homework1

January 29, 2019

1 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 [11]: from mxnet import ndarray as nd
         import numpy as np
         import mxnet as mx
         from mxnet import nd
         from mxnet.gluon import nn
```

1.1 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?

1. Construct two new 4096*4096 matrices, called A and B.

```
In [2]: A = nd.random.normal(0, 1, shape = (4096, 4096))
        B = nd.random.normal(0, 1, shape = (4096, 4096))
```

2. Compute **C = AB** using matrix-matrix operations and report the time.

```
In [3]: import time
        tic = time.time()
        C = nd.dot(A, B.T)
        C.wait_to_read()
        print(time.time() - tic)
```

```
2.11325740814209
```

3. Compute **C = AB** treating A as a matrix but computing the result for each column of B one at a time.

```
In [4]: tic = time.time()
        C = nd.zeros((4096, 4096))
        D = B.T
        for i in range(0, 4096):
            C[i] = nd.dot(A, D[i])
        C = C.T
        C.wait_to_read()
        print(time.time() - tic)
```

```
46.52561640739441
```

4. Compute **C = AB** treating A and B as collections of vectors. Report the time.

```
In [5]: tic = time.time()
        C = nd.zeros((4096, 4096))
        for j in range(0, 4096):
            for i in range(0, 4096):
                C[j][i] = nd.dot(A[j], D[i])
        C.wait_to_read()
        print(time.time() - tic)
```

```
1832.019658088684
```

1.2 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. **Proof** To show $B = ADA^T$ is a positive semidefinite matrix, we must take an arbitrary vector $z \in \mathbb{R}^n$, such that $z^T Bz$ are non-negative.

   Since we know that $A \in \mathbb{R}^{m \times n}$ is an arbitrary matrix, and we know that $z^T Bz = z^T ADA^T z$, it is easily to see that $(A^T z)^T = z^T A$, where $A^T z \in \mathbb{R}^n$ is also an arbirtrary vector.

   Let $C = A^T z$, then \$ C^T = z^{T}A\$, we have $z^T Bz = z^T ADA^T z = C^T DC$, where $C \in \mathbb{R}^n$.

   Since $C^T DC = \begin{bmatrix} c_1 & c_2 & c_3 & \dots & c_n \end{bmatrix} \begin{bmatrix} d_{11} & 0 & 0 & \dots & 0 \\ 0 & d_{22} & 0 & \dots & 0 \\ \dots \\ 0 & 0 & 0 & \dots & d_{nn} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \dots \\ c_n \end{bmatrix} = c_1^2 d_{11} + c_2^2 d_{22} + \cdots + c_n^2 d_{nn}$.

   Each term of the result $c_i^2 d_{ii} \geq 0$, so $c_1^2 d_{11} + c_2^2 d_{22} + \cdots + c_n^2 d_{nn} \geq 0$.

   Thus $z^T Bz \geq 0$ is true for arbitrary $z \in \mathbb{R}^n$, B is a positive semidefinite matrix.

2. It is useful to use B when we want to store the result for future calculation, but it is easier to use A and D when we want to calculate $z^T Bz$, and we can use the stored A to get $C = A^T z$, then we can get C^T, so it is easier to calculate $C^T DC$.

1.3 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 [5]: !nvidia-smi
```

```
Tue Jan 29 20:09:05 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 387.26                 Driver Version: 387.26                    |
|-------------------------------+----------------------+----------------------+
| 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  | 00000000:00:1E.0 Off |                    0 |
| N/A   50C    P0    59W / 149W |      0MiB / 11439MiB |     99%      Default |
+-------------------------------+----------------------+----------------------+

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

```
In [12]: mx.cpu(), mx.gpu(), mx.gpu(1)
```

```
Out[12]: (cpu(0), gpu(0), gpu(1))
```

```
In [13]: x = nd.ones((2, 2), ctx=mx.gpu())
         x
```

```
Out[13]:
         [[1. 1.]
          [1. 1.]]
         <NDArray 2x2 @gpu(0)>
```

1.4 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 [6]: A = nd.random.normal(0, 1, shape = (4096, 4096))
        B = nd.random.normal(0, 1, shape = (4096, 4096))
```

```
In [7]: tic = time.time()
        c = np.zeros((4096, 1))
        B = B.T
        for i in range(0, 4096):
            c[i, 0] = nd.dot(A, B[i]).norm().asscalar()
        print(time.time() - tic)
```

```
47.323471784591675
```

```
In [8]: tic = time.time()
        d = nd.zeros((4096, 1))
        for i in range(0, 4096):
            d[i, 0] = nd.dot(A, B[i]).norm()
        c = d.asnumpy()
        print(time.time() - tic)
```

```
59.10517883300781
```

1.5 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 [9]:
```
A = nd.random.normal(0, 1, shape = (4096, 4096))
B = nd.random.normal(0, 1, shape = (4096, 4096))
C = nd.zeros((4096, 4096))

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

Out[9]:
```
[[ -46.71145     -45.413414    -31.004791   ...   50.940186
   106.2559     -172.54005    ]
 [  -6.4245415   -12.3445835   -63.299656   ...    9.028375
    -1.4529381    -7.109889   ]
 [  66.85704     -34.95294     138.59096    ...   51.894375
  -169.08273     -66.29572    ]
 ...
 [  29.455101    148.26987      48.858917   ...  -49.052483
   -99.60325     -78.30951    ]
 [-101.75108     -74.69542      50.57899    ...   49.34268
   -45.483482     22.58878    ]
 [  -1.4139974    24.427887     49.047188   ...   70.21543
    -0.81529427  -56.932446   ]]
<NDArray 4096x4096 @cpu(0)>
```

1.6 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 [10]:
```
x = nd.arange(201)
x = x/10 - 10
x = x.repeat(20)
x = x.reshape((201, 20))

y = nd.arange(20)
y = y + 1
y = y.repeat(201)
y = y.reshape((20, 201))
```

```python
y = y.T

A = x ** y
A
```

Out[10]:

```
[[-1.0000000e+01  1.0000000e+02 -1.0000000e+03 ...  9.9999998e+17
  -1.0000000e+19  1.0000000e+20]
 [-9.8999996e+00  9.8009995e+01 -9.7029895e+02 ...  8.3451338e+17
  -8.2616820e+18  8.1790647e+19]
 [-9.8000002e+00  9.6040001e+01 -9.4119208e+02 ...  6.9513558e+17
  -6.8123289e+18  6.6760824e+19]
 ...
 [ 9.7999992e+00  9.6039986e+01  9.4119177e+02 ...  6.9513434e+17
   6.8123162e+18  6.6760692e+19]
 [ 9.8999996e+00  9.8009995e+01  9.7029895e+02 ...  8.3451338e+17
   8.2616820e+18  8.1790647e+19]
 [ 1.0000000e+01  1.0000000e+02  1.0000000e+03 ...  9.9999998e+17
   1.0000000e+19  1.0000000e+20]]
<NDArray 201x20 @cpu(0)>
```