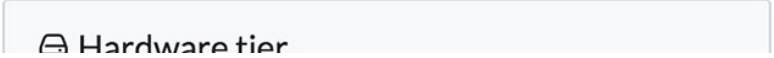
Faster deep learning with GPUs and Theano

Posted by Manojit Nandi

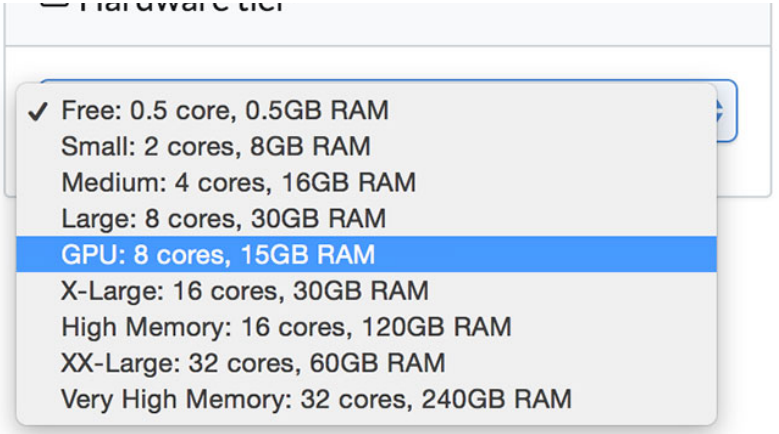
04 AUG 2015

Python, Data Science

Domino recently added support for GPU



instances. To celebrate this release, I will show you how to:



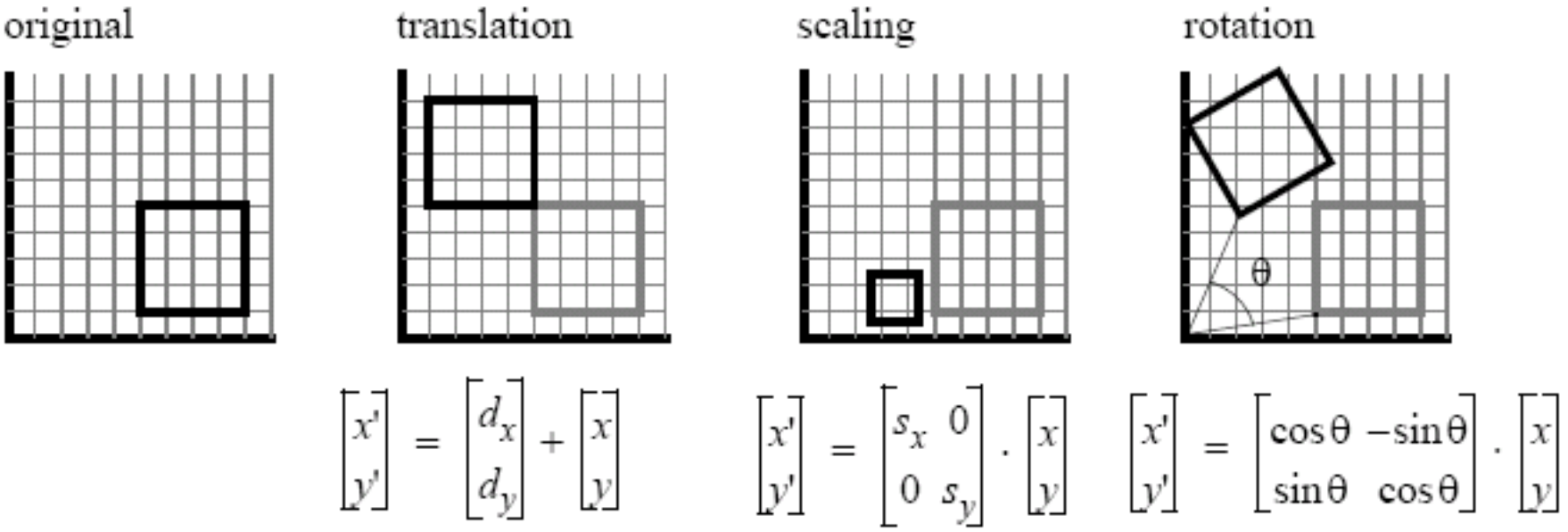
- Configure the Python library Theano to use the GPU for computation.
- Build and train neural networks in Python.

Using the GPU, I'll show that we can train deep belief networks up to 15x faster than using just the CPU, cutting training time down from hours to minutes. You can see my code, experiments, and results on Domino.

Why are GPUs useful?

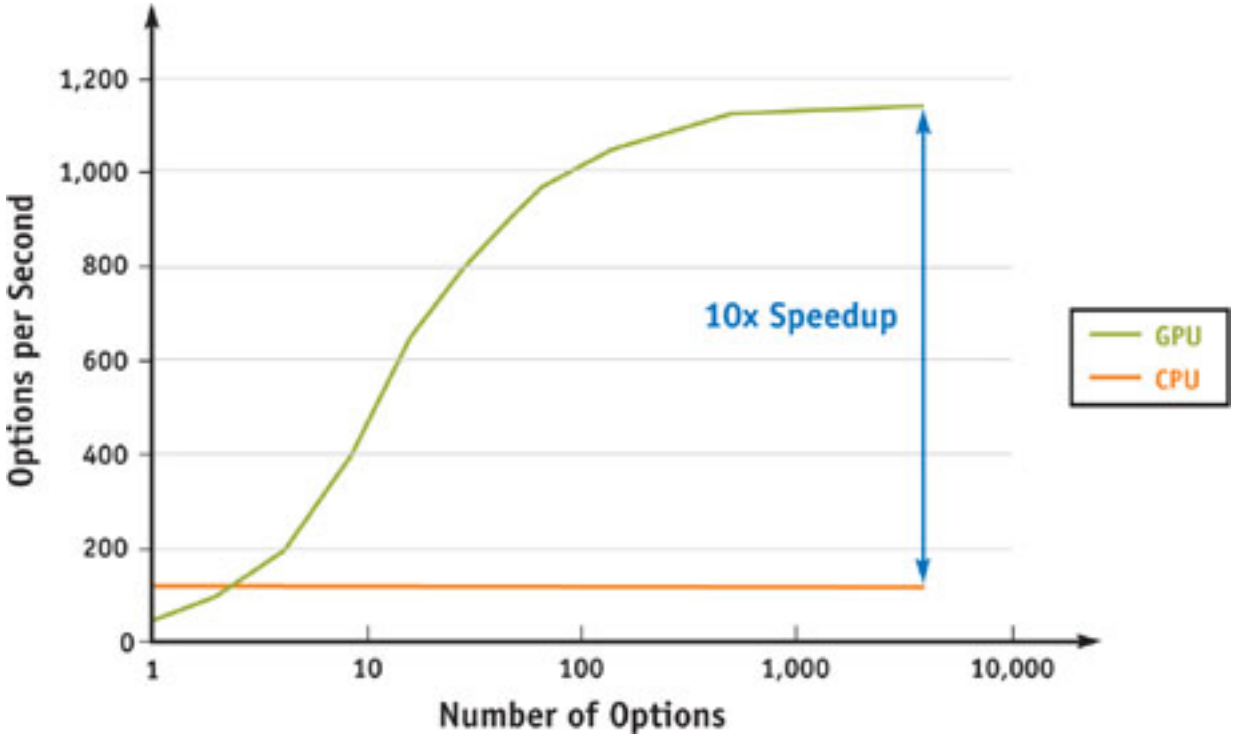
When you think of high-performance graphics cards, data science may not be the first thing that comes to mind. However, computer graphics and data science have one important thing in common: matrices!

Images, videos, and other graphics are represented as matrices, and when you perform a certain operation, such as a camera rotation or a zoom in effect, all you are doing is applying some mathematical transformation to a matrix.



Source: http://www.pling.org.uk/cs/cgvimg/transformations.png

What this means is that GPUs, compared to CPUs (Central Processing Unit), are more specialized at performing matrix operations and other advanced mathematical transformations. In some cases, we see a 10x speedup in an algorithm when it runs on the GPU.

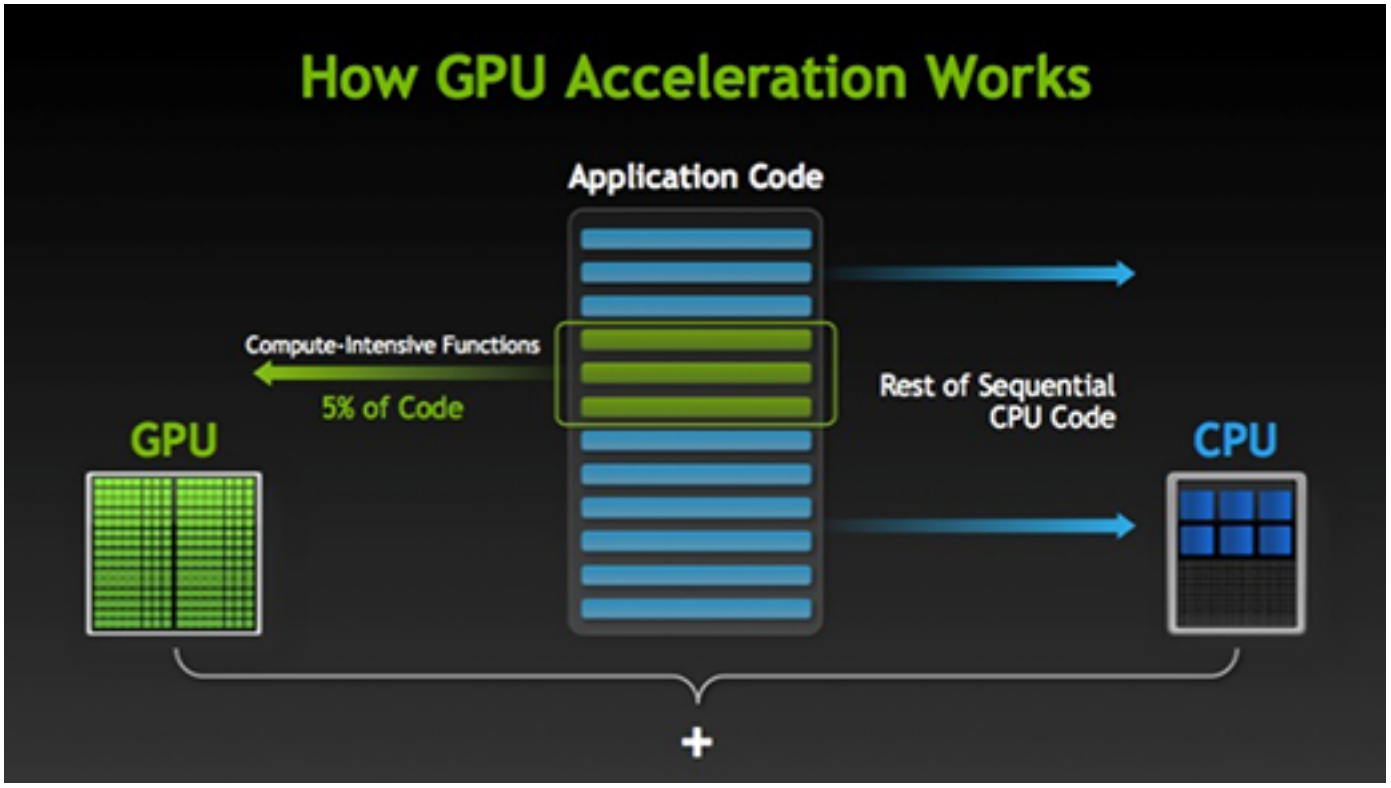


Source: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter45.html

GPU-based computation have been employed in a wide variety of scientific applications, from genomic to epidemiology

Recently, there has been a rise in GPU-accelerated algorithms in machine learning thanks to the rising popularity of deep learning algorithms. Deep Learning is a collection of algorithms for training neural network-based models for various problems in machine learning. Deep Learning algorithms involve computationally intensive methods, such as convolutions, Fourier Transforms, and other matrix-based operations which GPUs are well-suited for

computing. The computationally intensive functions, which make up about 5% of the code, are run on the GPU, and the remaining code is run on the CPU.



Source: http://www.nvidia.com/docs/IO/143716/how-gpu-acceleration-works.png

With the recent advances in GPU performance and support for GPUs in common libraries, I recommend anyone interested in deep learning get ahold of a GPU.

Now that I have thoroughly motivated the use of GPUs, let's see how they can be used to train neural networks in Python.

Deep Learning in Python

The most popular library in Python to implement neural networks is Theano. However, Theano is not strictly a neural network library, but rather a Python library that makes it possible to implement a wide variety of mathematical abstractions. Because of this, Theano has a high learning curve, so I will be using two neural network libraries built on top of Theano that have a more gentle learning curve.

The first library is Lasagne. This library provides a nice abstraction that allows you to construct each layer of the neural network, and then stack the layers on top of each other to construct the full model. While this is nicer than Theano, constructing each layer and then appending them on top of one another becomes tedious, so we'll be using the Nolearn library which provides a Scikit-Learn style API over Lasagne to easily construct neural networks with multiple layers.

Because these libraries do not come default with Domino's hardware, you need to create a requirements.txt with the following text:

```
-r https://raw.githubusercontent.com/dnouri/nolearn/master/requirements.txt
git+https://github.com/dnouri/nolearn.git@master#egg=nolearn==0.7.git
```

Setting up Theano

Now, before we can import Lasagne and Nolearn, we need to configure Theano, so that it can utilize the GPU hardware. To do this, we create a .theanorc file in our project directory with the following contents:

```
[global]
device = gpu
floatX = float32

[nvcc]
fastmath = True
```

The .theanorc file must be placed in the home directory. On your local machine this could be done manually, but we cannot access the home directory of Domino's machine, so we will move the file to the home directory using the following code:

```
import os
import shutil

destfile = "/home/ubuntu/.theanorc"
open(destfile, 'a').close()
shutil.copyfile(".theanorc", destfile)
```

The above code creates an empty .theanorc file in the home directory and then copy the contents of the .theanorc file in our project directory into the file in the home directory.

After changing the hardware tier to GPU, we can test to see if Theano detects the GPU using the test code provided in Theano's documentation.

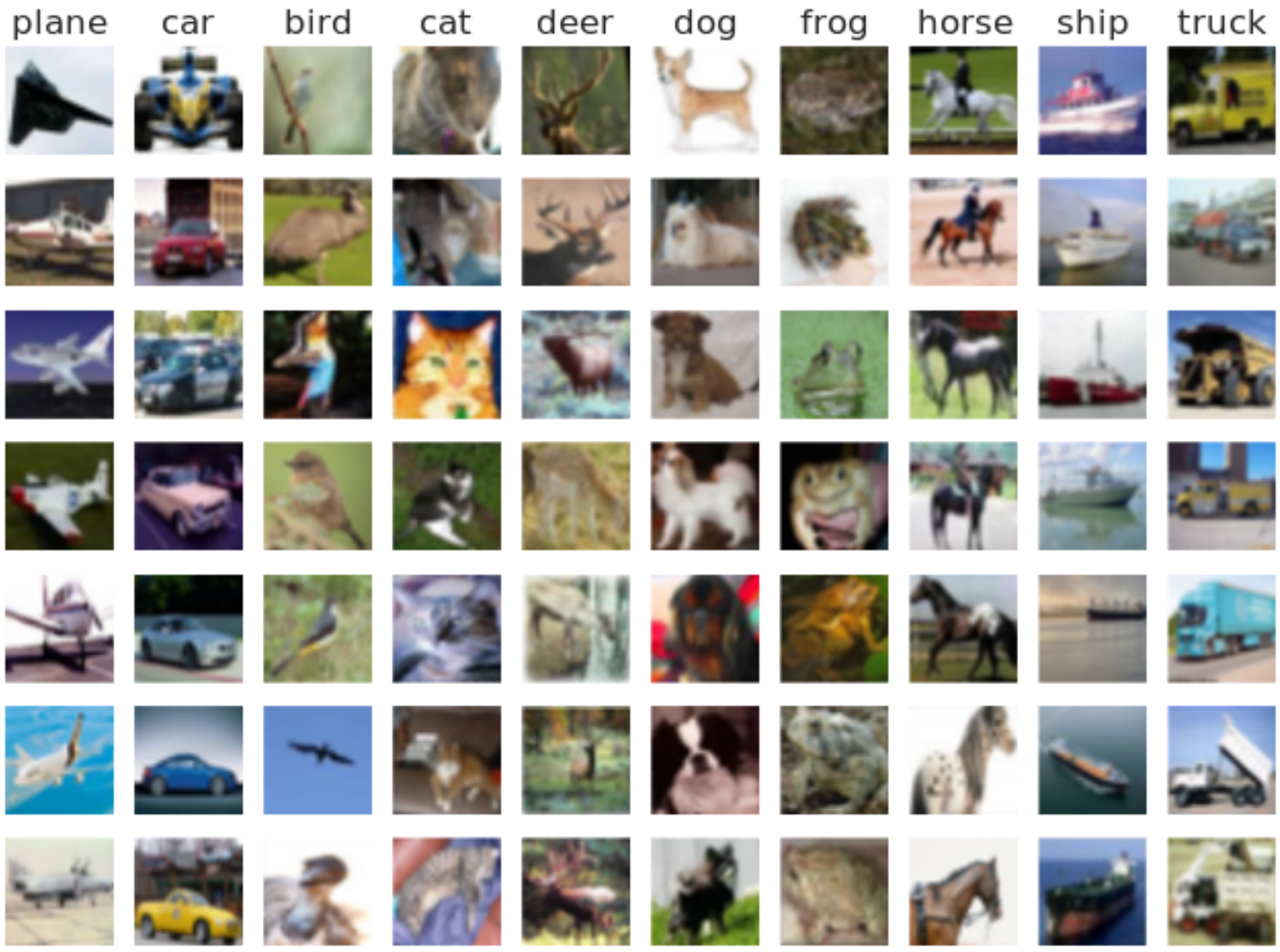
```
import os
import shutil

destfile = "/home/ubuntu/.theanorc"
open(destfile, 'a').close()
shutil.copyfile(".theanorc", destfile)

from theano import function, config, shared, sandbox
import theano.tensor as T
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], T.exp(x))
print f.maker.fgraph.toposort()
```

```
t0 = time.time()
for i in xrange(iters):
    r = f()
t1 = time.time()
print 'Looping %d times took' % iters, t1 - t0, 'seconds'
print 'Result is', r
if numpy.any([isinstance(x.op, T.Elemwise) for x in f.maker.fgraph.toposort()]):
    print 'Used the cpu'
else:
    print 'Used the gpu'
```

If Theano detects the GPU, the above function should take about 0.7 seconds to run and will print 'Used the gpu'. Otherwise, it will take 2.6 seconds to run and print 'Used the cpu'. If it outputs this, then you forgot to change the hardware tier to GPU.

The Dataset

For this project, we'll be using the CIFAR-10 image dataset containing 60,000 32x32 colored images from 10 different classes.



Fortunately, the data come in a pickled format, so we can load the data using helper functions

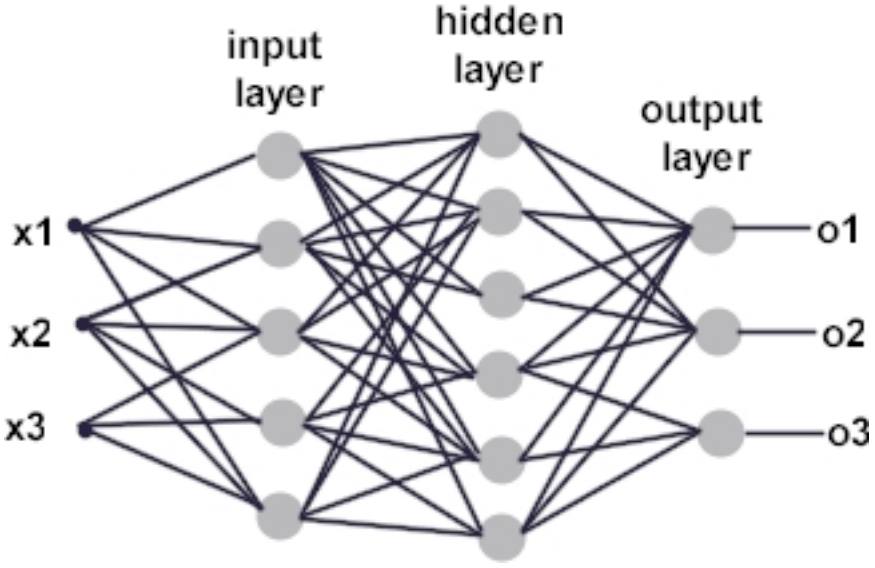
to load each file into NumPy arrays to produce a training set (Xtr), training labels (Ytr), testing set (Xte), and testing labels (Yte). Credit for the following code goes to Stanford's CS231n course staff.

```
import cPickle as pickle
import numpy as np
import os

def load_CIFAR_file(filename):
    '''Load a single file of CIFAR'''
    with open(filename, 'rb') as f:
        datadict= pickle.load(f)
        X = datadict['data']
        Y = datadict['labels']
        X = X.reshape(10000, 3, 32, 32).transpose(0,2,3,1).astype('float32')
        Y = np.array(Y).astype('int32')
        return X, Y


def load_CIFAR10(directory):
    '''Load all of CIFAR'''
    xs = []
    ys = []
    for k in range(1,6):
        f = os.path.join(directory, "data_batch_%d" % k)
        X, Y = load_CIFAR_file(f)
        xs.append(X)
        ys.append(Y)
```

```
    Xtr = np.concatenate(xs)
    Ytr = np.concatenate(ys)
    Xte, Yte = load_CIFAR_file(os.path.join(directory, 'test_batch'))
    return Xtr, Ytr, Xte, Yte
```

Multi-Layered Perceptron

A multi-layered perceptron is one of the most simple neural network models. The model consists of an input layer for the data, a hidden layer to apply some mathematical transformation, and an output layer to produce a label (either categorical for classification or continuous for regression).



Source: http://dms.irb.hr/tutorial/tut_nnets_short.php

Before we can use the training data, we need to grayscale it and flatten it into a two-dimensional matrix. In addition, we will divide each value by 255 and subtract 0.5. When we grayscale the image, we convert each (R,G,B) tuple in a float value between 0 and 255. By dividing by 255, we normalize the grayscale value to the interval [0,1]. Next, we subtract 0.5 to map the values to the interval [-0.5, 0.5]. Now, each image is represented by a 1024-dimensional array where each value is between -0.5 and 0.5. It's common practice to standardize your input features to the interval [-1, 1] when training classification networks.

```
X_train_flat = np.dot(X_train[...,:3], [0.299, 0.587,
0.114]).reshape(X_train.shape[0],-1).astype(np.float32)
X_train_flat = (X_train_flat/255.0)-0.5
X_test_flat = np.dot(X_test[...,:3], [0.299, 0.587,
0.114]).reshape(X_test.shape[0],-1).astype(np.float32)
X_test_flat = (X_test_flat/255.0)-.5
```

Using Nolearn's API, we can easily create a multi-layered perceptron with an input, hidden, and output layer. The hidden_num_units = 100 means our hidden layer has 100 neurons, and the output_num_units = 10 means our output layer has 10 neurons, one for each of the label. Before outputting, the network applies a softmax function to determine the most probable label. If The network is trained for 50 epochs and with verbose = 1, the model prints out the result of each training epoch and how long the epoch took.

```
net1 = NeuralNet(
    layers = [
        ('input', layers.InputLayer),
        ('hidden', layers.DenseLayer),
        ('output', layers.DenseLayer),
        ],
        #layers parameters:
        input_shape = (None, 1024),
        hidden_num_units = 100,
        output_nonlinearity = softmax,
        output_num_units = 10,

        #optimization parameters:
        update = nesterov_momentum,
        update_learning_rate = 0.01,
        update_momentum = 0.9,

        regression = False,
        max_epochs = 50,
        verbose = 1,
        )
```

As a side remark, this API makes it easy to build deep networks. If we wanted to add a second hidden layer, all we would have to do is add it to the layers parameter and specify how many units in the new layer.

```
net1 = NeuralNet(
    layers = [
        ('input', layers.InputLayer),
        ('hidden1', layers.DenseLayer),
        ('hidden2', layers.DenseLayer), #Added Layer Here
        ('output', layers.DenseLayer),
        ],
        #layers parameters:
        input_shape = (None, 1024),
        hidden1_num_units = 100,
        hidden2_num_units = 100, #Added Layer Params Here
```

Now, as I mentioned earlier about Nolearn's Scikit-Learn style API, we can fit the Neural Network with the fit method.

```
net1.fit(X_train_flat, y_train)
```

When the network is trained on the GPU hardware, we see each training epoch usually takes 0.5 seconds.

```
  DenseLayer              (None, 10)              produces       10 outputs
  DenseLayer              (None, 100)             produces      100 outputs
  InputLayer              (None, 1024)            produces     1024 outputs

 Epoch  |  Train loss  |  Valid loss  |  Train / Val  |  Valid acc  |  Dur
--------|--------------|--------------|---------------|-------------|-------
```

1	1.973778	1.857211	1.062764	34.94%	0.5s
2	1.813298	1.786763	1.014851	37.18%	0.5s
3	1.739768	1.740810	0.999401	39.03%	0.5s
4	1.682765	1.707125	0.985731	40.40%	0.5s
5	1.637490	1.683162	0.972865	41.57%	0.5s
6	1.599529	1.664906	0.960732	42.06%	0.5s
7	1.566990	1.651795	0.948659	42.35%	0.5s
8	1.538967	1.643163	0.936588	42.79%	0.5s
9	1.513615	1.635668	0.925381	42.88%	0.5s
10	1.490286	1.632236	0.913033	42.96%	0.5s
11	1.468767	1.628404	0.901967	43.02%	0.5s
12	1.448626	1.626985	0.890374	43.30%	0.5s
13	1.429661	1.628134	0.878098	43.33%	0.5s
14	1.411795	1.629462	0.866417	43.23%	0.5s
15	1.394168	1.630591	0.855008	43.22%	0.5s
16	1.377811	1.633130	0.843663	43.31%	0.5s
17	1.361821	1.636734	0.832036	43.28%	0.5s
18	1.346741	1.640999	0.820683	43.36%	0.5s
19	1.332388	1.646839	0.809058	43.21%	0.5s
20	1.318268	1.652022	0.797972	43.20%	0.5s
21	1.304940	1.656016	0.788000	43.21%	0.5s
22	1.291717	1.662055	0.777181	43.06%	0.5s
23	1.279134	1.669582	0.766140	42.83%	0.5s
24	1.266824	1.676214	0.755765	42.97%	0.5s
25	1.254751	1.682640	0.745704	42.89%	0.5s
26	1.243170	1.690090	0.735564	42.76%	0.5s
27	1.231465	1.697184	0.725593	42.67%	0.5s
28	1.220072	1.704231	0.715907	42.81%	0.5s
29	1.208638	1.713412	0.705399	42.41%	0.5s
30	1.197366	1.721794	0.695417	42.47%	0.5s
31	1.186738	1.730339	0.685841	42.40%	0.5s
32	1.175844	1.739060	0.676138	42.55%	0.5s
33	1.165466	1.747675	0.666867	42.56%	0.5s
34	1.155507	1.757438	0.657495	42.28%	0.5s

35	1.145337	1.765907	0.648583	42.42%	0.5s
36	1.135607	1.774428	0.639985	42.28%	0.5s
37	1.125634	1.785256	0.630517	42.18%	0.5s
38	1.116582	1.793918	0.622427	42.12%	0.5s
39	1.107208	1.803488	0.613926	42.13%	0.5s
40	1.098400	1.813203	0.605779	42.15%	0.5s
41	1.089075	1.822381	0.597611	42.13%	0.5s
42	1.079767	1.833795	0.588815	42.09%	0.5s
43	1.070845	1.842143	0.581304	42.32%	0.5s
44	1.062722	1.853621	0.573322	42.08%	0.5s
45	1.053891	1.865944	0.564803	42.01%	0.5s
46	1.045613	1.878719	0.556556	41.66%	0.5s
47	1.037189	1.887940	0.549376	41.65%	0.5s
48	1.029303	1.897695	0.542396	41.77%	0.5s
49	1.021136	1.909518	0.534761	41.68%	0.5s
50	1.013229	1.922584	0.527014	41.74%	0.5s

On the other hand, when Domino's hardware is set to XX-Large (32 core, 60 GB RAM), each training epoch takes usually takes 1.3 seconds.

```
  DenseLayer              (None, 10)              produces      10 outputs
  DenseLayer              (None, 100)             produces     100 outputs
  InputLayer              (None, 1024)            produces    1024 outputs

 Epoch   |  Train loss  |  Valid loss  |  Train / Val  |  Valid acc  |  Dur
--------|--------------|--------------|---------------|-------------|-------
     1   |    1.978579  |    1.858524  |     1.064597  |     34.93%  |  1.3s
     2   |    1.812780  |    1.781771  |     1.017404  |     37.94%  |  1.3s
     3   |    1.738579  |    1.739062  |     0.999723  |     39.29%  |  1.3s
     4   |    1.683223  |    1.708612  |     0.985140  |     40.61%  |  1.3s
```

5	1.639533	1.686888	0.971928	41.18%	1.3s
6	1.602723	1.670663	0.959333	41.76%	1.3s
7	1.571114	1.659067	0.946986	42.36%	1.3s
8	1.543218	1.650633	0.934925	42.78%	1.3s
9	1.517887	1.643566	0.923532	42.87%	1.3s
10	1.494585	1.640513	0.911048	43.03%	1.3s
11	1.472859	1.639615	0.898296	43.21%	1.3s
12	1.452628	1.639248	0.886155	43.29%	1.3s
13	1.433942	1.639807	0.874458	43.30%	1.3s
14	1.416047	1.642280	0.862245	43.36%	1.3s
15	1.398969	1.645423	0.850219	43.26%	1.3s
16	1.382327	1.650609	0.837465	43.12%	1.3s
17	1.366932	1.653975	0.826453	43.20%	1.3s
18	1.351511	1.658945	0.814681	43.44%	1.3s
19	1.337196	1.663227	0.803977	43.56%	1.3s
20	1.323045	1.667842	0.793268	43.41%	1.3s
21	1.309340	1.673240	0.782518	43.42%	1.3s
22	1.295969	1.680451	0.771203	43.42%	1.3s
23	1.282977	1.685236	0.761304	43.45%	1.3s
24	1.269948	1.692756	0.750225	43.31%	1.3s
25	1.257807	1.700985	0.739458	43.24%	1.3s
26	1.245719	1.710145	0.728429	42.95%	1.3s
27	1.233884	1.716744	0.718735	42.91%	1.3s
28	1.222683	1.725197	0.708721	42.90%	1.3s
29	1.211569	1.732954	0.699135	42.97%	1.3s
30	1.200041	1.740839	0.689346	42.80%	1.3s
31	1.189644	1.749726	0.679903	42.83%	1.3s
32	1.179003	1.757149	0.670975	42.79%	1.3s
33	1.168253	1.765421	0.661742	42.69%	1.3s
34	1.158188	1.775509	0.652313	42.46%	1.3s
35	1.148412	1.787519	0.642461	42.42%	1.3s
36	1.138447	1.796984	0.633532	42.44%	1.3s
37	1.129187	1.805471	0.625425	42.65%	1.3s
38	1.119788	1.817147	0.616234	42.50%	1.3s

39	1.110091	1.826464	0.607781	42.56%	1.3s
40	1.100841	1.834480	0.600083	42.42%	1.3s
41	1.091596	1.845204	0.591586	42.42%	1.3s
42	1.082325	1.854293	0.583686	42.27%	1.3s
43	1.073911	1.864415	0.576004	42.28%	1.3s
44	1.064921	1.875343	0.567854	42.11%	1.3s
45	1.056440	1.885622	0.560261	42.15%	1.3s
46	1.047966	1.896729	0.552512	42.04%	1.3s
47	1.039865	1.908653	0.544816	41.98%	1.3s
48	1.031394	1.921710	0.536707	41.93%	1.3s
49	1.023448	1.934420	0.529072	41.91%	1.3s
50	1.015672	1.946204	0.521873	41.75%	1.3s

By training this network on the GPU, we see roughly a 3x speedup in training the network. As expected, both the GPU-trained network and the CPU-trained network yield similar results. Both yield about a similar validation accuracy of 41% and similar training loss.

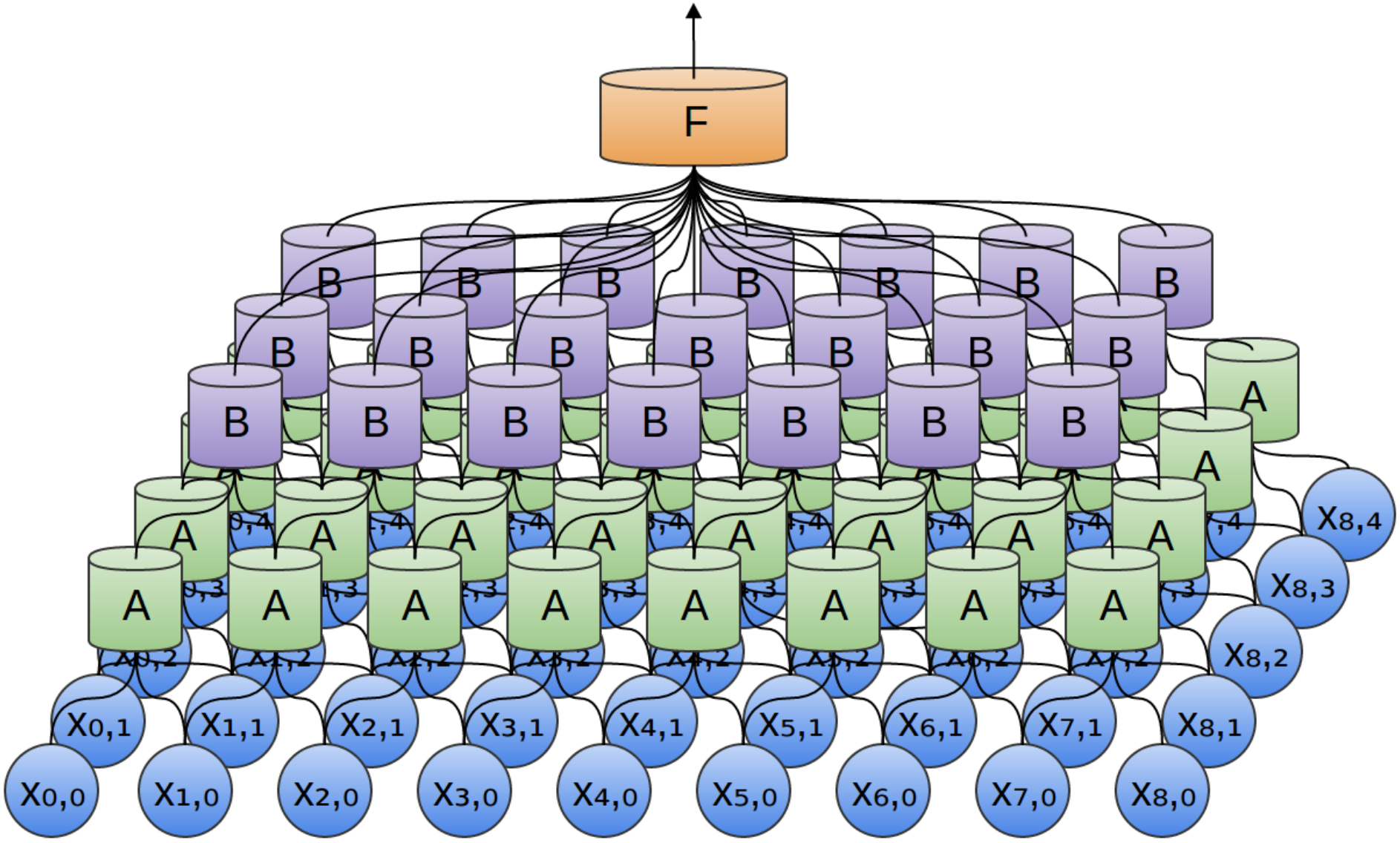
We can test the network on the testing data:

```
y_pred1 = net1.predict(X_test_flat)
print "The accuracy of this network is: %0.2f" % (y_pred1 == y_test).mean()
```

And we achieve an accuracy of 41% on the testing data.

Convolutional Networks

A convolutional neural network is a more complex neural network architecture in which neurons in a layer are connected to a subset of neurons from the previous layer. As a result, convolutions are used to pool the outputs from each subset.



Source: http://colah.github.io/posts/2014-07-Conv-Nets-Modular/

Convolutional Neural Networks are popular in industry and Kaggle Competitions due to their flexibility to learn different problems and scalability.

Once again, before we can build our convolutional neural network, we have to first grayscale and transform the data. This time we will keep the images in their 32x32 shape. I have modified the order of the rows of the matrix, so each image is now represented as (color, x, y). Once again, I divide the features by 255, and subtract by 0.5 to map the features within the interval (-1, 1).

```
X_train_2d = np.dot(X_train[...,:3], [0.299, 0.587,
0.114]).reshape(-1,1,32,32).astype(np.float32)
X_train_2d = (X_train_2d/255.0)-0.5
X_test_2d = np.dot(X_test[...,:3], [0.299, 0.587,
0.114]).reshape(-1,1,32,32).astype(np.float32)
X_train_2d = (X_train_2d/255.0)-0.5
```

Now we can construct the Convolutional Neural Network. The Network consists of the input layer, 3 convolution layers, 3 2x2 pooling layers, a 200-neuron hidden layer, and finally the output layer.

```
net2 = NeuralNet(
```

```
    layers = [
        ('input', layers.InputLayer),
        ('conv1', layers.Conv2DLayer),
        ('pool1', layers.MaxPool2DLayer),
        ('conv2', layers.Conv2DLayer),
        ('pool2', layers.MaxPool2DLayer),
        ('conv3', layers.Conv2DLayer),
        ('pool3', layers.MaxPool2DLayer),
        ("hidden4", layers.DenseLayer),
        ("output", layers.DenseLayer),
        ],
        #layer parameters:
        input_shape = (None, 1, 32, 32),
        conv1_num_filters = 16, conv1_filter_size = (3, 3), pool1_pool_size = (2,2),
        conv2_num_filters = 32, conv2_filter_size = (2, 2) , pool2_pool_size =  (2,2),
        conv3_num_filters = 64, conv3_filter_size = (2, 2), pool3_pool_size = (2,2),
        hidden4_num_units = 200,
        output_nonlinearity = softmax,
        output_num_units = 10,

        #optimization parameters:
        update = nesterov_momentum,
        update_learning_rate = 0.015,
        update_momentum = 0.9,
        regression = False,
        max_epochs = 5,
        verbose = 1,
        )
```

Once again, we can fit the model using the fit method.

```
net2.fit(X_train_2d, y_train)
```

Compared to the Multi-Layer Perceptron, the Convolutional Neural Network will take longer to train. With the GPU enabled, most training epochs take 12.8 seconds to complete. However, the Convolutional Neural Network achieves a validation loss of about 63%, beating the Multi-Layered Perceptron's validation loss of 40%. So by incorporating the convolution and pooling layers, we can improve our accuracy by 20%.

```
DenseLayer              (None, 10)              produces      10 outputs
DenseLayer              (None, 200)             produces     200 outputs
MaxPool2DLayer          (None, 64, 3, 3)        produces     576 outputs
Conv2DLayer             (None, 64, 6, 6)        produces    2304 outputs
MaxPool2DLayer          (None, 32, 7, 7)        produces    1568 outputs
Conv2DLayer             (None, 32, 14, 14)      produces    6272 outputs
MaxPool2DLayer          (None, 16, 15, 15)      produces    3600 outputs
Conv2DLayer             (None, 16, 30, 30)      produces   14400 outputs
InputLayer              (None, 1, 32, 32)       produces    1024 outputs

 Epoch  |  Train loss  |  Valid loss  |  Train / Val  |  Valid acc  |  Dur
--------|--------------|--------------|---------------|-------------|-------
     1  |    2.205753  |    2.033620  |     1.084644  |     25.98%  |  12.6s
     2  |    1.921355  |    1.791342  |     1.072579  |     37.19%  |  12.5s
     3  |    1.733160  |    1.659929  |     1.044117  |     41.24%  |  12.5s
     4  |    1.624975  |    1.575980  |     1.031088  |     44.36%  |  12.6s
     5  |    1.547256  |    1.499345  |     1.031954  |     47.34%  |  12.6s
     6  |    1.473698  |    1.432022  |     1.029103  |     50.36%  |  12.5s
     7  |    1.406060  |    1.379502  |     1.019251  |     52.19%  |  12.5s
```

8	1.344989	1.335270	1.007279	53.68%	12.6s
9	1.290247	1.297111	0.994708	55.17%	12.6s
10	1.240108	1.264417	0.980774	56.16%	12.6s
11	1.193527	1.230191	0.970196	57.36%	12.5s
12	1.149543	1.197414	0.960021	58.22%	12.5s
13	1.108363	1.169984	0.947332	59.08%	12.6s
14	1.069448	1.148327	0.931309	60.01%	12.6s
15	1.033021	1.127059	0.916563	60.89%	12.6s
16	0.997762	1.112026	0.897247	61.52%	12.5s
17	0.964757	1.096791	0.879618	62.16%	12.5s
18	0.934106	1.086514	0.859728	62.86%	12.5s
19	0.904148	1.079389	0.837649	63.42%	12.5s
20	0.874887	1.077590	0.811892	63.78%	12.5s
21	0.846687	1.077766	0.785595	64.18%	12.5s
22	0.818874	1.073743	0.762635	64.26%	12.5s
23	0.791202	1.074263	0.736507	64.39%	12.5s
24	0.764455	1.076237	0.710304	64.38%	12.5s
25	0.738000	1.085841	0.679657	64.44%	12.5s
26	0.712825	1.097601	0.649439	64.27%	12.5s
27	0.687417	1.097491	0.626353	64.61%	12.5s
28	0.662787	1.113365	0.595301	64.43%	12.4s
29	0.638305	1.132577	0.563586	64.31%	12.4s
30	0.614533	1.148427	0.535109	64.45%	12.4s
31	0.590563	1.175184	0.502528	64.40%	12.4s
32	0.567124	1.192100	0.475735	64.42%	12.4s
33	0.542636	1.208512	0.449012	64.45%	12.4s
34	0.519985	1.236731	0.420451	64.52%	12.4s
35	0.496460	1.257824	0.394697	64.51%	12.4s
36	0.473001	1.291366	0.366280	64.28%	12.4s
37	0.450065	1.335728	0.336944	63.81%	12.5s
38	0.427229	1.385318	0.308398	63.52%	12.5s
39	0.404840	1.441188	0.280907	63.36%	12.5s
40	0.382691	1.476996	0.259101	63.49%	12.4s
41	0.361423	1.531115	0.236052	63.32%	12.4s
42	0.340386	1.599722	0.212778	63.18%	12.4s

43	0.319396	1.673809	0.190820	63.10%	12.4s
44	0.301071	1.765651	0.170516	62.80%	12.4s
45	0.283112	1.830162	0.154692	62.69%	12.4s
46	0.266026	1.886216	0.141037	62.72%	12.4s
47	0.253665	1.948299	0.130198	62.74%	12.4s
48	0.241424	1.999700	0.120730	62.72%	12.4s
49	0.234656	2.065408	0.113612	62.56%	12.4s
50	0.229542	2.124553	0.108042	63.15%	12.4s

With only the CPU on Domino's XX-Large Hardware tier, each training epoch takes about 177 seconds, or close to 3 minutes, to complete. Thus, by training using the GPU, we see about a 15x speedup in the training time.

```
DenseLayer              (None, 10)              produces      10 outputs
DenseLayer              (None, 200)             produces     200 outputs
MaxPool2DLayer          (None, 64, 3, 3)        produces     576 outputs
Conv2DLayer             (None, 64, 6, 6)        produces    2304 outputs
MaxPool2DLayer          (None, 32, 7, 7)        produces    1568 outputs
Conv2DLayer             (None, 32, 14, 14)      produces    6272 outputs
MaxPool2DLayer          (None, 16, 15, 15)      produces    3600 outputs
Conv2DLayer             (None, 16, 30, 30)      produces   14400 outputs
InputLayer              (None, 1, 32, 32)       produces    1024 outputs
```

Epoch	Train loss	Valid loss	Train / Val	Valid acc	Dur
1	2.246350	2.124588	1.057311	23.33%	177.5s
2	1.973923	1.826300	1.080832	36.16%	176.8s
3	1.749182	1.680961	1.040585	40.52%	176.8s
4	1.633140	1.596726	1.022805	43.65%	176.8s
5	1.557960	1.528299	1.019408	46.29%	176.7s
6	1.492055	1.472755	1.013104	48.02%	176.7s
7	1.431668	1.425322	1.004452	49.72%	176.9s

8	1.377004	1.383452	0.995339	51.33%	176.7s
9	1.327587	1.346949	0.985626	52.67%	176.9s
10	1.282365	1.314363	0.975655	54.10%	177.0s
11	1.240280	1.285391	0.964905	55.29%	176.8s
12	1.200430	1.260074	0.952666	56.65%	176.9s
13	1.162588	1.240644	0.937084	57.30%	177.1s
14	1.126327	1.220884	0.922550	58.33%	177.0s
15	1.091960	1.203659	0.907201	59.04%	177.0s
16	1.058333	1.185229	0.892936	59.57%	176.9s
17	1.025399	1.170829	0.875789	60.31%	177.3s
18	0.994079	1.157426	0.858870	61.11%	177.1s
19	0.963416	1.144531	0.841756	61.75%	177.0s
20	0.934234	1.137136	0.821567	62.07%	177.2s
21	0.905911	1.129705	0.801900	62.40%	177.3s
22	0.878504	1.127287	0.779308	62.52%	177.2s
23	0.851404	1.124866	0.756894	62.80%	177.0s
24	0.824258	1.126260	0.731854	62.67%	177.1s
25	0.797099	1.126738	0.707440	62.94%	177.4s
26	0.771684	1.136834	0.678801	63.02%	177.5s
27	0.745962	1.146494	0.650646	63.21%	177.4s
28	0.721063	1.151355	0.626274	63.41%	177.2s
29	0.696683	1.163693	0.598683	63.49%	177.4s
30	0.671730	1.180111	0.569209	63.52%	177.4s
31	0.647605	1.195997	0.541477	63.41%	177.5s
32	0.623296	1.216051	0.512557	63.10%	177.5s
33	0.599012	1.238086	0.483821	63.00%	177.2s
34	0.575137	1.269758	0.452950	62.75%	177.2s
35	0.551720	1.298648	0.424842	62.58%	177.4s
36	0.528849	1.340816	0.394423	62.35%	177.5s
37	0.505668	1.378510	0.366822	62.46%	177.3s
38	0.483216	1.423057	0.339562	62.38%	177.4s
39	0.460213	1.468248	0.313444	62.17%	177.4s
40	0.438207	1.507542	0.290677	62.08%	177.5s
41	0.416352	1.557685	0.267289	61.95%	177.6s
42	0.394626	1.612770	0.244688	61.80%	177.6s

43	0.372949	1.659062	0.224795	61.62%	177.4s
44	0.352200	1.730373	0.203540	61.70%	177.4s
45	0.330830	1.807168	0.183065	61.25%	177.3s
46	0.310925	1.889098	0.164589	61.19%	177.7s
47	0.292424	1.973518	0.148174	60.73%	177.6s
48	0.277566	2.072964	0.133898	60.41%	177.4s
49	0.264899	2.109496	0.125575	60.52%	177.5s
50	0.253137	2.218654	0.114095	59.51%	177.4s

Once again, we see similar results for the Convolutional Neural Network trained on the CPU compared to the Convolutional Neural Network trained on the GPU, with similar validation accuracies and training losses.

When we test the convolutional neural network on the testing data, we achieve an accuracy of 61%.

```
y_pred2 = net2.predict(X_test_2d)
print "The accuracy of this network is: %0.2f" % (y_pred2 == y_test).mean()
```

All the code for building the convolutional neural network can be found in this file.

As you can see, using the GPU to train deep neural networks results in a speed up in runtime, ranging from 3x faster to 15x faster in this project. In industry and academia, we often use multiple GPUs as this cuts the training runtime of deep networks down from weeks to days.

Additional Resources

As always, the code for this project is publicly available on Domino.

NVIDIA has recently launched a free online course on GPUs and deep learning, so those of who are interested in learning more about GPU-accelerated deep learning can check this course out!

Additionally, Udacity has a free online course in CUDA Programming and Parallel Programming for those who are interested in general GPU and CUDA programming.

If you are more interested in convolutional neural networks, the Neural Networks and Deep Learning ebook recently released a a chapter on convolutional neural network.

I would like to extend a special acknowledgement to Reddit user sdsfs23fs for their criticism and correcting incorrect statements I had in my original version of this blog post. In addition, this Reddit user provided me with code for the image pre-processing step that greatly improves the accuracy of the neural networks.

Edit: I have updated the requirements.txt file for this project to overcome an error caused by

one of the Python libraries. The new requirements.txt is listed above in the Deep Learning in Python section.

Share this post

like it includes Theano preinstalled. Nice. :)

http://exxactcorp.com/index.ph...

Reply • Share ›

Cameron Alexander • 5 months ago

It seems like the posted GPU setup files are a little out of date. You have to actually import the .theanor program in the test GPU program for it to work. If you click the links provided those work fine, it's just th code snippets that are wrong.

Reply • Share ›

Manojit Nandi • 5 months ago

Hi Everyone,

I've edited the blog post based on the feedback provided by a user on Reddit. I have included a special acknowledgement at the end of the post for the user.

Reply • Share ›

6 comments • 4 months ago

songchao — Thanks for your illustrated explanation! Can you recommend some open source libraries for OPTICS which quality and …

6 comments • 7 months ago

Jonathan Moore — Installation of package of satisfying dependency stacks, is a key for me in choosing a language …