Deep Learning Lab

Unit 1 – Python and Numpy

Prof. Oswald Lanz

T.A. Sofia Casarin

Faculty of Computer Science

Free University of Bozen-Bolzano

Agenda

1.1 Setup utilities

1.2 Python

1.3 Numpy

1.1
Setup utilities

Virtual environments

We strongly suggest to use **virtual environments** to keep your file system organized. Environments are basically folders containing a bunch of libraries and scripts.

We recommend to use Anaconda (available for Linux, Windows and Mac) which installs Python by itself and makes the environment management easier; it also works with other programming languages. Still, you are free to use plain virtual environments if you prefer.

```
conda create -n <env_name> --python=3.9          # creates a new environment with python
3.9
conda activate <env_name>                        # activates the environment
conda deactivate                                 # deactivates the active environment
```

Package managers

Virtual environments go hand in hand with package managers. They provide the user with a vast choice of libraries, installing them effortless with a simple command.

The most used (and supported) package managers for Python are **Pip** and **Conda**. While Pip is specific for Python, Conda can also handle other programming languages.

We will use Pip for its easier, more mnemonic usage and greater support, but feel free to use Conda if preferred. **It is advised not to mix the two**: they can surely work together, but their combination might cause weird problems sometimes.

Pip package search

Anaconda package search

1.2
Python

Python

- Python is a high-level, **dynamic**, open-source, **interpreted** programming language.
- Python is **garbage-collected**: you will not care about memory allocation.
- It can be used for a variety of different application, from back-end to front-end.
- Few lines of readable Python code can convey **powerful concepts**.
- Fast to code, slow to run: still, a very good performance-complexity compromise.
- High-performing when used properly (the less for loops, the better).
- It has the greatest support for **neural networks** design and training.

Data types

Python variables **do not have a fixed type**: the latter can change dynamically during the execution of the program. You can ask for it with the built-in function *type( )*. Basic Python types include integers, floating-point numbers booleans and strings.

Numerical operators look similar to other programming languages:

```
x = 2
y = 3

print(x + y)          # 2 + 3; prints "5"
print(x - y)          # 2 - 3; prints "-1"
print(x * y)          # 2 * 3; prints "6"
print(x / y)          # 2 / 3; prints "0.666666"
x += 1                # increments x by 1

print(x ** y)         # 2 to the power of 3;
                        prints "8"
print(x // y)         # 2 integer_division 3;
                        prints "0"
print(x % y)          # 2 mod. 3 (2);
                        prints "2"
print(type(x))        #prints «int»; prints "1"
x++                   # SyntaxError: invalid
                        syntax
```

Booleans and strings

Python supports boolean operators:

```
x = True
y = False

print(x and y)  # True AND False; prints False
print(x or y)   # True OR False; prints True
print(not x)    # NOT True; prints False
print(x != y)   # True XOR False;  prints True
```

And fancy operations with strings:

```
x = "hello"     # double quotes
y = 'world'     # and single quotes are the same
z = 42

print(x + ' ' + y)      # "hello" CONCAT *blank_space* CONCAT "world"; prints "hello world"
print(f"{x} {y} {z}")   # format strings, available from python 3.6; prints "hello world 42"
print(len(x))           # string length; prints "5"
```

Lists

Python supports many operations with linked lists:

```
x = list()                                  # creates an empty list
y = []                                      # creates an empty list
z = [1, 2, "a", [4, 5], "hello", True]  # lists can contain different types, also other lists

x.append(2)                             # adds 2 to the end of the list
y = x + z                               # operator + concatenates lists
x += z                                  # also works inplace
"a" in x                                # prints True
z[2] = False                            # modify the third element
x.pop()                                 # removes and returns the last element
z[2:5]                                  # access from the third to the fifth element
z[3:]                                   # access from the fourth to the last element
y[-1]                                   # access to the last element
z[:-2]                                  # access from the first to the second-last element
z[3:5] = [1, 9]                         # modify the content of the fourth and fifth elements
z[1:6:2] = [1, 9, 5]                     # modify the content of the second, fourth and sixth elements
list(range(5))                          # creates [0, 1, 2, 3, 4]
z[::-1]                                 # selects all items in reversed order; prints [True, 9, 1, False, 2, 1]
```

Loops

For and while loops are supported as in other languages:

```
z = [1, 2, "a", [4, 5], "hello", True]

for item in z:
    print(item)         # prints every element in the list

i = 0
while i < 5:
    print(z[i])         # prints every element in the list
    i += 1

for i in range(10):
    print(i)            # prints the numbers from 0 to 10
```

List comprehensions

You can create lists on the fly:

```
x = list(range(5))
y = [item ** 2 for item in x]

print(y)         # prints [0, 1, 4, 9, 16]
```

Also, with conditions inside:

```
x = list(range(5))
y = [item ** 2 for item in x if item % 2 == 0]

print(y)         # prints [0, 4, 16]
```

Dictionaries

Dictionaries stores (key, value) pairs, they are similar to c++ and java Maps:

```
x = dict()                                  # creates an empty dict
y = {}                                      # creates an empty dict
z = {"a": "b", "c": 3}

x["key"] = "value"
x["key"]["subkey"] = "subvalue"
"key" in x                                  # prints True
x["hello"]                                  # KeyError: "hello»
```

```
d = {1:'one', 2:'two', 3:'three'}
print(d.get(3))                            # prints three
print(d.get(4))                            # prints None
print(d.get(4, 'empty'))                   # prints 'empty'
```

Sets

Sets are unordered collections of unique **immutable** elements:

```
x = set()                                   # create an empty set
animals = {"cat", "dog", "cow"}
x = {1, 3, [3, 4]}                          # TypeError: unhashable type: 'list'

animals.add("cat"); print(x)                # prints {"cat", "dog", "cow"}
animals.add("bird"); print(x)               # prints {"cat", "dog", "cow", "bird"}
"bird" in animals                           # prints True
print(len(animals))                         # prints 4
```

Sets does not preserve the order of their items.

Sets comprehensions and loops:

```
squares_below_100 = {i ** 2 for i in range(10)}
for animal in animals:
    print(animal)
```

Tuples

Tuples are **immutable** lists of values: you can store them in sets or use them as keys in a dictionary. You can use them as they were lists.

```
x = ()                                  # creates an empty tuple
y = (1, "a", (4, 8))                    # creates a tuple

z = x + y                               # operator + creates another tuple which is
                                        # the concatenation of the two
```

Tuple comprehensions do not exist: if you try guessing their syntax, you end up creating a generator:

```
print(type((i for i in range(10))))          # prints "<class 'generator'>"
for i in (i for i in range(10)): print(i)    # prints the number from 0 to 9
                                             # also, it does not make sense
```

Functions

You can define functions with the **def** keyword:

```
def power(base, exponent):
    ret = 1
    if exponent >= 0:
        for i in range(exponent):
            ret *= base
    else:
        for i in range(-exponent):
            ret /= base
    return ret

print(power(2, 7))      # prints 128
print(power(180, 45))              # prints 307087917757367862301953030734556160971465480105589997568
```

Note that integer numbers in python can be as long as you want.

Functions

Functions can accept optional arguments:

```
def greet(greeting, arabic=False):
    if arabic:
        print(greeting[::-1])
    else:
        print(greeting)

print("hello")                  # prints "hello"
print("hello", arabic=True)     # prints "olleh"
```

Classes

You can define classes with the **class** keyword. you can use inheritance:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self, loud=False):
        pass

class Dog(Animal):
    def __init__(self, name):
        super(Dog, self).__init__(name)

    def speak(self, loud=False):
        if loud:
            print("WOOF")
        else:
            print("woof")
```

Exercise

Python is easy! Let’s try to code a Sudoku game from the terminal.

You can find a crappy user interface here: https://we.tl/t-6fKVh5e3S2

Also, if you never played Sudoku before: Sudoku wiki

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

1.2
Numpy

Numpy

- Numpy is a Python library (written in C) for **array-oriented programming**.
- It supports **multidimensional arrays** and allows almost every operation with them.
- It has become the foundation of the Scientific Python Stack.

NumPy

Arrays

Numpy arrays are sequences of values that are arranged in **dimensions**, homogeneous in type and **contiguous in memory**.
Single-dimensional sequence of values are commonly referred to as **arrays**, two-dimensional ones are known as **matrices**. If the dimensions are more than two, we will always refer to these objects as **arrays** (as it is the name of the class from Numpy).

```
import numpy as np

x = np.array([1, 2, 3, 4])                  # creates a single-dimensional array
print(x.shape)                              # prints (4,)
print(x.dtype)                              # prints dtype(int64)
y = np.array([[1, 2.], [3, 4])              # creates a matrix
print(y.shape)                              # prints (2, 2)
print(y.dtype)                              # prints dtype(float64)
x[0] = 2.7                                  # set the first value to 2 (truncates the value)
x = np.array([1, 10, 4, 8], dtype=np.uint8) # creates a single-dimensional array with
                                            # dtype 'unsigned 8 bits integer'
```

Useful initializations

```
np.arange(10)                    # array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
np.linspace(0, 1, 6)             # array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])

np.zeros((2, 5))                 # array([[0., 0., 0., 0., 0.],
                                 #        [0., 0., 0., 0., 0.]])

np.ones((2, 4))                  # array([[1., 1., 1., 1.],
                          #        [1., 1., 1., 1.]])

np.empty((1, 3))                           # array([[3.9486e-320, 0.0000e+000, 1.5810e-322]])

np.eye(4)                                  # array([[1., 0., 0., 0.],
                          #        [0., 1., 0., 0.],
                          #        [0., 0., 1., 0.],
                          #        [0., 0., 0., 1.]])

np.diag([1, 2, 3])               # array([[1, 0, 0],
                          #        [0, 2, 0],
                          #        [0, 0, 3]])
```

Indexing and slicing

Indexing and slicing do not make a copy of the array itself: they returns **views**.

As in Python, simple assignments do not make copy either.

Views always have flags.owndata set to False.

```
x = np.arange(10)
x[4:7]                                          # array([4, 5, 6])
x[4:9:2]                                        # array([4, 6, 8])
y = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) # array([[1, 2, 3],
                                                #        [4, 5, 6],
                                                #        [7, 8, 9]])



y[:1, :]                                        # array([[1, 2, 3]])

z = y[:1, 1:-1]                                 # array([[2]])
z.flags.owndata                                 # False
```

Operators

```
x = np.ones((3,)) * 5                # array([5, 5, 5])
y = np.arange(3)                     # array([0, 1, 2])

print(x + y)                         # element-wise addition; prints array([5., 6., 7.])
print(x - y)                         # element-wise subtraction; prints array([5., 4., 3.])
print(x * y)                         # element-wise multiplication; prints array([0., 5., 10.])
print(x / y)                         # element-wise division; prints array([inf, 5., 2.5])
print(x ** y)                        # element-wise power; prints array([1., 5., 25.])
print(x.T @ y)                       # matrix multiplication; prints array([15.])

x = np.array([True, False, False, True])
y = np.array([True, False, True, False])

print(x | y)                         # element-wise OR; prints array([True, False, True, True])
print(x & y)                         # element-wise OR; prints array([True, False, False, False])
print(x ^ y)                         # element-wise XOR; prints array([False, False, True, True])
print(~x)                            # element-wise NOT; prints array([False, True, True, False])
```

Reducing on one axis (or dimension)

Some operations such as np.sum(), np.mean(), np. prod() etc. consider the whole array as default: this means they examine all axes. You can force the computation to be done on one or more chosen axes.

```
x = np.ones((3, 3, 4)) * 5
y = np.arange(36).reshape(3, 3, 4)

print(x.shape)                          # (3, 3, 4)
print(y.shape)                          # (3, 3, 4)
print(x.sum().shape)                    # ()
print(x.sum(axis=1).shape)              # (3, 4)
print(y.mean(axis=2).shape)             # (3, 3)
print(y.mean(axis=(0, 1)).shape)        # (4,)
```

Broadcasting

For the moment, we just saw examples where the two arrays had the same shape. In practice, it is common to find different shapes: what about matrix-vector multiplication, or matrix-scalar subtraction?

Numpy takes care of the missing dimensions, but just if you are clear about your aim: **element-wise operations need arrays with the same shape on all axes, except one. The one axis that differs needs to be of unitary size on one array, and Numpy repeats the values to reach the size of the corresponding axis of the other array.**

Scalars, instead, can perform operations with arrays of any shape.

Broadcasting - examples

```
x = np.ones((3, 3))
y = np.ones((4,))
z = np.ones((3,))
print(x * y)                    # operands could not be broadcast together
print(x * z)                    # z is repeated over the second axis to match the shape of x
print(z[None].shape)            # creates a new unitary axis of z; prints (1, 3)
print(z[np.newaxis].shape)      # creates a new unitary axis of z; prints (1, 3)

x = np.ones((3, 7, 4))
y = np.ones((3, 4,))
print(x * y)                    # operands could not be broadcast together
print(x * y[:, np.newaxis])     # does the job

x = np.ones((3, 7, 4))
y = np.ones((3, 4, 6))
print(x[:, :, :, np.newaxis] * y[:, np.newaxis])          # if we have two axes that differs, we need to
                                                          # broadcast on two new axes
```

Exercises

Enjoy this repo: https://github.com/rougier/numpy-100