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
import numpy as np
import time
```

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 [2]:

```python
# 1. Construct two matrices A and B with Gaussian random entries of size  4096×4096.
A = nd.array(nd.random.normal(shape=(4096,4096)))
B = nd.array(nd.random.normal(shape=(4096,4096)))
print("A is ", A)
print("B is ", B)
# 2. Compute C=AB using matrix-matrix operations and report the time.
tic = time.time()
C = nd.dot(A,B)
C.wait_to_read()
print("C is ", C)
print("Time taken to compute C = AB using matrix-matrix operations is ",time.time() - tic)
# 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.
tic = time.time()
for i in range(4096):
    C[:,i] = nd.dot(A, B[:,i])
C.wait_to_read()
print("C is ", C)
print("Time taken to compute C = AB using a series of matrix-vector operations is ",time.time() - tic)
# 4. Compute C=AB, treating A and B as collections of vectors. Report the time.
tic = time.time()
C = nd.zeros((4096,4096))
for i in range(4096):
    A_ = A[:,i].reshape(4096,1)
    B_ = B[i,:].reshape(1,4096)
    C += nd.dot(A_,B_)
C.wait_to_read()
print("C is ", C)
print("Time taken to compute C = AB using a series of matrix-vector operations is ",time.time()- tic)
```

```
A is 
[[ 1.1630787   0.4838046   0.29956347 ...  1.6892971   1.0433246
   1.4861063 ]
 [ 1.1784046  -0.90841913 -0.37429735 ...  1.8522056  -1.8105638
  -1.0253092 ]
 [ 0.01498137  0.12917037  1.0849217  ... -0.8227322   0.23276128
  -1.4382302 ]
 ...
 [ 0.89425045  0.04806393  0.06638991 ... -0.47666615 -0.8127311
   1.0233623 ]
 [ 0.47591528 -0.34719887  0.54473215 ...  0.03410461  0.05183558
  -0.20439206]
```

```
 [-0.03636989 -2.5792546  -1.0895499  ...  0.8099311  -1.8143338
   0.92715806]]
<NDArray 4096x4096 @cpu(0)>
B is
[[ 0.90497434 -0.8291843   0.5853669  ... -1.7501848  -0.5421702
   0.9640367 ]
 [-0.3738427   1.5078996  -1.4877979  ... -2.4101987   0.8088441
   1.518381  ]
 [-0.67060167 -2.0757017   0.1026388  ... -1.1176999  -0.7563752
  -1.0046532 ]
 ...
 [-1.0118687   0.04218809  0.70467323 ...  0.02222571  0.61793566
  -0.9676132 ]
 [ 1.3284602  -0.93981117 -1.621257   ... -0.02329569 -1.4178524
   1.1862246 ]
 [ 1.3548605  -2.0296354   0.7279279  ...  0.43979615  0.06064158
  -1.8767449 ]]
<NDArray 4096x4096 @cpu(0)>
C is
[[-3.21697140e+00 -6.86587830e+01 -4.47078362e+01 ...  3.16677055e+01
   3.89036942e+01  4.07550659e+01]
 [-7.48151474e+01  7.92359467e+01  3.06249123e+01 ... -2.78601837e+00
   6.37111816e+01 -4.89600868e+01]
 [-6.30885925e+01 -8.01208572e+01  2.52887459e+01 ...  1.59381142e+01
  -5.30301399e+01  1.40918732e-01]
 ...
 [ 9.48303127e+00 -5.44954872e+00  6.78874969e+01 ...  6.01659012e+01
  -1.54544525e+01  6.66106720e+01]
 [ 4.86723709e+01  1.34490738e+01 -7.44261169e+01 ... -1.20164461e+01
  -5.14430695e+01  6.23531532e+01]
 [-7.94518471e-01 -1.04769993e+01 -1.99739594e+02 ... -1.86650410e+01
   3.55103607e+01  5.43588333e+01]]
<NDArray 4096x4096 @cpu(0)>
Time taken to compute C = AB using matrix-matrix operations is  1.2901279926300049
C is
[[-3.2169876e+00 -6.8658798e+01 -4.4707859e+01 ...  3.1667704e+01
   3.8903690e+01  4.0755062e+01]
 [-7.4815155e+01  7.9235962e+01  3.0624916e+01 ... -2.7860165e+00
   6.3711197e+01 -4.8960094e+01]
 [-6.3088600e+01 -8.0120857e+01  2.5288750e+01 ...  1.5938101e+01
  -5.3030125e+01  1.4090347e-01]
 ...
 [ 9.4830132e+00 -5.4495668e+00  6.7887497e+01 ...  6.0165894e+01
```

```
  -1.5454445e+01  6.6610695e+01]
 [ 4.8672352e+01  1.3449069e+01 -7.4426125e+01 ... -1.2016447e+01
  -5.1443054e+01  6.2353165e+01]
 [-7.9450417e-01 -1.0477011e+01 -1.9973959e+02 ... -1.8665014e+01
   3.5510372e+01  5.4358799e+01]]
<NDArray 4096x4096 @cpu(0)>
Time taken to compute C = AB using a series of matrix-vector operations is  15.006129026412964
C is
[[-3.2170029e+00 -6.8658775e+01 -4.4707787e+01 ...  3.1667793e+01
   3.8903690e+01  4.0755001e+01]
 [-7.4815132e+01  7.9235962e+01  3.0624928e+01 ... -2.7860281e+00
   6.3711205e+01 -4.8960133e+01]
 [-6.3088573e+01 -8.0120720e+01  2.5288673e+01 ...  1.5938072e+01
  -5.3030170e+01  1.4096212e-01]
 ...
 [ 9.4830513e+00 -5.4494839e+00  6.7887573e+01 ...  6.0165806e+01
  -1.5454427e+01  6.6610603e+01]
 [ 4.8672279e+01  1.3449055e+01 -7.4426094e+01 ... -1.2016452e+01
  -5.1443039e+01  6.2353195e+01]
 [-7.9452956e-01 -1.0477141e+01 -1.9973969e+02 ... -1.8665001e+01
   3.5510372e+01  5.4358917e+01]]
<NDArray 4096x4096 @cpu(0)>
Time taken to compute C = AB using a series of matrix-vector operations is  136.92127990722656
```

5. Running these again in GPU we see that....its lightning fast!

```
A is
[[ 2.2122064   0.7740038   1.0434405  ...  0.878721   -0.38373846
   1.6916761 ]
 [ 2.6962957   0.22153018  0.32801175 ...  0.3843791  -1.2372673
  -0.1757338 ]
 [-0.71967256 -1.0548805   1.1552448  ... -0.36272427  0.05136198
  -1.34558   ]
 ...
 [ 0.49840376 -0.2570526   0.5101299  ... -0.5226169   0.49959022
   0.87793326]
 [ 1.0912747   0.48266006  1.7476733  ...  0.58202994 -1.2874165
   2.0073023 ]
 [ 0.5698917   1.4166281   0.3263331  ...  0.92536634  0.40351006
   0.7122988 ]]
<NDArray 4096x4096 @gpu(0)>
```

```
B is
[[-1.06241226e-01 -5.04471242e-01  5.55512011e-01 ...  1.39394909e-01
  -5.07346749e-01  1.33616505e-02]
 [ 9.91607845e-01 -6.94113731e-01 -5.93653977e-01 ...  1.04657161e+00
  -2.11773947e-01 -3.14220071e-01]
 [ 1.28537118e-01 -9.41475093e-01  3.76590896e+00 ... -3.25436592e-01
  -2.12810442e-01  3.90049338e-01]
 ...
 [-1.06211054e+00  2.25660896e+00 -5.40844202e-01 ...  8.60690832e-01
  -1.33645964e+00  8.87757242e-01]
 [ 1.05679415e-01  2.44933033e+00 -1.08395338e+00 ... -1.44457805e+00
  -7.99180120e-02  1.61282802e+00]
 [-1.48620653e+00 -4.72419977e-01  5.87508738e-01 ... -6.83359278e-04
  -1.74312353e-01  4.02012736e-01]]
<NDArray 4096x4096 @gpu(0)>
C is
[[   5.10157    -26.479774    51.007145  ...   -7.6662807  101.42907
  -101.52188  ]
 [  -2.135006    42.957558   -27.059433  ...   50.633404    29.264585
    85.4819   ]
 [ 179.61967    104.32008    -42.850132  ...    4.7838554  -12.166611
    33.12831  ]
 ...
 [ -19.171581   -24.856707    72.80985   ...   36.384487   -28.801613
     8.885338 ]
 [-181.80136      2.5133238   -8.995174  ...  -78.884056   -61.09847
   -65.8122   ]
 [  64.218025    35.15145    -58.099964  ...   39.043835    62.03632
     1.4770927]]
<NDArray 4096x4096 @gpu(0)>
Time taken to compute C = AB using matrix-matrix operations is  0.6295089721679688
C is
[[   5.101576   -26.479836    51.00712   ...   -7.666273   101.429054
  -101.521935 ]
 [  -2.135056    42.957554   -27.059343  ...   50.633377    29.26459
    85.482    ]
 [ 179.61958    104.32005    -42.850147  ...    4.7839017  -12.166657
    33.12832  ]
 ...
 [ -19.17152    -24.856756    72.80983   ...   36.38446    -28.801636
     8.885363 ]
 [-181.8013      2.5132952   -8.995153  ...  -78.88405    -61.098423
   -65.81217  ]
```

```
 [  64.21802     35.151493   -58.09991   ...   39.043823    62.03624
     1.4771056]]
<NDArray 4096x4096 @gpu(0)>
Time taken to compute C = AB using a series of matrix-vector operations is  1.1822915077209473
C is
[[   5.10151    -26.479908    51.007046  ...   -7.6662846  101.42892
  -101.52199  ]
 [  -2.1350617   42.957516   -27.059277  ...   50.633324    29.264465
    85.48187  ]
 [ 179.61974    104.31986    -42.850132  ...    4.7839103  -12.166648
    33.1283   ]
 ...
 [ -19.17156    -24.856695    72.80984   ...   36.38449    -28.801596
     8.885324 ]
 [-181.8011       2.5132995   -8.995185  ...  -78.88388    -61.098442
   -65.81219  ]
 [  64.21795     35.151497   -58.09991   ...   39.043945    62.03618
     1.4770678]]
<NDArray 4096x4096 @gpu(0)>
Time taken to compute C = AB using a series of matrix-vector operations is  1.7138192653656006
```

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?

1. A symmetrix matrix $B \in \mathbb{S}^n$ is said to be positive semidefinite (PSD) if the associated quadratic form is non-negative, i.e.,
$$x^T Bx \geq 0, \forall x \in \mathbb{R}^n.$$
$$Hence,\ x^T Bx \ = \ x^T ADA^T x \ = \ yDy^T \ = \sum_{i=1}^{n} \lambda_i y_i^2 \geq 0 \ \ (change\ of\ variable\ y = A^T x)$$
2. B, a real symmetrix matrix has a complete set of orthogonal eigenvectors for which the corresponding eigenvalues are all real numbers, which prevents us from having to work with complex field. Furthermore, B, a PSD matrix, is convex, which is a useful property for optimization. Using ADA^T is more helpful when operations like products of B is involved.

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) for details.

```
ubuntu@ip-172-31-37-85:~$ nvidia-smi
Sun Jan 27 08:13:31 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.79       Driver Version: 410.79       CUDA Version: 10.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000000:00:1E.0 Off |                    0 |
| N/A   36C    P0    26W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
ubuntu@ip-172-31-37-85:~$ python run.py

[[1. 2.]
 [3. 4.]]
<NDArray 2x2 @gpu(0)>
ubuntu@ip-172-31-37-85:~$
```

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 [3]:
```
A = nd.array(nd.random.normal(shape=(4096,4096)))
B = nd.array(nd.random.normal(shape=(4096,4096)))
# 1. Compute ‖ABi‖2 one at a time and assign its outcome to ci directly.
start = time.time()
c = np.zeros((4096,1))
for i in range(4096):
    c[i] = nd.norm(nd.dot(A, B[:,i])).asscalar()
end = time.time()
print("Method 1 takes ", end - start)

# 2. Use an intermediate storage d in NDArray for assignments and copy to NumPy at the end.
start = time.time()
c = np.zeros((4096,1))
d = nd.zeros((4096,1))
for i in range(4096):
    d[i] = nd.norm(nd.dot(A, B[:,i])).asscalar()
c = d.asnumpy()
end = time.time()
print("Method 2 takes ", end - start)
```

```
Method 1 takes  14.870416164398193
Method 2 takes  16.87332510948181
```

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 [4]:
```
A = nd.array(nd.random.normal(shape=(4096,4096)))
B = nd.array(nd.random.normal(shape=(4096,4096)))
C = nd.array(nd.random.normal(shape=(4096,4096)))
C += nd.dot(A,B)
print("C is ", C)
```

```
C is
[[-1.49538794e+01 -2.90343456e+01 -1.20786583e+02 ...  1.12601952e+02
  -3.49232483e+01 -1.09423615e+02]
 [ 3.03782139e+01  2.29860592e+01 -5.66329117e+01 ...  2.33377476e+01
   5.01700706e+01  1.37430887e+01]
 [-2.43770561e+01 -3.49905968e+01 -3.10774288e+01 ... -4.34055090e+00
  -1.08096857e+01  7.63302536e+01]
 ...
 [-1.50713013e+02  1.06534897e+02 -1.25168953e+02 ...  5.68572617e+01
   1.28955231e+02 -1.09834503e+02]
 [-4.46852951e+01 -2.38864880e+01 -6.48403692e+00 ...  1.19722977e+02
  -5.85413208e+01 -9.49381638e+01]
 [-8.27188416e+01 -5.92363691e+00  3.18313980e+01 ... -1.21761522e+01
  -1.21261887e+02 -3.75423431e-02]]
<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 [5]:
```
x = np.linspace(-10,10,201)
x = nd.array(x).reshape(201,1)
y = np.linspace(1,20,20)
y = nd.array(y)
A = x ** y
print("A is ", A)
```

```
A is
[[-1.0000000e+01  1.0000000e+02 -1.0000000e+03 ...  9.9999998e+17
  -1.0000000e+19  1.0000000e+20]
 [-9.8999996e+00  9.8009995e+01 -9.7029889e+02 ...  8.3451318e+17
  -8.2616803e+18  8.1790629e+19]
 [-9.8000002e+00  9.6040001e+01 -9.4119208e+02 ...  6.9513558e+17
  -6.8123289e+18  6.6760824e+19]
 ...
 [ 9.8000002e+00  9.6040001e+01  9.4119208e+02 ...  6.9513558e+17
   6.8123289e+18  6.6760824e+19]
 [ 9.8999996e+00  9.8009995e+01  9.7029889e+02 ...  8.3451318e+17
   8.2616803e+18  8.1790629e+19]
 [ 1.0000000e+01  1.0000000e+02  1.0000000e+03 ...  9.9999998e+17
   1.0000000e+19  1.0000000e+20]]
<NDArray 201x20 @cpu(0)>
```