Configuration Files

# Overview

The Computational Network ToolKit (CNTK) consists of a number of components to complete machine learning task. These include Network Builders, Network Trainers, Data Readers, and various other components. Most of these components need some configuration information available in order to function, and these configuration parameters are provided through configuration files in CNTK.

## Configuration Data

Configuration files are collections of name-value pairs. The configuration data can be one of the following types:

|  |  |  |
| --- | --- | --- |
| config type | Examples | Description |
| Simple | deviceId=auto | A single value is assigned to the configuration parameter |
| Array | minibatchSize=256:512:512:512:1024  minibatchSize=256:512\*3:1024  files=(;c:\data.txt;c:\labels.txt) | * a configuration parameter is assigned an array of values which need not be of a uniform type * a ‘:’ is the default separator for arrays, but may be changed by enclosing the array values in parenthesis and placing the new separator character immediately following the open parenthesis * an ‘\*’ repeat character allows a particular value to be repeated multiple times in the array |
| Parameter Set | section1=[id=1;size=256]  section2=[  subsection=[string=”hi”;num=5]  value=1e-10  array=10:”this is a test”:1.25  ] | * Parameter sets contain sets of configuration parameters of any type * parameter sets can be nested * the default separator for parameter sets is the ‘;’ if multiple items are included on one line * line separators also serve as separators for items |

## Organization

In CNTK configuration files Parameter Sets are organized in a hierarchal fashion. The actual data values are not evaluated until a CNTK components requests the value. When a value is requested, by a component, it will search that components section of the configuration file, if the value is not found, it will continue looking in the parent parameter set and continue looking in parent parameter sets until the parameter is found, or the top level of the configuration hierarchy is reached without a match.

## Default Values

Most parameters in configuration files have a default value which will be used if no configuration value is specified. If there is no default value and the value cannot be found in a search, an exception will be displayed and the program will exit.

## Repeated Values

If a parameter name is specified more than once, the last value set to that value is the one that will be maintained. The only exception to this is in parameter sets, which are surrounded by ‘[‘ square braces ‘]’, in these cases the values inside the braces are considered to be a parameter set, and will be added to the currently existing parameter set. For example:

params=[a=1;b=2;c=3]  
params=[c=5;d=6;e=7]

is effectively equal to:

params=[a=1;b=2;c=5;d=6;e=7]

this “append” processing is not used for arrays elements, and the entire array will be replaced if it is set multiple times.

## Comments

The ‘#’ character signifies the beginning of a comment, everything that occurs after the ‘#’ is ignored. The ‘#’ must be preceded by whitespace or be at the beginning of the line to be interpreted as a comment. The following are valid comments, and an example of a value that will not be interpreted as a comment:

#commands used will be appended the stderr name to create a path

stderr=c:\cntk\log\cntk # “\_mnistTrain\_mnistTest.log” would be appended

# the parameter below is set to infinity, the ‘#’ in ‘1#INF’ is not a comment marker

var = 1#INF

# Example

This section will go through a sample configuration file that creates a simple DNN (Deep Neural Network), trains the network and then tests the results. If you would rather see the Comprehensive documentation of configuration parameters skip to the Configuration Reference section.

Here is a simple example of a configuration file:

# sample configuration file for CNTK

command=mnistTrain:mnistTest

#global parameters, all commands use these values unless overridden at a higher level

precision=float

deviceId=auto

#commands used will be appended the stderr name to create a path

stderr=c:\cntk\log\cntk # “\_mnistTrain\_mnistTest.log” would be appended

traceLevel=0 # larger values mean more output

ndlMacros=C:\cntk\config\DefaultMacros.ndl

modelPath=c:\cntk\model\sample.dnn

labelMappingFile=c:\cntk\data\mnist\labels.map

mnistTrain=[

action=train

minibatchSize=32

epochSize=60000

NDLNetworkBuilder=[

networkDescription=c:\cntk\config\sample.ndl

run=ndlMacroUse

]

SGD=[

#modelPath - moved to root level to share with mnistTest

learningRatesPerMB=0.001

maxEpochs=50

]

reader=[

readerType=UCIFastReader

file=c:\cntk\data\mnist\mnist\_train.txt

features=[

dim=784

start=1

]

labels=[

dim=1

start=0

labelDim=10

]

]

]

mnistTest=[

action=eval

maxEpochs=1

epochSize=10000

minibatchSize=1000

reader=[

readerType=UCIFastReader

randomize=None

file=c:\data\mnist\mnist\_test.txt

features=[

dim=784

start=1

]

labels=[

dim=1

start=0

labelDim=10

]

]

]

## Commands and actions

The first thing at the top of the configuration is the command:

command=mnistTrain:mnistTest

This command instructs CNTK to execute the **mnistTrain** section of the config file, followed by mnistTest. Each of these Config sections has an action associated with it:

mnistTrain=[

action=train

…

The **mnistTrain** section will execute the **train** action, and the **mnistTest** section will execute **eval**. The names of the sections is arbitrary, but the configuration parameter names must be command and action.

## Precision

The accuracy and speed of obtaining results in any machine learning experiment will be affected by the precision of the floating point values that are used in creating the model. The precision is specified as follows:

precision=float

The other option is double and will be more precise, but slower depending on your hardware. Most GPU hardware is much faster using float precision, but some experiments require the added precision of double.

## Device Identifier

CNTK supports CPU and GPU computation, and the determination of which device should be used is based on the **deviceId** parameter. The default value and the value used in the example config is:

deviceId=auto

This setting will pick the best device available on the machine. If multiple GPUs are available, it will choose the fastest, least busy GPU, and a CPU if no usable GPUs can be found. The following settings can be used for **deviceId**:

|  |  |
| --- | --- |
| Value | Description |
| auto | choose the best GPU, or CPU if no usable GPU is available. |
| cpu | use the CPU for all computation |
| 0 | The device Identifier (as used in CUDA) for the GPU device you wish to use (1, 2, etc.) |
| all | Use all the available GPU devices (will use PTask engine if more than one GPU is present) |
| \*2 | Will use the 2 best GPUs, any number up to the number of available GPUs on the machine can be used. (will use PTask engine) |

## Log Files

Log files are redirection of the normal standard error output. All log information is sent to standard error, and will appear on the console screen unless the stderr parameter is defined, or some other form of user redirection is active. The stderr parameter defines the directory and the prefix for the log file. The suffix is defined by what commands are being run. As an example if “abc” is the setting “abc\_mnistTrain.log” would be the log file name. It is important to note that this file is overwritten on subsequent executions if the stderr parameter and the command being run are identical.

#commands used will be appended the stderr name to create a path

stderr=c:\cntk\log\cntk # “\_mnistTrain\_mnistTest.log” would be appended

traceLevel=0 # larger values mean more output

The **traceLevel** parameter is uniformly used by the code in CNTK to specify how much extra output (verbosity) is desired. The default value is 0 (zero) and specifies minimal output, the higher the number the more output can be expected. Currently 0-limited output, 1-medium ouput, 2-verbose output are the only values supported.

## Top Level Parameters

It is often advantageous to set some values at the top level of the config file. This is because config searches start with the target section and continue the search to higher level sections. If the same parameter is used in multiple sections putting the parameter at a higher level where both sections can share it can be a good idea. In our example the following parameters are used by both the train and the test step:

ndlMacros=C:\cntk\config\DefaultMacros.ndl

modelPath=c:\cntk\model\sample.dnn

labelMappingFile=c:\cntk\data\mnist\labels.map

It can also be advantageous to specify parameters that often change all in one area, rather than separated into the sections to which the parameters belong. These commonly modified parameters can even be placed in a separate file if desired. See the layered config files in the reference section for more information.

## Command Section

A command section is a top level section of the configuration file that contains an action parameter. The command parameter at the root level determines which command sections will be executed, and in what order. The **mnistTrain** command section uses the **train** **action** to train a Deep Neural Network (DNN) using Stochastic Gradient Descent (SGD). It uses the Network Description Language (NDL) to define the network, and the UCIFastReader component to obtain its data. All the necessary information to complete this training is included in the command section. There are many other parameters that could be provided, but most have reasonable default values that have been found to provide good results for many datasets.

Under the command section there may be one or more sub-sections. These sub-sections vary by the **action** specified in the command section. In our example we used the **train action**, which includes the following subsections:

|  |  |  |
| --- | --- | --- |
| sub-section | Options | Description |
| Network Builder | SimpleNetworkBuilder | Creates simple layer-based networks with various options |
| NDLNetworkBuilder | Create a network defined in NDL (Network Description Language). This allows arbitrary networks to be created and offers more control over the network structure. |
| Trainer | SGD | Stochastic Gradient Descent trainer, currently this is the only option provided with CNTK |
| Reader | UCIFastReader | Reads the text-based UCI format, which contains labels and features combined in one file |
| HTKMLFReader | Reads the HTK/MLF format files, often used in speech recognition applications |
| BinaryReader | Reads files in a CNTK Binary format. It is also used by UCIFastReader to cache the dataset in a binary format for faster processing. |
| SequenceReader | Reads text-based files that contain word sequences, for predicting word sequences. |
| LUSequenceReader | Reads text-based files that contain word sequences, used for language understanding. |

For the Network Builder and the Trainer the existence of the sub-section name tells the train action which component to use. For example, **NDLNetworkBuilder** is specified in our example, so CNTK will use the NDL Network Builder to define the network. Similarly **SGD** is specified, so that trainer will be used. The reader sub-section is a little different, and is always called **reader**, the **readerType** parameter in the sub-section defines which reader will actually be used. Readers are implemented as separate DLLs, and the name of the reader is also the name of the DLL file that will be loaded.

mnistTrain=[

action=train

minibatchSize=32

epochSize=60000

NDLNetworkBuilder=[

networkDescription=c:\cntk\config\sample.ndl

run=ndlMacroUse

]

SGD=[

#modelPath - moved to root level to share with mnistTest

learningRatesPerMB=0.001

maxEpochs=50

]

reader=[

readerType=UCIFastReader

file=c:\cntk\data\mnist\mnist\_train.txt

features=[

dim=784

start=1

]

labels=[

dim=1

start=0

labelDim=10

]

]

]

The rest of the parameters in the mnistTrain Command Section are briefly explained here, more details about the parameters available for each component are available in the Configuration Reference section of this document.

## SGD Parameters

The parameters at the top of the command section are actually SGD parameters. These parameters have moved up a level in the configuration mainly to provide easier visibility to the parameters. Configuration searches continue searching parents until the root of the configuration is reached or the parameter is found.

minibatchSize=32

epochSize=60000

**minibatchSize** is the number of records that will be taken from the dataset and processed at once. There is often a tradeoff in training accuracy and the size of the minibatch. Particularly for GPUs, a larger minibatch is usually better, since the GPUs are most efficient when doing operations on large chunks of data. For large dataset values that are powers of 2 are most often specified, 512 and 1024 are often good choices to start with. Since the MNIST dataset is so small, we have chosen a smaller minibatch size for the example.

**epochSize** is the number of dataset records that will be processed in a training pass. It is most often set to be the same as the dataset size, but can be smaller or larger that the dataset. It defaults to the size of the dataset if not present in the configuration file. It can also be set to zero for SGD, which has the same meaning.

SGD=[

#modelPath - moved to root level to share with mnistTest

learningRatesPerMB=0.001

maxEpochs=50

]

**modelPath** is the path to the model file, and will be the name used when a model is completely trained. For epochs prior to the final model a number will be appended to the end signifying the epoch that was saved (i.e. myModel.dnn.5). These intermediate files are important to allow the training process to restart after an interruption. Training will automatically resume at the first non-existent epoch when training is restarted.

**learningRatesPerMB** is the learning rate per minibatch used by SGD to update parameter matrices. This parameter is actually an array, and can be used as follows: 0.001\*10:0.0005 to specify that for the first 10 epochs 0.001 should be used and then 0.0005 should be used for the remainder of the epochs.

**maxEpochs** is the total number of epochs that will be run before the model is considered complete.

## NDLNetworkBuilder Parameters

The NDL (Network Description Language) Network Builder component will be used by this config because the NDLNetworkBuilder section is present. Had there been a SimpleNetworkBuilder section instead, that network builder would be used.

**networkDescription** is the file path of the NDL script to execute. If there is no networkDescription file specified then the NDL is assumed to be in the same configuration file as the NDLNetworkBuilder subsection, specified with the “run” parameter. Note that only one file path may be specified via the “networkDescription” parameter; to load multiple files of macros, use the “ndlMacros” parameter.

**run** is the section containing the NDL that will be executed. If using an external file via the “networkDescription” parameter, as in the example, the **run** parameter identifies the section in that file that will be executed as an NDL script. This parameter overrides any **run** parameters that may already exist in the file. If no **networkDescription** file is specified, **run** identifies a section in the current configuration file. It must exist where a regular configuration search will find it (peer or closer to the root of the hierarchy)

**load** specifies what sections of NDL to load. Multiple sections can be specified via a “:” separated list. The sections specified by **load** are generally used to define macros, for use by the **run** section. If using an external file via the **networkDescription** parameter, as in the example, the **load** parameter identifies the section(s) in that file to load. This parameter overrides any **load** parameters that may already exist in the file. If no **networkDescription** file is specified, **load** identifies a section in the current configuration file. It must exist where a regular configuration search will find it (peer or closer to the root of the hierarchy)

**ndlMacros** is the file path where NDL macros may be loaded. This parameter is usually used to load a default set of NDL macros that can be used by all NDL scripts. Multiple NDL files, each specifying different sets of macros, can be loaded by specifying a “+” separated list of file paths for this “ndlMacros” parameters. In order to share this parameter with other Command Sections which also expect an “ndlMacros” parameter (eg, for MEL scripts), one should define it at the root level of the configuration file.

**randomSeedOffset** is a parameter which allows you to run an experiment with a different random initializations to the learnable parameters which are meant to be initialized randomly (either uniform or Gaussian). Whatever non-negative number is specified via “randomSeedOffset” will be added to the seed which would have otherwise been used. The default value is 0.

## Reader Parameters

The reader section is used for all types of readers and the **readerType** parameter identifies which reader will be used. For our example, we are using the UCIFastReader, which reads text-based UCI format data. The format of UCI data is a line of space-delimited floating point feature and label values for each data record. The label information is either at the beginning or the end of each line, if label information is included in the dataset.

readerType=UCIFastReader

Each of the readers uses the same interface into CNTK, and each reader is implemented in a separate DLL. There are many parameters in the reader section that are used by all the different types of readers, and some are specific to a particular reader. Our example reader section is as follows:

reader=[

readerType=UCIFastReader

file=c:\cntk\data\mnist\mnist\_train.txt

features=[

dim=784

start=1

]

labels=[

dim=1

start=0

labelDim=10

]

]

The two sub-sections in the reader section identify two different data sets. In our example they are named **features** and **labels**, though any names could be used. These names need to match the names used in the NDL network definition Inputs in our example, so the correct definition is used for each input dataset. Each of these sections for the UCIFastReader have the following parameters:

**dim** is the number of columns of data that are contained in this dataset

**start** is the column (zero-based) where the data columns start for this dataset

**file** is the file that contains the dataset. This parameter has been moved up from the features and labels subsections, because UCIFastReader requires the file be the same, and moving up a level is an excellent way to make sure this restriction is met.

**labelDim** is the number of possible label values that are possible for the dataset, and belongs in any label section. In MNIST this value is 10, because MNIST is a number recognition application, and there are only 10 possible digits that can be recognized

**labelMappingFile** is the path to a file that lists all the possible label values, one per line, which might be text or numeric. The line number is the identifier that will be used by CNTK to identify that label. In our example this file has been moved to the root level to share with other Command Sections. In this case, it’s important that the Evaluation Command Section share the same label mapping as the trainer, otherwise, the evaluation results will not be accurate.

The final Command section in the example is the **mnistTest** section. This section takes the trained model and tests it against a separate test dataset. All the parameters that appear in this section also appeared in the **mnistTrain** section.

## Command Line syntax

The config file can be specified on the command line when launching the Computational Network Process (cn.exe):

cn.exe configFile=config.txt

This will load the requested configuration file, and execute any command section listed in the **command** parameters in the configuration file. In our example it will execute **mnistTrain** followed by **mnistTest**.

## Configuration override

It is common to have a configuration that can be used as a base configuration, and modify only a few parameters for each experimental run. This can be done in a few different ways, one of which is to override settings on the command line. For example if I wanted to override the model file path, I could simply modify my command line:

cn.exe configFile=config.txt stderr=c:\temp\newpath

this will override the current setting for stderr, which is defined at the root level of the configuration file, with the new value. If a parameter inside a command section needs to be modified, the section also needs to be specified. For example, if I wanted to change the minibatchSize for an experiment from the command line:

cn.exe configFile=config.txt mnistTrain=[minibatchSize=256]

or to modify the data file used for an experiment:

cn.exe configFile=config.txt mnistTrain=[reader=[file=mynewfile.txt]]

Another way to do this is with layered configuration files:

## Layered Configuration Files

Instead of overriding some parts of a configuration file using command line parameters, one can also specify multiple configuration files, where the latter files override the earlier ones. This allows a user to have a “master” configuration file, and then specify, in a separate configuration file, which parameters of the master they would like to override for a given run of CNTK. This can be accomplished by either specifying a ‘+’ separated list of configuration files, or by using the “configFile=” tag multiple times. The following are equivalent:

cn.exe configFile=config1.txt+config2.txt  
cn.exe configFile=config1.txt configFile=config2.txt

If config2.txt contains the string “mnistTrain=[reader=[file=mynewfile.txt]]”, then both of these commands would be equivalent to:

cn.exe configFile=config1.txt mnistTrain=[reader=[file=mynewfile.txt]]

Note that the value of a variable is always determined by the last time it is assigned. It is also possible to mix command-line parameters, and layered configuration files, in arbitrary combinations. For example:

cn.exe configFile=config1.txt+config2.txt var1=value configFile=config3.txt

This would process these configuration parameters in the order they appear on the command line.

## Including Configuration Files

In addition being able to specify multiple configuration files at the command line, a user can “include” one configuration file within another. For example, if the first line of config2.txt was “include=config1.txt”, then simply running “cn.exe configFile=config2.txt” would be equivalent to running “cn.exe configFile=config1.txt+config2.txt” (where in this latter case, config2.txt doesn’t contain the “include” statement). Note that these include statements can appear anywhere inside a configuration file; wherever the include statement appears, that is where the specified configuration file will be “included”. Including a configuration file is equivalent to pasting the contents of that file at the location of the include statement. Include statements are resolved recursively (using a depth-first search), meaning that if configFileA.txt includes configFileB.txt, and configFileB.txt includes configFileC.txt, then the full chain will be resolved, and configFileC.txt will effectively be included in configFileA.txt. If a configuration file is included multiple times (eg, ‘A’ includes ‘B’ and ‘C’, and ‘B’ also includes ‘C’), then it will effectively only be included the first time it is encountered.

## Stringize variables

While layered configuration files allow users to reuse configuration files across experiments, this can still be a cumbersome process. For each experiment, a user might have to override several parameters, some of which might be long file paths (eg, ‘stderr’, ‘modelPath’, ‘file’, etc). The “stringize” functionality can make this process much easier. It allows a user to specify configuration like the following:

command=SpeechTrain  
stderr=$Root$\$RunName$.log  
speechTrain=[  
 modelPath=$Root$\$RunName$.model  
 SGD=[  
 reader=[  
 features=[

type=Real

dim=$DataSet1\_Dim$

file=$DataSet1\_Features$  
]]]]

Here, “Root”,“RunName”, “DataSet1\_Dim”, and “DataSet1\_Features” are variables specified elsewhere in the configuration (at a scope visible from the point at which they are used). When interpreting this configuration file, the parser would replace every string of the form “$VarName$” with the string “VarValue”, where “VarValue” represents the value of the variable called “VarName”. The variable resolution process is recursive; for example, if A=$B$, B=$C$, and C=HelloWorld.txt, then A would be resolved as “HelloWorld.txt”.  
  
Notice that because it is equivalent for a user to specify the value of a variable in a configuration file vs. at the command line, the values for these variables can be specified in either location. Recall that the value of a variable is determined by the last time it is assigned, whether that happens to be in a configuration file, or on the command line. Thus, if “Root” is defined in config1.txt, but overridden at the command-line, then the value specified at the command-line would be the one used to resolve instances of $Root$ in configFile1.txt. One useful feature is that if ‘stderr’ or ‘modelPath’ point to directories which do not exist, these directories will be created by CNTK; this allows you to specify something like: “stderr=$Root$\$RunName$\$RunName$.log”, even if the directory “$Root$\$RunName$” doesn’t exist.

# User Reference

This section is intended as a reference to all the possible configuration settings used in CNTK. See the example section for usage by example.

## Parameter Search

Config files are hierarchal organizations of Configuration Parameter Sets, a collection of name-value pairs. Any value that is expected to occur in a parameter set can alternately be defined at a higher level. The search for a parameter name will continue through parent parameter sets until it is resolved, or not found.

Default values are often assigned to parameters, these values will be used should the search for the parameter fail. If no default is available, the parameter is a required parameter, and an error will occur if it is not provided.

If a parameter occurs more than once in a given parameter set, the last occurrence of that value will have precedence.

## Commands and actions

There must be a top-level command parameter, which defines the commands that will be executed in the configuration file. Each command references a Command section of the file, which must contain an action parameter defining the operation that section will perform:

command=mnistTrain:mnistTest

mnistTrain=[

action=train

…

]

mnistTest=[

action=eval

…

]

This snippet will execute the **mnistTrain** section which executes the **train** action, followed by the **mnistTest** section.

## Command sections

The following actions are currently supported in the CNTK. The command sections that contain these action properties also require other configuration settings. The names contained in square braces (i.e. [reader]) are configuration sections, and values in curly braces (i.e. {true}) are default values used when the parameter is not specified:

* **train** – Train a model
  + [Reader] – reader configuration section to read the dataset
  + [Trainer] – trainer configuration section, currently SGD is the only trainer supported
  + [Network Builder] – network builder configuration section, the method of creating the network
  + [cvReader] – (optional) reader configuration section for cross-validation data
  + makeMode-[{true},false] – start from scratch even if an interrupted training session exists (default true)
* **test, eval** – Evaluate/Test a model for accuracy, usually with a test dataset
  + [Reader] – reader configuration section to read the test dataset
* **createLabelMap** – creates a label mapping file from the dataset for readers that support it. Currently UCIFastReader is the only reader that supports this action.
  + section – the section name (usually a *train* section) which has the reader sub-section that will be used to generate the label mapping file. The labelMappingFile property in this reader section will be written to with the results of the map file generation.
  + minibatchSize – the minibatch size to use when creating the label mapping file
* **edit** – execute an Model Editing Language (MEL) script.
  + editPath – the path to the Model Editing Language (MEL) script to be executed
  + ndlMacros - the path to the Network Definition Language (NDL) macros file that will be loaded and usable in the MEL script.
* **testUnroll** – Evaluate/Test a model for accuracy, by unrolling it
  + [reader] - reader configuration section to read the test dataset
  + minibatchSize – the minibatch size to use when reading and processing the dataset
  + epochSize – {0} size of epoch, if not specified or set to zero entire dataset will be read once.
  + modelPath – path to the model file to evaluate
  + path2EvalResults – optional, if provided evaluation results will be dumped to this file
* **adapt** – adapt an already Trained model, supports KL divergence regularization
  + [Reader] – reader configuration section to read the dataset
  + [Trainer] – trainer configuration section, currently SGD is the only trainer supported
  + [cvReader] – (optional) reader configuration section for cross-validation data
  + makeMode-[{true},false] – start from scratch even if an interrupted training session exists (default true)
  + originalModelFileName – file name for the model that will be adapted
  + refNodeName – name of the node in the computational network which will be used for KL divergence regularization (see SGD section for additional parameters required)
* **cv** – Use Cross Validation to evaluate a series of epoch model for the best results
  + [reader] - reader configuration section to read the test dataset
  + minibatchSize – the minibatch size to use when reading and processing the dataset
  + epochSize – {0} size of epoch, if not specified or set to zero entire dataset will be read once.
  + modelPath – path to the model file to evaluate, epoch files have the epoch number appended to the end of this path name
  + crossValidationInterval – array of 3 integers identifying the starting epoch, epoch increment and final epoch to evaluate.
  + sleepTimeBetweenRuns – how many seconds to wait between runs
  + numMBsToShowResult – after how many minibatches should intermediate results be shown?
  + evalNodeNames – an array of one or more node names to evaluate
* **write** – Write the output of a network to a file
  + [reader] - reader configuration section to read the dataset
  + [writer] – writer configuration section to the data writer for output data. If this value is not specified the outputPath parameter will be used.
  + minibatchSize – the minibatch size to use when reading and processing the dataset
  + epochSize – {0} size of epoch, if not specified or set to zero entire dataset will be read once.
  + modelPath – path to the model file we are using to process the input data
  + outputPath – output path to write file in a text based format. Either the writer config, or the outputPath will be used.
  + outputNodeNames – an array of one or more output node names to be written to a file
* **dumpnode** – Dump the node(s) to an output file. Note: this can also be accomplished in MEL with greater control.
  + modelPath – path to the model file containing the nodes to dump
  + nodeName – the name of the node to be written to a file, if not specified all nodes will be dumped
  + outputFile – path to the output file, will be generated in the same file as the modelPath if not specified.
  + printValues – [{true}, false] prints the values associated with a node if applicable.

The following table identifies the options for sub-section types associated with each of the action types:

|  |  |  |
| --- | --- | --- |
| sub-section | Options | Description |
| Network Builder | SimpleNetworkBuilder | Creates simple layer-based networks with various options |
| NDLNetworkBuilder | Create a network defined in NDL (Network Description Language). This allows arbitrary networks to be created and offers more control over the network structure. |
| Trainer | SGD | Stochastic Gradient Descent trainer, currently this is the only option provided with CNTK |
| Reader | UCIFastReader | Reads the text-based UCI format, which contains labels and features combined in one file |
| HTKMLFReader | Reads the HTK/MLF format files, often used in speech recognition applications |
| BinaryReader | Reads files in a CNTK Binary format. It is also used by UCIFastReader to cache the dataset in a binary format for faster processing. |
| SequenceReader | Reads text-based files that contain word sequences, for predicting word sequences. |
| LUSequenceReader | Reads text-based files that contain word sequences, used for language understanding. |

## Top Level Parameters

* **command** – an array of Command sections (which contain action parameters) that should be executed
* **precision** – [float, double], required parameter. Float values (32-bit floating point) is usually faster on most hardware, but less precise. This applies to all floating point values in CNTK.

|  |  |
| --- | --- |
| Value | Description |
| auto | choose the best GPU, or CPU if no usable GPU is available. |
| cpu | use the CPU for all computation |
| 0 | The device Identifier (as used in CUDA) for the GPU device you wish to use (1, 2, etc.) |
| all | Use all the available GPU devices (will use PTask engine if more than one GPU is present) |
| \*2 | Will use the 2 best GPUs, any number up to the number of available GPUs on the machine can be used. (will use PTask engine) |

* **deviceId** – [{auto}, cpu, #, all, \*#], default is auto. Which hardware device should be used for an action. This is used at lower levels but is often defined at the top level.
* **stderr** – optional path to where the log files will be stored. This is a redirection of stderr to a file, if not specified the output will be output to the normal stderr device (usually the console). The path here defines the directory and the prefix for the log file. The suffix is defined by what commands are being run and will be appended to create the file name. For example if “stderr=c:\abc” and “command=mnistTrain” the log file would be named “c:\abc\_mnistTrain.log”.
* **traceLevel** – [{0}] the level of output to stderr that is desired. The higher the number the more output can be expected. Currently 0-limited output, 1-medium output, 2-verbose output are the only values supported.

## Network Builders

Network builders provide a way to create a network. There are two network builders currently supported, SimpleNetworkBuilder and NDLNetworkBuilder. The sub-sections with one of these names define which network builder will be used for the train action.

### SimpleNetworkBuilder

### NDLNetworkBuilder

The NDL (Network Description Language) Network Builder component takes a config section written in NDL and interprets it to create a model. For more information on NDL please refer to the Network Description Language section of the document.

* **networkDescription –** (optional) file path of the NDL script to execute, the run parameter in this subsection can be used to override the run parameter in the file if desired. If there is no networkDescription file specified then the NDL is assumed to be in the same configuration file as the NDLNetworkBuilder subsection.
* **run** – (optional) the section containing the NDL that will be executed. If using an external file the run parameter may already exist in that file and identifies the sections in that file that will be executed as NDL scripts. This parameter in NDLNetworkBuilder section will override those settings. If no networkDescription file is specified, a section with the given name will be searched for in the current configuration file. It must exist where a regular configuration search will find it (peer or closer to the root of the hierarchy)
* **load** – (optional) the section(s) in the same file that contain NDL macros to be loaded. If specified t must exist where a regular configuration search will find it (peer or closer to the root of the hierarchy)
* **ndlMacros** – (optional) path to an NDL macros file, normally defined at the root level so it could be shared with other Command Sections. This parameter is usually used to load a default set of NDL macros that can be used by all NDL scripts.

## Trainers

### SGD

## Readers

The readers all share the same section name, which is **reader**. The **readerType** parameter identifies which reader will be used.

readerType=UCIFastReader

Each of the readers uses the same interface into CNTK, and each reader is implemented in a separate DLL. CNTK takes the **readerType** parameter value and appends “.dll” and dynamically loads that DLL to read data. This interface to data readers allows for new data formats to be supported in CNTK simply by writing a new reader component. For more information on the reader interface, and how to write a reader, refer to the Programmer documentation section.

There are many parameters in the reader section that are used by all the different types of readers, and others are specific to a particular reader. There are sub-sections under the reader section which are used to define the data records to be read. For UCIFastReader these look like:

reader=[

readerType=UCIFastReader

file=c:\cntk\data\mnist\mnist\_train.txt

features=[

dim=784

start=1

]

labels=[

dim=1

start=0

labelDim=10

]

]

The sub-sections in the reader section identify the different data records. In our example they are named **features** and **labels**, though any names could be used. If NDL was used to create the network, these section names should match the names of the input nodes in the NDL network definition. These names will be used as the matrix names passed back from the reader and name matching ensures the correct records are assigned to the correct inputs.

### UCIFastReader

UCIFastReader reads text-based UCI format data. The format of UCI data is a line of space-delimited floating point feature and label values for each data record. The label information is either at the beginning or the end of each line, if label information is included in the dataset.

The following parameters can be used to customize the behavior of the reader:

* **randomize** – [{Auto}, None, #] the randomization range (number of records to randomize across) for randomizing the input. This needs to be an integral factor of the epochSize and an integral multiple of minibatch size. Setting it to Auto will let CNTK find something that works.
* **minibatchMode** – [{Partial},Full] the mode for minibatchs when the end of the epoch is reached. In partial minibatch mode, if the remaining records are less than a full minibatch, only those read will be returned (a partial minibatch). I Full minibatch mode, no partial minibatches will be returned, instead those records will be skipped.

Each of the data record sub-sections have the following parameters:

* **dim** - the number of columns of data that are contained in this data record
* **start** -the column (zero-based) where the data columns start for this data record
* **file** - the file that contains the dataset. This parameter may be moved up to the reader section level to ensure the file is the same for all the sections, as UCIFastReader requires.

In addition if the data record sub-section is defining labels the following parameters need to be defined:

* **labelDim** – the number of possible label values that are possible for the dataset. For example, for the MNIST dataset this value is 10, because MNIST is a number recognition application, and there are only 10 possible digits that can be recognized
* **labelMappingFile** – the path to a file that lists all the possible label values, one per line, which might be text or numeric. The line number is the identifier that will be used by CNTK to identify that label. This parameter is often moved to the root level to share with other Command Sections. For example, it’s important that the Evaluation Command Section share the same label mapping as the trainer, otherwise, the evaluation results will not be accurate.
* **labelType** – [{Category},Regression,None] the type of label

The UCIFastReader is a text-based reader, and though it is optimized for speed, is still significantly slower than reading binary files. To improve training speed that may be limited by the data reader the content of the dataset can be cached the first time through the dataset, and subsequently read from a binary cache saved to disk. UCIFastReader uses BinaryReader and BinaryWriter to make this cache and to read from it. BinaryReader and BinaryWriter support multiple datasets in a single file, so data read from multiple files can all be cached in a single file. Please refer to BinaryWriter to see a sample of how to setup a UCIFastReader with caching.

### HTKMLFReader

HTKMLFReader reads files in the HTK/MLF format, which is used for speech datasets. The reader has two different modes, one for training and evaluating datasets and another for processing a dataset and producing another dataset.

For training and evaluation the following need to be defined:

* **randomize** – [{auto},None,#] randomization range used for randomizing data. Auto automatically picks a randomization range, None does no randomization, and a specific number sets the range to that number.
* **readMethod** – [{blockRandomize},rollingWindow] the method of randomization that will occur. Block randomize randomizes in a block format and does not require extra disk storage. RollingWindow creates a temporary file and randomizes data anywhere within a rolling window around the current file location.
* **framemode** – [{true}, false] is the reader reading frames, or utterances
* **minibatchMode** – [{Partial},Full] the mode for minibatchs when the end of the epoch is reached. In partial minibatch mode, if the remaining records are less than a full minibatch, only those read will be returned (a partial minibatch). I Full minibatch mode, no partial minibatches will be returned, instead those records will be skipped.
* **readAhead** – [true,{false}] have the reader read ahead in another thread. NOTE: some known issues with this feature
* **verbosity** – [0-9] default is ‘2’. The amount of information that will be displayed while the reader is running.
* **addEnergy** – {0} the number of energy elements that will be added to each frame (initialized to zero). This only functions if readMethod=rollingWindow.
* **unigram** – (optional) path to unigram file
* **[input**] – subsection that holds all the input subsections  
  subsections with arbitrary names occur under the input subsection, which will contain:
  + **dim** – dimension of the input data
  + **type** – [{real},Category] type of input
  + **file** – input file path
* **[output]** – subsection that holds all the output subsections  
  subsections with arbitrary names occur under the output subsection, which will contain:
  + **dim** – dimension of the input data
  + **type** – [{real},Category] type of input
  + **file** – input file path
  + **labelMappingFile** – (optional) state list to the labelMappingFile
  + **labelToTargetMappingFile** – (optional) filename for the labelToTargetMappingFile
  + **targetDim** – dimension of the target if labelToTargetMapping is desired.

The following two sections can currently be used instead of input and output subsections if there is only one input and one output. However, the previous syntax is recommended

* **[features]** – subsection defining the features data, must use this name
  + **dim** – dimension of the features data
  + **file** – path to the “.scp” script file that lists the locations of feature data
* **[labels]** – subsection defining labels data, must use this name
  + **labelDim** – dimension of the labels data
  + **file** – path to the MLF file describing the labels
  + **labelMappingFile** – (optional) state list to the labelMappingFile
  + **labelToTargetMappingFile** – (optional) filename for the labelToTargetMappingFile
  + **targetDim** – dimension of the target if labelToTargetMapping is desired.

For dataset processing the following parameters are used:

* **[input**] – subsection that holds all the input subsections  
  subsections with arbitrary names occur under the input subsection, which will contain:
  + **dim** – dimension of the input data
  + **file** – input file path
* **[write]** – subsection that holds all the output subsections  
  subsections with arbitrary names occur under the output subsection, which will contain:
  + **dim** – dimension of the input data
  + **path** – output file path
  + **ext** – {mfc} file extention to use for output files
  + **type** – (optional) if type=scaledLogLikelihood output will be scaled by Prior
  + **nodeName** – node name for output to be captured
  + **scpFile** – (optional) file name for SCP file if desired

### SequenceReader

SequenceReader is a reader that reads text string. It is mostly often used for language modeling tasks. An example of the text string is as follows:

</s> pierre <unk> N years old will join the board as a nonexecutive director nov. N </s>

</s> mr. <unk> is chairman of <unk> n.v. the dutch publishing group </s>

Symbol </s> is used to denote both beginning and ending of a sentence. However, this symbol can be specified by beginSequence and endSequence.

The parameters used for the SequenceReader are shared with other reader types. However, it has some unique parameters as follows:

* randomize – [None, Auto] the mode for whether doing sentence randomization of the whole corpus.
* Nbruttsineachrecurrentiter – this set the maximum number of allowed sentences in each minibatch.
* Wordclass – word class information. This is used for class-based language modeling.
* File – the corpus file.

A subsection is for input label information.

* lableIn – the section for input label. It contains the following setups
  + labelDim – the input vocabulary size
  + beginSequence – the sentence beginning symbol
  + endSequence – the sentence ending symbol
* labels – the section for output label. In the language modeling case, it is the same as labelIn.

### LUSequenceReader

LUSequenceReader is similar to SequenceReader. It however is used for language understanding tasks which have input and output strings that are different. The content of an example file is listed below

BOS O

i O

want O

to O

fly O

from O

boston B-fromloc.city\_name

at O

1110 B-arrive\_time.time

in O

the O

morning B-arrive\_time.period\_of\_day

EOS O

consists of some unique setups as follows:

The LUSequenceReader assumes that the last column is the label and all other columns are inputs. The beginning and ending of a sentence are specified using beginning and ending symbols. In the above example, they are BOS and EOS, respectively, for the beginning and ending symbols.

The LUSequenceReader has some unique setups as follows:

* wordContext – this specifies a context window. For example, wordContext=0:1:2 specifies a context window of 3. In this context window, it reads input at a current time, the next time and the time after the next time. Another example would be wordContext=0:-1. In this example, LUSequencReader reads a context window of 2 that consist of the current input and the immediate last input.
* Nbruttsineachrecurrentiter – this specifies the maximum number of sentences in a minibatch.
* Unk – this specifies the symbol to represent unseen input symbols. Usually, this symbol is “unk”.
* Wordmap – this specifies a file that maps inputs to other inputs. This is useful if the user wants to map some inputs to unknown symbols. For example

buy buy

trans <unk>

* File – the corpus file

A subsection is for input label information.

* lableIn – the section for input label. It contains the following setups
  + usewordmap – [Ture, False] specifies if using word map to map input words to other input words.
  + beginSequence – the sentence beginning symbol
  + endSequence – the sentence ending symbol
  + token – token file contains a list of input words. Their orders are not important.
* labels – the section for output label. In the language modeling case, it is the same as labelIn.
  + Token – token file contains a list of output labels. Their order is not important as long as the tokens are unique.

### BinaryReader

BinaryReader is a reader that uses a hierarchal file format to store potentially multiple datasets in an efficient way. It uses memory mapped files with a moving window of viewable data to support files larger than can be held in memory at once. More details about the file format are in the BinaryWriter Section.

The parameters used for the binary reader are quite simple as most of the required information concerning the file is contained in the file headers. The binary reader will also be called when a configuration is setup to cache UCIFastReader if the binary cached file exists.

The following parameters can be used to customize the behavior of the reader:

* **minibatchMode** – [{Partial},Full] the mode for minibatchs when the end of the epoch is reached. In partial minibatch mode, if the remaining records are less than a full minibatch, only those read will be returned (a partial minibatch). I Full minibatch mode, no partial minibatches will be returned, instead those records will be skipped.
* **file** – array of files paths to load. Each file may contain one or more datasets. The dataset names used when the file was created will be used when the file is read.

## Writers

Writers are used in a similar way to Readers in CNTK. Writers are often implemented in the same DLL as the Reader for the same format. Just as in the reader case, the writer is dynamically load based on the name in the writerType parameter:

writerType=BinaryReader # NOTE: BinaryReader.dll also implements BinaryWriter

### BinaryWriter

BinaryWriter is an implementation of a hierarchal file format the mirrors the configuration file. I uses memory mapped files to enable large files that do not fit in memory to be written and read using a moving window into the file. The binary writer is also used as a Cache mechanism for UCIFastReader to allow for much faster access to data after the dataset has been read once.

The following is an example of a BinaryWriter definition. Since it is most commonly used as a cache for UCIFastReader, this definition is show as a UCIFastReader cache. The parameters needed for BinaryWriter are in bold type below:

# Parameter values for the reader with cache

reader=[

# reader to use

readerType=UCIFastReader

# if writerType is set, we will cache to a binary file

# if the binary file exists, we will use it instead of parsing this file

**writerType=BinaryReader**

miniBatchMode=Partial

randomize=Auto

**windowSize=10000**

#### write definition

**wfile=c:\data\mnist\mnist\_train.bin**

#wsize - inital size of the file in MB

# if calculated size would be bigger, that is used instead

**wsize=256**

#wrecords - number of records we should allocate space for in the file

# files cannot be expanded, so this should be large enough.

**wrecords=60000**

features=[

dim=784

start=1

file=c:\data\mnist\mnist\_train.txt

### write definition

#wsize=200

#wfile=c:\data\mnist\mnist\_train\_features.bin

**sectionType=data**

]

labels=[

dim=1

start=0

file=c:\data\mnist\mnist\_train.txt

labelMappingFile=c:\temp\labels.txt

labelDim=10

labelType=Category

#### Write definition ####

# sizeof(unsigned) which is the label index type

#wsize=10

#wfile=c:\data\mnist\mnist\_train\_labels.bin

**elementSize=4**

**wref=features**

**sectionType=labels**

**mapping=[**

#redefine number of records for this section,

#since we don't need to save it for each data record

**wrecords=10**

#variable size so use an average string size

**elementSize=10**

**sectionType=labelMapping**

**]**

**category=[**

**dim=10**

#elementSize=sizeof(ElemType) is default

**sectionType=categoryLabels**

**]**

]

]

]

The example above shows all the parameters necessary to create a Binary file with BinaryWriter:

* **writerType** – The definition of the writer to use. The CNTK code takes this name and appends “.DLL” to dynamically load the DLL and access the writer implementation. BinaryWriter is implemented in the same DLL as BinaryReader, so BinaryReader is the correct value for this setting.
* **windowSize** – the size of the moving window that will be used to access the data in the file. BinaryReader and BinaryWriter both us memory mapped files to support extremely large datasets which cannot be contained in memory. This window size is the minimum amount that will be present in memory at one time.
* **wfile** – the filename and path to the binary file. This can appear at the base of the hierarchy (as in this case) which means all sub-sections defined will be saved in a single file. Alternately, separate files can be defined for different sub-sections. The commented out sections in the feature and labels sections show how separate binary files could be saved. Simply comment out the based definition and uncomment the sub-section definitions and two separate files will be created.
* **wsize** – used in conjunction with wfile, defines how large (in Megabytes) the initial filesize should be. It must be large enough to contain all data or an error will occur. Once the file is completely written it will shrink down to its actual size.
* **wrecords** – the number of records that will be written to disk. Different subsections can override this value if the number of records for a subsection are different (as is the case for the label mapping table subsection)

Each subsection in the configuration will create a corresponding subsection in the binary file. The name used for the subsection in the configuration file will be saved for each subsection, and will be name used to refer to the data when it is read later. Each subsection must have a few parameters:

* **sectionType** – the type of section this config section is describing. The possibilities are:

|  |  |
| --- | --- |
| **Section Type** | **Description** |
| Data | data section (floating point values) |
| Labels | label data (floating point values). For category labels the integer part of this floating point value will be interpreted as an index into the label mapping table. |
| LabelMapping | label mapping table (array of strings) |
| Stats | data statistics. can compute the following statistics across the entire dataset: sum:count:mean:variance:stdDev:max:min:range:rootMeanSquare |
| CategoryLabels | labels in category format (floating point type - all zeros with a single 1.0 per column that matches the mapping table) This format is directly usable in this form for training, so it can be stored in this form. |

* **elementSize** – size in bytes of the elements. If this value is not specified, it will default to the sizeof(ElemType), where ElemType is either float or double based on the precision specified in the configuration file.
* **wref** – A reference to the section that holds the data referenced by these labels. It is often best to store both the labels and the data in the same file so they remain associated with each other. If separate label and data binary files that were generated at different times are used together an error will occur (as they are likely not aligned to each other)
* **dim** – used for categoryLabels format sections. It contains the number of columns of category data, which will contain all zeros except for a single 1.0 for each row.

# Programmer Reference

This section covers topics that are of interest to those who wish to modify the code and use the provided classes in there code. The first section covers configuration files and their use from a programmer’s perspective. The second section covers Reader/Writer interfaces for data input/output. The third section covers the CNTKMath library, and the last section covers using PTask to enable a computation node to participate in multi-GPU processing.

## Configuration Files

Configuration files, and easy manipulation of these files is a main feature of CNTK. The configuration files make the users life much easier, and the programmer interface is also quite simple to use. The programmer interface to the configuration files is contained in a few C++ classes and focuses on “just-in-time” evaluation of the parameter values. The idea is simple, leave the configuration values in string format until they actually need to be parsed into some other form.

### Configuration Formats

The following is a summary of the different formats the configuration classes can support:

|  |  |  |  |
| --- | --- | --- | --- |
| Config type | C++ type | Format | Notes |
| integer | int  long, short  size\_t | [-]###### | Optional sign and numeric digits  all signed and unsigned integer numeric types |
| floating-point | float  double | [-]####.#####[e{+-}###] | Numeric digits with a decimal point, optional sign,optional scientific notation |
| string | std::wstring  std::string | Any valid character | If contained in an array or dictionary and the default separator is contained in the string (i.e. c:\temp in an array) use alternate separator. |
| boolean | bool | T,True,1  F,False,0 | True or False values, may be specified by existence or absence of a boolean parameter with no ‘=’ or value after the parameter name |
| array | ConfigArray | value:value:value  value:value\*#:value  {|value|value|value}  {|value|value\*#|valve}  {  value  value  value\*#  } | Multiple values in an array are separated by colons ‘:’. A value may be repeated multiple times with the ‘\*’ character followed by an integer (the # in the examples).  Values in an array may be of any supported type and need not be uniform. The values in a vector can also be surrounded by curly braces ‘{}’, braces are required if new lines are used as separators.  An alternate separation character can be specified immediately following the opening brace if desired. |
| dictionary | ConfigParameters | parameter1=value1;parameter2=value2;boolparam  [#parameter1=value1#parameter2=value2#boolparam]  [  parameter1=value1  parameter2=value2  boolparam  ] | Multiple parameters grouped together in a dictionary. The contents of the dictionary are each named values and can be of different types. Dictionaries can be used to create a configuration hierarchy. When specified on the same line a ‘;’ semicolon is used as the default separator. The values can optionally be surrounded by square braces ‘[]’. Braces are required when using newlines as separators in a config file. An unnamed dictionary is also allowed in the case of an array of dictionaries.  An alternate separation character can be specified immediately following the opening brace if desired. |

### Configuration classes

There are three main classes that are used to access configuration files. *ConfigParameters* and *ConfigArray* contain instances of *ConfigValue*. The main definitions are as follows:

class ConfigValue : public std::string

class ConfigParameters : public ConfigParser, public ConfigDictionary

class ConfigArray:public ConfigParser, public std::vector<ConfigValue>

#### ConfigValue

The key class that is used to allow the JIT evaluation of configuration strings is the *ConfigValue* class. This class inherits from std::string, and stores an optional configuration path string, which is mainly used for error messages. It contains many cast operators that do the actual parsing of the string value into the target type on demand.

#### ConfigParameters

Configuration files are mainly made up of a hierarchy of Configuration Sets (*ConfigParameters*), which are dictionaries of *ConfigValue*. This class provides access to the configuration values and automatically searches up the hierarchy of ConfigParameter classes if a value is not found on the current level. The hierarchy is maintained by the order of class instantiations on the stack. ConfigParameters should only be created on the stack.

In configuration files the ‘name=value’ named pair are usually separated by newlines. However, they also can be separated by other characters and placed on the same line. The default separator for ConfigParmeters is a ‘;’ (semicolon). This can be overridden by placing the alternate separator character immediately following the opening brace. For example ‘[|’ causes ‘|’ to be the separator for that ConfigParameter instance:

name=[|parameter1=value1|parameter2=value2|parameter3=value3]

There are two types of ConfigParameters type accessors:

**value = config(“name”)** – which will return the named parameter cast to the type of the value parameter. If the named configuration parameter does not exist an exception will be thrown.

**value = config(“name”, “defaultValue”)** – returns the named parameter, if it doesn’t exist returns defaultValue.

**config.Exists(“name”)** – returns the existence of a named value in the *ConfigParameters* instance.

To insert elements into a *ConfigParameters* variable the following methods can be used:

**config.Insert(“name”, value)** – insert a new value into the existing *ConfigParameters* instance. If the value already exists, it will be replaced unless the value is itself another *ConfigParameters* instance, or string representation surrounded by ‘[]’ square braces, in which case the parameters are “merged”.

**config.Insert(value)** – insert the passed string into the dictionary, it is expected to be in ‘name=value’ format.

#### ConfigArray

This type inherits from std::vector<ConfigValue> and can be used to hold arrays of values. Since ConfigValue is a just-in-time evaluation type. The values in the array need not be homogeneous, as long as the code that evaluates the array in the end knows how to interpret the values.

In a ConfigArray the array values are normally separated by the default separator character, which is a ‘:’ (colon). However, they also can be separated by other characters and or place each value on a separate line. The default separator can be overridden by placing the alternate separator character immediately following the opening brace. For example ‘{|’ causes ‘|’ to be the separator for a ConfigArray instance:

array={|c:\temp\new.txt|12\*3|1e-12}

A value may be repeated multiple times with the ‘\*’ character followed by an integer. In the above example, there are 5 elements in the array, with three ‘12’ values occupying the center 3 positions.

The values in a ConfigArray can be accessed just like values in a normal std::vector type, but the automatic type conversion of ConfigValue will still be in affect.

### Other Useful Configuration Methods

Another convenient method that exists for both ConfigParameters and ConfigArray types is a method to load a config file into an existing instance. It is implemented in ConfigParser, which both of these classes inherit from:

config.LoadConfigFile(path.c\_str());

To use this method with a ConfigArray, the file can simply contain a list of values each on their own line and they will be read into the ConfigArray. More complex types can also be contained in the array using the config syntax discussed earlier in this document. An array can contain other arrays as well as ConfigParameters.

ConfigArray instances can also be converted to argvector<T> instances simply by assigning them. Care should be taken to assign to a local variable, and not just passing as a parameter due to lifetime issues, as follows:

ConfigArray configLearnRatesPerMB = config("learningRatesPerMB");

argvector<float> learnRatesPerMB = configLearnRatesPerMB;

ConfigParameters and ConfigArray instances are very flexible, but require parsing every time a value is accessed. argvector<T> ,on the other hand, parses once and then accesses values as a standard vector.

### Configuration Program Example

Some sample code that would parse the example configuration file given at the beginning of this document follows. This is a revised version of actual code in CNTK:

#include "commandArgUtil.h"

// process the command

void DoCommand(const ConfigParameters& config)

{

ConfigArray command = config("command");

for (int i=0; i < command.size(); i++)

{

//get the configuration parameters that match the command

ConfigParameters commandParams=config(command[i]);

ConfigArray action = commandParams("action","train");

// determine the action to perform, and do it

for (int j=0; j < action.size(); j++)

{

if (action[j] == "train")

DoTrain(commandParams);

else if (action[j] == "test" || action[j] == "eval")

DoEval(commandParams);

else

throw runtime\_error("unknown action: " + action[j] + " in command set: " + command[i]);

}

}

}

void DoTrain(const ConfigParameters& config)

{

ConfigParameters configSGD=config("SGD");

ConfigParameters readerConfig = config("reader");

IComputationNetBuilder\* netBuilder = NULL;

ConfigParameters configNDL = config("NDLNetworkBuilder");

netBuilder = (IComputationNetBuilder\*)new NDLBuilder(configNDL);

DataReader\* dataReader = new DataReader(readerConfig);

ConfigArray learningRatesPerMBStr = configSGD("learningRatesPerMB", "");

floatargvector learningRatesPerMB = learningRatesPerMBStr;

ConfigArray minibatchSize = configSGD("minibatchSize", "256");

size\_t epochSize = configSGD("epochSize", "0");

if (epochSize == 0)

{

epochSize = requestDataSize;

}

size\_t maxEpochs = configSGD("maxEpochs");

wstring modelPath = configSGD("modelPath");

int traceLevel = configSGD("traceLevel", "0");

SGD = sgd(learningRatesPerMB, minibatchSize, epochSize, maxEpochs, modelPath, traceLevel);

sgd.Train(netBuilder, dataReader);

delete netBuilder;

delete dataReader;

}

The code above is very easy to code, you simply delare a config, or basic type variable on the stack and assign something from a ConfigParameters class to that variable (i.e. int i = config(”setting”,”default”). Both parameters with defaults and those that don’t are used in the sample code above. The ConfigValue class takes care of parsing the value to be the correct type, and is returned by config() references above.

The Config classes are meant to be used on the stack as shown in this example. Storing them in member variables or allocating using ‘new’ or other methods is not supported. The reason for this is an internal pointer is used to link to parent instances of config classes. This allows us to trace “up the stack” and look at all the config parameters that exist at a higher level. Since our search traverses up the stack, we need to ensure that all the parent configuration classes still exist, which is guaranteed if all config parameters are stack allocated and have lifetimes that extend past any children.

## Data Interfaces

CNTK was designed with the idea that data input and output would need to transpire in many different formats. So data interfaces were designed in an attempt to cover various data needs. Currently there are two data interfaces designed, one for input and the other for output called IDataReader and IDataWriter respectively. The reader/writer code is housed in separate DLLs that which are dynamically loaded to provide data services. This allows the user to simply change a configuration setting and have a different reader provide the data.

Other possible scenarios are also enabled by using a common interface, for example one reader/writer can act as a cache for another slower reader. UCIFastReader is a text-based reader, and though it is very fast, there is still a significant amount of overhead to parsing, so BinaryWriter/BinaryReader can act as a cache for UCIFastReader. The caching code is currently implemented in UCIFastReader.

The five readers and one writer provided with CNTK all use these same interfaces and each is housed in its own DLL. CNTK loads the DLL and looks for exported functions that will return the interface of interest. The functions are defined as follows:

extern "C" DATAREADER\_API void GetReaderF(IDataReader<float>\*\* preader);

extern "C" DATAREADER\_API void GetReaderD(IDataReader<double>\*\* preader);

extern "C" DATAWRITER\_API void GetWriterF(IDataWriter<float>\*\* pwriter);

extern "C" DATAWRITER\_API void GetWriterD(IDataWriter<double>\*\* pwriter);

each reader or writer DLL exports the appropriate functions, and will return the interface when called. The following sections defined the interfaces:

### Reader Interface

/ Data Reader interface

// implemented by DataReader and underlying classes

template<class ElemType>

class DATAREADER\_API IDataReader

{

public:

typedef std::string LabelType;

typedef unsigned LabelIdType;

virtual void Init(const ConfigParameters& config) = 0;

virtual void Destroy() = 0;

virtual void StartMinibatchLoop(size\_t mbSize, size\_t epoch, size\_t requestedEpochSamples=requestDataSize) = 0;

virtual bool GetMinibatch(std::map<std::wstring, Matrix<ElemType>\*>& matrices) = 0;

virtual const std::map<typename LabelIdType, typename LabelType>& GetLabelMapping(const std::wstring& sectionName) = 0;

virtual void SetLabelMapping(const std::wstring& sectionName, const std::map<typename LabelIdType, typename LabelType>& labelMapping) = 0;

virtual bool GetData(const std::wstring& sectionName, size\_t numRecords, void\* data, size\_t& dataBufferSize, size\_t recordStart) = 0;

virtual bool DataEnd(EndDataType endDataType) = 0;

// Recursive network specific methods

virtual size\_t NumberSlicesInEachRecurrentIter() = 0;

virtual void SetNbrSlicesEachRecurrentIter(const size\_t) = 0;

virtual void ReloadLabels() = 0;

virtual void SaveLabels() = 0;

virtual void SetSentenceEndInBatch(vector<size\_t> &sentenceEnd)=0;

};

The methods are as follows:

* **Init** – initialize the reader from a set of ConfigurationParameters. See the reader documentation for elements of readers that should be similar across all types.
* **Destroy** – the “destructor” for the reader. Since we are being called from external code we use an explicit method rather than a normal c++ destructor, but the intent is the same.
* **StartMinibatchLoop** – Starts the minibatch loop with the following parameters:
  + **mbSize** – minibatch size
  + **epoch** – epoch number we are currently processing
  + **requestedEpochSize –** the number of records in an epoch. This value is not required to be the same as the dataset size, it also could be larger or smaller. If the datasetSize is not known the constant requestDataSize can be used to request a single pass through the dataset equal the epochSize.
* **GetMinibatch –** Get the minibatch data
  + **matrices –** a dictionary that maps from the matrix name to the actual matrix. The names of the matrices in the dictionary should be equal to the subsections in the reader configurations.
  + returns – true if there is more data, false if end of epoch is reached.
* **GetLabelMapping –** Get the label map from the reader
  + **sectionName –** the section which contains the label map, if applicable. Some readers do not need a section name if only one label map is supported
  + **returns –** the label map from labelId (integer) to label (std::string)
* **SetLabelMapping –** Set the label map for the reader
  + **sectionName –** the section which is assigned to the label mapping, if applicable. Some readers do not need a section name, they generally only support one label map.
  + **labelMapping –** the labelMap that is being set
* **GetData –** Get some data from a predefined section
  + **sectionName –** the section which contains the data
  + **numRecords –** the number of records to read
  + **data –** pointer to the data buffer, must have enough room to hold the data
  + **dataBufferSize –** size of the buffer, if zero is passed in, or null is passed for the data pointer the number of bytes required in the buffer will returned in this variable
  + **recordStart –** the record to start reading from
* **DataEnd** – Are we at the end of a data section?
  + **endDataType** – type of ending we are checking (Dataset, Epoch, Sentence)
  + **returns** – true or false
* NumberSlicesInEachRecurrentIter
* SetNbrSlicesEachRecurrentIter
* ReloadLabels
* SaveLabels
* SetSentenceEndInBatch

### Writer Interface

// Data Writer interface

// implemented by some DataWriters

template<class ElemType>

class DATAWRITER\_API IDataWriter

{

public:

typedef std::string LabelType;

typedef unsigned LabelIdType;

virtual void Init(const ConfigParameters& config) = 0;

virtual void Destroy() = 0;

virtual void GetSections(std::map<std::wstring, SectionType, nocase\_compare>& sections) = 0;

virtual bool SaveData(size\_t recordStart, const std::map<std::wstring, void\*, nocase\_compare>& matrices, size\_t numRecords, size\_t datasetSize, size\_t byteVariableSized) = 0;

virtual void SaveMapping(std::wstring saveId, const std::map<typename LabelIdType, typename LabelType>& labelMapping) = 0;

};

The methods are as follows:

* **Init** – initialize the writer from a set of ConfigurationParameters. See the writer documentation for an example of BinaryWriter.
* **Destroy** – the “destructor” for the writer. Since we are being called from external code we use an explicit method rather than a normal c++ destructor, but the intent is the same.
* **GetSections** – Gets the sections that are available in the file to write to:
  + **sections** – sections that are defined to write to
* **SaveData** – Save data to the file
  + **recordStart –** the record to start reading from
  + **matrices –** a dictionary that maps from the section name to the data pointers. The names of the sections in the dictionary should be equal to the sections returned by GetSections().
  + **numRecords –** number of records to write
  + **datasetSize –** size of the dataset
  + **byteVariableSized –** for variable sized data, the number of bytes used by the data
* **SaveMapping –** save the mapping table
  + **saveId –** the section name or other id where the mapping will be saved
  + **labelMapping –** the label map from labelId (integer) to label (std::string)

## CNTKMath Library

The CNTK Math library is implemented in the DLL CNTKMath.dll and provides a library of math routines for dealing with matrix operations. The library supports CPU and GPU computation with sparse and dense matrix formats.

The library contains a wrapper class called Matrix<ElemType> (where ElemType is float or double) that hides the differences between the multiple matrix implementations and takes care of data transfers between the GPU and CPU. GPUs and CPUs have different memory spaces, and copying data between them is necessary to access or modify the data from either device. The library attempts to keep data on the GPU as much as possible if a GPU is being used.

When data is accessed or modified from the CPU, if the data is currently on the GPU the matrix will automatically be relocated to the CPU, and relocated back when the GPU attempts to access or modify the data. Currently the entire matrix object is transferred, so care should be taken when accessing matrix data from the CPU.

The library uses BLAS libraries from NVidia for the GPU (CuBLAS) and AMD for the CPU (AMCL). Other third party libraries that are used include CuRand (for random number generation) and CuSparse (for sparse matrix implementations),

## PTask support

PTask is a library used in CTNK to enable multiple GPU computation on a single machine. PTask uses the concept of a “Tasks organized in a filter graph. It allows fully asynchronous operation of the tasks, each only depending on inputs being available to execute. PTask distributes the tasks across the available hardware and handles data transfers.

CTNK is organized in a different fashion with Computation Nodes. However, each node has two methods that do all the computation work: EvaluateThisNode() and ComputeInputPartial(), which can be used as the “Tasks”. However, since Tasks can be executed asynchronously, they need to be stateless. To enable these methods as task a static version of each method that takes all inputs and outputs as parameters are created. The class methods simply call these “Task” functions with the class variables for their implementation.

The PTaskGraphBuilder component takes a computation network and transforms it into a filter graph. In order to do this work it requires the parameter description for each of the tasks. Since C++ does not have a reflection mechanism as in available in C# and some other languages, a class method has been introduced to ComputationNode to provide this information. The method GetPTaskDescriptor() provides this information to PTaskGraphBuilder so it can build the graph.

The following is an example of a GetPTaskDescriptor() implementation. This function returns a TaskDescriptor class containing all the parameter and other information necessary to build the filter graph for a particular node. This node is the “TimesNode” and does a matrix multiply. The following implementation of the two important member functions are:

virtual void EvaluateThisNode()

{

EvaluateThisNodeS(FunctionValues(), Inputs(0)->FunctionValues(), Inputs(1)->FunctionValues());

}

virtual void ComputeInputPartial(const size\_t inputIndex)

{

if (inputIndex > 1)

throw std::invalid\_argument("Times operation only takes two inputs.");

if (inputIndex == 0) //left derivative

{

ComputeInputPartialLeft(Inputs(1)->FunctionValues(), Inputs(0)->GradientValues(), GradientValues());

}

else //right derivative

{

ComputeInputPartialRight(Inputs(0)->FunctionValues(), Inputs(1)->GradientValues(), GradientValues());

}

}

The GPTaskDescriptor() method describes the necessary parameter information for each method. Each node has a FunctionValue matrix and a GradientValue matrix associated with it, and the descriptor methods identify which values are needed, and if they come from the current node or one of its inputs as follows:

// GetTaskDescriptor - Get a task descriptor for this node

// taskType - task type we are generating a task for

virtual TaskDescriptor<ElemType>\* GetPTaskDescriptor(TaskType taskType, size\_t inputIndex=0) const

{

TaskDescriptor<ElemType>\* descriptor = new TaskDescriptor<ElemType>(this, taskType, inputIndex);

switch(taskType)

{

case taskComputeInputPartial:

descriptor->FunctionParam(1-inputIndex, paramOptionsInput);

descriptor->GradientParam(inputIndex, paramOptionsInput | paramOptionsOutput | paramOptionsInitialize);

descriptor->GradientParam();

descriptor->SetFunction( (inputIndex?(FARPROC)ComputeInputPartialRight:(FARPROC)ComputeInputPartialLeft));

break;

case taskEvaluate:

descriptor->FunctionParam();

descriptor->FunctionParam(0, paramOptionsInput);

descriptor->FunctionParam(1, paramOptionsInput);

descriptor->SetFunction((FARPROC)EvaluateThisNodeS);

break;

default:

assert(false);

throw std::logic\_error("Unsupported task requested");

}

return descriptor;

}

For the Evaluate method, the first parameter is an output to the FunctionValue matrix of the current node.

descriptor->FunctionParam();

The default value for this method is “current node, output” so no parameters are needed. The next two parameters are inputs and are the function values from the two inputs:

descriptor->FunctionParam(0, paramOptionsInput);

descriptor->FunctionParam(1, paramOptionsInput);

The last call passes a pointer to the task function:

descriptor->SetFunction((FARPROC)EvaluateThisNodeS);

and the descriptor is complete. The two ComputeInputPartial task function parameters are very similar. Depending on the inputIndex, the values are switched. The first parameter is an input of the function value of one of the inputs, and the second is an output value to the gradient matrix of the other input:

descriptor->FunctionParam(1-inputIndex, paramOptionsInput);

descriptor->GradientParam(inputIndex, paramOptionsInput | paramOptionsOutput | paramOptionsInitialize);

The second parameter is interesting because it is required to retain it value from one call to the next, this is done in a filter graph by having a parameter be input and output at the same time, meaning it updates itself. There is a clear distinction between values that need to be maintained and those that are transcient in a filter graph, and this idiom is how we instruct PTaskGraphBuilder to retain the value. The Initialize option is also necessary so on the first iteration the matrix will be cleared out (zeros).

The last parameter is the gradient matrix of the current node, and is an input (defaults for this function).

descriptor->GradientParam();

Lastly, the task functions must be set. They are different based on which input we are computing the gradient for:

descriptor->SetFunction((inputIndex ? (FARPROC)ComputeInputPartialRight : (FARPROC)ComputeInputPartialLeft));

For reference the three task functions are as follows:

static void WINAPI ComputeInputPartialLeft(Matrix<ElemType>& inputFunctionValues, Matrix<ElemType>& inputGradientValues, const Matrix<ElemType>& gradientValues)

static void WINAPI ComputeInputPartialRight(Matrix<ElemType>& inputFunctionValues, Matrix<ElemType>& inputGradientValues, const Matrix<ElemType>& gradientValues)

static void WINAPI EvaluateThisNodeS(Matrix<ElemType>& functionValues, const Matrix<ElemType>& input0, const Matrix<ElemType>& input1)

## NDL classes and processing

The ability to describe a network architecture in NDL (Network Description Language) is one of the major features of CNTK. However, it is not immediately obvious to the developer looking at the code how it all works. This is meant to shed a little light on the inner workings of NDL processing in CNTK.

NDL is based on the same configuration parser that is used for config files and MEL (Model Editing Language). While this is convenient to share code, it also makes things a little less clear when viewing the code. The configuration file classes, MEL class, and NDL classes all inherit from ConfigParser, which provides the basic parsing, bracket matching, and other common features (quote handling, etc.). The parsing engine implemented in ConfigParser calls back to a virtual method called ParseValue() when it has a token that needs to be interpreted. So ParseValue() in the NDLScript class is the main location where interpretation of tokens takes place.

NDL supports Macros, which makes things much more convenient, but a bit messier for the developer to deal with. All the macros are parsed and stored in a Global script so they can be accessed by any NDL script. It also means that you don’t want to load or define a set of macros more than once, or you will get “function already exists” errors.

The processing of NDL proceeds through the following phases:

1. Parsing the script
2. Evaluation – initial pass – create ComputationNodes for all NDLNodes that require it
3. Evaluation – second pass – connect the inputs of the ComputationNodes
4. Validate the network – This also has the side effect or allocating all the matrix classes to their correct dimensions, and computing dimensions derived from input nodes
5. Evaluation – final pass – All operations that must have the matrix values present occur here. For example, matrix initialization happens here

There is a helper class in NDLUtil, which will take care of executing through these various phases. It also tracks how far along in the current processing phase a script has progressed. Processing can continue statement by statement as needed. This is the want in-line NDL is processed.

Now a little more detail is in order for these layers:

### Parsing

* The script in question is first parsed, and as each Macro Definition, macro call, parameter, variable, or function call is encountered an NDLNode is created. This NDLNode describes the entity and a reference is stored in the NDLScript class which owns it so it can be freed at some later point in time.
* If the NDLNode is an executable statement, it will be added in order to the list of statements to execute
* All variable names used in a script will be added to a symbol table in the NDLScript class that owns the NDLNode.
* If the NDLNode is a macro or function call its parameters will be parsed and added to a parameter list in the NDLNode. Note that parameters may actually be other function and macro calls. The actual parameter names used in the call and the names used in the macro that will be called are recorded.
* If the NDLNode is a macro, it will have its own NDLScript, and contain its own list of executable statements. It will also be stored in the Global Script repository (just a global NDLScript class)

### Evaluation – initial pass

* The main purpose of this pass is to create a Computation Node for every NDL node that requires one. Effectively every “Function call” in NDL maps to a Computation Node. The full “dot path” will be the name of the node in the Computational Network.
* Each pass evaluates the entire script, but only certain actions are performed based on what pass is being executed. So though all the parameters are evaluated in NDL, only Function Calls will create computation nodes.

### Evaluation – second pass

* This pass goes through the entire evaluation process again, but this time all Computation Network nodes should already exist. The main purpose of this pass is to hook up all the inputs between nodes.
* Doing this in a separate pass allows nodes to be referenced before they are actually defined in the NDL Script. This is a necessary feature for Recursive Neural Networks with a DelayNode.
* Having a separate pass allows inline-NDL to support DelayNodes, and “Just-in-time” execution of inline-NDL can occur, where depending on the MEL command that is being executed, we can evaluate the NDL to the appropriate level. Some MEL commands need a “final-state” network to excute safely, others may only require the initial pass to be completed. The initial pass is executed when the inline-NDL is encountered, and how much evaluation has happened for each node is tracked.
* At the end of this pass the computational network is fully connected and complete

### Validation

Validation multiple purposes:

* Ensure that the network is fully connected and that all necessary network nodes exist
* Ensure that the dimensions of the matrices passed to nodes are compatible with the nodes
* Calculates dimensions that depend on input dimensions. This is a feature of Convolutional Networks to make them easier to define.
* Allocate the memory for the matrices
* Ensure the special nodes (CriteriaNode, Features, Labels, Output, etc.) exist if required for this network

### Evaluation – final pass

* This pass does special processing that requires the matrices to exist. As an example there is an optional parameter for the Parameter() function that allows a parameter to be initialized in various ways (zero, random values, from a file), this requires the matrix to be there, so it is done in the final pass.

### Evaluation Process

Now that the entire has been explained, a little more detail on how each Evaluation pass occurs seems relavant.

* First the NDLScript::Evaluate() method is called. This take a few parameters the nodeEvaluator (which creates the actual Computation Nodes), and a baseName, and which pass we are on.
* NDLScript::Evaluate() loops through it’s list of NDL statements and calls NodeEvaluator::Evaluate() of NodeEvaluator::EvaluateMacro() on each of them with the current baseName
* For Macros EvaluateMacro() is called. This takes care of getting the arguments into the target macros symbol table with the correct actual parameters names associated with them. And calls the NDLScript::Evaluate() on the macro script, it also appends to the baseName so it’s still correct.
* In subsequent passes, the node is looked up using the same name instead of created
* The parameters (if any) to the node are evaluated in the first two passes, and in the second pass are assigned as inputs to the computation nodes.
* In the second pass optional parameters are also processed. The “tag” optional parameter must be processed to identify features, labels, etc.
* “dotNames” are currently handled separately, whenever one is referenced the baseName of the parent is added as a prefix to the current scoped name and looked up directly in the computational network.
* There is an “evalValue” that gets set whenever a computational node is resolved to be associated with an NDLNode. However, when macros are called, this will usually be the wrong value, since it will hold the last time the macro was called, not necessarily the instance desired. To alleviate this issue, evalValues are always cleared out on every macro call and the values are used immediately after evaluation, otherwise the values may be different than expected.

Though this explanation may not be complete hopefully it has been instructive, and if all else fails, trace through it in the debugger, and it will likely help.