Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

M8 – Osztályozás mély neurális hálókkal

Mérési Útmutató

Irányítástechnika és Képfeldolgozás Laboratórium

Szemenyei Márton

Tartalomjegyzék

1 Neurális hálók

A számítógépes látás a számítástudomány egyik legrohamosabban fejlődő területe, amelynek egyre több gyakorlati felhasználása létezik. Ez a tendencia nem meglepő, hiszen az ember az érzékszervei közül a szemre hagyatkozik a leginkább a napi feladatinak ellátásakor. Ebből következik, hogy ha a számítógépes látás algoritmusai képesek megközelíteni, vagy akár felülmúlni az emberi látás képességeit, akkor számos fontos feladatot leszünk képesek automatizálni. Könnyen belátható azonban az is, hogy a gyakorlatban ez egy rendkívül nehéz feladat, mivel általában olyan feladatokat tudunk könnyedén algoritmusok formájában leírni, ahol a feladat elvégzésének a menetét pontosan, tudatos szinten megértjük. A szemből érkező jelek feldolgozásának azonban a jelentős része tudatalatti szinten történik, így aligha tudjuk ezeket a folyamatokat könnyedén megérteni, és algoritmusok formájában lemásolni.

Ennek a problémának a megoldásához a mesterséges intelligencia, ezen belül is a gépi tanulás módszereihez kell nyúlnunk. A tanuló számítógépes látás legegyszerűbb formája az osztályozás, vagyis amikor egy képhez egyetlen címkét rendelünk, amely a képen található objektum kategóriáját kódolja. Bizonyos esetekben a címke mellé már egy az adott objektumot körbefogó téglalapot is rendelünk, ebben az esetben lokalizációról beszélhetünk. Adott esetben ennél magasabb szintű információt is szeretnénk kinyerni a képből: érdekelhet minket az, hogy melyek azok a pixelek az adott képen, amelyek az észlelt osztályhoz tartoznak. Amennyiben ezt az információt egy képsorozaton minden pixelre meghatározzuk, akkor követésről beszélhetünk.

A neurális hálózatokról és azok tanításáról bővebben a Számítógépes Látórendszerek c. tárgy jegyzetében [1] olvashatunk.

2 A mérés környyezete

A mérés során a *PyCharm* elnevezésű IDE áll rendelkezésre, amely rendkívül sokoldalú szolgáltatásokkal könnyíti meg a szoftverfejlesztést, például konfigurálható automatikus formázási lehetőségek állnak rendelkezésünkre. További részletekért érdemes lehet a JetBrains ide vonatkozó weboldalát [2] felkeresni. Függvények, objektumok esetében a ***Ctrl+P*** billentyűkombináció pop-up segítségként szolgálva mutatja nekünk a paramétereket. A mérés során használt programnyelv a Python 3-as verziója lesz.

2.1. A Python nyelv

A Python programnyelv egy ***interpretált szkriptnyelv*** (a saját bytekódját hajtja végre), mely alapvetően az *objektumorientált* paradigma által vezérelve került kialakításra. Jelenleg kétféle, kismértékben eltérő főverziója érhető el, ezek közül a mérésen a 3-as verziót használjuk majd.

Az OOP szemlélet olyannyira központi szerepet tölt be Pythonban, hogy minden változó objektumnak tekinthető – ami azt jelenti, hogy az integer és float típusok is objektumok. A nyelv számára jelentős hátrány adatintenzív számítási feladatok elvégzésekor ennek ténye, ugyanis arról beszélünk, hogy a Python ***nem rendelkezik natív számtípussal***. Nem véletlen, hogy a hatékony modulok alacsonyabb szintű nyelveken nyugszanak, így a következő alfejezetben ismertetett PyTorch is.

További nem szokványos jellemzője a nyelvnek, hogy ***nem erősen típusos***, a változónevekhez futási időben rendelődik hozzá a referált objektum (azaz egy ***változónév*** igazából egy referens, vagyis adott ***példányra mutató referenciát*** tartalmaz). Habár Pythonban a referenciák teljesen úgy viselkednek, mint a változók (vagyis semmilyen szintaktikai kiegészítésre nincs szükség, még argumentumok átadása esetében sem, mint ahogy azt C++-ban láttuk). (A félreértések elkerülése végett fontos szem előtt tartani, hogy amennyiben a segédletben a továbbiakban változó szerepel, akkor is igazából referencia van a háttérben, ezért hangsúlyozott általában, hogy nem a változó, hanem a változónév tartalmazza az objektumra mutató referenciát.)

Vagyis lehetőség van arra, hogy egy referenshez (változónévhez) a programban különböző típusú objektumokat rendeljünk hozzá – ez teljesen logikusnak tűnik, ha belegondolunk abba, hogy a „Pythonban minden objektum" korábbi kijelentés lényegében azt sejteti, hogy minden az objektum ősosztály leszármazottja. A referenciák kezelése referenciaszámláló segítségével történik – a koncepció analóg a például C++-ban megtalálható *shared_ptr* típus esetén használttal.

Pythonban nincsen továbbá pointer sem, az argumentumok átadása ***referencia szerint*** történik. Itt azonban meg kell különböztetnünk az objektumokat az alap-

ján, hogy módosíthatóak-e. A következőket érdemes alaposan átgondolni, különben hibás működésű programot kaphatunk.

Kétféle objektumtípus létezik, ***módosítható*** (mutable), ill. ***nem módosítható*** (immutable). Nem módosíthatóak többek között az egyszerű adattípusok (POD, plain old data), mint az egész vagy lebegőpontos számok, valamint a sztringek. Habár a gondolatmenet e megfontolás mögött elsőre furcsának tűnik, a következőképpen magyarázható: minden adott szám vagy szöveg egyedi, vagyis ha pl. két változóhoz hozzárendeljük az 5 értéket, akkor a kettő tartalma egymással teljesen azonos, ha az egyiket megváltoztatjuk, akkor az már egy másik objektum lesz: nem egy tagváltozót módosítunk, hanem magát az objektumot teljes mértékben és kizárólagosan azonosító elemet. Talán érdemes a fordított programnyelvekből egy szemléletes példát átgondolni: a fordító ugyanolyan értékeket helyettesít be a gépi kódba (érdeklődők számára: a Pythonnak esetében is elérhető egyfajta disassembly a dis modul segítségével).

Módosítható lényegében minden egyéb típus, így a Pythonba beépített konténer jellegű típusok, mint list, dict (ezek különlegessége, hogy nem csak egy típust képesek egyidejűleg magukba foglalni, hanem bármilyen objektumot), de a saját osztályok példányai is ide tartoznak. Ebben az esetben a módosítás nem eredményezi új objektum létrehozását. A tuple egy különleges eset, ugyanis ez a konténer típus immutable, azonban, ha mutable objektumokat tartalmaz, akkor azok módosulhatnak.

Ezek a különbségek objektumok másolása esetében is jelentkezik. Fontos különbség, hogy Pythonban alapvetően a hozzárendelés (assignment) igazából C++-szemszögből inkább a copy constructor hívásának feleltethető meg. Mutable esetben az eredeti objektumhoz tartozó referenciszámláló kerül megnövelésre, módosítás esetén pedig új objektum jön létre, értelemszerűen ugyanarra az értéket tartalmazó változók egyikének megváltoztatása nem hat ki a többire. Immutable esetben azonban nem ilyen egyszerű a helyzet: mivel alapvetően referenciákat tartalmaznak a változónevek, amelyek módosítható objektumokat referálnak, így ugyanarra a példányra mutató referenciák bármelyikének módosítása változást eredményez a referált egy darab objektum esetében, vagyis bármelyik változóval hivatkozunk rá, a változás mindegyik esetben látható lesz számunkra. Az ilyen jellegű másolást shallow copy-nak szokás nevezni, melynek párja a deepcopy (elérhető a copy modulban), ami Pythonban immutable esetben is új példányt hoz létre, így az új objektum független lesz a többitől.

Paraméterek átadása hozzárendeléssel történik, ez azonban nagy objektumok esetében sem okoz komolyabb problémát, ugyanis a referenciák kerülnek csak másolásra. Ez a láthatóságra a következőképpen van hatással: ha a paraméter immutable és a függvénytörzsben módosításra kerül, akkor lényegében a függvény scope-jában egy ideiglenes objektum kerül létrehozásra, a függvényből visszatérve a változó megtartja eredeti értékét. Immutable esetben, mivel a referencia kerül másolásra, az objektumok másolásánál láttuk, hogy az általuk referált objektumok módosítása érvényes, bármelyik referenciájával hivatkozunk is rá, így a függvény visszatérését követően az objektum már módosult értékével használható. Rendkívül hasznos tulajdonság, hogy a Python gyakorlatilag bármennyi visszatérési értéket támogat. A nyelvi koncepciók ismertetése után a következőkben a szintaktikai részletek kerülnek összefoglalásra.

Pythonban a programkód tagolása indentálással történik, vagyis a kódblokkokat egy tabulátorral beljebb kell kezdeni (ha valamilyen okból üres függvényt, ciklust, stb, kívánunk írni, akkor is kell egy indentált blokk, ezt egy sorban, a pass utasítással valósíthatjuk meg, ami nem hajt végre semmilyen műveletet).

Modulok betöltésére az import utasítással van lehetőségünk, mégpedig kétféle módon: importálhatjuk a teljes modult, ekkor a modul minden osztálya/függvénye a modul neve után írt „.” (pont) operátorral érhető el, ha from-ot használunk, lehetőségünk van csak egyes elemeket betölteni, ekkor a modul nevét nem kell az importált elem neve elé kiírni.

```
import module
m = module.MyClass()

import module as md
m = md.MyClass()

from module import MyClass, my_func
m = myClass()
```

Függvények a következő módon hozhatók létre:

```
def func(x):
x += 1
print("x = ", x)
```

A def kulcsszó után a függvény neve, majd a paraméterlista kerül megadásra, azt követően pedig az indentált függvénytörzs következik.

Osztályok esetében sem bonyolult a konstrukció:

```
class my_class(object):
def __init__(self):
 self .x = 5
```

Pythonban a konstruktort az __init__ (2-2 aláhúzással) rutin testesíti meg, mint látható, a tagfüggvények is majdnem teljesen megegyeznek az általános függvényekkel, azzal a különbséggel, hogy az első argumentum mindenképpen az adott példányra vonatkozik (mint ahogy a this C++-ban) – ezt konvenció szerint self-nek szoktuk nevezni. Öröklés esetén nincs más teendőnk, mint az object osztály helyett megadni az általunk választott ősosztályt, majd a konstruktorban meghívni az ősosztály konstruktorát a super, általánosan az ősosztályra használható objektum segítségével.

```
class base_class(object):
def __init__(self):
print("I am Groot")
```

```
class inherited_class(base_class):
def __init__(self):
super().__init__()
print("I am inherited")
```

Aki mélyebben érdeklődik a Python nyelv iránt, annak érdemes felkeresnie további példaprogramokat és kódrészleteket [3], valamint a TMIT SmartLab blogjának bejegyzéseit [4], [5] (angolul).

A Python programnyelvhez számos hasznos függvénykönyvtár tartozik, melyek a mérési feladatok megvalósítását nagymértékben megkönnyítik. A Python nyelv egyik rendkívül kényelmes funkciója a beépített package manager, amelynek segítségével az egyes könyvtárak automatikusan telepíthetők, telepítsük után pedig minden további beállítás nélkül használhatók. A Pythonhoz két ilyen package manager is tartozik, az egyik a Pip, amely a legtöbb telepíthető Python verzió mellé automatikusan települ, a másik pedig az Anaconda [6], ami a könyvtárkezelési funkciókon túl virtuális környezeteket is képes kezelni.

A Python egyik legfontosabb függvénykönyvtára a Numpy, amely tömbök kezelésére, illetve számtalan numerikus algoritmus használatára ad lehetőséget. A Numpy funkcionalitását kiegészíti a MatplotLib, melynek segítségével különböző ábrákat készíthetünk a tömbjeinkről. Egy harmadik rendkívül hasznos könyvtárcsalád a scikit, ami számos tudományos számításhoz szükséges alkönyvtárt foglal össze. A sckit-image képek kezelésére, a scikit-learn gépi tanulás algoritmusok használatára, míg a scikit-fuzzy fuzzy logika használatára ad lehetőséget. Ezek a könyvtárak tulajdonképpen együttesen kiadják a Matlab funkcionalitásának jelentős részét.

2.2. PyTorch

A PyTorch [7], [8] a Facebook Research által fejlesztett Deep Learning keretrendszer, melyet a mérés során használni fogunk, így a következőkben röviden áttekintésre kerülnek a fontosabb koncepciók. A PyTorch egyik előnye a TensorFlow-val, Keras-szal szemben, hogy a hálózatot leíró gráf dinamikusan kerül létrehozásra, nincs fordítási szakasza a tanításnak – ez a kevesebb lépés mellett akár azt is lehetővé teszi számunkra, hogy a háló rétegeit tanítás közben változtassuk (mert az egyik hiperparaméterünk lehet a lineáris rétegek száma).

A PyTorch alapja az akár GPU-n is futtatható tenzor típus, mely egy pratikus tagfüggvényekkel felvértezett, multidimenzionális tömböt megvalósító osztály – tehát felettébb alkalmas például képek tárolására is.

Tenzorokon alapul a rétegeket/hálózatokat megvalósító Module osztály, melyből minden saját hálózatunknak örökölnie kell, így kerülnek regisztrálásra az előre-, ill. visszaterjesztést megvalósító függvényeink. Szerencsére nem kell a hibavisszaterjesztést kézzel megírni, a PyTorvh beépített automatikus differenciálásra képes modulja (autograd) megteszi ezt helyettünk.

Fontos megjegyezni, hogy az automatikus differenciálás nem szimbolikus differenciálás (azaz nem az a célja, hogy paraméteresen felírja egy bonyolult összefüggés deriváltfüggvényét) és nem is numerikus differenciálás (tehát az sem cél, hogy iteratívan közelítsünk egy értéket, mert a kvantizálás/kerekítés jelentős eltérésekben nyilvánulhat meg). Automatikus differenciálás során egy számítási gráf segítségével, a láncszabályt alkalmazva vagyunk képesek, viszonylag jól skálázhatóan, parciális deriváltakat kiszámítani, ami feltétlenül szükséges gradiens-alapú tanulóalgoritmusok esetében.

A PyTorch optim nevű almodulja tartalmazza a különböző, a tanítás során használt algoritmusokat, ezek közül nekünk most csak az SGD-re (stochastic gradient descent) lesz szükségünk. Az SGD objektum számára megadható a modell, a tanulási

ráta (learning rate), a momentum, ill. van lehetőség normán alapuló regularizáció konfigurálására is.

Ahhoz, hogy tanítani tudjunk egy hálózatot, a következő struktúrájú programot kell megvalósítanunk és iteratívan meghívnunk. A következőkben a PyTorch egyik kiegészítésének (Ignite) forráskódjából látható egy részlet, ami tömören magába foglalja a szükséges lépéseket – a mérésen ugyanezen lépések segítségével, de egy kicsit más struktúrában fogjuk a tanítást végezni.

```
def _update(batch):
    model.train()
    optimizer.zero_grad()
    x, y = _prepare_batch(batch, device=device)
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    loss.backward()
    optimizer.step()
    return loss.item()
```

Részleteiben a következő történik: tudatnunk kell a programmal, hogy most tanítjuk a hálót (hogy pl. a dropout csak ilyen esetben kerüljön aktiválásra). Ezt követően a gradiensek előző értékét kell nulláznunk (ugyanis megtehetnénk, hogy valamihez még felhasználjuk, ezért nem történik automatikusan). Ahhoz, hogy a következő előreterjesztést véghez vihessük, elő kell készíteni a következő minibatchet (ezt a kódban egy saját függvény, _prepare_batch jelöli). Az előreterjesztés végrehajtása felettébb egyszerű, magának a modellnek adjuk át argumentumként a minibatchet. A hiba kiszámítása a következő lépés, ezt az általunk konfigurált hibafüggvény segítségével tehetjük meg, majd a hiba backward tagfüggvényét meghívva végrehajtjuk a hibavisszaterjesztést. Nagyon fontos az utolsó lépés az optimalizáló algoritmus (SGD) lépésének végrehajtása, így kerülnek a súlyok frissítésre.

Számos gyakran használt adatbázis elérhető a torchvision modulból, ezek előkészítésére, ill. adat augmentációra is használható transzformációk is rendelkezésünkre állnak. Ezen felül van lehetőség DataLoader objektumok segítségével (torch.utils.data modul) az adatok betöltését automatizálni. Továbbá lehetőségünk van például a tanulási ráta automatikus megváltoztatására adott szabályrendszer szerint.

2.3. OpenCV

Az OpenCV [9] egy nyílt forráskódú számítógépes látás algoritmusokat tartalmazó függvénykönyvtár. Az OpenCV elsődleges nyelve a C++, azonban elérhetőek hozzá hivatalos wrapperek többek között Java és Python nyelven. Az OpenCV rengeteg hivatalosan támogatott algoritmust tartalmaz, melyen felül a külön letölthető Contrib modulban harmadik felek által kifejlesztett további funkciók is elérhetők.

3 Mérési feladatok

A mérés folyamán az alábbi feladatokat kell elvégezni:

1. Készítsen egy egyszerű néhány rétegből álló neurális háló architektúrát az **??**. Táblázatban megadott paraméterek alapján.
2. Készítsen eljárást egyszerű neurális háló tanítására a méréshez használt adatbázison.
3. Végezzen el különböző tanításokat a **??**. Táblázatban megadott hiperparaméterek alapján.
4. Valósítson meg egy egyszerű közlekedési tábla detektáló és osztályozó algoritmust a mérésvezető által szolgáltatott képekre.

Típus	**Csatornák száma**	**Méret**	**Stride**
Konvolúciós	N_{CH}	3×3	1
Konvolúciós	$N_{CH} * 2$	3×3	2
Konvolúciós	$N_{CH} * 2$	3×3	1
Konvolúciós	$N_{CH} * 4$	3×3	2
Konvolúciós	$N_{CH} * 4$	3×3	1
Konvolúciós	$N_{CH} * 8$	3×3	2
Konvolúciós	$N_{CH} * 8$	3×3	1
Konvolúciós	$N_{CH} * 16$	3×3	2
Konvolúciós	$N_{CH} * 16$	3×3	1
Konvolúciós	$N_{CH} * 32$	3×3	2
Osztályozó	55	3×3	1

3.1. táblázat. *Az elkészítendő neurális háló architektúrája. Az osztályozó réteg lehet* 1×1 *konvolúció, vagy Fully Connected.*

N_{CH}	**LR**	**Eta_min**	**Momentum**	**Decay**
8	1e-1	1e-2	0.1	1e-3
8	1e-1	1e-2	0.9	1e-4
8	1	1e-2	0.9	1e-1
8	1e-3	1e-4	0.9	1e-5
16	1e-1	1e-2	0.9	1e-5

3.2. táblázat. *A Vizsgálandó hiperparaméter kombinációk.*

4 Hasznos kódrészletek

Különböző rétegek létrehozása PyTorch környezeteben

```
self.conv = nn.Conv2d(inplanes,planes,size,padding=size//2,stride=stride)
self.bn = nn.BatchNorm2d(planes)
```

Rétegek meghívása

```
out = F.relu(self.bn(self.conv(x)))
```

Térbeli dimenziók eldobása az osztályozó réteg után

```
out = out.squeeze()
```

Adat augmentációs függvények

```
transforms.RandomCrop(32, padding=4),
transforms.RandomHorizontalFlip(),
transforms.ColorJitter(brightness=0.25, contrast=0.25, saturation=0.25, hue=0.2),
transforms.ToTensor(),
transforms.Normalize((0.49139968, 0.48215827, 0.44653124),
                     (0.24703233, 0.24348505, 0.26158768))
```

Különböző augmentációs módszerek összefűzése

```
transform = transforms.Compose( <list of transforms> )
```

Adatbázis létrehozása

```
trainSet = torchvision.datasets.CIFAR10(root=root, download=True,
train=True/False, transform=transform)
```

Adat betöltő létrehozása

```
trainLoader = torch.utils.data.DataLoader(trainSet, batch_size=128, shuffle=True,
    num_workers=2)
```

Költségfüggvény létrehozása

```
criterion = nn.CrossEntropyLoss()
criterion = nn.MultiLabelMarginLoss()
```

Optimalizáló módszer létrehozás

```
optimizer = optim.SGD(net.parameters(), lr=0.1, momentum=0.9, nesterov=True,
    weight_decay=1e-4)
```

Tanulási ráta ütemező létrehozása

```
scheduler = lr_scheduler.StepLR(optimizer,10)
```

Neurális háló módjainak állítása

```
net.train()
net.eval()
```

Progress bar készítése

```
# Create progress bar
bar = progressbar.ProgressBar(0, len(trainLoader), redirect_stdout=False)
    # Inside the loop:
    bar.update(i)
# Upon finishing:
bar.finish()
```

Tipikus tanításra használt epoch

```
# Epoch loop
for i, data in enumerate(trainLoader, 0):

    # get the inputs
    inputs, labels = data

    # Convert to cuda conditionally
    if haveCuda:
        inputs, labels = inputs.cuda(), labels.cuda()


    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

Tanulási ráta ütemező léptetése

```
scheduler.step()
```

Modell mentése

```
torch.save(net, root + '/model.pth')
```

OpenCV kép PyTorch tenzorrá történő konvertálása

```
# BGR to RGB
img2 = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
# Convert to PIL image
pil_im = Image.fromarray(img2)
# Apply transform function, unsqueeze and convert to cuda
imageT = transform(pil_im).unsqueeze(0).cuda()
```

Küszöbözés határok közt

```
img_bin = cv2.inRange(img_hsv,np.array([0,100,80]),np.array([255,255,255]))
```

Morfológia

```
SE = np.ones((3, 3), np.uint8)
img_bin = cv2.morphologyEx(img_bin, cv2.MORPH_CLOSE, SE,iterations=4)
```

Kontúrok keresése

```
contours, _ = cv2.findContours(binImage,cv2.RETR_EXTERNAL,cv2.
    CHAIN_APPROX_SIMPLE)
```

Lista komprehenzió

```
outList = [function(element) for element in list  if condition]
```

Képrészlet kivágása (hard copy)

```
img2 = img[y1:y2,x1:x2].copy()
```

Kép átméretezése

```
cv2.resize(img,(32,32))
```

Numpy - torch konverzió és deimenziók cserélése

```
imgROI = torch.Tensor(imgROI)
permuted = imgROI.permute(2,0,1)
```

Téglalap rajzolása és kiírás

```
cv2.rectangle(img,(x1,y1),(x2,y2),(255,0,255),2)
cv2.putText(img,className,(x1,y1-10),cv2.FONT_HERSHEY_COMPLEX_SMALL
    ,1.0,(0,255,0))
```

5 Ellenőrző kérdések

1. Ismertesse röviden a Hinge és a Kereszt-entrópia költségfüggvényeket! Mi az előnyük/hátrányuk?
2. Hogyan működik a gradiens módszer? Milyen fontos kiegészítései vannak?
3. Mire való a backpropagation? Hogyan működik?
4. Ismertesse egy konvolúciós háló és gyakori rétegei felépítését!
5. Mi az a saliency? Hogyan állítható elő? Mi a guided backpropagation? Mi értelme van?
6. Milyen módszerekkel kerülhető el az overfitting? Ismertesse ezeket egy-egy mondatban!
7. Milyen módszerekkel javítható egy neurális háló konvergenciája? Ismertesse ezeket egy-egy mondatban!
8. Ismertesse röviden a PyTorch könyvtárat! Milyen alapvető adattípusai, moduljai léteznek? Milyen szolgáltatásokat nyújt neurális hálók számára?
9. Készítsen egy egyszerű szkriptet Python nyelven:
   (a) Töltse be a torch könyvtár nn nevű modulját!
   (b) Definiáljon egy új, MyNet nevű osztályt, melynek ősosztálya az nn.Module
   (c) Az osztály konstruktorában hozzon létre egy mult nevű tagváltozót, értéke legyen 5.
   (d) Definiáljon egy forward nevű tagfüggvényt, melynek bemeneti paramétere x, és a mult nevű tagváltozóval szorozza meg, az eredményt pedig adja vissza.
10. Hogyan néz ki egyetlen tanulási ciklus PyTorch-ban? (Pszeudó kód elég)

Bibliography

[1] M. Szemenyei, *Számítógépes Látórendszerek*, M. Szemenyei, Ed. BME, 2019. [Online]. Available: https://edu.vik.bme.hu/pluginfile.php/53608/mod_resource/content/0/jegyzetFull.pdf.

[2] *PyCharm Quick Start Guide*. [Online]. Available: https://www.jetbrains.com/help/pycharm/quick-start-guide.html.

[3] P. Reizinger, *Python Példaprogramok*. [Online]. Available: https://gist.github.com/search?utf8=%5C%E2%5C%9C%5C%93%5C&q=user%5C%3Arpatrik96%5C&ref=searchresults.

[4] ——, *Python under the hood — tips and tricks from a C++-programmers' perspective (Part I)*. [Online]. Available: https://medium.com/@SmartLabAI/python-under-the-hood-tips-and-tricks-from-a-c-programmers-perspective-01-b5f96895663.

[5] ——, *Python under the hood — tips and tricks from a C++-programmers' perspective (Part II)*. [Online]. Available: https://medium.com/@SmartLabAI/b52675c7c0af.

[6] *Anaconda*. [Online]. Available: https://www.anaconda.com/.

[7] *PyTorch Documentation*. [Online]. Available: https://pytorch.org/docs/stable/index.html.

[8] *PyTorch Tutorials*. [Online]. Available: https://pytorch.org/tutorials/index.html.

[9] *OpenCV Documentation*. [Online]. Available: https://docs.opencv.org/4.1.1/.