

TDK DOLGOZAT



HAT SZABADSÁGFOKÚ PÓZ MEGHATÁROZÁS IPARI ROBOTOK VEZÉRLÉSÉRE

Borján Katalin

2022

DUNAÚJVÁROSI EGYETEM

Informatikai Intézet

Konzulensek:

Burkus Ervin

Egyetemi tanársegéd

Dr. Király Zoltán

Egyetemi docens

Készítette:

Borján Katalin

Mérnök informatikus hallgató

KIVONAT

A dolgozatomban részletesen prezentálok egy általam fejlesztett, ipari robotkarok vezérlésére alkalmas, gépi látáson alapuló rendszert, amely példány szegmentációs maszk segítségével képes egy adott célobjektum hat szabadságfokú pózának (pozíció és orientáció) meghatározására. A gépi látással történő példány szegmentációt Pytorch keretrendszerben valósítottam meg, YOLACT konvolúciós neurális háló modellel segítségével. A modell tanításához szükséges adatbázist szintetikusán állítottam elő a nyílt forráskódú Blender programban, ami háromdimenziós grafikák, animációk elkészítésére szolgál, illetve valósághű megjelenés érhető el vele képalkotás terén. A képek generálásának automatizálásáról Python script gondoskodik ezzel felgyorsítva az adatbázis elkészítésének folyamatát, így áthidalva a manuális adatgyűjtéssel és címkézéssel járó problémákat és nehézségeket. A neurális háló által szolgáltatott objektum maszk adatokon Hough geometriai transzformáció alkalmazásával állapítottam meg a szükséges kulcspontok koordinátáit a kétdimenziós térben. Ezen pontok alapján Perspective-n-Point algoritmussal került kiszámításra a translációs és a rotációs mátrix és valósult meg az objektum póz meghatározása. A számításokat az OpenCV függvénykönyvtárának segítségével, Python programban végeztem el. A dolgozat bemutatja a szintetikus adatbázis elkészítésének folyamatát, a szintetikus adatbázissal történő tanítás eredményeit, a póz meghatározás folyamatának matematikai hátterét, illetve a megvalósítás folyamán felmerülő problémákat és azok megoldásait.

TARTALOMJEGYZÉK

1.	BEVEZETÉS	1
1.1	Célkitűzés	2
1.2	A dolgozat felépítése	3
2.	Neurális hálók	4
3.	Szintetikus adatbázis készítése	6
3.1	Háromdimenziós jelenet felépítése	6
3.2	Háromdimenziós jelenet módosításának automatizálása	8
3.2.1	A célobjektumot kezelő metódusok	8
3.2.2	A zavaróobjektumokat módosító metódusok	9
3.2.3	Háttérrel módosító metódus	10
3.3	Ciklusok	11
3.4	COCO adatbázis formátum	15
4.	YOLACT modell	18
4.1	YOLACT modell tanítása	18
4.2	Validációs adatok kiértékelése és eredmények	19
5.	Kulcsponatok kinyerése	22
5.1.1	Konvex hull keresésével elért eredmények	22
5.1.2	Kulcsponatok keresése egyenesek illesztésével	22
6.	PnP algoritmus	25
7.	Összefoglaló	30
	Irodalomjegyzék	32
	Ábrajegyzék	35

1. BEVEZETÉS

A gépi gondolkodást minden szempontból, számos területen komoly érdeklődés övezi, attól a pillanattól kezdve, hogy a mai számítógépek elődei elvégezték első számításaikat. A már akkor emberfeletti teljesítmény természetesen megmozgatta az emberek fantáziáját, felmerült több filozófiai kérdés is a számítógépek logikájával kapcsolatban és mivel ez merőben új területnek számított elkerülhetetlen volt, hogy feltegyük a kérdést, vajon értelmet teremtettünk? Ahogy haladtunk előre az időben annál inkább nyilvánvalóvá vált, hogy ilyesmiről nincs szó, de a mai napig vannak viták az intelligens rendszerek kapcsán. Felmerült, hogy a mesterséges intelligenciával rendelkező robotokat, szoftvereket megilletik-e szerzői jogok műalkotások önálló létrehozásából adódóan és már sor került az első robot állampolgár, Sophia nyilvántartásba vételére [1]. Komolyan megosztja az emberek véleményét ez a téma. Van, aki a jövőt látja benne és van, aki egyenesen a világvégétől tart. Ez pedig kimeríthetetlen alapanyagul szolgál kedvenc science fiction irodalmi –, filmművészeti alkotóink számára, de kiapadhatatlan motivációt nyújt mérnökeinknek is, akik jelenleg is küzdenek a kihívással, hogy emberi lényhez minél inkább hasonlóan gondolkozó és funkcionáló robotot hozzanak létre. Habár jelenleg ez sajnos továbbra is egy távoli, emberi léptékben elérhetetlennek tűnő célkitűzés, de ezzel párhuzamosan az egy-egy specifikus feladatra kiélezett megoldások is megvetették a lábukat a mesterséges intelligencia tudományágának talaján. Ez utóbbi terület az elmúlt évtizedben komoly virágzásnak indult a technikai fejlődésnek és a felhalmozott adattömegekben rejlő értékek felismerésének köszönhetően. Ennek eredménye képpen kinyíltak a lehetőségek az iparban is. Nyilvánvalóvá vált, hogy az önálló döntéshozó rendszerekre hatalmas a kereslet. Az emberek sok területen szeretnék kiküszöbölni az emberi tényezőt a kényelem, a pontosság, a biztonság érdekében. Ez volt a motiváló tényező számomra is a hat szabadságfokú póz meghatározó rendszer elkészítése során.

A célom az volt, hogy egy adott objektumról készült kétdimenziós képből kinyerjem a translációs- és rotációs mátrixot, melyek alapján meghatározható a kétdimenziós kulcspontok három dimenzióban elhelyezkedő megfelelője. Az eredményt ipari felhasználásra, egy robotkar vezérlésére szántam.

1.1 Célkitűzés

A gépi látás nagyon fontos szerepet játszik az ipari robotikában. Széleskörű elterjedése párhuzamba vonható a kameratechnológiák rohamos fejlődésével. A gépi látás segítségével a robotok képesek vizualizálni tevékenységüket. Ez lehetőséget teremt arra, hogy a meghatározott feladatkörükön belül az önálló döntéshozatal képességének segítségével rugalmas és sokoldalú funkciókat lássanak el, alkalmazkodjanak környezetükhöz. Ennek hiányában a robotok „vakok”, és csak ugyanazokat az előre beprogramozott feladatokat képesek folyamatosan ismételni, szigorúan meghatározott körülmények között. [2]

A legfontosabb problémák, amikre a gépi látás képes megoldást szolgáltatni az akadálykerülés, valamint célobjektumok pózának meghatározása. A 3D objektumdetektálást és a 6D (6 szabadságfokú) póz esztimációt jelentős érdeklődés övezi kutatói és mérnöki körökben, amit a nagy számítási teljesítményű hardverek megjelenése, rohamos fejlődése csak fokozott. A leggyakrabban alkalmazott, mély neurális hálózat alapú 6D póz esztimáció betanításához óriási, felcímkézett képi adatbázisok szükségesek. Ez manuális adatgyűjtés során problémát jelent, szintetikus adatok használatával viszont szinte korlátlan mennyiségű képet tudunk létrehozni, automatizált címkézéssel. A mély neurális hálózatokon alapuló 6D póz esztimációra egyre jobb eljárásokat fejlesztenek. Ezek közül érdemes kiemelni a PoseCNN és a DOPE (Deep Object Pose Estimation) megvalósításokat. [3] [4]

A dolgozatban bemutatott eljárás egy konkrét ipari problémát szeretne megoldani. A cél egy elektromos autó töltő csatlakozójának felismerése és pózának meghatározása. Ez a feladat egy, az egyetemen végzett kutatás részfeladata, melynek megoldására több eljárás is ki lett illetve ki lesz próbálva, többek között a DOPE póz esztimációs eljárás is. A kész eljárásoknak azonban mindig van hátránya is. Az eddig létrejött 6 DOF póz esztimációs eljárások univerzális megoldásoknak számítanak és nehéz őket áttekinteni. A saját megoldásommal egy specifikus, cél orientált és az adott feladathoz illeszkedő megvalósítást szerettem volna létrehozni. Ez alatt főleg arra gondolok, hogy a modell előállítását és a kulcspontok meghatározását a célobjektum jellegzetességeihez lehet igazítani. Ez összetettebbé teszi a szintetikus adatgenerálást és a kulcspontok meghatározását.

A saját megoldásom a 6 szabadságfokú póz meghatározását szintetikus adat létrehozás és címkézés, neurális hálózaton alapuló példány szegmentáció, kulcspont jellemző kinyerés és

Perspektíva-n-Pont esztimáció használatával végzi. A megoldás elsődlegesen a töltőcsatlakozó esetére lett kifejlesztve, de elképzeléseim szerint bármilyen más célobjektum felismerésére is alkalmazható az adott lépések finomhangolásával.

A fejlesztés céljai 4 fő pontba sorolhatók:

- Konvolúciós neurális háló tanítására alkalmas adatbázis létrehozása szintetikus úton.
- Neurális háló tanítása, egy adott objektum detektálására és szegmentálására
- Neurális háló által szolgáltatott képadaton kulcspontok megállapítása
- Kétdimenziós kulcspontok alapján translációs és rotációs mátrix kiszámítása

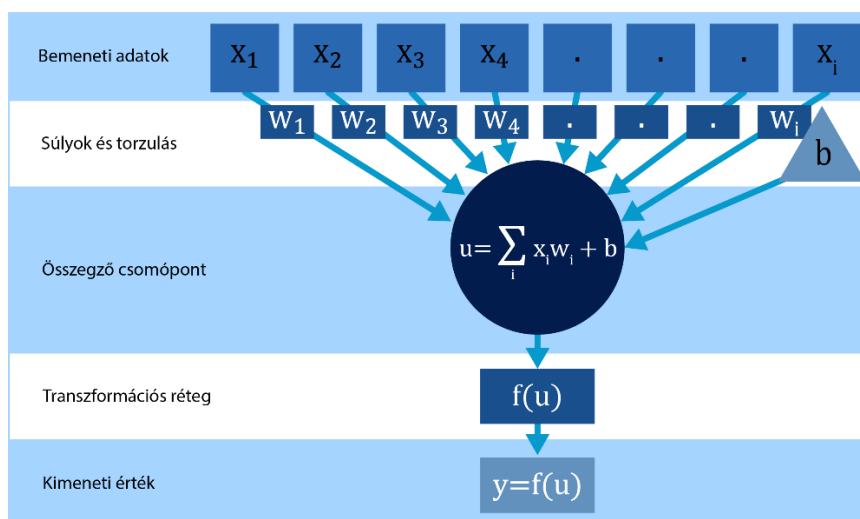
1.2 A dolgozat felépítése

1. Az első fejezetben bemutatom a neurális hálókat
2. A második fejezetben bemutatom a szintetikus adatbázis generálásának folyamatát és annak tanítása során szerzett tapasztalatokat
3. A harmadik fejezetben részletezem a kulcspontok kinyerésre tett kísérleteket a felmerülő problémákat és azok megoldásait
4. A negyedik fejezetben ismertetem az algoritmust, amely segítségemre volt a translációs és rotációs mátrix kiszámításában
5. Az ötödik fejezetben összefoglalom a fejlesztés eredményeit, konklúzióit.

2. Neurális hálók

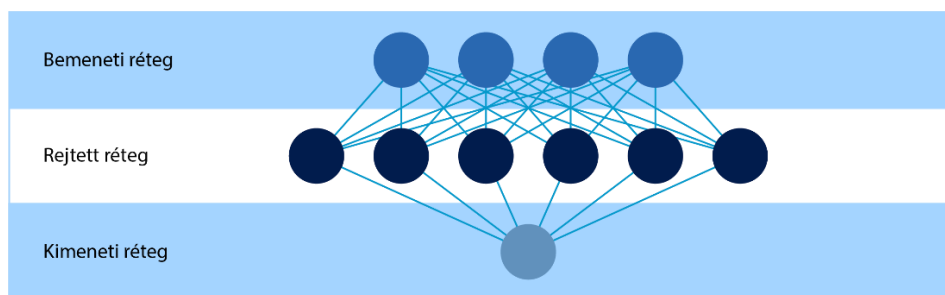
A neurális hálók olyan mesterséges intelligens rendszerek melyek működését a biológiai agy és idegrendszer dinamikája ihlette. Ezek a hálózatok a hagyományos algoritmikus számítási rendszerekhez képest jobbnak bizonyultak különféle felismerési feladatokban, például kézírás-, képfelismerés, alakzatdetektálás, ezzel biztosítva számunkra a gépi látás széleskörűen felhasználható technológiájának alapjait. [5]

Erre a neurális hálót alkotó processzáló elemek sokasága nyújt lehetőséget. Ezeket neuronoknak nevezzük a biológiai megfelelőjükhöz hasonlóan. Az 1. ábrán megfigyelhetjük részletes felépítésüket.



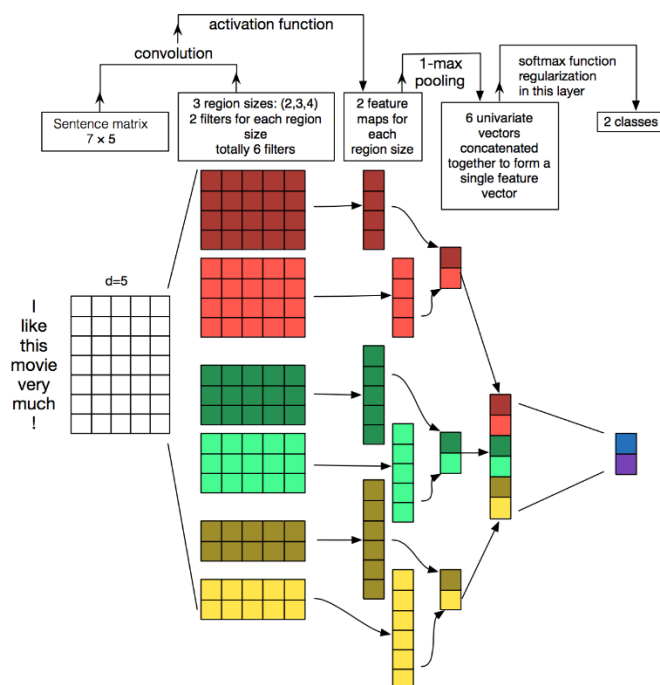
1. Elemi neuron felépítése (saját ábra)

Habár egy neuron önmagában alkotja a lehető legkisebb neurális hálót, jellemzően nagyobb, többrétegű modelleket használunk. Egy-egy réteg szerepét a hálóban elfoglalt helye határozza meg. Minden esetben van egy bemeneti és egy kimeneti réteg. Ezek a rétegek a külvilággal való kapcsolatról gondoskodnak. Amíg a bemeneti réteg felveszi a bejövő adatokat a kimeneti réteg szolgáltatja számunkra az eredményeket. Ahogy a 2. ábrán is látható, a két réteg között találhatók a rejtett rétegek, ebből tetszőleges számú lehet egy hálóban. A több rejtett réteggel rendelkező neurális hálókat mély neurális hálóknak nevezzük. [5]



2. Neurális háló felépítése (saját ábra)

A bemeneti információk a rejtett rétegeken áthaladva egyre elvontabb formát vesznek fel, minden rétegben más jellegűek az információk. Ennek köszönhetően a folyamat során megtanulja magától a rendszer a szükséges összefüggéseket, nincs szükség előzetesen a jellemzők definiálására. Ezeknél a modelleknél azonban a több rejtett réteg miatt jelentősen megnövekszik a tanulandó súlyok száma, annak érdekében, hogy az ebből eredő hosszú tanulási idő ne okozzon problémát, egy tömörítési eljárással úgynevezett konvolúciók alkalmazásával egyszerűsítik a folyamatot. Ezt a fajta mozgó ablakos megoldással működő gépi tanuló rendszert nevezzük konvolúciós neurális hálónak. [6] [5]



3. Konvolúciós neurális háló felépítése [7]

A mély neurális hálók egyéni tanulása fellendítette az algoritmikusan nehezen leírható, bonyolultan meghatározható feladatok területét, azonban mivel kizárólag nyers bemeneti

adatok állnak rendelkezésre a tanuláshoz, azért nagy mennyiségű adatra van szükség. Számításba kell venni, hogy a neurális háló kizárólag arra képes, amire betanítjuk és az önálló tanulás során kialakult szabályszerűségek, összefüggések nem értelmezhetők ember számára. Így különös figyelmet igényel a helyes adatbázis megtervezése, elkészítése. [6]

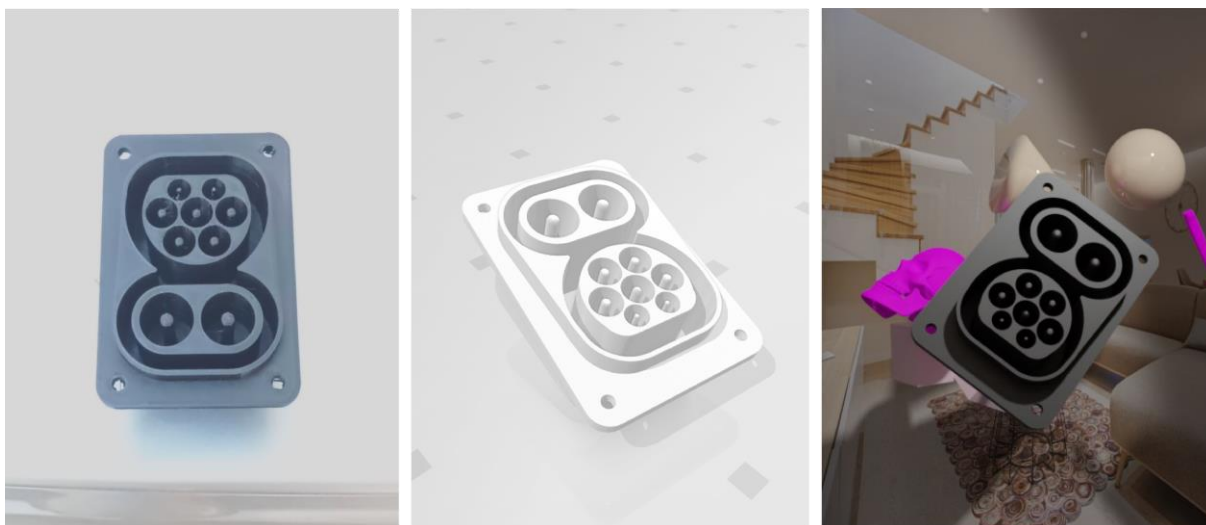
3. Szintetikus adatbázis készítése

A neurális hálók nagy adatigénye miatt a gépi tanulás fejlesztésének a folyamatát nagy részben rendszerint az adatbázis elkészítése teszi ki. Gépi látás modellek esetén több ezer kép elkészítése-összegyűjtése, az adatok címkézése hosszadalmas feladat és egy fejlesztési folyamatot évekkel visszavethet. Így gyakran hiába nagy pontosságot igénylő feladatról van szó, szükségszerű ezt feláldozni az idő oltárán.

Az ebből fakadó problémák kiküszöbölése volt a cél a szintetikus adatbázis alkalmazásával. A szintetikus adatbázis jelen esetben az olyan kép és címkegyűjteményt jelenti, amely a manuális fényképgyűjtéstől és kézzel történő címkézéstől eltérően, valamilyen mesterséges úton került létrehozásra. Az elkészítéséhez a Blender nyílt forráskódú grafikai programot választottam, amelyet széleskörűen háromdimenziós grafikák, animációk elkészítésére használnak. A választásom fő oka a beépített Python fejlesztő felület melyen keresztül script segítségével automatizálhatók a szükséges módosítások

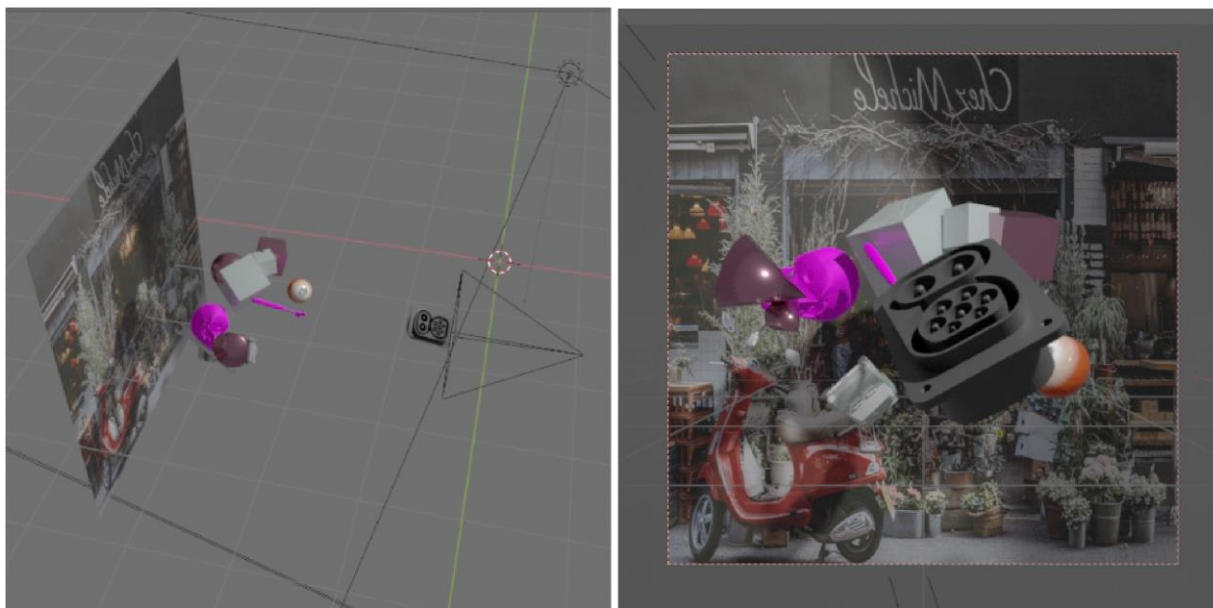
3.1 Háromdimenziós jelenet felépítése

Egy a tanításhoz megfelelő kép rendereléséhez több fontos tényezőnek kell szerepelnie a háromdimenziós jelenetben. Az legfontosabb maga az objektum, aminek a detektálására és szegmentálására szeretnénk tanítani a neurális hálót. Esetünkben ez egy elektromos autó töltőcsatlakozója, amit .STL fájl formátumban importáltam és elláttam egy fekete műanyagot imitáló textúrával, hogy megjelenésében hasonlítson az eredeti tárgyra. A 4. ábrán látható egy fénykép a csatlakozóról, az .STL fájl előnézete, és egy renderelt kép a textúrával ellátott csatlakozóról.



4, Fénykép a csatlakozóról, 3D modell, textúrált renderkép (saját ábra)

A célobjektumnak meg kell teremteni ezen kívül egy környezetet, amit folyamatosan módosítva változatos képsokaságot kaphatunk eredményül. A renderelési idő csökkentésének érdekében a háttérrel kétdimenziós képek gondoskodtak egy síkidom textúrájaként. Ez még önmagában azt jelentené, hogy minden egyes renderelt kép esetén másik háttérkép kell, azaz így is szükségünk van több ezer egyedi kép összegyűjtésére. Ennek kiküszöbölésére elhelyeztem számos zavaró objektumot a jelenetben, ezek a célobjektumtól eltérő háromdimenziós testek, egyszerű geometriai objektumok és internetről származó szabad felhasználású objektum fájlok. Elhelyezésükkel egy olyan összetett hátteret hozunk létre, amelynek az elemei módosításával, nagyon nagy számú kép generálása esetén is változatos adatokat eredményeznek. Ezen felül létrehoztam három különböző textúrát a geometriai objektumok számára, egy matt felületet, egy fényes felületet és egy fényes metál felületet.



5. A jelenet látszata és az erről a kamera által leképezett látvány (saját ábra)

3.2 Háromdimenziós jelenet módosításának automatizálása

A jelenet automatizálásához a Blender beépített Python kezelőfelületét használtam. Először a korábbi fejezetben felsorolt elemeket, lista formájában inicializáltam. A külön listák az alapján alakultak, hogy a bennük felsorolt objektumokon milyen műveleteket lesz szükséges elvégezni.

```
objects = ['Connector']
scene_objs = ['Car', 'Skull', 'Eye', 'Hammer', 'Wrench']
scene_objects =
['Sphere', 'Cone', 'Cube', 'Sphere.001', 'Cone.001', 'Cube.001', 'Sphere.002', 'Co
ne.002', 'Cube.002']
obj_materials = ['Matte', 'Metal', 'Glossy']
```

3.2.1 A célobjektumot kezelő metódusok

Az „objects” lista tartalmazza a célobjektumot. Ez az az objektum, aminek a pozíciója, és a rotációja egyedi módon kerül megadásra. Ennek az oka az, hogy ez egy háromdimenziós objektum, azonban a gyakorlatban később nem fog előfordulni, hogy a csatlakozó hátulját kell detektálnunk. Így ha erről a területről is készülnének felvételek, felesleges információkra tanítanánk a neurális hálót a későbbiekben, ez pedig ronthat a modell hatékonyságán. A pozícióját tekintve ez az objektum előtérben fog elhelyezkedni a zavaró objektumokhoz képest

így azt is egyedileg meghatározott módon kell módosítani. A fókuszobjektum textúrája állandó, így az későbbi módosításra nem szorul.

```
def change_focusobj_location():  
    bpy.data.objects['Connector'].location = Vector((random.uniform(0.65,  
1.65),random.uniform(1.5, 2.5),random.random()))
```

A fenti kódrészletben látható a metódus, ami a célobjektum pozícióját random módosíthatja az X tengelyen 0.65m és 1.65m között, az Y tengelyen 1.5m és 2.5m között, a Z tengelyen 0.0m és 1.0m között.

```
def change_focusobj_rotation():  
    random_rot = (random.uniform(6.108653,6.457718),  
random.uniform(6.108653,6.457718),random.uniform(1.396263,1.745329) )  
    bpy.data.objects['Connector'].rotation_euler = Euler(random_rot, 'XYZ')
```

A fenti kódrészletben látható a metódus, ami a célobjektum rotációját random módosíthatja az X tengelyen 350° és 370° között, az Y tengelyen 350° és 370° között, a Z tengelyen pedig szintén 350° és 370° között.

3.2.2 A zavaróobjektumokat módosító metódusok

A zavaróobjektumok a „scene_objs” és a „scene_objects” listában kerültek felsorolásra. A „scene_objs” listában szereplő elemek a <https://free3d.com/> weblapról származó, szabad felhasználású, kész .OBJ fájlok, a „scene_objects” pedig a Blenderen belül létrehozott egyszerű testeket tartalmazza. A kettő között a különbség csupán annyi, hogy a külső forrásból származó listán nem kerül módosításra a textúra. A textúrák módosításához hoztam létre az „obj_materials” listát, amely a korábban létrehozott felületeket tartalmazza.

```
def change_sceneobj_location(object):  
    bpy.data.objects[object].location = Vector((random.uniform(2.0,  
3.0),random.uniform(1.0, 3.0),random.random()))
```

A fent látható kódrészlet mindkét listán alkalmazásra kerül és egy random pozíciót határoz meg az X tengelyen 2 m és 3m között, az Y tengelyen 1m és 3m között, a Z tengelyen 0.0m és 1.0m között egy paraméterként átadott objektumnak. Az X tengely határértékeit megfigyelve

láthatjuk, hogy az nem esik egybe a célobjektum határértékeivel, a zavaróobjektumok minden esetben a célobjektum mögött helyezkednek el.

```
def change_sceneobj_rotation(object):  
    random_rot = (random.uniform(0.0,6.283185),  
random.uniform(0.0,6.283185),random.uniform(0.0,6.283185) )  
    bpy.data.objects[object].rotation_euler = Euler(random_rot,  
'XYZ')
```

Ez a módszer a paraméterként átadott objektum random rotációjáról gondoskodik, minden tengelyen 0°-tól 360°-ig mozgó skálán.

```
def change_obj_materials(object):  
    bpy.data.objects[object].data.materials[0] =  
bpy.data.materials[obj_materials[random.randrange(3)]]
```

Ezt a módszert kizárólag az egyszerű geometriai testeken hajtjuk végre, ezzel a textúra listából átadva a paraméterként kapott objektumnak egy véletlenszerű felületet. A textúrák alapszínét az alábbi módszerrel szintén randomizálva módosítjuk.

```
def change_material_color(obj_material):  
    color = Color()  
    hue = random.random()  
    sat = random.random()  
    val = random.random()  
    color.hsv = (hue, sat, val)  
    rgba = [color.r, color.g, color.b, 1]  
    obj_material.node_tree.nodes['Principled BSDF'].inputs[0].default_value  
= rgba
```

3.2.3 Háttérrel módosító módszer

A hátterek esetében gyűjtöttem össze fotókat .jpg kiterjesztésben, összesen száz darabot, majd ezeket elhelyeztem egy könyvtárban. A képek fájlnevei 0-99 ig terjedő számok, ezek alapján hivatkozunk rájuk a kódban.

```
def change_background():  
    bgrnd_material = random.randrange(100);  
    bpy.data.images.load(f"C:/Users/Public/data_generator/backgrounds/{bgrnd_ma  
terial}.jpg", check_existing=True)  
    image_node =  
    bpy.data.objects['Background'].data.materials[0].node_tree.nodes["Image  
Texture"]  
    image_node.image = bpy.data.images[f'{bgrnd_material}.jpg']
```

Ezen kódrészlet során, egy 0-100 közötti randomszám alapján betöltésre kerül egy fénykép, amit képtextúráként beállítunk a „Background” objektum anyagtulajdonságai között.

3.3 Ciklusok

A ciklusok megvalósításához az alábbi változókat volt szükséges deklarálni:

```
obj_renders_per_split = [('train', 5000), ('val', 500), ('test', 0)]  
total_render_count = sum([len(objects) * r[1] for r in  
obj_renders_per_split])  
start_idx = 0
```

Az „object_renders_per_split” változó módosításával megadhatjuk, hogy mely célra szánt képből, hány darabot szeretnénk elkészíteni. A tanítás során szükségünk van a tanuló adatbázistól független, validációs adatokra, amivel tesztelhetjük a betanult súlyokat, hogy elkerülhessük a túltanulás jelenségét. Validációs adatbázis gyanánt elég, ha az adatbázis 10-20%-át leválasztjuk, azonban mindenképpen különböznie kell a tanító adatoktól. A „total_render_count” változó összesíti a renderelni kívánt képmennyiséget.

```
for split_name, renders_per_object in obj_renders_per_split:
    print(f'Starting split : {split_name} | Total renders:
{renders_per_object*len(objects)}')
    for object in objects:
        print(f'Starting object: {split_name}/{object}')
        for i in range(start_idx, start_idx + renders_per_object):
            change_background()
            change_focusobj_rotation()
            change_focusobj_location()
            for s_object in scene_objs :
                change_sceneobj_rotation(s_object)
                change_sceneobj_location(s_object)
            for s_obj in scene_objects :
                change_sceneobj_rotation(s_obj)
                change_sceneobj_location(s_obj)
                change_obj_materials(s_obj)

change_material_color(bpy.data.objects[s_obj].data.materials[0])
```

A ciklus a fent megadott „object_renders_per_split” alapján ismétlődik aszerint, hogy milyen értékeket adtunk meg neki korábban. Jelen esetben ötezer és ötszáz kép elkészítésére, van beállítva, a ciklus ötezer-ötszázszor fog lefutni. Következőkben bejárjuk az „objects” listát, jelen esetben ez a lista egyetlen elemet tartalmaz, így nincs jelentősége, azonban, ha több célobjektumot szeretnénk tanítani a neurális hálónak, nem lenne szükség módosításra, elegendő csupán felvenni a további elemeket a listába. A következő cikluson belül kerülnek meghívásra a metódusok, amelyeket a korábbiakban definiáltunk. Leváltásra kerül a háttér, majd a fókuszobjektum rotációja és lokációja. Majd a „scene_objs” lista bejárása során módosításra kerül a külső forrásból származó zavaróobjektumok térben foglalt helyzete. Ezt követően kerül sor a geometriai formák módosítására, szintén lokáció és rotáció, továbbá a textúra és annak a színének tekintetében.

A módosításokat követően készen áll a jelenet egy új kép létrehozására. Ehhez a programban beállítottam a képméretet $512\text{px} \times 512\text{px}$ értékre. Ez a méret ideális a neurális háló tanításához. Megfelelő mennyiségű információval tud szolgálni a tanulás során, azonban még nem túl nagy méret ahhoz, hogy a renderelési- és a tanítási időt jócskán visszavesse.

A Blenderen belül több renderelő motort is alkalmazhatunk. Én ezek közül a Cycles-t választottam. Ezt arra tervezték, hogy fizikailag hiteles, valóság-hű eredményeket szolgáltatson [8]. Illetve átadható számára grafikus videokártya a számítások elvégzésére, ez pedig jelentősen felgyorsítja a képalkotás folyamatát.

```
bpy.context.scene.render.filepath = str(output_path / split_name / object /  
f'{str(i).zfill(6)}.jpg')  
bpy.ops.render.render(write_still=True)
```

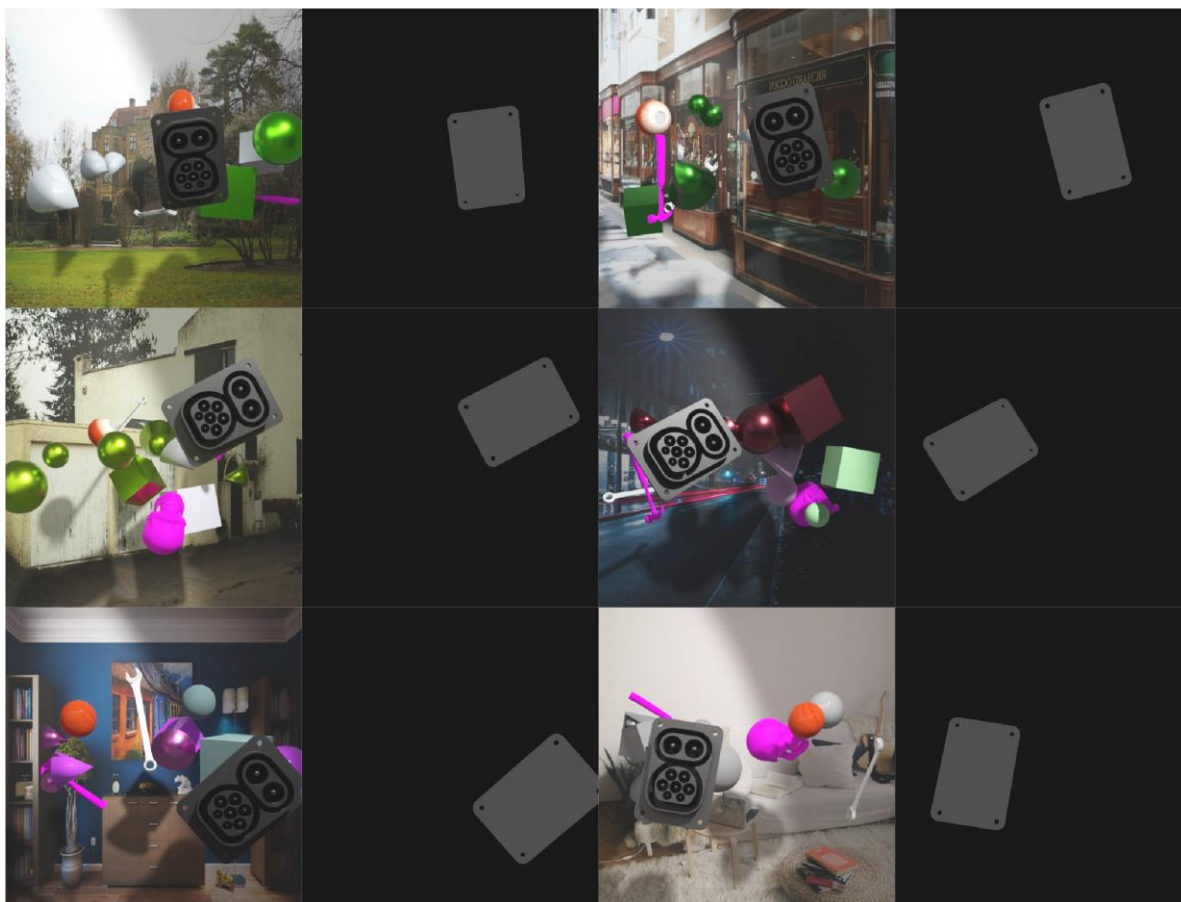
A fenti kódrészlet gondoskodik az RGB kép rendereléséről és egy megadott útvonalra való kiírásáról.

Ahhoz, hogy a neurális hálót példány szegmentációra tanítsuk, szükségünk van egy képre, illetve egy ahhoz tartozó maszkra. A maszk hordozza a neurális háló számára azt az információt, hogy egy adott kép esetén, mely képpontokon található meg a célobjektum, így a tanulás során ezekről a területekről gyűjtheti a jellemzőket, amik alapján majd a modell is képes lesz visszaadni egy szegmentációs maszkot új, számára ismeretlen képanyagokról.

A szegmentációs maszk kinyeréséhez minden RGB képnek elkészítettem egy olyan változatát, amelyen a célobjektum homogén formában látható, és csakis kizárólag a célobjektum látható.

```
counter=1
for obj in bpy.data.objects:
    if obj.type == "MESH":
        obj = bpy.context.active_object
        obj["inst_id"] = counter
        counter=counter+1
result = bpy.cv.render_data()
cv2.imwrite(f"C:/Users/Public/data_generator/simple_object_segmentation/{split_name}/{object}/Segmentation/{str(i).zfill(6)}.png", result["inst"])
```

A fent látható kódrészlet végzi el egy példány szegmentációs kép készítését. Ahhoz hogy minden alkalommal ugyanaz a homogén szín tartozzon egy objektumhoz, egyedi „inst_id”-t kell meghatároznunk a célobjektumnak. A fenti kódrészletben az egyedi „inst_id” az aktív objektumokhoz kerül hozzárendelésre. Ezt követően a „result” változónak átadjuk a renderelési adatokat. Ez tartalmazza az RGB kép és a szegmentációs adatokat is, ezért egy OpenCV `imwrite()` függvénnyel kizárólag a szegmentációs adatokat írjuk ki .PNG fájlformátumba. A 6. ábrán láthatóak az eredmények, a renderelt kép és a hozzá tartozó szegmentációs maszk.



6. RGB képek és a szegmentált megfelelőjük (saját ábra)

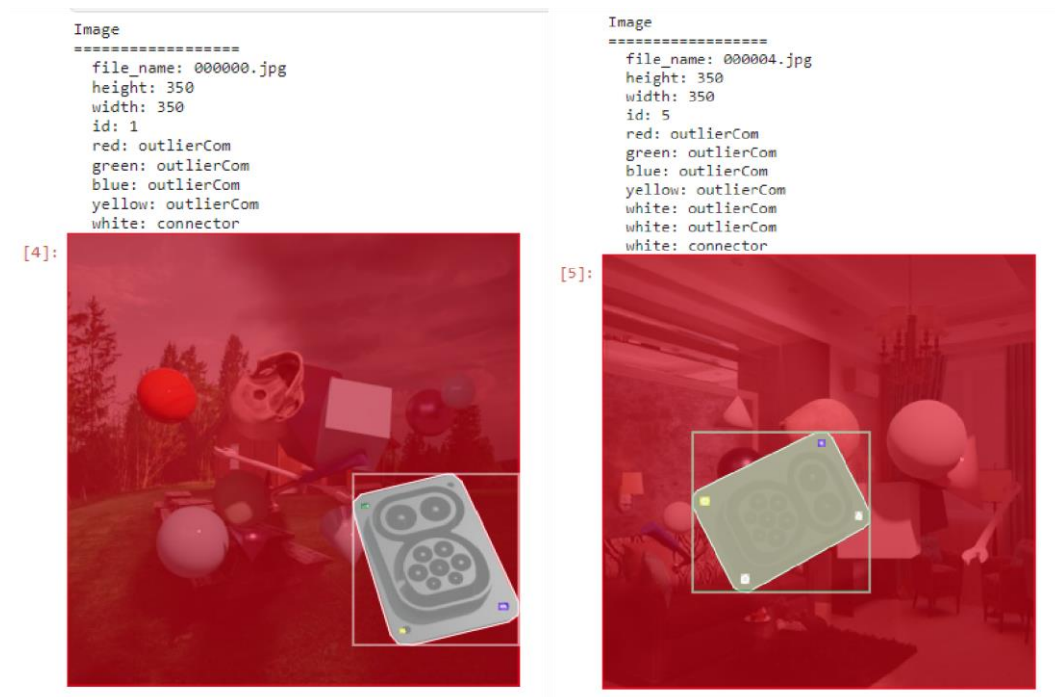
3.4 COCO adatbázis formátum

A Microsoft COCO (Common Objects in Context) megalkotásának célja, egy széleskörűen alkalmazható nyilvános adatbázis létrehozása volt, melyben a leggyakoribb objektumok találhatóak meg. Ezzel szerették volna elősegíteni az objektumfelismerés fejlődésének fellendülését, levéve az adatbázis elkészítésének terhét a kutatók és az adattudósok válláról. Az adatbázis 91 objektumtípus 328 000 képét tartalmazza, ez pedig összesen 2.5 millió címke példányt. [9]

Az adatbázis akkora népszerűségnek örvend, hogy a formátuma, általános sablonként szolgál a legtöbb objektumfelismerésre alkalmas modell tanításához készült adatbázishoz. Az általam kiválasztott YOLACT (példányszegmentációhoz készült, konvolúciós neurális háló modell) is ezt alkalmazza, így az elkészített adatbázist is át kellett alakítani. Erre a célra egy publikus, Github-on elérhető programot használtam [10]. A program az átadott képfájlokat

feldolgozza, a szegmentált felületet poligonná alakítja, majd annak a körvonalát numpy tömbként rögzíti egy .json fájlba minden szükséges egyéb adattal együtt.

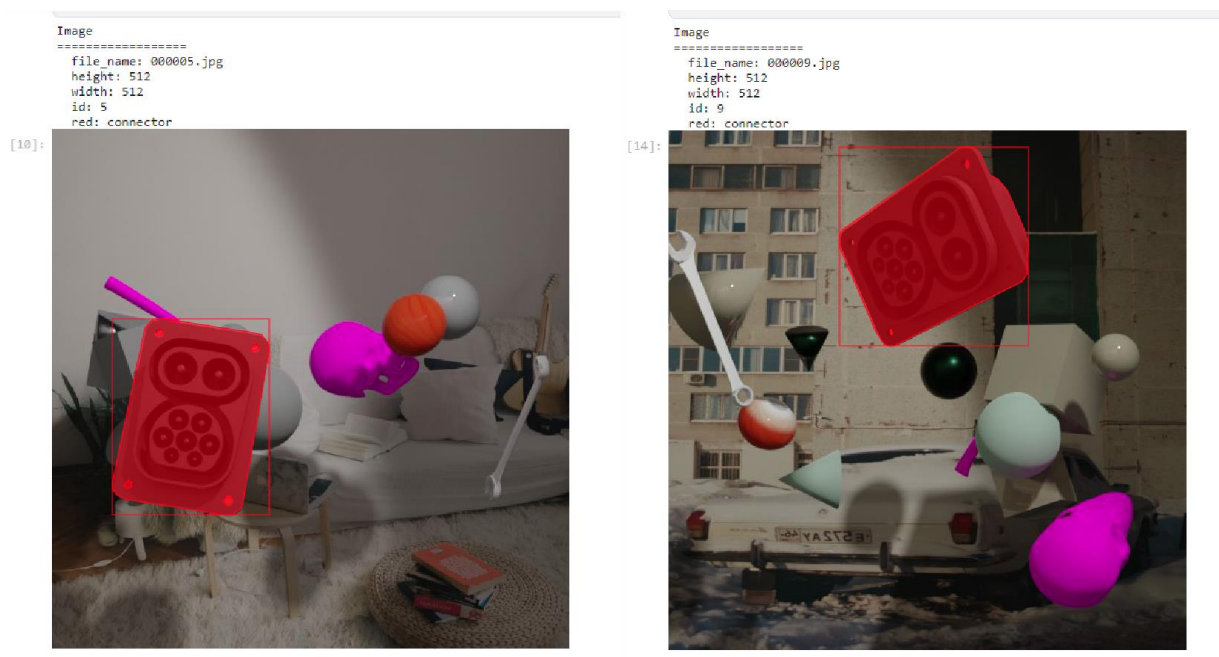
Az első futtatást követően, egy online elérhető szegmentációs adatbázist vizualizálni képes eszközzel ellenőriztem az eredményeket. [11] Ennek során fény derült néhány problémára.



7. COCO formátumra konvertálás felmerülő problémái (saját ábra)

A 7. ábrán jól látható, hogy az annotációk között nem kizárólag az egy darab konnektor szerepel, hanem a háttér is címkézésre került. Ez a konnektoron kívüli területeket fedő egybefüggő, áttetsző piros maszkból látható. illetve a konnektoron belül található negatív terekben is külön megjelenik egy-egy szegmentált terület. Ez az apró zöld, kék, sárga maszkokból látható a csatlakozó rögzítéseinél.

Módosítottam a beállításokon, a konnektort összetett poligonként definiáltam, illetve a programkódot kiegészítettem egy sorral, amivel elértem, hogy a fekete, csatlakozón kívüli területek ne kerüljenek címkézésre.



8. Jól működő címkeadatok (saját ábra)

A nyolcadik ábrán látható, hogy a módosításokat követően egyetlen címkeadat tartozik egy képhez, kizárólag a csatlakozó került szegmentálásra. Ebből megállapítható, hogy az adatbázis megfelelően működik, alkalmas a neurális háló tanítására.

4. YOLACT modell

A YOLACT (You Only Look At CoefficienTs) egy teljes mértékben konvolúciós neurális hálózat, ami valós idejű példányszegmentációra készült. Azért esett rá a választásom, mert a kulcspontok meghatározásának szempontjából minél pontosabb maszkra van szükségünk. Ez a modell pedig egyedülálló módon az egylépcsős működési mechanizmusnak és a nagyobb méretű maszkoknak köszönhetően a többi példánál magasabb minőségű szegmentálást ígér, ezt a 9. ábra részletesen illusztrálja. [12]

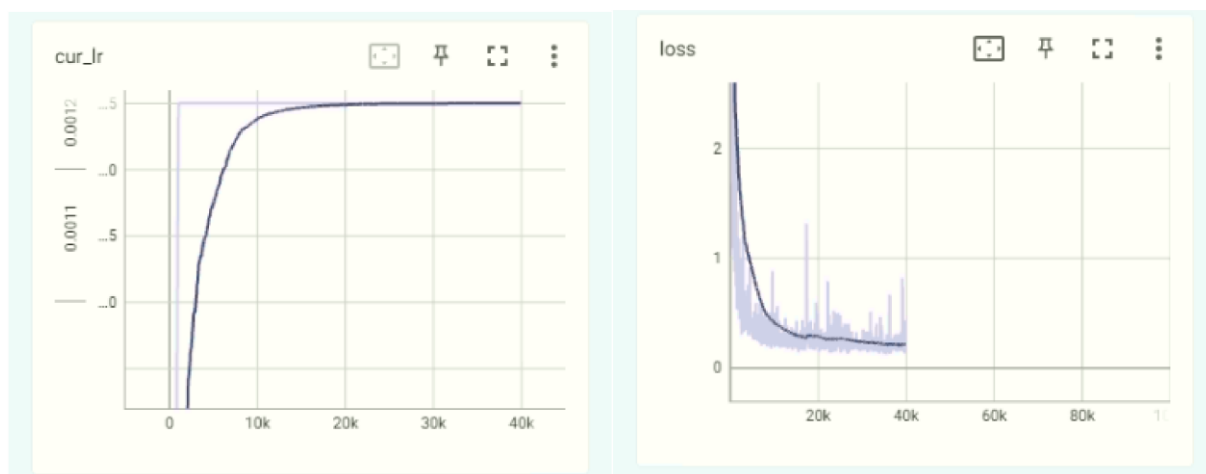


9. Különböző modellek eredményeinek összehasonlítása [12]

4.1 YOLACT modell tanítása

A neurális hálót egy 2100 képből álló adatbázissal tanítottam, ami 2100 darab címkét tartalmaz, képenként egyet, kizárólag a töltőcsatlakozót. Ebből 2000 alkotta a tanító,- 100 pedig a validációs adatokat.

Egy előre betanított Resnet-50 konvolúciós neurális hálót használtam alapként, a készítőktől származó súlyokkal. 10-es batch mérettel futott a tanítás, a learning rate értékét pedig $1e-5$ - től indítottam és a tanítás folyamán folyamatosan növekedett.



10. A Tanulási ráta növekedése és a veszteség csökkenése a tanulás során (saját ábra)

A tanulási ráta növelésére azért van szükség, mert minél alacsonyabb az érték a modellünk annál pontosabb lesz, azonban nagyon hosszú tanulási folyamatot eredményez, ha pedig magas az érték a tanítás gyors lesz ugyan, de előfordul, hogy a modellünk nem megfelelően tanulja be a feladatát. Az értékek folyamatos növelése azért megfelelő megoldás, mert a pontos súlyok kialakítása a folyamat elején a legfontosabb, így ezek remek alapként szolgálnak majd a tanítás későbbi szakaszaival, amikor a gyorsaság érdekében a tanulási ráta emelkedik. A 10. ábrán megfigyelhető veszteséggörbe meredek esését ez a módszer indokolja.

4.2 Validációs adatok kiértékelése és eredmények

A validációs értékek vizsgálatára azért van szükség, mert a tanulás vége felé jellemzően előfordul, hogy a modell ugyan egyre jobban teljesít a tanító adatbázison, azonban a független teszt példákra romlanak az eredmények. Ennek az oka, hogy elkezdenek a súlyok túlzottan illeszkedni a tanítási mintákra, és így a modell elveszíti az általánosítási képességét, ami nélkülözhetetlen az új, ismeretlen problémák hatékony megoldásához. [6]

A tanítás során figyelemmel kísértem a teszt eredményeket és a 40000. iterációhoz közeledve kezdett csökkenni a teljesítmény a független adatokon ezért leállítottam a folyamatot. A leállítás során elmentésre kerülnek a súlyok, illetve minden 10000. iterációt követően készül egy mentés a súlyokról éppen a túltanulás problémájából kifolyólag.

Ezután az adatok kiértékelésénél, melyeket a 11-14. ábrán vizuálisan is szemlétetek, egyértelműen megmutatkozott a túltanult modell gyengesége. A 14. ábrán jól látható, hogy bár a csatlakozó maszk, szép eredményeket szolgáltatott, azonban a modell emellett felismert még

egy egyöntetű fekete foltot is, míg a 12. ábrán megtekintve az eredményeket a 30000. iterációt követő mentés súlyai gyakorlatban jobban működnek, annak ellenére is, hogy a 11. és 13. ábrán látható validációs táblázatok szerint a 39911 iterációt futott tanítás jobb eredményeket mutat a teszt adatbázison.

	all	.50	.55	.60	.65	.70	.75	.80	.85	.90	.95
box	98.71	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	87.06
mask	99.84	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	98.36

11. Validációs adatbázis kiértékelésének eredménye 30000 iteráció után (saját ábra)



12. Szegmentálás 30000 végrehajtott iteráció után (saját ábra)

	all	.50	.55	.60	.65	.70	.75	.80	.85	.90	.95
box	99.63	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	96.27
mask	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0

13. Validációs adatbázis kiértékelésének eredménye 39911 iteráció után (saját ábra)



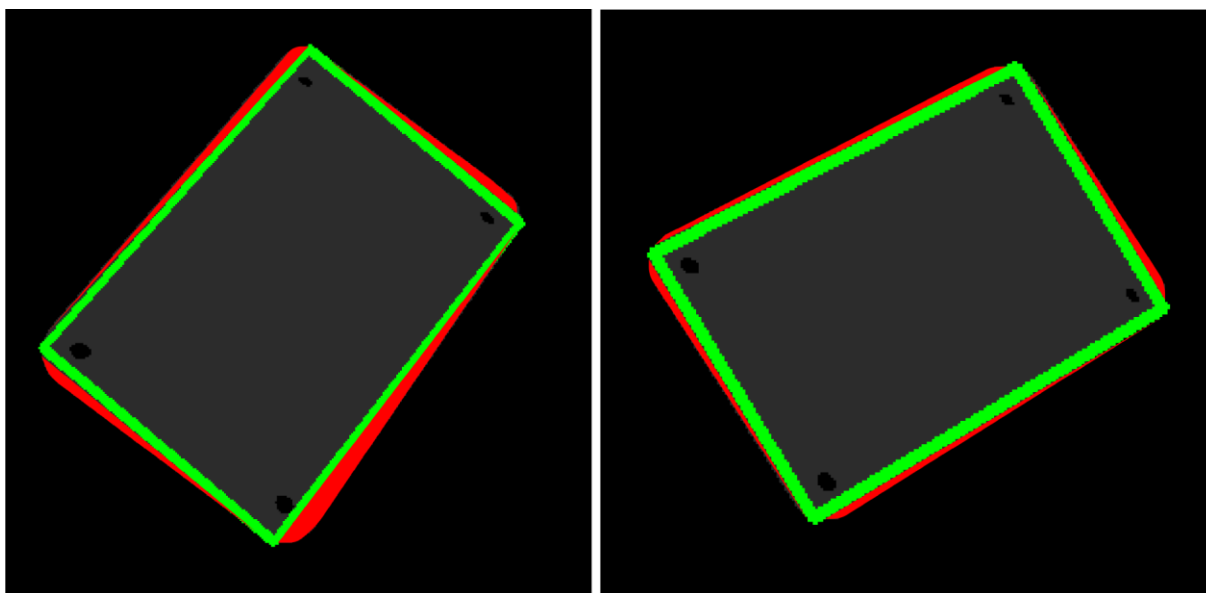
14. Szegmentálás 39911 végrehajtott iteráció után (saját ábra)

5. Kulcsponthoz kinyerése

A kulcsponthoz kinyerésére több módszer is alkalmazásra került. Először kontúrkereséssel és annak konvex hullá alakításával kísérleteztem, melynek leegyszerűsítésével szerettem volna megkapni a négy kulcsponthoz az objektum kétdimenziós perspektív vetületéről. Majd ezt követően tértem át az egyenesek illesztéséhez a felmerülő problémák kiküszöbölésére.

5.1.1 Konvex hull keresésével elért eredmények

A kontúrok konvex hullá alakítása során fény derült rá, hogy a töltőcsatlakozó lekerekített sarkai miatt nem ez a várt eredményeket hozó megoldás.



15. Kontúrokból előállított 4 pontból álló konvex hull (saját ábra)

A 15. ábrán látható pirossal kiemelt részek a burkoló dobozon kívül esnek, de még az objektum részét képezik a zöld vonal pedig a létrehozott négy pontból álló konvex hull. Ezen megfigyelhető, hogy az így kinyert 4 pont torz képet vetít az objektum perspektivikus torzulásáról, így ezekből a pontokból nem lehetséges pontosan kiszámolni azok háromdimenziós térben elfoglalt helyét.

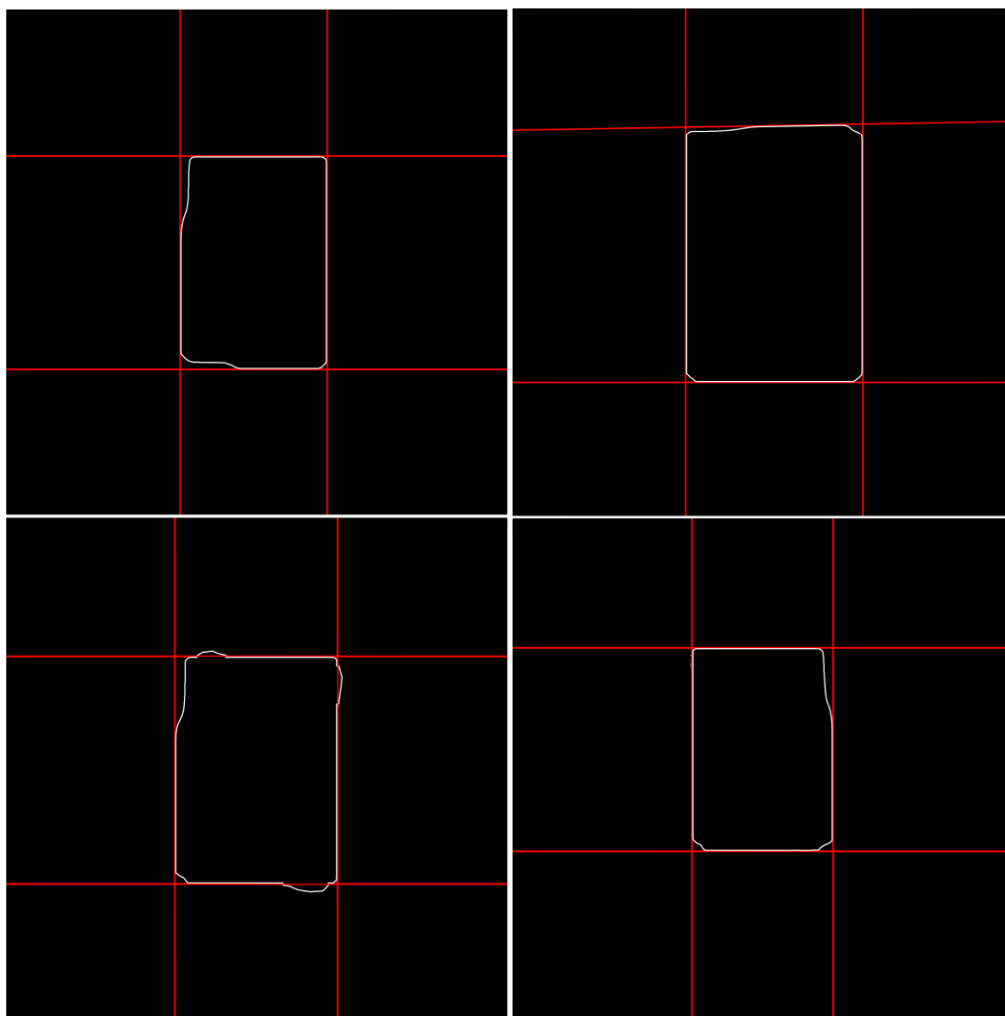
5.1.2 Kulcsponthoz keresése egyenesek illesztésével

A korábbi hiányosságok nyomán megállapítottuk, hogy a kulcsponthoz az objektum területén kívül kell meghatároznunk, a lekerekített sarkok miatt. Ezt a célobjektum oldalaihoz illesztett egyenesek metszéspontjának meghatározásával értük el.

```
dst = cv2.GaussianBlur(segment, (55,55), cv2.BORDER_DEFAULT)
dst = cv2.Canny(dst, 50, 150, apertureSize = 3)
lines = cv2.HoughLines(dst, 1, np.pi/180, 10,)
```

A fenti kód első sorában a bemeneti képen gauss-elmosást alkalmaztam, ez azért fontos, mert így a neurális háló által szolgáltatott maszk élei elmosódnak, ezzel eltüntetve a maszk apróbb egyenetlenségeit, biztosítva a hosszú egyenes szakaszokat az objektum négy oldalán az egyenesek illesztéséhez.

Ezt követően a módosított képet, átadtam egy éldetektáló Canny algoritmusnak. Majd Hough transzformáció segítségével létrehoztam az egyeneseket. A 16. ábrán piros vonalakkal kerültek szemléltetésre az egyenesek, fehérrel pedig a kontúr látható. Jól megfigyelhető, hogy a maszk egyenetlenségeitől függetlenül remekül illeszkednek az egyenesek az csatlakozó éleire.



16. Hough- transzformáció eredménye (saját ábra)

A póz esztimációhoz azonban az egyenesek metszéspontjait keressük, ehhez vízszintes és függőleges vonalak szétválasztására volt szükség. A csoportosított vonalakon ezek után, párosával vizsgálva megállapíthatók a metszések.

A Hough transzformáció során előállított ρ , θ párból álló egyenesek egyenlete a következő képpen írható fel [13]:

$$r = x \times \cos\theta + y \times \sin\theta$$

Ez alapján a két egyenes metszéspontjának megtalálásához, két ρ , θ pár esetén az x, y ismeretlenek kiszámolására van szükség. Ehhez az alábbi egyenletrendszert kell megoldani [14] [13]:

$$r_1 = x \times \cos\theta_1 + y \times \sin\theta_1$$

$$r_2 = x \times \cos\theta_2 + y \times \sin\theta_2$$

azaz,

$$AX = b$$

ahol

$$A = [\cos\theta_1 \quad \sin\theta_1] \quad b = |r_1| \quad X = |x|$$

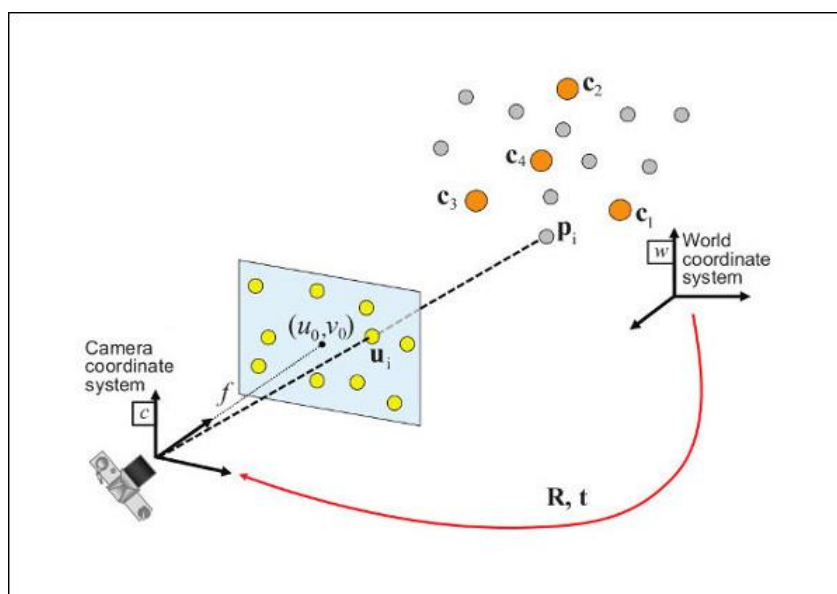
$$[\cos\theta_2 \quad \sin\theta_2] \quad |r_2| \quad |y|$$

Az alábbiakban látható a Python környezetbe foglalt formula. A lineáris egyenlet megoldása az `np.linalg.solve()` Numpy függvénnyel történik.

```
rho1, theta1 = line1[0]
rho2, theta2 = line2[0]
A = np.array([
    [np.cos(theta1), np.sin(theta1)],
    [np.cos(theta2), np.sin(theta2)]]
b = np.array([[rho1], [rho2]])
x0, y0 = np.linalg.solve(A, b)
x0, y0 = int(np.round(x0)), int(np.round(y0))
```

6. PnP algoritmus

A hat szabadságfokú póz esztimáció problémája megoldható azzal, hogy kiszámítjuk a translációs és a rotációs mátrixot, ami minimalizálja a 3D-2D pontmegfelelésekből származó hibákat. A solvePnP funkció képes erre az objektum legalább három pontja, azok kétdimenzióban meghatározott vetülete és az esztimáció során alkalmazott kamera kalibrált kamera mátrixának segítségével a 17. ábrán látható módon. [15]



17. Perspective n Point probléma vizualizálása [15]

```

points_2D = np.array([(intersections[0][0][0],intersections[0][0][1]),
                      (intersections[1][0][0],intersections[1][0][1]),
                      (intersections[2][0][0],intersections[2][0][1]),
                      (intersections[3][0][0],intersections[3][0][1])],
                      np.float32)
ind = np.lexsort((points_2D[:,1], points_2D[:,0]))
sorted = points_2D[ind]
points_3D = np.array([(-45.0,60.0,0.0),
                      (-45.0,-60.0,0.0),
                      (45.0,60.0,0.0),
                      (45.0,-60.0,0.0),])

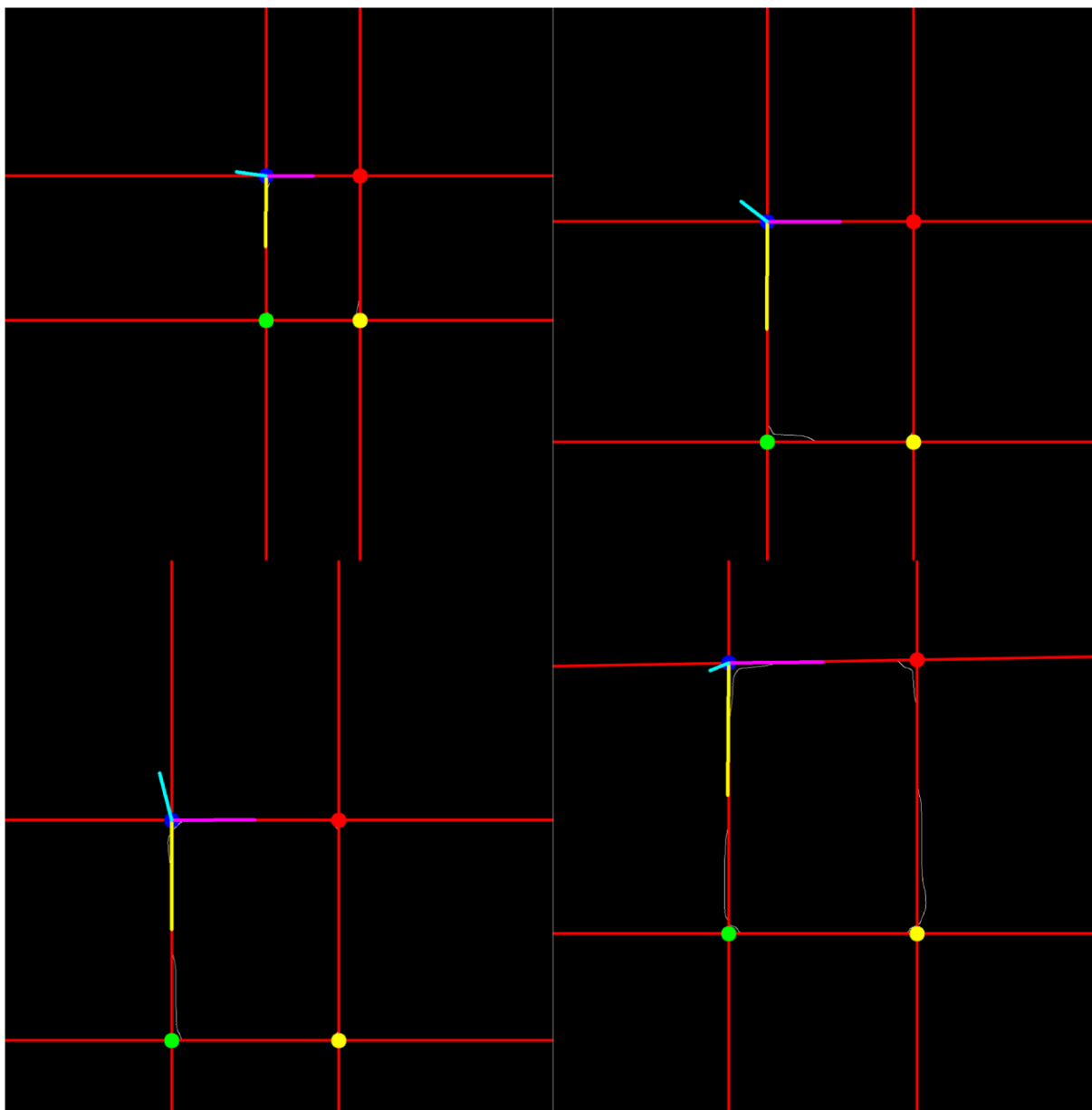
```

A program során, először az egyenesek metszetéből kinyert pontokat felvettem egy tömbbe. Ezek alkotják az objektum kétdimenziós képvetületének pontjait. Ahhoz, hogy megfeleltessem ezeket a csatlakozó saját dimenziójában meghatározott pontjaival sorba kellett rendeznem a kétdimenziós koordinátákat.

```
axis = np.float32([[ -45,0,0], [ 0,60,0], [ -45,60,45]]).reshape(-1,3)
success, rotation_vector, translation_vector = cv2.solvePnP(points_3D,
sorted, camera_matrix, dist_coeffs, flags=0)
end_point2D, jacobian = cv2.projectPoints(axis, rotation_vector,
translation_vector, camera_matrix, dist_coeffs)
```

Az axis változóban rögzítettem a pontokat, melyeknek a vetületét a számítás végén szeretném megkapni. Ezeket a pontokat egy szemléltető koordináta tengelyhármas megjelenítéséhez vettem fel. Ezt követően a cv2.solvePnP() függvénynek átadjuk a valós objektum saját dimenziójában meghatározott háromdimenziós koordinátáit, a sorbarendezett kétdimenziós pontokat, a kameramátrixot és a kamera disztorzióját és kimenetként megkapjuk a rotációs és translációs vektorokat. Így már adott minden információ ahhoz, hogy az objektum bármely pontjának a valós térben foglalt koordinátáit meghatározzuk. Ezt a cv2.projectPoints() függvénnyel tehetjük meg. A 18. ábrán látható rajzolatokat az alábbi kódsorokkal valósítottam meg:

```
cv2.line(cdst, ( int(sorted[0][0]), int(sorted[0][1])),(
int(end_point2D[0][0][0]), int(end_point2D[0][0][1])), (0,255,255), 5)
cv2.line(cdst, ( int(sorted[0][0]), int(sorted[0][1])),(
int(end_point2D[1][0][0]), int(end_point2D[1][0][1])), (255,0,255), 5)
cv2.line(cdst, ( int(sorted[0][0]), int(sorted[0][1])),(
int(end_point2D[2][0][0]), int(end_point2D[2][0][1])), (255,255,0), 5)
```



18. PnP algoritmus eredményeinek vizualizálása (saját ábra)

A 19. ábrán megfigyelhető a szegmentálás, a kinyert kulcspontok és a megalkotott koordináta-rendszer az eredeti képeken elhelyezve.

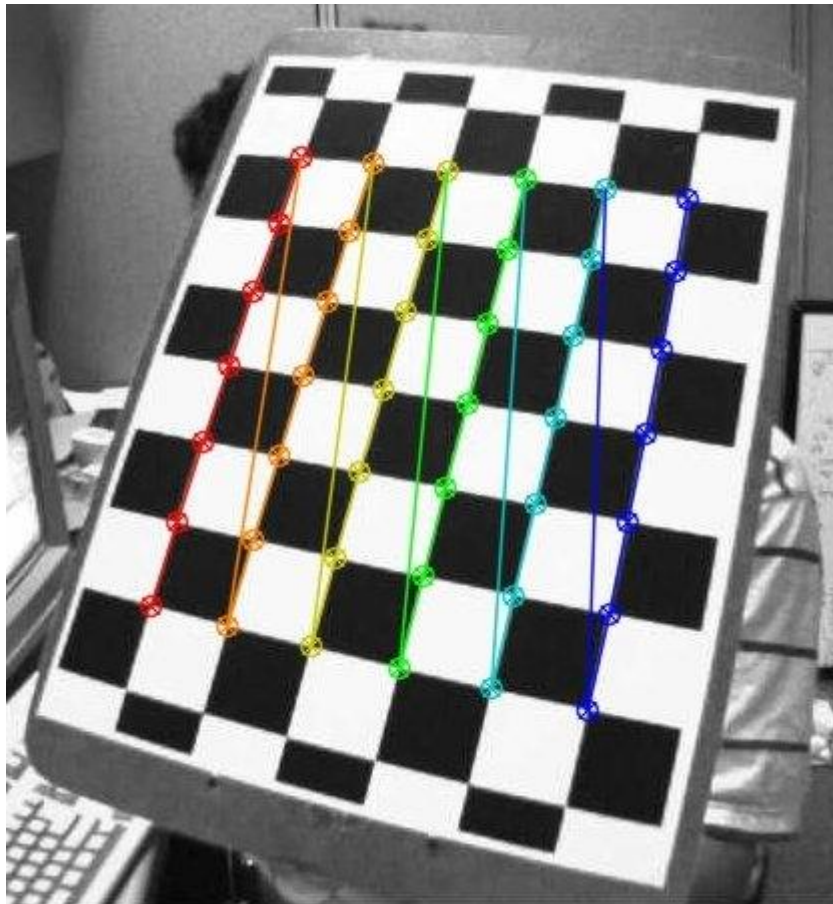


19. Eredeti képeken vizualizált eredmények (saját ábra)

A fejlesztés ideje alatt, még nem került sor kamera eszköz beszerzésére, ezért egy általános kameramátrixot alkalmaztam a projekt során. A későbbiekben azonban a lehető legnagyobb pontosság elérése érdekében feltétlenül sort kel keríteni kamera kalibrálására és ennek során a kameramátrix kinyerésére.

Egy kamera felfogható egy olyan eszközként, amely a látóterében elhelyezkedő háromdimenziós bemenetet kétdimenzióba képezi le. Ez a matematikailag jól megközelíthető tulajdonság, kulcsfontosságú szerepet játszik a gépi látás alkalmazásában. A kamera mátrix hordozza magában az összes információt, amellyel egy tetszőleges háromdimenziós pontot le tudunk képezni, vagy épp, mint esetünkben is, vissza tudunk vetíteni [16]. A kamera kalibráció során megállapítjuk a kamera gyújtótávolságát, a főtípust és annak eltolódását valamint a tengely torzulás mértékét. Erre a célra az OpenCV függvénykönyvtárából a

`cv2.calibrateCamera()`, illetve `cv2.getOptimalNewCameraMatrix()` függvények lehetnek segítségünkre. A kalibrálás során a kiválasztott eszközzel készítenek fotót, jellemzően egy sakktáblamintáról, mert ezen könnyedén meghatározhatók pontok az összehasonlításhoz. A 20. ábra épp egy ilyen mintán elvégzett kalibrációs folyamatot szemléltet.



20. Kalibrálás sakktáblamintán [17]

7. Összefoglaló

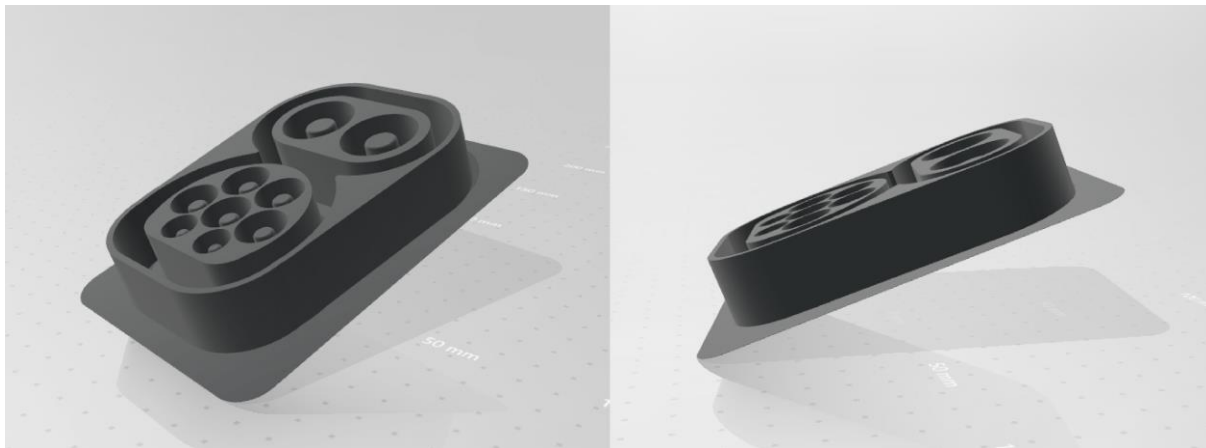
A feladatomban az volt, hogy prezentáljam a megoldást egy elektromos autó töltőcsatlakozójának felismerésére és hat szabadságfokú pózának meghatározására. A fejlesztés során felhasználtam az objektum megjelenésének jellegzetességeit, így egy specifikus, célorientált megvalósítást kaptam eredményül, amely könnyen áttekinthető és az adott feladathoz illeszkedik. Ezt a széleskörű felhasználás rovására tudtam elérni, azonban az elképzelésem az, hogy a szükséges lépések finomhangolásával bármilyen célobjektumra kiterjeszthető az alkalmazása. A 21. ábrán egy alternatív kulcspont jellemző kinyerési lehetőség látható. A körök detektálását Hough transzformációval végeztem a `cv2.HoughCircles()` függvény segítségével.



21. Körök detektálása a csatlakozón (saját ábra)

A továbbiak során szeretném javítani a neurális háló teljesítményét, mert a pontosabb szegmentációs maszkból pontosabban határozható meg a kulcspontok, megbízhatóbb eredményeket kapunk a póz esztimáció során is. Ezt az adatbázis növelésével tudom elérni. Jelenleg 2000 db képből és 2000 db annotációból áll a tanító adatbázis, ezt 10000-re, majd 20000-re szeretném növelni. Az adatbázis generálása során használt objektum, mely a csatlakozót ábrázolja, lecserélésre kerül. Az új objektum a csatlakozó legjellegzetesebb, legfontosabb látszati jellemzőivel rendelkezik, ezzel biztosítva, hogy a neurális háló

detektáláshoz és szegmentáláshoz csak a lényeges jellemzőket tanulja meg. Ezt az új objektum a 22. ábrán látható laposságával, felesleges dimenzióinak eltávolításával éri el.



22. Lapos csatlakozó (saját ábra)

A következőkben az adatbázis generálás során, a háromdimenziós szimulált jelenethez több mesterséges fényforrást is hozzáadok, hogy különböző fényviszonyok között is határozott maszkot kaphassak vissza a valós képadatokról.

A fejlesztés folyamán, sikeresen produkáltam egy speciális célra készült szintetikus adatbázist, amely betanításával egy jól teljesítő, valós idejű, példányszegmentálásra képes gépi látó modellt hoztam létre. Az ebből nyert szegmentációs maszkról egyenesek illesztésével sikeresen nyertem ki kulcspont jellemzőket, melyeket felhasználva Perspektíva-n-Pont algoritmus segítségével sikeresen határoztam meg a translációs és rotációs mátrixot.

Irodalomjegyzék

- [1] R. L. HANSON, „sophia,” [Online]. Available: <https://www.hansonrobotics.com/sophia/>. [Hozzáférés dátuma: 29 10 2022].
- [2] „IPG Photonics Website,” IPG Photonics Corporation , [Online]. Available: <https://www.genesis-systems.com/blog/role-machine-vision-industrial-robotics>. [Hozzáférés dátuma: 02 11 2022].
- [3] T. Jonathan, T. Thang, S. Balakumar, X. Yu, F. Dieter és B. Stan, „arxiv.org,” 27 09 2018. [Online]. Available: <https://arxiv.org/abs/1809.10790>. [Hozzáférés dátuma: 02 11 2022].
- [4] X. Yu, S. Tanner, N. Venkatraman és F. Dieter, „arxiv.org,” 26 05 2018. [Online]. Available: <https://arxiv.org/abs/1809.10790>. [Hozzáférés dátuma: 02 11 2022].
- [5] M. Altrichter, G. Horváth, B. Pataki, G. Strausz, G. Takács és J. Valyon, Neurális hálózatok, Budapest: Hungarian Edition Panem Könyvkiadó Kft., 2006.
- [6] R. Farkas, „inf.u-szeged.hu,” [Online]. Available: <https://www.inf.u-szeged.hu/~rfarkas/ML20/index.html>. [Hozzáférés dátuma: 30 10 2022].
- [7] D. Britz, „Understanding Convolutional Neural Networks for NLP,” [Online]. Available: <https://dennybritz.com/posts/wildml/understanding-convolutional-neural-networks-for-nlp/>. [Hozzáférés dátuma: 31 10 2022].
- [8] „Blender 3.3 Manual,” [Online]. Available: <https://docs.blender.org/manual/en/latest/contribute/index.html#contacts>. [Hozzáférés dátuma: 01 11 2022].
- [9] „COCO Common Objects in Context,” [Online]. Available: <https://cocodataset.org/#home>. [Hozzáférés dátuma: 01 11 2022].

- [10] C. Eijgenstein, „Image to coco json converter,” [Online]. Available: <https://github.com/chrise96/image-to-coco-json-converter>. [Hozzáférés dátuma: 11 09 2022].
- [11] „kaggle.com,” [Online]. Available: <https://www.kaggle.com/code/stargarden/coco-image-viewer>. [Hozzáférés dátuma: 11 09 2022].
- [12] B. Daniel, Z. Chong, X. Fanyi és J. L. Yong, „arxiv.org,” 24 10 2019. [Online]. Available: <https://arxiv.org/abs/1904.02689>. [Hozzáférés dátuma: 11 09 2022].
- [13] F. Zoltán, „matlabtricks.com,” 2013. [Online]. Available: <https://web.archive.org/web/20200430031840/http://matlabtricks.com/post-39/understanding-the-hough-transform>. [Hozzáférés dátuma: 05 11 2022].
- [14] L. Jonathan, „stackoverflow.com,” 20 12 2008. [Online]. Available: <https://stackoverflow.com/questions/383480/intersection-of-two-lines-defined-in-rho-theta-parameterization/383527#383527>. [Hozzáférés dátuma: 05 11 2022].
- [15] „docs.opencv.org,” [Online]. Available: https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html. [Hozzáférés dátuma: 05 11 2022].
- [16] Z. Kató és L. Czúni, Számítógépes látás, Typotex Kiadó, 2011.
- [17] „docs.opencv.org,” [Online]. Available: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html. [Hozzáférés dátuma: 06 11 2022].
- [18] K. Gábor, „Mesterséges intelligencia és gépi tanulás,” 25 11 2020. [Online]. Available: <https://science-cloud.hu/eloadasok/mesterseges-intelligencia-es-gepi-tanulas>. [Hozzáférés dátuma: 01 11 2022].



Ábrajegyzék

1. Elemi neuron felépítése (saját ábra).....	4
2. Neurális háló felépítése (saját ábra).....	5
3. Konvolúciós neurális háló felépítése [7]	5
4. Fénykép a csatlakozóról, 3D modell, textúrált renderkép (saját ábra)	7
5. A jelenet látszata és az erről a kamera által leképezett látvány (saját ábra)	8
6. RGB képek és a szegmentált megfelelőjük (saját ábra)	15
7. COCO formátumra konvertálás felmerülő problémái (saját ábra)	16
8. Jól működő címkeadatok (saját ábra)	17
9. Különböző modellek eredményeinek összehasonlítása [12]	18
10. A Tanulási ráta növekedése és a veszteség csökkenése a tanulás során (saját ábra) ..	19
11. Validációs adatbázis kiértékelésének eredménye 30000 iteráció után (saját ábra) ..	20
12. Szegmentálás 30000 végrehajtott iteráció után (saját ábra).....	20
13. Validációs adatbázis kiértékelésének eredménye 39911 iteráció után (saját ábra) ..	20
14. Szegmentálás 39911 végrehajtott iteráció után (saját ábra).....	21
15. Kontúrokból előállított 4 pontból álló konvex hull (saját ábra).....	22
16. Hough- transzformáció eredménye (saját ábra).....	23
17. Perspective n Point probléma vizualizálása [15]	25
18. PnP algoritmus eredményeinek vizualizálása (saját ábra)	27
19. Eredeti képeken vizualizált eredmények (saját ábra)	28
20. Kalibrálás sakktáblamintán [17]	29
21. Körök detektálása a csatlakozón (saját ábra).....	30
22. Lapos csatlakozó (saját ábra).....	31