homework1

January 28, 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
```

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?

```
In [2]: import time

        # 1
        A = nd.random.normal(loc=0, scale=1, shape=[4096, 4096])
        B = nd.random.normal(loc=0, scale=1, shape=[4096, 4096])
```

```
# 2
t = time.time()
C = nd.dot(A, B)
C.wait_to_read()
print(time.time() - t)

# 3
t = time.time()
columns = [nd.dot(A, B[:, i]).reshape(4096, 1) for i in range(4096)]
C = nd.concat(*columns, dim=1)
C.wait_to_read()
print(time.time() - t)

# # 4
# t = time.time()
# C = nd.empty(shape=[4096, 4096])
# for i in range(4096):
#     for j in range(4096):
#         C[i, j] = nd.dot(A[i, :], B[:, j])
# C.wait_to_read()
# print(time.time() - t)
print("The last one takes very long - Timeout")
```

```
1.0074262619018555
28.05392622947693
The last one takes very long - Timeout
```

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. Let x be a m-dimensional vector, and let C be a diagonal matrix containing the square roots of the diagonal entries in D (so $C^2 = D$). Then $x^T B x = x^T A D A^T x = \|CA^T x\|^2 >= 0$, and this completes the proof that B is positive semidefinite.

2. It would be useful to work with A and D if you're doing computations that may require the square root of B or different powers of B (because then you can exponentiate D). Otherwise, just using B by itself would be faster.

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 [3]: !nvidia-smi
```

```
Tue Jan 29 00:41:08 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 V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   42C    P0    38W / 300W |      0MiB / 16152MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

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

```
In [4]: x = nd.ones((2, 2), ctx=mx.gpu())
        x
```

```
Out[4]:
        [[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 [5]: import numpy as np
        A = nd.random.normal(loc=0, scale=1, shape=[4096, 4096])
        B = nd.random.normal(loc=0, scale=1, shape=[4096, 4096])

        t = time.time()
        c = np.zeros(4096)
        for i in range(4096):
            c[i] = nd.norm(nd.dot(A, B[i]), ord=2).asscalar()
        print(time.time() - t)

        t = time.time()
        d = nd.zeros(4096)
        for i in range(4096):
            d[i] = nd.norm(nd.dot(A, B[i]), ord=2)
        d = d.asnumpy()
        print(time.time() - t)
```

```
29.546343564987183
28.222331762313843
```

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 [6]: # Assuming C already exists from q1, gets overwritten here
        nd.elemwise_add(nd.dot(A, B), C, out=C)
```

```
Out[6]:
        [[ -89.2842      73.64483    -25.558414  ...   90.13895     95.803665
          -162.3457   ]
         [ -37.078636   -6.3714905   11.097342  ...   44.199265   -45.428867
            78.536705 ]
         [ 134.2446    214.59921     68.38664   ...  -66.71899    -20.114132
            40.64216   ]
         ...
         [ -95.55035    41.435562   122.75974   ...   69.42801    -94.71486
            46.208897 ]
         [-231.5051    -25.642136   -83.37668   ...  -16.456684  -135.8109
           -83.9735   ]
         [  75.27684    -1.9213676   21.20346   ...   81.998985    -7.3830795
           -84.509315 ]]
        <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 [4]: x = nd.arange(-10, 10.1, 0.1).reshape(201, 1)
        js = nd.arange(1, 21)
        A = nd.power(x, js)
        A
```

```
Out[4]:
        [[ -1.00000000e+01   1.00000000e+02  -1.00000000e+03 ...,   9.99999984e+17
           -9.99999998e+18   1.00000002e+20]
         [ -9.89999962e+00   9.80099945e+01  -9.70298889e+02 ...,   8.34513176e+17
           -8.26168034e+18   8.17906293e+19]
         [ -9.80000019e+00   9.60400009e+01  -9.41192078e+02 ...,   6.95135578e+17
           -6.81232885e+18   6.67608243e+19]
         ...,
         [  9.80000114e+00   9.60400238e+01   9.41192322e+02 ...,   6.95136815e+17
            6.81234150e+18   6.67609519e+19]
         [  9.89999962e+00   9.80099945e+01   9.70298889e+02 ...,   8.34513176e+17
            8.26168034e+18   8.17906293e+19]
         [  1.00000000e+01   1.00000000e+02   1.00000000e+03 ...,   9.99999984e+17
            9.99999998e+18   1.00000002e+20]]
        <NDArray 201x20 @cpu(0)>
```