MXNet review: Amazon's scalable deep learning

Amazon's favorite deep learning framework scales across multiple GPUs and hosts, but it's rough around the edges

By Martin Heller Contributing Editor, InfoWorld | Dec 14, 2016

http://www.infoworld.com/article/3149598/artificial-intelligence/mxnet-review-amazons-scalable-deep-learning.html

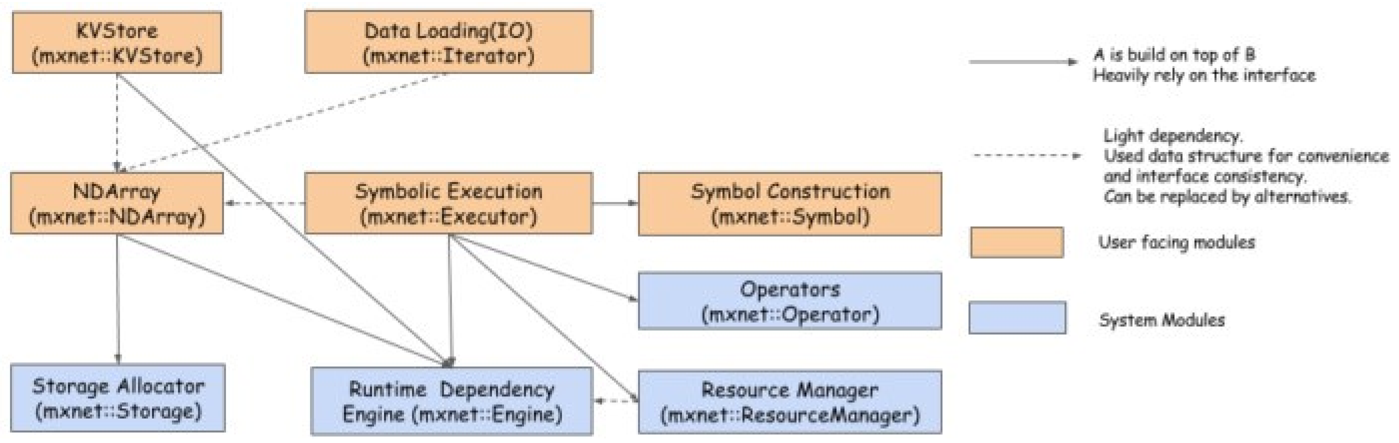
Deep learning, which is basically neural network machine learning with multiple hidden layers, is all the rage—both for problems that justify the complexity and high computational cost of deep learning, such as image recognition and natural language parsing, and for problems that might be better served by careful data preparation and simple algorithms, such as forecasting the next quarter's sales. If you actually need deep learning, there are many packages that could serve your needs: Google TensorFlow, Microsoft Cognitive Toolkit, Caffe, Theano, Torch, and MXNet, for starters.

I confess that I had never heard of MXNet (pronounced "mix-net") before Amazon CTO Werner Vogels noted it in his blog. There he announced that in addition to supporting all of the deep learning packages I mentioned above, Amazon decided to contribute significantly to one in particular, MXNet, which it selected as its deep learning framework of choice. Vogels went on to explain why: MXNet combines the ability to scale to multiple GPUs (across multiple hosts) with good programmability and good portability.

MXNet originated at Carnegie Mellon University and the University of Washington. It is now developed by collaborators from multiple universities and many companies, including the likes of Amazon, Baidu, Intel, Microsoft, Nvidia, and Wolfram. MXNet allows you to mix symbolic programming (declaration of the computation graph) and imperative programming (straight tensor operations) to maximize both efficiency and productivity.

The MXNet platform is built on a dynamic dependency scheduler that automatically parallelizes both symbolic and imperative operations on the fly, although you have to tell it what GPU and CPU cores to use. A graph optimization layer on top of the scheduler makes symbolic execution fast and memory efficient.

MXNet currently supports building and training models in Python, R, Scala, Julia, and C++; trained MXNet models can also be used for prediction in Matlab and JavaScript. No matter what language you use for building your model, MXNet calls an optimized C++ back-end engine.


KVStore (mxnet::KVStore)
Data Loading(IO) (mxnet::Iterator)
NDArray (mxnet::NDArray)
Symbolic Execution (mxnet::Executor)
Symbol Construction (mxnet::Symbol)
Operators (mxnet::Operator)
Storage Allocator (mxnet::Storage)
Runtime Dependency Engine (mxnet::Engine)
Resource Manager (mxnet::ResourceManager)
A is build on top of B
Heavily rely on the interface
Light dependency.
Used data structure for convenience and interface consistency.
Can be replaced by alternatives.
User facing modules
System Modules

An overview of the MXNet architecture: NDArrays are representations of tensors. The KVStore is a distributed key-value store for data synchronization over multiple devices.

MXNet features

In the original paper on MXNet, the authors explain why they combined the symbolic declaration of a computation graph, which allows more opportunity for optimization of neural network solutions, and imperative programming of tensors, which allows for more flexibility, more natural variation of parameters, and easier debugging. They went to some effort to bind their API to multiple programming languages and to implement autodifferentiation to derive gradients.

The authors consider the MXNet API a superset of what's offered in Torch, Theano, Chainer, and Caffe, while offering more portability and support for GPU clusters. They consider MXNet similar to TensorFlow but with the ability to embed imperative tensor operations.

The MXNet architecture, shown in the figure above, has five user-facing modules and five system modules. The SimpleOp module, which is *not* shown in the diagram, has been split off from the Operator module, which *is* shown. That doesn't really matter to a programmer, however. The Operator module includes operators that define static forward and gradient calculation. SimpleOp includes operators that extend to NDArray operators and symbolic operators in a unified fashion.

Looking at the Python API, there are six high-level interfaces:

- The Module API is a flexible high-level interface for training neural networks.
- The Model API is an alternative, simple high-level interface for training neural networks.
- The Symbolic API performs operations on NDArrays to assemble neural networks from layers.
- The IO Data Loading API performs parsing and data loading.
- The NDArray API performs vector/matrix/tensor operations.
- The KVStore API performs multi-GPU and multihost distributed training.

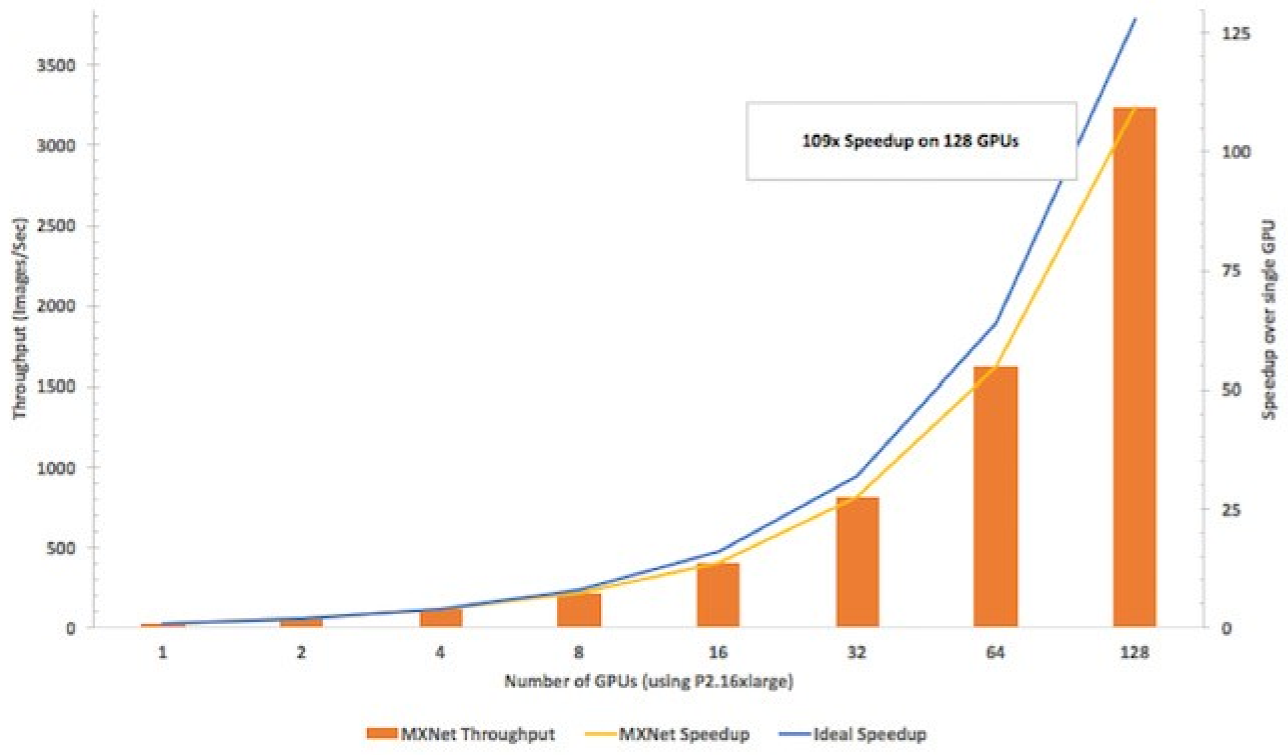
The MXNet shared back-end library is written in C++ with the standard C++ library for efficiency and portability. The language embeddings are written in the target languages Python, R, and Scala, with some shims in C++ for the R API.

A fair chunk of the engine code provides support for Nvidia CUDA general-purpose GPUs. For the highest performance, build and run MXNet to use the CUDA SDK and CUDA Deep Neural Network (CUDNN) library. As the MXNet authors noted when comparing their implementation with competitors on the convnet-benchmarks, most deep neural network computations are spent on the CUDA/CUDNN kernels. At the time the original paper was written, TensorFlow used an older version of CUDNN than the other packages and, therefore, ran slower; that has since been corrected.

When researching scalability for that paper, the MXNet authors ran a GoogLeNet benchmark (a 22-layer incarnation of the Inception convolutional neural network for image object detection and classification) on one and 10 Amazon EC2 g2.8xlarge instances, and they showed a superlinear speedup. When Amazon tested a MXNet implementation of the related Inception v3 algorithm on P2.16xlarge instances for varying numbers of GPUs, the results showed a scaling efficiency of 85 percent of the number of GPUs in use, as shown in the figure below.

Those are impressive results. The p2.16xlarge instances have 16 Nvidia Tesla K80 GPUs (eight boards), and Amazon ran the scaling test out to 128 GPUs. That's *cluster* performance.

I haven't tried to reproduce either experiment. Your mileage may vary and will depend heavily on your neural network design (the more layers, the longer it takes) and choice of optimizer (some converge better than others for specific algorithms and data sets).



Amazon

Amazon tested an Inception v3 algorithm implemented in MXNet on P2.16xlarge instances and found a scaling efficiency of 85 percent.

MXNet installation

MXNet supports training on Linux, OS X, Windows, and Docker; it also supports prediction from JavaScript in a browser and on an iOS or Android device using an amalgamated C++ package.

I did my initial installation on a MacBook Pro with a Nvidia GeForce GT 650M GPU and both CUDA and CUDNN installed. Based on previous experience with TensorFlow, I initially built MXNet for the CPU, not the GPU.

On a Mac, you first use Homebrew to install some prerequisites, then clone MXNet from GitHub, copy the Mac makefile, and build the shared library. After some initial fumbles, it took me about eight minutes to compile and link the library. Once the library has been built, you need to install the language packages you want to use. If you want to use Jupyter notebooks with Python to run the MXNet samples, you need to install Jupyter as well, although the MXNet instructions don't mention that.

Using the Amazon Deep Learning AMI, which has MXNet and four other deep learning packages pre-installed along with all of the required languages and the Anaconda Python library package (including Jupyter), can save you the effort of building the packages from source or even downloading them. Using the older Amazon G2 (K520 GPUs), newer P2 instances (K80 GPUs), or Microsoft Azure NC instances (K80 GPUs) will enable you to run MXNet on serious floating point processors. IBM SoftLayer also offers servers with K80 GPUs on monthly and hourly terms. The Google Cloud will offer VM instances with K80 GPUs and even faster P100 GPUs in 2017.

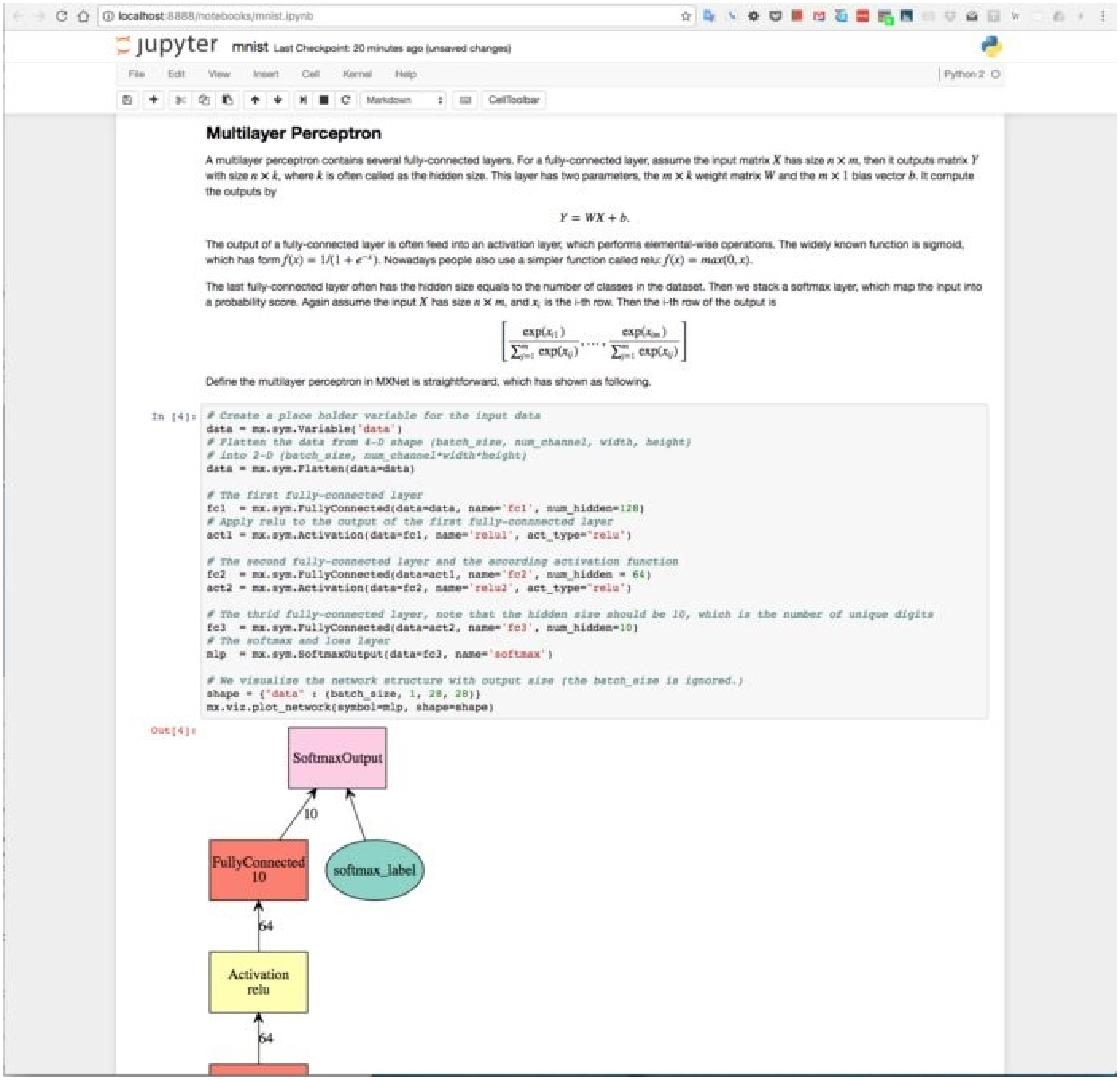
Running MXNet

The “getting started” exercise for MXNet is (surprise!) a simple network for training a classifier on the MNIST hand-drawn numerals data set, one of the easiest standard machine learning problems. This exercise covers how to train a multilayer perceptron model using Python, R, Scala, and Julia. The multilayer perceptron and a more accurate convolutional solution (LeNet) are covered in a Python tutorial. You’ll find the source code for this tutorial in Jupyter notebook form within the mxnet-notebooks/python/tutorials folder of the MXNet repository that you downloaded as part of the MXNet installation.

I ran the notebook locally on my MacBook Pro, only to have it fail at the `import mxnet` statement. After smacking myself on the forehead for making a newbie omission (aided and abetted by confusing documentation), I installed the Python language module with the setup.py script in the python directory of the repository. I used the current-user option,

```
python setup.py develop —user
```

When I restarted the MNIST notebook, it failed at a call to `mx.viz.plot_network`. Alas, the documentation hadn’t mentioned the need to install Graphviz. I tried to install Graphviz with pip, which seemed like the obvious thing to do, but the notebook failed again with a message saying that it couldn’t find the Graphviz executables. I had no idea what was wrong, so I Googled the error message, found many reports of similar problems, and guessed about which of the multiple answers (none of which had feedback) might be correct for my system. After tossing yarrow stalks I reinstalled Graphviz with Homebrew, then the MNIST notebook worked and displayed the network graph.


localhost:8888/notebooks/mnist.ipynb
jupyter mnist Last Checkpoint: 20 minutes ago (unsaved changes)
File Edit View Insert Cell Kernel Help
Python 2
Markdown
CellToolbar

Multilayer Perceptron

A multilayer perceptron contains several fully-connected layers. For a fully-connected layer, assume the input matrix X has size $n \times m$, then it outputs matrix Y with size $n \times k$, where k is often called as the hidden size. This layer has two parameters, the $m \times k$ weight matrix W and the $m \times 1$ bias vector b. It compute the outputs by

$$Y = WX + b.$$

The output of a fully-connected layer is often feed into an activation layer, which performs elemental-wise operations. The widely known function is sigmoid, which has form $f(x) = 1/(1 + e^{-x})$. Nowadays people also use a simpler function called relu: $f(x) = max(0, x)$.

The last fully-connected layer often has the hidden size equals to the number of classes in the dataset. Then we stack a softmax layer, which map the input into a probability score. Again assume the input X has size $n \times m$, and x_i is the i-th row. Then the i-th row of the output is

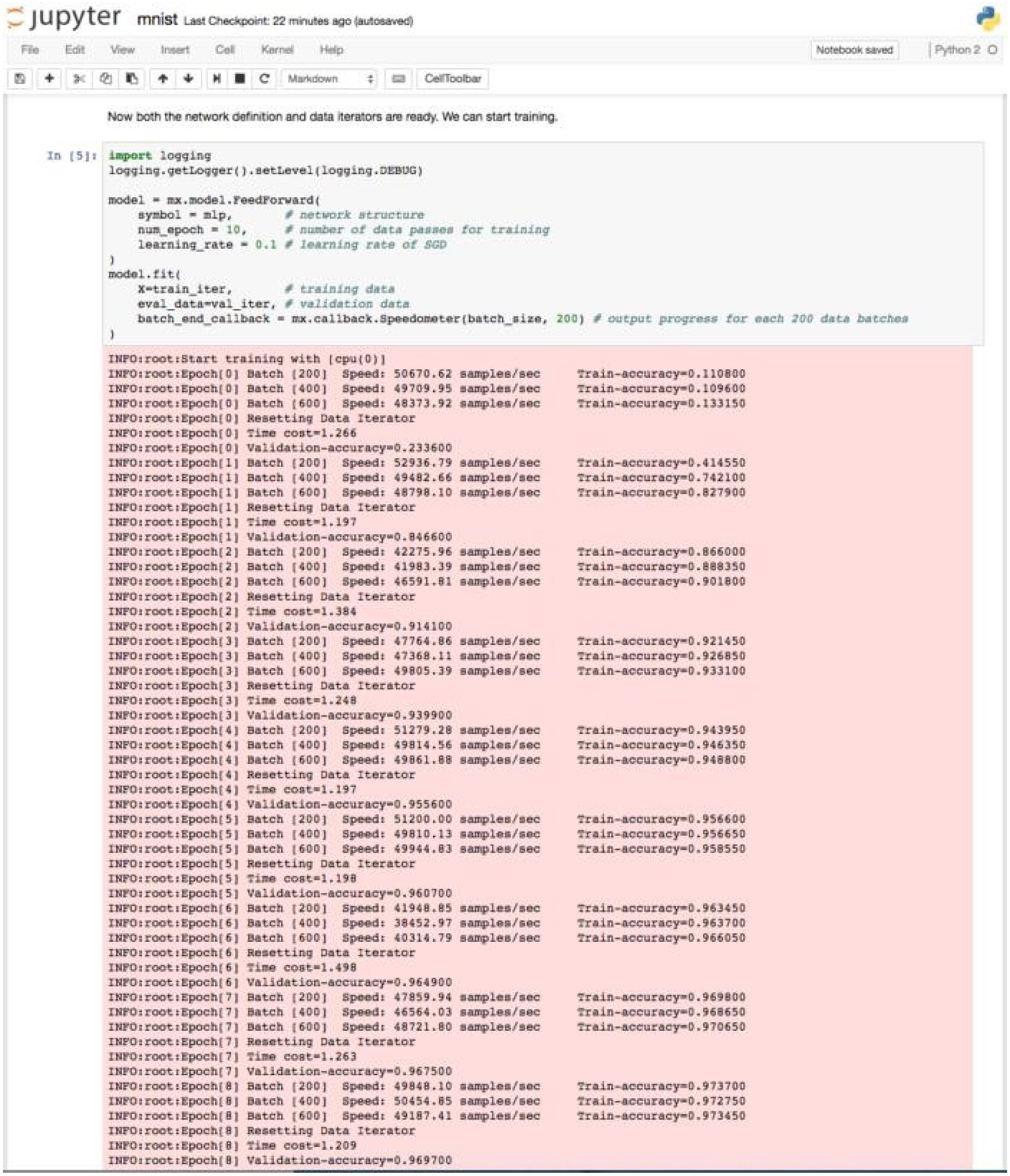
$$\left[\frac{\exp(x_{i1})}{\sum_{j=1}^{m}\exp(x_{ij})}, \ldots, \frac{\exp(x_{im})}{\sum_{j=1}^{m}\exp(x_{ij})}\right]$$

Define the multilayer perceptron in MXNet is straightforward, which has shown as following.

```
In [4]: # Create a place holder variable for the input data
        data = mx.sym.Variable('data')
        # Flatten the data from 4-D shape (batch_size, num_channel, width, height)
        # into 2-D (batch_size, num_channel*width*height)
        data = mx.sym.Flatten(data=data)

        # The first fully-connected layer
        fc1  = mx.sym.FullyConnected(data=data, name='fc1', num_hidden=128)
        # Apply relu to the output of the first fully-connnected layer
        act1 = mx.sym.Activation(data=fc1, name='relu1', act_type="relu")

        # The second fully-connected layer and the according activation function
        fc2  = mx.sym.FullyConnected(data=act1, name='fc2', num_hidden = 64)
        act2 = mx.sym.Activation(data=fc2, name='relu2', act_type="relu")

        # The thrid fully-connected layer, note that the hidden size should be 10, which is the number of unique digits
        fc3  = mx.sym.FullyConnected(data=act2, name='fc3', num_hidden=10)
        # The softmax and loss layer
        mlp  = mx.sym.SoftmaxOutput(data=fc3, name='softmax')

        # We visualize the network structure with output size (the batch_size is ignored.)
        shape = {"data" : (batch_size, 1, 28, 28)}
        mx.viz.plot_network(symbol=mlp, shape=shape)

Out[4]:
```

MXNet comes with a Python tutorial on classifying the MNIST data set with a multilayer perceptron model. Here we have used the Graphviz package to plot the network defined with calls to the MXNet symbolic module.

A few cells later, training the multilayer perceptron model succeeded and took approximately one second per epoch.


jupyter mnist Last Checkpoint: 22 minutes ago (autosaved)
File Edit View Insert Cell Kernel Help
Notebook saved
Python 2
Markdown
CellToolbar

Now both the network definition and data iterators are ready. We can start training.

```
In [5]: import logging
        logging.getLogger().setLevel(logging.DEBUG)

        model = mx.model.FeedForward(
            symbol = mlp,       # network structure
            num_epoch = 10,     # number of data passes for training
            learning_rate = 0.1 # learning rate of SGD
        )
        model.fit(
            X=train_iter,       # training data
            eval_data=val_iter, # validation data
            batch_end_callback = mx.callback.Speedometer(batch_size, 200) # output progress for each 200 data batches
        )
```

```
INFO:root:Start training with [cpu(0)]
INFO:root:Epoch[0] Batch [200]  Speed: 50670.62 samples/sec     Train-accuracy=0.110800
INFO:root:Epoch[0] Batch [400]  Speed: 49709.95 samples/sec     Train-accuracy=0.109600
INFO:root:Epoch[0] Batch [600]  Speed: 48373.92 samples/sec     Train-accuracy=0.133150
INFO:root:Epoch[0] Resetting Data Iterator
INFO:root:Epoch[0] Time cost=1.266
INFO:root:Epoch[0] Validation-accuracy=0.233600
INFO:root:Epoch[1] Batch [200]  Speed: 52936.79 samples/sec     Train-accuracy=0.414550
INFO:root:Epoch[1] Batch [400]  Speed: 49482.66 samples/sec     Train-accuracy=0.742100
INFO:root:Epoch[1] Batch [600]  Speed: 48798.10 samples/sec     Train-accuracy=0.827900
INFO:root:Epoch[1] Resetting Data Iterator
INFO:root:Epoch[1] Time cost=1.197
INFO:root:Epoch[1] Validation-accuracy=0.846600
INFO:root:Epoch[2] Batch [200]  Speed: 42275.96 samples/sec     Train-accuracy=0.866000
INFO:root:Epoch[2] Batch [400]  Speed: 41983.39 samples/sec     Train-accuracy=0.888350
INFO:root:Epoch[2] Batch [600]  Speed: 46591.81 samples/sec     Train-accuracy=0.901800
INFO:root:Epoch[2] Resetting Data Iterator
INFO:root:Epoch[2] Time cost=1.384
INFO:root:Epoch[2] Validation-accuracy=0.914100
INFO:root:Epoch[3] Batch [200]  Speed: 47764.86 samples/sec     Train-accuracy=0.921450
INFO:root:Epoch[3] Batch [400]  Speed: 47368.11 samples/sec     Train-accuracy=0.926850
INFO:root:Epoch[3] Batch [600]  Speed: 49805.39 samples/sec     Train-accuracy=0.933100
INFO:root:Epoch[3] Resetting Data Iterator
INFO:root:Epoch[3] Time cost=1.248
INFO:root:Epoch[3] Validation-accuracy=0.939900
INFO:root:Epoch[4] Batch [200]  Speed: 51279.28 samples/sec     Train-accuracy=0.943950
INFO:root:Epoch[4] Batch [400]  Speed: 49814.56 samples/sec     Train-accuracy=0.946350
INFO:root:Epoch[4] Batch [600]  Speed: 49861.88 samples/sec     Train-accuracy=0.948800
INFO:root:Epoch[4] Resetting Data Iterator
INFO:root:Epoch[4] Time cost=1.197
INFO:root:Epoch[4] Validation-accuracy=0.955600
INFO:root:Epoch[5] Batch [200]  Speed: 51200.00 samples/sec     Train-accuracy=0.956600
INFO:root:Epoch[5] Batch [400]  Speed: 49810.13 samples/sec     Train-accuracy=0.956650
INFO:root:Epoch[5] Batch [600]  Speed: 49944.83 samples/sec     Train-accuracy=0.958550
INFO:root:Epoch[5] Resetting Data Iterator
INFO:root:Epoch[5] Time cost=1.198
INFO:root:Epoch[5] Validation-accuracy=0.960700
INFO:root:Epoch[6] Batch [200]  Speed: 41948.85 samples/sec     Train-accuracy=0.963450
INFO:root:Epoch[6] Batch [400]  Speed: 38452.97 samples/sec     Train-accuracy=0.963700
INFO:root:Epoch[6] Batch [600]  Speed: 40314.79 samples/sec     Train-accuracy=0.966050
INFO:root:Epoch[6] Resetting Data Iterator
INFO:root:Epoch[6] Time cost=1.498
INFO:root:Epoch[6] Validation-accuracy=0.964900
INFO:root:Epoch[7] Batch [200]  Speed: 47859.94 samples/sec     Train-accuracy=0.969800
INFO:root:Epoch[7] Batch [400]  Speed: 46564.03 samples/sec     Train-accuracy=0.968650
INFO:root:Epoch[7] Batch [600]  Speed: 48721.80 samples/sec     Train-accuracy=0.970650
INFO:root:Epoch[7] Resetting Data Iterator
INFO:root:Epoch[7] Time cost=1.263
INFO:root:Epoch[7] Validation-accuracy=0.967500
INFO:root:Epoch[8] Batch [200]  Speed: 49848.10 samples/sec     Train-accuracy=0.973700
INFO:root:Epoch[8] Batch [400]  Speed: 50454.85 samples/sec     Train-accuracy=0.972750
INFO:root:Epoch[8] Batch [600]  Speed: 49187.41 samples/sec     Train-accuracy=0.973450
INFO:root:Epoch[8] Resetting Data Iterator
INFO:root:Epoch[8] Time cost=1.209
INFO:root:Epoch[8] Validation-accuracy=0.969700
```

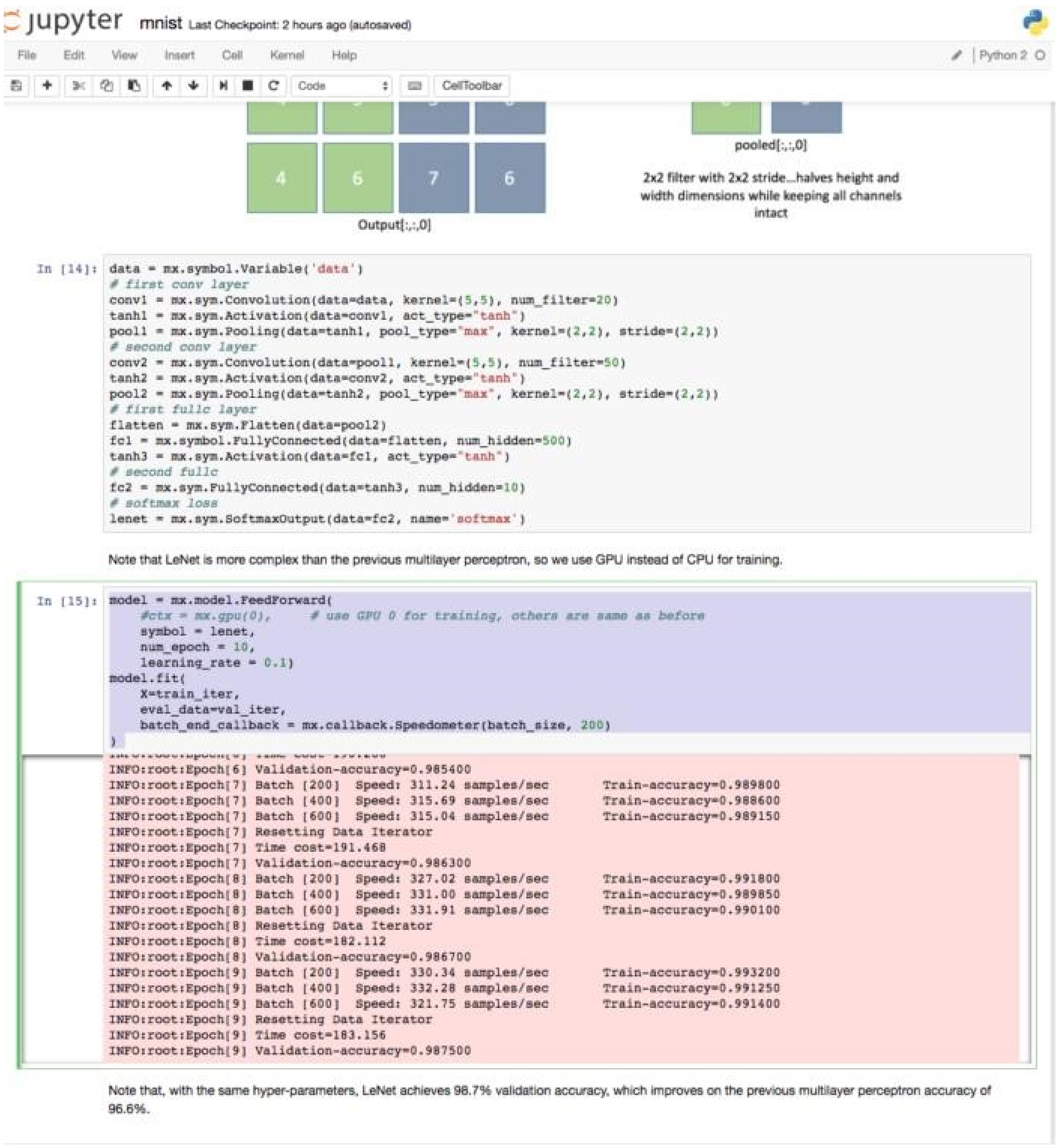
Multilayer perceptron network training with MXNet. As you can see, each epoch took about one second, and after nine epochs the validation accuracy was 96.97 percent.

The tutorial continues with another model, the LeNet Convolutional Neural Network (CNN), which failed with the message “Compile with USE_CUDA=1 to enable GPU usage.” I pretty much knew from working with TensorFlow that compiling for CUDA wouldn’t work on my Mac, but I tried it anyway and got a message that the Nvidia CUDA compiler driver NVCC is not compatible with Xcode 8. (Nvidia has been promising to correct this for at least a month.)

To cover all the bases, I switched the active compiler to Xcode 7.3.1, which is compatible with NVCC but caused compile errors. I switched back to Xcode 8, restarted the notebook, commented out the line setting the GPU context, and ran the CNN on the default CPU:

```
model = mx.model.FeedForward(
    # ctx = mx.gpu(0),
    symbol = lenet,
    num_epoch = 10,
    learning_rate = 0.1)
model.fit(
    X=train_iter,
    eval_data=val_iter,
    batch_end_callback = mx.callback.Speedometer(batch_size, 200)
)
```

That worked, but was quite slow. It took about three minutes per epoch and about half an hour for the entire run. It didn’t seem to be using more than one core, based on the Mac Activity Monitor. Apparently the default MXNet context runs on CPU core 0.


jupyter mnist Last Checkpoint: 2 hours ago (autosaved)
File Edit View Insert Cell Kernel Help
Python 2
Code
CellToolbar
4
6
7
6
Output[:,:,0]
pooled[:,:,0]

2x2 filter with 2x2 stride...halves height and width dimensions while keeping all channels intact

```
In [14]: data = mx.symbol.Variable('data')
         # first conv layer
         conv1 = mx.sym.Convolution(data=data, kernel=(5,5), num_filter=20)
         tanh1 = mx.sym.Activation(data=conv1, act_type="tanh")
         pool1 = mx.sym.Pooling(data=tanh1, pool_type="max", kernel=(2,2), stride=(2,2))
         # second conv layer
         conv2 = mx.sym.Convolution(data=pool1, kernel=(5,5), num_filter=50)
         tanh2 = mx.sym.Activation(data=conv2, act_type="tanh")
         pool2 = mx.sym.Pooling(data=tanh2, pool_type="max", kernel=(2,2), stride=(2,2))
         # first fullc layer
         flatten = mx.sym.Flatten(data=pool2)
         fc1 = mx.symbol.FullyConnected(data=flatten, num_hidden=500)
         tanh3 = mx.sym.Activation(data=fc1, act_type="tanh")
         # second fullc
         fc2 = mx.sym.FullyConnected(data=tanh3, num_hidden=10)
         # softmax loss
         lenet = mx.sym.SoftmaxOutput(data=fc2, name='softmax')
```

Note that LeNet is more complex than the previous multilayer perceptron, so we use GPU instead of CPU for training.

```
In [15]: model = mx.model.FeedForward(
             #ctx = mx.gpu(0),     # use GPU 0 for training, others are same as before
             symbol = lenet,
             num_epoch = 10,
             learning_rate = 0.1)
         model.fit(
             X=train_iter,
             eval_data=val_iter,
             batch_end_callback = mx.callback.Speedometer(batch_size, 200)
         )
```

```
INFO:root:Epoch[6] Validation-accuracy=0.985400
INFO:root:Epoch[7] Batch [200]  Speed: 311.24 samples/sec      Train-accuracy=0.989800
INFO:root:Epoch[7] Batch [400]  Speed: 315.69 samples/sec      Train-accuracy=0.988600
INFO:root:Epoch[7] Batch [600]  Speed: 315.04 samples/sec      Train-accuracy=0.989150
INFO:root:Epoch[7] Resetting Data Iterator
INFO:root:Epoch[7] Time cost=191.468
INFO:root:Epoch[7] Validation-accuracy=0.986300
INFO:root:Epoch[8] Batch [200]  Speed: 327.02 samples/sec      Train-accuracy=0.991800
INFO:root:Epoch[8] Batch [400]  Speed: 331.00 samples/sec      Train-accuracy=0.989850
INFO:root:Epoch[8] Batch [600]  Speed: 331.91 samples/sec      Train-accuracy=0.990100
INFO:root:Epoch[8] Resetting Data Iterator
INFO:root:Epoch[8] Time cost=182.112
INFO:root:Epoch[8] Validation-accuracy=0.986700
INFO:root:Epoch[9] Batch [200]  Speed: 330.34 samples/sec      Train-accuracy=0.993200
INFO:root:Epoch[9] Batch [400]  Speed: 332.28 samples/sec      Train-accuracy=0.991250
INFO:root:Epoch[9] Batch [600]  Speed: 321.75 samples/sec      Train-accuracy=0.991400
INFO:root:Epoch[9] Resetting Data Iterator
INFO:root:Epoch[9] Time cost=183.156
INFO:root:Epoch[9] Validation-accuracy=0.987500
```

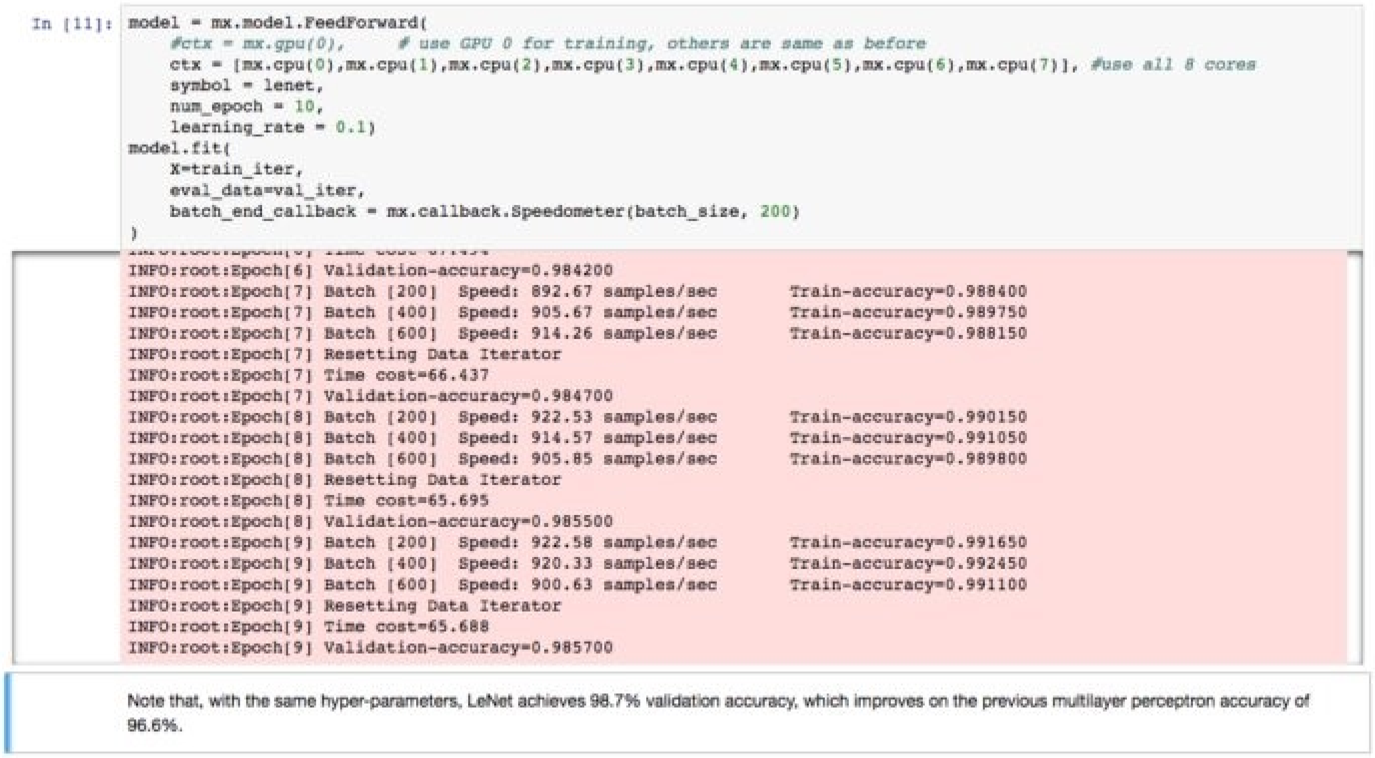
Note that, with the same hyper-parameters, LeNet achieves 98.7% validation accuracy, which improves on the previous multilayer perceptron accuracy of 96.6%.

LeNet convolutional model training on MNIST data. The time per epoch on one core was about three minutes, and the validation accuracy after 10 epochs was 98.75 percent.

I naively tried to run the LeNet training on all eight cores, passing a list for the context:

```
ctx = [mx.cpu(0), mx.cpu(1), mx.cpu(2), mx.cpu(3), mx.cpu(4), mx.cpu(5), mx.cpu(6),
mx.cpu(7)]
```

That sped up the training, but only by a factor of three, and the convergence wasn't as good running in parallel:



```
In [11]: model = mx.model.FeedForward(
             #ctx = mx.gpu(0),     # use GPU 0 for training, others are same as before
             ctx = [mx.cpu(0),mx.cpu(1),mx.cpu(2),mx.cpu(3),mx.cpu(4),mx.cpu(5),mx.cpu(6),mx.cpu(7)], #use all 8 cores
             symbol = lenet,
             num_epoch = 10,
             learning_rate = 0.1)
         model.fit(
             X=train_iter,
             eval_data=val_iter,
             batch_end_callback = mx.callback.Speedometer(batch_size, 200)
         )
```

```
INFO:root:Epoch[6] Validation-accuracy=0.984200
INFO:root:Epoch[7] Batch [200]  Speed: 892.67 samples/sec       Train-accuracy=0.988400
INFO:root:Epoch[7] Batch [400]  Speed: 905.67 samples/sec       Train-accuracy=0.989750
INFO:root:Epoch[7] Batch [600]  Speed: 914.26 samples/sec       Train-accuracy=0.988150
INFO:root:Epoch[7] Resetting Data Iterator
INFO:root:Epoch[7] Time cost=66.437
INFO:root:Epoch[7] Validation-accuracy=0.984700
INFO:root:Epoch[8] Batch [200]  Speed: 922.53 samples/sec       Train-accuracy=0.990150
INFO:root:Epoch[8] Batch [400]  Speed: 914.57 samples/sec       Train-accuracy=0.991050
INFO:root:Epoch[8] Batch [600]  Speed: 905.85 samples/sec       Train-accuracy=0.989800
INFO:root:Epoch[8] Resetting Data Iterator
INFO:root:Epoch[8] Time cost=65.695
INFO:root:Epoch[8] Validation-accuracy=0.985500
INFO:root:Epoch[9] Batch [200]  Speed: 922.58 samples/sec       Train-accuracy=0.991650
INFO:root:Epoch[9] Batch [400]  Speed: 920.33 samples/sec       Train-accuracy=0.992450
INFO:root:Epoch[9] Batch [600]  Speed: 900.63 samples/sec       Train-accuracy=0.991100
INFO:root:Epoch[9] Resetting Data Iterator
INFO:root:Epoch[9] Time cost=65.688
INFO:root:Epoch[9] Validation-accuracy=0.985700
```

Note that, with the same hyper-parameters, LeNet achieves 98.7% validation accuracy, which improves on the previous multilayer perceptron accuracy of 96.6%.

Using eight CPU cores, we only get a 3X speedup over using one core, and the final validation accuracy is not quite as good.

I tried this again with only four cores:

```
ctx = [mx.cpu(n) for n in range(4)], #use 4 cores
```

That gave me roughly the same training speed as eight cores, with a final validation accuracy closer to the one-core result. I don't know why exactly, but I think the limit on core usage has to do with the number of layers in the model. The variation in results does bring home the fact that ML training is a random process. The subject of running MXNet in parallel is documented, but I haven't found an answer to the question of optimum parallelism.

You've probably gotten the idea by now that that the MXNet documentation leaves something to be desired. I hope that will be one of the items that Amazon helps to improve, now that it has adopted the project.

MXNet tutorials and models

In addition to MNIST digit classification, the computer vision MXNet tutorials in Python cover image classification and segmentation using convolutional neural networks (CNN); object detection using Faster R-CNN; neural art; and classification of ImageNet using a deep CNN. In R, only two of these have been implemented; in Scala, only the MNIST tutorial is shown.

For natural language processing (NLP), the Python tutorials are for character-level long short-term memory (LSTM); text classification using a CNN; and noise-contrastive estimation (NCE) loss with an LSTM model. In R, there's a different character language model. There is no Scala example for NLP.

There are two Python examples for speech recognition; a Python generative adversarial network trained on MNIST; three Python unsupervised machine learning tutorials; and one R supervised machine learning tutorial. There are additional models in the example folder of the MXNet repository, mostly in Python.

Overall, I have to agree with the authors of MXNet that their deep learning framework is similar to TensorFlow in many respects. As for differences, while MXNet lacks the visual debugging available for TensorFlow in TensorBoard, it offers an imperative language for tensor calculations that TensorFlow lacks.

The MXNet "model zoo" is not yet as complete as TensorFlow's, and the MXNet documentation needs some work. The multi-GPU scaling performance reported by Amazon is exciting, however. Given the complexity and computational cost of training deep learning models, this might be enough to attract deep learning practitioners to MXNet even in its present state.

InfoWorld Scorecard	**Models and algorithms (25%)**	**Ease of development (25%)**	**Documentation (20%)**	**Performance (20%)**	**Ease of deployment (10%)**	**Overall Score (100%)**
MXNet v0.7	8	8	7	10	8	8.2

At a Glance

- **MXNet v0.7**

  InfoWorld Rating

  Learn more

  on Distributed Machine Learning...

  This portable, scalable deep learning library combines symbolic declaration of neural network geometries with imperative programming of tensor operations.

  Pros

  - Scales to multiple GPUs across multiple hosts with scaling efficiency of 85 percent
  - Excellent development speed and programmability
  - Excellent portability
  - Supports Python, R, Scala, Julia, and C++
  - Allows you to mix symbolic and imperative programming flavors

  Cons

  - Documentation still feels unfinished
  - Few examples for languages other than Python