Model Editing Language

# Definition

The Model Editing Language (MEL) of the Computational Network ToolKit (CNTK) provides a means to modify an existing trained network using a set of provided commands. It provides a number of functions to modify the network and can use Network Description Language (NDL) to define new elements. It looks similar to a scripting language in syntax, but is not a programming “language”, but a simple way to modify an existing network. This network must have been defined in a format that CNTK can read, currently only the CNTK computational network disk format is supported. This document assumes some knowledge of NDL, reading the NDL document prior to this document is recommended.

# Example

This section will cover the features of the MEL by example. If you would rather see the “programmer documentation” just skip to MEL Reference section.

Here is a simple example of a MEL script:

model1 = LoadModel(“c:\models\mymodel.dnn”, format=cntk)

SetDefaultModel(model1)

DumpModel(model1, “c:\temp\originalModel.dmp”, includeData = true)

#Let’s create another hidden layer

Copy(L3.\*, L4.\*, copy=all)

#Now hook up the layer

SetInput(L4.\*.T, 1, L3.RL) # Layer 3 output to Layer 4 input

SetInput(CE.\*.T, 1, L4.RL) # Layer 4 output to Top layer input

#Add mean variance normalization using in-line NDL

meanVal = Mean(features)

invstdVal = InvStdDev(features)

inputVal = PerDimMeanVarNormalization(features,meanVal,invstdVal)

#make the features input now take the normalized input

SetInput(L1.BFF.FF.T, 1, inputVal)

#save model

SaveModel(“c:\models\mymodel4HiddenWithMeanVarNorm.dnn”)

This MEL script is using a network that was defined originally by the following NDL script:

# constants defined

# Sample, Hidden, and Label dimensions

SDim=784

HDim=256

LDim=10

features=Input(SDim, tag=feature)

labels=Input(LDim, tag=label)

# Layer operations

L1 = RBFF(features, HDim, SDim)

L2 = RBFF(L1, HDim, HDim)

L3 = RBFF(L2, HDim, HDim)

CE = SMBFF(L3, LDim, HDim, labels, tag=Criteria)

Err=ErrorPrediction(labels, CE.F, tag=Eval)

# rootNodes defined here

OutputNodes=(CE.F)

## Loading a model

The first thing command executed in a MEL script is usually a LoadModel() command. This function takes the name of a model file on disk, and an optional parameter specifying the format of the model file. Currently only CNTK format model files are accepted, and CNTK format is the default value. Programmers can write file converters to support more model formats.

model1 = LoadModel(“c:\models\mymodel.dnn”, format=cntk)

SetDefaultModel(model1)

‘model1’ is the identifying name this model is given for use in the MEL script. This identifier is used in the next line to this model as the default model. The default model defines what model will be assumed in all name references within the script, and the model to which any NDL (Network Definition Language) commands will apply. This line isn’t really necessary in this case, because the first model loaded will be the default model without explicitly calling the SetDefaultModel() function.

## Viewing a model file

It is often necessary to view a model file to determine the names used in the model file. MEL uses the node names in most commands, to specify which node(s) should be modified. The Dump() command dumps the node names and optionally values to a file.

DumpModel(model1, “c:\temp\originalModel.dmp”, includeData = true)

the parameters are the model name, the file name, and if the dump should include data. The includeData optional parameter defaults to false. The dump looks something like this:

…

features=InputValue [784,32]

L1.BFF.B=LearnableParameter [256,1] NeedGradient=true

0.0127850091

-0.00473949127

0.0156492535

…

0.00529919751

####################################################################

L1.BFF.FF.P=Plus ( L1.BFF.FF.T , L1.BFF.B )

L1.BFF.FF.T=Times ( L1.BFF.W , normInput )

L1.BFF.W=LearnableParameter [256,784] NeedGradient=true

0.0174789988 0.0226208009 -0.00648776069 0.0346485041 -0.0449098013 -0.0233792514

0.0154407881 0.000157605857 0.0206625946 0.0491085015 0.00128563121

…

These variables are set to scalar numeric values in this case and are used as parameters in the NDL Functions. These values are the dimensions of the data samples, hidden layers, and labels used in training. This particular setup is for the MNIST dataset, which is a collection of images that contain 784 pixels each. Each image is a handwritten digit (0-9), so there are 10 possible labels that can be applied to each image. The hidden matrix dimensions are determined by the user depending on their needs.

## Copy

The copy command will copy a node, or a group of nodes from one location to another location. This can be done within the same model, or between different models:

#Let’s create another hidden layer

Copy(L3.\*, L4.\*, copy=all)

The first parameter is the source of the copy and must exist, the second is the target and may or may not exist. If it does exist, those matching nodes will be overwritten by the copy. The optional parameter **copy** can be used to change this behavior, the options are: **all** the default which copies all node data and links to other nodes also copied, or **value** whichcopies the node values only, leaving the connections between nodes (if any) unchanged.

In this command an entire layer is duplicated in the same model creating a new L4 layer in the model. The Copy() command will copy the nodes and connections between the nodes being copied by default so the optional parameter was not required in this case.

The L3 used in this copy command was originally defined in NDL as follows:

L3 = RBFF(L2, HDim, HDim)

So the new L4 layer will contain all the nodes L3 contains (RectifiedLinear, Plus, Times, W and B Parameters) all connected just as they were in the L3 layer.

## SetInput

To integrate this new layer into the model, the inputs and outputs must still be set properly. After the copy any node whose connected nodes were not copied will have those connections set to an invalid value. These need to be fixed up in order to have a valid model. Attempts to Save a model will first validate the model in the case where some nodes were not reconnected.

You can change connections between nodes with the SetInput() command. This takes a node to modify, the input number to modify (zero-based input#), and the new value for that input. The following commands hook up the inputs and outputs for our copied nodes:

#Now hook up the layer

SetInput(L4.\*.T, 1, L3.RL) # Layer 3 output to Layer 4 input

SetInput(CE.\*.T, 1, L4.RL) # Layer 4 output to Top layer input

To connect our new L4 layer, we need to set the second input of the Times node (L4.BFF.FF.T) to L3.RL, which is the output of the L3 layer. The input number is zero-based, so the first input is zero and the second input would be '1'.  
Likewise we need to hook the output of the L4 layer nodes to the input of the top layer. Once again this ends up being a Times node (CE.BFF.FF.T)

## Name Matching

You may have noticed the use of the ‘\*’ wildcard character in the commands presented to this point. Those are name matching wildcards, and are useful in matching a group of related nodes. Because of the hierarchal “dot naming” scheme used by NDL, it is easy to select all the nodes that a particular macro generated because they will all start with the same prefix. Nodes generated by NDL macros have the following structure:

[name]{.[macroName]}.[nameNode]

Where **name** is the name assigned in NDL, **macroName** is the name given to a macro called by the initial macro, and can be several layers deep, and **nameNode** is the name given to a single node in the final macro. For Example this macro in NDL:

L3 = RBFF(L2, HDim, HDim)

Generates the following nodes:

|  |  |
| --- | --- |
| 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 |

These patterns can be used to access these nodes:

|  |  |
| --- | --- |
| L3.\* | Select all the L3 nodes |
| L3.\*.P | Select the L3.BFF.FF.P node |
| \*.W | Select L3.BFF.W and any other node named ‘W’ in the model |
| model1.L3.\* | All the L3 nodes in the ‘model1’ model |
| model1[.\*] | All the nodes in model1 (the ‘.\*’) is optional |

There are also methods that will copy nodes based on the structure of the graph. Look for CopySubTree() in the reference section for details.

## Adding new nodes

Adding new nodes to an existing model can be done just like a model can be originally defined, in NDL. There are two ways to do this, the simplest is to just type the NDL definitions into the MEL script, as if it was NDL, like so:

#Add mean variance normalization using in-line NDL

meanVal = Mean(features)

invstdVal = InvStdDev(features)

inputVal = PerDimMeanVarNormalization(features,meanVal,invstdVal)

This is called in-line NDL and can be used for most tasks. This sequence of nodes does a mean variance normalization on the dataset. The new nodes will be placed in the current default model in the MEL script. In our example script, we only use one model, and it was set as the default model using the SetDefaultModel() command. If no model has been explicitly set to be the default model, the last loaded model is used as the default. However, It is recommended that the SetDefaultModel() command be used to make it explicit.

Notice the variable **features** that is used in the NDL is actually a node from the default model. It is legal to use nodes from the model in in-line NDL and vise-versa. However, no name matching '\*' patterns are allowed in NDL commands, and macros cannot be defined in in-line NDL.

An NDL macro can also be used from in-line NDL, as long as it appears in the default macros defined for the editing script, or it is defined in an NDL Snippet (see below).

## Connecting in-line NDL

The sequence of nodes used to do mean variance normalization are now in the model. However, we have to use the output of these NDL nodes to replace the previous InputNode that provided the features. This node is called ‘features’ in this model, and we need to set the input to the L1 layer to be ‘inputVal’ (the output from the NDL nodes we just created) instead. This is done, again, using the SetInput() command:

#make the features input now take the normalized input instead

SetInput(L1.BFF.FF.T, 1, inputVal)

Now the nodes have all been connected and the model is valid, a mean variance normalization step has just been added to the model. The mean() and variance() nodes both execute before the any training begins and are called ‘pre-compute’ nodes. The mean and variance are calculated over the training data set, and then those values are used during training to normalize the data.

# NDL Snippets

NDL snippets are sections of NDL definitions that generate a new model. Any NDL construct that is legal in an NDL script can be used. This includes defining macros and other advanced NDL features. For example, instead of loading an existing NDL file, an NDL snippet could have been used to define the network structure. The NDL Snippet looks like:

model1=[

# constants defined

# Sample, Hidden, and Label dimensions

SDim=784

HDim=256

LDim=10

features=Input(SDim, tag=feature)

labels=Input(LDim, tag=label)

# Layer operations

L1 = RBFF(features, HDim, SDim)

L2 = RBFF(L1, HDim, HDim)

L3 = RBFF(L2, HDim, HDim)

CE = SMBFF(L3, LDim, HDim, labels, tag=Criteria)

Err=ErrorPrediction(labels, CE.F, tag=Eval)

# rootNodes defined here

OutputNodes=(CE.F)

]

When snippets are used, wildcard naming, and use of symbols from another model are not allowed. The syntax rules are identical to creating an NDL script.

## SaveModel

After the model edits are complete, it’s time to save the model:

#save model

SaveModel(“c:\models\mymodel4HiddenWithMeanVarNorm.dnn”)

This command saves the default model (still ‘model1’) to the path name specified. ‘model1’ could have been specified as the first parameter with the path as the second to obtain the same affect. Before the save happens the model is validated to ensure it is a valid model before save can occur. Should there be an error in the model, an error message will be displayed on the console and the model edit will terminate.

# MEL Reference

Model Editing Language (MEL) is a language that provides a means to modify an existing CNTK network, or a trained model to create new networks and models. MEL allows nodes of a network to be copied, new nodes created, and node values to be duplicated to create new networks based on other previously done work.

## Commands

Commands in MEL are the operations that can be used to modify a network or model. The commands are represented in a function call like syntax:

Command(parameter1, parameter2, optionalParameter=value)

Commands do not return values, with the exception of the CreateModel() and LoadModel() commands, and some may have optional parameters. The parameters are delimited with a comma. The commands are:

|  |  |  |
| --- | --- | --- |
| **Command Name** | **Example** | **Notes** |
| CreateModel | m1=CreateModel() | Returns a value |
| CreateModelWithName | CreateModelWithName(model1) | Alternate no return value |
| LoadModel | m1=LoadModel(“new.dnn”, format=cntk) | Returns a value |
| LoadModelWithName | LoadModelWithName(m1, “new.dnn”, format=cntk) | Alternate no return value |
| LoadNDLSnippet | LoadNDLSnippet(mNDL, “net.ndl”) |  |
| SaveDefaultModel | SaveDefaultModel(“new.dnn”, format=cntk) |  |
| SaveModelWithName | SaveModelWithName(m1, “new.dnn”, format=cntk) |  |
| SetDefaultModel | SetDefaultMode(m1) |  |
| UnloadModel | UnloadModel(m1) |  |
| Dump[Model] | Dump[Network](m1, “dump.txt”, includeData=false) | DumpModel is alternate name |
| DumpNode | DumpNode(node, “node.txt”, includeData=false) |  |
| Copy[Node] | Copy(fromNode, toNode, copy=all) | CopyNode is alternate name |
| CopySubTree | CopySubTree(fromNode, toNetwork, toNodeNamePrefix, copy=all) |  |
| Copy[Node]Inputs | CopyInputs(fromNode, toNode) | CopyNodeInputs is alternate name |
| Set[Node]Input | SetInput(fromNode, inputID, inputNode) | SetNodeInput is alternate name |
| Set[Node]Inputs | SetInputs(fromNode, inputNode1[, inputNode2, inputNode3]) | SetNodeInputs is alternate name, variable number of parameters |
| SetProperty | SetProperty(toNode, propertyName, propertyValjue) |  |
| SetPropertyForSubTree | SetPropertyForSubTree(rootNode, propertyName, propertyValue) |  |
| Remove[Node]  Delete[Node] | Remove(node[, node2, node3, …])  Delete(node[, node2, node3, …]) | RemoveNode, DeleteNode are alternate names |
| Rename | Rename(nodeOld, nodeNew) |  |

## Name Matching

MEL provides a way to perform a command on more than one node at a time. This is done through wildcard name matching. Because of the hierarchal “dot naming” scheme used by NDL, related nodes are easy to select with a wildcard name matching scheme. Nodes generated by NDL macros have the following structure:

{[modelName].}[name]{.[macroName]}.[nameNode]

|  |  |
| --- | --- |
| **modelName** | an optional prefix that defines which model should be applied to the rest of the name. If no **modelName** is specified, the current default model is assumed. |
| **name** | the name of the node in question, or if NDL was used to create the network, the top level symbol used to identify the node (i.e. L3 in the following example. |
| **macroName** | the name given to a macro called by the initial macro and can be several layers deep. Usually the names are the same as the macros called. A user is unlikely to know these names unless they dump the network nodes, so wildcard name matching can be used instead of the **macroName** (s) |
| **nameNode** | the name given to a single node in the final macro. |

For Example this macro in NDL:

L3 = RBFF(L2, HDim, HDim)

Generates the following nodes:

|  |  |
| --- | --- |
| 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 |

The following wildcard patterns can be used to select nodes within a model. If a [model] prefix is not specified the default model is assumed:

|  |  |  |
| --- | --- | --- |
| **Pattern** | **Example** | **Result** |
| [prefix]\* | L3.\* | Select all the nodes starting with [prefix] |
| [prefix]\*[suffix] | L3.\*.P | Select all nodes with [prefix] and [suffix] |
| \*[suffix] | \*.W | Select all the nodes with [suffix] |
| [model].[pattern] | model1.L3.\* | Select all the nodes matching a pattern in [model] |
| [model]{\*} | model1.\* | Select all nodes in the model, ‘\*’ is optional |

There are also methods that will copy nodes based on the structure of the graph. Look for CopySubTree() in the reference section for details.

## Optional Parameters

Many commands have optional parameters that will change the behavior of the command. For example:

Copy(L1.\*, L2.\*, copy=all)

In this example all the nodes starting with "L1." are copied to nodes starting with "L2.", the values of the nodes as well as any links between the nodes (the network structure) are copied. If the destination “L2.\*” nodes already exist, they will be overwritten. The other option is copy=value, which would be used when the network structure desired already exists, and the values contained in the node are all that are required to be copied. This can be used to copy over the values of Parameter() nodes to a new model with identical structure.

Each command may have optional parameters, look in the Command reference section for details of the optional parameters that are accepted by a command.   
  
Stringize variables  
MEL supports a “stringize” feature similar to the one supported by configuration files. Anywhere in a MEL script file, you can specify “$VarName$”, and this entire string will be replaced by the value of the variable called “VarName”. Note that the variables that are considered in scope for this purpose are the configuration variables that are visible from the configuration section where the path to this MEL script is specified (via the “editPath” parameter). For example, if the variables “OldModelPath” and “NewModelPath” were defined at the root level of the configuration file, the following would be a proper MEL script:  
  
 m1=LoadModel("$OldModelPath$",format=cntk)  
 # make change to model here  
 SaveModel(m1,"$NewModelPath$",format=cntk)

# NDL Integration

NDL (Network Description Language) can be used freely in MEL to create new nodes and integrate them into an existing model. Please refer to the NDL Section of the documentation to get the details on all the NDL Functions that are available. The NDL Functions can be used in two different ways in MEL. In-line and as a snippet.

## In-line NDL

In-line NDL is, as it sounds, NDL lines mixed in with MEL Command calls. This is an easy way to define new nodes in a MEL script. In-line NDL only works on the default model at the time the NDL function is encountered. The default model is set with the SetDefaultModel() command, or if no such command has been encountered the last LoadModel() or CreateModel() command. It is recommended that the SetDefaultModel() command appear before any In-line NDL to make it clear which model is being modified.

In-line NDL may use node names from the default model as parameters, and MEL commands may use NDL symbols as parameters. There are a number of restrictions using in-line NDL:

1. ‘\*’ names may not be used in In-line NDL, only fully quantified node names are accepted.
2. NDL symbols only apply to the default model at the time they were created when used in MEL commands
3. Macros may not be defined in in-line NDL (though they can in an NDL snippet)
4. Only macros defined in the default macro file referenced in the config file, or macros defined in an NDL snippet in the MEL Script may be used
5. NDL will be processed when the next MEL command that requires it to be processed is encountered. It is only at this time that the new nodes are fully created. If forward references are used to variables, they must be resolved before the next MEL command that requires the variables to be resolved.

## NDL Snippets

NDL snippets are sections of NDL definitions that generate a new model. Any NDL construct that is legal in an NDL script can be used. This includes defining macros and other advanced NDL features. The syntax for defining and NDL snippet are as follows:

[modelName]=[

#ndl commands go here

]

Upon the completion of the snippet, the modelName will be the name of the newly defined model. This model need not be fully defined, for example, the special nodes (i.e. criteria nodes) do not need to be defined in the model. However, all referenced variables must be defined in the snippet. It is often easier to use in-line NDL to define new nodes in MEL, and NDL Snippets to define any macros. Macros are defined in a global namespace and can be defined in any model and used from any other model.

One possible use of an NDL snippet is to define an entirely new model, and then use MEL to populate the new model with values. Here is an example of how an NDL snippet could have been used to define the entire network structure:

model1=[

# constants defined

# Sample, Hidden, and Label dimensions

SDim=784

HDim=256

LDim=10

features=Input(SDim, tag=feature)

labels=Input(LDim, tag=label)

# Layer operations

L1 = RBFF(features, HDim, SDim)

L2 = RBFF(L1, HDim, HDim)

L3 = RBFF(L2, HDim, HDim)

CE = SMBFF(L3, LDim, HDim, labels, tag=Criteria)

Err=ErrorPrediction(labels, CE.F, tag=Eval)

# rootNodes defined here

OutputNodes=(CE.F)

]

When snippets are used, wildcard naming, and use of symbols from another model are not allowed. The syntax rules are identical to creating an NDL script. Alternately, the LoadNDLSnippet() command can be used to load NDL from an external file.

# Comments

Comments in MEL are identical to those used in the NDL and configuration files. 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:

# Layer operations

L1 = RBFF(features, HDim, SDim) # define the first layer

# the following variable is set to infinity and the ‘#’ in ‘1#INF’ is not interpreted as a comment marker

var = 1#INF

# MEL Commands

This section contains the currently implemented MEL Command functions.

### CreateModel

Creates a new model which is empty.

m1=CreateModel()

#### Parameters

none

#### Returns

the new model

#### Notes

This command is one of only a few that return a value. If you prefer to easily distinguish between NDL functions (which always return a value) and MEL commands (which normally do not) you may wish to use the alternate CreateModelWithName() call, which takes the new model identifier as a parameter instead of returning it as a return value.

### CreateModelWithName

Creates a new model which is empty.

CreateModelWithName(m1)

#### Parameters

the identifier for the newly created model

#### Notes

The alternate form of the command is CreateModel() and returns a value. If you prefer to easily distinguish between NDL functions (which always return a value) and MEL commands (which normally do not) you may wish to use this version of the command.

### LoadModel

Load a model from a disk file and assign it a name. The format of the file may be specified as an optional parameter.

m1=LoadModel(modelFileName, [format=cntk])

#### Parameters

modelFileName – name of the model file, can be a full path name. If it contains spaces, it must be enclosed in double quotes.

#### Returns

model identifier for the model that will be loaded

#### Optional Parameters

format=[*cntk*] – Specifies the format of a file, defaults to ‘cntk’. Currently only the native CNTK format of model file is accepted. Other formats may be supported in the future.

#### Notes

This command is one of only a few that return a value. If you prefer to easily distinguish between NDL functions (which always return a value) and MEL commands (which normally do not) you may wish to use the alternate LoadModelWithName() call, which takes the new model identifier as a parameter instead of returning it as a return value.

### LoadModelWithName

Load a model from a disk file and assign it a name. The format of the file may be specified as an optional parameter.

LoadModelWithName(model, modelFileName, [format=cntk])

#### Parameters

model-identifier associated with the model that will be loaded.

modelFileName – name of the model file, can be a full path name. If it contains spaces, it must be enclosed in double quotes.

#### Optional Parameters

format=[cntk] – Specifies the format of a file, defaults to ‘cntk’. Currently only the native CNTK format of model file is accepted. Other formats may be supported in the future.

#### Notes

The alternate form of the command is LoadModel() and returns a value. If you prefer to easily distinguish between NDL functions (which always return a value) and MEL commands (which normally do not) you may wish to use this version of the command.

### LoadNDLSnippet

Load an NDL Snippet from a file, and process it, assigning the results to a symbol

LoadNDLSnippet(model, nsdSnippetFileName[, section=first])

#### Parameters

model – the identifier that will be used to reference this model.

ndlSnippetFileName – name of the file that contains the snippet we want to load

#### Optional Parameters

section=sectionName – name of the section that contains the snippet we want to load. If the entire file is the snippet no section name should be specifiedmsmswscar cars Adam

### SaveModel

Save a model to disk in the specified model format

SaveModel(model, modelFileName[, format=cntk])

#### Parameters

model – the identifier of the model which will be saved  
modelFileName – the file name to save the model as

#### Optional Parameters

format=cntk – the format of file to save. The only valid value currently is CNTK format, which is the default. It is expected that different formats will be added in the future

### SaveDefaultModel

Save the current default model to a file. The format can be specified with an optional parameter

SaveDefaultModel(modelFileName, format=cntk)

#### Parameters

modelFileName – name of the model file to save

#### Optional Parameters

format=cntk – the format of file to save. The only valid value currently is CNTK format, which is the default. It is expected that different formats will be added in the future

### UnloadModel

Unload the specified model from memory.

UnloadModel(model)

#### Parameters

model – model identifier.

#### Notes

In general it is unnecessary to unload a model explicitly since it will happen automatically at the end of the MEL script. It is also not recommended that you reuse a model identifier after unloading a model.

### Dump, DumpModel

Create a text file that represents the contents and structure of the Computational network.

Dump(model, dumpFileName[, includeData=false])  
DumpModel(model, dumpFileName[, includeData=false])

#### Parameters

model – model Identifier  
dumpFileName – file name to save the output

#### Optional Parameters

includeData=[true,false] – (default = false) Include the data contained in a node. This will output the contents of nodes that contain matrix values.

### DumpNode

Create a text file that represents the contents of a node.

DumpNode(node, dumpFileName[, includeData=false])

#### Parameters

node – node Identifier, a wildcard name may be used to output multiple nodes in one call  
dumpFileName – file name to save the output

#### Optional Parameters

includeData=[true,false] – (default = false) Include the data contained in a node. This will output the contents of nodes that contain matrix values.

### Copy, CopyNode

Copy a node, or a group of nodes from one location to another location. This can be done within the same model, or between different models. The copy can create new nodes or overwrite/update existing nodes. The network structure can be copied with multiple nodes, or just the values in the nodes.

Copy(fromNode, toNode[, copy=all)  
CopyNode(fromNode, toNode[, copy=all)

#### Parameters

fromNode – node identifier we are copying from. This can also be a wildcard pattern.

toNode – node identifier we are copying to. This can also be a wildcard pattern, but must match the fromNode pattern. A copy from a single node to multiple nodes is also permitted.

#### Optional Parameters

copy=[all,value] – (default = all). Specifies how the copy will be performed:

|  |  |  |
| --- | --- | --- |
|  | If destination node exists | If destination node does not exist |
| All | Copies over the values of the nodes and any links between them overwriting the existing node values. Any node inputs that are not included in the copy set will remain unchanged. | Copies over the values of the nodes and any links between them creating new nodes. All nodes that include inputs in the copy set will still be connected. All other nodes will have no inputs and will need to be set using SetInput() |
| Value | Copies over the node contents, the node inputs will remain unchanged | Not a valid option, the nodes must exist to copy only values. |

#### Examples

Copy(L1.\*, L2.\*) – copies all the nodes and the inputs in the L1.\* copy set to L2.\*. If the L2.\* nodes did not exist, they will be created

Copy(L1.BFF.FF.W, model2.\*.W, copy=value) – copies values in the L1.BFF.FF.W node to all the nodes in model2 that are use the name ‘W’.

#### Notes

If an entire network is to be copied, it is easier to save the network first (possibly to a temporary location) and reload that model under a new name.

### CopySubTree

Copy all nodes in a subtree of a computational network from one location to another location. This can be done within the same model, or between different models.

CopySubTree(fromRootNode, toRootNode[, copy=all)

#### Parameters

fromRootNode – node identifier we are copying from. This can also be a wildcard pattern.

toRootNode – node identifier we are copying to. This can also be a wildcard pattern, but must match the fromRootNode pattern.

#### Optional Parameters

copy=[all,value] – (default = all). Specifies how the copy will be performed:

|  |  |  |
| --- | --- | --- |
|  | If destination node exists | If destination node does not exist |
| All | Copies over the values of the nodes and any links between them overwriting the existing node values. Any node inputs that are not included in the copy set will remain unchanged. | Copies over the values of the nodes and any links between them creating new nodes. All nodes that include inputs in the copy set will still be connected. All other nodes will have no inputs and will need to be set using SetInput() |
| Value | Copies over the node contents, the node inputs will remain unchanged | Not a valid option, the nodes must exist to copy only values. |

#### Notes

If the fromRootNode is a wildcard pattern then the toRootNode must also be a similar wildcard pattern. The CopySubTree() command will execute separately for each root node.

### SetInput, SetNodeInput

Set an input of a node to a value

SetInput(node, inputNumber, inputNode)

#### Parameters

node – node whose input we are modifying . This can also be a wildcard pattern.

inputNumber – a zero-based index to the input that will be set.

inputNode – node identifier for input node. This must be a single node.

#### Notes

SetInput() or SetInputs() are often required after a Copy() command in order to hook up all the copied nodes into the network.

### SetInputs, SetNodeInputs

Set all the inputs of a node. If only one input needs to be set use the SetInput() command instead.

SetInputs(node, inputNode1[, inputNode2, inputNode3])

#### Parameters

node – node whose input we are modifying .

inputNode1, inputNode2, inputNode3 – node identifier for input node. The number of input parameters must match the number of inputs **node** requires.

#### Notes

SetInput() or SetInputs() are often required after a Copy() command in order to hook up all the copied nodes into the network.

### SetProperty

Set the property of a node to a specific value.

SetProperty(node, propertyName, propertyValue)

#### Parameters

node – the node whose properties will set

propertyName – name of the property to modify.

propertyValue – the value the Property will receive.

The acceptable propertyNames and propertyValues are as follows:

|  |  |  |
| --- | --- | --- |
| PropertyName | Description | PropertyValue |
| ComputeGradient  NeedsGradient | A flag that determine if a node participates in gradient calculations. Applies to Parameter nodes | true/false |
| Feature | Sets the node as a feature input. Applies to Input nodes | true/false |
| Label | Set the node as a label input. Applies to Input nodes | true/false |
| FinalCriterion  Criteria | Sets the node as one of the Criteria nodes of the network | true/false |
| Evaluation  Eval | Set the node as one of the evaluation nodes | true/false |
| Output | Set the node as one of the output node | true/false |

#### Notes

Most of these properties can be set on nodes through alternate methods. All of these properties except for the ComputeGradient can be added (but not removed) using the special node syntax in NDL.

### SetPropertyForSubTree

Set the property of a node to a specific value.

SetProperty(rootNode, propertyName, propertyValue)

#### Parameters

rootNode – the node at the root of the subtree

propertyName – name of the property to modify.

propertyValue – the value the Property will receive.

The acceptable propertyNames and propertyValues for this command are as follows:

|  |  |  |
| --- | --- | --- |
| PropertyName | Description | PropertyValue |
| ComputeGradient  NeedsGradient | A flag that determine if a node participates in gradient calculations. Applies to Parameter nodes | true/false |

#### Notes

The ComputeGradient property only applies to Parameter nodes in the subtree.

### Remove, RemoveNode, Delete, DeleteNode

Delete or Remove node(s) from a model. All alternate command names provide the same option.

Remove(node[, node2, node3, …])

Delete(node[, node2, node3, …])

RemoveNode(node[, node2, node3, …])

DeleteNode(node[, node2, node3, …])

#### Parameters

node – the node to be removed. This can be a wildcard name.

node2, node3 – additional optional nodes that will also be removed, These can be wildcards

#### Notes

This command can leave unconnected nodes in a model which would need to be reconnected using the SetInput() or SetInputs() commands.

### Rename

Rename a node

Rename(oldNode, newNode)

#### Parameters

oldNode – the node name of the old node, wildcard naming may be used.

newNode – the node name for the new node, matching wildcard naming may be used if oldNode contains wildcards.

#### Notes

Renaming nodes has no effect on the node inputs, even if a name changes the association will remain intact.