CS 4820

Tutorial 0: Python and Linear Algebra Refresher

This course heavily uses Tensorflow (and Teras). Tensorflow is an open source deep learning library created by Google. Currently it is the most popular and widely used library. However, before touching Tensorflow, we all need to be reasonably familiar with basic Python and linear algebra, which are what Tensorflow depends on.

So, this tutorial is a refresher on Python 3 and basic linear algebra. It also introduces Numpy, the standard Python library for manipulating arrays algebraically.

Jupyter notebook

In Jupyter notebook, each cell is of one of a few basic types: Markdown, code, raw NBConvert, and Heading.

Also, there are two modes for each cell: command mode and edit mode.

keyboard shortcuts (http://www.tetraph.com/blog/machine-learning/jupyter-notebook-keyboard-shortcut-command-mode-edit-mode/)

1. Python

Hello World

Printing in Python can be done with the `print` function.

In [4]:
```
print("Hello World!")
```

```
Hello World!
```

Libraries

Python has a number of built-in modules and libraries that offer convenient access to useful functions. These libraries can be imported by using the built-in `import` function followed by the library name.

Here is one example with the `random` library that can be used for generating a series of random integers within some specified range. Note that `for i in range(5)` is analogous to `for (int i = 0; i < 5; i++)` in Java.

In [ ]:
```
import random
for i in range(5):
    print(random.randint(10,99))
```

Indentation

Notice that Python uses indentation and colons in order to specify scope, as opposed to brackets. This means that you need to be careful to make sure that all of your code is indented correctly.

In [ ]:
```
x = 0

while x < 10:
    if x % 2 == 0:
        print(x)
    x += 1

print('done.')
```

Dynamic Typing

In Python, variables are associated with single objects and no data types. Furthermore, primitive data types in Python are immutable.

In [ ]:
```
var = 5
print(var)
print(type(var))

var = 3.2
print(var)
print(type(var))

var = 'spam'
print(var)
print(type(var))
```

Strings

Python supports strings along with the expected indexing schema and methods.

In [11]:
```
mystring = 'ham and eggs'
print(mystring[0:5]) # note that the first index is inclusive and the secon
print(mystring.find('and'))
print(mystring.split(' '))
```

```
ham a
4
['ham', 'and', 'eggs']
```

Lists

Lists/arrays are mutable objects in Python.

In [12]:
```
mylist = [1, 2]
mylist.append("three")

print(mylist)

newlist = [1, 1, 2, 3, 5, 8, 13]
newlist[4] = 3000
print(newlist[4])
print(newlist[-1]) # get the last item in the list
newlist.pop()
print(newlist)
```

```
[1, 2, 'three']
3000
13
[1, 1, 2, 3, 3000, 8]
```

Tuples

Tuples are like immutable lists.

In [3]:
```
tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')

try:
    tup1[0] = 100;
except TypeError:
    print('See why this returns an error?')


tup3 = tup1 + tup2 # this concatenates the two tuples
print(tup3)
print('length of tup3:', len(tup3))
for x in tup3: print(x)
```

```
See why this returns an error?
(12, 34.56, 'abc', 'xyz')
length of tup3: 4
12
34.56
abc
xyz
```

Dictionaries

Python also supports dictionaries (hash maps) for mapping between specified keys and values.

In [17]:
```
numbers = {'one': 1, 'two': 2, 'three': 3, 'four': 4 }
print(numbers['one'])
del numbers['one']  # Remove an etry from the dictionary

try:
    print(numbers['one'])
except KeyError:
    print("This shouldn't work, since we deleted numbers['one'] above")

print(numbers)
print(numbers.keys())
print(numbers.values())
```

```
1
This shouldn't work, since we deleted numbers['one'] above
{'two': 2, 'three': 3, 'four': 4}
dict_keys(['two', 'three', 'four'])
dict_values([2, 3, 4])
```

Name binding

Notice that Python assignment binds a name to a particular object. In other words, objects are pass-by-reference. Primitives like integers, however, are pass-by-value.

If your goal is to make an independent clone of an object, you should use the `deepcopy` function from Python's `copy` library. Alternatively, you can use the `str` function to copy a string, `list` to copy a list, and so on.

In [19]:
```
a = [1, 2]
b = a
print(b, a)
b.append(3)
print(a)

a = 1
b = a
print(b, a)
b = b + 1
print(a)
print(b)
```

```
[1, 2] [1, 2]
[1, 2, 3]
1 1
1
2
```

Control Flow

Here are examples of if-else statements, for loops, and while loops in Python. Notice how identation controls scope in each statement.

In [20]:
```
age = 22

if age < 13:
    print('kid')
elif age < 18:
    print('teen')
else:
    print('adult')
```

```
adult
```

In [21]:
```
for i in range(5):
    pass

for i in [0, 1, 2, 3, 4]:
    if i > 5:
        break
else:
    print('Python supports the else keyword for for-loops, which execute if
```

```
Python supports the else keyword for for-loops, which execute if the loop
completes without breaking
```

In [22]:
```
x = 1024

while x > 1:
    x = x / 2
    if (x % 10) != 2:
        continue
    print(x)
```

```
512.0
32.0
2.0
```

Try-Except

Python also supports try-except statements for error handling.

In [23]:
```
try:
    1 / 0
except:
    print('Exception!')
else:
    print('No exception!')
finally:
    print('Done.')
```

```
Exception!
Done.
```

Functions

Specify functions using the `def` keyword.

In [24]:
```
def example_func(s="hello!"):
    print(s)

example_func("goodbye!")
example_func() # This is equivalent to exapmle_func("hello!") since we give
```

```
goodbye!
hello!
```

Classes

Specify classes using the `class` keyword. Notice the `__` around the first method of this class; this denotes what are more commonly referred to as ["magic methods" (http://minhhh.github.io/posts/a-guide-to-pythons-magic-methods)](http://minhhh.github.io/posts/a-guide-to-pythons-magic-methods) in Python. The magic method defined for this class is the constructor that you will need to define for all your classes.

In [25]:
```python
import math

class Vector2:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def len(self):
        return math.sqrt(self.x ** 2 +
                         self.y ** 2)

    _DoNotTouch = 10


v = Vector2(3, 4)
print("({},{}):".format(v.x, v.y), \
      "len = {}".format(v.len()))
```

```
(3,4): len = 5.0
```

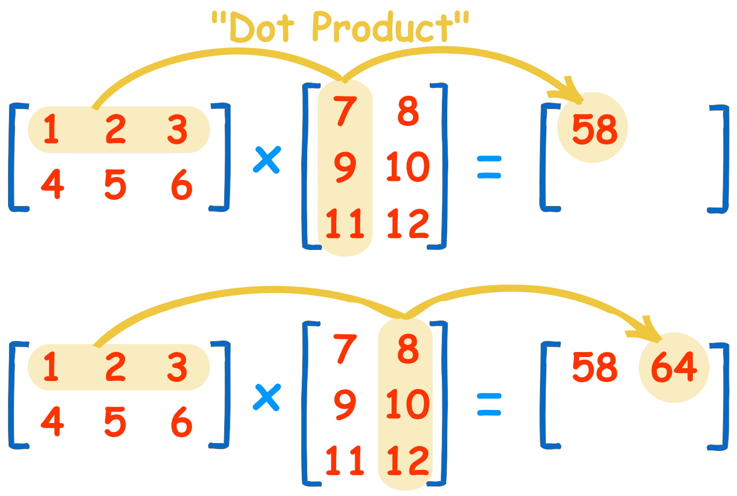
2. Linear Algebra Refresher

Before moving on to NumPy, I need to talk about my favourite type of math: Linear Algebra. Most of the operations in Deep Learning are done by matrices. It's both practical and easy to optimise using very powerful parallel hardwares like GPUs. Good news is, for the most part of this course, I only need to make sure you know about **matrix multiplications** because other matrix operations are fairly straight-forward. Let's take a look at how that works.

Given the following two matrices:

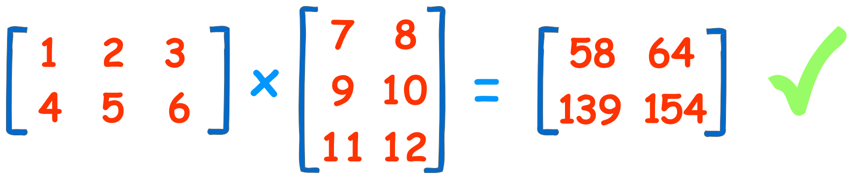
```
A = [[1,2,3],[4,5,6]]        # Shape=(2,3)
B = [[7,8],[9,10],[11,12]] # Shape=(3,2)
```

And we want to find A * B, to do this we dot the rows of A and the columns of B to find each element in AB:



$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}7 & 8\\9 & 10\\11 & 12\end{bmatrix} = \begin{bmatrix}58 & \\ & \end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}7 & 8\\9 & 10\\11 & 12\end{bmatrix} = \begin{bmatrix}58 & 64\\ & \end{bmatrix}$$



Take note of the resultant shape of the multiplication. When we have a matrix of shape (N, M) multiplied by a matrix of shape (M, V), we end up with a matrix of shape (N, V). If the last dimension of the first matrix and the first dimension of the matrix do not match, the multiplication won't work.

3. Numpy (Numeric Python)

For much of this course, you will often find yourself in need of creating, modifying, and combining n-dimensional arrays. Numpy is the standard Python library for quickly, cleanly, and efficiently performing all of these functions.

Here are just a few examples with basic Numpy arrays.

For a more in-depth view of the other useful features of Numpy, visit this tutorial (http://cs231n.github.io/python-numpy-tutorial/#numpy).

Basics

In [12]:
```
import numpy as np

a = np.array([1, 2, 3])   # Creates a rank 1 array, or a 1D array, or a vec
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array, or a 2D array,
print(type(b))                     # Prints "<class 'numpy.ndarray'>"
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"

c = np.array([[1, 2, 3]]) # Creates a row matrix
print(c.shape)            # Prints "(1, 3)" --> Make sure you understand th
                          #                     between the vector "a" and

d = np.array([[1], [2], [3]])      # Create a column matrix
print(d.shape)                     # Prints "(3, 1)"
```

```
<class 'numpy.ndarray'>
(3,)
1 2 3
[5 2 3]
<class 'numpy.ndarray'>
(2, 3)
1 2 4
(1, 3)
(3, 1)
```

Some custom functions to create arrays.

In [13]:
```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

```
[[0. 0.]
 [0. 0.]]
[[1. 1.]]
[[7 7]
 [7 7]]
[[1. 0.]
 [0. 1.]]
[[0.94816218 0.67477923]
 [0.29306738 0.96598233]]
```

Array operations.

Assuming x and y are two arrays of the same shape, basic mathematical functions operate **elementwise** on arrays, and are available both as **operator overloads** like x+y and as **functions in the numpy module** like np.add(x,y).

What if x and y are of different shapes? See the section of "broadcasting" below.

In [14]:

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce an array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce an array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce an array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce an array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces an array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))

# Reshape; changes the dimensions of a matrix
# [[ 1.  2.  3.  4.]]
x2 = np.reshape(x, [1, 4])
print(x2)

# Max; returns the value of the largest element
# 4.0
print(np.max(x2))

# Argmax; returns the index of the largest element
# 3
print(np.argmax(x2))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
[[ 5. 12.]
 [21. 32.]]
```

```
[[ 5. 12.]
 [21. 32.]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[1.         1.41421356]
 [1.73205081 2.        ]]
[[1. 2. 3. 4.]]
4.0
3
```

Matrix multiplication

In [17]:
```python
import numpy as np

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors (sum of element-wise products); both produce 219
print(v.dot(w))
print(np.dot(v, w))

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# Matmul and dot are the same for 2D operations, but differ when we increas
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.matmul(x, y))  # Preferred. Alternatively, you may also do x@y

# Matrix / matrix product; both produce the rank 2 array
z=np.array([[9],[10]])
print(x.dot(z))
```

```
219
219
[29 67]
[29 67]
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
[[29]
 [67]]
```

Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

In [39]:
```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Multiple Dimensional Matrix Multiplication

In [41]:
```
import numpy as np

# Let's say we have the following matrix A
A = np.random.random((50, 100, 20))
# We can imagine A as 50 instances of (100,20) matrices.
# We have the following matrix B
B = np.random.random((20,40))
# We want to multiply each (100,20) instance of A by B, we can do this beca
print(np.dot(A,B).shape)
# should be (50, 100, 40), each of the 50 (100,20) is multiplied by (20,40)
```

```
(50, 100, 40)
```