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 [33]:

```
from mxnet import ndarray as nd
import time
import numpy as np
```

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 (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 [50]:

```
A = nd.random.normal(0,1,(4096,4096))
B = nd.random.normal(0,1,(4096,4096))
```

In [35]:

```
tictok = time.time()
C1  = nd.dot(A, B)
C1.wait_to_read()
print(time.time() - tictok)
```

```
2.7938380241394043
```

In [36]:

```
tictok2 = time.time()
C2  = nd.ones((4096,4096))
for column in range(4096):
    C2[:,column] = nd.dot(A, B[:,column])

C2.wait_to_read()
print(time.time() - tictok2)
```

```
59.87040686607361
```

In [ ]:

```
tictok3 = time.time()
C3 = nd.ones((4096,4096))
B = B.T

for row in range(4096):
    for col in range(4096):
        C3[row , col] = nd.dot(A[row, :], B[:, col]).asscalar()

C3.wait_to_read()

print(time.time() - tictok3)
```

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 $B = ADA^\top$ 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?

A positive semidefinite matrix is one that has all nonnegative entries. A * AT gives us the identity matrix.

$B = ADA^\top$

$A^\top * B = A^\top * A * D * A^\top$

$A^\top * B = I * D * A^\top$

$A^\top * B = D * A^\top$

We know that D has nonnegative entries, so to continue, so does B

Q2 It is useful to work for B when we are not guaranteed to have non-negative entries in A but we need to process non-negative entries. It is better to use A and D when they are well formed and inform us about our data ie in the form of eigenvalues

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 (http://d2l.ai/chapter_deep-learning-computation/use-gpu.html)](http://d2l.ai/chapter_deep-learning-computation/use-gpu.html) for details.

waiting on AWS stuff to be sorted

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 $\mathbf{c} \in \mathbb{R}^{4096}$ where $c_i = \|AB_{i \cdot}\|^2$ where $\mathbf{c}$ is a **NumPy** vector.

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

1. Compute $\|AB_{i \cdot}\|^2$ one at a time and assign its outcome to $\mathbf{c}_i$ directly.
2. Use an intermediate storage vector $\mathbf{d}$ in NDArray for assignments and copy to NumPy at the end.

In [37]:

```
A = nd.random.normal(0,1,(4096,4096))
B = nd.random.normal(0,1,(4096,4096))
```

In [38]:

```
C = np.ones(4096)
tictok4 = time.time()
for i in range(4096):
    C[i] = (nd.dot(A, B[:, i]).norm() ** 2).asscalar()

print(time.time() - tictok4)
```

```
70.54376912117004
```

In [39]:

```
d = nd.ones((1,4096))
#then copy to numpy


tictok4 = time.time()
for i in range(4096):
    d[0,i] = (nd.dot(A, B[:, i]).norm() ** 2).asscalar()

C = d.asnumpy()[0]
print(C)
print(time.time() - tictok4)
```

```
[16167373. 16117177. 16230506. ... 15435451. 16595865. 16724168.]
67.20023226737976
```

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 [42]:

```
C = nd.ones((4096,4096))
C += nd.dot(A,B)
C
```

Out[42]:

```
[[  -3.886014    -67.21042     -16.672274    ...   39.15998
    17.55918      95.34131    ]
 [  64.25952      27.15741    -185.90944    ...  -20.283806
   -45.280098     22.743095   ]
 [  21.87984      17.43692     -17.752872    ...   22.392632
    35.490776     28.172995   ]
 ...
 [ -40.733322     14.0265465   -65.464294    ...  -31.194862
    -0.24336243  -84.897766   ]
 [ -97.823456    -95.82811      78.30678     ...    8.717602
   165.5814       17.810436   ]
 [  31.97062     -15.854549    -46.087063    ...  -64.235374
   -19.461498    -48.029125   ]]
<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 $x = \{-10, -9.9, \ldots 10\}$. Implement code that generates such a matrix.

In [44]:

```
bases_j = np.arange(1,21,1)
j = nd.array(bases_j)
bases_x = np.arange(-10,10,0.1).reshape(10,20)
x = nd.array(bases_x)
A = x ** j
print(A)
```

```
[[-1.00000000e+01  9.80099945e+01 -9.41192078e+02  8.85292773e+03
  -8.15372891e+04  7.35091875e+05 -6.48477400e+06  5.59581920e+07
  -4.72161280e+08  3.89416269e+09 -3.13810596e+10  2.46990275e+11
  -1.89790670e+12  1.42321134e+13 -1.04106308e+14  7.42510859e+14
  -5.16116234e+15  3.49466736e+16 -2.30389674e+17  1.47808970e+18]
 [-8.00000000e+00  6.24099998e+01 -4.74552032e+02  3.51530371e+03
  -2.53552520e+04  1.77978516e+05 -1.21512812e+06  8.06460250e+06
  -5.19986840e+07  3.25524320e+08 -1.97732672e+09  1.16463319e+10
  -6.64685240e+10  3.67322104e+11 -1.96407892e+12  1.01534516e+13
  -5.07060358e+13  2.44416271e+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  1.80167705e+12]
 [-4.00000000e+00  1.52100010e+01 -5.48719978e+01  1.87416107e+02
  -6.04661682e+02  1.83826562e+03 -5.25233594e+03  1.40640850e+04
  -3.51843750e+04  8.19628047e+04 -1.77147000e+05  3.53814938e+05
  -6.50211000e+05  1.09419012e+06 -1.67725838e+06  2.32830650e+06
  -2.90798000e+06  3.24414975e+06 -3.20649900e+06  2.78218175e+06]
 [-2.00000000e+00  3.60999990e+00 -5.83199930e+00  8.35210133e+00
  -1.04857607e+01  1.13906250e+01 -1.05413494e+01  8.15730476e+00
  -5.15978241e+00  2.59374309e+00 -1.00000000e+00  2.82429457e-01
  -5.49755916e-02  6.78222906e-03 -4.70185274e-04  1.52587891e-05
  -1.71798732e-07  3.87420762e-10 -5.24288153e-14  1.00000029e-20]
 [-3.55271368e-14  1.00000007e-02  8.00000038e-03  8.10000114e-03
   1.02400007e-02  1.56250000e-02  2.79936083e-02  5.76480031e-02
   1.34217739e-01  3.48678350e-01  1.00000000e+00  3.13842916e+00
   1.06993265e+01  3.93737450e+01  1.55568054e+02  6.56840820e+02
   2.95147974e+03  1.40630918e+04  7.08235000e+04  3.75899625e+05]
 [ 2.00000000e+00  4.40999937e+00  1.06480007e+01  2.79840984e+01
   7.96262589e+01  2.44140625e+02  8.03180786e+02  2.82429565e+03
   1.05784541e+04  4.20707383e+04  1.77147000e+05  7.87662500e+05
   3.68934950e+06  1.81633140e+07  9.37959200e+07  5.07094272e+08
   2.86511667e+09  1.68900577e+10  1.03726146e+11  6.62662414e+11]
 [ 4.00000000e+00  1.68099995e+01  7.40879898e+01  3.41880157e+02
   1.64916248e+03  8.30376562e+03  4.35817578e+04  2.38112797e+05
   1.35260600e+06  7.97922800e+06  4.88281240e+07  3.09629280e+08
   2.03255949e+09  1.37994701e+10  9.68069448e+10  7.01137224e+11
   5.23837217e+12  4.03410466e+13  3.19986875e+14  2.61240419e+15]
 [ 6.00000000e+00  3.72099991e+01  2.38327972e+02  1.57529626e+03
   1.07374189e+04  7.54188906e+04  5.45516000e+05  4.06067575e+06
   3.10871080e+07  2.44619440e+08  1.97732672e+09  1.64096799e+10
   1.39740496e+11  1.22045058e+12  1.09263695e+13  1.00225956e+14
   9.41523068e+14  9.05384045e+15  8.90835745e+16  8.96482751e+17]
 [ 8.00000000e+00  6.56100082e+01  5.51367981e+02  4.74583252e+03
   4.18211836e+04  3.77149531e+05  3.47927925e+06  3.28211620e+07
   3.16478432e+08  3.11817062e+09  3.13810596e+10  3.22475655e+11
   3.38252975e+12  3.62044059e+13  3.95291543e+14  4.40126666e+15
   4.99587160e+16  5.77951072e+17  6.81232885e+18  8.17906293e+19]]
<NDArray 10x20 @cpu(0)>
```

In [ ]:

```

```