Numpy Basics

CSE 4311 – Neural Networks and Deep Learning

Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

The numpy Library

- The numpy library implements various functions useful in mathematics, statistics, science and engineering.
- On Google Colab (and on Anaconda) it is available, you just need to import it.

```
import numpy as np
```

- Files numpy_examples.py and numpy_benchmarks.py are posted on the material for this lecture. They contain the code that you will see in the next slides.

Creating Arrays

- Creating an 1-dimensional array of ints from a list:

```
array_1d = np.array([10, 5, 2, 4])
print("array_1d:", array_1d)
print("array_1d data_type:", array_1d.dtype)
print("array_1d shape:", array_1d.shape)
```

Output:

```
array_1d: [10  5  2  4]
array_1d data_type: int64
array_1d shape: (4,)
```

Creating Arrays

- Creating a 2-dimensional array of floats from a list:

```
array_2d = np.array([[1,2,3], [4,5,6]], dtype='float')
print("array_2d:\n", array_2d, "\n")
print("array_2d data_type:", array_2d.dtype)
print("array_2d shape:", array_2d.shape)
```

Output:

```
array_2d:
 [[1. 2. 3.]
 [4. 5. 6.]]

array_2d data_type: float64
array_2d shape: (2, 3)
```

Creating Arrays

- Creating a zero-initialized array of a specified shape:

```
array_2d = np.zeros((3,4))
print("array_2d:\n", array_2d, "\n")
print("array_2d data_type:", array_2d.dtype)
print("array_2d shape:", array_2d.shape)
```

Output:

```
array_2d:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

array_2d data_type: float64
array_2d shape: (3, 4)
```

Creating Arrays

- Creating a 2D array of random values uniformly distributed between 0 and 1:

```
(rows, cols) = (3, 4)
random_values = np.random.rand(rows, cols)
print("random_values:\n", random_values, "\n")
print("random_values data_type:", random_values.dtype)
print("random_values shape:", random_values.shape)
```

Output:

```
random_values:
 [[0.63075311 0.21212589 0.88126268 0.22670053]
 [0.89718472 0.19447788 0.42529485 0.29785337]
 [0.38629528 0.04719182 0.59240337 0.72252642]]

random_values data_type: float64
random_values shape: (3, 4)
```

Formatted Printing of Arrays

- We use numpy.set_printoptions.
  - There are a lot of options, see documentation.
- Here, we specify to use four decimal digits.

```
np.set_printoptions(precision = 4)
print("random_values:\n", random_values*200, "\n")
```

Output:

```
random_values*200:
 [[ 30.2748 124.8525  74.0704 196.1591]
 [143.4591 169.1471  79.427    7.814 ]
 [ 34.0032 181.9523  99.1757  15.1455]]
```

Formatted Printing of Arrays

- We use numpy.set_printoptions.
  - There are a lot of options, see documentation.
- Here we print each value using the printf-like "%9.f" format.
  - 9 spaces minimum, 4 decimal digits.

```
np.set_printoptions(formatter={'all':lambda x: "%9.4f"
% (x)})
print("random_values*200:\n", random_values*200, "\n")
```

Output:

```
random_values*200:
 [[  30.2748  124.8525   74.0704  196.1591]
 [ 143.4591  169.1471   79.4270    7.8140]
 [  34.0032  181.9523   99.1757   15.1455]]
```

Means and Standard Deviations

- Compute mean and standard deviation of values in a matrix.
  - Notice use of "printf-style" formatting of output.

```
print("random_values mean value = %.4f, std = %.4f" %
      (np.mean(random_values), np.std(random_values)))
```

Output:

```
random_values mean value = 0.4595, std = 0.2699
```

Operations along Specific Dimensions

- You can do many operations along a specific dimension
- Here we compute the mean of every row in the matrix.

```
row_means = np.mean(random_values, 1)
print("row means:", row_means)
```

Output:

```
row means: [   0.4877    0.4537    0.4371]
```

Operations along Specific Dimensions

- You can do many operations along a specific dimension
- Here we compute the mean of every column in the matrix.

```
col_means = np.mean(random_values, 0)
print("col means:", col_means)
```

Output:

```
col means: [   0.6381    0.1513    0.6330    0.4157]
```

Inverting a Matrix

- To invert a matrix, use np.linalg.pinv
  - pinv stands for “pseudoinverse”
  - Do not use np.linalg.inv, it is far more prone to numerical problems.

```
a = np.array([[2.1, 3.2], [1.6, 5.3]])
print("a:\n", a, "\n")
inv_a = np.linalg.pinv(a)
print("inverse of a:\n", inv_a, "\n")
```

Output:

```
a:
 [[   2.1000    3.2000]
 [   1.6000    5.3000]]

inverse of a:
 [[   0.8819   -0.5324]
 [  -0.2662    0.3494]]
```

Pointwise Multiplication

- Regrettably (for me), a * b in numpy does element-wise multiplication, and NOT matrix multiplication.
  - In the code below, for each i and j:

    r1[i,j] = a[i,j] * inv_a[i,j].

```
r1 = a*inv_a
print("r1:\n", r1, "\n")
```

Output:

```
r1:
 [[   1.8519   -1.7038]
 [  -0.4260    1.8519]]
```

Matrix Multiplication

- Use np.dot for matrix multiplication.

```
r2 = np.dot(a, inv_a)
print("r2:\n", r2, "\n")
```

Output:

```
r2:
 [[   1.0000    0.0000]
 [  -0.0000    1.0000]]
```

Array Operations vs. Loops

- Array operations in numpy are optimized.
- For large arrays or matrices, using a built-in numpy operation can be as fast as in C.
- Writing your own code for those operations, using loops to access each element of the arrays and the result, can be hundreds of time slower.
- Two examples are illustrated in file python_benchmarks.py

Benchmark 1: Matrix Addition

```
# my function, adds two matrices element by element
# using loops.
def matrix_addition(first, second):
    (rows1, cols1) = first.shape
    result = np.zeros((rows1, cols1))

    for i in range(0, rows1):
        for j in range(0, cols1):
            result[i,j] = first[i,j] + second[i,j]

    return result
```

Benchmark 1: Matrix Addition

```
rows = 5000
cols = rows
A = np.random.rand(rows, cols)
B = np.random.rand(rows, cols)

# using my function with loops.
result = matrix_addition(A, B)

# using the numpy + operator
result = A + B
```

- My function took 18.59 seconds on Google Colab.
- The numpy version took 0.141 seconds (131 times faster).

Parenthesis: Measuring Time

- If you want to measure execution time, this example shows one way to do it:

```
import time

start_time = time.time_ns()
result = matrix_addition(A, B)
end_time = time.time_ns()
#print(result)

total_time = (end_time-start_time) / 1000000000
```

Benchmark 2: Matrix Multiplication

```
# My function, multiplies matrices using loops.
def matrix_multiplication(first, second):
    (rows1, cols1) = first.shape
    (rows2, cols2) = second.shape
    result = np.zeros((rows1, cols2))

    for i in range(0, rows1):
        for j in range(0, cols2):
            sum = 0
            for k in range(0, rows2):
                sum = sum + first[i,k]*second[k,j]
            result[i,j] = sum

    return result
```

Benchmark 2: Matrix Multiplication

```
rows = 500
cols = rows
A = np.random.rand(rows, cols)
B = np.random.rand(rows, cols)

# using my function with loops.
result = matrix_multiplication(A, B)

# using the np.dot function.
result = np.dot(A, B)
```

- My function took 55.68 seconds on Google Colab.
- The numpy version took 0.00958 seconds (5800 times faster).