Department of Computer Science
University of Toronto
`http://learning.cs.toronto.edu`

6 King's College Rd, Toronto
M5S 3G4, Canada
fax: +1 416 978 1455

July 9, 2010

UTML TR 2010–002

Gnumpy: an easy way to use GPU boards in Python

Tijmen Tieleman
Department of Computer Science, University of Toronto

Abstract

This technical report describes Gnumpy, a Python module that uses a GPU for computations, but has numpy's convenient interface.

Gnumpy: an easy way to use GPU boards in Python

Tijmen Tieleman
Department of Computer Science, University of Toronto

1 Introduction

Video cards, also known as graphics processing units (GPU's), have recently become interesting for scientific computation that has nothing to do with graphics. They contain many compute units (small processors), which in themselves are not very fast, but together pack a lot of compute power. Many operations, such as matrix multiplication and most elementwise operations, can be performed quite efficiently on such hardware: typically 10 to 100 times faster than on a conventional CPU.

Nvidia, one company that manufactures these units, has made software available that makes programming them easier. Together with the fact that they cost little, this has made GPU computing very interesting for scientific computing (Raina et al., 2009).

Cudamat (Mnih, 2009) has brought GPU's even closer to the everyday researcher, by wrapping much of the Nvidia software in a Python module, and adding some of its own. Cudamat has been used quite a bit already (Mohamed et al., 2009; Ranzato & Hinton, 2010; Ranzato et al., 2010; Martens & Sutskever, 2010; Mnih & Hinton, 2010), as well as several projects in progress.

However, programming in Cudamat, while much easier than programming GPU's directly, is still much less convenient than programming using Python's de facto standard for numerical computations: numpy. Most Cudamat functions serve to manipulate state: they expect as parameter a matrix object to which the result of the computation will be written. This can make Cudamat programming feel a bit like C programming: most code is statements which manipulate state, as opposed to expressions which describe values. Especially with complex computations that have many intermediate results, Cudamat can be quite inconvenient. To truly use the expression-based programming that Python enables (sometimes called "Pythonic" programming), the numpy interface is required.

Gnumpy is the next step, building on Cudamat but providing the convenient numpy interface. It is a library that interfaces almost exactly like numpy, but internally uses a GPU to do its computations. Internally, it uses Cudamat, but the user never sees Cudamat. The user only sees the convenient numpy interface, and sees that the computations are performed fast, using the GPU. Thus, Gnumpy provides the speed of GPU's, while not sacrificing the programming convenience of numpy. Most numpy-using programs will run on Gnumpy after only minimal modifications, if any.

Compared to using Cudamat, programming using Gnumpy is easier in many ways. Gnumpy-based programs are typically shorter, more intuitive, and therefore easier to write, inpect, debug, and maintain. Programmers who are used to numpy will find that they can use almost all of their numpy experience in exactly the same way, when they switch to Gnumpy.

2 Code Example

To illustrate the difference in programming style between using Cudamat and using Gnumpy, here is the code example that is included in Cudamat, for the Machine Learning task of training a Restricted Boltzmann Machine (Hinton et al., 2006). The details of the algorithm are not important here; instead, look at the general appearance of the program.

These two implementations were written by different people, which results in slightly different programming style. However, the Gnumpy version is a very direct adaptation of the Cudamat program.

Notice that the Gnumpy version looks exactly like an implementation using numpy. The only difference is that instead of "import numpy", it starts with "import gnumpy". If you prefer "from numpy import *", then you can use "from gnumpy import *" in exactly the same way.

The Gnumpy version is shorter, easier to understand, and easier to write, debug, and maintain, especially for people who are used to numpy.

2.1 Implementation with cudamat

In Cudamat, the following is a reasonable implementation:

```
import time
import numpy as np
import cudamat as cm
import util

# initialize CUDA
cm.cublas_init()
cm.CUDAMatrix.init_random(1)

# load data
util.load('mnist.dat', globals())
dev_dat = cm.CUDAMatrix(cm.reformat(dat/255.))

# training parameters
epsilon = 0.1
momentum = 0.9

num_epochs = 30
batch_size = 128
num_batches = dat.shape[1]/batch_size

# model parameters
num_vis = dat.shape[0]
num_hid = 4096

# initialize weights
w_vh = cm.CUDAMatrix(0.1 * np.random.randn(num_vis, num_hid))
w_v = cm.CUDAMatrix(np.zeros((num_vis, 1)))
w_h = cm.CUDAMatrix(-4.*np.ones((num_hid, 1)))

# initialize weight updates
wu_vh = cm.CUDAMatrix(np.zeros((num_vis, num_hid)))
wu_v = cm.CUDAMatrix(np.zeros((num_vis, 1)))
wu_h = cm.CUDAMatrix(np.zeros((num_hid, 1)))

# initialize temporary storage
```

```
v = cm.empty((num_vis, batch_size))
h = cm.empty((num_hid, batch_size))
r = cm.empty((num_hid, batch_size))

start_time = time.time()
for epoch in range(num_epochs):
  print "Epoch " + str(epoch + 1)
  err = []

  for batch in range(num_batches):
    # get current minibatch
    v_true = dev_dat.slice(batch*batch_size,(batch + 1)*batch_size)
    v.assign(v_true)

    # apply momentum
    wu_vh.mult(momentum)
    wu_v.mult(momentum)
    wu_h.mult(momentum)

    # positive phase
    cm.dot(w_vh.T, v, target = h)
    h.add_col_vec(w_h)
    h.apply_sigmoid()

    wu_vh.add_dot(v, h.T)
    wu_v.add_sums(v, axis = 1)
    wu_h.add_sums(h, axis = 1)

    # sample hiddens
    r.fill_with_rand()
    r.less_than(h, target = h)

    # negative phase
    cm.dot(w_vh, h, target = v)
    v.add_col_vec(w_v)
    v.apply_sigmoid()

    cm.dot(w_vh.T, v, target = h)
    h.add_col_vec(w_h)
    h.apply_sigmoid()

    wu_vh.subtract_dot(v, h.T)
    wu_v.add_sums(v, axis = 1, mult = -1.)
    wu_h.add_sums(h, axis = 1, mult = -1.)

    # update weights
    w_vh.add_mult(wu_vh, epsilon/batch_size)
    w_v.add_mult(wu_v, epsilon/batch_size)
    w_h.add_mult(wu_h, epsilon/batch_size)
```

```
    # calculate reconstruction error
    v.subtract(v_true)
    err.append(v.euclid_norm()**2/(num_vis*batch_size))

  print "Mean squared error: " + str(np.mean(err))
  print "Time: " + str(time.time() - start_time)

w_vh.copy_to_host()
util.save('weights.dat', 'w_vh', {'w_vh': w_vh.numpy_array})

cm.cublas_shutdown()
```

2.2 Implementation with Gnumpy

Using Gnumpy instead of Cudamat, the implementation looks quite different:

```
def test_gnumpy(num_epochs):
  import gnumpy as gpu
  # load data. <dat> is 2 dimensional: 60000 X 784
  dat = gpu.garray(load('mnist_cudaTest').T/255.)
  # training parameters
  epsilon = 0.1
  momentum = 0.9
  batch_size = 128
  num_batches = dat.shape[0]/batch_size
  # model parameters
  num_vis = dat.shape[1]
  num_hid = 4096
  # initialize weights
  w_vh = 0.1 * gpu.randn(num_vis, num_hid)
  w_v = gpu.zeros(num_vis)
  w_h = -4. * gpu.ones(num_hid)
  # initialize weight updates
  wu_vh = gpu.zeros((num_vis, num_hid))
  wu_v = gpu.zeros(num_vis)
  wu_h = gpu.zeros(num_hid)
  for epoch in range(num_epochs):
    err = []
    for batch in range(num_batches):
      # positive phase
      v1 = dat[batch*batch_size : (batch + 1)*batch_size]
      h1 = (gpu.dot(v1, w_vh) + w_h).logistic()
      # sample hiddens
      hSampled = h1.rand() < h1
      # negative phase
      v2 = (gpu.dot(hSampled, w_vh.T) + w_v).logistic()
      h2 = (gpu.dot(v2, w_vh) + w_h).logistic()
      # update weights
      wu_vh = wu_vh * momentum + gpu.dot(v1.T, h1) - gpu.dot(v2.T, h2)
```

```
    wu_v = wu_v * momentum + v1.sum(0) - v2.sum(0)
    wu_h = wu_h * momentum + h1.sum(0) - h2.sum(0)

    w_vh += wu_vh * (epsilon/batch_size)
    w_v += wu_v * (epsilon/batch_size)
    w_h += wu_h * (epsilon/batch_size)
    # calculate reconstruction error
    err.append((v2-v1).euclid_norm()**2/(num_vis*batch_size))
  print "Mean squared error: " + str(mean(err))
  return w_vh, w_v, w_h
```

3 Differences From Numpy

Gnumpy was carefully designed to mimic numpy. However, some differences turned out to be unavoidable. A detailed description can be found in the Gnumpy documentation.

The main difference is in aliasing. In numpy, many operations produce an alias: a new array object, that however shares its data with another array object. This happens when you take a transpose, a slice, or a reshaped version of an existing array. In Gnumpy, transposes and some types of slices are NOT aliases, but instead new arrays, with copied data. If, like the author, you use expressions much more than statements, and you rarely modify the content of your arrays, then this difference is irrelevant. However, if you do change the content of an array, and expect that the content of any aliases of the array are equally changed, then it is important to know exactly which operations produce aliases and which do not. In Gnumpy, simple slices along the first dimension ("row slices"), produce an alias just like numpy, but slices along other dimensions, as well as more complicated slices (with stride not equal to 1), behave differently. In numpy, those produce an alias, but in Gnumpy, they produce a copy.

The second difference is that Gnumpy knows only one datatype: float32. If you are used to working with numpy arrays of integers, booleans, or even strings and other general Python objects, you will need to keep those arrays as numpy arrays, and convert only numerical arrays to Gnumpy.

4 Features

The main feature of gnumpy is the similarity to numpy - similarity in interface. However, there are a few components that numpy does not have. Full details can be found in the online documentation.

4.1 Memory usage

A GPU has more compute power and less memory than a CPU, so memory availability may well become a bottleneck. To the author's knowledge, there are GPU boards with up to 4 GB of memory, but most boards have less. You may end up having to keep some data on the CPU and only temporarily copy it to the GPU when it's needed there.

Gnumpy contains some memory usage debugging tools. It can tell you how much memory is in use, and where in your program it was allocated. Gnumpy does not know how much memory your board has available, but you can tell it a maximum amount to use.

4.2 Emulation mode

Gnumpy can run on a CPU, if you tell it to. This emulation mode is made possible by a library called *npmat*, written by Ilya Sutskever. It can be useful in various situations.

- One is obviously when you are writing a program that will use a GPU some time later, but you are writing it on your laptop which does not have a GPU.
- By running the same program first on the CPU and then on a GPU, you can find out what speed-up the GPU provides.
- Sometimes, you may need to perform a computation with more precision than a float32 can provide. A typical example is a gradient check with finite difference gradient approximation. To get an accurate finite difference approximation, you can run your program using the CPU, and specify the desired precision: float32, float64, or float128 (if your version of numpy supports it).

5 Using Gnumpy

If you would like to use Gnumpy, you can find it at
`http://www.cs.toronto.edu/~tijmen/gnumpy.html` for free. However, if you use it in scientific work that gets published, you should cite this technical report.

Full documentation, including a manual describing how to get started, and details about such things as the differences between gnumpy and numpy, can also be found on that website.

References

Hinton, G., Osindero, S., & Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, *18*, 1527–1554.

Martens, J., & Sutskever, I. (2010). Parallelizable Sampling of Markov Random Fields. *Artificial Intelligence and Statistics, 2010.*

Mnih, V. (2009). *CUDAMat: A CUDA-based matrix class for Python* (Technical Report UTML TR 2009-004). University of Toronto, Department of Computer Science.

Mnih, V., & Hinton, G. (2010). Learning to Detect Roads in High-Resolution Aerial Images. *European Conference on Computer Vision, 2010.* To appear.

Mohamed, A., Dahl, G., & Hinton, G. (2009). Deep Belief Networks for phone recognition.

Raina, R., Madhavan, A., & Ng, A. (2009). Large-scale deep unsupervised learning using graphics processors. *Proceedings of the 26th Annual International Conference on Machine Learning* (pp. 873–880).

Ranzato, M., & Hinton, G. (2010). Modeling Pixel Means and Covariances Using Factorized Third-Order Boltzmann Machines. *IEEE Conference on Computer Vision and Pattern Recognition, 2010. CVPR'10.*

Ranzato, M., Krizhevsky, A., & Hinton, G. (2010). Factored 3-Way Restricted Boltzmann Machines for Modeling Natural Images. *Proc. Conference on AI and Statistics (AI-Stats).*