Deep Learning for Automatic Speech Recognition

Dong Yu
Microsoft Research
Outline

• Major Modeling Techniques in ASR
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Long Short-Term Memory (LSTM) RNNs
  • Model Adaptation

• Build Deep ASR Models Using CNTK
  • Computational Networks and CNTK
  • Network Definition Language (NDL)
  • Examples: DNN, LSTM and PAC-RNN
  • Model Editing Language (MEL)
  • Stochastic Gradient Descent (SGD)

• Summary
ANN/HMM Hybrid System
(Morgan and Bourlard 1990, 1995)

- Used a large context window as inputs
- Models monophone state posterior probability with a neural network
- One-to-two hidden layers. Hidden layer size is comparable with what we use today (but not the output layer size)
- Can easily handle continuous speech due to the integration of HMM
DNN/HMM Hybrid System  
(Mohamed, Dahl and Hinton 2009)

• DNN on phoneme recognition  
  • TIMIT phone recognition:  
    • DNN System: 23.0% phone error rate (PER)  
    • Ref: GMM systems  
      • maximum likelihood training (MLT) 25.6%,  
      • sequence-discriminative training (SDT) 21.7%

• Same architecture as 1990s but deep and pretrained: models monophone states, frame-discriminative training, MFCC

• Observation: Deep network helps; pretraining helps; has potential

<table>
<thead>
<tr>
<th>GMM MLT</th>
<th>CI-DNN FDT 23.0%</th>
<th>GMM SDT 21.7%</th>
<th>Best GMM 20.0%</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.6%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CD-DNN-HMM

• DNN on large vocabulary ASR
  • Architectural Difference: model tied triphone states (senones) directly with DNN
• On voice search (24 hr) dataset
  • CI-DNN-HMM modeling monophone states, frame-discriminative training (FDT): 37.3% word error rate (WER)
  • Context-Dependent DNN-HMM (CD-DNN-HMM) modeling senones, FDT: 30.1%
• Ref: GMM: MLT 39.6%; SDT 36.2%

GMM MLT 39.6%  CI-DNN FDT 37.3%  GMM SDT 36.2%  CD-DNN FDT 30.1%  better
CD-DNN-HMM
(Kingsbury 2009)

• CD-DNN-HMM on benchmark task Switchboard (FDT)
  • Scaled to hundreds of hours of speech and thousands of senones
  • Confirmed the three key elements on larger training set: modeling senones directly, using deep models, and using a contextual window of features as input

<table>
<thead>
<tr>
<th>Layers X Neurons</th>
<th>WER (%) [300hrs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 2k</td>
<td>24.2</td>
</tr>
<tr>
<td>3 x 2k</td>
<td>18.4</td>
</tr>
<tr>
<td>5 x 2k</td>
<td>17.2</td>
</tr>
<tr>
<td>7 x 2k</td>
<td>17.1</td>
</tr>
<tr>
<td>9 x 2k</td>
<td>17.0</td>
</tr>
<tr>
<td>1 x 16k</td>
<td>22.1</td>
</tr>
</tbody>
</table>

1/3 error cut

7/1/2015
Dong Yu: Deep Learning for Automatic Speech Recognition
DNN Sequence Discriminative Training
(Seide, Li & Yu 2011, Mohamed, Yu & Deng 2010, Kingsbury, Sainath & Soltau 2012)

• Sequence Discriminative Training on CD-DNN-HMM
  • Lattice generation: generate lattice with your best system (e.g., FDT CD-DNN-HMM instead of MLT CD-GMM-HMM) or generate lattices during SDT using the current best model
  • Lattice compensation: handle run-away silence frames, augment lattice with reference transcription, reject bad frames
  • Over-fit control: smooth the SDT criterion with the FDT criterion
  • Learning rate control: use 1/5-1/10 of the learning rate used in the FDT
  • Training criterion: SDT training criterion used does not have huge effect on performance; MMI is simple to implement and thus preferred

GMM SDT 23.6%
CD-DNN FDT 16.1%
BEST GMM 14.5%
CD-DNN SDT 13.3%
1/6 error cut
Why DNNs Work
(Seide, Li, Chen & Yu 2011)

• DNN learns the log-linear classifier and the complicated feature transformation jointly

• Many simple nonlinearities combine to form complex nonlinearities for better feature transformation

• DNN is more robust to speaker variations than shallow models

• Feature engineering techniques (e.g., VTLN, fMLLR) help less in deep networks than in shallow models

• Hint: can rewind many feature processing steps usually done in the GMM system, has no assumption on input features
Why DNNs Become Effective Only Now?

- For highly constrained tasks simple solution works
- Lack of training data and computation power limited the potential of some models

Tiger or Cat?

Pictures are from web
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)
Outline

• Major Modeling Techniques in ASR
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Long Short-Term Memory (LSTM) RNNs
  • Model Adaptation

• Build Deep ASR Models Using CNTK
  • Computational Networks and CNTK
  • Network Definition Language (NDL)
  • Examples: DNN, LSTM and PAC-RNN
  • Model Editing Language (MEL)
  • Stochastic Gradient Descent (SGD)

• Summary
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

Detect and sum

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]
**TDNN**
(Waibel et al. 1989)

- Convolution over time
- Top layer integration allows for handling variable-length inputs
- Used melscale filterbank coefficient as the inputs.
- Used two hidden layers
- Difficult to handle large vocabulary continuous speech recognition
CNN-HMM


• On TIMIT phone recognition task:
  • CNN with log filter-bank (LFB) features: 20.0% PER
  • Ref: DNN with LFB features 20.7%

• Key technique: Use CNN at the frequency axis to normalize the speaker differences. Only feasible with LFB features
CNN-HMM

• On LVCSR
  • Voice search (18 hr training) FDT: 33.4% WER with CNN vs 35.4% with DNN
  • Switchboard (309 hr training) SDT: 11.8% WER with CNN vs 12.2% with DNN

• Combine CNN and DNN (IBM)
  • Switchboard (309 hr) CNN+DNN+Adaptation+SDT 10.4%
  • Switchboard (2000 hr) CNN+DNN+Adaptation+SDT+Stronger LM 8.0%
  • Ref: best number with all tricks and adaptation techniques using GMM is 14.5%
Limitations of CNNs

• CNNs mainly deal with translational variability
• It cannot handle variations such as
  • 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
• More importantly, CNNs cannot take advantage dependencies and correlations between samples (and labels) in a sequence
• Recurrent neural networks (RNNs) are designed for this
Outline

• **Major Modeling Techniques in ASR**
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Long Short-Term Memory (LSTM) RNNs
  • Model Adaptation

• **Build Deep ASR Models Using CNTK**
  • Computational Networks and CNTK
  • Network Definition Language (NDL)
  • Examples: DNN, LSTM and PAC-RNN
  • Model Editing Language (MEL)
  • Stochastic Gradient Descent (SGD)

• **Summary**
Recurrent Neural Networks

- Models dependencies and correlations between samples (and labels) in a sequence
- 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 (LSTM)

• An extension of RNN that addresses vanishing gradient problem
  • Memory cell is linearly time-recurrent
  • Use gates to control and keep long-range information
Long Short-Term Memory (LSTM)
(Graves, Mahamed, Hinton 2013, Graves, Jaitly, Mahamed 2013)

• LSTM for phone recognition
  • **Key techniques**: Bidirectional LSTM, Connectionist Temporal Classification (CTC)
  • TIMIT phone recognition: 18.4% PER (with same LM)
    • Ref: CNN-HMM 20.0%

• LSTM-HMM for LVSR
  • LSTM-HMM FDT: WSJ 11.7% WER
  • Ref: CD-DNN-HMM 12.3%
LSTM-HMM
(Sak, Senior, Beaufays 2014, Sak et al. 2014)

- Long-Short Term Memory
  - Key techniques: LSTM-HMM with a projection layer, delay output label by 5 frames, single-frame input, SDT
  - 9.8% WER on VS
    - Ref: 10.4% WER using DNN
LSTM Direct Model on LVCSR
(Sak et al. 2015)

• Use connectionist temporal classification (CTC)
  • Introduce a blank symbol to indicate “uncertain to emit a phone”: blank blank a = a
  • Allow repetition of symbols: a a a = a
  • Optimize the sum of log probability of all valid state sequences - no alignment is needed: abc → blank blank a a b blank blank c c c blank
  • Initialize CTC model with CE trained model. Replace the existing softmax layer with a new random softmax layer – can be considered as a pretraining step.
  • Need to clip both the gradient and activation of the memory cell to make training stable.
  • Apply sequence discriminative training on top of CTC
• A bidirectional LSTM RNN CTC model using phone units perform as well as an LSTM RNN model using HMM state alignments
Outline

• **Major Modeling Techniques in ASR**
  - Deep Neural Networks (DNNs)
  - Convolutional Neural Networks (CNNs)
  - Long Short-Term Memory (LSTM) RNNs
  - Model Adaptation

• **Build Deep ASR Models Using CNTK**
  - Computational Networks and CNTK
  - Network Definition Language (NDL)
  - Examples: DNN, LSTM and PAC-RNN
  - Model Editing Language (MEL)
  - Stochastic Gradient Descent (SGD)

• **Summary**
DNN Adaptation (Auxiliary Info)

• Noise (or speaker)-aware training
  • Key technique: estimate noise (or speaker, e.g., i-vector and speaker code) and use it as part of the input to the DNN
  • Effectively used a different (adaptive) bias for different noise condition

• DNN FDT on Aurora4 13.4% WER, + noise-aware training 12.4%
  • Ref: GMM: SDT 22.5%; +adaptive training 15.3%, +VAT+Joint compensation 13.4%

• DNN SDT on SWB 14.1%; + SaT: 12.4% WER \(\rightarrow\) 12% error cut
DNN with Auxiliary Information

- Augment the raw feature with auxiliary information
- Auxiliary information needs to be provided separately
PAC-RNN: Predict Auxiliary Info

- Predict the auxiliary information
- Both components are jointly trained
PAC-RNN: Feedback

• Forms a recurrent network
• Both components are adaptive and trained jointly
• Better than LSTM and DNN on both TIMIT phone recognition and babel evaluation
New Models

• To further improve the accuracy new models are needed
• This requires new tools
Outline

• Major Modeling Techniques in ASR
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Long Short-Term Memory (LSTM) RNNs
  • Model Adaptation

• Build Deep ASR Models Using CNTK
  • Computational Networks and CNTK
  • Network Definition Language (NDL)
  • Examples: DNN, LSTM and PAC-RNN
  • Model Editing Language (MEL)
  • Stochastic Gradient Descent (SGD)

• Summary
Computational Networks

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

• Examples of computational networks
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Recurrent Neural Networks (RNNs)
  • Long Short-Term Memory (LSTM) RNNs
  • PAC-RNN

• …and some other common machine learning models
  • Gaussian Mixture Models (GMMs)
  • Logistic Regression Models (LRMs)
  • Log-Linear Models (LLMs)
Example: One Hidden Layer NN

Output Layer

- O: Softmax
- P(2)

Hidden Layer

- S(1): Sigmoid
- P(1)
- X: Input

Weights:
- W(1), b(1)
- W(2), b(2)
Example: CN with Multiple Inputs
Example: CN with Shared Parameters
Example: CN with Recurrence
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.

- **Supports Windows and Linux**
Source Code Enlist

• Source code repository
  • http://cntk.codeplex.com

• Enlist instruction:
  • https://cntk.codeplex.com/documentation

• Uses git for source version control management
  • Suggest to read git introduction and manual
  • Suggest to install git-extension for visual studio http://code.google.com/p/gitextensions/
  • Much better than the Git manager integrated in VS 2013.

• Clone the source code
  • git clone https://git01.codeplex.com/cntk (on Windows)
  • git clone https://git.codeplex.com/cntk and then git checkout linux-gcc (on Linux to check out the Linux branch)
Prerequisites to Build CNTK

- Compilers:
  - Windows: Visual Studio 2013 or above (since project and solution files are in VS 2013 format)
  - Linux: g++ 4.8.3 (or above)

- CPU BLAS library (do one of the followings)
  - Install the ifort64 variant (e.g., acml5.3.1-ifort64.exe) of ACML 5.3.1 or above from http://developer.amd.com/tools/cpu-development/amd-core-math-library-acml/acml-downloads-resources/ (free). Set the system environment variable ACML_PATH to C:\AMD\acml5.3.1\ifort64_mp or the folder you installed ACML
  - Install Intel MKL library from https://software.intel.com/en-us/intel-math-kernel-library-evaluation-options and define USE_MKL in the CNTKMath project (requires license)

- GPU library
  - Install CUDA 7.0 from https://developer.nvidia.com/cuda-downloads
  - If you don’t want to install CUDA (e.g., no disk space or don’t care GPU), define CPUONLY in the CNTKMath project

- MPI library
  - To use model averaging based data parallelization, define MPI_SUPPORT
CNTK Architecture

- Provides flexibility and freedom to enhance and tailor for different purposes
Functionality

• Supports automatic differentiation
• Supports SGD (BP, BPTT, and truncated BPTT, Adagrad, and rmsprop)
• Supports arbitrary valid computational networks
  • Building DNN, CNN, RNN, LSTM, GMM, MDN is as simple as describing the operations of the networks.
• Supports both dense and sparse inputs
• Supports multiple inputs/outputs, and multi-objective training
• Efficient computation
  • Remove duplicated computations in both forward and backward computations
  • Use minimal memory and don’t reallocate memory if possible
  • Optimized CPU and GPU computation
  • Do batch computation whenever possible
To Run CNTK: TIMIT Example

• cntk `configFile=yourConfigFile.config` `DeviceNumber=1` `ExpDir=xyz`
• Inside the config file

```plaintext
stderr=$ExpDir$\TrainNDLNetwork\log\log
command=\TIMIT_TrainNDL
precision=float

TIMIT_TrainNDL=[
  action=train
  deviceId=$DeviceNumber$
  #Stringize Variables
  modelPath=$ExpDir$\TrainNDLNetwork\model\cntkSpeech.dnn

  traceLevel=1

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

CPU: -1 or CPU
GPU: >=0
Auto: Auto
To Run CNTK: TIMIT Example

- `cntk configfile=yourConfigFile.config`  
  `DeviceNumber=1`  
  `ExpDir=xyz`  

- Inside the config file

  ```
  stderr=$ExpDir$/TrainNDLNetwork/log/log
  command=TIMIT_TrainNDL
  precision=float

  TIMIT_TrainNDL=[
    action=train
    deviceId=$DeviceNumber$  # Stringize Variables
    modelPath=$ExpDir$/TrainNDLNetwork/model/cntkSpeech.dnn

    traceLevel=1

    SimpleNetworkBuilder=[…]
    SGD=[…]
    reader=[…]
  ]
  ```
Inside Configuration File

TIMIT_TrainNDL=[
  #parameters used by SimpleNetworkBuilder block may be defined here #
  SimpleNetworkBuilder=[
    layerSizes=792:512*3:183
    trainingCriterion=CrossEntropyWithSoftmax
    evalCriterion=ErrorPrediction
    layerTypes=Sigmoid
    applyMeanVarNorm=true
    needPrior=true
  ]
  SGD=[
    epochSize=0
    minibatchSize=256:1024
    learningRatesPerMB=0.8:3.2*14:0.08
    momentumPerMB=0.9
    dropoutRate=0.0
    maxEpochs=$MaxNumEpochs$
  ]
]
Insight Config File

reader=[
    readerType=HTKMLFReader
    readMethod=rollingWindow
    miniBatchMode=Partial
    randomize=Auto

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

    labels=[
        mlfFile=$MlfDir$\TIMIT.train.align_cistate.mlf.cntk
        labelDim=183
        labelMappingFile=$MlfDir$\TIMIT.statelist
    ]
]

features (792)
labels (183)
Top-Level Action Commands

- **Train** - train a model
- **Adapt** - adapts an already trained model using KL divergence regularization
- **Eval** - evaluate/test a model for accuracy, usually with a test dataset
- **CV** - evaluates a series of models from different epochs on a development (or cross validation) set and displays the information of the best model
- **Write** - writes the value of an output node to a file
- **Edit** - executes a model editing language (MEL) script
- **Dumpnode** - dumps the information of node(s) to an output file. MEL can do the same thing.
Existing Readers

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

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

• Major Modeling Techniques in ASR
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Long Short-Term Memory (LSTM) RNNs
  • Model Adaptation

• Build Deep ASR Models Using CNTK
  • Computational Networks and CNTK
  • Network Definition Language (NDL)
  • Examples: DNN, LSTM and PAC-RNN
  • Model Editing Language (MEL)
  • Stochastic Gradient Descent (SGD)

• Summary
Network Definition Language

• Provides a simple yet powerful way to define a network in a code-like fashion.

• **Variables**, e.g., SDim=784
  • Any alphanumeric sequence that starts with a letter
  • Can not use reserved words (e.g., function names)
  • Case-insensitive
  • Immutable – can only assign value once

• **Inputs**, e.g., features=\texttt{Input}(SDim)
  • Represent input data and labels associated with the samples
  • Values note saved in the model

• **Parameters**, e.g., B0=\texttt{Parameter}(HDim)
  • Represent model parameters
  • Values saved as part of the model
Network Definition Language

- **Functions**, e.g., \( \text{Times1} = \text{Times}(W0, \text{features}) \)
  - Describe computation steps
  - Links different nodes to form a network

- **Special nodes**
  - Specify features, labels, default output nodes, default evaluation nodes and training criteria nodes
  - Can be specified directly or through node tagging
  - Example (direct):
    - FeatureNodes = (features1, features2)
    - LabelNodes = (labels)
    - CriteriaNodes = (CE)
    - EvalNodes = (ErrPredict)
    - OutputNodes = (Plus2)
  - Example (through tagging):
    - myFeatures = Input(featDim, tag=feature)
Functions

• Input, ImageInput
• Parameter, Constant
• ReLU, Sigmoid, Tanh, Log, Exp, Cos, Dropout, Negate, Softmax, LogSoftmax
• SumElements
• RowSlice, RowStack
• Scale, Times, DiagTimes, Plus, Minus, ElementTimes
• KhatriRaoProduct,
• GMMLogLikelihood
• SquareError, CrossEntropy, CrossEntropyWithSoftmax, ClassificationError, ClassCrossEntropyWithSoftmax, Logistic
• CosDistance, CosDistanceWithNegSamples (for DSSM)
• MatrixL1Reg, MatrixL2Reg,
• Mean, InvStdDev, PerDimMVNorm
• Convolution, MaxPooling, AveragePooling
• Delay (for recursion)
Macros

• Can be defined as a one line function, e.g.,
  \[ RFF(x_1, w_1, b_1) = \text{RectifiedLinear}(\text{Plus}(\text{Times}(w_1, x_1), b_1)) \]

• Or as a block of code, e.g.,
  \[
  \begin{aligned}
  &\text{FF}(X_1, W_1, B_1) \\
  &\{ \\
  &\quad T=\text{Times}(W_1, X_1) \\
  &\quad \text{FF}=\text{Plus}(T, B_1) \\
  &\}
  \end{aligned}
  \]

• A macro may call another macro but not recursively

• Can access internal variables via the dot syntax, e.g.,
  \[ CE = \text{SMBFF}(L_1, LDim, HDim, labels) \]
  \[ Err = \text{ErrorPrediction}(labels, CE.F) \]

• Optional Parameters
  • Ordered: Based on the order of arguments
  • Named: Based on the argument name
  • \( B_0 = \text{Parameter}(HDim, \text{init}=\text{zero}) \)
  • \( W_0 = \text{Parameter}(HDim, SDim, \text{init}=\text{uniform}) \)
Example Macros

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| FF(X1, W1, B1)   | \[
|                  | \{ \text{T}=\text{Times}(W1,X1) \text{P}=\text{Plus}(T, B1) \} \]        |
| BFF(in, rows, cols) | \[
|                  | \{ \text{B}=\text{Parameter}(\text{rows}, \text{init=fixedvalue, value=0}) \text{W}=\text{Parameter}(\text{rows, cols}) \text{FF}=\text{FF}(\text{in, W, B}) \} \] |
| SBFF(in, rows, cols) | \[
|                  | \{ \text{BFF}=\text{BFF}(\text{in, rows, cols}) \text{S}=\text{Sigmoid}(\text{BFF}) \} \] |
Macros in Effect: Auto-encoder Example

• Without Macros

\[
\text{featDim}=1000
\]
\[
\text{hiddenDim}=100
\]
\[
\text{features} = \text{Input}(\text{featDim}, \text{tag}=\text{feature})
\]
\[
\text{Wh} = \text{Parameter}(\text{hiddenDim}, \text{featDim})
\]
\[
\text{Bh} = \text{Parameter}(\text{hiddenDim}, \text{init}=\text{fixedvalue}, \text{value}=0)
\]
\[
\text{Th} = \text{Times}(\text{Wh}, \text{features})
\]
\[
\text{Ph} = \text{Plus}(\text{Th}, \text{Bh})
\]
\[
\text{Sh} = \text{Sigmoid}(\text{Ph})
\]
\[
\text{Wo} = \text{Parameter}(\text{featDim}, \text{hiddenDim})
\]
\[
\text{Bo} = \text{Parameter}(\text{featDim}, \text{init}=\text{fixedvalue}, \text{value}=0)
\]
\[
\text{To} = \text{Times}(\text{Wo}, \text{Sh})
\]
\[
\text{Po} = \text{Plus}(\text{To}, \text{Bo})
\]
\[
\text{MSE} = \text{SquareError}(\text{features}, \text{Po}, \text{tag}=\text{criteria})
\]
\[
\text{EvalNodes} = (\text{MSE})
\]
Macros in Effect: Auto-encoder Example

• With Macros

\[
\text{featDim}=1000 \\
\text{hiddenDim}=100
\]

\[
\text{features} = \text{Input}(\text{featDim}, \text{tag}=\text{feature})
\]

\[
\text{L1} = \text{SBFF}(\text{features}, \text{hiddenDim}, \text{featDim})
\]

\[
\text{L2} = \text{BFF}(\text{L1}, \text{featDim}, \text{hiddenDim})
\]

\[
\text{MSE} = \text{SquareError}(\text{features}, \text{L2}, \text{tag}=\text{criteria})
\]

\[
\text{EvalNodes} = (\text{MSE})
\]
Outline

• Major Modeling Techniques in ASR
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Long Short-Term Memory (LSTM) RNNs
  • Model Adaptation

• Build Deep ASR Models Using CNTK
  • Computational Networks and CNTK
  • Network Definition Language (NDL)
  • Examples: DNN, LSTM and PAC-RNN
  • Model Editing Language (MEL)
  • Stochastic Gradient Descent (SGD)

• Summary
NDL Example: DNN

```plaintext
NDLNetworkBuilder=[
    ndlMacros=${Ndldir}\default_macros.ndl
    networkDescription=${ Ndldir}\classify.ndl
]

load=ndlMacroDefine
run=ndlCreateNetwork
ndlMacroDefine=[
    MeanVarNorm(x)
    {
        xMean = Mean(x)
        xStdDev = InvStdDev(x)
        xNorm=PerDimMeanVarNormalization(x,xMean,xStdDev)
    }
    LogPrior(labels)
    {
        Prior=Mean(labels)
        LogPrior=Log(Prior)
    }
]
```

Decide which macro set to use
Decide which block to run
# NDL Example: DNN

```python
ndlCreateNetwork=[
    featDim=792
    labelDim=183
    hiddenDim=512
    myFeatures=Input(featDim, tag=feature)
    myLabels=Input(labelDim, tag=label)

    # define network
    featNorm = MeanVarNorm(myFeatures)
    L1 = SBFF(featNorm,hiddenDim,featDim)
    L2 = SBFF(L1,hiddenDim,hiddenDim)
    L3 = SBFF(L2,hiddenDim,hiddenDim)
    CE = SMBFF(L3,labelDim,hiddenDim,myLabels,tag=Criteria)
    Err = ErrorPrediction(myLabels,CE.BFF.FF.P,tag=Eval)

    # define output (scaled loglikelihood)
    logPrior = LogPrior(myLabels)
    ScaledLogLikelihood=Minus(CE.BFF.FF.P,logPrior,tag=Output)
]
```

- `featDim`, `labelDim`, and `hiddenDim` define the dimensions of the network.
- `myFeatures` and `myLabels` are the input layers.
- The network is defined as a series of `SBFF` (Single-Batch Fully-Connected) layers, followed by a `SMBFF` (Single-Batch Multi-Layer Fully-Connected) layer.
- The error prediction is calculated with `ErrorPrediction`.
- The output is a scaled loglikelihood calculated with `Minus` and `LogPrior`.

---

**Annotations:**
- Indicate this is a feature node
- Must match that in the reader
- Training criterion
- Evaluation criterion
- Output of CN
NDL Example: LSTM

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

\[
\begin{align*}
\text{delayH} &= \text{Delay}(\text{outputDim}, \text{output}, \text{delayTime}=1) \\
\text{delayC} &= \text{Delay}(\text{outputDim}, \text{ct}, \text{delayTime}=1) \\
\text{WxiInput} &= \text{Times}(\text{Wxi}, \text{inputVal}) \\
\text{WhidelayHI} &= \text{Times}(\text{Whi}, \text{delayH}) \\
\text{Wc delayedCI} &= \text{DiagTimes}(\text{Wci}, \text{delayC}) \\
\text{it} &= \text{Sigmoid} \left( \text{Plus} \left( \text{Plus} \left( \text{Plus} \left( \text{WxiInput}, \text{bi} \right), \text{WhidelayHI} \right), \text{Wc delayedCI} \right) \right) \\
\text{WhfdelayHF} &= \text{Times}(\text{Whf}, \text{delayH}) \\
\text{WcfdelayCF} &= \text{DiagTimes}(\text{Wcf}, \text{delayC}) \\
\text{Wxfinput} &= \text{Times}(\text{Wxf}, \text{inputVal}) \\
\text{ft} &= \text{Sigmoid} \left( \text{Plus} \left( \text{Plus} \left( \text{Plus} \left( \text{Wxfinput}, \text{bf} \right), \text{WhfdelayHF} \right), \text{WcfdelayCF} \right) \right)
\end{align*}
\]

\[
\begin{align*}
\text{i}_t &= \sigma \left( \mathbf{W}^{(xi)} \mathbf{x}_t + \mathbf{W}^{(hi)} \mathbf{h}_{t-1} + \mathbf{W}^{(ci)} \mathbf{c}_{t-1} + \mathbf{b}^{(i)} \right) \\
\text{f}_t &= \sigma \left( \mathbf{W}^{(xf)} \mathbf{x}_t + \mathbf{W}^{(hf)} \mathbf{h}_{t-1} + \mathbf{W}^{(cf)} \mathbf{c}_{t-1} + \mathbf{b}^{(f)} \right)
\end{align*}
\]
NDL Example: LSTM

\[
\begin{align*}
WxcInput &= \text{Times}(Wxc, \text{inputVal}) \\
\text{WhcdelayHC} &= \text{Times}(\text{Whc}, \text{delayH}) \\
\text{bit} &= \text{ElementTimes}(\text{it}, \text{Tanh}(\text{Plus}(WxcInput, \text{Plus}(\text{WhcdelayHC}, \text{bc})))) \\
\text{bft} &= \text{ElementTimes}(\text{ft}, \text{delayC}) \\
\text{ct} &= \text{Plus}(\text{bft}, \text{bit}) \\
\text{Wxoinput} &= \text{Times}(Wxo, \text{inputVal}) \\
\text{WhodelayHO} &= \text{Times}(\text{Who}, \text{delayH}) \\
\text{Wcoct} &= \text{DiagTimes}(Wco, \text{ct}) \\
\text{ot} &= \text{Sigmoid}(\text{Plus}(\text{Plus}(\text{Plus}(\text{Wxoinput}, \text{bo}), \text{WhodelayHO}), \text{Wcoct})) \\
\text{output} &= \text{ElementTimes}(\text{ot}, \text{Tanh}(\text{ct}))
\end{align*}
\]

\[
\begin{align*}
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tanh \left( W^{(xc)} x_t + W^{(hc)} h_{t-1} + b^{(c)} \right) \\
\sigma \left( W^{(xo)} x_t + W^{(ho)} h_{t-1} + W^{(co)} c_t + b^{(o)} \right) \\
\text{h}_t &= o_t \cdot \tanh (c_t)
\end{align*}
\]
**NDL Example: Prediction Based AM**

- A recurrent system with two major components
- Predict, adapt, and correct

---

**Diagram: Multi-objective Prediction Based AM**

- **Prediction DNN**
  - Input features $o_t$
  - Sigmoid layers
  - Bottleneck layer
  - Softmax layer
  - Auxiliary info for prediction $p_{\text{pred}}^{\text{pred}}(l_{t+n} | o_t, y_t)$

- **Correction DNN**
  - Softmax layer
  - Sigmoid layers
  - Context expansion $x_t = [h_{t-T_{\text{corr}}}^{\text{pred}}, \ldots, h_{t-1}^{\text{pred}}]^T$
  - Prediction $y_t = h_t^{\text{corr}}$
  - Projection layer $p_{\text{corr}}^c(s_t | o_t, x_t)$

---

7/1/2015

Dong Yu: Deep Learning for Automatic Speech Recognition
#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)
cr1=Constant(0.8)
cr2=Constant(0.2)
NDL Example: Prediction Based AM

# 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)
NDL Example: Prediction Based AM

\[
\text{DNN}_A\_L1 = \text{SBFF\_multi8}(\text{featNorm}, \text{DNN}_A\_delayfeat1, \text{DNN}_A\_delayfeat2, \\
\text{DNN}_A\_delayfeat3, \text{DNN}_A\_delayfeat4, \text{DNN}_A\_delayfeat5, \text{DNN}_A\_delayfeat6, \\
\text{DNN}_A\_delayfeat7, \text{DNN}_A\_delayfeat8, \text{DNN}_A\_delayfeat9, \text{DNN}_A\_delayfeat10, \\
\text{hiddenDim}, \text{featDim}, \text{bottleneckDim})
\]

\[
\text{DNN}_A\_L2 = \text{SBFF}(\text{DNN}_A\_L1, \text{hiddenDim}, \text{hiddenDim})
\]

\[
\text{DNN}_A\_L2\_B = \text{SBFF}(\text{DNN}_A\_L1, \text{bottleneckDim2}, \text{hiddenDim})
\]

\[
\text{DNN}_A\_CE\_BFF = \text{BFF}(\text{DNN}_A\_L2, \text{labelDim}, \text{hiddenDim})
\]

\[
\text{DNN}_B\_L1 = \text{SBFF\_multi}(\text{featNorm}, \text{DNN}_A\_L2\_B\_BFF.FF.P, \text{hiddenDim}, \\
\text{featDim, bottleneckDim2})
\]

\[
\text{DNN}_B\_L2 = \text{SBFF}(\text{DNN}_B\_L1, \text{bottleneckDim}, \text{hiddenDim})
\]

\[
\text{DNN}_B\_CE\_BFF = \text{BFF}(\text{DNN}_B\_L2, \text{labelDim2}, \text{bottleneckDim})
\]

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)
**NDL Example: Prediction Based AM**

```plaintext
reader=[
    readerType=HTKMLFReader
    readMethod=blockRandomize
    frameMode=false
    Truncated=true
    nbruttsineachrecurrentiter=32
features=[
    dim=1845
    scpFile=$scpFilePath$
]

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

- **Utterance mode for RNN**
- **Truncated BPTT**
- **# of parallel utterances**
- **Main label**
- **Prediction label**
Outline

• Major Modeling Techniques in ASR
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Long Short-Term Memory (LSTM) RNNs
  • Model Adaptation

• Build Deep ASR Models Using CNTK
  • Computational Networks and CNTK
  • Network Definition Language (NDL)
  • Examples: DNN, LSTM and PAC-RNN
  • Model Editing Language (MEL)
  • Stochastic Gradient Descent (SGD)

• Summary
Model Editing Language

• Provides a means to modify both the structure and the model parameters of an existing network
• Can use NDL to define new elements
• Supports the use of the ‘*’ wildcard in the commands
  • If available nodes are
    • L3.RL: RectifiedLinear node
    • L3.BFF.B: Parameter node - used for bias
    • L3.BFF.W: Parameter node - used for weight
    • L3.BFF.FF.T: Times node
    • L3.BFF.FF.P: Plus node
  • Then
    • L3.*: Select all the L3 nodes
    • L3.*.P: Select the L3.BFF.FF.P node
    • L3.*: All the L3 nodes in the model
MEL Commands

- CreateModel, CreateModelWithName
- LoadModel, LoadModelWithName
- SaveDefaultModel, SaveModel
- UnloadModel
- LoadNDLSnippet
- Dump, DumpModel, DumpNode
- Copy, CopyNode, CopySubTree
- SetInput, SetNodeInput, SetInputs, SetNodeInputs
- SetProperty
- SetPropertyForSubTree
- Remove, RemoveNode, Delete, DeleteNode
- Rename
MEL Commands

• Copy, CopyNode, CopySubTree
  • Copy a node, a group of nodes, or all nodes in a subtree from one location to another
  • Format:
    • Copy(fromNode, toNode, [copy=all | value])
    • CopyNode(fromNode, toNode, [copy=all | value])
    • CopyNode(fromNode, toNode, [copy=all | value])
  • Copy=all (default. copies both values and links)
  • Copy=value (copies only the values of a node)

• SetInput, SetNodeInput, SetInputs, SetNodeInputs
  • change connections between nodes
  • Formats
    • SetInput(node, inputNumber, inputNode)
    • SetInputs(node, inputNode1[, inputNode2, inputNode3])
MEL Commands

• SetProperty, SetPropertyForSubTree
  • Set the property of a node or a subtree to a specific value
  • Format
    • SetProperty(node, propertyName, propertyValue)
    • SetProperty(rootNode, propertyName, propertyValue)
  • propertyName: ComputeGradient and tags

• Remove, RemoveNode, Delete, DeleteNode
  • Delete node(s) from a model
  • Format (all have the same effect)
    • Remove(node, [node2, node3])
    • Delete(node, [node2, node3])
    • RemoveNode(node, [node2, node3])
    • DeleteNode(node, [node2, node3])
MEL Example: DPT

CE.S=Softmax(CE.P)
CE.P=Plus(CE.T,b0)
CE.T=Times(W0,L1.S)

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

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

X
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)
```
Outline

• Major Modeling Techniques in ASR
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Long Short-Term Memory (LSTM) RNNs
  • Model Adaptation

• Build Deep ASR Models Using CNTK
  • Computational Networks and CNTK
  • Network Definition Language (NDL)
  • Examples: DNN, LSTM and PAC-RNN
  • Model Editing Language (MEL)
  • Stochastic Gradient Descent (SGD)

• Summary
Learner

• Given a training set and an optimization criterion, the learner finds a set of model parameters that optimizes the criterion.

• In CNTK the learner always minimizes the criterion.
  • If you want to maximize an objective function, you must first convert it to a minimization problem:

\[
argmax(a) = argmin(-a)
\]

• CNTK supports the stochastic gradient descent (SGD) learner and its variants AdaGrad and RmsProp

\[
W_{t+1}^\ell \leftarrow W_t^\ell - \epsilon \Delta W_t^\ell
\]

• Used to support L-BFGS as well. Removed when we make it open source to public
• Can add other learners relatively easily as long as it requires only the first-order gradient.
SGD Learner Configuration

• The behavior of the SGD algorithm is controlled by the SGD block of the options.

• These options can be classified as:
  • Training process control
  • Learning rate and momentum control
  • Gradient control
  • Others

SGD=

epochSize=0
minibatchSize=256:1024
learningRatesPerMB=0.8:3.2*14:0.08
momentumPerMB=0.9
dropoutRate=0.0
maxEpochs=$\text{MaxNumEpochs}$
Training Process Control

• **modelPath**
  • The full path used to save the final model.
  • Must be provided and points to a valid file name.

• **epochSize**
  • The number of samples in each epoch. In CNTK epoch can be different from the sweep of the full dataset.
  • When set to 0 the whole dataset size is used (handled by the data reader)
  • An intermediate model and check point info is saved for each epoch.

• **maxEpochs**
  • Maximum number of epochs to run.
  • May terminate earlier if learning rate becomes too small

• **minibatchSize**
  • Minibatch size for each epoch. Default value is 256.
  • Supports different size for different epochs. E.g., 128*2:1024 (128 for 2 epochs and then 1024 for the rest).

• **keepCheckpointFiles**
  • Whether you want to keep the check point file after a new epoch starts.
  • Default is false so that the previous check point files are deleted.
Training Process Control

• `trainCriterionNodeName`
  • The name of the training criterion node.
  • If not provided the default training criterion node is used.

• `evalCriterionNodeName`
  • The name of the evaluation criterion node.
  • If not provided the default evaluation criterion is used.

• `dropoutRate`
  • Dropout rate during the training procedure. Default is 0.0.
  • Different epoch can have different dropout rate but all dropout nodes share the same rate for the same epoch.
  • Has no effect if there is no dropout node

• `maxTempMemSizeInSamplesForCNN`
  • Maximum temporary memory used (in number of samples) when packaging and unpackaging input features for CNN.
  • Default is 0: means using any value as needed.
  • Useful to control the memory footprint esp. when run under GPU.

• `executionEngine`
  • The execution engine to use. Valid value is synchronous (default).
Learning Rate and Momentum Control

• `learningRatesPerMB`
  • Learning rates per minibatch. Useful when you want to use the same learning rate while the minibatch size is changed.
  • Different epochs can have different rates, e.g., 0.8*10:0.2

• `learningRatesPerSample`
  • Learning rates per sample. The effective learning rate equals to `minibatchSize × learningRatesPerSample`
  • Useful when you want to keep the learning rates per sample constant, i.e., automatically increases effective learning rate for the minibatch when the minibatch size is increased.
  • Different epochs can have different rates

• `momentumPerMB`
  • Momentum per minibatch. Default is 0.9.
  • Different epochs can have different momentum, e.g., 0.1*2:0.9
Learning Rate and Momentum Control

• **autoAdjust**
  • Information related to automatic learning rate control.
  • Default value is empty (""") no auto learning rate control.

• **autoAdjustLR**
  • Which learning rate adjustment algorithm to use.
  • Valid values are
    • **None**: default, don't auto adjust learning rate
    • **AdjustAfterEpoch**: check the training criterion after each epoch using the development set (if available) or the training set to decide whether to adjust the learning rate
    • **SearchBeforeEpoch**: search the learning rate based on performance on a small portion of the training set before each epoch starts
  • Automatic learning rate adjustment is applied only if the learning rate for the epoch is not provided explicitly.
### If AutoAdjustLR Set to AdjustAfterEpoch

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>autoAdjust=[</code></td>
<td>Reduce learning rate if the improvement is less than this value. Default is 0.</td>
</tr>
<tr>
<td><code>autoAdjustLR=AdjustAfterEpoch</code></td>
<td>Reduce learning rate by this factor. Default is 0.618.</td>
</tr>
<tr>
<td><code>reduceLearnRateIfImproveLessThan=0</code></td>
<td>Increase learning rate if the improvement is larger than this value. Default is 1#INF.</td>
</tr>
<tr>
<td><code>learnRateDecreaseFactor=0.618</code></td>
<td>Increase learning rate by this factor. Default is 1.382.</td>
</tr>
<tr>
<td><code>increaseLearnRateIfImproveMoreThan=1#INF</code></td>
<td>Weather to load the best model if the current model decreases the performance. Default is true.</td>
</tr>
<tr>
<td><code>learnRateIncreaseFactor=1.382</code></td>
<td>The frequency of applying the learning rate adjustment check. Default is 1 epoch.</td>
</tr>
<tr>
<td><code>loadBestModel=true</code></td>
<td></td>
</tr>
<tr>
<td><code>learnRateAdjustInterval=1</code></td>
<td></td>
</tr>
</tbody>
</table>
Gradient Control

• `gradientClippingWithTruncation`
  - True (default): use the truncation based gradient clipping to control gradient explosion.
  - False: use the norm based clipping which is more expensive

• `clippingThresholdPerSample`
  - The clipping thread for each sample. Default value is \(1\times\text{INF}\)

• `L2RegWeight`
  - The L2 regularization weight. Default is 0.
  - Adds a scaled version of the parameters into the gradient, biasing parameter values to zero

• `L1RegWeight`
  - The L1 regularization weight. Default is 0.
  - Uses the proximal gradient descent algorithm to shrink the weights, i.e., with the soft-threshold function

• `gaussianNoiseInjectStd`
  - The standard deviation of the Gaussian noise added to the gradient. Default is 0.
**Gradient Update Variations**

- **gradUpdateType**
  - Gradient update type
  - Valid values are none (default, normal SGD), Adagrad, or rmsprop.

- **Default SGD learner**
  - Apply learning rate to gradients.
  - Apply momentum to gradients.
  - Subtract result from parameter values.
Gradient Update: RMSProp

- **rms_wgt_inc**
  - multiplicative increment of the learning rate scale. Default is 1.2.

- **rms_wgt_dec**
  - multiplicative decrement of the learning rate scale. Default is 0.75.

- **rms_wgt_max**
  - maximum learning rate scale allowed.
  - A value closer to 1 makes the learning rate adjustment more stable but slower. Default is 10.

- **rms_wgt_min**
  - minimum learning rate scale allowed.
  - A value closer to 1 makes the learning rate adjustment more stable but slower. Default is 0.1.

- **rms_gamma**
  - smoothing factor used to estimate the moving average of the variance.
  - The smaller the value, the quicker it forgets the past information. Default is 0.99.
Other Info

• **traceLevel**
  • Trace level to decide what information to print out in the stderr.
  • Valid values are 0 (default) and 1 (more info).

• **numMBsToShowResult**
  • Display training statistics after how many minibatches.
  • Default is 10. Larger values update information slower but require less IO and thus are faster.

• **gradientCheck**
  • Determines whether to use the gradient checker. Default is false.
  • When using the gradient checker you need to use a minibatch size that is larger than the sequence length for RNNs due to the truncated back-propagation through time (BPTT) algorithm used to train RNNs, and a smaller learning rate to prevent numerical issues caused by divergence. In addition, precision should be set to double.
  • Should be used to check your implementation when new nodes are added.
Hyper-Parameter search

• Manual search
  • Based on experience
  • Start from hyper-parameters known to work well for similar problems
  • Analyze the result and adjust the hyper-parameters accordingly

• Automatic search
  • Often based on the Bayesian optimization algorithms
  • Basic idea: Given the set of hyper-parameter-result pairs, adjust the distribution of good parameters and randomly select from it
  • Many tools available
  • (tool) spearmint / whetlab https://github.com/HIPS/Spearmint and whetlab.com
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
Outline

• Major Modeling Techniques in ASR
  • Deep Neural Networks (DNNs)
  • Convolutional Neural Networks (CNNs)
  • Long Short-Term Memory (LSTM) RNNs
  • Model Adaptation

• Build Deep ASR Models Using CNTK
  • Computational Networks and CNTK
  • Network Definition Language (NDL)
  • Examples: DNN, LSTM and PAC-RNN
  • Model Editing Language (MEL)
  • Stochastic Gradient Descent (SGD)

• Summary
Summary

• Significant progress has been made in the recent years with the integration of deep learning models such as DNNs, CNNs and LSTMs.

• 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

• CNTK implements CNs so that you only need to focus on designing the CNs instead of implementing learning algorithms for your specific CN