Python Wrapper for CNTK 1.5 Documentation

Release 1.5

Microsoft

August 12, 2016

Contents

CNTK, the Computational Network Toolkit, is a system for describing, training, and executing computational networks, a unified framework for describing arbitrary learning machines, such as deep neural networks (DNNs), convolutional neural networks (CNNs), recurrent neural networks (RNNs), long short term memory (LSTM), logistic regression, and maximum entropy model. CNTK is an implementation of computational networks that supports both CPU and GPU.

This page describes the Python Wrapper for CNTK version 1.5. This is an ongoing effort to expose such an API to the CNTK system, thus enabling the use of higher-level tools such as IDEs to facilitate the definition of computational networks, to execute them on sample data in real time.

CHAPTER 1

Getting started

1.1 Installation

This page will guide you through the following three required steps:

1. Make sure that all Python requirements are met
2. Build and install CNTK
3. Install the Python API and set it up

1.1.1 Requirements

You will need the following Python packages:

Python 2.7+ or 3.3+

NumPy 1.10

Scipy 0.17

On Linux a simple `pip install` should suffice. On Windows, you will get everything you need from Anaconda.

1.1.2 Installing CNTK

Please follow the instructions on CNTK's GitHub page. After you have built the CNTK binary, find the build location. It will be something like `<cntkpath>/x64/Release/cntk`. You will need this for the next step.

1.1.3 Installing the Python module

1. Go to `<cntkpath>/contrib/Python` and run `python setup.py install`
2. Set up the environment variable `CNTK_EXECUTABLE_PATH` to point to the CNTK executable. Make sure the executable is also included
3. Enjoy Python's ease of use with CNTK's speed:

```
>>> import cntk as C
>>> C.__version__
1.5
>>> with C.LocalExecutionContext('demo', clean_up=False) as ctx:
...     a = C.constant([[1,2], [3,4]])
```

```
    ...     i = C.input_numpy([[[10,20], [30, 40]]])
    ...     print(ctx.eval(a + i))
    [[11.0, 22.0], [33.0, 44.0]]
```

In this case, we have set `clean_up=False` so that you can now peek into the folder `_cntk_demo` and see what has been created under the hood for you.

Most likely, you will find issues or rough edges. Please help us improve CNTK by posting any problems to https://github.com/Microsoft/CNTK/issues. Thanks!

1.2 Overview and first run

CNTK is a powerful toolkit appropriate for everything from complex deep learning research to distributed production environment serving of learned models. It is also great for learning, however, and we will start with a basic regression example to get comfortable with the API. Then, we will look at an area where CNTK shines: working with sequences, where we will demonstrate state-of-the-art sequence classification with an LSTM (long short term memory network).

1.2.1 First basic use

The CNTK Python API allows users to easily define a computational network, define the data that will pass through the network, setup how learning should be performed, and finally, train and test the network. Here we will go through a simple example of using the CNTK Python API to learn to separate data into two classes. Following the code, some basic CNTK concepts will be explained:

```
import cntk as C
import numpy as np

# 500 samples, 250-dimensional data
N = 500
d = 250

# create synthetic data using numpy
X = np.random.randn(N, d)
Y = np.random.randint(size=(N, 1), low=0, high=2)
Y = np.hstack((Y, 1-Y))

# set up the training data for CNTK
x = C.input_numpy(X)
y = C.input_numpy(Y)

# define our network parameters: a weight tensor and a bias
W = C.parameter((d, 2))
b = C.parameter((1, 2))

# create a dense 'layer' by multiplying the weight tensor and
# the features and adding the bias
out = C.times(x, W) + b

# setup the criterion node using cross entropy with softmax
ce = C.cross_entropy_with_softmax(y, out, name='loss')
ce.tag = 'criterion'

# define our SGD parameters and train!
my_sgd = C.SGDParams(epoch_size=0, minibatch_size=25, learning_rates_per_mb=0.1, max_epochs=3)
with C.LocalExecutionContext('logreg') as ctx:
```

```
    ctx.train(root_nodes=[ce], training_params=my_sgd)
    print(ctx.test(root_nodes=[ce]))
```

In the example above, we first create a synthetic data set of 500 samples, each with a 2-dimensional one-hot vector representing 0 (`[1 0]`) or 1 (`[0 1]`). We then begin describing the topology of our network by setting up the data inputs. This is typically done using the `cntk.reader.CNTKTextFormatReader` by reading data in from a file, but for interactive experimentation and small examples we can use the `input_numpy` reader to access numpy data.

Next, we define our network. In this case it's a simple 1-layer network with a weight tensor and a bias. We multiply our data *x* with the weight tensor *W* and add the bias *b*. We then input the model prediction into the `cntk.ops.cross_entropy_with_softmax()` node. This node first runs the data through a *softmax* to get probabilities for each class. Then the Cross Entropy loss function is applied. We tag the node *ce* with "criterion" so that CNTK knows it's a node from which the learning can start flowing back through the network.

Finally, we define our learning algorithm. In this case we use Stochastic Gradient Descent (SGD) and pass in some basic parameters. First, *epoch_size* allows different amounts of data per epoch. When we set it to 0, SGD looks at all of the training data in each epoch. Next, *minibatch_size* is the number of samples to look at for each minibatch; *learning_rates_per_mb* is the learning rate that SGD will use when the parameters are updated at the end of each minibatch; and *max_epochs* is the maximum number of epochs to train for.

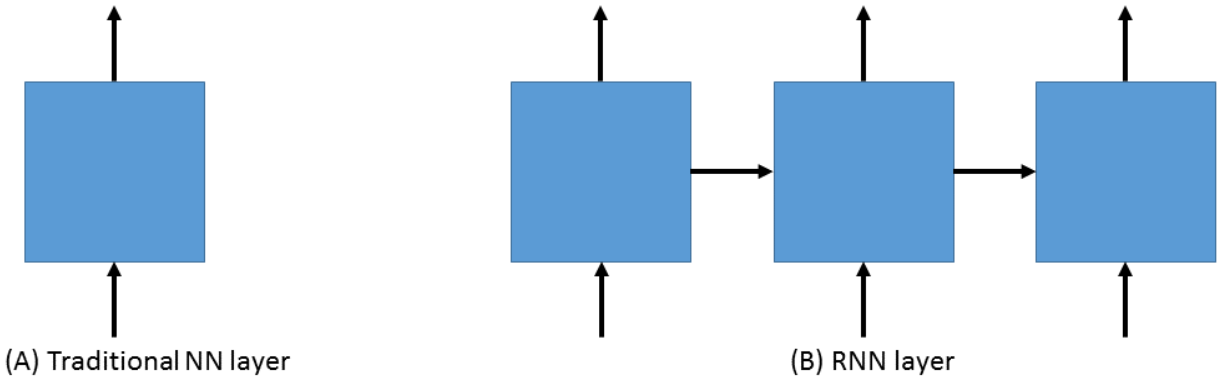
The last step is to set up an execution context. An execution context can be either *Local* or *Deferred*. In the former case, as we use here, the methods (such as training and testing the network) are done locally and immediately so that the result is returned interactively to python. With a *Deferred* context, the methods simply set up a configuration file that can be used with CNTK at a later date. Here, with the local execution context, we train the network by passing in the root node and the optimizer we are using, and finally, we test its performance. Here is the output of the above example:

```
{'SamplesSeen': 500, 'Perplexity': 1.1140191, 'loss': 0.10797427}
```

Now that we've seen some of the basics of setting up and training a network using the CNTK Python API, let's look at a more interesting deep learning problem in more detail.

1.2.2 Sequence classification

One of the most exciting areas in deep learning is the powerful idea of recurrent neural networks (RNNs). RNNs are in some ways the Hidden Markov Models of the deep learning world. They are networks with loops in them and they allow us to model the current state given the result of a previous state. In other words, they allow information to persist. So, while a traditional neural network layer can be thought of as having data flow through as in the figure on the left below, an RNN layer can be seen as the figure on the right.



As is apparent from the figure above on the right, RNNs are the natural structure for dealing with sequences. This includes everything from text to music to video; anything where the current state is dependent on the previous state.

While RNNs are indeed powerful, the "vanilla" RNN suffers from an important problem: long-term dependencies. Because the gradient needs to flow back through the network to learn, the contribution from an early element (for example a word at the start of a sentence) on a much later elements (like the last word) can essentially vanish.

To deal with the above problem, we turn to the Long Short Term Memory (LSTM) network. LSTMs are a type of RNN that are exceedingly useful and in practice are what we commonly use when implementing an RNN. For more on why LSTMs are so powerful, see, e.g. http://colah.github.io/posts/2015-08-Understanding-LSTMs. For our purposes, we will concentrate on the central feature of the LSTM model: the *memory cell*.

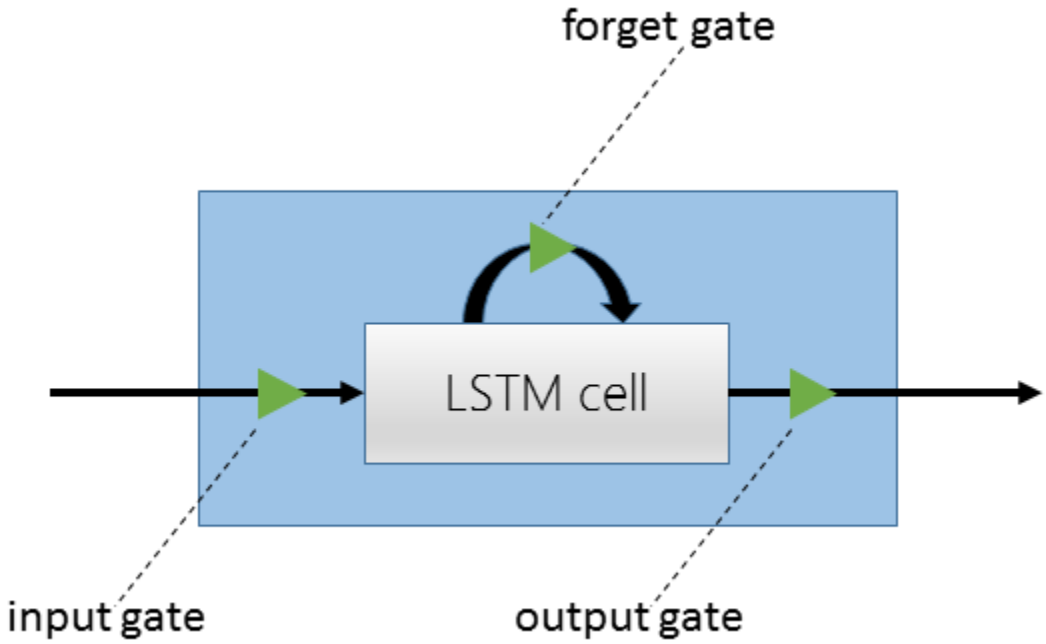


Fig. 1.1: An LSTM cell.

The LSTM cell is associated with three gates that control how information is stored / remembered in the LSTM. The "forget gate" determines what information should be kept after a single element has flowed through the network. It makes this determination using data for the current time step and the previous hidden state.

The "input gate" uses the same information as the forget gate, but passes it through a *tanh* to determine what to add to the state. The final gate is the "output gate" and it modulates what information should be output from the LSTM cell. This time we also take the previous state's value into account in addition to the previous hidden state and the data of the current state. We have purposely left the full details out for conciseness, so please see the link above for a full understanding of how an LSTM works.

In our example, we will be using an LSTM to do sequence classification. But for even better results, we will also introduce an additional concept here: word embeddings. In traditional NLP approaches, words are seen as single points in a high dimensional space (the vocabulary). A word is represented by an arbitrary id and that single number contains no information about the meaning of the word or how it is used. However, with word embeddings each word is represented by a learned vector that has some meaning. For example, the vector representing the word "cat" may somehow be close, in some sense, to the vector for "dog", and each dimension is encoding some similarities or differences between those words that were learned usually by analyzing a large corpus. In our task, we will use a pre-computed word embedding model (e.g. from GloVe) and each of the words in the sequences will be replaced by their respective GloVe vector.

Now that we've decided on our word representation and the type of recurrent neural network we want to use, let's define the computational network that we'll use to do sequence classification. We can think of the network as adding

a series of layers:

1. Embedding layer (individual words in each sequence become vectors)
2. LSTM layer (allow each word to depend on previous words)
3. Softmax layer (an additional set of parameters and output probabilities per class)

We can define this network as follows in the CNTK Python API:

import cntk as C

def seqcla(): # model num_labels = 5 vocab = 2000 embed_dim = 50

LSTM params input_dim = 50 output_dim = 128 cell_dim = 128

t = C.dynamic_axis(name='t') # temporarily using cntk1 SparseInput because cntk2's input() will simply allow sparse as a parameter features = cntk1.SparseInput(vocab, dynamicAxis=t, name='features') labels = C.input(num_labels, name='labels')

train_reader = C.CNTKTextFormatReader(train_file)

setup embedding matrix embedding = C.parameter((embed_dim, vocab),

learning_rate_multiplier=0.0, init_from_file_path=embedding_file)

get the vector representing the word sequence = C.times(embedding, features, name='sequence')

add an LSTM layer L = lstm_layer(output_dim, cell_dim, sequence, input_dim)

add a dense layer on top w = C.parameter((num_labels, output_dim), name='w') b = C.parameter((num_labels), name='b') z = C.plus(C.times(w, L), b, name='z') z.tag = "output"

and reconcile the shared dynamic axis pred = C.reconcile_dynamic_axis(z, labels, name='pred')

ce = C.cross_entropy_with_softmax(labels, pred) ce.tag = "criterion"

Let's go through some of the intricacies of the above network definition. First, we define some parameters of the data and the network. We have 5 possible classes for the sequences; we're working with a vocabulary of 2000 words; and our embedding vectors have a dimension of 50. Because the word vectors are input to the LSTM, the *input_dim* of the LSTM is also 50. We can, however, output any dimension from the LSTM; our *cell_dim* and *output_dim* are the same and we output 128-dimensional tensors.

We then set up our training data. First, we create a dynamic axis. The dynamic axis is a key concept in CNTK that allows us to work with sequences without having to pad our data when we have sequences of different lengths (which is almost always the case). We then set up our features by defining a *SparseInput*. In this release, `cntk.ops.input()` only supports dense features so we have to use the legacy *cntk1.SparseInput* until 1.5. Each word has a dimension of size *vocab* and we attach the dynamic axis *t* that we created just above. Then we set up our labels using the standard `cntk.ops.input()` where the dimension is of size *num_labels*.

Our final piece of setup before beginning to define the network is creating a *reader* for our training data. We use the `cntk.reader.CNTKTextFormatReader` and pass in the name of our training data file.

Now we can start defining our network. The first layer is the word embedding. We define this using a *parameter* of shape *(embed_dim, vocab)* that is initialized from a file where our embedding matrix is stored. We set the *learning_rate_multiplier* parameter to 0.0 so that this is treated as a constant.

To view the input data words as vectors, we multiply the embedding matrix with the one-hot vector words which results in the data being represented by vectors. An LSTM layer is then added which returns the last hidden state of the unrolled network. We then add the dense layer followed by the criterion node that adds a softmax and then implements the cross entropy loss function. Before we add the criterion node, however, we call `cntk.ops.reconcile_dynamic_axis()` which will ensure that the minibatch layout for the labels and the data with dynamic axes is compatible.

For the full explanation of how `lstm_layer()` is defined, please see the full example (seqcla.py) in the Examples section.

1.2.3 How to pass Python data as train/test data

The Python CNTK API allows to pass training / testing data either by specifing external input files or by using Python data directly to CNTK. This second alternative - using internal Python data - is usefull especially if you want to do some quick experimentation with small synthetic data sets. In what follows you will learn in what structure these data has to be provided.

Let us start with a scenario coming from one of our code examples (logreg_numpy.py). In this example we want to classify a 250 dimensional feature vector into one of two classes. In this case whe have two *inputs*:

- The features values for each training item. In the example these are 500 vectors each of dimension 250.
- The expected class. In this example the class is encoded with a two-dimensonal vector where the element for expected class is set to 1 and the other to 0.

For each of these inputs we have to provide one data structure containing all training instances.

You might notice that this is conceptually different to the case where we provide the data from external files using the CNTKTextReader. In the input file for CNTKTextReader we provide data for different *inputs* of one instance on the same line, so the data from different inputs are much more interwined.

In Python the feature data are reprensented by a NumPy array of dimension `number_of_instances X dimension_of_feature_space` so in out example its a NumPy array of dimension `500 X 250`. Likewise the expected output is reprensented by another NumPy array of dimension `500 X 2`.

1.2.4 Passing sequence data from Python

CNTK can handle sequences with arbitrary maximal length. This feature is also called *dynamic-axis*. To represent an input with a dynamic-axis in Python you have to provide each sequence as a NumPy-array where the first axis has a dimension equal to the sequence length. The complete dataset is then just a normal one-dimensional numpy array of these sequences.

Take as an artifical example a sentence classification problem. Each sentence has a different number of words, i.e. it is a *sequence* of words. The individual words might each be represented by some lantent vector. So each sentence is represented by a NumPy array of dimension `sequence_length X embedding_dimension`. The whole set of instances (sentences) is then represented by putting them into a one-dimensional array with the size equal to the number of instances.

CHAPTER 2

Concepts

There is a common property in key machine learning models, such as deep neural networks (DNNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). All of these models can be described as *computational networks*.

The directed edges of these *computational networks* are vectors, matrices, or in general n-dimensional arrays (tensors) which represent input data and model parameters. The vertices are *functions* (also called operations) that are performing a computation on these input tensors.

2.1 Tensors

The underlying data structure in CNTK is that of a *tensor*. It is a multidimensional array on which computations can be performed. Every dimension in these arrays is referred to as an *axis* to distinguish it from the scalar size of every axis. So, a matrix has two *axes* which both have a certain *dimension* corresponding to the number of rows and columns of the *axes*.

Using tensors makes the framework generic in that it can be used e.g. for classification problems where the inputs are vectors, black-and-white images (input is a matrix of points), color images (includes a separate dimension for r, g, and b) or videos (has an extra time dimension).

- Tensors have a *shape* which describes the dimensions of its axes. E.g. a shape `[2,3,4]` would refer to a tensor with three axes that have, respectively, 2, 3, and 4 dimensions.
- CNTK allows for the last axis to be a *dynamic axis*, i.e. an axis whose size might vary between input samples. This allows for easily modelling sequences (for recurrent networks) without needing to introduce masks or padding. See below for a detailed explanation.
- All data inside of a tensor is of a certain data type. Right now, CNTK implements *float* (32 bit) and *double* (64 bit) precision floating point types, and all tensors in a network have the same type.
- Tensors come either in *dense* or *sparse* form. Sparse tensors should be used whenever the bulk of its values are 0. The Python API currently doesn't expose sparse tensors; this will be added in the next release.

2.1.1 Usages of Tensors

Tensors are introduced in CNTK in one of three places:

- **Inputs**: These represent data inputs to the computation which are usually bound to a data reader. Data inputs are organized as (mini) batches and therefore receive an extra minibatch dimension. In addition, inputs can have a "ragged" axis called "dynamic axis" which is used to model sequential data. See below for details.

- **Parameters**: Parameters are weight tensors that make up the bulk of the actual model. Parameters are initialized using a constant (e.g. all 0's, randomly generated data, or initialized from a file) and are updated during *backpropagation* in a training run.
- **Constants**: Constants are very similar to parameters, but they are not taking part in backpropagation.

All of these represent the *leaf nodes* in the network, or, in other words, the input parameters of the function that the network represents.

To introduce a tensor, simply use one of the methods in the cntk namespace. Once introduced, overloaded operators can be applied to them to form an operator graph:

```
import cntk as C
x = C.input((2,3), name='features') # Input with shape [2,3,*]
c = C.constant(2)
w = C.parameter((2,3))          # Model parameter of shape [2,3], randomly initialized
op  = x * c                     # Elementwise multiplication operation
op2 = x * 2                     # Same as above (2 will be converted to constant)
op3 = x * [[1,2,3], [4,5,6]]   #  Elementwise multiplication of two 2x3 matrices
```

2.1.2 Broadcasting

For operations that require the tensor dimensions of their arguments to match, *broadcasting* is applied automatically whenever a tensor dimension is 1. Examples are elementwise product or plus operations. E.g. the following are equivalent (the outermost brackets are for sequences, see later for more details):

```
>>> C.eval(C.element_times([2,3],2))
  [array([[ 4.,  6.]])]
>>> C.eval(C.element_times([2,3],[2,2]))
  [array([[ 4.,  6.]])]
```

2.1.3 A Note On Tensor Indices

Multi-dimensional arrays are often mapped to linear memory in a continous manner. There is some freedom in which order to map the array elements. Two typical mappings are *row-major order* and *column-major order*.

For two-dimensional arrays (matrices) with *row-major order*, consecutive elements of the rows of the array are contiguous in memory; in column-major order, consecutive elements of the columns are contiguous.

For example the matrix

1	2
3	4
5	6

is linearized as [1, 2, 3, 4, 5, 6] using row-major order, but as [1, 3, 5, 2, 4, 6] using column-major order.

This concept extends to arrays of higher dimension than two: it is always about how a specific combination of index values is mapped to linear memory. If you go through elements in memory one by one and observe the corresponding tensor-indices then in *row major order* the right-most index changes fastest, while in *column-major order* the leftmost index changes fastest. (see https://en.wikipedia.org/wiki/Row-major_order)

In many programming languages like C or C#, row-major order is used. The same is true for the Python library NumPy (at least by default). CNTK, however, uses column-major order.

There are two circumstances where you have to be aware of this ordering:

1. When preparing input-files for CNTK. The values have to be provided in column-major order.

2. When parsing data outputted by CNTK.

2.2 Computational Networks

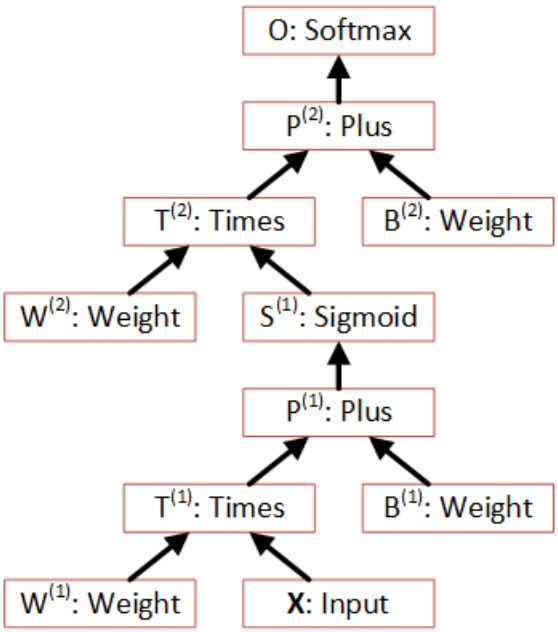
Once the input tensors are defined, CNTK allows for building up descriptions of the computations that are applied to it. These are translated into computational networks that describe the data flow as the data are transformed from input (leaf nodes) through computations, to one or more output (root) nodes.

The Python API allows us to specify such a computational network. For example, a one-hidden-layer sigmoid neural network can be described as shown below:

```
from cntk import *
# X is a data input, W1, W2, B are parameters
def one_hidden_layer_nn(X, W1, W2, B1, B2):
   T1 = W1 @ X      # Connect hidden layer T1 to input X through weight matrix W1
   P1 = T1 + B1     # Add bias term B1
   S1 = sigmoid(P1) # Elementwise sigmoid function
   T2 = W2 @ S1     # Second layer weight matrix
   P2 = T2 + B2     # Each column of B2 is the bias b2
   O = softmax(P2)  # Apply softmax column-wise to get output O
   return O
```

The example uses "@" as the infix matrix multiplication operator, which has been introduced in Python 3.5. For previous Python versions, the "times" function needs to be used instead: `T1 = times(W1, X)`.

The above creates a computational network like the following:



Here, `X` represents the input data as a tensor. During a training run, this would contain, in aggregated form, all the input samples for a particular minibatch. For the particular model this would have to be a two-dimensional tensor: the data in the first dimension would represent the feature vector, the second would refer to all the samples in the minibatch.

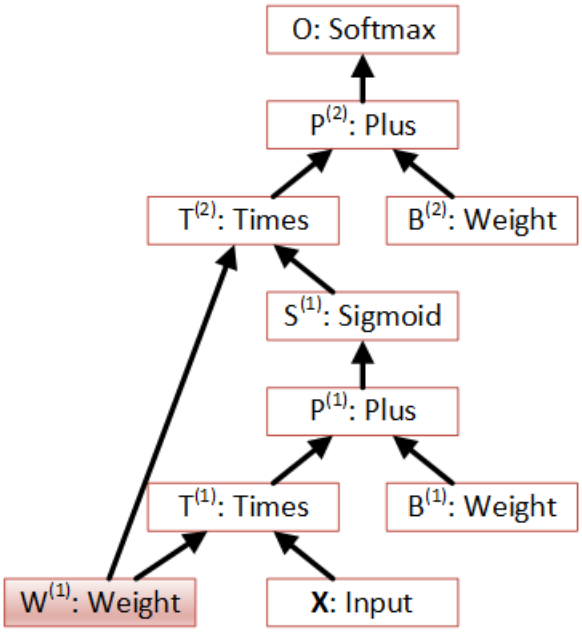
> Note: The above creates a network for *deferred computation*. The inputs are symbolic descriptions of tensors, not the data itself. As such the code above represents a higher-level function that returns a "lambda" rather than performing a computation by itself.

Of course, the above can also be written shorter:

```
def one_hidden_layer_nn(X, W1, W2, B1, B2):
   L1 = sigmoid(W1 @ X + B)
   L2 = W2 @ L1 + B2
   return softmax(L2)  # Apply softmax column-wise to get output O
```

Computational networks are flexible in several dimensions:

- They can have more than one input (leaf node). This feature is used, for example, to input features and labels on different inputs and model the loss function as part of the network. Note that CNTK doesn't apply a particular semantics to any of the inputs - they're just tensors. The semantics only come in through markup of model output, training criterion, and evaluation criterion nodes. See below.
- Inputs can be fed to several parts of the network. This allows for easily modelling shared model parameters, as shown in the following:



- They can have more than one output (root node). E.g. a single network can model a) the network output; b) the loss function, which represents the training criterion; amd c) an evaluation criterion which is used for verification. All of these functions differ only partially and can be modelled as part of the same network. CNTK makes sure that a) only requested root node outputs are computed and that b) shared parts between the functions represented at root nodes are only computed once.

2.2.1 Properties of Computation Nodes

In CNTK the computational nodes have a number of properties. Some of these can or must be set by the user.

- **name** - The symbolic name for the node. If left out, the name is assigned automatically to a numeric value.:

```
    S1 = sigmoid(P1, name='S1') # Elementwise sigmoid function
    S1.name = 'S1'              # Alternative way of assigning a name
```

  Assigning a name to a node is only necessary if it is the target of a loop. Otherwise, it can also be used for debugging.

- **tag** - This is a string that is attached to the node and has to be set for certain nodes. There purpouse is not documentary but controls the behaviour of CNTK. Namely, the SGD algorithm or output writers query the network for certain node tags to decide which nodes to treat as root nodes:

  S1 = sigmoid(P1, name='S1') # Elementwise sigmoid function S1.tag = 'output'

  The *tag* property can have the following values that can be set by the user:

  - *criterion* The output of such nodes as the optimisation criterion. See *Neural Net Training*
  - *output* The output of these nodes is written of the output.
  - *eval* The output of these nodes are used of evaluation. They might e.g. provide the error rate of a classification problem.

- **shape** - This is a derived property that is automatically inferred from the layout of the graph. *The value of this property is currently only output on the stderr of a training run*.
- **output** - At the moment every node has exactly one output tensor. Thus, a computation node can be used wherever a tensor is requested as an input. Therefore this is not exposed as a separate property.

2.3 Recurrent Networks

Efficiently modelling recurrent networks was an important design goal for CNTK. As such, in contrast to other toolkits, they are *first-order citizens* of the system. CNTK therefore allows for modelling of loops as part of the networks, and for dynamically sized input data. As such, it offers a very efficient implementation for training recurrent networks and does not require applying tricks to the input (like padding or masking) to simulate dynamically sized input data.

2.3.1 Dynamic Axes

Every input tensor in CNTK receives an additional (implicit) dimension usually referred to as "*". This is called the *dynamic axis* of the input. For a non-sequential task, this axis always has a length of 1 and thus reduces the behavior to that of any non-sequential machine learning tool. An example would be an image classification task, in which every image stands on its own. Nevertheless, in CNTK, a dynamic axis "*" will be printed, but it is benign.

For a task that involves sequences, input tensors (which are also often referred to as "samples") are concatenated along this axis, and every sequence may be of different length (hence the term "dynamic").

CNTK then manages all the intricate details of this: Loading dynamically sized tensors in memory in the best way possible such that the parallel computation on GPUs is maximized.

In a CNTK model description,

- every input can have its own dynamic axis
- dynamic axes can be shared between inputs. In fact, the default behavior is that all inputs share the same dynamic axis definition called "*". This makes it suitable to run two types of tasks without any further declaration:
  - tasks which do not have any sequence- or time dimension, such as a classification task on static input data, image convolutions etc.
  - tasks where all inputs share the same sequence dimension, such as language understanding or part-of-speech-tagging tasks

A specific dynamic axis is introduced by adding a `dynamic_axis()` node to the network and using it as an input argument to an `input()` node. The `dynamic_axis()` node thus acts as a "holder" for the layout information of the dynamic axis.

As an example, consider the following definition of inputs which comes from the *sequence classification* example. Here, the features input contains sequences which we want to classify by reading one label per sequence from the *labels* input:

```
t = C.dynamic_axis(name='t')
features = C.input(vocab, dynamicAxis=t, name='features')
labels = C.input(num_labels, name='labels')
```

These two inputs use two different dynamic axes, namely "*" (the labels input) and a newly introduced one called "t". At model verification time, CNTK now by default treats these two axes as incompatible, meaning that one could not simply run operations on them that require the dimensions to be the same for all elements.

Any operation that changes the cardinality of the dynamic axis introduces a new type. An example is a reduction operation that reduces the elements on this axis to 1. The output of this operation would have a new name assigned to the dynamic axis part.

What if, as a user, we know that two dynamic axes actually *have* the same layout? In these cases, the check for equality needs to be moved from verification time to runtime. This is done using the `reconcile_dynamic_axis()` operation. It performs a check whether all elements on its first input have the same dimension on the dynamic axis as the second one and, if so, output the dynamic axis name of the second input.

So, for the example above, a command like:

```
f2 = C.reconcile_dynamic_axis(labels, features)
```

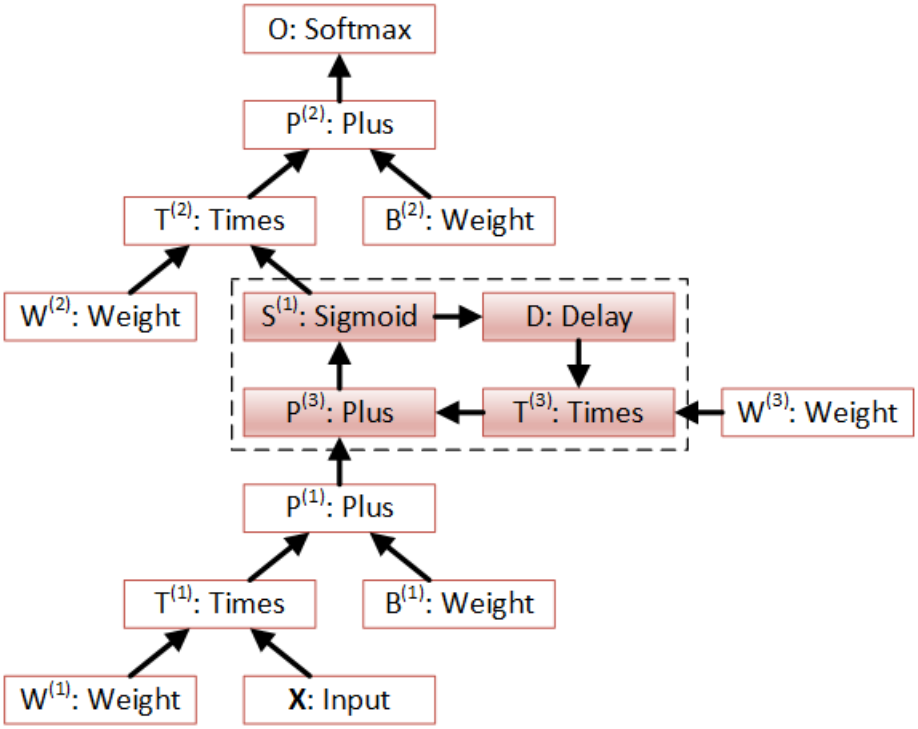
would output a tensor shape for *labels* that is exactly that of its input, but with the dynamic axis name changed to 't' (that of the features input).

2.3.2 Loops in Computational Networks

Different from the CN without a directed loop, a CN with a loop cannot be computed for a sequence of samples as a batch since the next sample's value depends on the the previous samples. A simple way to do forward computation and backpropagation in a recurrent network is to unroll all samples in the sequence over time. Once unrolled, the graph is expanded into a DAG and the forward computation and gradient calculation algorithms we just discussed can be directly used. This means, however, all computation nodes in the CN need to be computed sample by sample and this significantly reduces the potential of parallelization.

In CNTK, a recurrent neural network can simply be modelled by using the `past_value()` (earlier known as `delay()` node) and `future_value()` operations. These connect the network to the output of a previous (or next) step on the dynamic axis. CNTK detects loops automatically that are created this way, and turns them into a forward or backward iteration along the dynamic axis.

An example CN that contains a delay node is shown in the following figure.



In this example, CNTK has identified that the nodes T3 -> P3 -> S1 -> D -> T3 form a loop which needs to be computed sample by sample. All the rest of the nodes can be computed in batches. Once the loops are identified, they can be treated as a composite node in the CN and the CN is reduced to a DAG. All the nodes inside each loop (or composite node) can be unrolled over time and also reduced to a DAG.

It is important to note that the shape of the output of any operation that participates in a loop *shares the dynamic axis with its input*. This way, a recurrent network like LSTM can output its hidden state, cell state etc., unrolled over the time dimension.

See the LSTM example how past_value is used to form recurrent loops.

2.4 Readers

In CNTK, a data reader is a separate concept from the network itself. It is called by the network training algorithm to provide information about the data, to load minibatches into memory, and to attach this memory to the input nodes in

Readers are designed to be high performance to not become a bottlneck in GPU-heavy computations. They provide special facilities for

- Data prefetch: Readers can split up reading and preprocessing of data such that parallel computations are optimized.
- Transformations: e.g. ImageReader allows for certain preprocessing of the data (decoding, scaling etc.)
- Randomization: The readers support reading input data in a random order, to reduce the effects of data ordering on the training result.

Several task-specific readers have been implemented. The most generic ones are the following:

- A generic CNTKTextFormatReader (`cntk.reader.CNTKTextFormatReader`), which defines a text format for reading tensors and attaching them to inputs. The reader supports multiple inputs defined in a single file, allows for specifying dynamic axes by grouping samples by work unit (sequence) ID, and supports both sparse and dense tensors.

- ImageReader - for reading in image data stored in directories. Not exposed in Python API at this point.
- HTKMLFReader - for reading in data for a popular speech format. Not exposed in Python API at this point.
- A Numpy reader (as part of the Python API) which allows for using NumPy arrays as inputs to `input()` nodes. Internally these are serialized first and read back using CNTKTextFormatReader. This can be used during the exploration phase when data sizes are small and the network topology is iterated upon in an interactive fashion.

2.5 Neural Net Training

To perform a neural net training run, we need every operation to be defined for *forward* and *backward* operation. The forward operation simply computes the function value; the backward operation computes the gradients with regards to all of the operation's inputs.

All of the built-in operations (as far as they can take part in neural net training) define both the forward and backward pass. As such, CNTK implements *automatic differentiation*, since, for any function that can be defined through the use of the built-in operations, CNTK knows how to compute its derivatives.

In order to set up a computational network for training, the following is needed (in addition to training data):

- A training criterion node. CNTK comes with several built-in criterion nodes such as cross-entropy (with built-in softmax) for classification and mean-squared error for regression. The node needs to be tagged with "criterion" to get picked up by the training algorithm. The built-in criterion nodes currently output a scalar value which contains the aggregate loss over a minibatch.
- Optionally, an evaluation criterion node, which summarizes performance within the training run.
- A training algorithm. Currently CNTK provides an implementation of SGD (stochastic gradient descent) with optional momentum. This means that gradients are computed and backpropagated once for every minibatch. The SGD implementation offers an extensive number of options, e.g. for changing the learning rate over the course of training, or for choosing algorithms for distributed computation using data parallelism. See the description of the SGDParams class for details.

CNTK also provides several variants of data parallelism. These options are all available, but are currently not exposed in the Python API. To use data parallelism, please export the CNTK configuration file using the DelayedExecutionContext and overlay it with one of the methods described here: https://github.com/Microsoft/CNTK/wiki/Multiple-GPUs-and-machines

CHAPTER 3

Readers

Data readers are used by the computational network to provide information about the data, to load minibatches of data into memory, and to attach this memory to the input nodes in the network. The CNTK Python API currently supports the following reader classes.

3.1 Usage

CHAPTER 4

Operators

CHAPTER 5

Training

In the current version, the Python wrapper does not expose the full optimizer of CNTK, but rather encapsulates the different configuration options of the SGD optimizer.

CHAPTER 6

Execution Context

An execution context defines how and where a CNTK network will be created and run. The context can be either *Local* or *Deferred*. In the former case the functions (such as training and testing the network) are done locally and immediately so that the result is returned interactively to your Python session.

With a *Deferred* context, the functions simply set up a configuration file that can be used with CNTK at a later date. For example, if you would like to develop your network locally to get things working, and then launch the training on a GPU cluster, you can use the deferred context to simply turn your Python script into a CNTK configuration and then send that configuration to your cluster.

6.1 Usage

CHAPTER 7

Examples

7.1 Logistic Regression

Examples for logistic regression you find here: https://github.com/Microsoft/CNTK/tree/master/contrib/Python/cntk/examples/LogReg/

- Using training and testing data *from a file* : logreg.py .
- Using training and testing data *from a NumPy array* : logreg_numpy.py .

7.2 LSTM-based sequence classification

An Example for training an LSTM-based sequence classification model with embedding you find here: https://github.com/Microsoft/CNTK/tree/master/contrib/Python/cntk/examples/LSTM/ . A typical application would be text classification where we leverage a precomputed word-embedding. This is also a good example to see how to provide *input data for sequences* and using *sparse input*.

- In Train_sparse.txt we have two inputs. The input *x* provides the sequence data in sparse form, while *y* provides the classes in dense form.
- The example also uses a predefined embedding (embeddingmatrix.txt) mapping each dimension *x* to an embedding vector in a lower dimensional space.

7.3 One hidden layer neural network

Example for training a *one hidden layer neural network* using the MNIST-data (recognition of handwritten digits) you find here: https://github.com/Microsoft/CNTK/tree/master/contrib/Python/cntk/examples/MNIST/ .

To obtain and prepare the MNIST data use fetch_mnist_data.py .

CHAPTER 8

Release Notes

8.1 Version 1.4 (April 2016)

New and improved features:

- Python API: This is the first release containing a limited version of the Python API. It exposes CNTKTextFormatReader, SGD, local and deferred execution, and 22 operators.
- CNTK Core:
  - This release contains a new generic text format reader called CNTKTextFormatReader. UCIFastReader has been deprecated. The new reader by definition supports all tensor formats (sparse and dense, sequences and non-sequences, multiple inputs) that can be fed to CNTK.
  - The concept of named dynamic axes has been exposed in the configuration, which enables modelling of inputs of varying length.

Current restrictions of the Python API:

- Although CNTK implements more than 100 operators through internal APIs, only s small subset have been exposed through the Python API at this point. We are using this API as a production gate, requiring unit tests and documentation before new functionality is exposed. More operators will be added in the following weeks. In particular, convolution operations and reductions are missing at this point.
- The Python API is a pure out-of-process API at this point. This means that only methods on the context interact with CNTK directly through command line calls. An in-process API with far greater extensibility options is planned later in 2016 through the 2.0 release.
- The training loop is monolithic at this point and cannot be broken up into single forward/backward passes. This restriction will be lifted with 2.0 release.
- Although inputs can be sparse, sparse features cannot be fed through the Python API at this point for immediate evaluation. They can only be fed through files read through the CNTKTextFormatReader.
- We are only exposing the CNTKTextFormatReader in Python at this point. More data formats (ImageReader, speech formats) will be added in a later release.
- We are not exposing a standard layer collection for LSTMs etc. at this point. A first version of this will be added in the next release.
- Tensor shapes are only available after a call to the context methods, which run graph validation and tensor inference.
- Only few examples have been translated from the CNTK-internal configuration format (NDL) to the Python API. More will be added in the next releases.

Current restrictions of CNTK Core:

- A tensor can have only one dynamic axis (the outermost one).
- The support for sparse inputs on the operators is... sparse. Operations might throw NotImplementedExceptions when a sparse tensor is fed. The exact level of support will be described in the next release.
- The built-in criterion nodes aggregate over the whole minibatch. The SGD algorithm divides value this by the number of samples found on the default dynamic axis, not the one that was used as input to the criterion node. In the next release, criterion nodes will not aggregate over the dynamic axis any longer. This logic is moved to SGD itself.

8.2 Roadmap for Version 1.5

We are planning monthly releases for this API. The items on the agenda for May release (due end of May/early June) are:

- Python API: Greatly increased list of operators: Shape operations, elementwise operations, reductions.
- Python API: Support for image and speech readers
- Python API: Support for sparse input tensors instead of NumPy arrays, where applicable.
- Python API: First version of a layer API
- Readers: New speach reader
- Readers: Combination of reader deserializers and transformers
- Core: Profiling support
- Core: More operators planned for core.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search