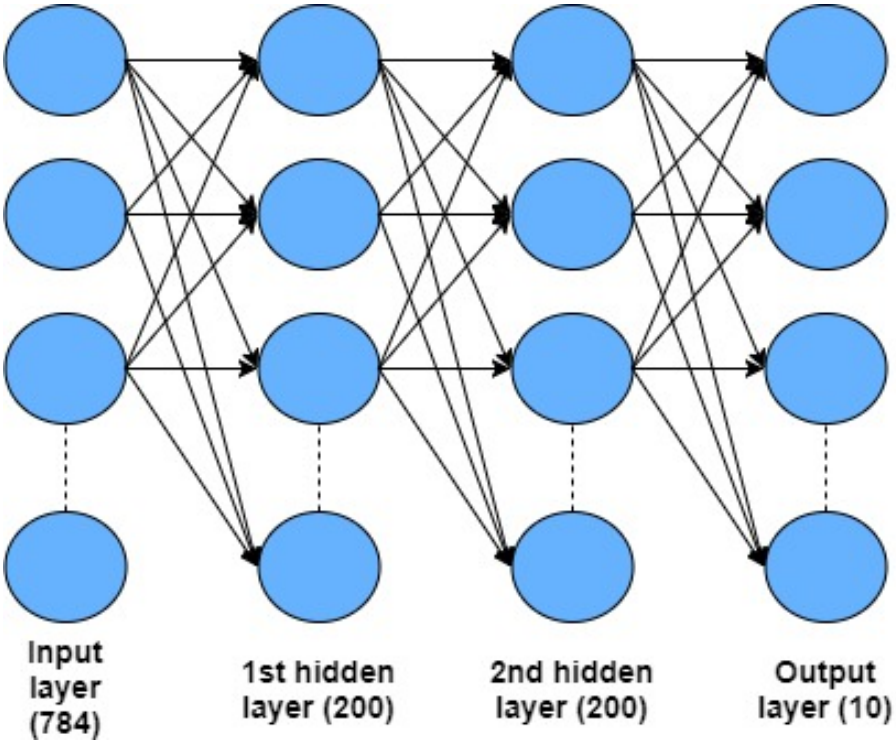
A Microsoft CNTK tutorial in Python – build a neural network

August 3, 2017 Andy CNTK, Deep learning 0



Dense MNIST architecture example

In previous tutorials (Python TensorFlow tutorial, CNN tutorial, and the Word2Vec tutorial) on deep learning, I have taught how to build networks in the TensorFlow deep learning framework. There is no doubt that TensorFlow is an immensely popular deep learning framework at present, with a large community supporting it.

However, there is another contending framework which I think may

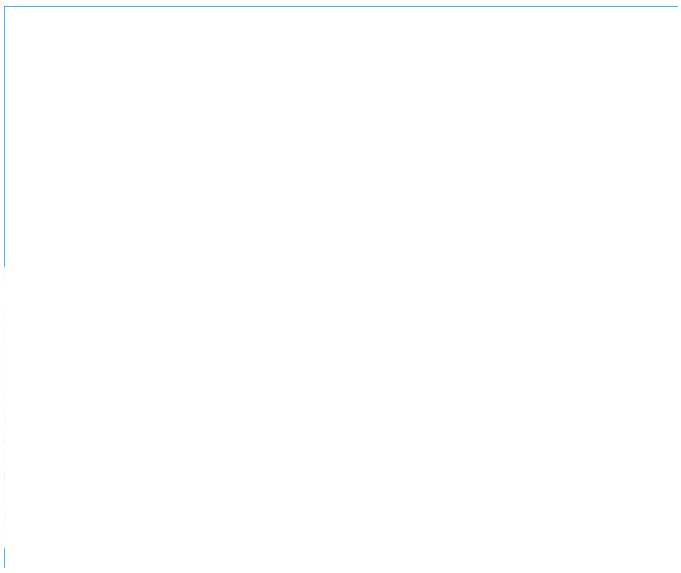
POPULAR TUTORIALS

Neural Networks Tutorial – A Pathway to Deep Learning
Python TensorFlow Tutorial – Build a Neural Network
Convolutional Neural Networks Tutorial in TensorFlow
Keras tutorial – build a convolutional neural network in 11 lines
Word2Vec word embedding tutorial in Python and TensorFlow

CATEGORIES

CNTK
Convolutional Neural Networks
Deep learning
Keras
Neural networks
NLP
Optimisation
TensorFlow
Word2Vec

actually be better – it is called the Microsoft Cognitive Toolkit, or more commonly known as CNTK. Why do I believe it to be better? Two main reasons – it has a more intuitive and easy to use Python API than TensorFlow, and it is *faster*. It also can be used as a back-end to Keras, but I would argue that there is little benefit to doing so as CNTK is already very streamlined. How much faster is it? Some benchmarks show that is it generally faster than TensorFlow and up to 5-10 times faster for recurrent / LSTM networks. That's a pretty impressive achievement. This article is a comprehensive CNTK tutorial to teach you more about this exciting framework.



Should you switch from using TensorFlow to CNTK? TensorFlow definitely has much more hype than Microsoft's CNTK and therefore a bigger development community, more answers on Stack Overflow and so on. Also, many people are down on Microsoft which is often perceived as a big greedy corporation. However, Microsoft has opened up a lot, and CNTK is now open-source, so I would recommend giving it a try. Let me know what you think in the comments. This post will be a comprehensive CNTK tutorial which you can use to get familiar with the framework – I suspect that you might be surprised at how streamlined it is.

Before going on – if you're not familiar with neural networks I'd recommend my neural networks tutorial or the course below as an introduction. Also, CNTK uses the computational graph methodology, as does TensorFlow. If you're unfamiliar with this concept – check out the first section of my TensorFlow tutorial. The code for this course, which is loosely based on the CNTK tutorial here, can be found on this site's Github page.

Recommended online course for neural networks: If you like video courses, I'd recommend the following inexpensive Udemy course on neural networks: Deep Learning A-Z: Hands-On Artificial Neural Networks

CNTK inputs and variables

The first thing to learn about any deep learning framework is how it deals with input data, variables and how it executes operations/nodes in the computational graph. In this CNTK tutorial, we'll be creating a three layer densely connected neural network to recognize handwritten images in the MNIST data-set, so in the below explanations, I'll be using examples from this problem. See the above-mentioned tutorials (here and here) for other implementations of the MNIST classification problem.

Variables

For the MNIST classification task, each training sample will have a flattened 28 x 28 = 784 pixels grey scale input and ten labels to classify (one for each hand-written digit). In CNTK we can declare the variables to hold this data like so:

```
import cntk as C
input_dim = 784
num_output_classes = 10
feature = C.input_variable(input_dim)
label = C.input_variable(num_output_classes)
```

These *input_variable* functions are like the placeholder variables in TensorFlow. However, CNTK removes the necessity to explicitly identify the number of samples/batch size and we simply supply the dimensions for each training/evaluation sample (in TensorFlow, one had to explicitly use the "?" symbol to designate unknown batch size). In this case, we have the flattened 28 x 28 = 784 pixel input and 10 output labels / classes. If we wanted to maintain the un-flattened input shape for, say, a convolutional neural network task, we would instead specify *input_dim = (1, 28, 28)*.

As will be shown later, these variables can be easily "loaded up" with our batch training or evaluation data.

Data readers

Some of the most time consuming (and annoying tasks) in machine learning involve getting your training data into a good format, reading it into your model and setting up batch retrieval functions (if you're not sure about what a "mini-batch" is in this context, check out my Stochastic Gradient Descent tutorial). The CNTK framework has some helper modules which can aid in at least some of this process. For this tutorial, we will be using the MNIST data-set obtained from the following website – http://yann.lecun.com/exdb/mnist/. To access this data, we need to download two Python files from the following repository: https://github.com/Microsoft/CNTK/tree/v2.0/Examples/Image/DataSets/MNIST called install_mnist.py and mnist_utils.py. After running install_mnist.py, it will download the data-set into your project directory. Once this has been done, you can run two different functions – the first of which is called *CTFDeserializer(). CTFDeserializer()* reads text files in a special CNTK format. The CNTK format requires a sample per line, with the data of the sample being given a name/label using the pipe "|" delimiter. For our MNIST case, the data has been put in the following format for a single sample:

|labels 0 0 0 0 0 1 0 0 0 0 |features 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0....

As can be observed, we have two fields per sample – the label of the MNIST image, which corresponds to the one-hot designation of the digit in the image (i.e. a digit of "2" will be represented as 0 0 1 0 0 0 0 0 0 0) and the features of the image which represent each of the 784 (flattened) pixels of the image. We read this CNTK file format by using the *CTFDeserializer()* like follows:

```
from cntk.io import MinibatchSource, CTFDeserializer,
StreamDef, StreamDefs
CTFDeserializer(path, StreamDefs(
        features=StreamDef(field='features',
shape=input_dim),
        labels=StreamDef(field='labels',
shape=num_output_classes)))
```

The first argument to the *CTFDeserializer()* function is the path of the data it is to read from. The second argument is a

StreamDefs() dictionary-like object. This object assigns a set of keys to different *StreamDef()* objects. A *StreamDef()* object specifies what pipe ("|") label it should be searching for in the CT file, and what the size of data per sample it should be retrieving. So if we take the "|labels" pipe designation in the file, we know that it should be followed by the number of labels: *num_output_classes=10*. We then feed the *StreamDefs()* object, which contains the description of each data-point per sample, to the *CTFDeserializer()* function. When this function is called, it will read the data into the specified keys (*features* and *labels*).

The next step is to setup the CNTK object which can draw random mini-batch samples from our *CTFDeserializer().* This object is called *MinibatchSource()*. To use it, we simply supply it a serializer, in this case, our previously mentioned *CTFDeserializer().* It will then automatically draw random mini-batch samples from our serialized data source of a size that we will specify later when we look at the *training_session()* object. To create the *MinibatchSource()* object, we simply do the following:

```
reader_train = MinibatchSource(CTFDeserializer(path,
StreamDefs(
        features=StreamDef(field='features',
shape=input_dim),
        labels=StreamDef(field='labels',
shape=num_output_classes))))
```

The above variables, functions, and operations now allow us to draw data into our training and evaluation procedures in the correct fashion.

CNTK Operations

As with TensorFlow, CNTK has operations which are nodes in a computational graph. These nodes and operations also flow into each other. In other words, if the output of an operation *B* first requires the output of another operation *A*, then CNTK understands this relationship. If we wish to know what the output of operation *B* is, all we have to do is call *B.eval()* and this will automatically call *A.eval()*. In CNTK there are all sorts of operations which we can run – from simple multiplication and division to softmax and convolutional operations. Again, if we need to explicitly evaluate any of these in our code all we have to do is call the operation by

executing *eval()* on the operation name. However, most operations we won't have to explicitly evaluate, they will simply be evaluated implicitly during the execution of the final layer of our networks.

In this MNIST example, a simple operation we need to perform is to scale the input features. The grey scale pixel images will have maximum values up to 256 in the raw data. It is better to scale these feature values to between 0 and 1, therefore we need to multiply all the values by 1/256 ~ 0.00390 to achieve this scaling:

```
from cntk.ops import relu, element_times, constant
scaled_input = element_times(constant(0.00390625),
feature)
```

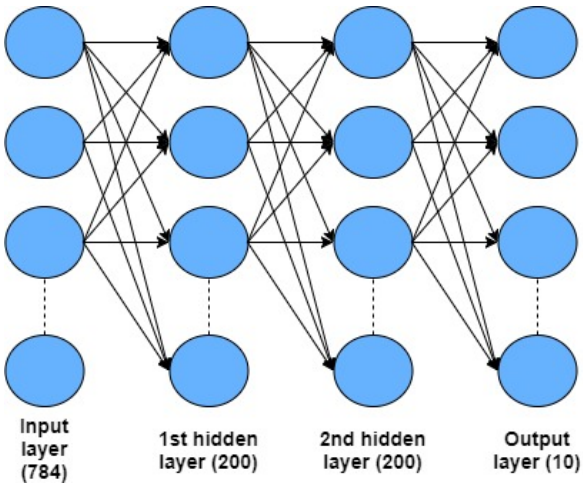
Here we declare a constant of 0.00390 and use the *element_times()* operation to multiply it by the input variable *feature* which we declared earlier. This scales our input data to between 0 and 1.

Now we are ready to start building our densely connected neural network using CNTK layers.

CNTK layers

Creating a single layer

As with any deep learning framework, CNTK gives us the ability to create neural network layers. These layers come in many flavors, such as Dense, Convolution, MaxPooling, Recurrent, and LSTM, all of which can be reviewed here. In our case, we wish to create some densely connected layers for our standard neural network classifier of the MNIST dataset. The architecture we are going to create looks like this:



Dense MNIST architecture example

So basically we have our 784 flattened pixel input layer, followed up by two hidden layers both of size 200, and finally, our output layer which will be softmax activated. So how do we create our first hidden layer? Quite easily in CNTK:

```
from cntk.ops import relu
from cntk.layers import Dense
hidden_layers_dim = 200
h1 = Dense(hidden_layers_dim, activation=relu)
(scaled_input)
```

To create a densely connected hidden layer, as shown in the example figure above, all we need to do is declare a *Dense()* object. The first argument to this object is the size of the hidden layer – in this case, it is equal to 200. Then we specify the node activation type – in this case, we will use rectified linear units or relu. Following this declaration, we pass the *scaled_input* variable/operation to the layer in brackets – this is the data that will be fed into the first hidden layer. CNTK takes care of all the relevant connections and handles the dimensions of the input tensor automatically. It also includes the weight and bias variables internally, so you don't have to go explicitly declaring them as you do in TensorFlow. We can supply an initialization function for these weights/bias values, by supplying an optional argument *init*. In the absence of this argument, the initialization defaults to the *glorot_uniform()* function/distribution.

We can then build a hierarchical graph of operations connecting the other hidden layer and the output layer:

```
h2 = Dense(hidden_layers_dim, activation=relu)(h1)
z = Dense(num_output_classes, activation=None)(h2)
```

For h2, we declare an identical dense hidden layer and feed it the output of the first layer (*h1*). Finally, we create an output layer equal to the number of output classes, but we set the activation function here to None – this is to make the output layer nodes simple summing nodes with no activation. We will apply the softmax function to this simple summation later.

CNTK includes some additional options to make the declaration of layers easier.

Some layer creation helpers

Often one wants to supply the same initialization functions and activation functions to multiple layers, and it is often the case that we have repeating structures in our networks. In this vein, CNTK has some helper functions/objects to make our network definitions more streamlined. The first, and probably most important of these is the Sequential() module – this is very similar to the Sequential() paradigm used in Keras (see this Keras tutorial). It allows the user to simply stack layers on top of each other in a sequential fashion without having to explicitly pass the output of one layer to the input of the next. In this example, the use of Sequential() looks like this:

```
z = Sequential([
        Dense(hidden_layers_dim, activation=relu),
        Dense(hidden_layers_dim, activation=relu),
        Dense(num_output_classes)])(scaled_input)
```

Here we can see that we've simply added our two hidden layers, and the final output layer, into a list argument required by the Sequential() module. We can reduce this code even further, by using the For() function available in the Layers CNTK interface. This is basically an embedded *for* loop that can be used in the Sequential() module:

```
num_hidden_layers = 2
z = Sequential([For(range(num_hidden_layers),
lambda i: Dense(hidden_layers_dim, activation=relu)),
Dense(num_output_classes)])(scaled_input)
```

The For() loop uses a lambda function to construct the layers – in the above, this is performed over two hidden layers, with the final output layer appended to the end of this For() construction.

One final construct in CNTK that can assist in stream-lining is the *layers.default_options()* module – this is not particularly useful in this example, but it can be useful in more complicated networks, so I include it here for illustrative purposes. Basically, it allows the user to specify common activation functions, weight initialization procedures, and other options. We use the Python *with* functionality to call it:

```
with default_options(activation=relu,
init=C.glorot_uniform()):
    z = Sequential([For(range(num_hidden_layers),
        lambda i: Dense(hidden_layers_dim)),
        Dense(num_output_classes, activation=None)])
(scaled_input)
```

Notice that we no longer have to specify the activation function for the hidden layers, yet we can override the default_option for the output layer to *None* which allows us to apply a softmax in the loss function (which will be demonstrated in the next section). The same initialization of the *glorot_uniform()* is applied to each layer.

We have now defined the structure of our neural network, and you can see how simple it is to do – especially using the Sequential() module in CNTK. Now we have to setup our loss function and training session.

CNTK losses, errors, and training

Defining the loss and error

The CNTK library has many loss functions to choose from in order to train our model. These range from the standard cross entropy and squared error to the cosine distance (good for measuring vector similarities, such as word embeddings) and lambda rank. You can also define your own custom loss functions. For our purposes, as is often the case for classification tasks, we'll use the *cross_entropy_with_softmax()* option:

```
ce = cross_entropy_with_softmax(z, label)
```

Here we simply supply the output layer *z* (which you'll remember doesn't yet have a nonlinear activation function applied) and the *label* output variable/placeholder and we get the cross entropy loss with softmax applied to *z*.

Next, we want to have some way of assessing the error (or conversely accuracy) on our test set, as well as when we are training our model. CNTK has a few handy metrics we can use, along with the ability to define our own. Seeing this is a classification task, we can use the handy *classification_error()* function:

```
pe = classification_error(z, label)
```

Now we need to setup the required training objects.

Training models in CNTK

In order to perform training in CNTK, we first have to define our input map, which is simply a dictionary containing all the input and output training pairs:

```
input_map = {
        feature: reader_train.streams.features,
        label: reader_train.streams.labels
    }
```

Note that here we are using the *MinibatchSource()* object (with the associated reader/deserializer) called *reader_train* that we created earlier. One can access the streams/data by using the dot notation shown above.

The next step is to create a progress writer object, called ProgressPrinter. This object allows us to output metrics, such as the loss and classification error, to the command window while training. It also allows the ability to print the data to text files, vary printing frequency and has a number of methods which bear examining. In this case, we will just use it fairly simply:

```
num_sweeps_to_train_with = 10
# Instantiate progress writers.
progress_writers = [ProgressPrinter(
    tag='Training',
    num_epochs=num_sweeps_to_train_with)]
```

In the above, the *tag* argument is what shows up in the log attached to each update. The *num_epochs* is used to allow a counter to count up to the total number of epochs during the training. This will become clear later as we print out the results of our training.

The learning rate schedule object

It is often the case that, during training, we wish to vary optimization variables such as the learning rate. This can aid in convergence to an optimal solution, with learning rates often reduced as the number of epochs increase – for example, reducing the step size in our gradient descent steps (for more information about gradient descent, see my neural networks tutorial). We can setup such a learning rate schedule using the *learning_rate_schedule()* object. With this object, we can select a learning rate of, say, 0.1 for the first 1,000 samples, then a learning rate of 0.01 for the next 1,000 samples, then 0.001 for all remaining samples. To do this, we would create a *learning_rate_schedule()* using the following code:

```
lr = learning_rate_schedule([0.1, 0.01, 0.001],
UnitType.sample, 1000)
lr[0], lr[1000], lr[2000]
0.1, 0.01, 0.001
```

The first argument in this declaration is the schedule, which is the set of learning rate values that will be used as the training progresses. The second argument *UnitType.sample* designates that the learning rate will change depending on a number of *samples*. The other alternative is *UnitType.minibatch* which means that the learning rate will change depending on a number of *mini batches*. Finally, we have the epoch size, which is the number of samples in this case, that need to be processed before the next change in learning rate.

In the case of our simple network, we can use the same learning rate for all samples, so we just declare our *lr* variable by:

```
from cntk.learners import learning_rate_schedule,
UnitType
lr = learning_rate_schedule(1, UnitType.sample)
```

For each sample, the learning rate will be 1.

The Trainer object

Next, we need to setup a Trainer object. This module is what performs the training of our model – therefore we need to feed it a number of pieces of information. First, we need to give it the output r operation (the variable *z* in our case) – the prior layers will be ned using the computational graph structure. Next, we need to it our loss function that we are going to use for computing our dients. We also want to supply our metric that we will track ng training. Then we need to specify what type of optimizer to – stochastic gradient descent, AdaGrad, Adam etc. (see the list lable here) and also our learning rate schedule. Finally, we need pecify any progress writers that we wish to use. In our case, it s like this:


```
ainer = Trainer(z, (ce, pe), [adadelta(z.parameters,
)], progress_writers)
```

first argument is our output layer operation *z*. The next ument can be either the loss function alone or a tuple containing loss function and a metric to track – this is the option we have n here. Third, we have a list of optimizers to use – in this case, we use a single optimizer called *adadelta()*. The optimizer takes the final layer's parameters (which implicitly include the previous layer parameters via the directed computational graph – see here for more information) and a learning rate schedule which we have already defined. Finally, we include our progress writer object.

Now we are ready to create a training session object which we can use to train the model.

The training session object

The CNTK library has a great way of training models using the *training_session()* object. This handles all of our mini-batch data extraction from the source, our input data, the frequency of logging

and how long we want to run our training for. This is what it looks like:

```
minibatch_size = 64
num_samples_per_sweep = 60000
num_sweeps_to_train_with = 10
training_session(
        trainer=trainer,
        mb_source=reader_train,
        mb_size=minibatch_size,
        model_inputs_to_streams=input_map,
        max_samples=num_samples_per_sweep *
num_sweeps_to_train_with,
        progress_frequency=num_samples_per_sweep
    ).train()
```

First, we supply the *training_session()* a trainer object over which the optimization and parameter learning will occur. Then we provide it a source from which to extract mini-batch data from – in this case, it is our *reader_train* MinibatchSource object that we created earlier. We then let the session object know how many samples we wish to extract per mini-batch.

Next comes our input map. This matches our input variables (*label* and *feature*) with the appropriate streams from the deserializer object embedded in *reader_train*. We then supply the maximum number of samples we wish the training session to process – once this limit of samples passed through the model has been reached, the training will cease. Finally, we supply the frequency at which we wish to print the progress via the *progress_writers* object we created.

There are other options that *training_session()* makes available, such as saving the model at checkpoints, cross validation, and testing. These will be topics for a future post.

In this case, we are going to try to cover all the data set *(num_samples_per_sweep=60,000) by* sweeping over it 10 times (*num_sweeps_to_train_with=10*). The selection of data will be random via the mini-batches, but it is statistically likely that most of the data will be sampled in a mini-batch at some point in the 10 sweeps.

This is what the output of the training looks like:

```
Learning rate per sample: 1.0
Finished Epoch[1 of 10]: [Training] loss = 0.304787 * 60032, metric = 8.60% * 60032 4.607s (13030.6 samples/s);
Finished Epoch[2 of 10]: [Training] loss = 0.139908 * 59968, metric = 4.08% * 59968 3.569s (16802.5 samples/s);
Finished Epoch[3 of 10]: [Training] loss = 0.102943 * 60032, metric = 3.07% * 60032 3.870s (15512.1 samples/s);
Finished Epoch[4 of 10]: [Training] loss = 0.081415 * 59968, metric = 2.41% * 59968 3.594s (16685.6 samples/s);
Finished Epoch[5 of 10]: [Training] loss = 0.066858 * 60032, metric = 2.00% * 60032 3.474s (17280.4 samples/s);
Finished Epoch[6 of 10]: [Training] loss = 0.055682 * 59968, metric = 1.61% * 59968 3.494s (17163.1 samples/s);
Finished Epoch[7 of 10]: [Training] loss = 0.047566 * 60032, metric = 1.41% * 60032 3.501s (17147.1 samples/s);
Finished Epoch[8 of 10]: [Training] loss = 0.040766 * 59968, metric = 1.16% * 59968 3.570s (16797.8 samples/s);
Finished Epoch[9 of 10]: [Training] loss = 0.035123 * 60032, metric = 0.99% * 60032 3.507s (17117.8 samples/s);
Finished Epoch[10 of 10]: [Training] loss = 0.030695 * 59968, metric = 0.90% * 59968 3.540s (16940.1 samples/s);
```

Output from the progress writer during training

We can see that our model error on the training data set gets down to 0.9% – not bad!

However, as any machine learning practitioner will tell you, we should test our model on a separate test data set, as we may have overfitted our training set.

Testing the model

First, we need to setup another *MinibatchSource* which reads in our test data, along with a new input map that refers to the test data:

```
# Load test data
path = abs_path + "\Test-28x28_cntk_text.txt"

reader_test = MinibatchSource(CTFDeserializer(path,
StreamDefs(
    features=StreamDef(field='features',
shape=input_dim),
    labels=StreamDef(field='labels',
shape=num_output_classes))))

input_map = {
    feature: reader_test.streams.features,
    label: reader_test.streams.labels
}
```

A handy thing about the *MinibatchSource* object is that we can extract mini-batches from the reader object by using *.next_minibatch()*. This allows us to run through a whole bunch of mini-batches from our test set to estimate what the average classification error is on our test data. We can also feed in this mini-batch data, one batch at a time, to our trainer object to get the classification error on that batch. To do so, the trainer object has a handy method called *.test_minibatch().* The code below shows how all this works:

```
# Test data for trained model
test_minibatch_size = 1024
num_samples = 10000
num_minibatches_to_test = num_samples /
test_minibatch_size
test_result = 0.0
for i in range(0, int(num_minibatches_to_test)):
    mb = reader_test.next_minibatch(test_minibatch_size,
input_map=input_map)
    eval_error = trainer.test_minibatch(mb)
    test_result = test_result + eval_error
```

By dividing the *test_result* value in the above code by the *num_minibatches_to_test* variable, we can get our average classification error on the test set. In this case, it is 1.98% – pretty good for a densely connected neural network. In a future post, I'll show you how to create convolutional neural networks in CNTK which perform even better.

So there you have it – in this post, I have shown you the basics of Microsoft's challenge to TensorFlow – the Cognitive Toolkit or CNTK. It is an impressive framework, with very streamlined ways of creating the model structure and training. This coupled with proven high speeds makes it a serious competitor to TensorFlow, and well worth checking out. I hope this post has been a help for you in getting started with this deep learning framework.

« **PREVIOUS**