lecture3-RISE

October 28, 2019

0.1 Python Programming for Machine Learning

Lecture 3

- Random distributions and sampling
  - Uniform, Gaussian, Multinomial
- Automatic gradient module
- Optimization
  - Parallel computing on GPUs
  - Cython

Import the required packages

[1]:
```
import numpy as np
import numpy.random as rnd
rnd.seed(42)
import matplotlib.pyplot as plt
%matplotlib inline
```

1 Sampling

[2]:
```
rnd.uniform?
```

Draw random samples from a uniform distribution.

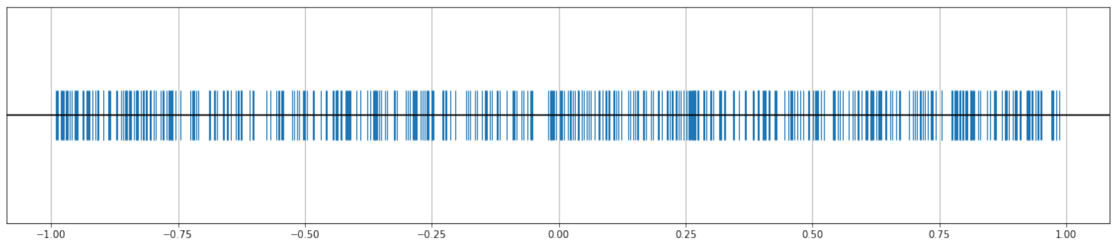
[3]:
```
U = rnd.uniform(-1,1, size = 500)
print(U.shape)
f"Mean: {U.mean():.3f} Variance: {U.var():.2f}"
```

```
(500,)
```

[3]:
```
'Mean: -0.003 Variance: 0.36'
```

Plot of 1-dimensional samples

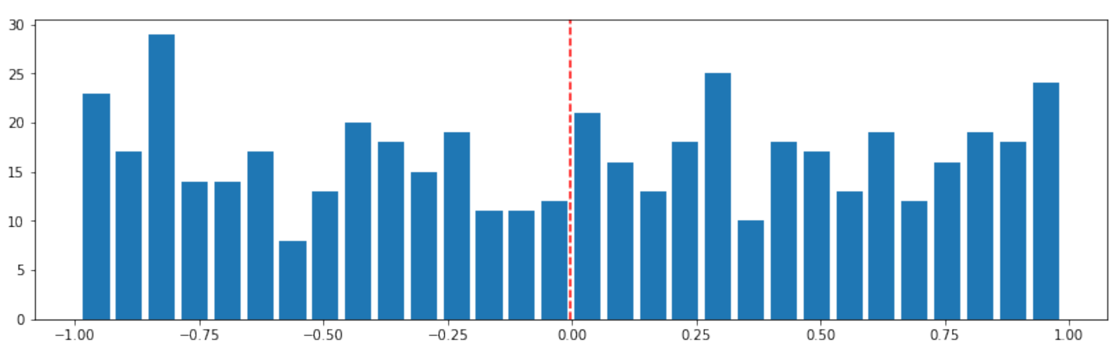
[4]:
```
plt.figure(figsize=(20,4))
plt.plot(U,np.zeros_like(U), '|', ms=50)
plt.axhline(y = 0, color='k')
plt.grid(axis='x')
_=plt.yticks([])
```



1.0.1 Represent data using histogram plots

[5]:
```
plt.figure(figsize=(14,4))
nums, ranges, _ = plt.hist(U, bins=30, rwidth=0.8)

# Equivalent:
nums, ranges = np.histogram(U, bins=30)

_=plt.axvline(x=U.mean(), ls='--',  c='r') # plot dashed mean line

print(nums)
print(ranges)
```

```
[23 17 29 14 14 17  8 13 20 18 15 19 11 11 12 21 16 13 18 25 10 18 17 13
 19 12 16 19 18 24]
[-0.98987683 -0.92401662 -0.8581564  -0.79229619 -0.72643598 -0.66057576
 -0.59471555 -0.52885533 -0.46299512 -0.3971349  -0.33127469 -0.26541448
 -0.19955426 -0.13369405 -0.06783383 -0.00197362  0.06388659  0.12974681
  0.19560702  0.26146724  0.32732745  0.39318766  0.45904788  0.52490809
  0.59076831  0.65662852  0.72248874  0.78834895  0.85420916  0.92006938
  0.98592959]
```



1.0.2 Univariate-normal (Gaussian) distribution.

$$\boxed{\mathcal{N}(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\Big(-\frac{(x-\mu)^2}{2\sigma}\Big)}$$

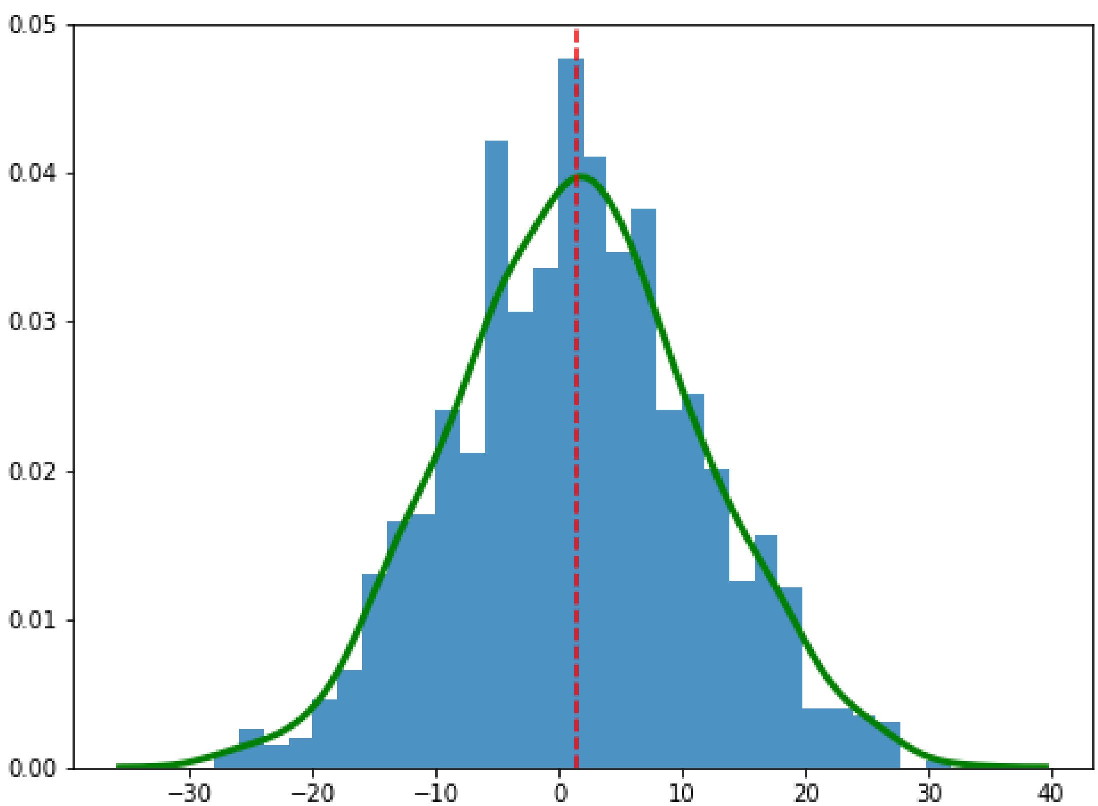
[6]:
```
X = rnd.normal(loc=1, scale=10, size=1000)
print(X.shape)
f'Mean:{X.mean():.3f} Variance: {X.var():.2f}'
```

```
(1000,)
```

[6]:
```
'Mean:1.402 Variance: 100.16'
```

1.0.3 Histogram plot

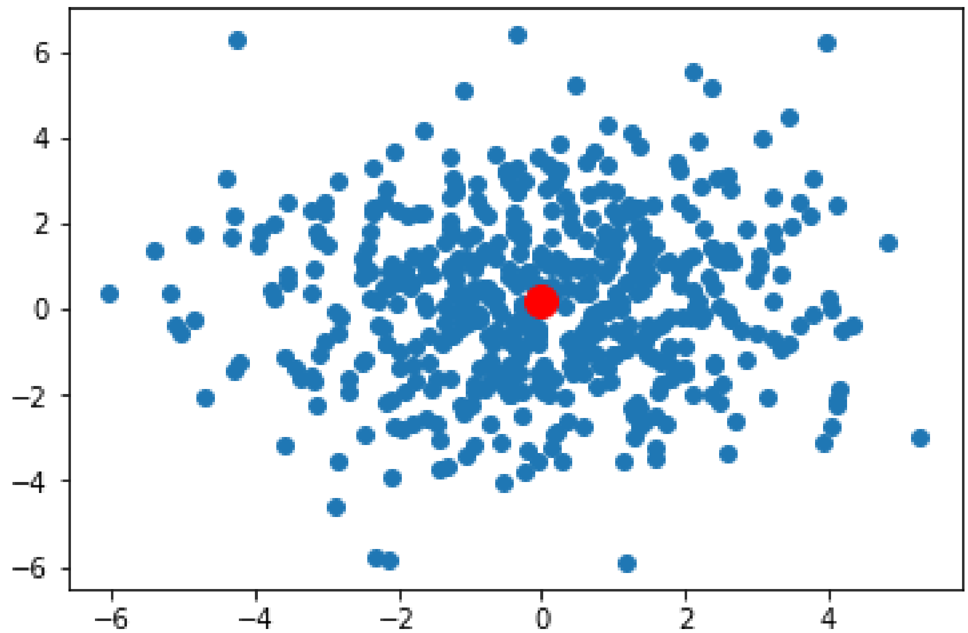
[7]:
```
import seaborn as sns
plt.figure(figsize=(8,6))
plt.hist(X, bins=30, density=True, alpha=0.8)
sns.kdeplot(X,color='g',linewidth=3)
_=plt.axvline(X.mean(), ls='--', c='r')
```



1.0.4 Multivariate-normal (Gaussian) distribution.

Sample from the distribution

[8]:
```
k=2; mu=np.zeros(k)
Sigma=4*np.eye(k)
X = rnd.multivariate_normal(mu,Sigma,size=500) # X -> (500,2)
plt.scatter(*X.T)
_=plt.plot(*X.mean(0),'o',c='r',ms=12)
```

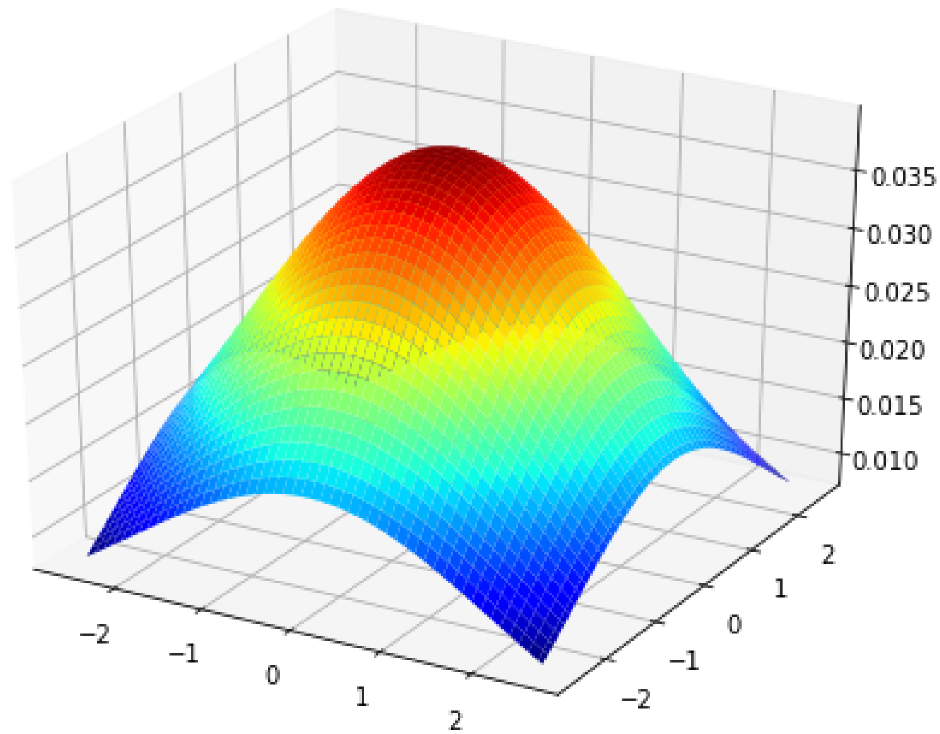


1.0.5 Probability density function

$ $\boxed{\mathcal{N}(x|\mu,\Sigma) = \frac{1}{\sqrt{(2\pi)^k|\Sigma|}}\exp\Big(-\frac{1}{2}(x-\mu)^\top\Sigma^{-1}(x-\mu)\Big), \quad \mu\in\mathbb{R}^{(k)}, \quad \Sigma\in\mathbb{R}^{(k,k)}}$ $

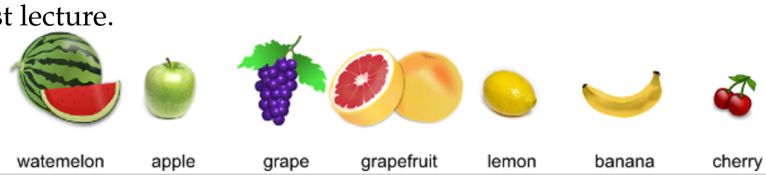
[9]:
```
from matplotlib import cm; from mpl_toolkits.mplot3d import Axes3D
from scipy.stats import multivariate_normal as gaussian
val = 0.4*np.abs(X.max())
x = np.linspace(-val, val, 100)
y = np.linspace(-val,val,100)
x, y = np.meshgrid(x, y)
pos = np.dstack((x, y))
z = gaussian(mu, Sigma).pdf(pos)
fig = plt.figure(figsize=(8,6))
```

```
ax = fig.gca(projection='3d')
_=ax.plot_surface(x, y, z, cmap=cm.jet)
```



1.1 Multinomial distribution

Let's draw some random samples from a multinomial distribution. We'll use our fruits from the first lecture.



[10]:
```
fruits = np.array([
    'watermelon',
    'apple',
    'grape',
    'grapefruit',
    'lemon',
    'banana',
    'cherry'
])
```

1.1.1 Sample from multinomial

[11]:
```
n = 5 # number of samples
p = np.ones(len(fruits))/len(fruits)

repeat = np.tile(fruits, (n,1)) # repeat multiple (5) times
print(repeat)

mlt = rnd.multinomial(1, p, size=(n)) # draw multinomial samples 5 times with an␣
 ↪equal probability
print(mlt)

samples = repeat[mlt.astype(bool)] # show drown samples
print(samples)
```

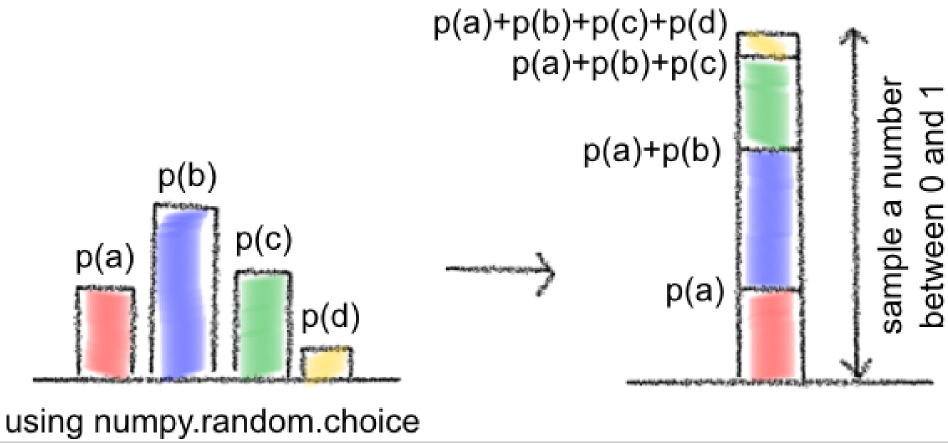
```
[['watermelon' 'apple' 'grape' 'grapefruit' 'lemon' 'banana' 'cherry']
 ['watermelon' 'apple' 'grape' 'grapefruit' 'lemon' 'banana' 'cherry']
 ['watermelon' 'apple' 'grape' 'grapefruit' 'lemon' 'banana' 'cherry']
 ['watermelon' 'apple' 'grape' 'grapefruit' 'lemon' 'banana' 'cherry']
 ['watermelon' 'apple' 'grape' 'grapefruit' 'lemon' 'banana' 'cherry']]
[[0 1 0 0 0 0 0]
 [0 0 0 0 0 0 1]
 [0 0 0 0 0 0 1]
 [0 1 0 0 0 0 0]
 [0 0 0 1 0 0 0]]
['apple' 'cherry' 'cherry' 'apple' 'grapefruit']
```

1.1.2 Adjust selection probabilities

[12]:
```
p = [0.05, 0.70, 0.05, 0.05, 0.05, 0.05, 0.05] # new probailities

mlt = rnd.multinomial(1, p, size=(5)) # draw multinomial samples 5 times with␣
 ↪given probabilities
print(mlt)

samples = repeat[mlt.astype(bool)] # show drown samples
print(samples)
```

```
[[0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 0 0 0 1 0 0]
 [0 1 0 0 0 0 0]]
['apple' 'apple' 'apple' 'lemon' 'apple']
```

1.1.3 Another way to make discrete choices

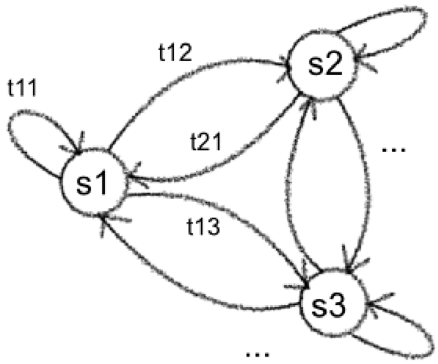


[13]:
```
p = [0.05, 0.70, 0.05, 0.05, 0.05, 0.05, 0.05]

# Cumulate them
l = np.cumsum([0] + p[:-1]) # lower-bounds
h = np.cumsum(p)            # upper-bounds

print(l)
print(h)

# Draw a number between 0 and 1
u = np.random.uniform(0, 1)

# Find which basket it belongs to
s = np.logical_and(u > l, u < h)
print(s)

# retrieve the label
fruits[np.argmax(s)]
```

```
[0.   0.05 0.75 0.8  0.85 0.9  0.95]
[0.05 0.75 0.8  0.85 0.9  0.95 1.  ]
[False  True False False False False False]
```

[13]: 'apple'

1.2 Markov Chain



A Markov chain transits between a set of states, where the transition between pairs of states is associated with a fixed probability. The set of probabilities can be stored in a transition matrix.

[14]:
```
# Transition matrix
T = np.array([
    [0.9,0.1,0.0], # transiting from state 1 to state 1,2,3
    [0.0,0.9,0.1], # transiting from state 2 to state 1,2,3
    [1.0,0.0,0.0], # transiting from state 3 to state 1,2,3
])
```

1.3 Markov step function

[15]:
```
# Add empty state to transition matrix
pad_shape = ((0, 0), (1, 0))  # ((before_1, after_1), (before_2, after_2))
P = np.pad(T, pad_shape, mode='constant')
print(P)
```

```
[[0.  0.9 0.1 0. ]
 [0.  0.  0.9 0.1]
 [0.  1.  0.  0. ]]
```

[16]:
```
def mcstep(X, P):
    Xp = np.dot(X, P)
    Xc = np.cumsum(Xp, axis=1)
    L,H = Xc[:, :-1], Xc[:, 1:]
    R = np.random.uniform(0, 1, (len(Xp), 1))
    states = np.logical_and((R > L), (R < H))
    #print(states.astype('int32'))
    return states.astype('int32')
```

[17]:
```
A = np.tile([1.0,0,0], (5,1))
# or
A = np.outer(np.ones([5]),[1.0,0,0]) # (5,1) x (1,3) -> (5,3)
num_steps = 10
for i in range(num_steps):
```

```
    A = mcstep(A, P)
A.mean(axis=0)
```

[17]:
```
array([0.6, 0.4, 0. ])
```

2 Autograd

(https://github.com/HIPS/autograd)

2.0.1 Univariate function example

$y = 3x^2 + 2, \quad y'_x = 6x$

[18]:
```
import autograd.numpy as ag_np
from autograd import grad
x = 10*ag_np.ones(1) # variable declaration
y = lambda x: 3 * x**2 + 2
print(grad(y)(x)) # evaluated at point x = 10
```

```
[60.]
```

2.0.2 Multivariate function example

$y = 3x_1^3 + 2^{x_2}, \quad \frac{y}{\partial x_1} = 9x_1^2 \quad \frac{y}{\partial x_2} = 2^{x_2} \ln 2$

[19]:
```
x1 = 2*ag_np.ones(1)
x2 = 3*ag_np.ones(1)

y = lambda x1,x2 : 3*x1**3 + 2**x2

print(grad(y,0)(x1,x2)) # 9 * 2
print(grad(y,1)(x1,x2)) # 2**(3)*ln(2)
```

```
[36.]
[5.54517744]
```

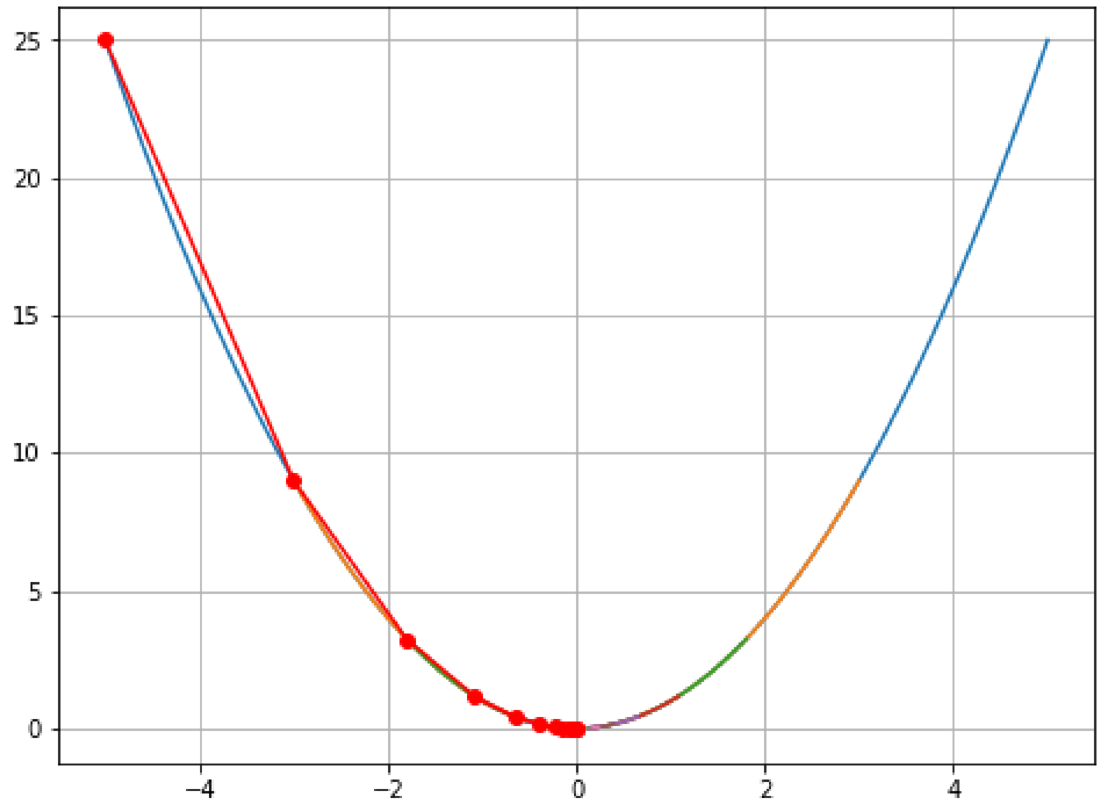
2.0.3 Gradient descent for finding minimum of a function

$y = x^2$

[20]:
```
y = lambda x: x**2
step_size = 0.2
xs = np.array([-5.0]) # starting point
while abs(grad(y)(xs[-1])) > 1e-2:
    curr_val = xs[-1]
    next_val = curr_val - step_size*grad(y)(curr_val)
    xs = np.append(xs, next_val)
```

[21]:
```
x = np.linspace(xs, np.abs(xs), 100)
plt.figure(figsize=(8,6))
```

```
plt.plot(x,y(x))
plt.grid()
_ = plt.plot(xs, y(xs), "-o", c="r")
```



2.1 GPUs

(https://github.com/cupy/cupy)

[22]:
```
import numpy as np
from time import time
n = 1000
X = np.random.normal(size = (n,n))
Y = np.random.normal(size = (n,n))
st = time()
X.dot(Y)
np_t = time() - st
```

[24]:
```
import cupy as cp
X = cp.array(X)
Y = cp.array(Y)
st = time()
cp.dot(X,Y)
```

```
cp_t = time() - st
ratio = np_t/cp_t
f'GPU {ratio:.2f} times faster'
```

[24]: 'GPU 219.32 times faster'

2.2 Cython

- Create new file *hello.pyx*. See the example file (hello.pyx) in the same Folder. The file contains a custom implementation of the matrix product with loops

Cython hello.pyx file cimport cython import numpy as np cimport openmp from cython.parallel cimport prange

@cython.boundscheck(False) @cython.wraparound(False)

cpdef dot(float[:,:] X, float[:,:] Y): cdef: int n,i,j,k float[:,:] Z n = X.shape[0] Z = np.zeros((n,n), dtype = 'float') n = len(X) for i in prange(n, nogil = True): for j in range(n): for k in range(n): Z[i,j] += X[i, k] * Y[k, j] return Z

- Create *setup.py* with compiler commands in order to build a new python package. See example file in the same folder

setup.py file from distutils.core import setup from Cython.Build import cythonize from distutils.extension import Extension from Cython.Distutils import build_ext

setup(name = "hello", cmdclass = {"build_ext": build_ext}, ext_modules = [Extension("hello", ["hello.pyx"], extra_compile_args = ["-O0", "-fopenmp"], extra_link_args=['-fopenmp'])])

Compile the *hello.pyx* file with the following command from terminal. * *python setup.py build_ext –inplace*

After your module is compiled you can import it into your notebook as usual

You can profile your cython code in order to know, either your computations are made efficiently. For the reason you may create an HTML file highlighting the line with the bad performance.

Create file profiling snapshot by calling the following command in your terminal * *cython -a hello.pyx*

Then open a new generated *hello.html* file in a browser. The lines colored yellow still need some python interactions and therefore slow, so you can still find a way to optimize them. But for now it's enought to have no yellow lines within the loops.

```
Generated by Cython 0.29.7

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: hello.c

+01: cimport cython
+02: import numpy as np
 03: cimport openmp
 04: from cython.parallel cimport prange
 05:
 06: @cython.boundscheck(False)
 07: @cython.wraparound(False)
 08:
+09: cpdef dot(float[:,:] X, float[:,:] Y):
 10:     cdef:
 11:         int n,i,j,k
 12:         float[:,:] Z
+13:     n = X.shape[0]
+14:     Z = np.zeros((n,n), dtype = 'float')
+15:     n = len(X)
+16:     for i in prange(n, nogil = True):
+17:         for j in range(n):
+18:             for k in range(n):
+19:                 Z[i,j] += X[i, k] * Y[k, j]
+20:     return Z
```

Import your brand new module as usual

[25]:
```
import hello
```

Test the performance

[26]:
```
from time import time
X = np.array(X.tolist()).astype('float32')
Y = np.array(X.tolist()).astype('float32')
st= time()
hello.dot(X,Y)
time() - st
```

[26]:
```
3.4155197143554688
```

As you already know normal python loops will take much more time to finish all the computations...

[30]:
```
st = time()
n=200
Z = np.zeros((n,n))
for i in range(n):
    for j in range(n):
        for k in range(n):
            Z[i,j] += X[i, k] * Y[k, j]
print(time() - st)
```

```
7.1818482875823975
```