**Introduction to the Microsoft CNTK Machine Learning Tool**

James McCaffrey

The Microsoft CNTK (Computational Network Tool Kit) is a very powerful, command-line system that can create neural network prediction systems. Because CNTK was originally developed for internal use at Microsoft, the documentation is a bit intimidating. In this article I walk you through the process of installing CNTK, setting up a demo prediction problem, creating a neural network model, making a prediction, and interpreting the results.

The screenshot shown in Figure 1 gives you an idea of what CNTK is and a preview of where this article is headed. Although it's not visible in the image, the CNTK tool was run by entering the command:

> cntk.exe configFile=MakeModel.cntk makeMode=false

The image shows only the last part of the output messages. The configuration file with a .cntk extension contains information about input files and information about the design of a neural network to use. A prediction model was created and saved to disk.

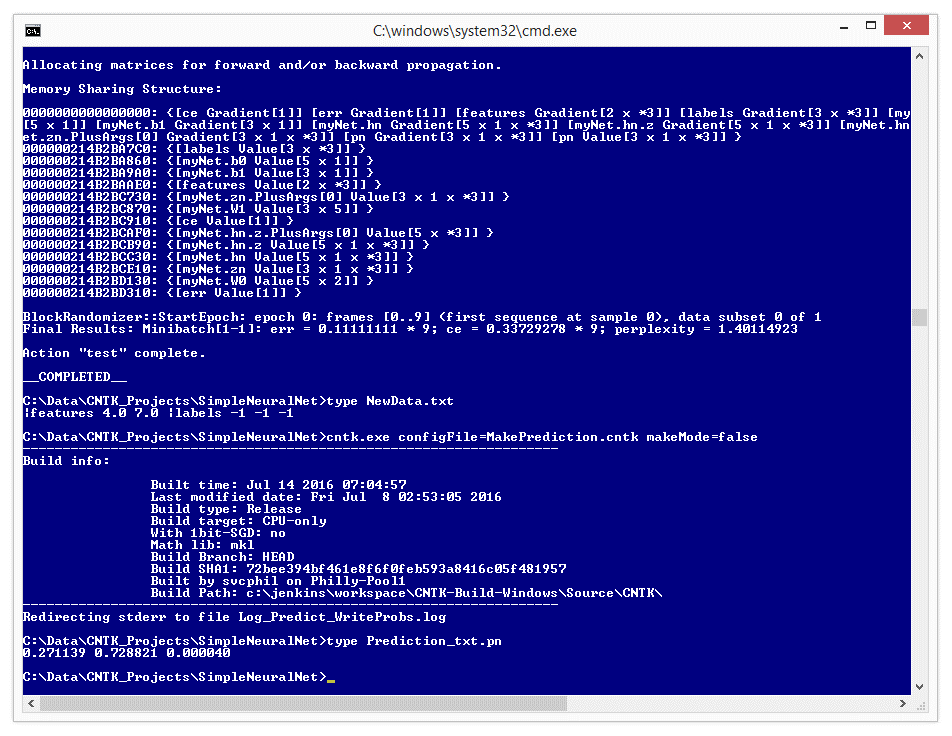


Figure 1 - CNTK in Action

After the prediction model was created it was used to make a prediction. The input values are in file NewData.txt and are (4.0, 7.0). The CNTK tool was used with a second configuration file named MakePrediction.cntk to compute a prediction. The prediction results were saved to file Prediction\_txt.pn and are (0.271139, 0.728821, 0.000040) which means that the predicted outcome is the second of three possible output values.

This article assumes you have familiarity working with command line programs and a rough idea of what neural networks are, but doesn't assume you are a machine learning (ML) expert or know anything about CNTK. All the code and data needed to run the demo program is presented in this article. You can also get the code and data from the download that accompanies this article.

**Setting up the Problem**

Imagine a problem where you want to predict a person's political leaning (conservative, moderate, liberal) from their age and annual income. This is called a classification problem. The idea is to take some data with known values and create a prediction model. Here, you can think of the model as a kind of a complex math function that accepts two numeric input values and which emits a value indicating one of three classes.

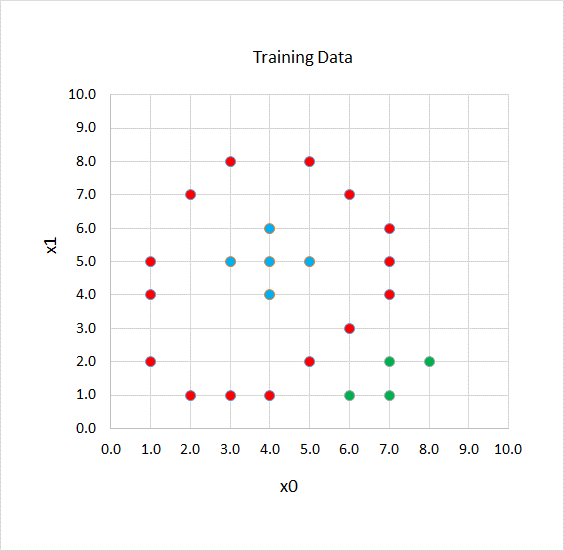


Figure 2 - Training Data

Now take a look at the graph in Figure 2. There are 24 data points. The two input variables, called features in ML terminology, are x0 and x1. The three colors indicate three different classes, sometimes called labels in ML terminology. Although a human being can quickly see a pattern, trying to create a prediction model for even this simple data set is very challenging for a computer system.



Figure 3 - Test Data

After CNTK uses the data shown in Figure 2 to create a neural network prediction model, the model will be applied to the nine test data points shown in Figure 3. As you'll see, CNTK will correctly predict eight out of the nine test cases. The test item at (8.0, 8.0) which is actually class "red" will be incorrectly predicted to be class "blue".

**Installing CNTK**

Installing CNTK is just a matter of downloading a .zip folder from GitHub to your machine and then extracting the files. The main CNTK portal site is located at https://github.com/Microsoft/CNTK. On that page you'll find a link to the current releases at https://github.com/Microsoft/CNTK/releases. One of the keys to the power of CNTK is that it can optionally use a GPU instead of your machine's CPU. The releases page gives you the choice to download binaries for a CPU-only version or a GPU plus CPU version.

For demonstration purposes, I recommend that you select the CPU-only version, even though you can direct the GPU+CPU version to use only CPU. Clicking on the associated link on the releases page will bring you to a page where you'll need to accept some licensing terms, and after you click the Accept button, you'll get a dialog where you can save a file with name something like CNTK-1-6-Windows-64bit-CPU-Only.zip (where there version number could be different of course) to your machine.

Download the .zip file to your Desktop or any convenient directory. Then extract all files directly to the C: drive (most common) or to the C:\Program Files directory.

The extracted download will have a single root directory named cntk. That root directory will contain several directories including another directory also named cntk which contains all the binaries, including the key cntk.exe file. To finish the installation process you should add the path to the cntk.exe file into your system PATH environment variable (typically C:\cntk\cntk).

**Creating the Data Files**

To create and run a CNTK project you need a configuration file that has a .cntk extension, and at least one file that contains training data. Most CNTK projects will have a file of test data too. Additionally, the CNTK tool will create several output files when a project is run.

There are many ways to organize the files used with a CNTK project. I recommend that you create a project root directory that holds your data files and the .cntk configuration file, and run CNTK from the project root directory.

For the demo, I had an existing C:\Data directory on my machine. Inside that directory I created a new subdirectory named CNTK\_Projects. And inside that directory I created a subdirectory named SimpleNeuralNet to act as the demo project root directory to hold my .cntk file and a training data file and a test data file.

The CNTK system can work with several different types of data files. The demo uses simple text files. Open an instance of Notepad and copy-paste the first 24 data items from Figure 4, or manually create the data using the information in Figure 2, then save the file as TrainData.txt in the SimpleNeuralNet directory.

The training data looks like:

|features 1.0 5.0 |labels 1 0 0

|features 1.0 2.0 |labels 1 0 0

. . .

|features 7.0 2.0 |labels 0 0 1

The "|features" tag indicates input values. Values can be delimited using a blank space or the tab character (the demo data uses blank spaces). Neural networks only understand numeric values so class labels like "red" and "blue" must be encoded as numbers. Neural network classifier models use what's called 1-of-N encoding. For three possible class labels, you'd use 1 0 0 for the first class ("red" in the demo), 0 1 0 for the second class ("blue"), and 0 0 1 for the third class ("green"). It's up to you to keep track of how each label value is encoded.

|features 1.0 5.0 |labels 1 0 0

|features 1.0 2.0 |labels 1 0 0

|features 3.0 8.0 |labels 1 0 0

|features 4.0 1.0 |labels 1 0 0

|features 5.0 8.0 |labels 1 0 0

|features 6.0 3.0 |labels 1 0 0

|features 7.0 5.0 |labels 1 0 0

|features 7.0 6.0 |labels 1 0 0

|features 1.0 4.0 |labels 1 0 0

|features 2.0 7.0 |labels 1 0 0

|features 2.0 1.0 |labels 1 0 0

|features 3.0 1.0 |labels 1 0 0

|features 5.0 2.0 |labels 1 0 0

|features 6.0 7.0 |labels 1 0 0

|features 7.0 4.0 |labels 1 0 0

|features 3.0 5.0 |labels 0 1 0

|features 4.0 4.0 |labels 0 1 0

|features 5.0 5.0 |labels 0 1 0

|features 4.0 6.0 |labels 0 1 0

|features 4.0 5.0 |labels 0 1 0

|features 6.0 1.0 |labels 0 0 1

|features 7.0 1.0 |labels 0 0 1

|features 8.0 2.0 |labels 0 0 1

|features 7.0 2.0 |labels 0 0 1

|features 1.0 1.0 |labels 1 0 0

|features 3.0 9.0 |labels 1 0 0

|features 8.0 8.0 |labels 1 0 0

|features 3.0 4.0 |labels 0 1 0

|features 5.0 6.0 |labels 0 1 0

|features 3.0 6.0 |labels 0 1 0

|features 8.0 3.0 |labels 0 0 1

|features 8.0 1.0 |labels 0 0 1

|features 9.0 2.0 |labels 0 0 1

Figure 4 - The Training and Test Data

In a non-demo scenario you might have to worry about input data normalization. In situations where the input values differ greatly in magnitude, you'll get better results if you scale data so that all magnitudes are roughly the same. For example, suppose the input data was a person's age (like 32.0) and annual income (like $48,500.00). You could preprocess the data by dividing all age values by 10 and all income values by 10,000 giving normalized input values like (3.20, 4.85). The three most common forms of input data normalization are called z-score normalization, min-max normalization, and order of magnitude normalization.

After you've created and saved the TrainData.txt file, create and save the nine-item TestData.txt file in the SimpleNeuralNet directory.

**Understanding Neural Network Input and Output**

In order to understand how to use CNTK you have to have a basic understanding of what a neural network is, how it computes output values, and how to interpret output values. Take a look at the diagram in Figure 5. The diagram shows the neural network that corresponds to the demo problem.

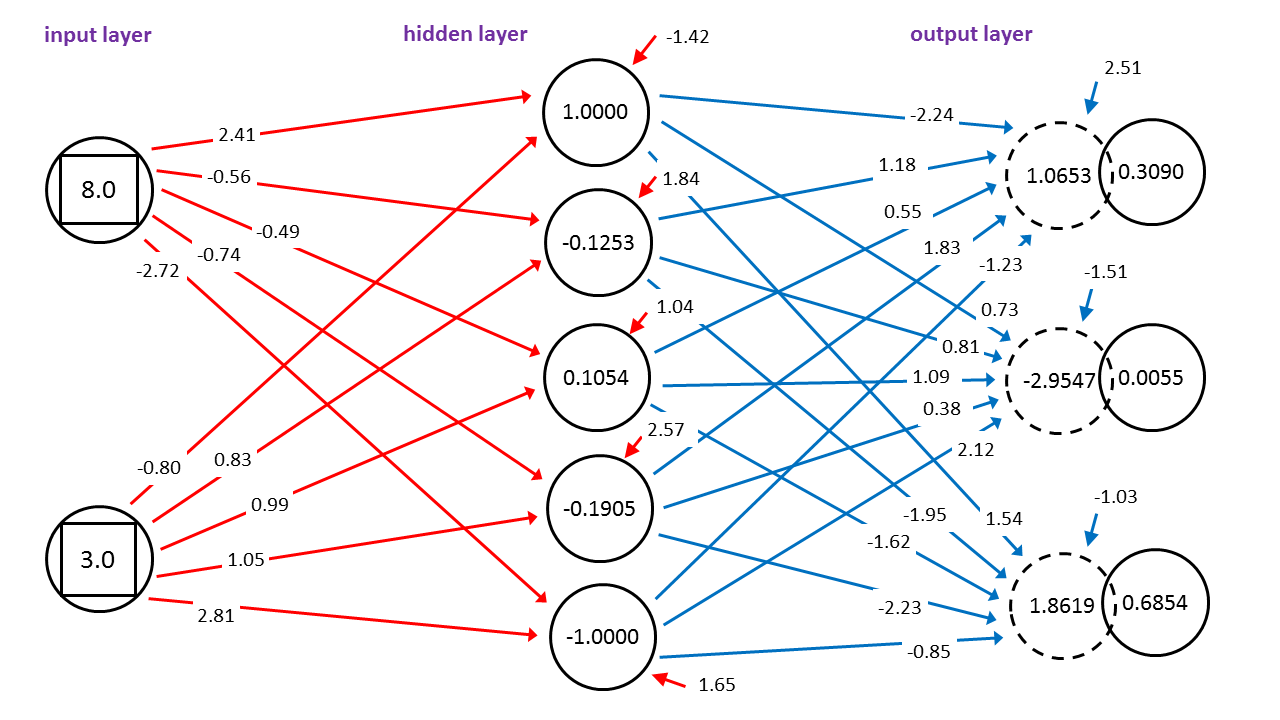


Figure 5 - Neural Network Input and Output

There are two input nodes that hold values (8.0, 3.0) and there are three output nodes that have values (0.3090, 0.0055, 0.6854). The network has five hidden processing nodes.

Each input node has lines connecting it to all hidden nodes. And each hidden node has lines connecting it to all output nodes. These lines represent numeric constants called weights. Nodes are identified using 0-based indexing so the top-most nodes are [0]. So, the weight from input[0] to hidden[0] is 2.41 and the weight from hidden[4] to output[2] is -0.85.

Each hidden node and each output node has an additional arrow. These are called the bias values. The bias for hidden[0] is -1.42 and the bias for output[2] is -1.03.

The input-output calculations are best explained by an example. First, hidden node values are computed. The middle hidden node, hidden[2] has value 0.1054 and is calculated by summing the products of all connected inputs and their associated weights plus the bias value, and then taking the hyperbolic tangent (abbreviated tanh) of that sum:

hidden[2] = tanh( (8.0)(-0.49) + (3.0)(0.99) + 1.04) )

= tanh( -3.92 + 2.98 + 1.04 )

= tanh( 0.1058 )

= 0.1054

The tanh function is called the hidden layer activation function. Neural networks can use one of several different activation functions. In addition to tanh, the other two most common hidden layer activation functions are logistic sigmoid (usually shortened to just "sigmoid") and rectified linear.

After all the hidden node values are calculated, the next step is to calculate the output nodes. First the sum of the products of connected hidden nodes and their associated weights plus the bias value is computed. For example, output[0] for this step is 1.0653 calculated as:

output[0] = (1.0000)(-2.24) + (-0.1253)(1.18) +

(0.1054)(0.55) + (-0.1905)(1.83) +

(-1.0000)(-1.23) + 2.51

= (-2.2400) + (-0.1478) +

(0.0580) + (-0.3486) +

(1.2300) + 2.51

= 1.0653

In the same way, output[1] is calculated to be -2.9547 and output[2] is 1.8619.

Next, the three preliminary output values are scaled so that they sum to 1.0 using what's called the softmax function:

output[0] = e^1.0653 / (e^1.0653 + e^-2.9547 + e^1.8619)

= 0.3090

output[1] = e^-2.9547 / (e^1.0653 + e^-2.9547 + e^1.8619)

= 0.0055

output[2] = e^1.8619 / (e^1.0653 + e^-2.9547 + e^1.8619)

= 0.6854

These three values are interpreted as probabilities. So, for inputs of (8.0, 3.0) and the given weights and biases values, the outputs are (0.3090, 0.0055, 0.6854). The highest probability is the third value so the prediction is the third class, "green" in this case.

Another way of interpreting the output values is to map them so that the highest probability is 1 and all others are 0. For this example you'd get (0, 0, 1) which maps to the encoded value of "green".

The processing of determining the values of the weights and the biases is called training the network, and that's what CNTK does.

**Creating the Configuration File**

The screenshot in Figure 1 gives you an idea of how CNTK is used. In an ordinary command shell, I navigated to the project root directory at C:\Data\SimpleNeuralNet. The project root directory contains files TrainData.txt and TestData.txt and a MakeModel.cntk configuration file. The CNTK tool was invoked by executing the command:

> cntk.exe configFile=MakeModel.cntk makeMode=false

Recall that the system PATH variable knows the location of the cntk.exe program so it doesn't have to be fully qualified. The .cntk configuration file has a lot of information. The makeMode=false means to run the program and overwrite any previous results. The CNTK command line arguments are not case sensitive.

The complete listing for the configuration file is presented in Figure 6. The overall structure of the configuration file is:

# MakeModel.cntk

command=Train:WriteProbs:DumpWeights:Test

# system parameters go here

Train = [

action="train"

BrainScriptNetworkBuilder = [

# define network here

]

]

Test = [

# training commands here

]

WriteProbs = [

# output commands here

]

DumpWeights = [

# output commands here

]

You can name a CNTK configuration file however you wish, but using a .cntk extension is standard practice. You can use the '#' character or "//" token for comments. Comments do not span lines. At the top of the configuration file you give a colon-delimited list of modules to run, four in this case:

command=Train:WriteProbs:DumpWeights:Test

Notice that the order in which modules are executed does not have to match the order in which they're defined. Modules names (Train, WriteProbs, DumpWeights, Test in this example) are not keywords so you can name modules as you wish. Notice the Train module has an instruction action="train". Here both words are keywords.

# MakeModel.cntk

command=Train:WriteProbs:DumpWeights:Test

modelPath = "Model\SimpleNet.snn"

deviceId = -1

dimension = 2

labelDimension = 3

precision = "float"

# =====

Train = [

action="train"

# network description

BrainScriptNetworkBuilder = [

FDim = $dimension$

HDim = 5

LDim = $labelDimension$

# define the neural network

neuralDef (ftrs) = [

W0 = Parameter (HDim, FDim)

b0 = Parameter (HDim, 1)

W1 = Parameter (LDim, HDim)

b1 = Parameter (LDim, 1)

hn = Tanh (W0 \* ftrs + b0)

zn = W1 \* hn + b1

].zn

# specify inputs

features = Input (FDim)

labels = Input (LDim)

# create network

myNet = neuralDef (features)

# define training criteria and output(s)

ce = CrossEntropyWithSoftmax (labels, myNet)

err = ErrorPrediction (labels, myNet)

pn = Softmax (myNet)

# connect to the NDL system.

featureNodes = (features)

inputNodes = (labels)

criterionNodes = (ce)

evaluationNodes = (err)

outputNodes = (pn)

]

# stochastic gradient descent

SGD = [

epochSize = 0

minibatchSize = 1

learningRatesPerSample = 0.04

maxEpochs = 500

momentumPerMB = 0.90

]

# configuration for reading data

reader = [

readerType = "CNTKTextFormatReader"

file = "TrainData.txt"

input = [

features = [

dim = $dimension$

format = "dense"

]

labels = [

dim = $labelDimension$

format = "dense"

]

]

]

]

# test

Test = [

action = "test"

reader = [

readerType = "CNTKTextFormatReader"

file="TestData.txt"

randomize = "false"

input = [

features = [

dim = $dimension$

format = "dense"

]

labels = [

dim = $labelDimension$

format = "dense"

]

]

]

]

# log the output node values

WriteProbs = [

action="write"

reader=[

readerType="CNTKTextFormatReader"

file="TestData.txt"

input = [

features = [

dim = $dimension$

format = "dense"

]

labels = [

dim = $labelDimension$

format = "dense"

]

]

]

outputPath = "TestProbs\_txt"

]

# dump weight and bias values

DumpWeights = [

action = "dumpNode"

printValues = "true"

]

Figure 6 - The Training Configuration File

The Train module uses a training data file to create the prediction model, so most CNTK configuration files will have a module, usually named Train, that contains an action="train" command. The Train module will write the resulting model information to disk in a binary format.

The Test module is optional but is usually included when creating a model. The Test module will use the newly created prediction model to evaluate the overall prediction accuracy and prediction error on the training data.

The WriteProbs module is optional. The module will write the actual prediction values for the test data items to a text file in the project root directory. This allows you to see exactly which test cases were correctly predicted and which were not.

The DumpWeights module will write a text file that contains the neural network weights and biases that define prediction model. You can use this information to uncover trouble spots, and to make predictions on new, previously unseen data.

**System Parameters**

The MakeModel.cntk configuration file sets up five system parameters:

modelPath = "Model\SimpleNet.snn"

deviceId = -1

dimension = 2

labelDimension = 3

precision = "float"

The modelPath variable specifies where to put the resulting binary model and what to call the model. Here "snn" stands for simple neural network but you can use any extension. The deviceId variable tells CNTK whether to use the CPU (-1) or the GPU (0).

The dimension variable specifies the number of values there are in an input vector. The labelDimension specifies the number of possible output values there are. The precision variable can take values of "float" or "double". In most cases "float" is preferable because it makes training much faster than "double".

**The Training Module**

The demo Train module in the configuration file has three major sub-sections: "BrainScriptNetworkBuilder", "SGD", and "reader". These sub-sections define the neural network architecture, how to train the network, and how to read training data.

The training module definition begins as:

Train = [

action="train"

BrainScriptNetworkBuilder = [

FDim = $dimension$

HDim = 5

LDim = $labelDimension$

. . .

The BrainScriptNetworkBuilder section of the Train module uses a special scripting language called BrainScript. Variables FDim, HDim, and LDim hold the number of features, number of hidden nodes, and number of label nodes for the neural network. These names are not required so you could use names like NumInput, NumHidden, and NumOutput if you wish. The number of input and output nodes are determined by the problem data but the number of hidden nodes is a free parameter and must be determined by trial and error. The '$' token is a substitution operator. The Train module definition continues with:

neuralDef (ftrs) = [

W0 = Parameter (HDim, FDim)

b0 = Parameter (HDim, 1)

W1 = Parameter (LDim, HDim)

b1 = Parameter (LDim, 1)

hn = Tanh (W0 \* ftrs + b0)

zn = W1 \* hn + b1

].zn

. . .

This is a BrainScript function. Variable W0 is a matrix that holds the input-to-hidden weights. The Parameter function means "construct a matrix". Variable b0 holds the hidden node bias values. All calculations in BrainScript are performed on matrices so b0 is a matrix with one column rather than an array.

Variables W1 and b1 hold the hidden-to-output weights and the output node bias values. The values of the hidden nodes are calculated into a variable named hn using a sum of products and the tanh function as explained earlier. Variable zn holds the pre-softmax output values. The closing bracket-dot-variable notation is how a BrainScript function returns a value. The Train definition continues with:

features = Input (FDim)

labels = Input (LDim)

myNet = neuralDef (features)

. . .

Here the input features and output labels are defined. Variable names "features" and "labels" are not keywords but they must match the strings used in the training and test data files. The neural network is created by calling the neuralDef function. Next the module defines information that will be used during training:

ce = CrossEntropyWithSoftmax (labels, myNet)

err = ErrorPrediction (labels, myNet)

pn = Softmax (myNet)

. . .

The CrossEntropyWithSoftmax function specifies that cross entropy error should be used when calculating how close calculated output values are to actual output values in the training data. Cross entropy error is the standard metric but squared error is an alternative.

The ErrorPrediction function instructs CNTK to compute and display the prediction model accuracy (percentage correct predictions on the training data) and cross entropy error and perplexity, which are measures of error between calculated outputs and actual outputs.

The Softmax function instructs CNTK to normalize computed output values so that they sum to 1.0 and can be interpreted as probabilities. For a neural network classifier, Softmax is used except in extremely rare situations. The training module definition concludes with:

. . .

featureNodes = (features)

inputNodes = (labels)

criterionNodes = (ce)

evaluationNodes = (err)

outputNodes = (pn)

]

Here the required system variables of featureNodes, inputNodes, criterionNodes, and outputNodes, and the optional evaluationNodes, are associated with the user-defined variables.

The SGD ("stochastic gradient descent") sub-section defines how CNTK will train the neural network. In the context of a neural network, SGD is usually called back-propagation. The sub-section definition is:

SGD = [

epochSize = 0

minibatchSize = 1

learningRatesPerSample = 0.04

maxEpochs = 500

momentumPerMB = 0.90

]

The epochSize variable specifies how much of the training data to use. A special value of 0 means to use all available training data. The minibatchSize variable specifies how much of the training data to process in each training iteration. A value of 1 means update the weights and biases after each training item is processed. This is often called "online" or "stochastic" training.

If the value of minibatchSize is set to the number of training items (24 in the case of the demo) then all 24 items would be processed, the results aggregated, and only then would the weights and biases be updated. This is sometimes called "full-batch" or "batch" training. Using a value between 1 and the training set size is called "mini-batch" training.

The learningRatesPerSample variable specifies how much to adjust the weights and biases on each iteration. The value of the learning rate, along with the other parameters in the SGD sub-section must be determined by trial and error. Neural networks are typically extremely sensitive to the value of the learning rate -- for example, using 0.04 might give you a very accurate prediction system but using 0.039 or 0.041 could give you a very poor system.

The maxEpochs variable specifies how many iterations to perform for training. Too small a value will result in a poor model ("model under-fit") but too many iterations will over-fit the training data. This leads to a model that predicts the training data very well but one that predicts new data very poorly.

The momentumPerMB (momentum per mini-batch, not per megabyte as you might assume) is a factor that increases or decreases the amount that weights and biases are updated. Just like the learning rate, a momentum value must be determined by trial and error, and neural network training is typically extremely sensitive to the value of momentum. The value of 0.90 used by the demo is the default value so the momentumPerMB parameter could have been omitted.

The training module of the demo configuration file concludes by setting the values of the parameters of the reader sub-section:

. . .

reader = [

readerType = "CNTKTextFormatReader"

file = "TrainData.txt"

input = [

features = [ dim = $dimension$; format = "dense" ]

labels = [ dim = $labelDimension$; format = "dense" ]

]

]

] # end Train

The CNTK tool has many different types of readers for use with different types of input data. The meaning of the parameters in the demo reader sub-section should be clear to you. Here I put multiple statement on a single line by using the semicolon delimiter.

**The Test Module**

To recap to this point, the MakeModel.cntk configuration file has some global system parameters (such as modelPath) plus four modules: Train, Test, WriteProbs, DumpWeights. The Train module has three sub-sections: BrainScriptNetworkBuilder, SGD, reader.

The Test module is mercifully very simple:

Test = [

action = "test"

reader = [

readerType="CNTKTextFormatReader"

file = "TestData.txt"

randomize = "false"

input = [

features = [ dim = $dimension$

format = "dense" ]

labels = [ dim = $labelDimension$

format = "dense" ]

]

]

]

The reader sub-section of the test module should match the reader sub-section of the training module except for the file parameter value and the addition of the randomize parameter. When training with SGD, it's extremely important that the data items be processed in random order, and true is the default value for randomize. But when walking through the test data there's no need to randomize the order of the data items.

The test module emits one accuracy metric and two error metrics to the shell. If you refer back to the screenshot in Figure 1 just before the "Action test complete" message, you'll see:

err = 0.11111111 \* 9

ce = 0.33729280 \* 9

perplexity = 1.40114927

The err = 0.1111 \* 9 means that 11% of the 9 test data items were incorrectly predicted using the model. In other words, 8 out of 9 test items were correctly predicted. The training output does not however tell you which data items were correctly and incorrectly predicted.

The ce = 0.3372 \* 9 means that the average cross entropy error is 0.3372. For this introduction to CNTK, just think of cross entropy as an error term so smaller values are better.

The perplexity = 1.4011 is a minor metric. You can think of perplexity as a measure of how strong the predictions are, where smaller values are better. For example, for three possible output values as in the demo, if the prediction is (0.33, 0.33, 0.33) you don't have a strong prediction at all. The perplexity in this case would be 3.0 which is a maximum for three output values.

**The WriteProbs Module**

The third module in the demo CNTK configuration file is WriteProbs. This module is optional but very useful because it gives you additional information about the predictions made on the test data. The module is defined:

WriteProbs = [

action="write"

reader=[

readerType="CNTKTextFormatReader"

file="TestData.txt"

input = [

features = [ dim = $dimension$

format = "dense" ]

labels = [ dim = $labelDimension$

format = "dense" ]

]

]

outputPath = "TestProbs\_txt"

]

The WriteProbs module is the same as the test module except for three changes. First, the action parameter is set to "write" instead of "test". Second, the randomize parameter has been removed (because false is the default). Third, an outputPath parameter has been added.

When the WriteProbs module iexecutes, it will write the exact output values for the test data to the specified file. In this case the file name will have ".pn" appended because that was the variable name used for the output nodes in the training module.

For the nine demo test items, the contents of file TestProbs\_txt.pn are:

0.837386 0.162606 0.000008

0.990331 0.009669 0.000000

0.275697 0.724260 0.000042

0.271172 0.728788 0.000040

0.264680 0.735279 0.000041

0.427661 0.572313 0.000026

0.309024 0.005548 0.685428

0.000134 0.000006 0.999860

0.000190 0.000008 0.999801

The first three probability vectors go with the first three test items which map to correct output of (1, 0, 0) so the first two test items were predicted correctly. But the third probability vector of (0.27, 0.74, 0.00) maps to (0, 1, 0) so it is an incorrect prediction.

The next three probability vectors go with test items that have output (0, 1, 0) so all three predictions are correct. Similarly, the last three probability vectors go with (0, 0, 1) and so they're also correct predictions.

To recap, the Test module will emit accuracy and error metrics to the shell but not tell you which individual test items are correct or give you their error. The WriteProbs module writes exact output values to file and you can use them to determine which test items are incorrectly predicted.

**The DumpWeights Module**

The last of the four modules in the demo configuration file is DumpWeights. It is defined as:

DumpWeights = [

action = "dumpNode"

printValues = "true"

]

When executed, this module will save the trained model's weights and bias values to file. By default, the file name will be the same as the binary model (SimpleNet.snn in the demo) with a " .\_\_AllNodes\_\_.txt" appended, and the file will be saved in the directory specified by the modelPath parameter ("Model" in the demo).

After running the MakeModel.cntk demo, if you open a file explorer and point it to directory \SimpleNeuralNet\Model you'll see 503 files:

SimpleNet.snn

SimpleNet.snn.\_\_AllNodes\_\_.txt

SimpleNet.snn.0

. . .

SimpleNet.snn.499

SimpleNet.ckp

The SimpleNet.snn is the trained model saved in binary format for use by CNTK. The 500 files that have names which end with a digit, and the one file that ends with a ".ckp" extension are binary checkpoint files. The idea here is that training a complex neural network can take hours or even days. Recall that the demo set a maxEpochs parameter to 500. The CNTK tool saves training information periodically so in case of a system failure, you don't have to restart training from scratch.

The first half of the contents of the AllNodes\_\_.txt file for the demo (with a few lines removed) is:

myNet.b0=LearnableParameter [5,1]

-1.42185283

1.84464693

1.04422486

2.57946277

1.65035748

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

myNet.b1=LearnableParameter [3,1]

2.51937032

-1.5136646

-1.03768802

These are the values of the hidden node biases (b0) and the output node biases (b1). If you refer back to the neural network diagram in Figure 4, you'll see where these values are (truncated to two decimals). The second half of the AllNodes\_\_.txt file looks like:

myNet.W0=LearnableParameter [5,2]

2.41520381 -0.806418538

-0.561291218 0.839902222

-0.490522146 0.995252371

-0.740959883 1.05180109

-2.72802472 2.81985259

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

myNet.W1=LearnableParameter [3,5]

-2.246624 1.186315 0.557211 1.837152 -1.232379

0.739416 0.814771 1.095480 0.386835 2.120146

1.549207 -1.959648 -1.627967 -2.235190 -0.850726

Recall that the demo network has two input values, five hidden nodes and three output nodes. Therefore there are 2 \* 5 = 10 input-to-hidden weights in W0, and there are 5 \* 3 = 15 hidden-to-output weights in W1.

**Making a Prediction**

Once you have a trained model, you can use it to make a prediction. One way to do this is to use the CNTK tool with an "eval" action module. The demo takes this approach. First, a new set of data with a single item is created and saved as file NewData.txt:

|features 4.0 7.0 |labels -1 -1 -1

Because this is new data, the output labels use dummy -1 values. Next, I created a configuration file named MakePrediction.cntk with two modules named Predict and WriteProbs. The complete file is presented in Figure 7.

# MakePrediction.cntk

stderr = "Log" # write all messages to file

command=Predict:WriteProbs

modelPath = "Model\SimpleNet.snn" # where to find model

deviceId = -1

dimension = 2

labelDimension = 3

precision = "float"

Predict = [

action = "eval"

reader = [

readerType="CNTKTextFormatReader"

file="NewData.txt"

input = [

features = [ dim = $dimension$; format = "dense" ]

labels = [ dim = $labelDimension$; format = "dense" ]

]

]

]

WriteProbs = [

action="write"

reader=[

readerType="CNTKTextFormatReader"

file="NewData.txt"

input = [

features = [ dim = $dimension$; format = "dense" ]

labels = [ dim = $labelDimension$; format = "dense" ]

]

]

outputPath = "Prediction\_txt" # dump with .pn extension

]

Figure 7 - Making a Prediction

When run, the output probabilities are saved in a file named Prediction\_txt.pn which has contents:

0.271139 0.728821 0.000040

This maps to output (0, 1, 0) which is "blue". If you look at the training data in Figure 2, you can see that (4.0, 7.0) could easily be either "red" (1, 0, 0) or "blue" (0, 1, 0).

Two alternative techniques for using a trained model are to use a C# program with the CNTK model evaluation library, or to use a custom Python script that uses the model weights and bias values directly.

**Wrapping Up**

To the best of my knowledge, CNTK is the most powerful neural network system that is generally available to developers. This article has covered only a very small part of what CNTK can do, but it should be enough to get you up and running with simple neural networks and allow you to understand the documentation. The real power of CNTK comes from working with deep neural networks -- networks that have two or more hidden layers and possibly complicated connections between nodes.

Thanks to the following Microsoft Research technical experts who reviewed this article: Adam Eversole and John Krumm.