Computational Networks

A Generalization of Deep Learning Models

A tutorial at ICASSP 2015

Dong Yu, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Jasha Droppo
Outline

• Motivation
• Introduction to Deep Learning and Prevailing Deep Learning Models
• Computational Network: A Unified Framework for Models Expressible as Functions
• Computational Network Toolkit: A Generic Toolkit for Building Computational Networks
• Examples: Acoustic Model, Language Model, and Image Classification
• Summary
Outline

• Motivation

• Introduction to Deep Learning and Prevailing Deep Learning Models

• Computational Network: A Unified Framework for Models Expressible as Functions

• Computational Network Toolkit: A Generic Toolkit for Building Computational Networks

• Examples: Acoustic Model, Language Model, and Image Classification

• Summary
Why Are We All Here Today?

• Deep learning and deep neural networks are hot stuff!
  • Big impact in academic & industrial research.

• Why the widespread adoption?
  • Implementing a (deep) neural network is not difficult
    • Many groups were able to quickly adopt this new approach
    • and it works!

• This led to the “era of low hanging fruit”
  • Apply DNN to new tasks, e.g. ASR, TTS, NLP
  • Invent simple extensions, e.g. NAT, SAT
  • Create deep version of other known networks, e.g. RNN

• But, what next?
The Implementation Bottleneck

• It is easy (and fun!) to dream up new architecture variations, topologies, training strategies
  • Recurrence across arbitrary layers
  • RNN across multiple time delays
  • Complicated weight tying strategies
  • Gating

• It is time-consuming to implement these
  • Requires new coding for forward and back propagation
  • CPU + GPU means twice as much coding & debugging!

• Also true if you want to reproduce published research
• This is a big bottleneck to progress
Motivation: Break the Bottleneck

• Our goal: create a tool to try out new ideas quickly.
  • High risk, high reward, fail fast
  • Achieve flexibility without sacrificing efficiency

• 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.

• For example, without writing any code you can
  • Create a DNN, RNN, or LSTM
  • Rearrange LSTM’s gating structure
  • Add novel recurrence
Outline

• Motivation

• Introduction to Deep Learning and Prevailing Deep Learning Models

• Computational Network: A Unified Framework for Models Expressible as Functions

• Computational Network Toolkit: A Generic Toolkit for Building Computational Networks

• Examples: Acoustic Model, Language Model, and Image Classification

• Summary
Deep Neural Networks

• Catchy name for multi-layer perceptron (MLP) with “many” hidden layers
  • In: observations (features)
  • Out: prediction (classes or features)

• Training with back propagation to minimize the cross-entropy at the frame or sequence level

• Optimization important & difficult

• Outputs used as is, or in downstream classifier, e.g. hidden Markov model (HMM), support vector machine (SVM)
Success Across Many Fields

- Including automatic speech recognition (ASR), image classification, and natural language processing.
- Example: Success in ASR across many tasks
Deep Neural Networks Raise All Boats

• Improve across all phonemes [Huang 2014]
Deep Neural Networks Raise All Boats

- Improve across all signal-to-noise ratios [Huang 2014]
The Power of Depth

• Error rates decrease with depth

<table>
<thead>
<tr>
<th># of Layers X # of Neurons</th>
<th>SWBD WER (%) [300hrs]</th>
<th>Aurora 4 WER (%) [10hrs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 2k</td>
<td>24.2</td>
<td>---</td>
</tr>
<tr>
<td>3 x 2k</td>
<td>18.4</td>
<td>14.2</td>
</tr>
<tr>
<td>5 x 2k</td>
<td>17.2</td>
<td>13.8</td>
</tr>
<tr>
<td>7 x 2k</td>
<td>17.1</td>
<td>13.7</td>
</tr>
<tr>
<td>9 x 2k</td>
<td>17.0</td>
<td>13.9</td>
</tr>
</tbody>
</table>
The Power of Depth

- Error rates decrease with depth
- Depth is not just a way to add parameters

<table>
<thead>
<tr>
<th># of Layers X # of Neurons</th>
<th>SWBD WER (%) [300hrs]</th>
<th>Aurora 4 WER (%) [10hrs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 2k</td>
<td>24.2</td>
<td>---</td>
</tr>
<tr>
<td>3 x 2k</td>
<td>18.4</td>
<td>14.2</td>
</tr>
<tr>
<td>5 x 2k</td>
<td>17.2</td>
<td>13.8</td>
</tr>
<tr>
<td>7 x 2k</td>
<td>17.1</td>
<td>13.7</td>
</tr>
<tr>
<td>9 x 2k</td>
<td>17.0</td>
<td>13.9</td>
</tr>
<tr>
<td>1 x 16k</td>
<td>22.1</td>
<td>--</td>
</tr>
</tbody>
</table>
Why DNNs Perform So Well

- It’s a combination of nonlinear feature extraction and log-linear classifier
- Many simple nonlinearities combine to form arbitrarily complex nonlinearities for better feature transformation
- Joint feature learning & classifier design
- Lower-layer feature representations are exploited by the higher layer feature detectors
- Features at higher layers more invariant and discriminative than at lower layers
Limitations of DNNs

• We want features that are discriminative and invariant
  • **Discriminative**: transfer the raw feature non-linearly into a higher dimensional space in which things that were non-separable become separable
  • **Invariant**: pool or aggregate features in the new space to introduce invariance

• DNNs achieve this through many layers of non-linear transformations with supervision.

• However,
  • DNNs do not explicitly exploit known structures (e.g., translational variability) in the input data
  • DNNs do not explicitly apply operations that reduces variability (e.g., pooling and aggregation)

• Can we build these properties directly in the neural networks?
  • Yes, e.g., convolutional neural networks (CNNs)
Convolutional Neural Networks

- Explicitly models translational variability and enables shift invariance
  - Shared local filters (weights) tiled across image to detect the same pattern at different locations
  - Sub-sampling through pooling (max, average, or other) to reduce variability

One kernel for each input-output channel pair
Convolutional Neural Networks

• Key to improve image classification accuracy
• Deep CNNs now state of the art for image classification

[Zeiler and Fergus, 2013]
Limitations of CNNs

• CNNs mainly deal with translational variability
• There are more types of variability in image classification
  • horizontal reflections
  • color intensity differences
  • scaling
• Techniques such as data synthesis and augmentation, and local response normalization are needed to deal with these additional variability
• CNNs cannot take advantage dependencies and correlations between samples (and labels) in a sequence
• Recurrent neural networks (RNNs) are designed for this
Recurrent Neural Networks

- Models dependencies and correlations between samples (and labels) in a sequence
- Information may be fed back from hidden and output layers in the previous time steps
Deep Recurrent Neural Networks

• Combine deep neural networks with recurrent neural networks

• Trained with backpropagation through time (BPTT) and truncated BPTT
Limitations of Simple RNNs

• Simple RNNs are difficult to train due to diminishing and explosion of gradients over time
  • Can be partially alleviated with gradient thresholding

• Simple RNNs have difficulty modeling long-range dependencies
  • The effect of information from past samples decreases exponentially

• Is it possible to solve the gradient diminishing problem so that we can model long-range dependencies

• Yes, with carefully designed recurrent structures such as long short-term memory (LSTM) RNNs.
Long Short-Term Memory RNNs

• An extension of RNN that addresses vanishing gradient problem
  • Memory cell is linearly time-recurrent
  • Use gates to control and keep long-range information

[Image credit: Graves 2013]
Long Short-Term Memory RNNs

• State-of-the-art performance for many sequential recognition problems:
  • ASR
  • hand written character recognition

• Is a generic model

• May not be optimal for the specific problems at hand which requires designing customized models
Outline

• Motivation

• Introduction to Deep Learning and Prevailing Deep Learning Models

• Computational Network: A Unified Framework for Models Expressible as Functions

• Computational Network Toolkit: A Generic Toolkit for Building Computational Networks

• Examples: Acoustic Model, Language Model, and Image Classification

• Summary
Generalization of Deep Learning Models

• Consider the models we just described…
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Recurrent Neural Networks (RNNs)
  • Long Short-Term Memory (LSTM) RNNs

• …and some other common machine learning models
  • Gaussian Mixture Models (GMMs)
  • Logistic Regression Models (LRMs)
  • Log-Linear Models (LLMs)

• Common property:
  • Can be described as a series of computational steps
Example: One Hidden Layer NN

Output Layer

\[ O \]

\[ \text{Softmax} \]

\[ P^{(2)} \]

\[ W^{(2)}, b^{(2)} \]

Hidden Layer

\[ S^{(1)} \]

\[ \text{Sigmoid} \]

\[ P^{(1)} \]

\[ W^{(1)}, b^{(1)} \]

X

\[ O: \text{Softmax} \]

\[ P^{(2)}: \text{Plus} \]

\[ T^{(2)}: \text{Times} \]

\[ B^{(2)}: \text{Weight} \]

\[ W^{(2)}: \text{Weight} \]

\[ S^{(1)}: \text{Sigmoid} \]

\[ P^{(1)}: \text{Plus} \]

\[ T^{(1)}: \text{Times} \]

\[ B^{(1)}: \text{Weight} \]

\[ W^{(1)}: \text{Weight} \]

\[ X: \text{Input} \]
Computational Networks

• A generalization of machine learning models that can be described as a series of computational steps.

• Representation:
  • A list of computational nodes denoted as
    \[ n = \{ \text{node name} : \text{operation name} \} \]
  • The parent-children relationship describing the operands
    \[ \{ n : c_1, \cdots, c_{K_n} \} \]
    • \( K_n \) is the number of children of node \( n \). For leaf nodes \( K_n = 0 \).
    • 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 describe models that are far more complicated than simple conventional neural networks.
Example: CN with Multiple Inputs
Example: CN with Shared Parameters

```
O: Softmax
   /   \
 p^{(2)}: Plus  T^{(2)}: Times  B^{(2)}: Weight
  |     /
 p^{(1)}: Plus  S^{(1)}: Sigmoid
    |     /
 T^{(1)}: Times  B^{(1)}: Weight
    |     /
 W^{(1)}: Weight  X: Input
```
Example: CN with Recurrence
Forward Computation – No Loop

• 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
Forward Computation – With Loop

- Very important in many interesting models

\[ v_{j}(\lambda, y) = y_{(j-\lambda)} \]

- Naive solution:
  - Unroll whole graph over time
  - Compute sample by sample
Forward Computation – With Loop

- Very important in many interesting models

Composite Node with loops (strongly connected components) inside

Implemented with delay node

\[ v_{ij}(\lambda, y) = y_{(j-\lambda)} \]

Better solution:
- Reduce each loop (strongly connected component) into a single node
- Only unroll over time inside loops
Forward Computation – With Loop

• Nodes inside the loops need to be computed sample by sample unrolled over time
Forward Computation – With Loop

• Nodes inside the loops need to be computed sample by sample unrolled over time

Key observation: Delay node can be treated as leaf since it’s already computed at t-1
Forward Computation – With Loop

• Nodes inside the loops need to be computed sample by sample unrolled over time

Remove the arrows to the delay node and convert it to a DAG
Forward Computation – With Loop

• May still be slow inside the loops, esp. if loop is long
• Solution: process multiple sequences in a batch
Forward Computation – With Loop

• May still be slow inside the loops, esp. if loop is long
• Solution: process multiple sequences in a batch
Forward Computation – With Loop

• May still be slow inside the loops, esp. if loop is long
• Solution: process multiple sequences in a batch
Forward Computation – With Loop

- What if sequences have different lengths
- Randomly select a new sequence and fill it in

You can signal the start of a new sequence for one or many sequences if sequences have different lengths

New sequence
Forward Computation Efficiency

• Add time stamps to reduce duplicate computation
Forward Computation Efficiency

• Add time stamps to reduce duplicate computation
Forward Computation Efficiency

• Add time stamp to reduce duplicate computation
Training

• Decide training criterion and add corresponding computation nodes

Model update with gradient descent

\[
W_{t+1} \leftarrow W_t - \varepsilon \nabla W_t,
\]

\[
\nabla W_t = \frac{1}{M_b} \sum_{m=1}^{M_b} \nabla W_t J(W; x^m, y^m)
\]
Automatic Gradient Computation

- Naive solution: compute and keep gradients at edges

\[
\begin{align*}
\nabla_J W^{(1)} &= \frac{\partial J}{\partial V^{(1)}} \frac{\partial V^{(1)}}{\partial W^{(1)}} \frac{\partial V^{(2)}}{\partial W^{(1)}} + \frac{\partial J}{\partial V^{(3)}} \frac{\partial V^{(3)}}{\partial W^{(1)}} \frac{\partial V^{(4)}}{\partial W^{(1)}} \\
\nabla_J W^{(2)} &= \frac{\partial J}{\partial V^{(1)}} \frac{\partial V^{(1)}}{\partial W^{(2)}} \frac{\partial V^{(2)}}{\partial W^{(2)}} .
\end{align*}
\]
Automatic Gradient Computation

• Better solution: compute and keep gradients at nodes

• Small footprint
• Factorize computation: node sums over all paths
• Isolated the gradient computation to each node
Gradient Computation at Node

- Each node has a ComputeInputPartial function to compute partial derivatives with regard to its children.

- Basic rule:
  \[
  \frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}}
  \]

- Sigmoid: since
  \[
  \frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} 
  v_{ij} (1 - v_{ij}) & m = i \land n = j \\
  0 & \text{else}
  \end{cases}
  \]

  we have

  \[
  \frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial v_{ij}} v_{ij} (1 - v_{ij})
  \]

  and

  \[
  \nabla^J_x \leftarrow \nabla^J_x + \nabla^J_n \bullet [\nu \bullet (1 - \nu)]
  \]
Gradient Computation at Node

- *Times*: since
  
  \[
  \frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} 
  y_{jn} & m = i \\
  0 & \text{else}
  \end{cases}
  \]

  we have

  \[
  \frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \sum_{n} \frac{\partial J}{\partial v_{in}} y_{jn}
  \]

  or

  \[
  \nabla^J_x \leftarrow \nabla^J_x + \nabla^J_n y^T
  \]

  since

  \[
  \frac{\partial v_{mn}}{\partial y_{ij}} = \begin{cases} 
  x_{mi} & n = j \\
  0 & \text{else}
  \end{cases}
  \]

  we have

  \[
  \frac{\partial J}{\partial y_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial y_{ij}} = \sum_{m} \frac{\partial J}{\partial v_{mj}} x_{mi}
  \]

  or

  \[
  \nabla^J_y \leftarrow \nabla^J_y + x^T \nabla^J_n
  \]
Gradient Computation of CN

• Reverse automatic differentiation: call each node’s ComputeInputPartial function following the order below

1: procedure DecideGradientComputationOrder(node, parentsLeft, order)
   ▷ Decide the order to compute the gradient of all descendents of node.
   ▷ parentsLeft is initialized to the number of parents of each node.
   ▷ order is initialized as an empty queue.

2: if IsNotLeaf(node) then
3:     parentsLeft[node] ←
4:     if parentsLeft[node] == 0 ∧ node ∉ order then
5:         order ← order + node
6:     for each c ∈ node.children do
7:         call DecideGradientComputationOrder(c, parentsLeft, order)
8:     end for
9: end if
10: end if
11: end procedure

• Result is another computation graph that can be optimized (e.g., remove trivial computation, cache duplicate computation) or computed asynchronously
Gradient Computation Efficiency

- Gradient computation is not needed for some nodes.
  - Set $\text{NeedGradient}=\text{false}$ for constant leaf nodes.
  - Propagate the flag up the graph using the depth-first traversal.
  - Only compute the gradient when $\text{NeedGradient}=\text{true}$.

```plaintext
1: procedure UPDATE_NEED_GRADIENT_FLAG(root, visited)
   ▷ Enumerate nodes in the DAG in the depth-first order.
   ▷ $\text{visited}$ is initialized as an empty set.
   2:   if $\text{root} \notin \text{visited}$ then
       ▷ The same node may be a child of several nodes and
       revised.
       3:       $\text{visited} \leftarrow \text{visited} \cup \text{root}$
       4:       for each $c \in \text{root}.\text{children}$ do
       5:         call UPDATE_NEED_GRADIENT_FLAG($c$, visited, order)
       6:       if IsNotLeaf(node) then
       7:         if node.AnyChildNeedGradient() then
       8:             node.needGradient $\leftarrow \text{true}$
       9:       else
       10:          node.needGradient $\leftarrow \text{false}$
       11:      end if
       12:    end if
       13:  end for
   14:  end if
15: end procedure
```
Outline

• Introduction to Deep Learning and Prevailing Deep Learning Models

• Computational Network: A Unified Framework for Models Expressible as Functions

• Computational Network Toolkit: A Generic Toolkit for Building Computational Networks

• Examples: Acoustic Model, Language Model, and Image Classification

• Summary
Design Goals of CNTK

• Expression: models and algorithms are descriptions
  • Network definition language (NDL) via plain text
  • Build network via compositions in source code (Simple Network Builder)

• Modularity: to extend to new tasks and new models
  • Abstraction of computation nodes
  • Task-specific readers

• Speed: for state-of-the-art models trained on large data
  • Data parallelism
Computational Network Toolkit (CNTK)

Project Description
Computational networks (CNs) generalize models that can be described as a series of computational steps such as DNN, CNN, RNN, LSTM, and maximum entropy models.

Disclaimer
CNTK is a research code and ongoing project. There will be bugs in places.

Citation
If you used this toolkit or part of it to do your research, please cite the work as

- Source code and documents are available at http://cntk.codeplex.com
- Uses GIT for version control
- Supports Windows and Linux
A Typical Workflow of Using CNTK: train

• Train models use **train** command
A Typical Workflow of Using CNTK: eval

• Evaluate models using **eval** command
A Typical Workflow of Using CNTK: edit

- Edit models to expand the model with more nodes, fewer nodes, etc, using *edit* command.
A Typical Workflow of Using CNTK: update

• Update the newly edited model using `train` command
A Typical Workflow of Using CNTK: adapt

- Adapt models on new data using adapt command
A Typical Workflow of Using CNTK: write

• Write outputs from a node in the trained model using **write** command

• A node can be designed to output
  • Activity, e.g., used for training bottleneck features
  • Decoding results, e.g., used for output semantic tags in SLU example
CNTK Architecture

• Abstraction
**CNTK Architecture**

- **Network description**

---

![Diagram](image-url)
CNTK Architecture

- Task-specific readers
CNTK Architecture

• Learning algorithms

Diagram:
- LSTM, RNN, DNN
- ICNBuilder
- IDataReader
- ILearner
- Task-specific reader
- SGD/AdaGrad
- IExecutionEngine
- CN
- Features & Labels
- CN Description

Use, Build, Evaluate/Gradient, Load, Get data
CNTK Architecture

• Computing resources

[Diagram]
- LSTM, RNN, DNN
- CN Description
- Features & Labels
- ICNBuilder
- IDataReader
- Task-specific reader
- IExecutionEngine
- CPU/GPU
- CN
- ILearner
- SGD/AdaGrad
- Evaluate/Gradient
Implemented Nodes

• Inputs
  • Input, ImageInput, LookupTable

• One operand
  • ReLU, Sigmoid, Tanh, Log, Cos, Dropout, Negate, Softmax, LogSoftmax

• Matrices and Vectors
  • SumElements, RowSlice, Scale, Times, DiagTimes, Plus, Minus, ElementTimes

• Training criterion
  • SquareError, CrossEntropyWithSoftmax, ClassificationError, ClassBasedCrossEntropyWithSoftmax, GMMLogLikelihood

• Sequence-level training
  • CRF

• Parameters
  • Parameter, Constant

• Tensor
  • KhatriRaoProduct,

• Regularization
  • MatrixL1Reg, MatrixL2Reg,

• Normalization
  • Mean, InvStdDev, PerDimMVNorm

• CNN related
  • Convolution, MaxPooling, AveragePooling

• RNN related
  • Delay

• Bi-directional models related
  • TimeReverse
CNTK Task-specific Readers

- **UCIFastReader**
  - Space delimited file formats
  - uses BinaryReader to cache and speed up

- **HTKMLFReader**
  - Speech feature and labels in HTK format

- **KaldiReader**
  - Speech feature and labels in Kaldi format

- **LMSequenceReader**
  - Text file sequence reader for language model

- **LUSequenceReader**
  - Text file sequence reader for language understanding

- **DSSMReader**
  - For training and evaluating DSSM model for query and document pairs
TIMIT Example

- `cn.exe configFile=yourConfigFile.config DeviceNumber=1`

```plaintext
command=TIMIT_TrainNDL

TIMIT_TrainNDL=[
  action=train
  deviceId=${DeviceNumber}$
  modelPath=${your_model_path}$

  SimpleNetworkBuilder=[…]
  SGD=[…]
  reader=[…]
]
```

CPU: -1 or CPU
GPU: >=0
TIMIT DNN Network

```plaintext
SimpleNetworkBuilder = [
    layerSizes = 792:512*3:183
    applyMeanVarNorm = true
    trainingCriterion = CrossEntropyWithSoftmax
    evalCriterion = ErrorPrediction
]
```
TIMIT SGD

\[ g_t = m \cdot g_{t-1} - \alpha \Delta L(\theta_t) \]
\[ \theta_t = \theta_{t-1} + g_t \]

\[
\text{SGD} = \[
\text{minibatch Size}=256:1024 \\
\text{learning Rates Per MB}=0.8:3.2*14:0.08 \\
\text{momentum Per MB}=0.9
\]
\]

Epoch 1: minibatch size=256, learning rate= 0.8
Epoch 2-15: minibatch size = 1024, learning rate = 3.2
Epoch 16+: minibatch size = 1024, learning rate = 0.08
TIMIT Reader

```
reader=[
    readerType=HTKMLFReader

    randomize=Auto

    features=[
        dim=792
        scpFile=$FBankScpShort$
    ]

    labels=[
        mlfFile=$MlfDir$/TIMIT.train.mlf
        labelDim=183

        labelMappingFile=$MlfDir$/TIMIT.statelist
    ]
]
```
SLU Example : Reader

```
reader=[
    readerType=LUSequenceReader
    features=[
        dim=2832
    ]
    labelIn=[
        token=$DataDir$/input.txt
    ]
    wordContext=0:1:2
    labels=[
        token=$DataDir$/output.txt
    ]
]
```
SimpleNetworkBuilder=

```
rnnType=LSTM
layerSizes=2832:50:300:127
recurrentLayer=2
trainingCriterion=CrossEntropyWithSoftmax
evalCriterion=CrossEntropyWithSoftmax
```

Diagram:
- `CE`
- `labels`
- `127`
- `LSTM 300`
- `Projection 50x3`
- `features 2832`
- `\( w_t \) 2832`
- `\( w_{t+1} \) 2832`
- `\( w_t \) 2832`
Network Definition Language (NDL)

• Non-standard networks can be created using Network Definition Language (NDL)

• Nodes and connections can be specified through a series of atomic or macro operations
NDL Example: Single layer Auto-encoder

```
featDim=1000
hiddenDim=100

features=Input(featDim, tag=feature)

Wh = Parameter(hiddenDim, featDim)
Th = Times(Wh, features)
Sh = Sigmoid(Th)

Wo = Parameter(featDim, hiddenDim)
Po = Times(Wo, Sh)

MSE = SquareError(features, Po, tag=criteria)
```

Encode: Hidden Layer

Decoder: Output

Training Criterion
Using Macros in NDL

• Macros can be defined to encapsulate a common set of parameters and/or sequence of operations

\[ RFF(x_1, w_1, b_1) = \text{RectifiedLinear}(\text{Plus}(\text{Times}(w_1, x_1), b_1)) \]
\[ FF(X_1, W_1, B_1) \]
\[
\begin{cases} 
T &= \text{Times}(W_1, X_1) \\
FF &= \text{Plus}(T, B_1)
\end{cases}
\]

• Can access internal variables via the dot syntax, e.g.,

\[ L_1 = FF(X_1, W_1, B_1) \]
\[ L_1.T \] can be used to access the result of Times Op in the macro FF.
Example Macros

**Example Macros**

1. **FF**\((X_1, W_1, B_1)\)
   
   \[
   \begin{align*}
   &\{ \\
   &\quad T = \text{Times}(W_1, X_1) \\
   &\quad FF = \text{Plus}(T, B_1) \\
   \} \\
   \end{align*}
   \]

   Affine transformation

2. **BFF**\((\text{in}, \text{rows, cols})\)
   
   \[
   \begin{align*}
   &\{ \\
   &\quad B = \text{Parameter}(\text{rows, init=fixedvalue, value=0}) \\
   &\quad W = \text{Parameter}(\text{rows, cols}) \\
   &\quad BFF = \text{FF}(\text{in, W, B}) \\
   \} \\
   \end{align*}
   \]

   Affine transformation given row and column number

3. **SBFF**\((\text{in, rows, cols})\)
   
   \[
   \begin{align*}
   &\{ \\
   &\quad BFF = \text{BFF}(\text{in, rows, cols}) \\
   &\quad SBFF = \text{Sigmoid}(BFF) \\
   \} \\
   \end{align*}
   \]

   Sigmoid transformation given row and column number
Macros in Effect: Auto-encoder Example

featDim=1000
hiddenDim=100

features=Input(featDim, tag=feature)

L1 = SBFF(features, hiddenDim, featDim)

L2 = BFF(L1, featDim, hiddenDim)

MSE = SquareError(features, L2, tag=criteria)
Invoking NDL Networks

• Invoked in the configuration file using **NDLNetworkBuilder**

• Each NDL file can contain many network definitions
  • Use **run** command to determine which network to use

```perl
NDLNetworkBuilder=[
    ndlMacros=$Nd1Dir$/default_macros.ndl
    AutoEncoderNDL=$Nd1Dir$/mynetwork.ndl
    run=AutoEncoderNDL
]
```
Stochastic Gradient Descent Learner

• Compute gradient of objective function with respect to model parameters.
  • Gradient descent uses the entire training set.
    • Provable linear convergence.
  • SGD uses a random subset (minibatch) of the training set.
    • Convergence requires decreasing learning rates.
    • Provable sub-linear convergence bound.
    • In practice, convergence only slow as model approaches the optimum.
    • Beneficial when training time is the training bottleneck.

• Parameter update is a function of this gradient.
SGD Learner Configuration

• Gradient Update Options
  • Options for CNTK are: none, adagrad, or rmsprop.

• Learning rate search

• Model Averaging (parallel training)
  • Compile code with USE_MPI defined.
  • At the end of every epoch, all nodes exchange and average their parameters.

• Regularization
  • Gradient Noise
    • Adds Gaussian noise to each gradient computation.
  • L2 regularizer
    • Adds a scaled version of the parameters into the gradient, biasing parameter values to zero.
  • L1 regularizer
    • Uses the proximal gradient descent algorithm to shrink the weights
Default Gradient Update

• Conventional SGD is the default
  • Apply learning rate to gradients.
  • Apply momentum to gradients.
  • Subtract result from parameter values.

• Equivalent to setting “Gradient update: none”
Gradient Update: Adagrad

• Scales each dimension by the $l_2$ norm of all gradients for that dimension up to, and including, the current time.

$$\beta_t[k] = \beta_{t-1}[k] + d_t^2[k], \quad d_t[k] \leftarrow \frac{d_t[k]}{\sqrt{\beta_t[k] + \epsilon}}$$

• Effective gradient scale factor is non-increasing over time.

• Optional: As a final step, the average multiplier over all features is found, and used to re-scale the gradient.

$$m = \sum_{k=0}^{K-1} \frac{1}{\sqrt{\beta_t[k] + \epsilon}}, \quad d_t[k] \leftarrow \frac{d_t[k]}{m}$$

• This preserves the absolute scale of the gradient, retaining only the relative scaling of adagrad.
Gradient Update: RMSProp

- Per-dimension scale factor with two components.
  \[ d_t[k] \leftarrow d_t[k] \frac{r_t[k]}{\sqrt{\beta_t[k] + \epsilon}} \]
  - A smoothed estimate of the \( l_2 \) norm of recent gradients for that dimension up to, and including, the current time.
  \[ \beta_t[k] = \gamma \beta_{t-1}[k] + (1 - \gamma)d_t^2[k] \]
  - An rprop style factor, that increases when the gradient sign matches from one time to the next, and decreases otherwise.
  \[ r_t[k] \leftarrow \begin{cases} 
  \min(r_{t-1}[k] \cdot w_{inc}, w_{max}) & \text{if signs match} \\
  \max(r_{t-1}[k] \cdot w_{dec}, w_{min}) & \text{otherwise} 
\end{cases} \]

- Parameters
  - rms_gamma: Smoothing factor for exponential window variance estimate.
  - rms_wgt_inc, rms_wgt_dec: Factors to multiply each weight by, to increase or decrease its value.
  - rms_wgt_max, rms_wgt_min: Ceiling and floor for the weight factors.
  - As in adagrad, the average multiplier over all features is found, and used to re-scale the gradient.
  \[ m = \sum_{k=0}^{K-1} \frac{r_t[k]}{\sqrt{\beta_t[k] + \epsilon}}, \quad d_t[k] \leftarrow \frac{d_t[k]}{m} \]
Model Editing Language

• Sometimes it is useful to edit a model
  • Copy parameters from one model or node to another
  • Overwrite parameters
  • Freeze or unfreeze certain parameters during training

• This can be done using CNTK’s Model Editing Language (MEL)

• Examples of MEL functionality:
  • Load multiple CNTK models
  • Copy parameters, a node, or a group of nodes
  • Add new nodes to existing model using “inline” NDL
  • Change node properties, e.g. needGradient or criteria node
MEL example

• MEL can be used to do layer-by-layer discriminative pre-training (DPT)
  • Construct a 1 layer model
  • Train model
  • Use MEL to remove output layer and add a new hidden layer and a new output layer
  • Train new model
  • Repeat until desired depth is reached
MEL Example: DPT

X

L1.T = Times(W1, X)
L1.P = Plus(L1.T, b1)
L1.S = Sigmoid(L1.P)

L2.T = Times(W2, L1.S)
L2.P = Plus(L2.T, b0)
L2.S = Sigmoid(L2.P)

CE.T = Times(WO, L2.S)
CE.P = Plus(CE.T, b0)
CE.S = Softmax(CE.P)

MODIFY
CREATE
MEL Example: DPT

• Configuration file to perform edit operation:

```mel
AddLayer2=[
    action=edit
    CurrModel=./cntkSpeech.dnn
    NewModel=./cntkSpeech.dnn.0
    editPath=./add_layer.mel
]
```

• MEL commands to add a layer to the current model

```mel
m1=LoadModel($CurrModel$, format=cntk)
SetDefaultModel(m1)
HDim=512
L2=SBFF(L1.S,HDim,HDim) #CREATE
SetInput(CE.*.T, 1, L2.S) #MODIFY
SetInput(L2.*.T, 1, L1.S)
SaveModel(m1,$NewModel$, format=cntk)
```
Extending the functionality of CNTK

• You can add a new data reader either to speed up loading time or to support a new data format by implementing the `IDataReader` interface with two key operations
  • `StartMinibatchLoop()`
  • `GetMinibatch()`

• You can add a new node type derived from class `ComputationNode` by implementing
  • `EvaluateThisNode()`
  • `ComputeInputPartial()`

And adding the related instantiation methods to the `ComputationNetwork` and `NetworkDefinitionLanguage` classes
Outline

• Motivation

• Introduction to Deep Learning and Prevailing Deep Learning Models

• Computational Network: A Unified Framework for Models Expressible as Functions

• Computational Network Toolkit: A Generic Toolkit for Building Computational Networks

• Examples: Acoustic Model, Language Model, and Image Classification

• Summary
DNN-HMM AM Example

• Design criteria
  • Frame stacking. Input layer is multiple of feature size.
  • Hidden layer width.
  • Model depth.

• Input data
  • Label files (HTK MLF format)
    • start_frame end_frame label_string label_index
  • Traditional script files pointing to HTK parameters
    • `\path\to\test\dr1\faks0\si1573.fbank_zda`
  • Or, HTK archive format script files
    • `test-dr1-faks0-si1573.fbank_zda=\path\to\big\file.fbank_zda[0,494]`
    • `test-dr1-faks0-si2203.fbank_zda=\path\to\big\file.fbank_zda[495,843]`
DNN-HMM AM Example

TIMIT_TrainSimple =
    action = train
    modelPath = $ExpDir$\cntkSpeech.dnn
    SimpleNetworkBuilder = 
        layerSizes = 792:512*3:183
        trainingCriterion = 
            CrossEntropyWithSoftmax
        evalCriterion = ErrorPrediction
        layerTypes = Sigmoid
        initValueScale = 1.0
        applyMeanVarNorm = true
        uniformInit = true
        needPrior = true
    ]
SGD= [
    epochSize = 0
    minibatchSize = 256:1024
    learningRatesPerMB = 
        0.8:3.2*14:0.08
    momentumPerMB = 0.9
    maxEpochs = 25
    ]

reader = [
    readerType = HTKMLFReader
    readMethod = rollingWindow
    miniBatchMode = Partial
    randomized = Auto
    verbosity = 1
    features = [ 
        dim = 792
        scpFile = $ScpDir$\TIMIT.train.scp
    ]
    labels = [ 
        mlfFile = $MlfDir$\TIMIT.mlf
        labelDim = 183
        labelMappingFile = $MlfDir$\TIMIT.statelist
    ]
]
DNN-HMM AM Example: LSTM

\[
\begin{align*}
    i_t &= \sigma \left( W^{(xi)} x_t + W^{(hi)} h_{t-1} + W^{(ci)} c_{t-1} + b^{(i)} \right) \\
    f_t &= \sigma \left( W^{(xf)} x_t + W^{(hf)} h_{t-1} + W^{(cf)} c_{t-1} + b^{(f)} \right) \\
    c_t &= f_t \cdot c_{t-1} + i_t \cdot \tanh \left( W^{(xc)} x_t + W^{(hc)} h_{t-1} + b^{(c)} \right) \\
    o_t &= \sigma \left( W^{(xo)} x_t + W^{(ho)} h_{t-1} + W^{(co)} c_t + b^{(o)} \right) \\
    h_t &= o_t \cdot \tanh (c_t) ,
\end{align*}
\]
DNN-HMM AM Example: LSTM

LSTMComponent(inputDim, outputDim, inputVal)
{
    Wxo = Parameter(outputDim, inputDim)
    Wxi = Parameter(outputDim, inputDim)
    Wxf = Parameter(outputDim, inputDim)
    Wxc = Parameter(outputDim, inputDim)
    bo = Parameter(outputDim, init=fixedvalue, value=-1.0)
    bc = Parameter(outputDim, init=fixedvalue, value=0.0)
    bi = Parameter(outputDim, init=fixedvalue, value=-1.0)
    bf = Parameter(outputDim, init=fixedvalue, value=-1.0)
    Whi = Parameter(outputDim, outputDim)
    Wci = Parameter(outputDim)
    Whf = Parameter(outputDim, outputDim)
    Wcf = Parameter(outputDim)
    Who = Parameter(outputDim, outputDim)
    Wco = Parameter(outputDim)
    Whc = Parameter(outputDim, outputDim)
}
DNN-HMM AM Example: LSTM

\[
delayH = \text{Delay}(\text{outputDim}, \text{output}, \text{delayTime}=1) \\
delayC = \text{Delay}(\text{outputDim}, \text{ct}, \text{delayTime}=1)
\]

\[
\text{WxiInput} = \text{Times}(\text{Wxi}, \text{inputVal}) \\
\text{WhidelayHI} = \text{Times}(\text{Whi}, \text{delayH}) \\
\text{WcidelayCI} = \text{DiagTimes}(\text{Wci}, \text{delayC})
\]

\[
i_t = \sigma(\text{W}^{(x)} x_t + \text{W}^{(hi)} h_{t-1} + \text{W}^{(ci)} c_{t-1} + b^{(i)})
\]

\[
it = \text{Sigmoid} (\text{Plus} (\text{Plus} (\text{Plus} (\text{WxiInput, bi}), \text{WhidelayHI}), \text{WcidelayCI}))
\]

\[
\text{WhfdelayHF} = \text{Times}(\text{Whf}, \text{delayH}) \\
\text{WcfdelayCF} = \text{DiagTimes}(\text{Wcf}, \text{delayC}) \\
\text{Wxfinput} = \text{Times}(\text{Wxf}, \text{inputVal})
\]

\[
f_t = \sigma(\text{W}^{(xf)} x_t + \text{W}^{(hf)} h_{t-1} + \text{W}^{(cf)} c_{t-1} + b^{(f)})
\]

\[
ft = \text{Sigmoid} (\text{Plus} (\text{Plus} (\text{Plus} (\text{Wxfinput, bf}), \text{WhfdelayHF}), \text{WcfdelayCF}))
\]
DNN-HMM AM Example: LSTM

\[ \text{WxcInput} = \text{Times}(\text{Wxc}, \text{inputVal}) \]
\[ \text{WhcdelayHC} = \text{Times}(\text{Whc}, \text{delayH}) \]
\[ \text{bit} = \text{ElementTimes}(\text{it}, \text{Tanh}(\text{Plus}(\text{WxcInput}, \text{Plus}(\text{WhcdelayHC}, \text{bc})))) \]
\[ \text{bft} = \text{ElementTimes}(\text{ft}, \text{delayC}) \]
\[ \text{ct} = \text{Plus}(\text{bft}, \text{bit}) \]
\[ \text{ct} = \frac{c_t}{c_t} = f_t \cdot c_{t-1} + i_t \cdot \tanh \left( W^{(xc)} x_t + W^{(hc)} h_{t-1} + b \right) \]
\[ \text{Wxoinput} = \text{Times}(\text{Wxo}, \text{inputVal}) \]
\[ \text{WhodelayHO} = \text{Times}(\text{Who}, \text{delayH}) \]
\[ \text{Wcoct} = \text{DiagTimes}(\text{Wco}, \text{ct}) \]
\[ \text{ot} = \text{Sigmoid}(\text{Plus}(\text{Plus}(\text{Plus}(\text{Wxoinput}, \text{bo}), \text{WhodelayHO}), \text{Wcoct})) \]
\[ \text{output} = \text{ElementTimes}(\text{ot}, \text{Tanh}(\text{ct})) \]
\[ \text{h}_t = o_t \cdot \tanh(c_t) \]
DNN-HMM AM Example: LSTM

• Top level command:

```action=train```

• Reader changes

```reader=[
    readerType=HTKMLFReader
    readMethod=blockRandomize
    frameMode=false
    Truncated=true
    nbrUttsInEachRecurrentIter=32```

• SGD block meaning change

```minibatchsize=20```

- Sequence mode
- Truncated BPTT
- Number of utterances in each minibatch
- Truncation size
Prediction-Based AM Example

- A recurrent system with two major components
- Predict, adapt, and correct
Prediction Based AM Example

#define basic i/o

featDim=1845
labelDim=183
labelDim2=61
hiddenDim=1024
bottleneckDim=80
bottleneckDim2=500
features=Input(featDim, tag=feature)
labels=Input(labelDim, tag=label)
statelabels=Input(labelDim2, tag=label)
ww=Constant(1)

4/19/2015


98
Prediction Based AM Example

# define network
featNorm = MeanVarNorm(features)

DNN_A_delayfeat1=Delay(bottleneckDim, DNN_B_L2.BFF.FF.P, delayTime=1)
DNN_A_delayfeat2=Delay(bottleneckDim, DNN_B_L2.BFF.FF.P, delayTime=2)
DNN_A_delayfeat3=Delay(bottleneckDim, DNN_B_L2.BFF.FF.P, delayTime=3)
DNN_A_delayfeat4=Delay(bottleneckDim, DNN_B_L2.BFF.FF.P, delayTime=4)
DNN_A_delayfeat5=Delay(bottleneckDim, DNN_B_L2.BFF.FF.P, delayTime=5)
DNN_A_delayfeat6=Delay(bottleneckDim, DNN_B_L2.BFF.FF.P, delayTime=6)
DNN_A_delayfeat7=Delay(bottleneckDim, DNN_B_L2.BFF.FF.P, delayTime=7)
DNN_A_delayfeat8=Delay(bottleneckDim, DNN_B_L2.BFF.FF.P, delayTime=8)
DNN_A_delayfeat9=Delay(bottleneckDim, DNN_B_L2.BFF.FF.P, delayTime=9)
DNN_A_delayfeat10=Delay(bottleneckDim, DNN_B_L2.BFF.FF.P, delayTime=10)
DNN_A_delayfeat=Delay(labelDim, DNN_B_CE_BFF.FF.P, delayTime=10)
Prediction Based AM Example

\[
\begin{align*}
DNN_A_L1 &= \text{SBFF}_\text{multi8}(\text{featNorm}, DNN_A\_delayfeat1, \\
&\quad \quad DNN_A\_delayfeat2, DNN_A\_delayfeat3, DNN_A\_delayfeat4, \\
&\quad \quad DNN_A\_delayfeat5, DNN_A\_delayfeat6, DNN_A\_delayfeat7, \\
&\quad \quad DNN_A\_delayfeat8, DNN_A\_delayfeat9, DNN_A\_delayfeat10, \\
&\quad \quad \text{hiddenDim}, \text{featDim}, \text{bottleneckDim}) \\
DNN_A_L2 &= \text{SBFF}(DNN_A_L1, \text{hiddenDim}, \text{hiddenDim}) \\
DNN_A_L2_B &= \text{SBFF}(DNN_A_L1, \text{bottleneckDim2}, \text{hiddenDim}) \\
DNN_A_CE_BFF &= \text{BFF}(DNN_A_L2, \text{labelDim}, \text{hiddenDim}) \\
DNN_B_L1 &= \text{SBFF}_\text{multi}(\text{featNorm}, DNN_A_L2_B.BFF.FF.P, \\
&\quad \quad \text{hiddenDim}, \text{featDim}, \text{bottleneckDim2}) \\
DNN_B_L2 &= \text{SBFF}(DNN_B_L1, \text{bottleneckDim}, \text{hiddenDim}) \\
DNN_B_CE_BFF &= \text{BFF}(DNN_B_L2, \text{labelDim2}, \text{bottleneckDim})
\end{align*}
\]

Concatenate information from past
Prediction Based AM Example

criterion1 = CrossEntropyWithSoftmax(labels, DNN_A_CE_BFF)

criterion2 = CrossEntropyWithSoftmax(statelabels, DNN_B_CE_BFF)

criterion = Plus(Scale(cr2, criterion2), Scale(cr1, criterion1), tag=Criteria)

Err = ErrorPrediction(labels, DNN_A_CE_BFF, tag=Eval)

LogPrior = LogPrior(labels)

ScaledLogLikelihood = Minus(DNN_A_CE_BFF, logPrior, tag=Output)
Prediction Based AM Example

reader=[
    readerType=HTKMLFReader
    readMethod=blockRandomize
    frameMode=false
    Truncated=true
    nbruttsineachcurrentiter=5
    features=[
        dim=1845
        scpFile=$scpFilePath$
    ]
]

labelDim=183
labelType=Category
labels=[
    mlfFile=$normalLabelFilePath$
    statelabels=[
        mlfFile=$predictLabelFilePath$
    ]
]

Utterance mode for RNN
Truncated BPTT
Main label
Prediction label
# Sample, Hidden, and Label dimensions
SDim=784
LDim=10

inputWidth=28
inputHeight=28
inputChannels=1

features=ImageInput(inputWidth, inputHeight, inputChannels, tag=feature)
labels=Input(LDim, tag=label)

# convolution
kernelWidth=5
kernelHeight=5
outputChannels=24
horizontalSubsample=2
verticalSubsample=2
# weight[outputChannels, kernelWidth * kernelHeight * inputChannels]
cvweight=Parameter(outputChannels, 25)

cv = Convolution(cvweight, features, kernelWidth, kernelHeight, outputChannels, horizontalSubsample, verticalSubsample, zeroPadding=false)

# one bias per channel
cvbias=Parameter(outputChannels, 1)

cvplusbias=Plus(cv, cvbias);
nlcv=Sigmoid(cvplusbias);

#outputWidth = (m_inputWidth-m_kernelWidth)/m_horizontalSubsample + 1;
outputWidth=12

#outputHeight = (m_inputHeight-m_kernelHeight)/m_verticalSubsample + 1;
outputHeight=12
NDL Example: CNN

```plaintext
#maxpooling
windowWidth=2
windowHeight=2
stepW=2
stepH=2
mp=MaxPooling(n1cv, windowWidth, windowHeight, stepW, stepH)

#m_outputSizePerSample = m_outputWidth * m_outputHeight * m_channels;
mpoutputSizePerSample=864

# Layer operations
HDim=128
L1 = SBFF(mp, HDim, mpoutputSizePerSample)
CE = SMBFF(L1, LDim, HDim, labels, tag=Criteria)
Err=ErrorPrediction(labels, CE.BFF, tag=Eval)

# rootNodes defined here
OutputNodes=(CE.BFF)
```
Class-based RNN LM Example

• RNN example with additional class info included in output layer

\[ w(t) \rightarrow s(t) \rightarrow y(t) \]

\[ s(t-1) \rightarrow c(t) \]
Class-based RNN LM Example

• Top level commands and parameters

```coffee
action=train
minibatchSize=10
deviceId=auto
defaultHiddenActivity=0.1
```

The initial (default) activity for delay nodes

• Network Definition

```python
ndlCreateNetwork=[
    featDim=10000
    labelDim=10000
    hiddenDim=200
    nbrClass=50
    initScale=6
    features=SparseInput(featDim, tag=feature)
]
```

Vocabulary Size

Hidden Layer Size

Number of Classes

Enlarge initial weight value ranges by 6 times

# labels in classbasedCrossEntropy is dense
# and contain 4 values for each sample
labels=Input(4, tag=label)
Class-based RNN LM Example

WFeat2Hid=Parameter(hiddenDim, featDim, init=uniform, initValueScale=initScale)
WHid2Hid=Parameter(hiddenDim, hiddenDim, init=uniform, initValueScale=initScale)

# WHid2Word is special that it is hiddenSize X labelSize
WHid2Word=Parameter(hiddenDim, labelDim, init=uniform, initValueScale=initScale)
WHid2Class=Parameter(nbrClass, hiddenDim, init=uniform, initValueScale=initScale)

PastHid = Delay(hiddenDim, HidAfterSig, delayTime=1, needGradient=true)
HidFromFeat = Times(WFeat2Hid, features)
HidFromRecur = Times(WHid2Hid, PastHid)
HidBeforeSig = Plus(HidFromHeat, HidFromRecur)
HidAfterSig = Sigmoid(HidBeforeSig)

Out = Times(WHid2Word, HidAfterSig) #word part
ClassProbBeforeSoftmax=Times(WHid2Class, HidAfterSig)

cr = ClassBasedCrossEntropyWithSoftmax(labels, HidAfterSig, WHid2Word, ClassProbBeforeSoftmax, tag=Criteria)
EvalNodes=(Cr)
OutputNodes=(Cr)

[ ]

Customized Initialization
Info from Past Hidden
Special Node for Class Based Classification
Same Node Can be Used for Difference Purposes
Class-based RNN LM Example

• SGD section

```
SGD=[
    learningRatesPerSample=0.1
    momentumPerMB=0
    gradientClippingWithTruncation=true
    clippingThresholdPerSample=15.0
    maxEpochs=40
    gradUpdateType=None

    modelPath=$ExpFolder$/modelRnnCNTK
    loadBestModel=true

    # settings for Auto Adjust Learning Rate
    AutoAdjust=[
        autoAdjustLR=adjustAfterEpoch
        reduceLearnRateIfImproveLessThan=0.001
        continueReduce=true
        learnRateDecreaseFactor=0.5
    ]
]
```

- Clip the gradient to prevent exploding
- load best model before training next epoch
- Adjust learning rate after each epoch
- Continue reducing learning rate once it’s reduced
Outline

• Motivation
• Introduction to Deep Learning and Prevailing Deep Learning Models
• Computational Network: A Unified Framework for Models Expressible as Functions
• Computational Network Toolkit: A Generic Toolkit for Building Computational Networks
• Examples: Acoustic Model, Language Model, and Image Classification
• Summary
Summary

• Computational networks generalize many existing deep learning models
• You may design new computational networks to attack new problems by exploiting problem-specific structures and domain knowledge
• We described the forward and backward computation algorithms and theories for CNs with and without recurrent loops
• CNTK implements CNs so that you only need to focus on designing the CNs instead of implementing learning algorithms for your specific CN
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

• Network definition language, macros, and model editing language makes network design and modification easy

• We have shown many examples to indicate that CNTK can support DNN, CNN, RNN, class-based LM, LSTM, and PAC-AM and to solve AM, LM, and SLU problems
Additional Resources

• CNTK Reference Book
  • Contains all the information you need to understand and use CNTK

• Codeplex 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
Please Contribute!

• If you write your own readers or computation nodes we would like them to be checked-in to the main branch

• CNTK becomes more powerful when all computation nodes and readers people want to use are available