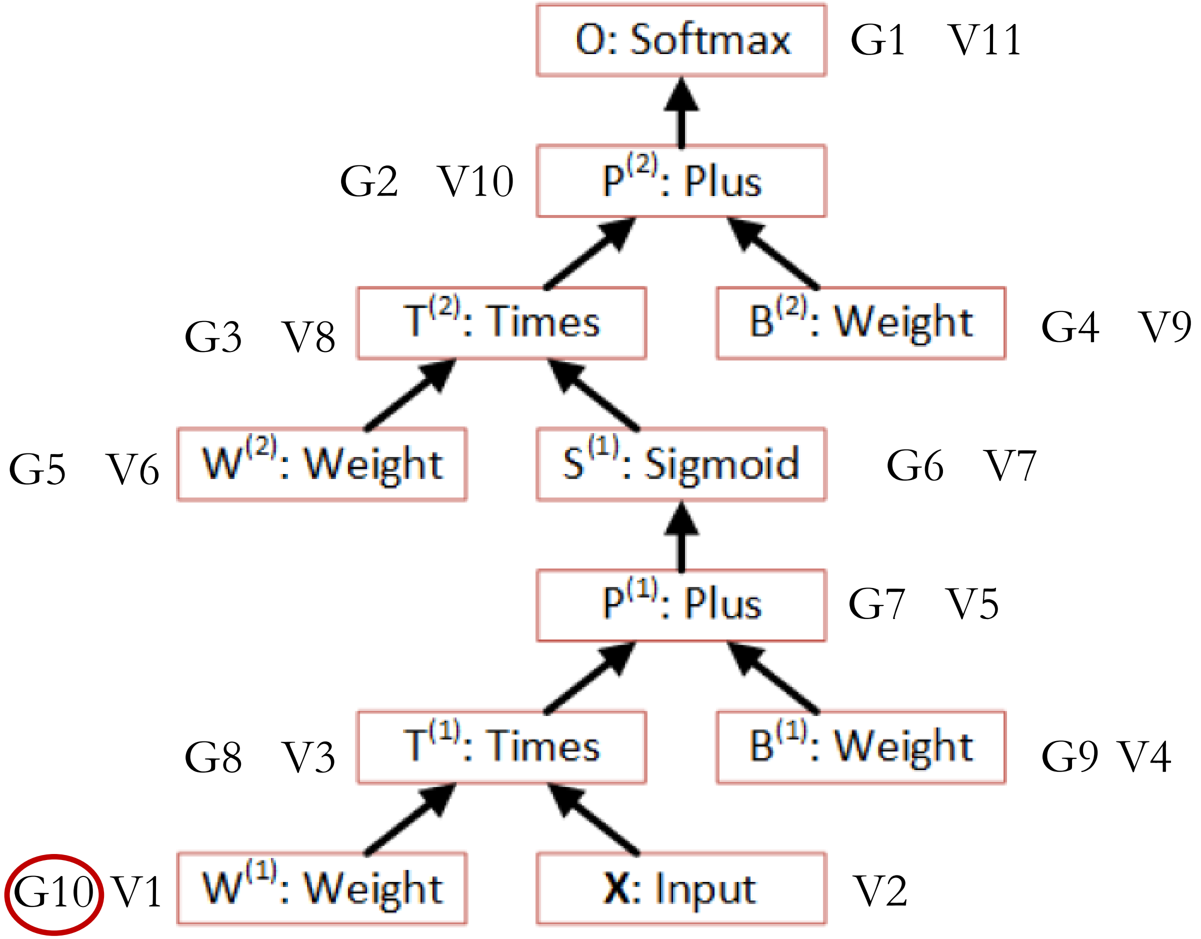


CNTK Computational Performance



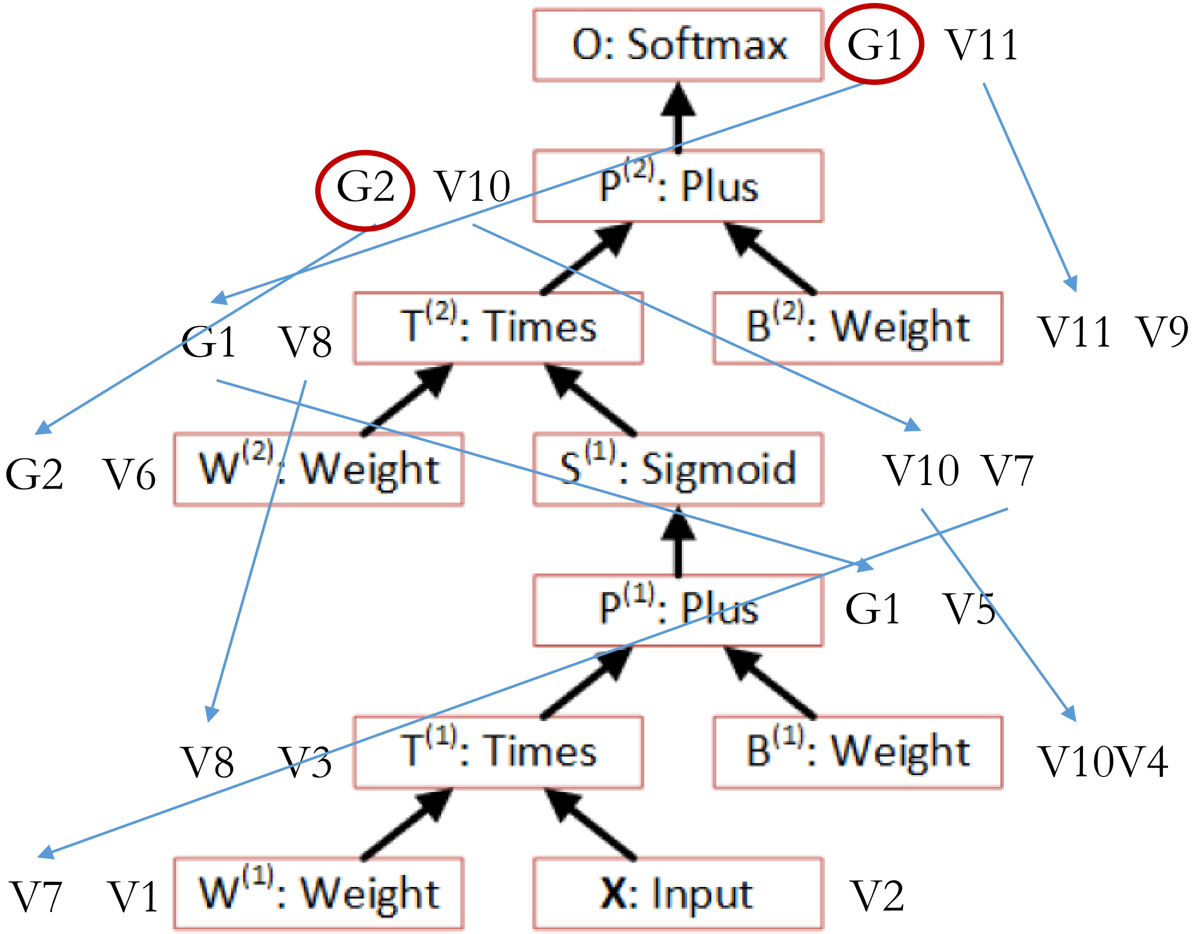
Memory Sharing (Training)

- Without memory sharing



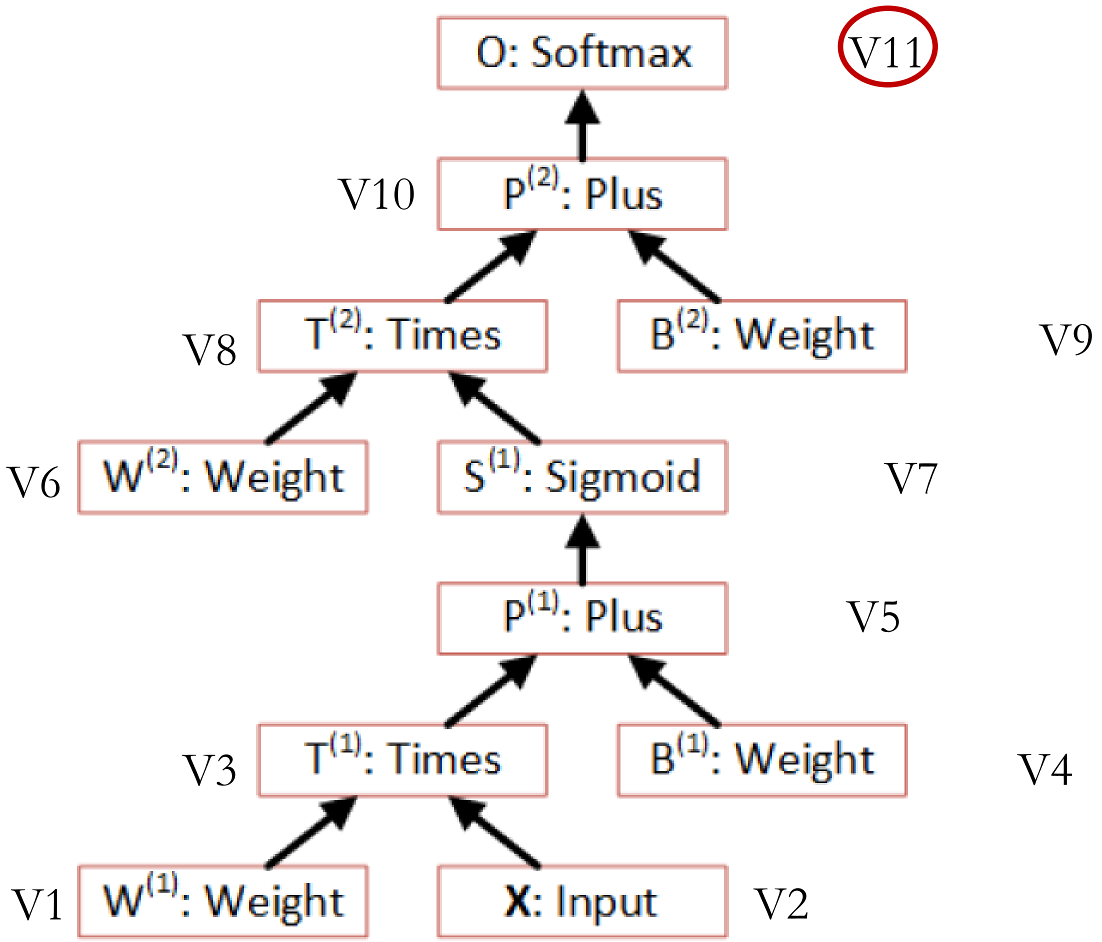
Memory Sharing (Training)

- With memory sharing



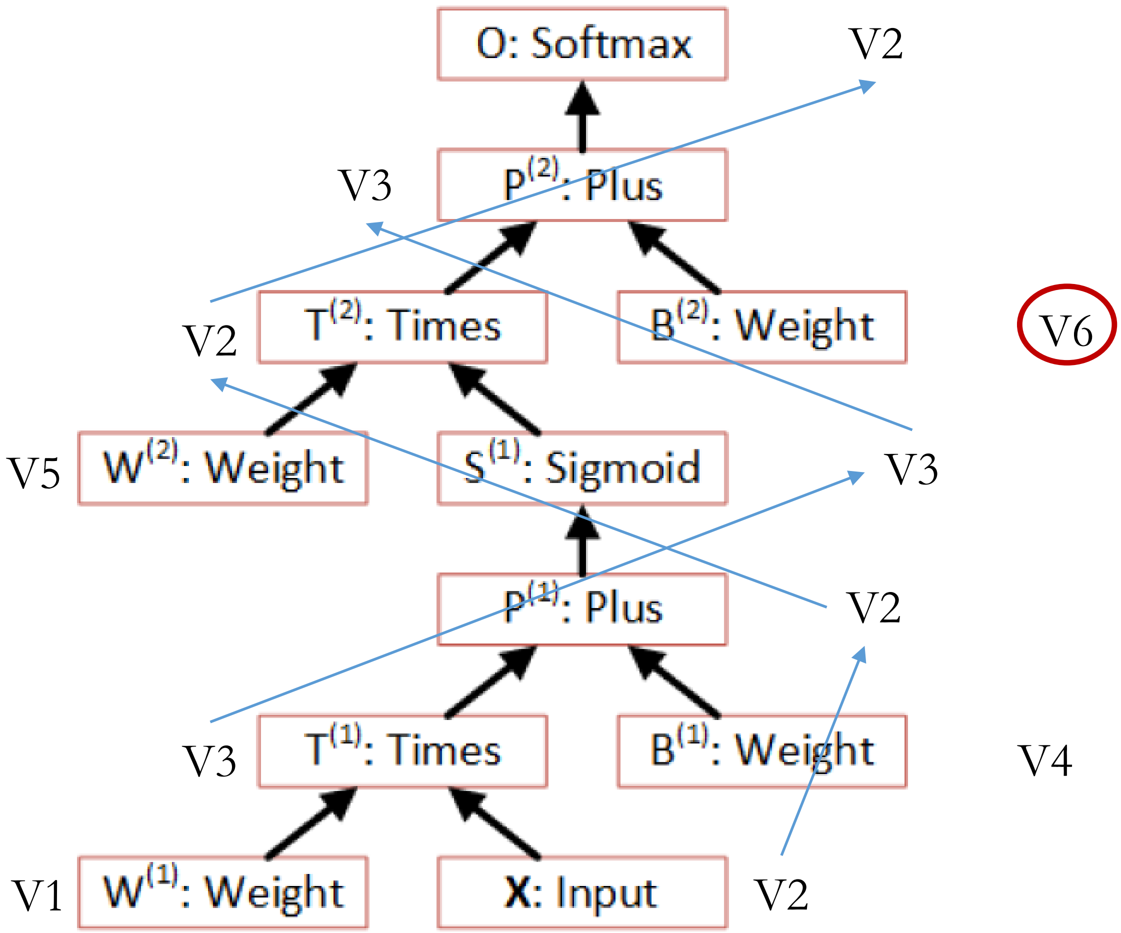
Memory Sharing (Evaluation)

- Without memory sharing



Memory Sharing (Evaluation)

- With memory sharing; weights have their own memory



Memory Sharing

- Use same memory across minibatches: don't destroy and reallocate memory at each minibatch.
- Share memory across computation nodes when possible
    - Analyze the execution plan and release the memory to a pool to be reused by other nodes or computation if possible
    - E.g., when a node finished computing all its children's gradients, the matrices owned by that node can all be released.
    - Can **reduce memory by 1/3 to 1/2 for training** in most cases
    - Can reduce memory even more if gradients are not needed.

Outline

- Introduction (Xuedong Huang)
- CNTK Deep Dive (Dong Yu)
- **Summary/Q&A (Dong & Xuedong)**

Summary

- CNTK is a powerful tool that supports CPU/GPU and runs under Windows/Linux
- CNTK is extensible with the low-coupling modular design: adding new readers and new computation nodes is easy with a new reader design
- Network definition language, macros, and model editing language (as well as Brain Script and Python binding in the future) makes network design and modification easy
- Compared to other tools CNTK has a great balance between efficiency, performance, and flexibility

Additional Resources

- CNTK Reference Book
    - A. Agarwal, E. Akchurin, C. Basoglu, G. Chen, S. Cyphers, J. Droppo, A. Eversole, B. Guenter, M. Hillebrand, R. Hoens, X. Huang, Z. Huang, V. Ivanov, A. Kamenev, P. Kranen, O. Kuchaiev, W. Manousek, A. May, B. Mitra, O. Nano, G. Navarro, A. Orlov, M. Padmilac, H. Parthasarathi, B. Peng, A. Reznichenko, F. Seide, M. L. Seltzer, M. Slaney, A. Stolcke, H. Wang, Y. Wang, K. Yao, D. Yu, Y. Zhang, G. Zweig (in alphabetical order),, "An Introduction to Computational Networks and the Computational Network Toolkit", Microsoft Technical Report MSR-TR-2014-112, 2014.
    - Contains all the information you need to understand and use CNTK
- Current source code site
    - https://cntk.codeplex.com/
    - Contains all the source code and example setups
    - You may understand better how CNTK works by reading the source code
    - New functionalities are added constantly
- Note: CNTK will be moved to GitHub early next year.