lecture4

November 6, 2018

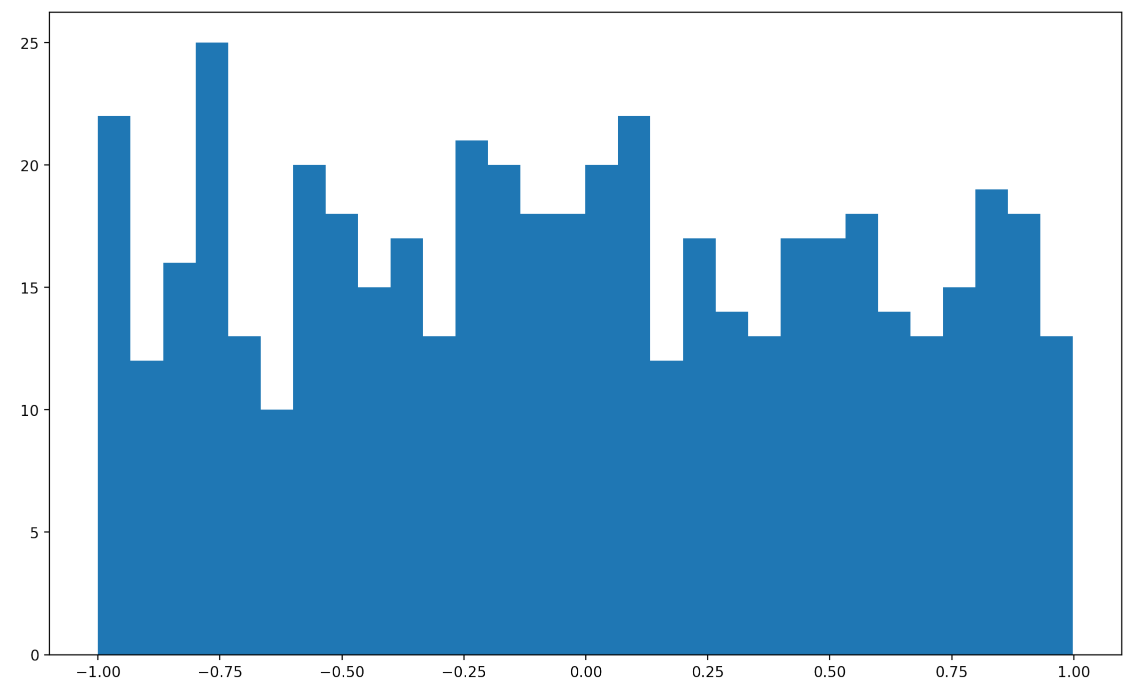
1 Lecture 4

- Sampling
- Automatic gradient
- Parallel computing on GPUs
- Cython

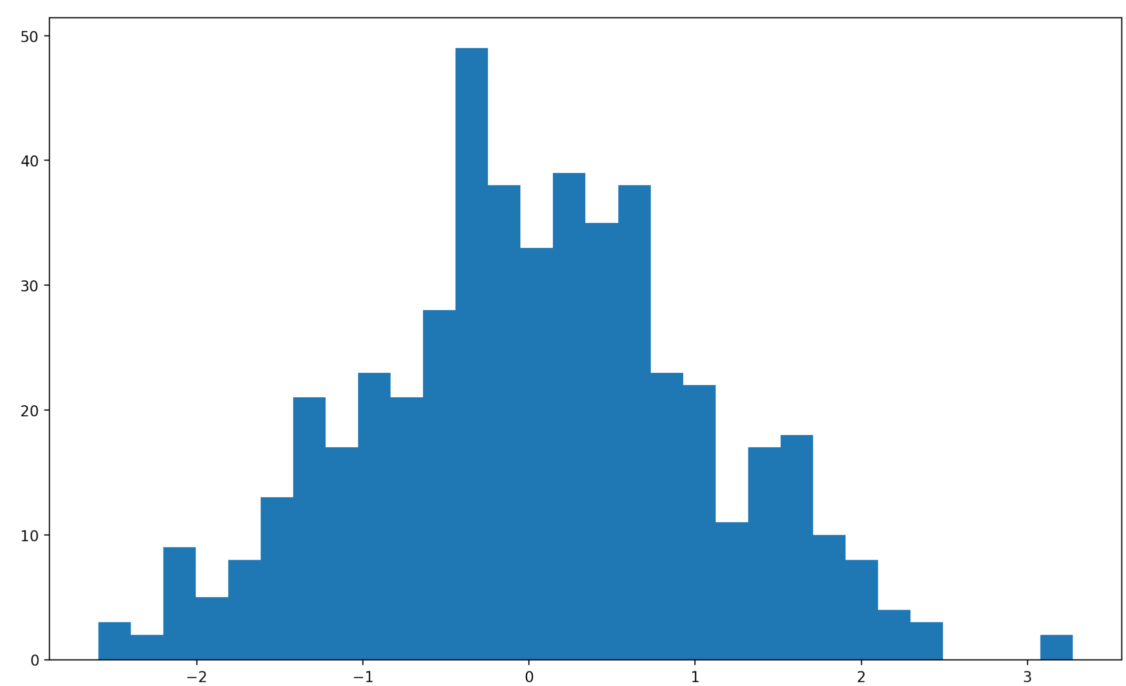
```
In [1]: import numpy as np
        import numpy.random as rng
        import matplotlib.pyplot as plt
```

1.1 Sampling

```
In [2]: rng.uniform?
        rng.normal
        rng.multinomial

Out[2]: <function RandomState.multinomial>

In [3]: plt.figure(figsize=(13,8), dpi=200)
        _ = plt.hist(rng.uniform(-1,1, 500), bins=30)
```

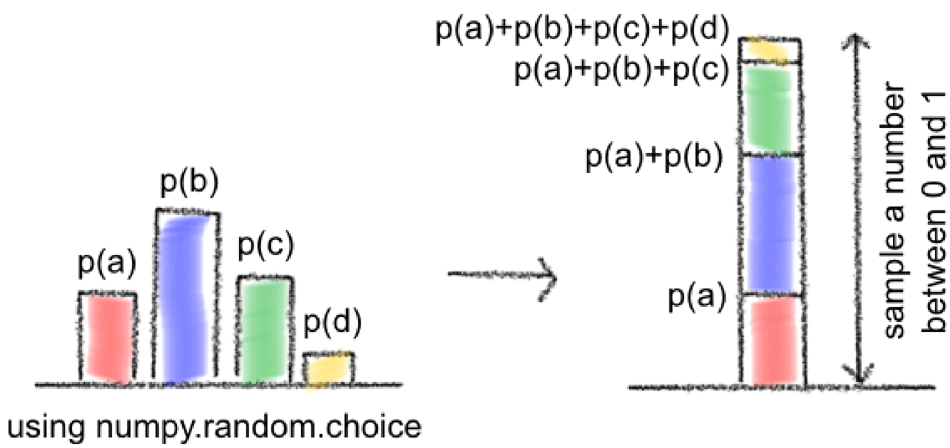



```
In [4]: plt.figure(figsize=(13,8), dpi=200)
        _ = plt.hist(rng.normal(0,1, 500), bins=30)
```



```
In [5]: fruits = np.array([
            'watermelon',
            'apple',
            'grape',
            'grapefruit',
            'lemon',
            'banana',
            'cherry'
        ])

In [6]: n = 5
        p = [1/len(fruits)]*len(fruits)
        np.tile(fruits, (n,1))[rng.multinomial(
                1,
                p,
                size=(5)
            ).astype(bool)
        ]

Out[6]: array(['banana', 'banana', 'grape', 'watermelon', 'apple'], dtype='<U10')

In [7]: p = [0.05, 0.70, 0.05, 0.05, 0.05, 0.05, 0.05]

        n = 5
        np.tile(fruits, (n,1))[rng.multinomial(
                1,
                p,
                size=(5)
            ).astype(bool)
        ]

Out[7]: array(['apple', 'apple', 'apple', 'apple', 'apple'], dtype='<U10')
```

1.1.1 Another way to make discrete choices

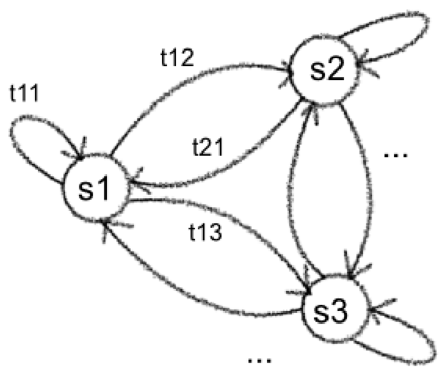


In [8]:
```
p = [0.05, 0.70, 0.05, 0.05, 0.05, 0.05, 0.05]

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

# 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)]
```

```
[False False  True False False False False]
```

Out[8]:
```
'grape'
```

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.

In [9]:
```
# 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
])
```

In [10]:
```
# 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. ]]
```

```
In [11]: 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))
             return np.logical_and((R > L), (R < H))
```

```
In [12]: A = np.outer(np.ones([30]),[1.0,0,0])

         for i in range(20):
             A = mcstep(A, P)

         A.mean(axis=0)
```

```
Out[12]: array([0.43333333, 0.53333333, 0.03333333])
```

1.3 Autograd

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

```
In [13]: import autograd.numpy as ag_np
         from autograd import grad


         x = ag_np.ones(1)

         y = lambda x: 3 * x**2 + 2

         grad(y)(x)
```

```
Out[13]: array([6.])
```

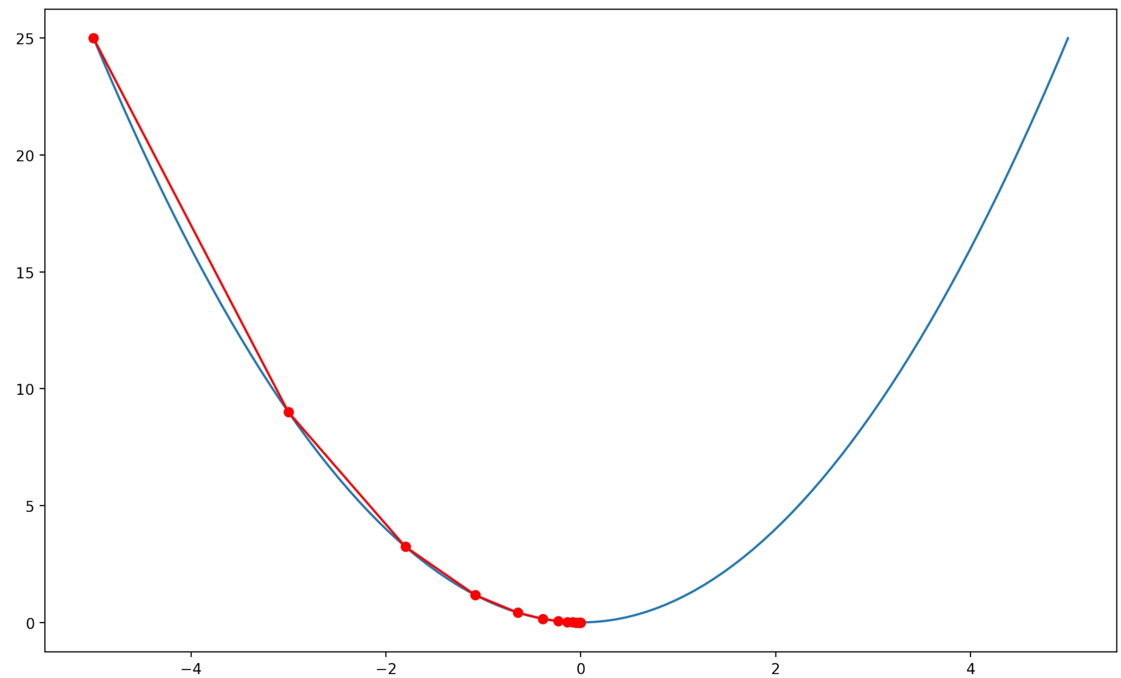
```
In [14]: x1 = ag_np.ones(1)
         x2 = ag_np.ones(1)

         y = lambda x1,x2 : 3*x1**3 + 4*2**x2

         print(grad(y,0)(x1,x2))
         print(grad(y,1)(x1,x2))
```

```
[9.]
[5.54517744]
```

In [15]:
```
y = lambda x: x**2

xs = [-5.0]
while abs(grad(y)(xs[-1])) > 1e-2:
    xs.append(
        xs[-1] - .2*grad(y)(xs[-1])
    )
```

In [16]:
```
plt.figure(figsize=(13,8), dpi=200)
plt.plot(np.linspace(-5, 5, 100),y(np.linspace(-5, 5, 100)))
plt.plot(xs, y(np.array(xs)), "-o", c="r")
```

Out[16]:
```
[<matplotlib.lines.Line2D at 0x112af1748>]
```



1.4 GPUs

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

In [18]:
```
import numpy as np
b = np.random.normal(size=(5000,5000))
%timeit b.dot(b)

#import cupy as cp
#a = cp.random.normal(size=(5000,5000))
#%timeit a.dot(a)
```

```
1.73 s ś 105 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

1.5 Cython

In [17]:
```
'''
# cython_dot.pyx
from numpy cimport ndarray
cimport numpy as np

def vec_dot(ndarray[np.float64_t, ndim=1] a, ndarray[np.float64_t, ndim=1] b):

        cdef np.float64_t result
        cdef Py_ssize_t n = a.shape[0]

        result = 0
        for i in range(n):
                result += a[i] * b[i]

        return result
'''

'''
# setup.py
from distutils.core import setup
from Cython.Build import cythonize
import numpy

setup(
    ext_modules=cythonize("cython_dot.pyx"),
    include_dirs=[numpy.get_include()]
)
'''

'''
# python_dot.py
def vec_dot(a, b):

        result = 0

        for i in range(a.shape[0]):
                result += a[i] * b[i]

        return result
'''
```

Out[17]:
```
'\nfrom numpy cimport ndarray\ncimport numpy as np\n\ndef vec_dot(ndarray[np.float64_
```

In [ ]:
```
import numpy as np
from cython_dot import vec_dot as cy_vec_dot
from python_dot import vec_dot as py_vec_dot
```

```
b = np.random.normal(size=(20000))
a = np.random.normal(size=(20000))

%timeit py_vec_dot(a,b)
%timeit cy_vec_dot(a,b)
%timeit np.dot(a,b)
```