Network Description Language

# Definition

The Network Description Language (NDL) of the Computational Network ToolKit (CNTK) provides a simple way to define a network in a code-like fashion. It contains variables, and Macros, and other well understood concepts. It looks similar to a scripting language in syntax, but is not a programming “language”, but a simple way to define a network.

# Example

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

Here is a simple example of a network definition:

SDim=784

HDim=256

LDim=10

B0=Parameter(HDim)

W0=Parameter(HDim, SDim)

features=Input(SDim)

labels=Input(LDim)

Times1=Times(W0, features)

Plus1=Plus(Times1, B0)

RL1=RectifiedLinear(Plus1)

B1=Parameter(LDim, 1)

W1=Parameter(LDim, HDim)

Times2=Times(W1, RL1)

Plus2=Plus(Times2, B1)

CrossEntropy=CrossEntropyWithSoftmax(labels, Plus2)

ErrPredict=ErrorPrediction(labels, Plus2)

FeatureNodes=(features)

LabelNodes=(labels)

CriteriaNodes=(CrossEntropy)

EvalNodes=(ErrPredict)

OutputNodes=(Plus2)

This is a simple Neural Network that consist of two layers.

## Variables

The first thing you will notice is that the SDim, HDim and LDim variables. Variable names can be any alphanumeric string (starting with a letter) and are case-insensitive.

SDim=784

HDim=256

LDim=10

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 dimension is determined by the user depending on their needs.

## Parameters

Parameters are matrices that constitute the learned model upon completion of training. The model parameter matrices are used to modify the sample data into the desired output data and are updated as part of the learning process.

B0=Parameter(HDim)

W0=Parameter(HDim, SDim)

These lines setup the parameters that will be trained, W0 is the weight matrix and B0 is the bias matrix. Parameters are matrices, and have two dimension parameters. If only one dimension is given the other dimension is assumed to be a ‘1’. By default Parameters are initialized with uniform random numbers, but other options exist (see NDL Function definitions)

## Inputs

The inputs into the network are defined by the sample data and the labels associated with the samples.

features=Input(SDim)

labels=Input(LDim)

The ‘features’ input will have the dimensions of the sample data, and the ‘labels’ input will have the dimensions of the labels. The variables chosen here are for convenience and could be any valid variable name.

## Computation

The computation portion of the network gets the product of the weight matrix and the features matrix and adds on the bias. It uses the matrix operators Times() and Plus().

Times1=Times(W0, features)

Plus1=Plus(Times1, B0)

RL1=RectifiedLinear(Plus1)

Following this computation we apply the energy function, in this case RectifiedLinear(), to the product. The Sigmoid() function is also available (see NDL Function definitions).

## Top Layer

The top layer in a network is where the neural network produces the probabilities that correspond to the labels provided in supervised learning. This network uses category labels, for the MNIST case these will appear as an array of 10 floating point values, all of which are zero except for the proper label category which is 1.0.

CrossEntropy=CrossEntropyWithSoftmax(labels, Plus2)

Networks will often use the SoftMax function to obtain the probabilities for each label. The error between the actual and the probability is then computed using CrossEntropy. In the CNTK these two actions can be combined in one function for efficiency. CrossEntropyWithSoftmax() takes the input, computes the SoftMax function, calculates the error from the actual value using CrossEntropy and that error signal is used to update the parameters in the network via back propagation.

## Back Propagation

CNTK does not require you to specify anything additional for the back propagation portion of the network. For this example Stochastic Gradient Descent (SGD) is used as the learning algorithm. Each function in CNTK also has a derivative counterpart function and the system automatically does the back propagation update of the network parameters.

## Error Prediction

Predicted error rates are often computed during the training phase to validate the system is getting better as the training progresses. This is handled in the CNTK using the following function:

ErrPredict=ErrorPrediction(labels, Plus2)

The probabilities produced by the network are compared to the actual label and an error rate is computed. This is generally displayed by the system. Though this is useful, it is not mandatory to use ErrorPrediction, and this can be left out of the network if desired.

## Defining special nodes

After defining the network, it’s important to let CNTK know where the special nodes are in the network. For example, the input nodes (which are features, and which are labels), the output nodes, evaluation nodes and Top Layer criteria nodes. CNTK supports multiple nodes for each type, so the values are arrays. The array syntax is comma separated variable names surrounded by parenthesis.

FeatureNodes=(features)

LabelNodes=(labels)

CriteriaNodes=(CrossEntropy)

EvalNodes=(ErrPredict)

OutputNodes=(Plus2)

# Macros

While creating a network using the syntax shown above is not all that difficult, it can get wordy when creating deep neural networks with many layers. To alleviate this problem, common definitions can be combined into Macros. Macros can be defined as nested calls on a single line, or can be in a more function like syntax as can be seen in the following examples:

## Examples

Macro examples:

RFF(x1, w1, b1)=RectifiedLinear(Plus(Times(w1,x1),b1))

This one macro is equivalent to the computation section in the previous section, but all in one line. The parameters used in macros are local to each macro.

FF(X1, W1, B1)

{

T=Times(W1,X1);

FF=Plus(T, B1);

}

This macro is a feed forward computation without the energy function. It shows the alternate format of macros. Semicolons are optional, but can be used if desired. The variables and parameters used inside the macros are local to the macro. The return value of a macro is defined by a local macro variable that has the same name as the macro. In this case the FF() macros return value will be the FF local variable. If no variables match, the last variable in the macro will be returned.

#Base Feed Forward network, includes Bias and weight parameters

BFF(in, rows, cols)

{

B=Parameter(rows)

W=Parameter(rows, cols)

BFF = FF(in, w, b)

}

This macro shows how parameters can be declared within a macro. It also shows the comment syntax using a ‘#’ as the first character in a line, signifies a comment line. As in this example, a macro may call another macro, however recursion is not supported.

RBFF(input,rowCount,colCount)

{

F = BFF(input, rowCount, colCount);

RBFF = RectifiedLinear(F);

}

This macro calls the previous macro adding the RectifiedLinear() energy function for a complete layer.

SMBFF(x,r,c, labels)

{

F = BFF(x,r,c);

SM = CrossEntropyWithSoftmax(labels, F)

}

This macro defines a full Top layer, also using the BFF macro as in the other full layer macro. In this case no variables match the name of the macro, so the SM variable will be used as the return value, since it’s the last value defined in the macro.

## Using Macros

The following example uses the macros defined above

# constants defined

# Sample, Hidden, and Label dimensions

SDim=784

HDim=256

LDim=10

features=Input(SDim)

labels=Input(LDim)

# Layer operations

L1 = RBFF(features, HDim, SDim)

CE = SMBFF(L1, LDim, HDim, labels)

Err=ErrorPrediction(labels, CE.F)

This shows the network definition equivalent to the original network shown but using the above macros. Much simpler to deal with, and understand. One new feature shown in this network definition is access to Macro variables. ErrorPrediction() needs to access the result of the FeedForward result before the CrossEntropyWithSoftmax() is applied to it. However the needed variable is local to the macro, but can still be accessed via “dot” syntax. The return value of the macro was CE, so to access the local F variable defined in the macro itself, CE.F can be used. In the single line version of macros, there are no user defined variable names, so this feature cannot be used.

# Optional Parameters

Optional Parameters are a feature that allows additional parameters to be specified on functions. While the optional parameters can be specified on any function or macro, they are limited to constant values and the underlying function must support the passed optional parameters, or there is no effect on the network. When used on a macro, the macro will have local variables defined that match the optional parameter name and value.

## Parameter initialization

One common use of these optional parameters is to define how parameters will be initialized:

B0=Parameter(HDim, init=zero)

W0=Parameter(HDim, SDim, init=uniform)

In this example the Bias matrix will be zero initialized, and the weight matrix will be initialized with uniform random numbers. Please consult the NDL Function reference to find which functions accept optional parameters

## Tagging special values

As an alternate to providing an array of special nodes that are used as features, labels, criteria, etc, optional parameters can be used. So instead of:

FeatureNodes=(features)

LabelNodes=(labels)

CriteriaNodes=(CrossEntropy)

EvalNodes=(ErrPredict)

OutputNodes=(Plus2)

The network can be defined as

# 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)

Which avoids adding elements to the node arrays, and instead sets the ‘tag’ optional parameter on the functions or macros that return the value that fits into a specified category. In this case, since the output node is actually computed inside a macro, we must specify it explicitly.

# NDL Reference

## Variables

Variables are defined in NDL when they appear on the left of an equal sign (‘=’). From that point on that variable name will be associated with the value it was assigned. Variables are immutable, and assigning new values to an existing variable is not supported.

Variable names may be any alphanumeric sequence that starts with a letter. The variables can contain a matrix or scalar value.

#### Reserved words

Any name that is also a function name is a reserved word and cannot be used for a variable. The special node names are also reserved and are as follows:

FeatureNodes

LabelNodes

CriteriaNodes

EvalNodes

OutputNodes

These may not be used as variable names.

#### Dot names

When it is necessary to access a variable that is defined in a macro (see Macros below), it can be accessed using dot-names. If the following macro is called from code:

L1 = FF(features, HDim, SDim)

And the macro is defined as follows:

FF(X1, W1, B1)

{

T=Times(W1,X1);

FF=Plus(T, B1);

}

If I want to access the result of the Times() function before the Plus happened, I can with the following variable:

L1.T

The variable name used in the script followed by a dot and the local name in the macro. This does requires the user to know the name used in the macro, so having all macro definitions available is important. Since macros can be nested, dot names can be several layers deep if necessary.

## Functions

Functions are called using function call syntax similar to most programming languages:

Times1=Times(W0, features)

The function name is followed by parenthesis which contains the comma separated parameter list. Each function returns a single value, which is identified by a variable.

## Macros

Macros are a combination of multiple Functions combined in a block. This can be done in a single-line nested fashion:

RFF(x1, w1, b1)=RectifiedLinear(Plus(Times(w1,x1),b1))

In this case the functions called will be evaluated from the innermost nested function call to the outermost.

The other method of defining macros uses a “programming block” style:

FF(X1, W1, B1)

{

T=Times(W1,X1);

FF=Plus(T, B1);

}

In this case the intermediate variables, which are local to the macro, can still be accessed from the outside using the dot syntax for variables.

## Optional Parameters

Many functions will have optional parameters that will change the behavior of the function. For example:

B0=Parameter(HDim, init=zero)

In this example the Bias vector will be zero initialized. The NDL Function reference will specify what optional parameters are accepted by each function

#### Tags

Tags are a special case of optional parameters, and are discussed in the Special Nodes section.

## Special nodes

Special nodes need to be identified for CNTK to automatically do back propagation updates of Learnable Parameters and identify inputs properly. These special nodes be specified in two different ways, the node arrays, or by use of special tags. If both methods are used the values are combined.

#### Node Arrays

CNTK supports multiple nodes for each type, so all these values are arrays. However, In many cases there will only be a single node for each node type. The array syntax (parenthesis) must be used when setting these special nodes, even if there is only one element. If more than one element is include, the entries are comma separated and surrounded by parenthesis. For example:

FeatureNodes=(features)

LabelNodes=(labels)

CriteriaNodes=(CrossEntropy)

EvalNodes=(ErrPredict)

OutputNodes=(Plus2)

#### Tags

A special optional parameter is a “tag”. These can be used as a shortcut way to identify special values in the network. For example features and labels can be tagged as such when the inputs are defined, as follows:

F1=Input(SDim, tag=feature)

L1=Input(LDim, tag=label)

The acceptable tag names correspond to the special node types and are as follows:

|  |  |
| --- | --- |
| Tag name | Meaning |
| feature | A feature input |
| label | A label input |
| criteria | criteria node, top level node |
| eval | evaluation node |
| Output | output node |

# NDL Functions

This section contains the currently implemented NDL functions. The CNTK is being expanded and additional functions will be available as development continues.

### Input, InputValue

Defines input data for the network. This defines the input that will be read from a datasource. The datasource information is specified in the configuration file separately, allowing the same network to be used with multiple datasets easily.

Input(rows, [cols=1])

InputValue(rows, [cols=1])

#### Parameters

rows – row dimension of the data.

cols – [optional] col dimension of the data. If this dimension is not specified, it is assumed to be 1

#### Notes

Input nodes are normally tagged with their intended purpose so the CNTK can use the inputs appropriately. The following tags may be used as optional parameters, and specify feature values, and label values respectively:

tag=feature

tag=label

### ImageInput

Defines image input data for the network. This defines the input that will be read from a datasource. The datasource information is specified in the configuration file separately, allowing the same network to be used with multiple datasets easily.

ImageInput(width, height, channels, [numImages=1])

#### Parameters

width – width of the image data.

height – height of the image data.

channels – number of channels in the image data (i.e. RGB would have 3 channels)

numImages – [optional] number of images in each sample, defaults to 1

#### Notes

Each data element is expected to be in 16-bit (single) or 32-bit (double) floating point format. The order of the data from least frequently changing to most frequently changing is Image, Col, Row, Channel.

Input nodes are normally tagged with their intended purpose so the CNTK can use the inputs appropriately. The following tags may be used as optional parameters, and specify feature values, and label values respectively:

tag=feature

tag=label

### Parameter, LearnableParameter

Defines a parameter in the network that will be trained. Normally used for weight and bias matrices/vectors.

Parameter(row, [cols])

LearnableParameter(rows, [cols])

#### Parameters

rows – number of rows in the parameter, this will normally be determined by the Input size, a hidden weight/bias matrix size, or an output size.

cols – (optional, defaults to 1) number of columns in the parameter data. This is often left at the default value to be determined by the minibatch size when processing the network.

#### Optional Parameters

ComputeGradient=[*true*,false] – Turns on (or off) automatic gradient calculation required for Stochastic Gradient Descent (SGD) training. Defaults to on.

InitValueScale=*number* – Initialization value for the input. Depending on the initialization technique this number is used to determine the range of the random numbers used for initialization. Defaults to 0.05 producing random numbers in a range of

Init = [*None, Zero*, *Uniform*, *Gaussian*] – Form of initialization for inputs

* None – No initialization is required, should only be used if the network will be initializing in some other way
* Zero – zero initialize the parameter matrix
* Uniform – Initializes the parameter matrix with random numbers based on the InitValueScale in the following range:
* Gaussian – Initializes the parameter matrix with random numbers using a Gaussian distribution in the range

### Sum

Calculate the sum of two matrices.

Sum(add1, add2)

#### Parameters

add1, add2 – matrix values, must be the same dimensions.

#### Returns

add1+add2, the element-wise matrix sum of the parameters. The result of the sum is stored in the add1 matrix (add1+=add2)

### Scale

Scale a matrix by a scalar value

Scale(scaleFactor, matrix)

#### Parameters

scaleFactor – floating point scalar scale factor

matrix - matrix values, must be the same dimensions.

#### Returns

, the element-wise product of the scaleFactor and matrix

### Times

Calculate the sum of two matrices.

Times(mult1, mult2)

#### Parameters

mult1, mult2 – matrix values, the mult1.rows must equal mult2.cols.

#### Returns

, the matrix product of the parameters

### Plus

Calculate the sum of two matrices.

Plus(add1, add2)

#### Parameters

add1, add2 – matrix values, must be the same dimensions.

#### Returns

add1+add2, the element-wise matrix sum of the parameters

### Minus

Calculate the difference of two matrices.

Minus(sub1, sub2)

#### Parameters

sub1, sub2 – matrix values, must be the same dimensions.

#### Returns

, the element-wise matrix difference of the parameters

### Negate

Negate the matrix.

Negate(matrix)

#### Parameters

matrix – matrix value.

#### Returns

, the element-wise negation of all elements of the matrix

### RectifiedLinear

Compute the RectifiedLinear operation on the matrix.

RectifiedLinear(matrix)

#### Parameters

matrix – matrix value.

#### Returns

, the element-wise rectified linear operation of all elements of the matrix

### Sigmoid

Compute the Sigmoid of the matrix.

Sigmoid(matrix)

#### Parameters

matrix – matrix value.

#### Returns

the element-wise sigmoid of all elements of the matrix

### Tanh

Compute the Hyperbolic Tangent of the matrix elements.

Tanh(matrix)

#### Parameters

matrix – matrix value.

#### Returns

the element-wise hyperbolic tangent of all elements of the matrix

### Log

Compute the Logarithm (base 10) of the matrix elements.

Log(matrix)

#### Parameters

matrix – matrix value.

#### Returns

the element-wise logarithm of all elements of the matrix

### Softmax

Compute the Softmax of the matrix.

Softmax(matrix)

#### Parameters

matrix – matrix value.

#### Returns

the softmax of the matrix

### SquareError

Compute the SquareError of the matrix.

SquareError(m1, m2)

#### Parameters

m1 – first matrix to compare.

m2 - second matrix to compare

#### Returns

the square error value of the matrix, returned in a 1x1 matrix

### CrossEntropyWithSoftmax, CEWithSM

Compute the Softmax of the matrix, compare against the ground truth labels and compute the CrossEntropy error matrix.

CrossEntropyWithSoftmax(labels, matrix)

CEWithSM(labels, matrix)

#### Parameters

labels – the ground truth labels

matrix – matrix value.

#### Returns

the CrossEntropy error matrix

#### Notes

This node will often be tagged as a “Criteria” node to allow the CNTK to identify the node producing the error matrix. To tag appropriate node(s) the following optional parameter should be added to the call(s):

tag=Criteria

### MatrixL1Reg, L1Reg

Compute the sum of the absolute value of the entire matrix.

MatrixL1Reg(matrix)

L1Reg(matrix)

#### Parameters

matrix – matrix to use in computation

#### Returns

the sum of the absolute value of the matrix elements, returned in a 1x1 matrix

### MatrixL2Reg, L2Reg

Compute the FrobeniusNorm of the matrix.

MatrixL2Reg(matrix)

L2Reg(matrix)

#### Parameters

matrix – matrix to compute the FrobeniusNorm on.

#### Returns

the FrobeniusNorm of the matrix, returned in a 1x1 matrix

### PerDimMeanVarNormalization, PerDimMVNorm

Compute the Mean-Variance Normalized Matrix

PerDimMeanVarNormalization(matrix, mean, invStdDev)

PerDimMVNorm(matrix, mean, invStdDev)

#### Parameters

matrix – matrix than needs to be normalized

mean – the mean for each sample index (same row dimensions as “matrix”)

invStdDev – 1/stddev for each sample index. (same row dimensions as “matrix”)

#### Returns

the mean variance normalized matrix

#### Notes

This function requires the Mean and InvStdDev to be already computed. They can either be loaded from a dataset, or computed in a pre-pass, before normalization is required.

### ErrorPrediction

Evaluate the accuracy of the current predictions made by the model. This is generally used to compute the training accuracy of a model. It finds the highest predicted probability from the model and compares it to the actual ground truth.

ErrorPrediction(labels, matrix)

#### Parameters

labels – the ground truth labels

matrix – matrix value.

#### Returns

the number of predicted values that do not match the labels in the current minibatch. Returns a 1x1 matrix

#### Notes

This node will often be tagged as an “Eval” node to allow the CNTK to print ongoing error statistics during training. To take appropriate node(s) the following optional parameter should be added to the call(s):

tag=Eval

### Dropout

Compute a new matrix with *dropoutRate* perecent set to zero. The values that are set to zero are randomly chosen. This is commonly used to prevent overfitting during the training process.

Dropout(matrix)

#### Parameters

matrix – source matrix

#### Returns

a new matrix with *dropoutRate* percent of the elements set to zero (dropped out).

#### Optional Parameters

dropoutRate – The percent (represented as a decimal 0.0-1.0) of values that will be dropped on each iteration.

### Mean

Compute the Per dim mean matrix for the entire dataset

Mean(matrix)

#### Parameters

matrix – source matrix

#### Returns

Where is the size of the entire dataset

#### Notes

This function is a pre-pass function, will only be called during a pre-pass through the entire dataset before the first training pass. This allows the Mean to be computed before it is required for Mean-Variance Normalization.

### Convolution, Convolve

Compute the convolution of an image input

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

#### Parameters

cvweight – convolution weight matrix, it has the dimensions of [outputChannels, kernelWidth \* kernelHeight \* inputChannels]

kernelWidth – width of the kernel

kernelHeight – height of the kernel

outputChannels – number of output channels

horizontalSubsample – subsamples in the horizontal direction

verticalSubsample – subsamples in the vertical direction

#### Optional Parameters

zeroPadding – [default = false] should the sides of the image be padded with zeros?

maxTempMemSizeInSamples – [default=0] maximum amount of memory (in samples) that should be reserved as temporary space

#### Returns

The convolved matrix according to the parameters passed

#### Notes

The input to this node must be an ImageInput(). This node automatically determines image size on input and output based on the size of the original input and which nodes the input has passed through. This function is often followed by another Convolution() or a MaxPooling() or AveragePooling() node.

### MaxPooling

Computes a new matrix by selecting the maximum value in the pooling window. This is used to reduce the dimensions of a matrix.

MaxPooling(matrix, windowWidth, windowHeight, stepW, stepH)

#### Parameters

matrix – input matrix

windowWidth – width of the pooling window

windowHeight – height of the pooling window

stepW – step used in the width direction

stepH – step used in the height direction

#### Returns

The dimension reduced matrix consisting of the maximum value within each pooling window.

#### Notes

This function is often associated with Convolution() operations.

### AveragePooling

Computes a new matrix by selecting the average value in the pooling window. This is used to reduce the dimensions of a matrix.

MaxPooling(matrix, windowWidth, windowHeight, stepW, stepH)

#### Parameters

matrix – input matrix

windowWidth – width of the pooling window

windowHeight – height of the pooling window

stepW – step used in the width direction

stepH – step used in the height direction

#### Returns

The dimension reduced matrix consisting of the maximum value within each pooling window.

#### Notes

This function is often associated with Convolution() operations.

### Delay

Delay node used in recurrent networks, allows creation of a loop in the convolutional network that will repeat a specified number of times.

Delay(rows, [cols], delayNode, delayTime=1, needGradient=true, defaultHiddenActivity=0.1)

#### Parameters

cvweight – convolution weight matrix, it has the dimensions of [outputChannels, kernelWidth \* kernelHeight \* inputChannels]

kernelWidth – width of the kernel

kernelHeight – height of the kernel

outputChannels – number of output channels

horizontalSubsample – subsamples in the horizontal direction

verticalSubsample – subsamples in the vertical direction

#### Optional Parameters

delayTime – [default = 1] the amount of delay that will be introduced (number of times the loop will happen)

needGradient – [default = true] does the gradient need to be computed for this node

defaultHiddenActivity – [default = 0.1] the numerical amount for the defaultHiddenActivity

#### Returns

The results of the completed Delay loop

#### Notes

This node is used in recurrent networks, where a delay is introduced to examine values from a previous time, such as the prior value (t-1). This has the affect of creating a loop in the computational network that will repeat delayTime number of iterations.