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

M9 – Szegmentálás mély neurális hálókkal

Mérési Útmutató

Irányítástechnika és Képfeldolgozás
Laboratórium 1.

Szemenyei Márton

Tartalomjegyzék

1 Neurális hálók

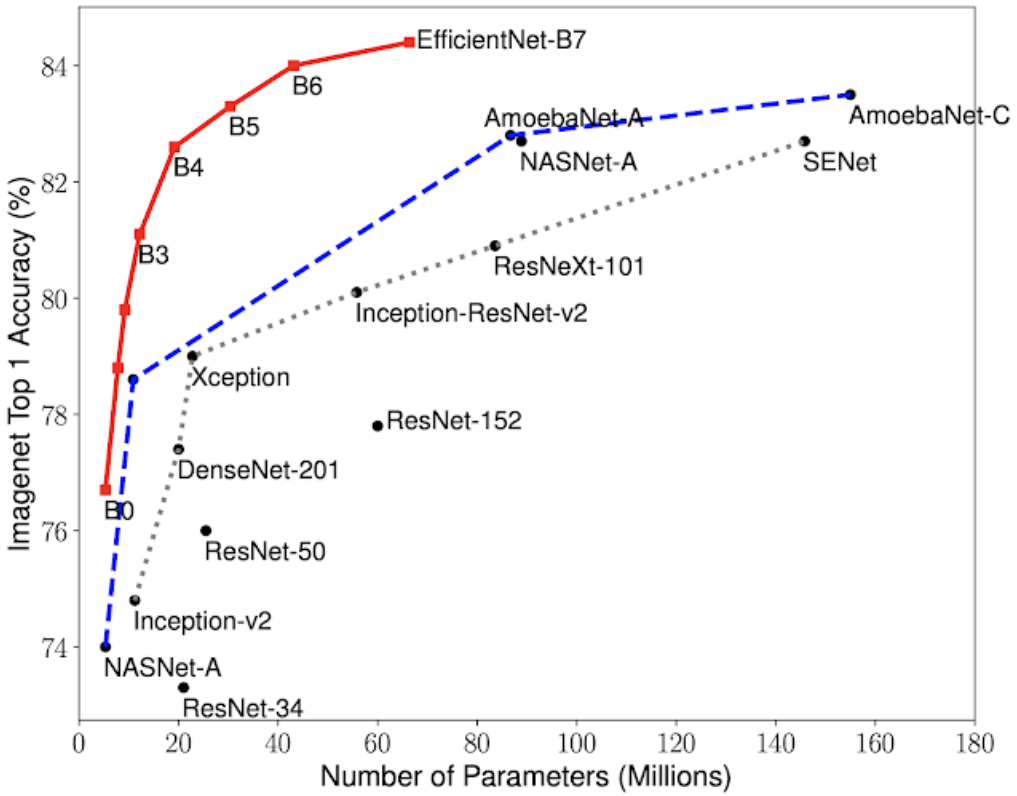
A számítógépes látás a számítástudomány egyik legrohamosabban fejlődő területe, amelynek egyre több gyakorlati felhasználása létezik. Ez a tendencia nem meglepő, hiszen az ember az érzékszervei közül a szemre hagyatkozik a leginkább a napi feladatinak ellátásakor. Ebből következik, hogy ha a számítógépes 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.

1.1. Az EfficientNet architektúra

Az EfficientNet [2] egy olyan konvolúciós neurális háló architektúra, amelyet automatikus optimalizáló eljárással (ún. Neural Architecture Search [3] módszerrel) fejlesztettek ki. Ez egy olyan eljárása, ahol egy megerősítéses tanulás ágenst arra tanítanak, hogy minél jobb neurális háló architektúrákat generáljon. Itt a jóság alatt a háló elért pontossága és az architektúra gyorsasága közötti valamilyen kompromisszumot értjük. Ezt jól illusztrálja az alábbi ábra:



1.1. ábra. *Az EfficientNet architektúra pontossága az ImageNet adatbázison a paraméterszám függvényében, más hálózatokkal összehasonlítva.*

Az EfficientNet architektúra implementálása maga nem a mérés része, azonban a konvolúciós blokkot érdemes vizsgálni, ugyanis ebben a korábban megismert trükkök közül több is megtalálható. Az EfficientNet alapvető blokkja ugyanis egy bottleneck-et tartalmazó reziduális blokk, amelynek fő konvolúciós eleme (amelyik 5×5 kiterjedésű) aszimmetrikus. Ez azt jelenti, hogy egy darab 5×5 konvolúció helyett egy 5×1 és egy 1×5 konvolúciót hajtunk végre egymás után. Ennek számítási igénye $O(5 \times 5 \times ch)$ helyett $O(2 \times 5 \times ch)$.

2 A mérés környezete

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 [4] 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.

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 [5], 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.

2.1. PyTorch

A PyTorch [6], [7] 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 kvantálás/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.

2.2. Haladó Python

A korábbi mérés(ek) során megismerkedtünk a Python nyelvvel, a jelen mérés során azonban a nyelv bonyolultabb funkcióira is szükségünk lesz. Az alábbiakban tekintsük át legfontosabb adattípusokat és nyelvi elemeket.

A Python legfontosabb konténer osztályai a list és a tuple. Ezek közt a különbség, hogy a tuple immutable, vagyis nem módosítható. Létrehozásuk és az elemeik elérése az alábbi módon történhet:

```
myList = [”wow”,”you”,”can”,”mix”,2,”or”,”more”,”different”,”types”]
myTuple = (”wow”,”you”,”can”,”mix”,2,”or”,”more”,”different”,”types”)
myList[4] = ”two” # 2 becomes ’two’
myTuple[4] = ”two” # Error
```

Érdemes tudni, hogy habár (1D) listák esetében a + operátor értelmezett, ez a listákat összefűzi, nem pedig összeadja.

```
myList = [1,2]+[3,4]
>>> [1,2,3,4] # Not [4,5]
```

Listákhoz új elemet az alábbi módon lehet hozzáadni:

```
myList = [1,2,3,4]
myList.append(5) # [1,2,3,4,5]
```

Listák, vagy Tuple-ök elemein az alábbi módokon lehet végigiterálni:

```
for elem in myList:
print(elem)

for i, elem in enumerate(myList):
print(i,elem) # i is the index
```

Adott esetben egyszerre több (egyenlő elemszámú) listán is végigiterálhatunk egyidejűleg:

```
if len(list1) == len(list2):
for (elem1,elem2) in zip(list1, list2):
print(elem1,elem2)
```

Érdemes megjegyezni, hogy az 'in' kulcsszó feltételeknél is használható annak eldöntésére, hogy egy adott érték szerepel-e a listában:

```
myList = [1,2,3,4]
if 1 in myList:
print("1 is in the list")
if 5 not in myList:
print("5 is not in the list")
```

A Python programnyelv támogat bizonyos funkcionális programozási elemeket is, melyek közül az egyik leghasznosabb az úgynevezett list comprehension. Ezt akkor használhatjuk, ha szeretnénk egy lista minden elemén elvégezni egy műveletet és ezek eredményeiből egy újabb listát készíteni. Az alábbi példa egy lista minden elemét négyzetre emeli:

```
myList = [1,2,3,4]
mySqrList = [elem**2 for elem in myList] # [1,4,9,16]
```

A list comprehension a korábban említett for ciklus változatokkal (zip, enumerate) is használható, az új lista elemének kiszámolásához pedig bármilyen érvényes Python kód írható. Ezen felül a kifejezés feltétellel is kiegészíthető:

```
myList = [9,-16,25,-4]
mySqrtList = [sqrt(elem) for elem in myList if elem >= 0] # [3,5]
```

2.3. Haladó NumPy

Fontos megjegyezni, hogy a Python list és tuple osztályok különböző típusú objektumokat tartalmazhatnak, vagyis egy lista egyik eleme lehet egy másik lista. Ez lehetőséget nyújt arra, hogy a lista osztályt N-dimenziós tömbként használjuk, azonban

egy listákat tartalmazó listában az egyes részlisták különböző hosszúak és típusúak lehetnek, ami hagyományos tömböknél nem fordulhat elő. Éppen ezért méret- és típus konzisztens tömbök tárolására a NumPy array osztályt célszerű használni.

A NumPy tömbök között definiált a $+$, $-$, $*$ és a / művelete, azonban ezeket elemenként végzi, így a tömbök méretének kompatibilisnek kell lennie. A NumPy broadcasting szabályai alapján két tömb az alábbi feltételek teljesítése esetén kompatibilis méretű:

1. A két tömb minden mérete megegyezik
2. Amelyik dimenzió mentén nem egyeznek meg a méretek, ott az egyik tömb mérete 1

A tömbök méretét az alábbi módon ellenőrizhetjük:

```
arr1.shape # [2,4,6]
arr2.shape # [1,4,1]  - Compatible
arr3.shape # [3,4,6]  - Incompatible
```

Bizonyos esetekben előfordulhat, hogy új elemet szeretnénk a tömbhöz adni, vagy két tömböt össze szeretnénk fűzni, amelyeket az alábbi módon tehetünk meg:

```
extArr = np.append(arr,elem,dim)
catArr = np.concatenate((arr1,arr2 ,..., arrn),dim)
```

A *dim* változó azt adja meg, hogy melyik dimenzió mentén történjen a kiegészítés/összefűzés. Ha None értéket adunk meg (nem adjuk meg a paramétert), akkor a NumPy megpróbálja magától kitalálni.

Gyakran előfordul, hogy egy általunk használt tömbben van egy extra dimenzió, aminek az értéke egy. Például, ha van egy 2D mátrixokból álló tömbünk, amelyiknek kivesszük egy elemét. Ez az extra dimenzió sok fejfájást tud okozni, mert ha konkatenálni, vagy mátrixokat szorozni szeretnénk, a NumPy nem megfelelő méret hibákat fog dobni. Ennek megoldására a squeeze parancs való.

```
mtxArr.shape # [N,3,3] - N 3x3 matrices
myMtx = mtxArr[i]
myMtx.shape # [1,3,3]
myMtx = mtxArr[i].squeeze()
myMtx.shape # [3,3]
```

Előfordulhat az is, hogy a tömbjeink megfelelő méretűek, de valamilyen oknál fogva a dimenziók sorrendje eltér. Például képek esetén elterjedt konvenció a $H \times W \times Ch$ sorrend használata, a Deep Learning könyvtárak viszont a $Ch \times H \times W$ sorrendet preferálják. Erre megoldás a NumPy transpose függvény.

```
myArr.shape # [N,3]
np.transpose(myArr).shape # [3,N]

# Több dimenzióban
myImg.shape # [1080,1920,3]
np.transpose(myImg,(2,0,1)).shape # [3,1080,1920]
```

Ezen felül érdemes még említeni, hogy a NumPy képes különböző lineáris algebrai műveletek elvégzésére, mint például a mátrixszorzás (amennyiben a dimenziók megfelelőek), valamint az invertálás:

```
myMtx = np.array  ([[1,2,3],[4,5,6],[7,8,9]])
myInv = np.linalg.inv(myMtx)

myData = np.randn((50,3)) # 50 3D coordinates
myTrData = np.matMul(myMtx,myData) # myTrData = myMtx*myData
```

Fontos megemlíteni, hogy a *numpy.ndarray* típuson elvégezhető műveletek nagy része pontosan ugyanúgy működik *torch.tensor* típuson is, bár esetenként ugyanaz a függvény más néven érhető el. A *numpy.transpose* például *torch.permute*.

3 Mérési feladatok

A mérés folyamán az alábbi feladatokat kell elvégezni:

1. Egészítse ki a mellékelt notebookban megadott adatbázis beolvasót adataugmentációs eljárásokkal.
2. Implementálja le az EfficientNet architektúra alapvető konvolúciós blokkját, és az asszimetrikus reziduális blokkját.
3. Készítsen eljárást egyszerű neurális háló tanítására a méréshez használt adatbázison.
4. Végezzen el különböző tanításokat a 3.1. Táblázatban megadott hiperparaméterek alapján, értékelje ezek eredményeit.

epochs	lr_decay_epochs	batch_size	lr	lr_decay	weight_decay
20	10	8	5e-4	0.1	2e-4
20	10	8	1e-1	0.1	2e-4
20	10	8	5e-4	0.001	2e-4
20	10	8	5e-4	0.1	1e-1
20	10	2	5e-4	0.1	2e-4
100	50	8	5e-4	0.1	2e-4

3.1. táblázat. *A Vizsgálandó hiperparaméter kombinációk.*

4 Hasznos kódrészletek

Érték elérése az argumentumokból

```
myVal = args["arg_name"]
```

Tenzor tükrözése a vízszintes tengely mentén (utolsó dimenzió)

```
tensor = tensor.flip(-1)
```

Transzformációk kompozíciója

```
image_transform = transforms.Compose(
    [transform1,
    transform2]
)
```

Átméretezés transzformáció

```
transforms.Resize((height, width), INTER_MODE)
#Interpolation can be Image.LINEAR or Image.NEAREST
```

Tenzor konverzió

```
# To float tensor (for images)
transforms.ToTensor()
# To long tensor (for labels)
ext_transforms.PILToLongTensor()
```

Dataloader létrehozása

```
dataLoader = torch.utils.data.DataLoader(trainSet, batch_size=..., shuffle=True/False,
    num_workers=...)
```

Neurális háló módjainak állítása

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

Optimizer kinullázása

```
optimizer.zero_grad()
```

Forward

```
outputs = net(inputs)
```

Költség

```
loss = criterion(outputs, labels)
```

Backward

```
loss.backward()
```

Egy lépés gradiens módszerrel

```
optimizer.step()
```

Gradiens számítás megelőzése

```
with torch.no_grad():
```

Különböző rétegek létrehozása PyTorch környezeteben

```
self.conv = nn.Conv2d(...)
self.bn = nn.BatchNorm2d(...)
```

Rétegek kombinálása

```
self.combined = nn.Sequential(layer_1,layer_2,)
```

Rétegek meghívása

```
out = self.layer_name(in)
```

Tenzor/Modell átvitele a megfelelő eszközre (GPU/CPU)

```
model = model.to(self.device)
```

Költségfüggvény létrehozása

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

Optimalizáló módszer létrehozás

```
optimizer = optim.Adam(model.parameters(), lr=..., weight_decay=...)
```

Tanulási ráta ütemező létrehozása

```
lr_updater = lr_scheduler.StepLR(optimizer, <lr_decay_epochs>, <lr_decay>)
```

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. Milyen módszerekkel kerülhető el az overfitting? Ismertesse ezeket egy-egy mondatban!
6. Milyen módszerekkel javítható egy neurális háló konvergenciája? Ismertesse ezeket egy-egy mondatban!
7. Ismertesse röviden a PyTorch könyvtárat! Milyen alapvető adattípusai, moduljai léteznek? Milyen szolgáltatásokat nyújt neurális hálók számára?
8. 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.
9. Hogyan néz ki egyetlen tanulási ciklus PyTorch-ban? (Pszeudó kód elég)

Irodalom

[1] M. Szemenyei, *Számítógépes Látórendszerek*, M. Szemenyei, szerk. BME, 2019. cím: http://deeplearning.iit.bme.hu/jegyzetFull.pdf.

[2] M. Tan és Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", *International Conference on Machine Learning, 2019*, 2019. máj. 28. arXiv: 1905.11946v4 [cs.LG].

[3] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard és Q. V. Le, "MnasNet: Platform-Aware Neural Architecture Search for Mobile", *CVPR 2019*, 2018. júl. 31. arXiv: 1807.11626v3 [cs.CV].

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

[5] *Anaconda*. cím: https://www.anaconda.com/.

[6] *PyTorch Documentation*. cím: https://pytorch.org/docs/stable/index.html.

[7] *PyTorch Tutorials*. cím: https://pytorch.org/tutorials/index.html.