Digitális képfeldolgozás

Vektorizálás: Gyors műveletek OpenCV / NumPy alatt

Horváth András , SZE GIVK

v 1.1

...

Alapok

C, C++, ..: “compiler”
A futás előtt a program értelmezése, gépi kódra fordítása.
A gépi kódú, optimalizált program fut.

Python, PHP, ...: “interpreter”
A futás előtt csak egy gyors előfeldolgozás.
Futás közben folytonosan értelmezi az utasításokat.

Alapok

C, C++, ..: “compiler”
A futás előtt a program értelmezése, gépi kódra fordítása.
A gépi kódú, optimalizált program fut.

Python, PHP, ...: “interpreter”
A futás előtt csak egy gyors előfeldolgozás.
Futás közben folytonosan értelmezi az utasításokat.

Python programok végrehajtása sokkal lassabb lehet, mint pl. C esetén.

Alapok

C, C++, ..: "compiler"
A futás előtt a program értelmezése, gépi kódra fordítása.
A gépi kódú, optimalizált program fut.

Python, PHP, ...: "interpreter"
A futás előtt csak egy gyors előfeldolgozás.
Futás közben folytonosan értelmezi az utasításokat.

Python programok végrehajtása sokkal lassabb lehet, mint pl. C esetén.

Mit lehet tenni ha lassú a Python programunk?

- C/C++-ban újraírni (időigényes)
- Csak a kritikus részeket alakítani C/C++-ra (tanulásigényes)
- Úgy írni Pythonban, hogy az értelmezés kevés időt vegyen el.

Vektorizálás: olyan jelölés, melyben egy művelet nemcsak egy számra, hanem egész tömbökre vagy részeikre vonatkozik.

Eredmény: gyorsabb és tömörebb kód.

A NumPy-ról

NumPy: “Numerical Python”
Python modul az azonos típusokból álló tömbök gyors, tömör kezelésére.

NumPy tömbökre való hivatkozások automatikusan ciklusokká fordulnak le.

A NumPy-ról

NumPy: “Numerical Python”
Python modul az azonos típusokból álló tömbök gyors, tömör kezelésére.

NumPy tömbökre való hivatkozások automatikusan ciklusokká fordulnak le.

Használat:

- Telepíteni a NumPy-t
- Importálni a NumPy modult (Pl. “import numpy as np”)

Ebben a kurzusban nem beszélünk általában a NumPy-ről, csak a szükséges pár dolgot tanuljuk meg.

(Érdemes többet is megtanulni a NumPy-ről, mert sok helyütt hasznos.)

NumPy adattípusok

Az adattípus jó megválasztása növeli a hatékonyságot.

Hasznos alap adattípusok:

- int8, int16, int32, int64: 8, 16, ... bites egész (előjeles)
- uint8, uint16, uint32, uint64: 8, 16, ... bites egész (előjel nélküli)
- float32, float64: egyszeres- és duplapontos lebegőpontos
- bool: logikai érték

NumPy adattípusok

Az adattípus jó megválasztása növeli a hatékonyságot.

Hasznos alap adattípusok:

- int8, int16, int32, int64: 8, 16, ... bites egész (előjeles)
- uint8, uint16, uint32, uint64: 8, 16, ... bites egész (előjel nélküli)
- float32, float64: egyszeres- és duplapontos lebegőpontos
- bool: logikai érték

A modulnevet is meg kell adni, ha használjuk őket!

```
import numpy as np  #

a=np.uint8(10)
b=np.uint8(-10)   # !!!
c=np.float32(10)
d=np.int16(123.456)
```

A képfeldolgozásban kiemelten fontos, hogy tudjuk, milyen típusú egy tömb!

NumPy tömbök létrehozása

A NumPy tömbök csak egyfajta típusból állhatnak. (nem úgy, mint a Python listák)
Néhány létrehozási mód:

```
import numpy as np

aa=np.zeros((10,20), np.uint8)
          # 10x20-as tömb bájtokból, 0-kkal feltöltve
bb=np.ones((30,30), np.float32) * 100.0
          # 30x30-as float tömb, 100.0-kkal feltöltve

cc=np.zeros(aa.shape, np.float64)
          # aa-val megegyező méretű tömb, de float64-ekkel

dd=np.zeros(aa.shape[0], np.int16)
          # olyan hosszú tömb, ahány sora van aa-nak

ee=aa.copy()    # lemásolja aa-t mindenestül
```

NumPy tömbök konvertálása

Listából:

```
ll = [[11,12,13], [21,22,23]]  # listák listája: 2D tömbféleség

tll = np.asarray(ll, np.uint8) # ez már NumPy bájttömb lesz
fll = tll.astype(np.float32)   # továbbalakítjuk float-ra

lll = fll.tolist()             # vissza listára
```

A Python listák és a NumPy tömbök hasonlónak tűnnek, de jelentős különbségek vannak!

- "`tll*2`" egy ugyanakkora tömb, duplázott elemekkel
- "`ll*2`" egy nagyobb lista (`ll` kétszer egymás után)

NumPy tömbök konvertálása

Listából:

```
ll = [[11,12,13], [21,22,23]]  # listák listája: 2D tömbféleség

tll = np.asarray(ll, np.uint8) # ez már NumPy bájttömb lesz
fll = tll.astype(np.float32)   # továbbalakítjuk float-ra

lll = fll.tolist()             # vissza listára
```

A Python listák és a NumPy tömbök hasonlónak tűnnek, de jelentős különbségek vannak!

- "`tll*2`" egy ugyanakkora tömb, duplázott elemekkel
- "`ll*2`" egy nagyobb lista (`ll` kétszer egymás után)

Az OpenCV képei valójában NumPy tömbök!

Színes képek: (sor, oszlop, csatorna)

Szürkeárnyalatos képek: (sor, oszlop)

Vektorizálás tartomány-kijelöléssel

A tömbökre vonatkozó tartomány-jelölések: elemi vektorizálás.

```
im[ 0:3 , : ] = [0,0,0]  # az első 3 sor nullázása
im2[ : , : ] *= 2        # az egész tömb értékeinek duplázása
im2 *= 2                 # előző tömörebben
im3[:,:,0] = im2[:,:,1]  # im2 1-es csatornáját másolja
                         #     im3 0-s csatornájába
im4 = 255 - im4          # im4 invertálása
im5 = im1 + im2          # összegzés
```

Vigyázat!

- Csak egymásnak megfelelő méretekre működik! (A pontos szabályok bonyolultak. Lásd: “broadcasting”.)
- Túl/alul csordulást kezelni kell!

Vektorizálás és matematikai műveletek

A műveleti jelek automatikusan vektorosak, de a “math” modul függvényei nem vektorizálhatók.

```
im5 = im1 + im2          # OK
im6 = im5**2 + 2         # OK
im7 = im5**2 - im6**3    # OK
im8 = math.sin(im7)      # ERROR!
```

A NumPy függvények viszont igen:

```
im8 = np.sin(im7)      # OK
```

Vektorizálás logikai operátorok értékeivel

Minta probléma: hogyan duplázzuk a pixelértékeket túlcsordulás nélkül?
Hasznos trükk: "True"=1, "False"=0

```
im2 = im1*2                 # túlcsordulhat!
im2 = (im1<128) * (im1*2)  # túlcsordulás helyett 0
#     ^^^^^^^^^ ez "True" vagy  "False" (1 vagy 0)
im2 = (im1<128)    *   (im1*2) +   (im1>=128)  *  255
# ha   im1<128  akkor duplázás ; ha im1>=128 akkor 255
```

Ez igen gyors megoldáshoz vezet! (Ha sikerül...)

Vektorizálás logikai operátorok értékeivel

A fentiekből bonyolultabb műveletek is összerakhatók.

```
im3 = (im2 > 10) * im1 + (im2 <= 10) * im2
# ahol im2 10-nél nagyobb, ott az im1 értékét vesszük,
# különben az im2-ét

im4 = im4 * ( (im4>100) & (im4<200) )
# meghagyjuk im4-et, ha 100 és 200 közti értékű
# különben nullázzuk

im4 *= ( (im4>100) & (im4<200) )
# ugyanez rövidebben
```

(Logikai műveletek: “&” (és), “|” (vagy), “^” (kizáró vagy), “~” (tagadás))

Vektorizálás feltételes indexeléssel

Ha a tömbindexbe egy feltételt írunk be, akkor a végrehajtás csak azokra az értékékre történik, ahol a feltétel teljesül.

```
im5[ im4<10 ] = 0    # ahol im4<10, ott nullázunk
im6[ im6<128 ] *=2   # ahol im6<128, ott duplázzuk
```

Sokszor elegáns megoldás, de a fenti logikai operátoros gyakran gyorsabb.

Vektorizálás "lookup table"-k használatával

Lookup table: bonyolult függvények értékeit előre kiszámoljuk és letároljuk.
Példa:

```
import numpy as np  # NumPy modul

lt=np.zeros(256, np.uint8)  # ez lesz a "lookup table"

for i in range(256):        # feltöltjük
        lt[i]=int( (np.sin(i/16.0)+1.0)*127 )
        # akármilyen bonyolultat ide írhatunk

# mostantól akárhányszor használhatjuk:
im3= lt[im2]    # im2 minden pixelére kikeresi a kimenetet lt-ből
```

Hatékony, gyors, de csak akkor működik, ha a tömbelemek nemnegatív egészek és nem túl nagyok.

Összegzés, átlag, ...

Összegzés : `.sum()`

- `aa.sum()`: aa minden elemének összegzése
- `aa[:,:,1].sum()`: ha aa egy BGR kép, akkor ez a G értékeket összegzi
- `aa.sum(dtype=np.uint32)`: összegzés np.uint32 típusú változóba

Összegzés, átlag, ...

Összegzés : `.sum()`

- `aa.sum()`: aa minden elemének összegzése
- `aa[:,:,1].sum()`: ha aa egy BGR kép, akkor ez a G értékeket összegzi
- `aa.sum(dtype=np.uint32)`: összegzés np.uint32 típusú változóba

Átlag : `.mean()`

- `aa.mean(dtype=np.float64)`: átlagolás duplapontosan

Összegzés, átlag, ...

Összegzés : `.sum()`

- `aa.sum()`: aa minden elemének összegzése
- `aa[:,:,1].sum()`: ha aa egy BGR kép, akkor ez a G értékeket összegzi
- `aa.sum(dtype=np.uint32)`: összegzés np.uint32 típusú változóba

Átlag : `.mean()`

- `aa.mean(dtype=np.float64)`: átlagolás duplapontosan

Szórás, variancia : `.std()`, `.var()`

- `aa.std(dtype=np.float64)`: aa szórása
- `aa.var(dtype=np.float64)`: aa varianciája (std**2=var)

Egyéb függvények

Minimum, maximum, ... :

- `aa.min()`, `aa.max()`: aa minimuma/maximuma
- `np.median(aa)`: aa mediánja
- `np.minimum(aa, bb)`, `np.maximum(aa, bb)`: elemenkénti minimum/maximum két tömbre

Egyéb függvények

Minimum, maximum, ... :

- `aa.min()`, `aa.max()`: aa minimuma/maximuma
- `np.median(aa)`: aa mediánja
- `np.minimum(aa, bb)`, `np.maximum(aa, bb)`: elemenkénti minimum/maximum két tömbre

Logikai operátorok :

- `np.logical_and(aa, bb)`: logikai és aa és bb értékeire
- `np.logical_or(aa, bb)`, `np.logical_xor(aa, bb)`, `np.logical_not(aa, bb)`: ...

Számítások kiválasztott irányokban

Gyakran csak soronként vagy pixelenként akarunk összegezni, átlagot számolni, stb.
NumPy: műveletek tengelyek (axis) mentén .
A következőkben `im` egy 100 soros, 200 oszlopos, 3 színcsatornás kép.

- `im.sum()`: im összes értékének összege, 1 szám
- `im.sum(axis=0)`: a sorok összeadása, 200x3-as tömb
- `im.mean(axis=2)`: pixelenkénti átlag, 100x200-as tömb
- `im.min(axis=(0,1))`: a 3 színcsatorna minimuma, 3-as tömb

Igen gyors és tömör műveletek. Kiválthatók ciklusokkal, de nem érdemes.

Hasznos műveletek

A NumPy igen sok mindent tud, melyekre csak röviden utalunk.

- Igen sok hasznos függvény van a NumPy-ban definiálva. Pl:
  - `np.clip()`: vágás két érték közé
    `img2=np.clip(img*5.0, 0.0, 255.0).astype(np.uint8)`

Hasznos műveletek

A NumPy igen sok mindent tud, melyekre csak röviden utalunk.

- Igen sok hasznos függvény van a NumPy-ban definiálva. Pl:
  - `np.clip()`: vágás két érték közé
    `img2=np.clip(img*5.0, 0.0, 255.0).astype(np.uint8)`
  - `np.full()`: új tömb adott értékekkel való feltöltése
    `np.full((100, 100), 42, dtype=np.uint8)`
    `numpy.full((200,100,3), [10,50,200], numpy.uint8)`

Hasznos műveletek

A NumPy igen sok mindent tud, melyekre csak röviden utalunk.

- Igen sok hasznos függvény van a NumPy-ban definiálva. Pl:
  - `np.clip()`: vágás két érték közé
    `img2=np.clip(img*5.0, 0.0, 255.0).astype(np.uint8)`
  - `np.full()`: új tömb adott értékekkel való feltöltése
    `np.full((100, 100), 42, dtype=np.uint8)`
    `numpy.full((200,100,3), [10,50,200], numpy.uint8)`
  - `np.fromfunction()`: tömb létrehozás, függvény szerinti értékekkel feltöltve
    `np.fromfunction(lambda i, j: 2*i+j,(3,3),dtype=np.uint8)`

Hasznos műveletek

A NumPy igen sok mindent tud, melyekre csak röviden utalunk.

- Igen sok hasznos függvény van a NumPy-ban definiálva. Pl:
  - `np.clip()`: vágás két érték közé
    `img2=np.clip(img*5.0, 0.0, 255.0).astype(np.uint8)`
  - `np.full()`: új tömb adott értékekkel való feltöltése
    `np.full((100, 100), 42, dtype=np.uint8)`
    `numpy.full((200,100,3), [10,50,200], numpy.uint8)`
  - `np.fromfunction()`: tömb létrehozás, függvény szerinti értékekkel feltöltve
    `np.fromfunction(lambda i, j: 2*i+j,(3,3),dtype=np.uint8)`
- ...

Érdemes nézegetni a dokumentációt, ha nem tudjuk egyszerűen / hatékonyan megoldani a feladatunkat. (de a vizsgán elég annyit tudni, ami az órai anyagokban van)

Egyszerű gyakorlati példa

Feladat: RGB értékek átlagának számolása. (szürke árnyalatos kép)
A túl/alul csordulások kikerülése: float-ra váltás:

```
im=cv2.imread("...")

fim_gray = im[:,:,0].astype(np.float32)  # B-t lebegőpontosan
fim_gray+= im[:,:,1].astype(np.float32)  # hozzáadjuk G-t
fim_gray+= im[:,:,2].astype(np.float32)  # hozzáadjuk R-t

im_gray  = (fim_gray/3.0).astype(np.uint8) # kész a szürke kép
```

Egyszerű gyakorlati példa

Feladat: RGB értékek átlagának számolása. (szürke árnyalatos kép)
A túl/alul csordulások kikerülése: float-ra váltás:

```
im=cv2.imread("...")

fim_gray = im[:,:,0].astype(np.float32)  # B-t lebegőpontosan
fim_gray+= im[:,:,1].astype(np.float32)  # hozzáadjuk G-t
fim_gray+= im[:,:,2].astype(np.float32)  # hozzáadjuk R-t

im_gray  = (fim_gray/3.0).astype(np.uint8) # kész a szürke kép
```

... vagy használjuk a NumPy irányokat

```
im_gray=im.mean(axis=2)
```

Összetettebb példa

Feladat: fényesítés, túlcsordulás elleni védelemmel

```
mul=1.5   # ennyiszeresre akarjuk növelni a fényességet


# logikai operátorok numerikus értéke:
im = (im*mul>255.0)*255 + (im*mul<=255)*im


# feltételes indexelés
kicsik=im<np.uint8(255/mul)   # azon helyek "térképe", ahol nem lesz túlcsordulás
im[kicsik] = im[kicsik]*mul   # itt bátran szorozhatunk
im[~kicsik]= 255              # ~kicsik = kicsik logikai inverze
```

Összetettebb példa (folyt.)

```
# lookup table:
lt=np.zeros(256, np.uint8)  # ez lesz a táblázat
for i in range(256):        # kezdőértékekkel való feltöltés
    if i*mul>255:
        lt[i]=255
    else:
        lt[i]=np.uint8(i*mul)

im=lt[im]  # itt történik a "számítás" az összes pixel-értékre


# speciális NumPy művelet, a clip használata
im=np.clip(im*mul, 0.0, 255.0).astype(np.uint8)
```

Gyakorlaton megnézzük, melyik a gyorsabb.