9/10

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).

Completed by Derek Topper

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 (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 [40]:
```
import time
tic = time.time()
import numpy as np
A = nd.random.normal(0, 1, shape = (4096, 4096))
B = nd.random.normal(0, 1, shape = (4096, 4096))
print("1. Matrix A: ", A)
print("1. Matrix B: ", B)
```

```
1. Matrix A:
[[ 0.56771386  0.7645655   0.11443903 ... -0.36707944  0.6214844
  -0.25809753]
 [-0.49916062 -1.4440985  -0.4240666  ...  1.9710336   1.2054701
  -0.3057277 ]
 [ 1.9037348  -1.2539812   0.29832017 ... -1.8558106  -0.5498901
  -2.0836408 ]
 ...
 [-0.40491968 -1.0990205  -0.20398566 ...  0.551011    0.7903397
  -0.3774016 ]
 [ 0.30279776 -0.49025026 -0.46261027 ...  0.83185124  0.56484663
   1.0392649 ]
 [ 2.6575677   1.1626844  -1.2029338  ... -0.28907916 -0.61258596
   1.2751752 ]]
<NDArray 4096x4096 @cpu(0)>
1. Matrix B:
[[-1.442437    0.9651673   0.56613606 ...  1.5337075   0.79065764
  -0.35848466]
 [ 0.6879258  -1.0146283  -1.1928986  ...  0.16327435 -0.27324632
   0.2036404 ]
 [-0.5992413  -0.6638654   0.41060147 ... -0.40307128  0.18503273
   0.02919667]
 ...
 [-0.8979426  -0.10159507 -0.58966035 ...  0.89585197  0.12146354
  -0.07107061]
 [-0.8536866   2.0668826   0.3119022  ... -0.20358665 -0.0587807
  -0.04785731]
 [-1.8013732  -1.9592597   0.41525716 ...  0.25073144  0.95281583
   0.31331718]]
<NDArray 4096x4096 @cpu(0)>
```

In [38]:
```
tic = time.time()
C = nd.dot(A, B)
C.wait_to_read()
print("Time for Part 2: ", time.time() - tic)
```

```
Time for Part 2:  3.4846460819244385
```

In [39]:

```
tic = time.time()
C = nd.empty((4096,4096))
for each in np.arange(4096):
    C[:,each] = nd.dot(A, B[:,each])
C.wait_to_read()
print("Time for Part 3: ", time.time() - tic)
```

```
Time for Part 3:  133.61406064033508
```

In [41]:

```
tic = time.time()
C = nd.empty((4096,4096))
for each in np.arange(4096):
    for each2 in np.arange(4096):
        C[each,each2] = nd.dot(A[each,:], B[:,each2])
C.wait_to_read()
print("Time for Part 4: ", time.time() - tic)
```

```
Time for Part 4:  31512.04070663452
```

5. It gets faster on GPU. A GPU accelerates the speed of computing, which would benefit us in this context.

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?

ANSWER:

1) If $B = ADA^\top$ is a positive semidefinite matrix, then we need to make sure one of two things is occuring. Either the diagonal entries of the matrix are nonnegative or the $A^\top DA$ is nonnegative.

So we can say $x^\top A^\top DAx = (Ax)D(Ax)^\top$.

Since D is assumed to have nonnegative entries, this can drop to zero only when Ax = 0.

Since A is assumed to have independent columns, Ax = 0 only happens when x = 0. Thus $A^\top DA$ is positive and is positive definite.

Additionally, another way to prove this, is that we could alter $ADA^\top$ to its vector form as $\sum A_i^\top D_i A_i$, which we know has to be nonnegative. Then We then could change that term to be $\sum A_i^2 D_i$, since A^2 must be nonnegative. Knowing this and combining it with the fact that D has nonneagtive entries, we know that $\sum A_i^2 D_i$ must be greater or equal to zero and thus $B = ADA^\top$ is positive semidefinite as well.

2) It would be useful to work with B much of the time. When we want to do a problem using every value in a matrix, then we use the real, full matrix. But, since we know B is diagonzable and symmetric, we can preform eigendecomposition on it, as shown earlier.

-1 {

However, decomposing a matrix, in terms of its eigenvalues (D) and its eigenvectors (A and $A^\top$) lets us do certain matrix calculations, like computing the power of the matrix, easier and faster when we use the eigendecomposition of the matrix. This can help eliminate inefficiencies and redundancies, over just using B for something.

ADA^T isn't necessarily the eigen-decomp.
↪ A isn't orthogonal

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.

In [2]:
```
!nvidia-smi
```

```
Tue Jan 29 09:18:06 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.44                 Driver Version: 396.44                    |
|-------------------------------+----------------------+----------------------+
| 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 K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   65C    P8    31W / 149W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

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

✓

In [4]:
```
from mxnet import ndarray as nd
x = nd.random.normal(0, 1, shape = (2, 2))
x
```

Out[4]:
```
[[2.2122064 0.7740038]
 [1.0434403 1.1839255]]
<NDArray 2x2 @cpu(0)>
```

This was done in Google Colab, using their GPU software.

In [1]:

```
!pip install mxnet-cu92 #To Make Google Colab Work
!pip install d2l
!nvidia-smi
```

```
al/lib/python3.6/dist-packages (from ipython->jupyter-console->jupyter->d2
l) (4.6.0)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.6/dist-pac
kages (from prompt-toolkit<2.1.0,>=2.0.0->jupyter-console->jupyter->d2l)
 (0.1.7)
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/py
thon3.6/dist-packages (from nbformat->notebook->jupyter->d2l) (2.6.0)
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.
6/dist-packages (from jinja2->notebook->jupyter->d2l) (1.1.0)
Requirement already satisfied: ptyprocess; os_name != "nt" in /usr/local/li
b/python3.6/dist-packages (from terminado>=0.3.3; sys_platform != "win32"->
notebook->jupyter->d2l) (0.6.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.6/dis
t-packages (from bleach->nbconvert->jupyter->d2l) (0.5.1)
Building wheels for collected packages: d2l
  Running setup.py bdist_wheel for d2l ... done
  Stored in directory: /root/.cache/pip/wheels/c7/87/29/22170afbd70e10df77b
e0339d4e5863f452faa4a2f37ed979f
Successfully built d2l
ipython 5.5.0 has requirement prompt-toolkit<2.0.0,>=1.0.4, but you'll have
```

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 [32]:

```
import time
tic = time.time()
import numpy as np
A = nd.random.normal(0, 1, shape = (4096, 4096))
B = nd.random.normal(0, 1, shape = (4096, 4096))
```

In [34]:

```
tic = time.time()
C = np.zeros(4096)

for each in range(4096):
    C[each] = (nd.dot(A, B[:, each]).norm()**2).asscalar()

print("1: Direct to NPArray: ", time.time() - tic)
```

✓

```
1: Direct to NPArray:  136.09623432159424
```

In [36]:

```
tic = time.time()
d = nd.empty(4096)
for each in np.arange(4096):
    nd.norm(nd.dot(A, B[:,each]), out=d[each]**2)
    #d[each] = d[each]**2
d.wait_to_read()
C = d.asnumpy()
print("2: NDArray Storage: ", time.time() - tic)
```

✓

```
2: NDArray Storage:  129.683664560318
```

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 [28]:

```
A = nd.arange(16).reshape(4,4)
B = nd.arange(16).reshape(4,4)
C = nd.arange(16).reshape(4,4)

nd.elemwise_add(nd.dot(A,B), C, out=C)
```

✓

Out[28]:

```
[[ 56.  63.  70.  77.]
 [156. 179. 202. 225.]
 [256. 295. 334. 373.]
 [356. 411. 466. 521.]]
<NDArray 4x4 @cpu(0)>
```

This meets the criteria outlined.

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 [37]:

```
j = np.arange(1,21)
x = np.arange(-10, 10.1, .1)
x[:, np.newaxis]**j
```

✓

Out[37]:

```
array([[-1.00000000e+01,  1.00000000e+02, -1.00000000e+03, ...,
          1.00000000e+18, -1.00000000e+19,  1.00000000e+20],
       [-9.90000000e+00,  9.80100000e+01, -9.70299000e+02, ...,
          8.34513761e+17, -8.26168624e+18,  8.17906938e+19],
       [-9.80000000e+00,  9.60400000e+01, -9.41192000e+02, ...,
          6.95135331e+17, -6.81232624e+18,  6.67607972e+19],
       ...,
       [ 9.80000000e+00,  9.60400000e+01,  9.41192000e+02, ...,
          6.95135331e+17,  6.81232624e+18,  6.67607972e+19],
       [ 9.90000000e+00,  9.80100000e+01,  9.70299000e+02, ...,
          8.34513761e+17,  8.26168624e+18,  8.17906938e+19],
       [ 1.00000000e+01,  1.00000000e+02,  1.00000000e+03, ...,
          1.00000000e+18,  1.00000000e+19,  1.00000000e+20]])
```

In [ ]: