# homework1_solutions

February 6, 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 [1]: from mxnet import ndarray as nd
        import mxnet as mx
        import numpy as np
        import time
```

### 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 $4096 \times 4096$.
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]: A = nd.random.normal(shape=(4096, 4096))
        B = nd.random.normal(shape=(4096, 4096))
```

```
tic = time.time()
C = nd.dot(A, B)
C.wait_to_read()
print("Matrix by matrix: " + str(time.time() - tic) + " seconds")
```

```
Matrix by matrix: 2.2103829383850098 seconds
```

```
In [3]: C = nd.empty((4096, 4096))
        tic = time.time()
        for i in range(4096):
          C[:, i] = nd.dot(A, B[:, i])

        C.wait_to_read()
        print("Matrix by vector: " + str(time.time() - tic) + " seconds")
```

```
Matrix by vector: 15.260442972183228 seconds
```

```
In [4]: C = nd.empty((4096, 4096))
        tic = time.time()
        for i in range(4096):
            for j in range(4096):
                C[i,j] = nd.dot(A[i, :], B[:, j])

        C.wait_to_read()
        print("Vector by vector: " + str(time.time() - tic) + " seconds")
```

```
Vector by vector: 6093.727718114853 seconds
```

```
In [ ]:
```

### 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. $B$ is positive semidefinite if for all $x \in \mathbb{R}^m$, $x^T Bx \geq 0$. Then let $x \in \mathbb{R}^m$ and let $y = A^T x$. Then
$$x^T Bx = x^T ADA^T x = (A^T x)^T DA^T x = y^T Dy = \sum_1^n d_i y_i^2 \geq 0 \tag{1}$$
2. It would be more useful to work with $B$ if $m << n$ as matrix multiplication for instance for two arbitrary matrices $X$ and $Y$ with dimesnions $a$ by $b$ and $b$ by $c$ is a runtime of $O(abc)$. If we multiply $B$ by a matrix $C$ that is $m$ by $k$, then $BC$ is computed in $O(m^2 k)$. This matrix multiplication with $ADA^T C$ is $O(2mnk + n^2 k)$. Thus if $m << n$, then it would be more efficient to use $B$ and if $n >> m$, then it would be better to use $A$ and $D$.

## 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 $2 \times 2$ matrix on the GPU and print it. See http://d2l.ai/chapter_deep-learning-computation/use-gpu.html for details.

```
In [6]: !nvidia-smi
```

```
Tue Jan 29 05:07:49 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.81                 Driver Version: 384.81                    |
|-------------------------------+----------------------+----------------------+
| 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   68C    P0   129W / 149W |    496MiB / 11439MiB |     74%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     21488      C   ...ubuntu/miniconda3/envs/gluon/bin/python   485MiB |
+-----------------------------------------------------------------------------+
```

```
In [7]: x = nd.ones((2, 2), ctx=mx.gpu())
        x
```

```
Out[7]:
        [[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 $4096 \times 4096$ 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 **d** in NDArray for assignments and copy to NumPy at the end.

```
In [5]: A = nd.random.normal(shape=(4096, 4096))
        B = nd.random.normal(shape=(4096, 4096))
        c = np.empty(4096)

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

        print("One at a time: " + str(time.time() - tic) + " seconds")
```

```
One at a time: 19.583129167556763 seconds
```

```
In [6]: d = nd.empty(4096)
        tic = time.time()
        for i in range(4096):
          d[i] = nd.norm(nd.dot(A, B[:, i]))**2

        c = d.asnumpy()
        print("Convert at end: " + str(time.time() - tic) + " seconds")
```

```
Convert at end: 16.5849871635437 seconds
```

### 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 [ ]: A = nd.random.normal((100,100))
        B = nd.random.normal((100,100))
        C = nd.random.normal((100,100))

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

### 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]:
```python
x = nd.arange(start=-10, stop=10.1, step=0.1).reshape((201, 1))
j = nd.arange(start=1, stop=21, step=1.0).reshape((1, 20))
print(x ** j)
```

```
[[-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]
 ...
 [ 9.8000011e+00  9.6040024e+01  9.4119232e+02 ...  6.9513681e+17
   6.8123409e+18  6.6760952e+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)>
```

In [ ]: