Deep Learning

Exercise 1: Perceptron Learning in Python

Instructor: Manuel Günther
Email: guenther@ifi.uzh.ch
Office: AND 2.54

Friday, February 26, 2020

Outline

1. Python
2. Modules
3. Perceptron Learning

Outline

1 Python
- Installation
- Running Python
- Datatypes and Operations
- Control Flow
- Functions
- Classes and Objects
- Exceptions
- String Formatting
- File I/O

Installation

Conda Installation

- Install Conda for your operating system: `http://docs.conda.io/en/latest/miniconda.html`
- Activate (required each time):
  - Open Console (`cmd.exe`, `/bin/bash`)
  - Activate: `Path\to\conda\Scripts\activate.bat` (Windows)
  - Activate: `source Path/to/conda/bin/activate` (Unix/Bash)
- Optional: create and activate local environment
  - Create environment: `conda create -name NAME python=3.8`
  - Activate environment: `conda activate NAME`
- Install packages
  - `conda install numpy scipy scikit-learn matplotlib`

Running Python

Console

- Start Python: `python`
- Code some operations: `4*5`
- Getting help: `help(print)`

Advantage

Interactive, quick

Disadvantage

Bad for complicated code

Script

- Edit `script.py` with text editor
- Results are not printed
  $\rightarrow$ use `print(4*5)` to output
- Run script: `python script.py`

Advantage

Easily repeatable

Disadvantage

Two-step process, non-interactive

Datatypes and Operations

Dynamically Typed Numbers

- Boolean:
  - `k = True`
- (Signed) integer:
  - Simple: `a = -42`
  - Long: `b = 12345678901234567890`
  - Hex: `c = 0x6AfEb5`
- Float (IEEE 754 double):
  - Simple: `pi = 3.14159265`
  - Scientific: `sci = 1.4142e-5`
- Complex (128 bit):
  - Simple: `cplx = -1.5 + 3.2j`

Operations

- Boolean:
  - `not`, `and`, `or`
- Numerical:
  - `+`, `-`, `*`, `/`, `//`, `%`, `**`
  - `4/2 = 2.`, but `4//2 = 2`
    Different in Python 2!
  - Modulo: `24 % 10 = 4`
  - Power: `5 ** 2 = 25`
- Assignment:
  - `=`, `+=`, `-=`, `/=`, `//=`, `%=`, `**=`
- Conversion:
  - `float(b)`, `int(pi)`

Datatypes in Python

Complex Types

- String (immutable):
  - `s1 = "Hello"`
  - `s2 = 'world'`
  - `s3 = """Multiline string with exact interpretation"""`
- List, Tuple (immutable)
  - `x = [-1e-5, 'text', s1]`
  - `y = (s2,)`
- Dictionary
  - `d = {"a" : 0.4, 17 : c}`
- Nonetype `None`

Operations

- Concatenation:
  - String: `"Hello" + ' world!'`
  - List: `x + [y]` and `x += [y]`
  - Tuples and lists: ~~`[1] + (2,)`~~
- Multiplication:
  - String: `"ab"*4`
  - List/Tuple: `[2]*10`, `('o',)*10`
- Indexing:
  - Dictionary: `d['a']`, `d[1.5]='x'`
  - List/Tuple: `y[0]`, `x[2]=None`
  - Range: `x[1:3]`, `x[:2]`, `x[2:]`
  - Negative: `x[-1]`, `x[-2:]`

Datatypes in Python

List Manipulation

- `append(x)` appends `x`
- `extend([x])` appends list
- `insert(i, x)` inserts `x` at `i`
- `remove(x)` removes `x`

  $\rightarrow$ only if exists, first occurrence
- `pop(i)` removes and returns element at index `i`
- `del` removes specific index

  $\rightarrow$ different syntax
- `clear()` removes all elements

Example

```
# create initial list
data = [0, "one", 2., 'three', [4]]

# append values
data.append("5")
data.extend((6, 7.))

# insert values
data.insert(0, -1)
data[0:0] = ["-3", -2.]

# delete values
data.remove("three")
data.pop(1)
data.pop(-2)
del data[3]
```

Datatypes in Python

Type checking via `type`

- `type(1)` → `<class 'int'>`
- `type(1.)` → `<class 'float'>`
- `type(1j)` → `<class 'complex'>`
- `type({})` → `<class 'dict'>`
- `type(())` → `<class 'tuple'>`
- `type((1))` → `<class 'int'>`
- `type((1,))` → `<class 'tuple'>`
- `type([])` → `<class 'list'>`
- `type("")` → `<class 'str'>`

Type testing via `isinstance`

- `isinstance(1, int)` → `True`
- `isinstance(1, float)` → `False`
- `isinstance(1., float)` → `True`
- `isinstance(1., (int, float))` → `True`
- `isinstance('', str)` → `True`
- `isinstance([], str)` → `False`
- `isinstance((1), tuple)` → `False`
- `isinstance((1,), tuple)` → `True`

Control Flow

Indentation

- Blocks have same indentation
- Default: 4 spaces
- Mixing tabs and spaces
  - → Strongly discouraged
  - → Setup editor to insert spaces

`if` Statements

```
if Boolean Condition:
    Code for True Condition
else:
    Optional Code for False Cd.
```

Control Flow

while Loops

```
while Boolean Condition:
    if Skip Condition:
        continue
    Repeated Code
    if Additional Condition:
        break
else:
    Code when exited normally
```

while Loop Example

```
data = [0, [1], "two", 3., '4', -5]
index = 0
while index < len(data):
    if isinstance(data[index], list):
        continue

    number = float(data[index])
    print (number*2)
    index += 1

    if data[index] < 0:
        break

else:
    print ("Success!")
```

Control Flow

while Loops

```
while Boolean Condition:
    if Skip Condition:
        continue
    Repeated Code
    if Additional Condition:
        break
else:
    Code when exited normally
```

Coding

Find three issues with this code.

while Loop Example

```
data = [0, [1], "two", 3., '4', -5]
index = 0
while index < len(data):
    if isinstance(data[index], list):
        continue

    number = float(data[index])
    print (number*2)
    index += 1

    if data[index] < 0:
        break

else:
    print ("Success!")
```

Control Flow

while Loops

```
while Boolean Condition:
    if Skip Condition:
        continue
    Repeated Code
    if Additional Condition:
        break
else:
    Code when exited normally
```

Coding

Find three issues with this code.

infinite loop, `float([1]])`, `"two" < 0`

while Loop Example

```
data = [0, [1], "two", 3., '4', -5]
index = 0
while index < len(data):
    if isinstance(data[index], list):
        continue

    number = float(data[index])
    print (number*2)
    index += 1

    if data[index] < 0:
        break

else:
    print ("Success!")
```

Control Flow

for Loops

```
for item in iterable:
    if Skip Condition:
        continue
    Code using item
    if Additional Condition:
        break
else:
    Code when exited normally
```

Iterables and Examples

- Lists/Tuples
  - → Elements: `for e in [1,4,7,10]`
- Ranges:
  - → `for i in range(0,20,2)`
  - ⇒ start, end (exclusive), stepwidth
- Enumerations:
  - → `for i, e in enumerate([1,4,7,10])`
  - ⇒ Tuple Unboxing
- Dictionaries
  - → Keys: `for k in {1:2, 3:4}`
  - → Both: `for k, v in {1:2, 3:4}.items()`

Control Flow

List Comprehensions

- Turn one list into another
- Filter elements (optional)
- Nested List Comprehensions

Definition

```
[f(x) for x in src if cond(x)]
```

Examples

```
squares = [x**2 for x in range(10)]
odds = [x for x in range(10) if x % 2]

triang = [[x for x in range(10) if x >= y]
              for y in range(10)]
```

Dictionary Comprehensions

- Create dictionaries
- Filter elements (optional)
- Nested comprehensions

Definition

```
{key : val for ... if ...}
```

Examples

```
squares = {x : x**2 for x in range(10)}
half = {x : x//2 for x in range(10)
                  if x % 2}

inv = {v : k for k, v in {...}.items()}
```

Functions

Function Definition

```
def func_name(arguments, keywords):
    Code
    return ...
```

Arguments

- Required to be specified
- No datatype is required

Keywords

- Arguments with default values
  $\rightarrow$ Syntax: `name=value`
- Warning: default values [] or {}

Function Call Examples

```
def func1(data, opt=None):
    if opt is None:
        print("data is", data)
    else:
        return data

func1(27)
  -> prints "data is 27"
  -> returns None

func1(27, True)
  -> returns 27

func1(opt = False, data = 27)
  -> returns 27
```

Functions

Generator Function

- To be used, e.g., with `for` loop
- Returns samples one by one

Function Definition

```
def func_name(arguments, keywords):
    while ...:
        yield ...
```

Predefined Generators

- `range`, `enumerate`
- `dict.keys()`, `dict.items()`

Generator Example

```
def sup(it1, it2):
    for v1 in it1:
        for v2 in it2:
            yield v1, v2

for t in sup("ab", range(2)):
    print(t)

('a', 0)
('a', 1)
('b', 0)
('b', 1)
```

Classes and Objects

Function Definition

```
class ClassName(parents):
    member = None

    def __init__(self, params):
        super(ClassName, self).__init__()
        Initialization of self

    def method(self, params):
        Can use self and params
        return ...

    @staticmethod
    def static(params):
        Cannot use self
        return ...
```

Members

- Prior definition optional
- Access through `self.member`
- Publicly accessible

Constructor

- Base class constructor
- Initializes object (`self`)

Functions

- First parameter: `self`
- Special function `__call__`

Classes and Objects

Example

```
class AddN(object):
    def __init__(self, n=1):
        super(AddN, self).__init__()
        self.n = n

    def __call__(self, x):
        return x + self.n

    def squared(self, x):
        return x**2 + self.n

    def sequence(self, x):
        for n in range(self.n):
            yield x + n
```

Example (Contd.)

```
add1 = AddN(n=1)
print(add1.squared(5))
print(add1(5))

add1.n = 4
add1.unused = None
print(add1.squared(5))

for n in add1.sequence(10):
    print(n)
```

Classes and Objects

Objects in Python

In Python (almost) everything is an object

Immutable Objects

```
def modify(x):
    x += 1

y = 10
modify(y)
print(y) -> 10
```

Mutable Objects

```
def append(x):
    x.append(1)

y = [10]
append(y)
print(y) -> [10,1]
```

Function Objects

```
def sqr(x):
    return x**2

def apply(f, data):
    return f(data)

print(apply(sqr, 5))
print(apply(add1, 5))
```

Exceptions

Exception Handling

```
try:
    Code

except Exception1 as ex:
    print(ex)
    raise Exception() from ex

except (Exception2, Exception3):
    Handle exception
    raise

finally:
    Always executed
```

Types of Exceptions

- Base class: `Exception`
- `SyntaxError`
- `ArithmeticError`
- `ValueError`
- `IndexError`, `KeyError`
- `MemoryError`
- `StopIteration`, `SystemExit`
- Own Exceptions
  $\rightarrow$ derive from `Exception`

String Formatting

Old Types of String Formatting

- The `%` syntax:
  → `"name: %s; age: %d" % ("John", 42)`
- The `format` syntax:
  → `"name: {}; age: {}".format("John", 42)`
  → `"name: {name}; age: {age}".format(age=42, name="John")`

Python `f`-Strings

- Special syntax `f""` or `F""`
- Evaluates expressions in `{}`
- Number formatting `f"{n:w.p}"`
  → number, width, precision

Example

```
data = {"name" : "John", "age" : 42}
print(f"name: {data['name']}; age: {data['age']}")

x = 3.14159265
print(f"pi = {x: 3.4}")
```

File I/O

The `open` function

- Signature: `open(name, mode, ...)`
- Filenames `"C:/path/to/file.txt"`
- Opening modes:
  - `'r'`ead, `'w'`rite, `'a'`ppend
  - `'t'`ext: `str`, `'b'`inary: `bytes`

The `with` Statement

- Ensures closure of object
- Syntax for files:

  `with open(...) as file:`

Example

```
# write file
with open("test.txt", 'wt') as file:
    for x in range(1,11):
        file.write(f"{x},{x**2}\n")

# read file
with open("test.txt", 'rt') as file:
    for line in file:
        x, y = line.rstrip().split(',')
        print(x, y)
```

Outline

2 Modules
- Builtin Modules
- Vector Math with `numpy` and `scipy`
- Scientific Plotting with `matplotlib`

Modules

Modules in Python

- Library containing related functionality
- Can contain submodules, classes, functions
- Need to be `import`ed before usage

Preferred

```
import os
os.listdir(".")
```

Often Seen

```
from os import listdir
listdir(".")

import numpy as np
np.ndarray(5)
```

Highly Discouraged

```
from os import *
listdir(".")

from os import listdir as ls
ls(".")
```

Builtin Modules

Operating System `os`

- Directory handling
  → `getcwd`, `chdir`, `listdir`, `mkdir`, `rmdir`, `walk`
- File handline
  → `link`, `rename`, `remove`, `chown`, `chgrp`
- Filename handline `os.path`
  → `join`, `split`, `splitext`, `abspath`, `exists`, `isdir`
- Process handling
  → `fork`, `spawn`, `exec`, `nice`, `wait`, `kill`

Python System `sys`

- Command line arguments `argv`
- Search path `path`
- Console In/Output
  → `stdin`, `stdout`, `stderr`

Mathematics `math`

- Constants: `pi`, `e`, `inf`, `nan`
- Rounding: `fabs`, `ceil`, `floor`
- Exponential: `exp`, `log`, `pow`, `sqrt`
- Trigonometrical: `sin`, `sinh`, `asin`

Builtin Modules

Random Numbers `random`

- Single value:
  → `random`, `uniform`, `gauss`, `randint`, `choice`
- Sequences
  → `shuffle`, `sample`
- Reproducibility: `seed`

Serealization `pickle`

- Writing: `dump`, `dumps`
- Loading: `load`, `loads`

Other Modules

- Temporary files: `tempfile`
- Compression: `tarfile`, `zipfile`
- Data Formats: `csv`, `json`, `html`
- Database Interface: `sqlite3`
- Argument parser: `argparse`, `getopt`
- Console/file logging: `logging`
- Parallelization: `multiprocessing`, `subprocess`, `threading`
- Communication: `urllib`, `http`, `ftplib`

Vector Math with `numpy` and `scipy`

Multi-dimensional Arrays `ndarray`

- Creating in `dims=(3,6,4)`:
  - → empty: `ndarray(dims, dtype)`
  - → zeros: `zeros(dims, dtype)`
  - → ones: `ones(dims, dtype)`
  - → from data: `array(data, dtype)`
  - → random: `random.random(dims)`
- Indexing:
  - → Single Element: `x[0,1,-2]`
  - → Slicing: `x[:2,:,2:]`
  - → Step: `x[1:3:2, 1::3, ::-1]`

Array Attributes

- Dimensionality: `shape`
- Elements: `size`
- Data type: `dtype`

Array Methods

- Fill with value: `fill`
- Deep copy: `copy`
- Change shape: `reshape`
- Change to 1D: `flatten`
- Transpose: `transpose`

Vector Math with `numpy` and `scipy`

Element-wise Operations on Arrays

- Arithmetic: `+`, `-` , `*`, `/`, `//`, `%`
- Assign: `+=`, `-=` , `*=`, `/=`, `//=`, `%=`
- Comparison: `<`, `>`, `<=`, `==`
  - ⇒ Boolean arrays

Broadcasting

- Repeating elements to higher dimensions
  - → Single value: `x += 1.`, `x > 0`
  - → Last dimension: `x * (1,4,3,2)`
- Dimensions must be equal or 1

Eamples

```
x = numpy.array([[1,2,3,4],
                 [5,6,7,8]])
# boolean arrays
x < 4
x == 4

# simple broadcasting
y = x * -1.
x *= -1.

# complex broadcasting
z = x * [[2],
         [-1]]
```

Vector Math with numpy and scipy

Arrays Reductions

- Reduce specific dimension (axis)
- min, max, argmin, argmax
- sum, cumsum, mean, std
- For Boolean arrays: any, all

Array Modifications

- Sorting: sort, argsort
- Inserting: insert, append
- Concatenate: vstack, hstack
- Dimensions: squeeze, expand_dims

Eamples

```
x = numpy.array([[1,2,3,4],
                 [5,6,7,8]])
# minimum
x.min()
numpy.min(x, axis=1)

# sum and std
x.sum(axis=0)
numpy.std(x)

# boolean
(x > 8).any()

# sorting
numpy.argsort(x, axis=0)
```

Vector Math with numpy and scipy

Basic Linear Algebra

- Matrix multiplication `numpy.dot`
- Vector multiplication `numpy.inner`, `numpy.outer`
- Norms `scipy.linalg.norm`
- Inverse `scipy.linalg.inv`
- Distances in `scipy.spatial.distance`
  $\rightarrow$ `euclidean`, `cosine`, `canberra`, `cdist`

Example

```
import numpy, scipy.spatial
# create two arrays
x = numpy.array([1.,2.,3.])
y = numpy.random.normal(size=3)

# compute vector prodcts
numpy.inner(x,y)
numpy.dot(x,y)
numpy.outer(x,y)

# normalize vector
x /= scipy.linalg.norm(x)

# compute cosine distance
scipy.spatial.distance.cosine(x,y)
```

Scientific Plotting with matplotlib

Plotting Functions

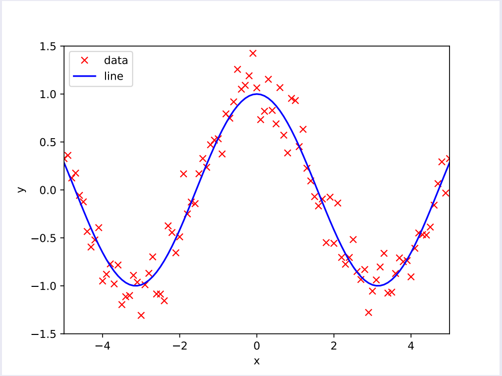
- `plot([x],y,[fmt],[label])`
  - → `x` and `y`: coordinates to plot
  - → `fmt`: color: `"rgbkmcy"` and style `".+xosd-:"`
  - → `label`: string for legend
- `legend([labels],[loc])`
  - → `loc = "upper left", "center", "best"`
- Limit axes `xlim`, `ylim`
- Axis labels `xlabel`, `ylabel`
- Save to file: `savefig`

Example

```
import numpy
from matplotlib import pyplot

# create data
x = numpy.arange(-5,5.001,0.1)
target = numpy.cos(x)
data = target + numpy.random.normal(0, .2, x.shape)

# plot data points and line
pyplot.plot(x, data, "rx", label="data")
pyplot.plot(x, target, "b-", label="line")

# limit axes and provide labels
pyplot.xlim((-5,5)); pyplot.ylim((-1.5,1.5))
pyplot.xlabel("x"); pyplot.ylabel("y")

# create legend and write to file
pyplot.legend(loc="upper left")
pyplot.savefig("Example.pdf")
```

Scientific Plotting with `matplotlib`

Example Plot



Example

```
import numpy
from matplotlib import pyplot

# create data
x = numpy.arange(-5,5.001,0.1)
target = numpy.cos(x)
data = target + numpy.random.normal(0, .2, x.shape)

# plot data points and line
pyplot.plot(x, data, "rx", label="data")
pyplot.plot(x, target, "b-", label="line")

# limit axes and provide labels
pyplot.xlim((-5,5)); pyplot.ylim((-1.5,1.5))
pyplot.xlabel("x"); pyplot.ylabel("y")

# create legend and write to file
pyplot.legend(loc="upper left")
pyplot.savefig("Example.pdf")
```

Scientific Plotting with `matplotlib`

Surface Plots

- Create `fig = figure([figsize])`
- Create subplots

  `fig.add_subplot(RCI, [projection])`

  $\rightarrow$ `R`ows, `C`olumns, `I`ndex (one-based)
- Single 3D subplot

  `ax = fig.add_subplot(111, projection="3d")`
- Create 3D data:

  `xx, yy = numpy.meshgrid(x,y), zz`
- Surface plot:

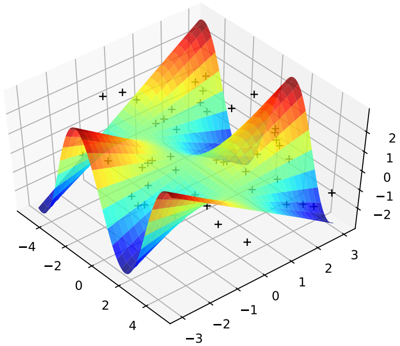
  `ax.plot_surface(xx, yy, zz)`

Example

```
# create data
x = numpy.arange(-5., 5.001, 0.1)
y = numpy.arange(-3., 3.001, 0.1)
xx, yy = numpy.meshgrid(x,y)
zz = numpy.sin(xx) * yy

# surface plot
fig = pyplot.figure()
ax = fig.add_subplot(111, projection='3d',
                     azim=-40, elev=50)
ax.plot_surface(xx, yy, zz, cmap="jet", alpha=.8)

# single point plot in 3D
pts = numpy.random.random((50,3))
pts = (pts - 0.5) * (10,6,4)
ax.plot(pts[:,0], pts[:,1], pts[:,2], "kx")

# write to file
pyplot.savefig("Surface.pdf")
```

Scientific Plotting with matplotlib

Example Surface Plot



Example

```
# create data
x = numpy.arange(-5., 5.001, 0.1)
y = numpy.arange(-3., 3.001, 0.1)
xx, yy = numpy.meshgrid(x,y)
zz = numpy.sin(xx) * yy

# surface plot
fig = pyplot.figure()
ax = fig.add_subplot(111, projection='3d',
                     azim=-40, elev=50)
ax.plot_surface(xx, yy, zz, cmap="jet", alpha=.8)

# single point plot in 3D
pts = numpy.random.random((50,3))
pts = (pts - 0.5) * (10,6,4)
ax.plot(pts[:,0], pts[:,1], pts[:,2], "kx")

# write to file
pyplot.savefig("Surface.pdf")
```

Outline

3 Perceptron Learning

Perceptron Learning

Task

1. Generate two separable normal distributed datasets in 2D
2. Label one dataset with `-1` and one with `1`
3. Shuffle both datasets together
4. Implement the perceptron with `numpy` matrices
5. Initialize perceptron weights randomly
6. Define an appropriate stopping criterion
7. Apply perceptron learning rule to the data
8. Plot the samples and the learned decision boundary together

Perceptron Learning

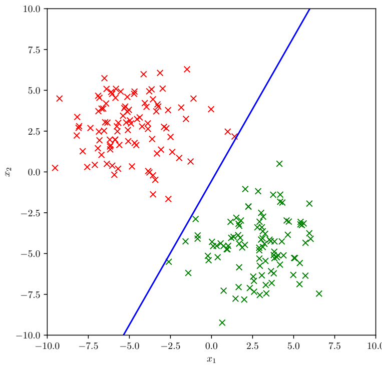
Perceptron Learning

1. Randomly initialize weights $\vec{w}$
2. Choose a training sample $(\vec{x}, t)$
3. Predict its class $y = \vec{w}^{\mathrm{T}}\vec{x}$
4. If wrongly classified $(y \cdot t < 0)$
   $\rightarrow$ Update weights: $\vec{w} = \vec{w} + t \cdot \vec{x}$

Decision Boundary

- Perceptron: $y = w_0 + w_1 x_1 + w_2 x_2$
- Boundary: $y = 0$
- Solve to: $x_2 = f(x_1)$

Possible Outcome



10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
x_2
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x_1