

Homework 1 - Berkeley STAT 157

Handout 1/22/2017, due 1/29/2017 by 4pm in Git by committing to your repository. Please ensure that you add the TA Git account to your repository.

- 1. Write all code in the notebook.
- 2. Write all text in the notebook. You can use MathJax to insert math or generic Markdown to insert figures (it's unlikely you'll need the latter).
- 3. Execute the notebook and save the results.
- 4. To be safe, print the notebook as PDF and add it to the repository, too. Your repository should contain two files: homework1.ipynb and homework1.pdf.

The TA will return the corrected and annotated homework back to you via Git (please give `rythei` access to your repository).

In [1]: `from mxnet import ndarray as nd`

1. Speedtest for vectorization

Your goal is to measure the speed of linear algebra operations for different levels of vectorization. You need to use `wait_to_read()` on the output to ensure that the result is computed completely, since `NDArray` uses asynchronous computation. Please see http://beta.mxnet.io/api/ndarray/autogen/mxnet.ndarray.NDArray.wait_to_read.html for details.

- 1. Construct two matrices A and B with Gaussian random entries of size 4096×4096 .
- 2. Compute $C = AB$ using matrix-matrix operations and report the time.
- 3. Compute $C = AB$, treating A as a matrix but computing the result for each column of B one at a time. Report the time.
- 4. Compute $C = AB$, treating A and B as collections of vectors. Report the time.
- 5. Bonus question - what changes if you execute this on a GPU?

In [2]:

```
# part 1
A = nd.normal(0, 1, shape = (4096, 4096))
B = nd.normal(0, 1, shape = (4096, 4096))

import time

# part 2
tic = time.time()
C = nd.dot(A,B)
print(C)
print(time.time() - tic)
```

[5.1015415	-26.479822	51.007053	...	-7.666253	101.429016
	-101.521935					
[-2.1350682	42.957554	-27.059336	...	50.63337	29.264576
	85.48195					
[179.61961	104.32006	-42.85021	...	4.7839203	-12.166644
	33.128284					
...						
[-19.171553	-24.856712	72.80983	...	36.38448	-28.801601
	8.885335					
[-181.80132	2.513298	-8.9951515	...	-78.88404	-61.0984
	-65.812164					
[64.217995	35.151516	-58.099884	...	39.043835	62.036263
	1.4770873]					

<NDArray 4096x4096 @cpu(0)>
1.947239637374878

In [5]:

```
# part 3

import numpy as np

tic = time.time()

C = nd.zeros((4096, 4096))
for i in range(4096):
    transp = B[np.arange(4096)], i).reshape(4096,1)
    new_col = nd.dot(A,transp).reshape(4096,)
    C[0:4096, i] = new_col

print(C)
print(time.time() - tic)
```

[5.1015415	-26.479822	51.007053	...	-7.666253	101.429016
	-101.521935					
[-2.1350682	42.957554	-27.059336	...	50.63337	29.264576
	85.48195					
[179.61961	104.32006	-42.85021	...	4.7839203	-12.166644
	33.128284					
...						
[-19.171553	-24.856712	72.80983	...	36.38448	-28.801601
	8.885335					
[-181.80132	2.513298	-8.9951515	...	-78.88404	-61.0984
	-65.812164					
[64.217995	35.151516	-58.099884	...	39.043835	62.036263
	1.4770873]					

<NDArray 4096x4096 @cpu(0)>
53.83018636703491

comment Q1.4

i couldn't find a solution that was efficient enough to compute in time for the deadline, so i did a 100x100 matrix instead. i know the point of the exercise was just to show how slow it is not using efficient vectorization, but i still solved the problem and i hope this will suffice. thanks!

(- :

In [6]:

```
# part 4

tic = time.time()

A_100 = nd.normal(0, 1, shape = (100, 100))
B_100 = nd.normal(0, 1, shape = (100, 100))

C_100 = nd.zeros((100, 100))

for j in range(100):
    b_col = B[np.arange(100)], j].reshape(100,1) #jth col of b

    for i in range(100):
        a_row_i = A[i].reshape(100, 1)
        calc = nd.zeros((100,1))
        test = b_col*a_row_i
        C_100[i,j] = sum(test)

print(C_100)
print(time.time() - tic)
```

[-12.168565	-4.50321	8.210232	...	0.20940703	-12.027266
	-2.3722553					
[-18.157333	-0.20229495	1.0201511	...	0.53903174	-5.057396
	2.2450316					
[-6.4128947	4.3309937	6.790632	...	-3.8496003	16.13936
	-2.2113798					
...						
[15.365839	-7.0519075	-7.716825	...	-18.962404	0.6059242
	0.72697425]					
[3.2971926	25.627968	-21.552677	...	2.0595	0.9457317
	16.829914					
[12.690775	-6.101557	-6.250719	...	-0.8239009	8.9878
	-1.5448539					

<NDArray 100x100 @cpu(0)>
47.93808698654175

2. Semidefinite Matrices

Assume that $A \in \mathbb{R}^{m \times n}$ is an arbitrary matrix and that $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix with nonnegative entries.

- 1. Prove that $B = ADA^T$ is a positive semidefinite matrix.
- 2. When would it be useful to work with B and when is it better to use A and D ?

part 1

$B = ADA^T$

$A^T B = A^T ADA^T$

$A^T BA = DA^T A$

$D = A^T BA$

it's given that D is diagonal with nonnegative entries, therefore D is a positive semidefinite matrix.

and so for an arbitrary matrix A , if D can be decomposed as $A^T BA$ and is ≥ 0 , B is also a PSD.

part 2

one might use A and D when trying to check if the eigen values for B are also nonnegative. also, calculating the determinant for D will be easier, especially if B is not diagonal. we may use B for covariance matrix.

3. MXNet on GPUs

- 1. Install GPU drivers (if needed)
- 2. Install MXNet on a GPU instance
- 3. Display `!nvidia-smi`
- 4. Create a 2×2 matrix on the GPU and print it. See http://d2l.ai/chapter_deep-learning-computation/use-gpu.html for details.

In [1]: `import mxnet as mx`

In [2]: `from mxnet import nd`

In [3]: `from mxnet.gluon import nn`

In [4]: `mx.cpu(), mx.gpu(), mx.gpu(1)`

Out[4]: `(cpu(0), gpu(0), gpu(1))`

In [5]: `x = nd.ones((2, 2), ctx = mx.gpu())`
`x`

Out[5]: `[[1. 1.]`
`[1. 1.]]`
<NDArray 2x2 @gpu(0)>

4. NDArray and NumPy

Your goal is to measure the speed penalty between MXNet Gluon and Python when converting data between both. We are going to do this as follows:

- 1. Create two Gaussian random matrices A, B of size 4096×4096 in `NDArray`.
- 2. Compute a vector $c \in \mathbb{R}^{4096}$ where $c_i = \|AB_i\|^2$ where c is a `NumPy` vector.

To see the difference in speed due to Python perform the following two experiments and measure the time:

- 1. Compute $\|AB_i\|^2$ one at a time and assign its outcome to c_i directly.
- 2. Use an intermediate storage vector d in `NDArray` for assignments and copy to `NumPy` at the end.

In [6]:

```
# part 1
A = nd.normal(0, 1, shape = (4096, 4096))
B = nd.normal(0, 1, shape = (4096, 4096))

c = sum((nd.dot(A, B))**2) ** (1/2)
c = c.asnumpy()
c
```

array([4068.9065, 4076.6711, 4124.69 , ..., 4072.759 , 4069.4392,
4062.4224], dtype=float32)

In [9]:

```
# part 2

tic = time.time()

d = nd.zeros((4096, 1))
product = nd.dot(A,B)
for i in range(4096):
    sum_sq = sum(product[0:4096, i]**2)
    d[i] = sum_sq

c_i = d.reshape(4096, 1).asnumpy()

print(c_i)
print(time.time() - tic)
```

[16556000.]
[16619247.]
[17013068.]
...
[16587366.]
[16560336.]
[16503276.]
851.4459578990936

In [18]: `c_i = (d.reshape(4096,1).asnumpy())**(1/2)`
`c_i`

Out[18]: `array([[4068.9065],`
`[4076.6711],`
`[4124.69],`
`...`
`[4072.759],`
`[4069.4392],`
`[4062.4224]], dtype=float32)`

5. Memory efficient computation

We want to compute $C \leftarrow A \cdot B + C$, where A, B and C are all matrices. Implement this in the most memory efficient manner. Pay attention to the following two things:

- 1. Do not allocate new memory for the new value of C .
- 2. Do not allocate new memory for intermediate results if possible.

In [19]: `C += nd.dot(A, B)`
`C`

Out[19]: `[-89.2842 73.64483 -25.558414 ... 90.13895 95.803665`
`-162.3457]`
`[-37.078636 -6.3714905 11.097342 ... 44.199265 -45.428867`
`78.536705]`
`[134.2446 214.59921 68.38664 ... -66.71899 -20.114132`
`40.64216]`
`...`
`[-95.55035 41.435562 122.75974 ... 69.42801 -94.71486`
`46.208897]`
`[-231.5051 -25.642136 -83.37668 ... -16.456684 -135.8109`
`-83.9735]`
`[75.27684 -1.9213676 21.20346 ... 81.998985 -7.3830795`
`-84.509315]`
<NDArray 4096x4096 @cpu(0)>

6. Broadcast Operations

In order to perform polynomial fitting we want to compute a design matrix A with

$$A_{ij} = x_i^j$$

Our goal is to implement this without a single for loop entirely using vectorization and broadcast. Here $1 \leq j \leq 20$ and $x = \{-10, -9.9, \dots, 10\}$. Implement code that generates such a matrix.

In [20]: `x = nd.array(np.arange(-10, 10, 0.1)).reshape(10, 20)`
`j = nd.array(np.arange(1, 21))`
`A = x**j`
`A`

Out[20]: `[[-1.00000000e+01 9.80099945e+01 -9.41192078e+02 8.85292676e+03`
`-8.15372812e+04 7.35091875e+05 -6.48477400e+06 5.59581920e+07`
`-4.72161280e+08 3.89416296e+09 -3.13810596e+10 2.46990275e+11`
`-1.89790670e+12 1.42321134e+13 -1.04106316e+14 7.42510859e+14`
`-3.51843750e+15 3.49466736e+16 -2.30389674e+17 1.47808970e+18]`
`[-8.00000000e+00 6.20499998e+01 -4.74552032e+02 3.51530371e+03`
`-2.53552520e+04 1.77978516e+05 -1.21512812e+06 8.06460250e+06`
`-5.19986840e+07 3.25524288e+08 -1.97732659e+09 1.16463319e+10`
`-6.64685240e+11 3.67322071e+11 -1.96407892e+12 1.01534516e+13`
`-5.07060400e+13 2.44416288e+14 -1.13616609e+15 5.08857954e+15]`
`[-6.00000000e+00 3.48100014e+01 -1.95112015e+02 1.05559998e+03`
`-5.50731738e+03 2.76806406e+04 -1.33892531e+05 6.22597062e+05`
`-2.77990500e+06 1.19042400e+07 -4.88281240e+07 1.91581280e+08`
`-7.18019648e+08 2.56666829e+09 -8.73710080e+09 2.82748436e+10`
`-8.68351672e+10 2.52599534e+11 -6.94602170e+11 1.80167691e+12]`
`[-4.00000000e+00 1.52100010e+01 -5.48719978e+01 1.87416107e+02`
`-6.04661682e+02 1.83826562e+03 -5.25235394e+03 1.40640840e+04`
`-3.51843750e+04 8.19628047e+04 -1.77147000e+05 3.53814938e+05`
`-5.50211000e+05 4.20941981e+06 -1.67725850e+07 2.32830650e+07`
`3.68934950e+06 1.81633140e+07 9.37959200e+07 5.07094240e+08`
`2.86511642e+09 1.68900588e+10 1.03726137e+11 6.62662414e+11]`
`[4.00000000e+00 1.68099995e+01 7.40879898e+01 3.41880157e+02`
`1.64916235e+03 8.30376562e+03 4.35817578e+04 2.38112781e+05`
`1.35260600e+06 7.97922800e+06 4.88281240e+07 3.09629280e+08`
`2.03255949e+09 1.37994691e+10 9.68069366e+10 7.01137224e+11`
`5.23837217e+12 4.03410424e+13 3.19986875e+14 2.61240419e+15]`
`[6.00000000e+00 3.20999991e+01 2.38327972e+02 1.57529626e+03`
`1.07374189e+04 7.54188906e+04 5.45516000e+05 4.60607575e+06`
`3.10871080e+07 2.44619440e+08 1.97732659e+09 1.64096788e+10`
`1.39740496e+11 1.22045058e+12 1.09263695e+13 8.00225956e+14`
`9.41523068e+14 9.05384045e+15 8.90835745e+16 8.9482683e+17]`