



BME Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Számítógépes látórendszerek

VIIIMA07 2021/22 2. félév (MSc)

Házi feladat dokumentáció

Fekete Balázs Attila (I552DO)

Jánoki Csaba (AELQUH)

Kolontári Péter (VSD92X)

Szécsényi Nándor (RJ448X)

Tartalomjegyzék

Feladat rövid leírása-feladatok felosztása	2
Objektum detektálás 2D-ben	2
Adathalmaz előállítása	2
Hálózat választása	3
Hálózat tanítása	3
Kapott eredmények értékelése	4
Továbbfejlesztési lehetőségek	8
Objektum detektálás 3D-ben	9
Objektumok körbevágása és maszkolása	9
Koordináta transzformálás	11
Tesztelés	11
Szemantikus szegmentálás	14
Adathalmaz előállítása	14
Hálózat megválasztása	17
Kapott eredmények értékelése	18
Út szegmentáció	20

Feladat rövid leírása-feladatok felosztása

A hatékony aszinkron munkavégzés érdekében a megadott feladatokat szétosztottuk, hogy mindenki feladattal tudjunk foglalkozni. Az első feladattal Fekete Balázs Attila foglalkozott. Ő végezte a 2D objektum detektálást. A 3D objektum detektálást, azaz a második feladatot Jánoki Csaba kapta. A szemantikus szegmentálás részét Szécsényi Nándor oldotta meg, míg az út detektálása pedig Kolontári Péter feladata volt.

Objektum detektálás 2D-ben

A feladat célja, hogy felismerjen öt fajta objektumtípust és az észlelt objektumok körbefoglaló téglalapok paramétereit meghatározza.

A tárgyban szerzett ismeretek alapján a feladat legelején azt kellett eldöntenem, hogy hagyományos látással vagy neurális hálók segítségével szeretném megoldani a feladatot, hiszen a feladat ezt nem specifikálta. Korábban neurális hálóval nem foglalkoztam, így az az út több újdonságot rejtett számomra, így azt a megoldási lehetőséget választottam.

Első nekifutásra nekiáltam egy saját háló építésének, azonban ez az első teszteken nagyon rossz eredményeket hozott, így egy rövid internetes keresés után talált előtanított hálóra esett a választásom.

Adathalmaz előállítása

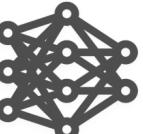
A neurális háló feltanítása előtt a megfelelő adatbázist kellett létrehoznom. Ehhez a rendelkezésre álló képeket két csoportra bontottam. A képek 80%-án tanítottam a hálót, a maradék 20%-ot pedig validációra használtam. Ahhoz, hogy ez a tanítás sikeres legyen, az adatbázisban található különböző mappák mindeneket 80-20 arányban válogattam szét, ezzel a hálót minden egyes szituációra tanítom. (éjszaka, esős idő, stb.) Ezek alapján a képeket újra számoztam és két mappába (train/images; valid/images) rendeztem őket.

A képekkel egyetemben a txt fájlokat is fel kell készíteni a tanításra. Ehhez a bennük található számunkra lényegtelen információkat töröltem, majd a megfelelő sorszámmal a megfelelő mappába mentettem. Továbbá töröltem azokhoz a txt-hez tartozó képeket, amelyeken egyetlen egy detektálandó objektum sem található (azaz a txt üres).

A háló tanítás során kiderült, hogy az így létrejött 979 tanító kép közül 153 db még így is corrupt volt, azaz például valamelyen oknál fogva a benne található értékek korruptak voltak. Ezeket a tanító algoritmus lekezelte, így ezzel nem foglalkoztam az adatbázis előállítása közben.

Hálózat választása

Az előtanított hálók közül a Pytorch ¹Yolov5-ös hálóját találtam a legalkalmasabbnak a feladat elvégzésére. Ezen háló csomag több méretű előre feltanított hálót tartalmaz.

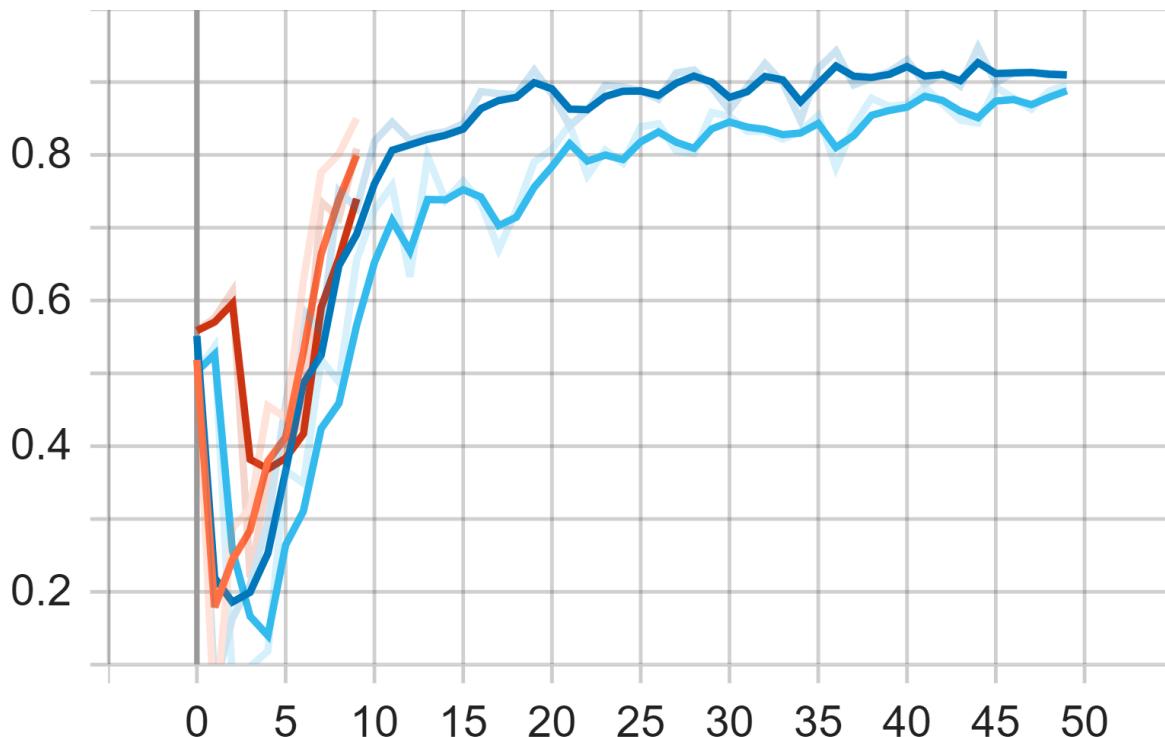
					
	Nano	Small	Medium	Large	XLarge
YOLOv5n					
4 MB _{FP16}	14 MB _{FP16}	41 MB _{FP16}	89 MB _{FP16}	166 MB _{FP16}	
6.3 ms _{V100}	6.4 ms _{V100}	8.2 ms _{V100}	10.1 ms _{V100}	12.1 ms _{V100}	
28.4 mAP _{COCO}	37.2 mAP _{COCO}	45.2 mAP _{COCO}	48.8 mAP _{COCO}	50.7 mAP _{COCO}	

Mivel az általunk detektálni kívánt objektumok 5 osztályba sorolhatók, így egyértelmű, hogy számukra egy kisebb háló is elég lesz. Azonban nem egyértelmű, hogy a nano, small hálók közül melyik lesz számunkra megfelelő, így minden háló tanítását elvégeztem és szerzett tapasztalatok alapján hoztam meg a döntésem.

Hálózat tanítása

Az előtanított hálózatot az általunk használt adatokkal újra tanítottam. Ehhez a fejleszők által megírt train.py függvényt használtam, azonban ahhoz, hogy megfelelő adatokon dolgozzon a coco128.yaml fájl-t módosítanom kellett. (Itt adtam meg az általam használt tanító, validáló adatok elérési útját, illetve a kimenti osztályok számát és nevét.) Szemléltetés és átláthatóság kedvéért a wandb programot telepítettem.

¹ https://pytorch.org/hub/ultralytics_yolov5/



A képen látható diagram a Precession-t ábrázolja az epochok számának függvényében. A narancssárga és a világoskék vonallal ábrázolt vonalak esetén a tanított háló nano háló volt, míg a pirossal, sötétkékkel jelöltek esetén small hálót tanítottam. A grafikonról a következő következtetéseket vontam le:

1. A small Yolov5 háló lesz számunkra a megfelelő.
2. 25 epoch-nál több használata nem javasolt, hiszen ekkor a pontosság már körülbelül állandó, a további ciklusokkal csak az overfitting valószínűségét növelnénk.

Ezek alapján elvégeztem háló tanítását.

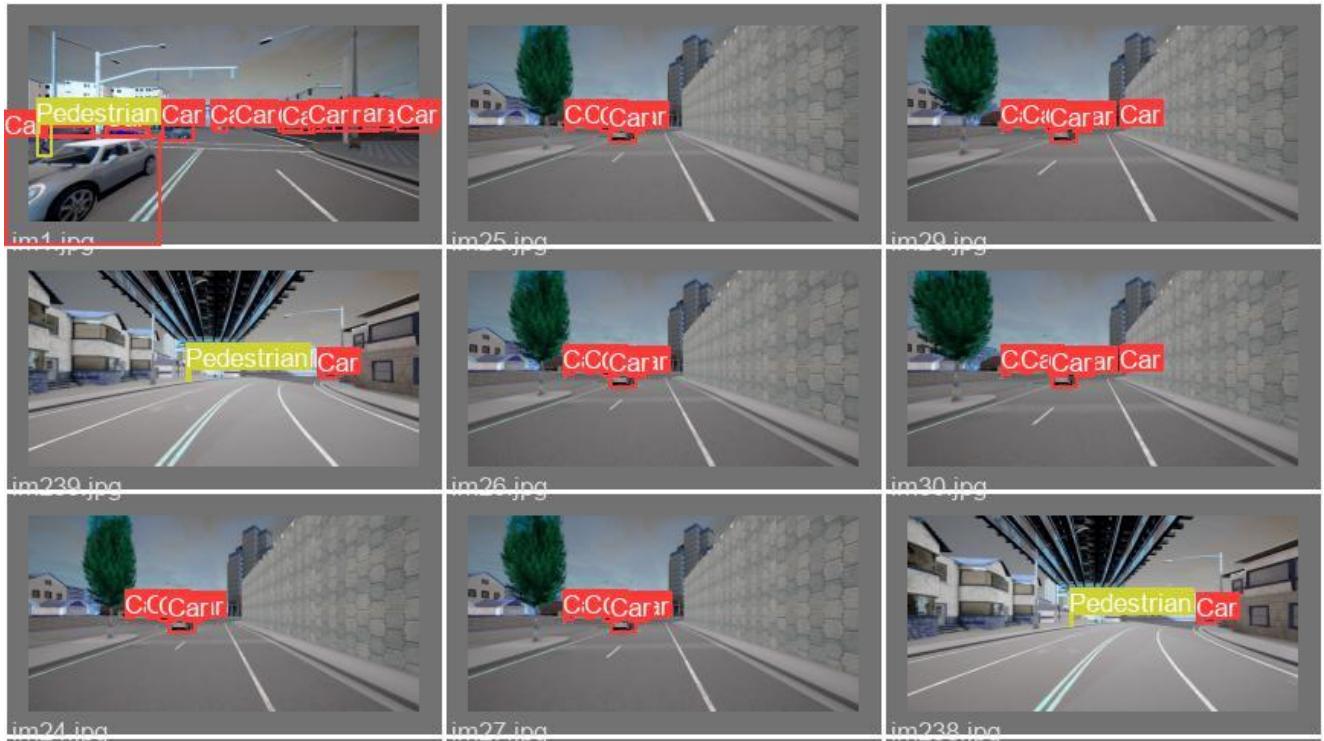
Kapott eredmények értékelése

A 25 epoch alatt kapott validációs értékek (Iou threshold 0.5, Confidences threshold 0.00001):

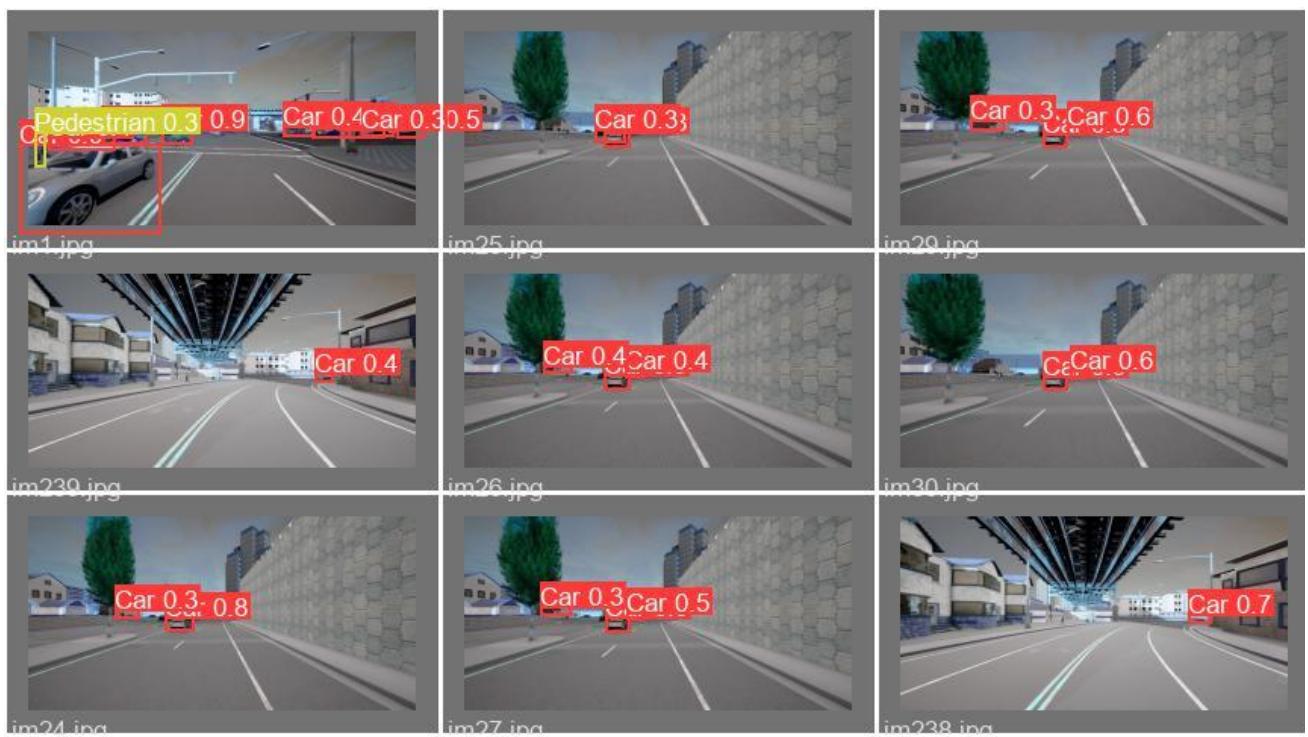
Model summary: 213 layers, 7023610 parameters, 0 gradients, 15.8 GFLOPs						
Class	Images	Labels	P	R	mAP@.5	mAP@.95: 100% 4/4 [00:06<00:00, 1.70s/it]
all	198	922	0.901	0.754	0.82	0.488
Car	198	740	0.876	0.834	0.876	0.577
Pedestrian	198	182	0.925	0.674	0.764	0.399

Ezek az értékek nem jók, azonban nem is tragikusak. Elmondható, hogy a valóban detektálandó objektumok végig detektálva maradnak több képkockán keresztül, míg false adatok max egy kép kockáig maradnak aktívak, így ezek szűrhetők.

A következőkben néhány a validációs folyamat során a háló által számolt képet hasonlítok össze az elvárt kimenettel.



Elvárt kimenet.



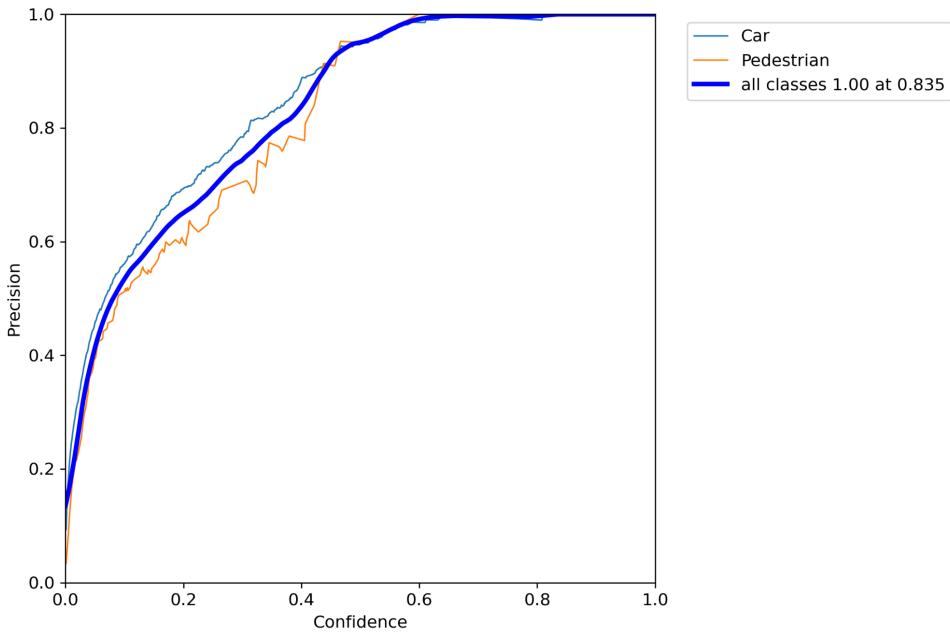
Valós kimenet.

Ahogy a képeken láthatjuk a neurális háló viszonylag jól működik. Azonban, ahogy természetesen minél messzebb van egy objektum, annál nehezebben ismeri fel azt. Továbbá a labelekben talált txt-eket vizsgálva kiderül,

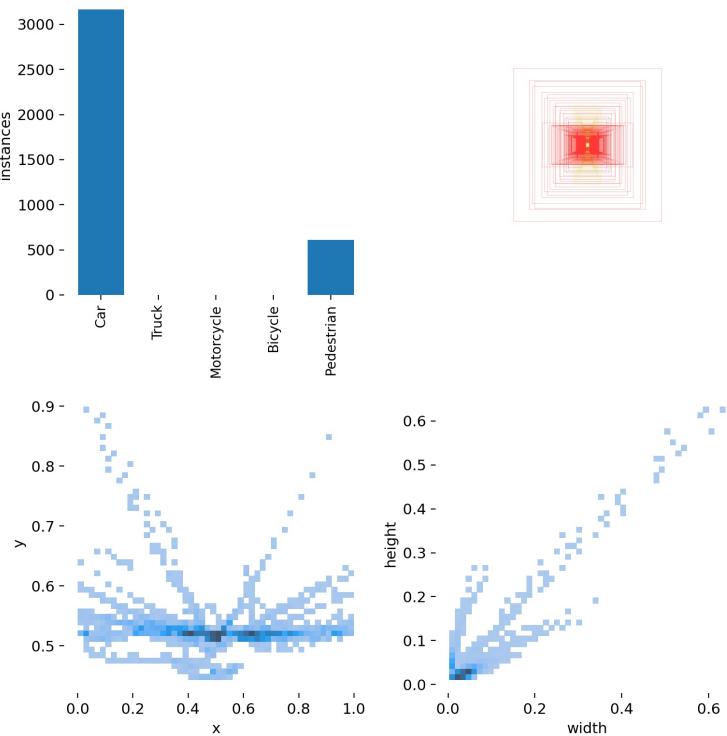
hogy általában kevesebb objektumot talál meg, mint ami a valóságban a kapott képen megtalálható.



Továbbá előfordul, hogy a távol lévő néhány pixel “széles” gyalogosokat egyáltalán nem ismeri fel, azonban ehhez közeledve minden megtalálja azokat, így ezt nem tekintettem komoly hibának. A működés szempontjából fontos közeli objektumokat mind nagy confidence szinttel felismeri. Az, hogy az autó egészére nem szerepel a bounding boxban nem okoz nagy hibát a 3D objektum detektálást végző csapattársam szerint.



A fentebbi képek alapján az objektum detektálásra használt függvény Confidence szintjét 0,2-re állítottam. Így valóban detektál néhány false objektumokat, azonban a tapasztalat azt mutatja, hogy ezek csak max 1,2 képkockáig élnek utána, melyeket lehet szűrni, illetve Jánoki Csaba csapattársammal egyeztetve a szegmentált kép segítségével ezeket a false objektum detektálásokat könnyen szűri.



Ahogy a fenti képen is láthatjuk az autón és gyalogoson kívül nem detektáltunk mást, ez is mutatja, hogy a tanító adatbázis fejlesztésre szorul.

Továbbfejlesztési lehetőségek

A nem tökéletes működés miatt sokat gondolkoztam a továbbfejlesztesőgi lehetőségeken:

1. A feladat által is említett mélyiségi kép hozzárendelése megoldás lehet, azonban Szécsényi Nándor csapattársammal egyeztetve nála ez a művelet nem oldotta meg a pontosság problémáját, szóval valószínűleg ennek a bevezetése nem lenne elég.
2. Hagyományos módszerekkel a kép előfeldolgozása a háló számára. Például a házak “levágása” a képről, hiszen gyakori probléma volt, hogy azokon false autó objektumokat érzékelt.
3. A leghatékonyabb továbbfejlesztés lenne, ha még több tanító adatot generálnánk.

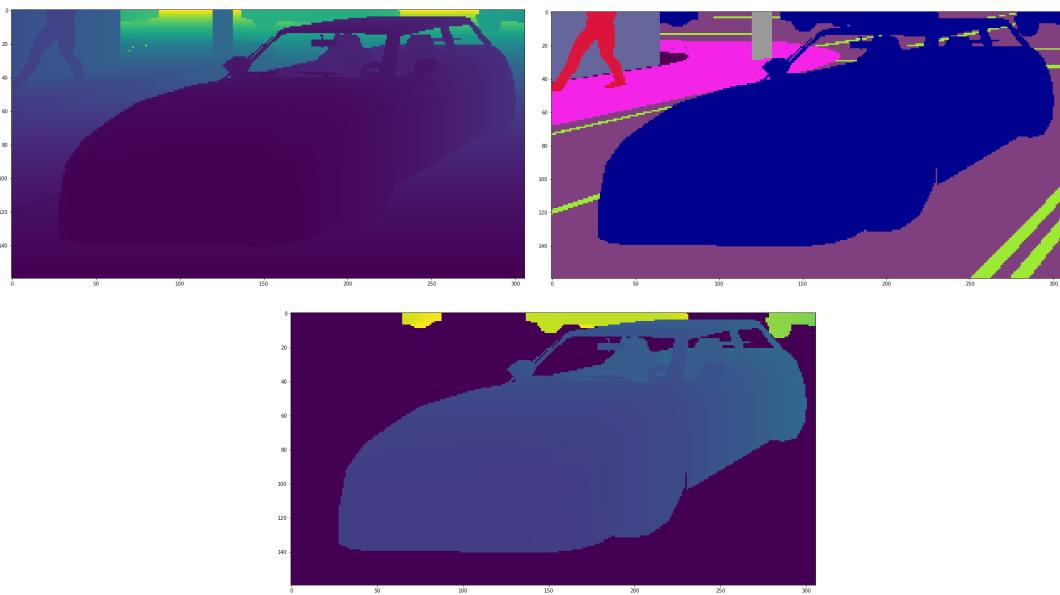
Objektum detektálás 3D-ben

A feladat célja, hogy a 2D részfeladatok eredményeit, illetve a mélyiségi képet felhasználva a lehető leg pontosabban megbecsüljük, hogy hol helyezkednek el a különféle detektált objektumok az autóhoz képest, mégpedig úgy, hogy meghatározzuk az adott objektum legközelebbi, illetve legtávolabbi pontját a mi autónkhoz képest minden koordináta tengely mentén.

Objektumok körbevágása és maszkolása

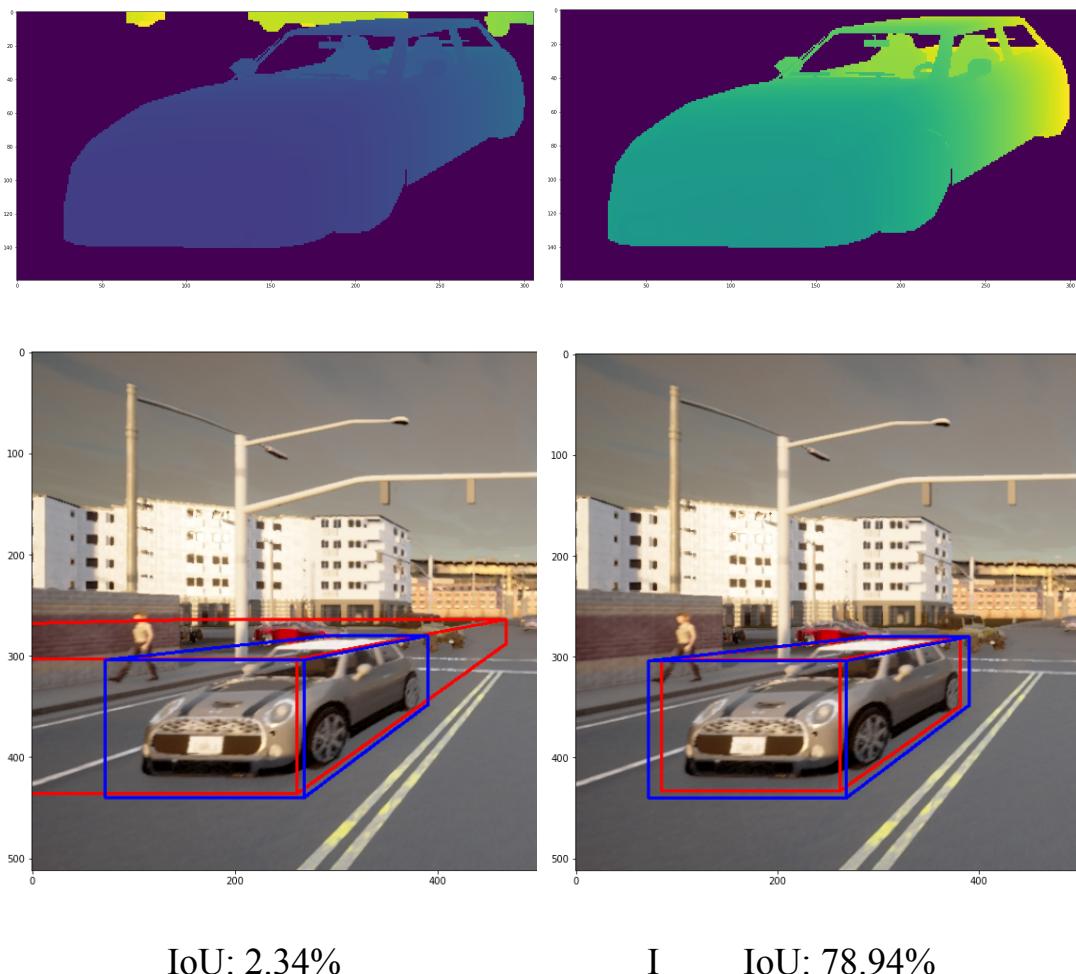
A 3D koordináták számításának első lépéseként a 2D feladat eredményei kerülnek felhasználásra. A crop_all_BBs függvény egy adott fényképen az összes objektumot kis képekre vágja szét a kapott bounding box paraméterek alapján. A függvény visszatéríti a körülvágott objektum képeket, illetve az eredeti képen ezek bal felső koordinátáját.

Az egyes körül vágásokért a crop_one_BB függvény felel, amely elsőként a körül vágást végez el, majd egy maszkolást hajt végre.



A folyamatot a fenti ábra szemlélteti: adott a bounding box mentén körbevágott mélységi (fent balra), illetve szegmentált kép (fent jobbra) Ezek alapján megalkotásra kerül az alsó kép, amely szintén egy mélységi kép, de azon képpontok, amelyek a szegmentált kép alapján nem tartoznak az objektumhoz 0 távolság értéket kapnak (maszkolt mélységi kép).

Ugyanakkor mivel a szegmentálás sosem lesz tökéletes, ezért minden lesznek olyan pixelek, amik ugyan nem tartoznak az objektumhoz, de mégis potenciális szélsőértékeként ronthatják a becslés pontosságát. A tapasztalatok azt mutatják, hogy ez a „z” tengely mentén okoz jelentős problémát, ugyanis, akár egyetlen pixelnyi félre szegmentálás is jelentheti azt, hogy az autó részének tekintünk egy messze, a háttérben lévő pontot, míg az „x” és „y” tengelyek esetén az ilyen tévedések nem okoznak jelentős különbséget. Éppen ezért valahogy mindenkorban ki kell szűrni ezeket az outlier értékeket. Eleinte z score számítás lett kipróbálva, és ez alapján a túl közel/távoli pontok depth értéke 0-ra lett állítva, de ez nem működött túlságosan jól, meglehetősen pontatlan becsléseket eredményezett. Helyette sokkal jobban bevált az, hogy a már (szegmentált kép alapján) kimaszkolt kép távolság értékeinek mediánja került megkeresésre, és ezen értékhez képest egy bizonyos \pm trashold tartományon kívül minden távolság érték kinullázásra kerül. Mivel jelenleg az objektumok nem túl nagy kiterjedésűek, ezért nem túl valószínű (későbbi tesztek is ezt mutatják), hogy ezen az intervallumon jelentősen túl nyúljanak.



A fenti képen látható az outlier kiszűrési folyamat. A bal felső képen a fent már bemutatott maszkolt mélységi kép, ahol a háttérben sárgás zöldes színnel láthatóak egyéb autók, amelyek jól látható problémát okoznak a bounding box koordináták meghatározásakor, ugyanis azt hiszi a program, hogy az autó leghátsó pontja sokkal messzebb van, mint valójában. A jobb oldali képen már a fent ismertetett medián körüli intervallummal szűrt kép látható, ami sokkal pontosabb eredményt ad.

Koordináta transzformálás

Ahhoz, hogy az autó nézőpontjához képet lehessen a koordinátákat meghatározni elsőként a kép koordinátáiból világ koordinátákba kell a képet transzformálni. A két koordináta rendszer közötti áttérésre a `world_to_image`, illetve `image_to_world` függvények szolgálnak.

A transzformálás gyakorlatilag úgy zajlik, hogy az előző lépésekben az objektum bounding box mentén körbevágott depth kép összes nem nulla eleme áttranszformálásra kerül a képen elfoglalt pixel pozíció és a távolság érték

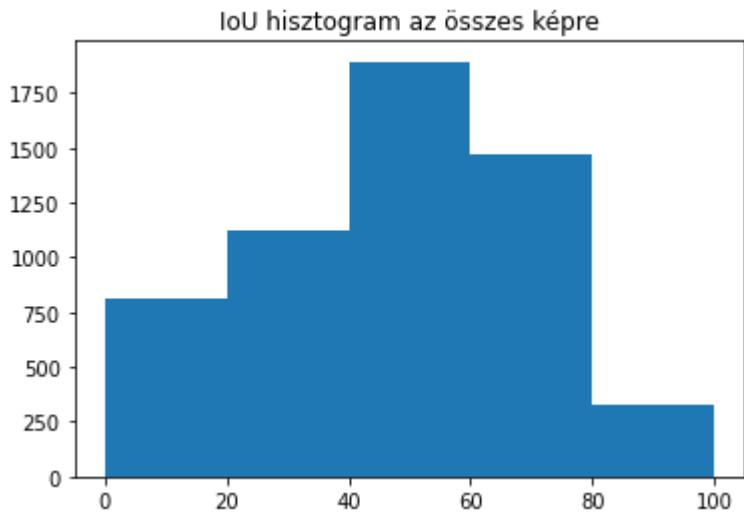
alapján. Ezáltal eredményül kapunk egy ponthalmazt, amely tartalmazza az objektumunkhoz tartozó 3D koordinátákat. Ezt követően pedig lehet két olyan ponttal jellemzni az objektumot a térben, amik minden tengely mentén a legkisebb, illetve legnagyobb értékű koordinátákkal rendelkeznek. Ezen értékek megkaphatóak a világ koordináta pontkból szélsőérték kereséssel, ezt végzi el a search_min_max_coordinates függvény. Ez a két kitüntetett pont gyakorlatilag egy 3D bounding box-ot definiál az objektum körül.

Problémát itt az tud jelenteni, hogy ha olyan 2D bounding boxot kapunk bemenetként, amin esetleg nincsen semmi hasznos objektum, ebben az esetben ugyanis a maszkolás során minden pixel 0 értéket fog kapni, és nyilván ebből semmilyen információt nem lehet majd kikövetkeztetni a minimum és maximum pontok tekintetében, ezért ekkor NaN értékeket térit vissza a függvény a szélsőértékekre. A program további részében pedig lehet ezt olyan módon szűrni, hogy ha NaN a szélsőérték, akkor szimplán nem foglalkozunk vele.

Tesztelés

A pontosság mérésre az IoU mérőszámot alkalmaztuk, amely 3D esetben az objektumot határoló prediktált és valós téglatestek térfogat metszeteinek, illetve uniójuknak hányadosa. Ezt követően az IoU érték kiszámolásra került az összes kép összes bounding boxára, ezen értékek átlaga eleinte 31.71% volt.

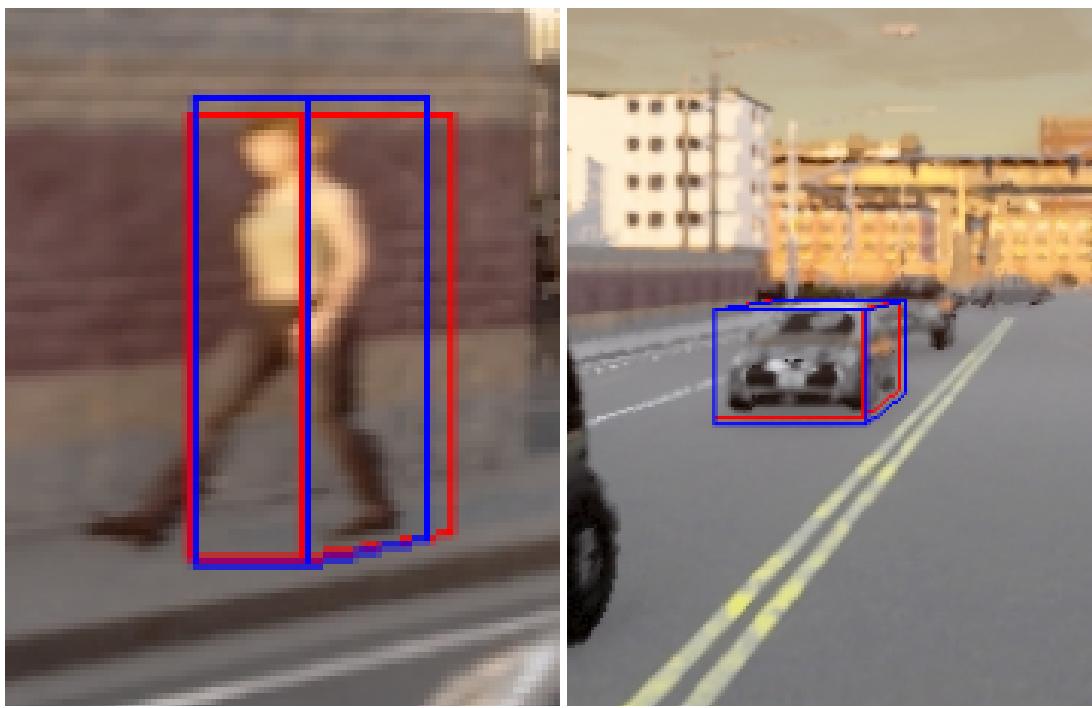
Ezen sokat javított a fent ismertetett medián értékes megoldás, amivel 49.11%-ig sikerült ezt az IoU értéket felvinni. Különféle toleranciasávok lettek kipróbálva, végül a legjobb eredményt a mediántól számított ± 305 cm adta. A medián helyett az átlagérték is kipróbálásra került, de az valamivel rosszabb eredményeket hozott (45% körül). A lenti ábrán a végleges paraméterek melletti bounding box IoU értékek eloszlása látható.



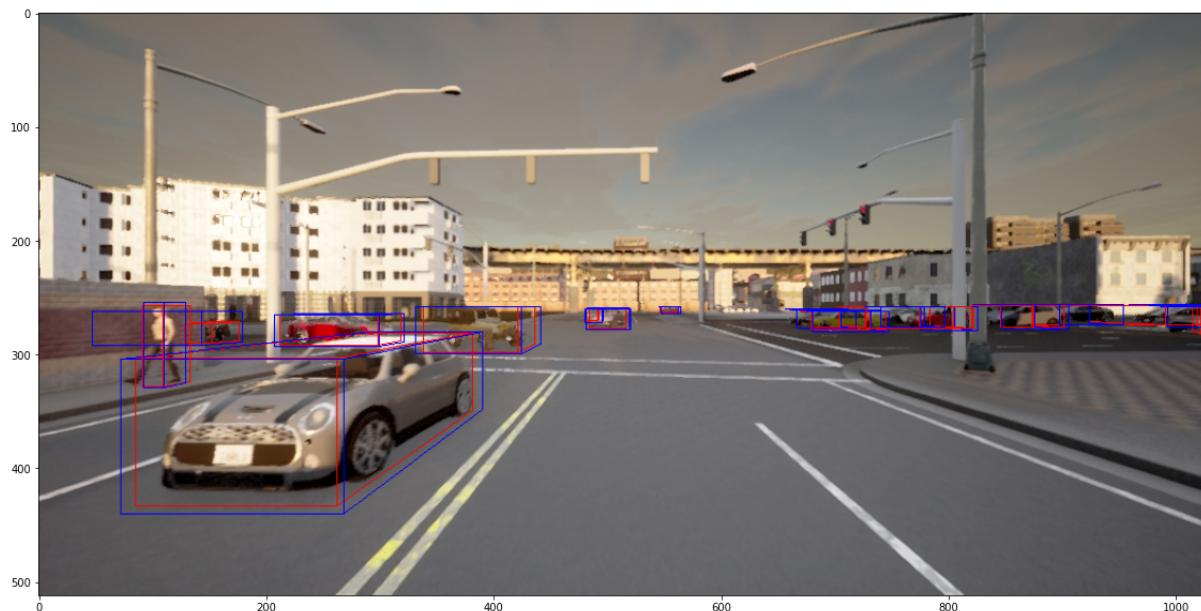
Ugyan megjelennek az alacsonyabb IoU százalékok is, viszont ezek jellemzően távol lévő, szabad szemmel is nehezen kivehető objektumok, a közeliek esetén meglehetősen pontos eredmények jellemzőek.

Mivel a koordináta transzformáció egy viszonylag lassú folyamat, ezért ezen a szakaszon külön törekedtünk az algoritmus gyorsítására. Abban az esetben tud különösen lassú lenni, ha sok, illetve nagy objektum található a képen, ugyanis akkor sokkal több pixelt kell áttranszformálni. Viszont egy nagy kép esetén szinte teljesen felesleges minden képpont transzformálása, ugyanis a közeli pixelek közel hasonló eredményt fognak produkálni, ezért bizonyos objektum pixelméret felett csupán minden második, harmadik stb. pixel kerül transzformálásra, ezzel meggyorsítva az algoritmus futását (2-szeres sebesség növekedés), cserébe a pontosság elhanyagolható mértékben romlott (49.11% IoU-ról 49.01%-ra). Végérényesen sikerült a valós idejű működést elérni, az összes rendelkezésre álló kép alapján számított átlagos feldolgozási idő adott futtatástól függően 15-16 ms.





A fenti képeken látható a végeredményből néhány válogatott objektum, ahol kékkel látható az ideálisnak kikiáltott 3D bounding boxok láthatóak, illetve piros színnel a saját algoritmus által prediktált eredmények. A többségük meglehetősen jól illeszkedik még a távoli objektumok esetén is, sőt valamikor ránézés alapján jobban illeszkednek, mint a hivatalos eredmény (pl.: bal felső kép). A fent mutatott boxok IoU értéke rendre: 82.65%, 59.35%, 56.86%, 78.2%.



Szemantikus szegmentálás

Ebben a feladatban az eredeti RGB képeket kellett pixelszinten szegmentálni az előre megadott objektumosztályok valamelyikébe, amelyhez neurális hálózatot kellett alkalmazni.

Adathalmaz előállítása

Első lépésként a megadott képeket fel kellett bontanunk tanító és validációs részhalmazra, hogy a hálózat teljesítményét megfelelően ki tudjuk értékelni. Ehhez célszerűnek tűnt a megadott videók 80%-át a tanító, 20%-át pedig a validációs adathalmazba sorolni, hiszen így minden környezeti beállítás (napszak, időjárás) képviselteti magát az egyes részekben. A szétválasztás eredményeképpen egy kb. 1000-es méretű tanító, valamint egy kb. 250 adatból álló validációs halmazt kaptunk, ami a hálózat megfelelő feltanításához kevésnek bizonyult, így megoldást kellett találnunk erre a problémára.

Két lehetőség vetődött fel: vagy már előre feltanított hálózatot futtatunk az adathalmazon és így azt fine-tune-oljuk (lényegében ez a transfer learning megoldás), vagy az eredeti képekre alkalmazunk különböző transzformációkat, amik a rajtuk lévő információkat megőrzik, mégis ezáltal a hálózat számára egy új mintát tudunk nyújtani.

A transzformációk kiválasztása előtt célszerű az eredeti képek méretét lecsökkentenünk, mivel a nagyméretű képekre való neruális hálózat tanítása sokkal nehezebb, valamint a méretcsökkentés nem okoz túlságosan nagy információveszteséget a szemantikus szegmentáció szempontjából. A feladat során ezért 128x256-as képekkel dolgoztunk, amely méret kedvező abból a szempontból is, hogy a hálózatban lévő leosztások, méret csökkentések (például Pooling rétegek) többszöri elvégzése sem eredményez páratlan számot, ami megnehezítené a hálózat felépítését a visszaterjesztésnél. Ezen kívül megőriztük az eredeti minták méretarányait, hiszen lényegében 4-el osztottuk minden kettő méretet (512x1024 volt az eredeti méret).

A transzformációk elvégzésénél fontos, hogy véletlenszerűen tudjuk azokat elvégezni a képeken: a tanító adathalmazban ha egy kép lényegében többször van jelen, akkor az elrontja a hálózat általánosító képességét. Fontos továbbá, hogy a változtatásokat csak a tanító adathalmazon végezzük el, míg a validációs adathalmazt nem módosítjuk, így a hálózat teljesítménye a valóságos képekre lesz értelmezhető. Másik lényeges szempont, hogy ha a bemenetet transzformáljuk, akkor az elvárt kimenetet is éppen ugyanúgy kell

megváltoztatnunk, hiszen csak így kaphatunk a hálózat számára hasznos tanító mintákat.

A fenti elvárásokat kielégítő program az [Albumentations](#), ami képes sokféle transzformáció elvégzésére az egyszerűbbektől az egészen összetettekig és egy viszonylag egyszerű módot kínál az ezekből összeállítható listák, pipeline-ok létrehozására. Alább látható az általunk a képeken alkalmazott transzformációk listája és paramétereik:

```
transform = A.Compose([
    A.HorizontalFlip(p=0.5),

    A.ShiftScaleRotate(scale_limit=0, rotate_limit=0.1, shift_limit=0, p=1, border_mode=cv2.BORDER_WRAP),
    A.GaussNoise(p=0.2),

    A.OneOf(
        [
            A.CLAHE(p=1),
            A.RandomBrightness(p=1),
            A.RandomGamma(p=1),
        ],
        p=0.9,
    ),

    A.OneOf([
        [
            A.Sharpen(p=1),
            A.Blur(blur_limit=3, p=1),
            A.MotionBlur(blur_limit=3, p=1),
        ],
        p=0.9,
    }),

    A.OneOf(
        [
            A.RandomContrast(p=1),
            A.HueSaturationValue(p=1),
        ],
        p=0.9,
    ),
],)
```

Igyekeztünk olyan átalakításokat választani, amelyek nem túl durvák, azaz elvégzésük után még felismerhetők maradnak az eredeti kép főbb vonásai, így főleg csak forgatást, zaj hozzáadását és különböző élesség és színkontraszt változtatásokat definiáltunk. Ezek eredményeképpen az alábbiakhoz hasonló minták jöttek létre:

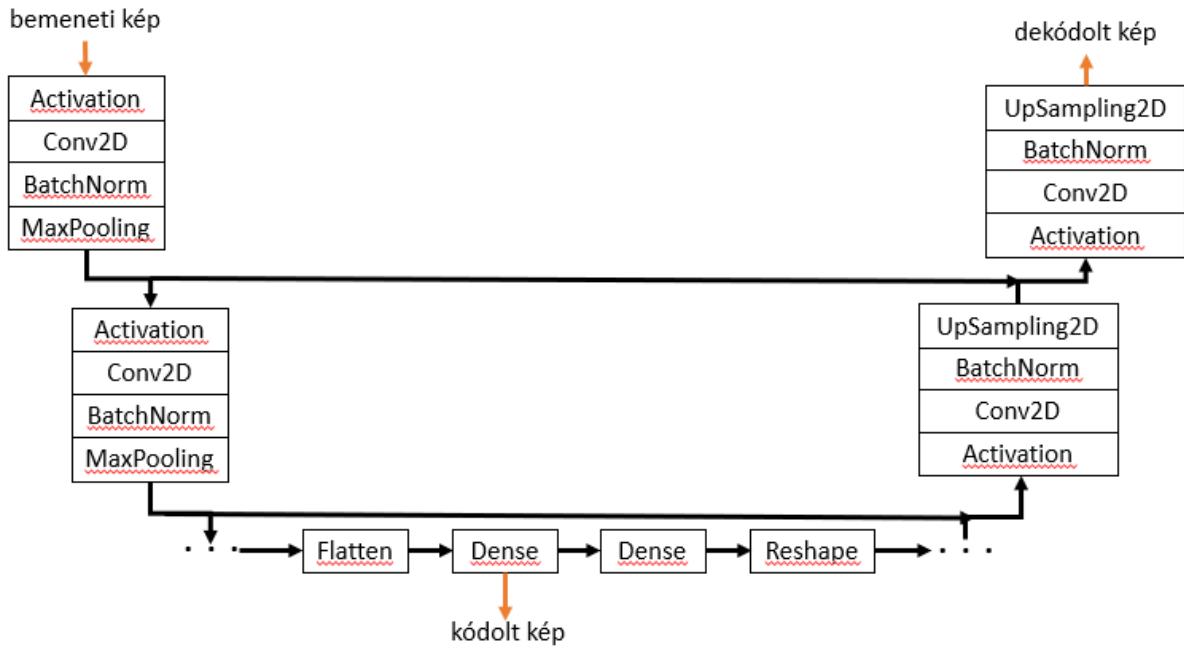


A fenti ábrán egy azonos eredeti képből készített 4 különböző tanító mintát láthatunk, amiken nagyrészt az összes alkalmazott transzformáció valamelyike megfigyelhető.

Az így elvégzett adathalmaz bővítés után 24900 tanító képet kaptunk, ugyanis minden képet 25-ször vittük végig a transzformációs pipeline-on.

Hálózat megválasztása

A szemantikus szegmentálás feladatához megfelelő architektúra az U-Net hálózat: ez lényegében egy enkóder és egy dekóder hálózatrészből áll, amelyeket összekapcsolva egy autoenkóder struktúrát kapunk. Az U-Net ezt még kiegészíti vízszintes átkötésekkel, amelyek az azonos képméretű enkóder és dekóder rétegeket vonják össze és adják a következő dekóder réteg bemenetére, így megtartva és beépítve a különböző komplexitású információkat a kimeneti képhez. Fontos még, hogy a kimenetünk ebben az esetben nem egyezik meg a bemenettel, hanem egy olyan tenzort állítunk elő, amelynek csatornaszáma megegyezik a lehetséges osztályok számával és minden egyik képpontbeli értéke megadja, hogy az adott pixel mekkora valószínűsséggel esik az egyes osztályokba. Ahhoz, hogy egy ilyen kimenetet kapunk, az U-Net utolsó konvolúciós rétege után egy másikat helyezünk, amelynek filterszáma megegyezik az osztályok számával, illetve aktivációként softmax-ot választunk, hogy ténylegesen valószínűségi értékeket kapunk. Az így kapott hálózat struktúra az alábbi ábrán látható:

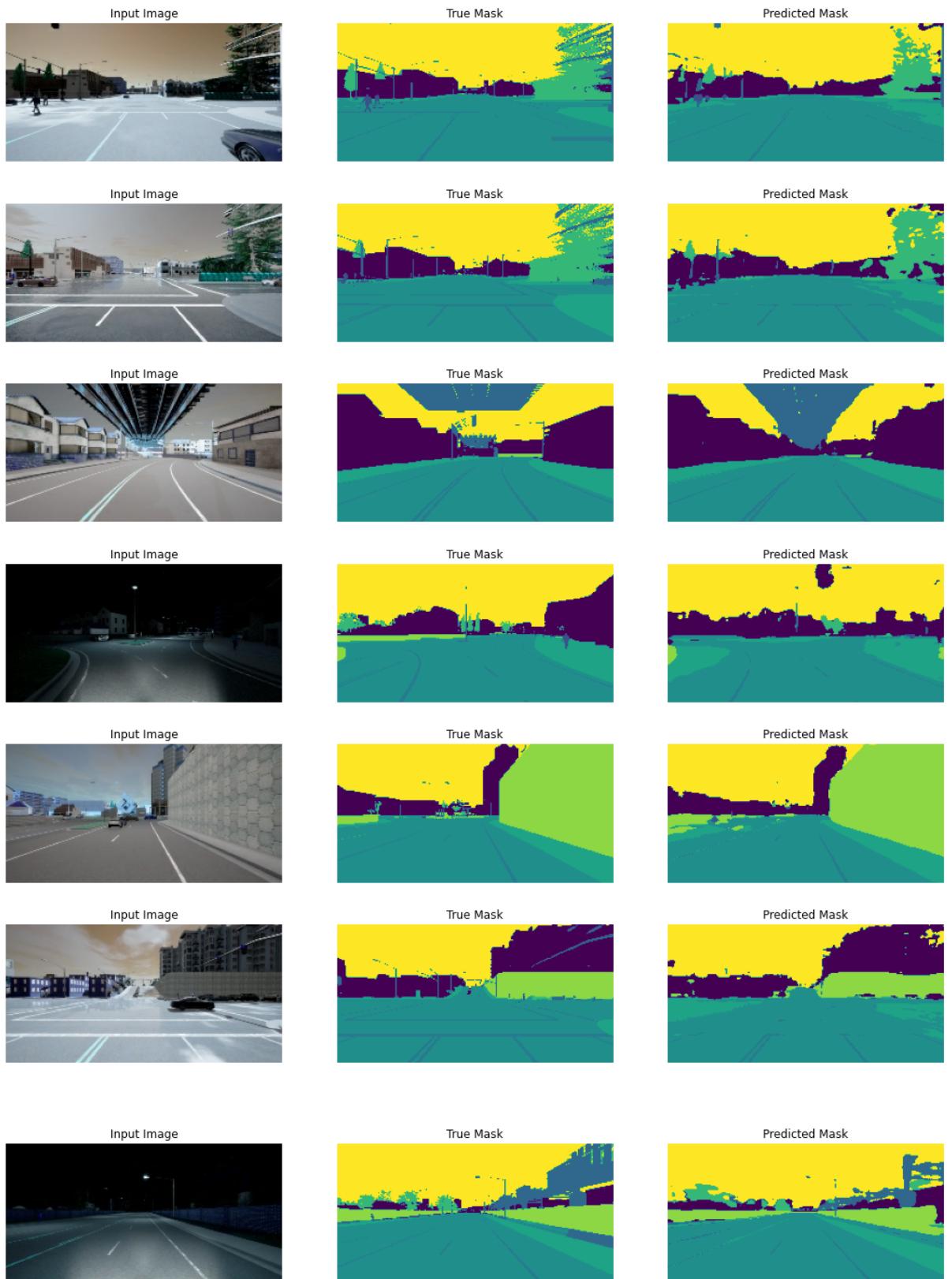


Mivel rendelkezésre állnak a mélységi képek is, ezért célszerű lehet azokat is a hálózat bemenetére adnunk, ezzel esetlegesen javítva a teljesítményt a hozzáadott információ által. Logikusan a hálózat meglévő bemenete mellett érdemes létrehozni egy újat, majd a két képet összevonni (konkatenálni) egy 4 csatornás képpé, majd ezt végigterjeszteni a hálózat többi részén. Fontos megemlíteni, hogy a tanító adathalmaz generálása során a mélységi képeket ugyanúgy kell transzformálnunk, mint az eredeti és a maszk bemeneteket: erre az albumations lehetőséget ad, hiszen megadhatunk maszkként több képet.

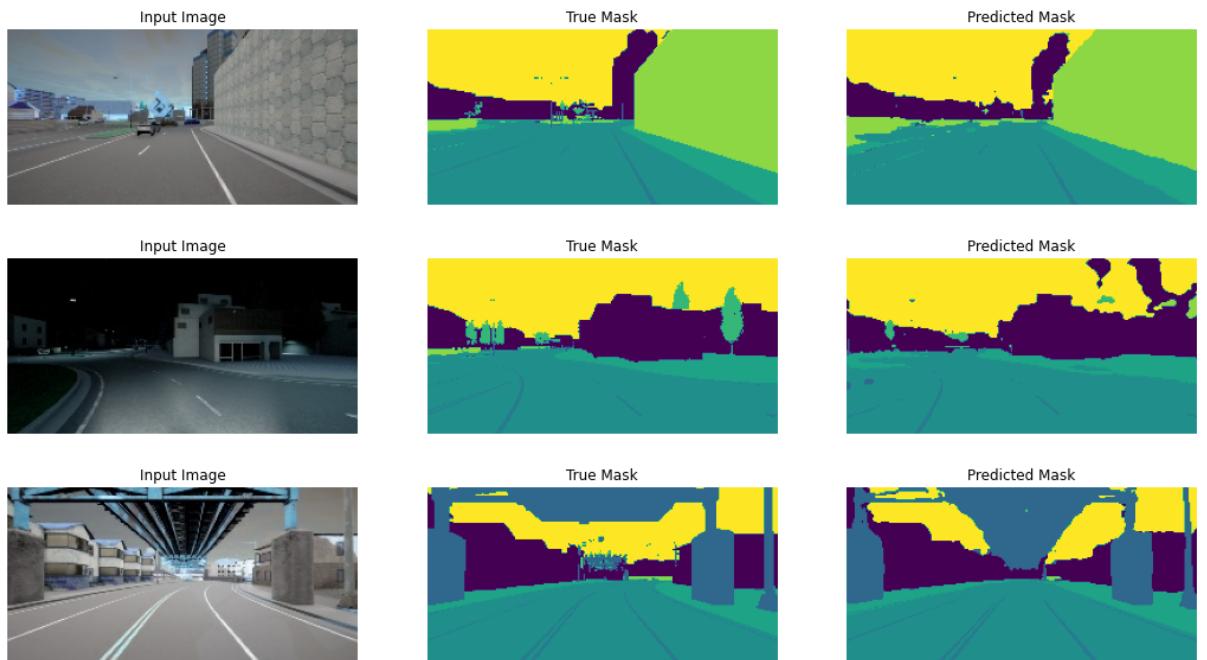
Másik fontos szempont a hálózat tanítása, ugyanis egy a feladaton jól teljesítő hálózat feltanítása még a kibővített adathalmazzal is nehézkes, így célszerű egy előre feltanított hálózatot felhasználni, és annak a súlyaiból kiindulni. Ilyen előre feltanított, szegmentálásra használt hálózatokat kínál a [Segmentation Models](#) program csomag, amely segítségével adott architektúrájú pretrained modellt kaphatunk, amit ezután tetszőlegesen módosíthatunk és taníthatunk. A rendelkezésre álló architektúrák közül az EfficientNet-et választottuk, mivel az megfelelő kapacitással bír a feladat megoldásához, azonban mégsem túlságosan bonyolult hozzá (10 millió paraméter). Ebben az esetben az U-Net architektúra enkóder része lesz a EfficientNet modellje az ImageNet adathalmazon való tanítás után kapott súlyokkal, míg a dekóder ugyanezt a sémát követi csak fordított rétegsorrenddel.

Kapott eredmények értékelése

Az így feltanított hálózat az alábbi eredményeket adta:



A tényleges és a kimenetként kapott maszkok a legtöbb esetben nagyon hasonlóak egymáshoz, egyedül az apróbb részletekben, kisebb objektumokban (pl. lámpaoszlopok) különböznek, ami ilyen méretű képeknél várható jelenség. Fontos megemlíteni, hogy bizonyos esetekben a Carla kimenetében fordulnak elő kisebb-nagyobb szegmentálási hibák (pl. az ábra 3. sorában lévő felüljáró nem teljes), így ezek is képesek rontani a háló teljesítményén. A hálózat által elérő pixelszintű validációs pontosság 88.18%, amelyet ha lebontunk osztályok szerint, akkor megfigyelhető, hogy valóban a nagyobb kiterjedésű objektumokat (ég, úttest, épületek) jól felismeri a hálózat, míg a kis kiterjedésű, ezért ritkábban előforduló osztályok esetében egy nagyságrrenddel rosszabbul teljesít a hálózat. Az átlagos IoU metrika alkalmazása esetében is hasonló jelenség tapasztalható: az összesített érték alacsony ugyan, de ha megvizsgáljuk az egyes osztályok értékeit, akkor ugyancsak a gyakrabban előforduló, szinte minden képen jelen lévő kategóriák kapnak nagyobb értéket (0.3-0.4 közötti), míg a kis méretű objektumok sokkal kisebb IoU értéket kapnak (0.05 körüli vagy még kisebb), így lehúzva az átlagot. Azonban a fenti képeket összehasonlítva elmondható, hogy a legtöbb esetben a hálózat sikeresen előállított egy a tényleges szegmentációhoz nagyon közel álló kimeneti képet, amelyen jól kivehetőek a felismerni kívánt objektumok.



A mélységi képeket hozzáadva a bemenethez az eredmények lényegében nem változnak (lásd fenti ábrákon), esetleg a képeken látható objektumok határai mosódnak el néhol, ami a hozzáadott információ miatt történhet.

Maga a hálózat futási időpont szempontjából jól teljesít: 0.64093 másodperc alatt végez a teljes validációs adathalmaz szegmentálásával, ami képenként 2.65946 ms-ot jelent.

Út szegmentáció

A feladat az úthoz tartozó pixelek elkülönítése minden más pixeltől. A várt kimenet pedig egy olyan kép, ahol az út pixelei fehérek, minden más fekete. A leírásban kép alatt természetesen a mélységi képet értem.

A feladat megoldásához RANSAC algoritmust használtam. Ez azt jelenti, hogy a képen egy olyan síkot próbáltam keresni, amihez a legtöbb pont tartozik. Természetesen az út geometriai elhelyezkedése alapján további feltételeket kellett szabni. Az algoritmus egy olyan feltételezéssel él, hogy az út a legnagyobb megadott orientációval rendelkező sík a képen. Az, hogy a kép mely pixeleire illeszt síkot az algoritmus az véletlenszám generálás eredménye. Az algoritmus részletes működése:

1. Legelső körben a teljes kép átalakítása történik képi koordinátákból világ koordinátákba a kamera mátrix segítségével a következő egyenletek felhasználásával:

$$\begin{aligned}x &= z * (u - px)/fx \\y &= z * (v - py)/fy\end{aligned}$$

Ahol u és v a kép pixelének a koordinátája a bal felső saroktól kiindulva. z a mélységi kép értéke az (u, v) pontban. A px, py, fx és fy pedig a kamera mátrix megfelelő elemei. A z világ koordináta megegyezik a mélységi kép értékével. Az eredményül kapott xyz pedig megadja minden pixel világ koordinátáját. Ezután már elkezdhető a sík illesztése.

2. Random pontok generálása. Ahhoz, hogy síkot lehessen illeszteni szükség van 3 pontra. 3 pont azt is jelenti, hogy erre biztosan lehet síkot illeszteni. Ennél a résznél azt vettetem figyelembe, hogy a kocsi előtt van a legnagyobb valószínűsséggel út, így a kép középső részének alsó felében generáltam ezeket a véletlen pontokat. A

területet a képen látható vastag piros keret jelöli. Természetesen ha a jármű rááll az előtte lévő autóra az így gondot okoz.



3. A sík paramétereinek kiszámítása. Miután kiválasztottam három véletlen pontot azok világ koordinátáira illesztettem síket a következő képletekkel:

$$\begin{aligned} a &= (By - Ay) * (Cz - Az) - (Bz - Az) * (Cy - Ay) \\ b &= (Bz - Az) * (Cx - Ax) - (Bx - Ax) * (Cz - Az) \\ c &= (Bx - Ax) * (Cy - Ay) - (By - Ay) * (Cx - Ax) \\ d &= - (a * Ax + b * Ay + c * Az) \end{aligned}$$

Ahol A, B, C jelöli a három különböző pontot, az utánuk írt x, y, z pedig jelöli, hogy az adott pont mely koordinátája szerepel a képletben. Innentől kezdve azt kell vizsgálni, hogy képnek mely pontjai elégítik ki a sík egyenletét.

$$a * x + b * y + c * z + d = 0$$

4. Meg kell vizsgálni, hogy a megtalált sík lehet-e az út. Ez a sík normálvektora alapján tehető meg. A normálvektornak alapvetően függőlegesnek kell lennie, y tengely irányába kell mutatnia. Az x irányba megengedett dőlés minimális lehet. Az út alapvetően nem dől nagy mértékben jobbra vagy balra. Z irányba a megengedett dőlés nagyobb a lejtők és emelkedők miatt. Számszerűen hatékony beállításnak bizonyult, ha

$$\begin{aligned} b &< -5000.0 \\ abs(a) &< 50 \\ abs(c) &< 1000 \end{aligned}$$

Addig keres a fenti paramétereknek megfelelő síket, amíg nem talál. Ez okozhat nemdeterminisztikus viselkedést, ugyanakkor ha

megengedjük, hogy olyan síkra is elkezdje az illeszkedő pontok vizsgálatát, ami nem felel meg ezeknek a paramétereknek, akkor előfordulhat, hogy akár 25 véletlen sík se felel meg az útnak és nem talál megoldást. Míg előre vizsgálva a sík orientációját, akár 1 megfelelő irányú sík is 95% feletti eredményt ad.

5. Ha megvan a sík, akkor meg kell vizsgálni mindegy egyes képpontra, hogy milyen távol van a síktól és hogy a megadott távolságon belül van-e. A síktól való távolság:

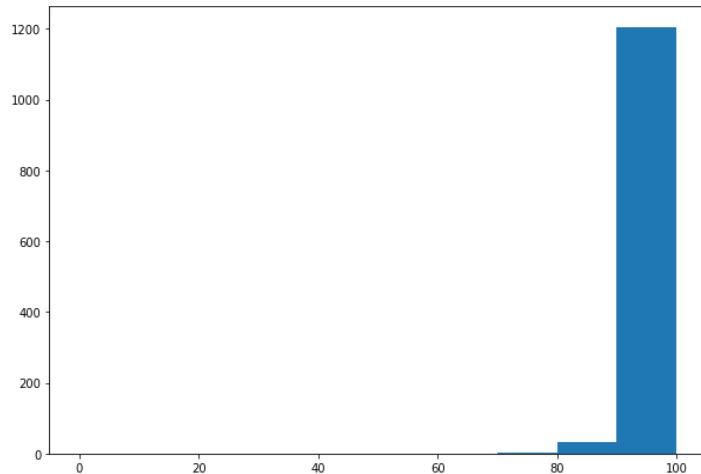
$$\frac{a*x+b*y+c*z+d}{\sqrt{a^2+b^2+c^2}}$$

Meg kell számolni, hogy az adott síkhöz így hány képpont tartozik. Több sík közül azt kell választani, amihez a legtöbb képpont tartozik. Még annyit figyelembe kell venni, hogy az égbolt végtelen távoli pontjai 0-s értéket vesznek fel a mélységi képen. Ezeket értelemszerűen ki kell hagyni.

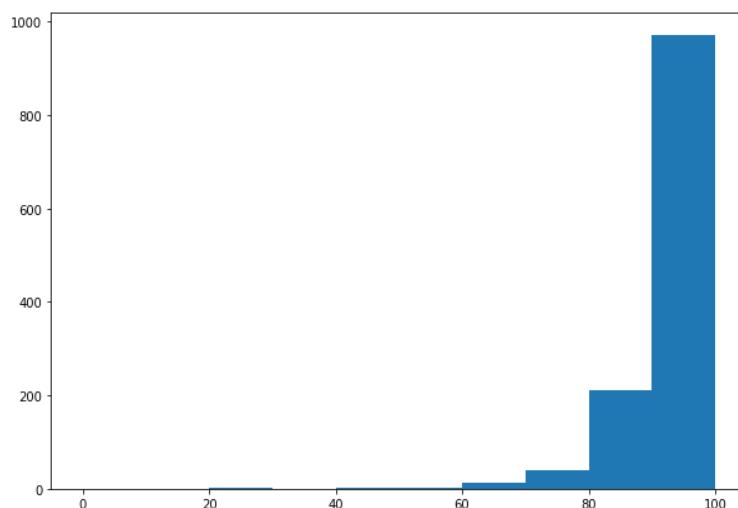
6. Végül a kiválasztott sík alapján osztályozni kell a pixeleket. Amik a síktól megengedett távolságra vannak azok 255-s értéket kapnak. Amik azon kívül, 0-s értéket kapnak.

Hátra van annak a meghatározása, hogy pontosan hány síkot próbáljon meg illeszteni egy képre, illetve milyen távolság legyen az, aminél a pontot még beleveszszük a síkba. Utóbbi értékének megválasztása fontos feladat, hiszen ha túl kicsit, akkor kihagy az úthoz tartozó pontokat. Ha túl nagy, akkor olyanokat is síkhoz vesz, ami nem tartozik az úthoz. A 7,5-ös távolság értéket próbálkozás alapján határoztam meg, ennél a számnál lett a legnagyobb a pontosság. Érdekes tapasztalat, hogy akkor éri el átlagban a legjobb eredményt az algoritmus, amikor csak egy síkot próbál illeszteni. minden más esetben több százalékkal alulteljesít az algoritmus.

Az eredmények értékelésére két fajta metrikát alkalmaztam. Első az elvárt kép és a kimeneti kép pixelenkénti összehasonlítása. Az algoritmus átlagos teljesítménye 96,85% körüli, legrosszabb képen 70-75% az egyezés. Pontos értékeket nehéz mondani, mivel véletlenszám generálást alkalmaz az algoritmus, így az átlag nagyon hasonló mindig. Azonban csak pár képnél (max. 10 kép) teljesít 80% alatt, így a minimum érték nagyon ingadozó. Eredmények eloszlása hisztogramon:

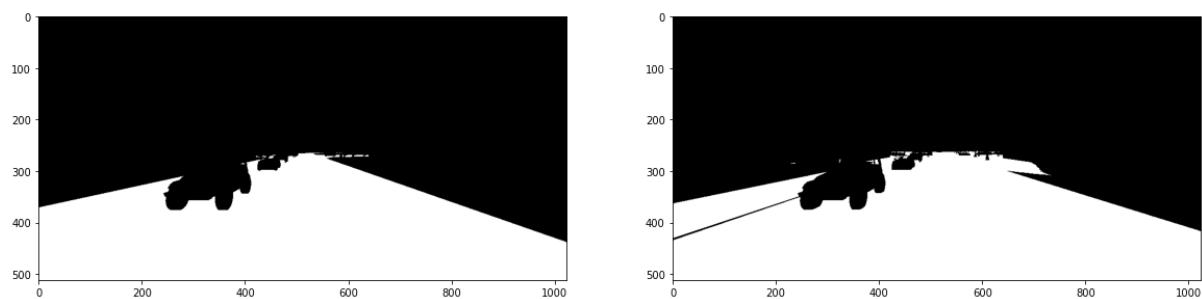


Másik mérték, amit használtam az az Intersection of Union. Ennél az eredmény nem olyan jó, de ez várható volt. Átlagosan 91,6% körül teljesít. A minimum érték viszont jelentősen ingadozik 30-50% között. A teljesítmény hisztogramon ábrázolva:

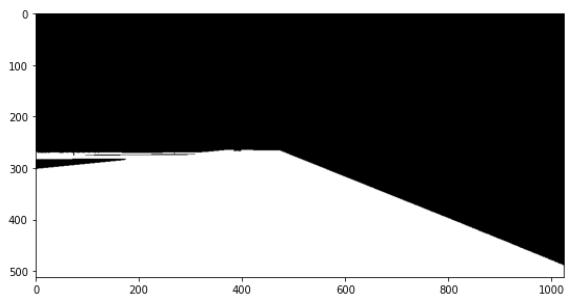
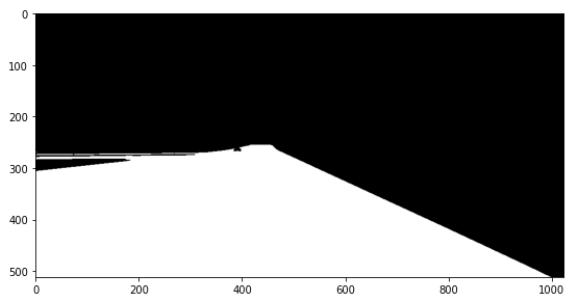
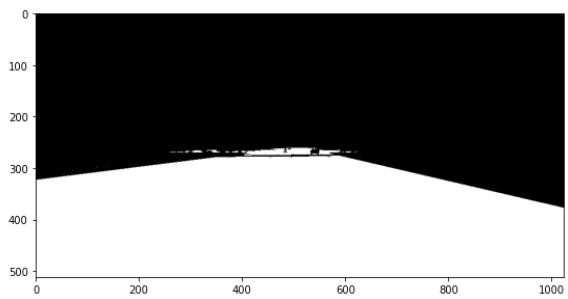
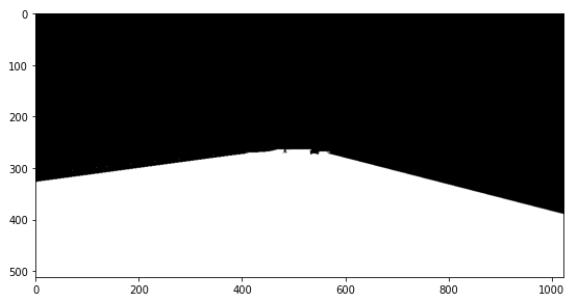
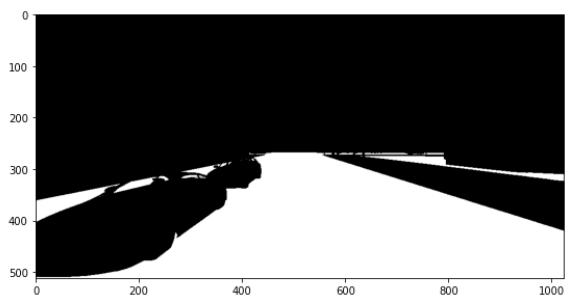
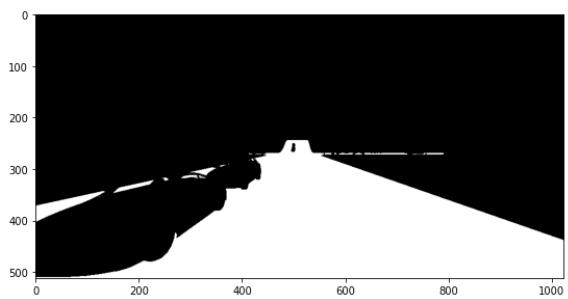
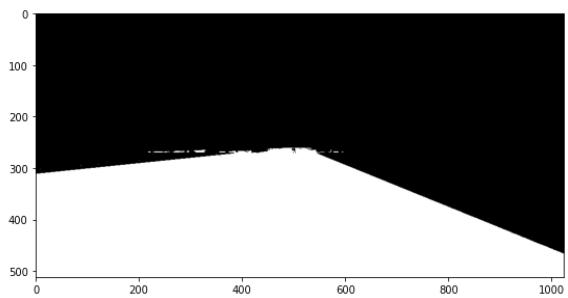
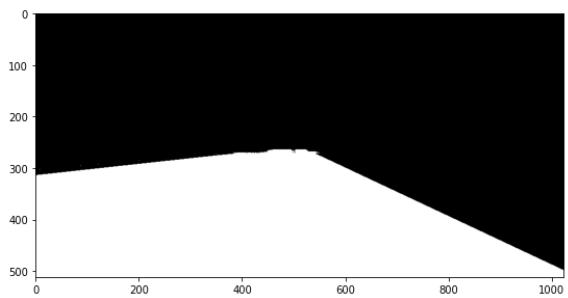
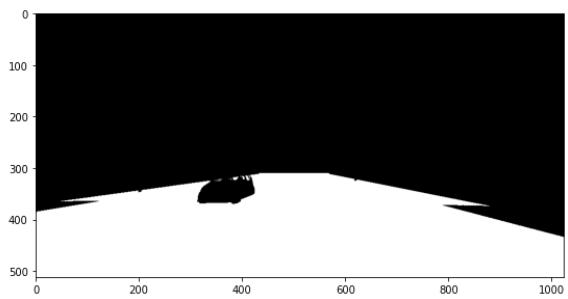
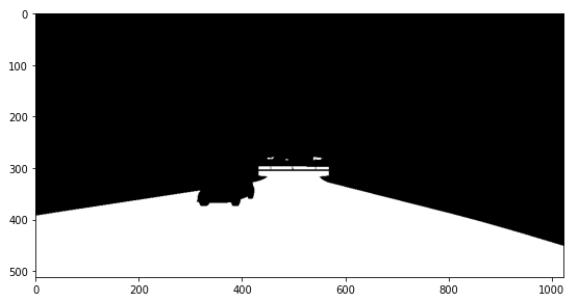


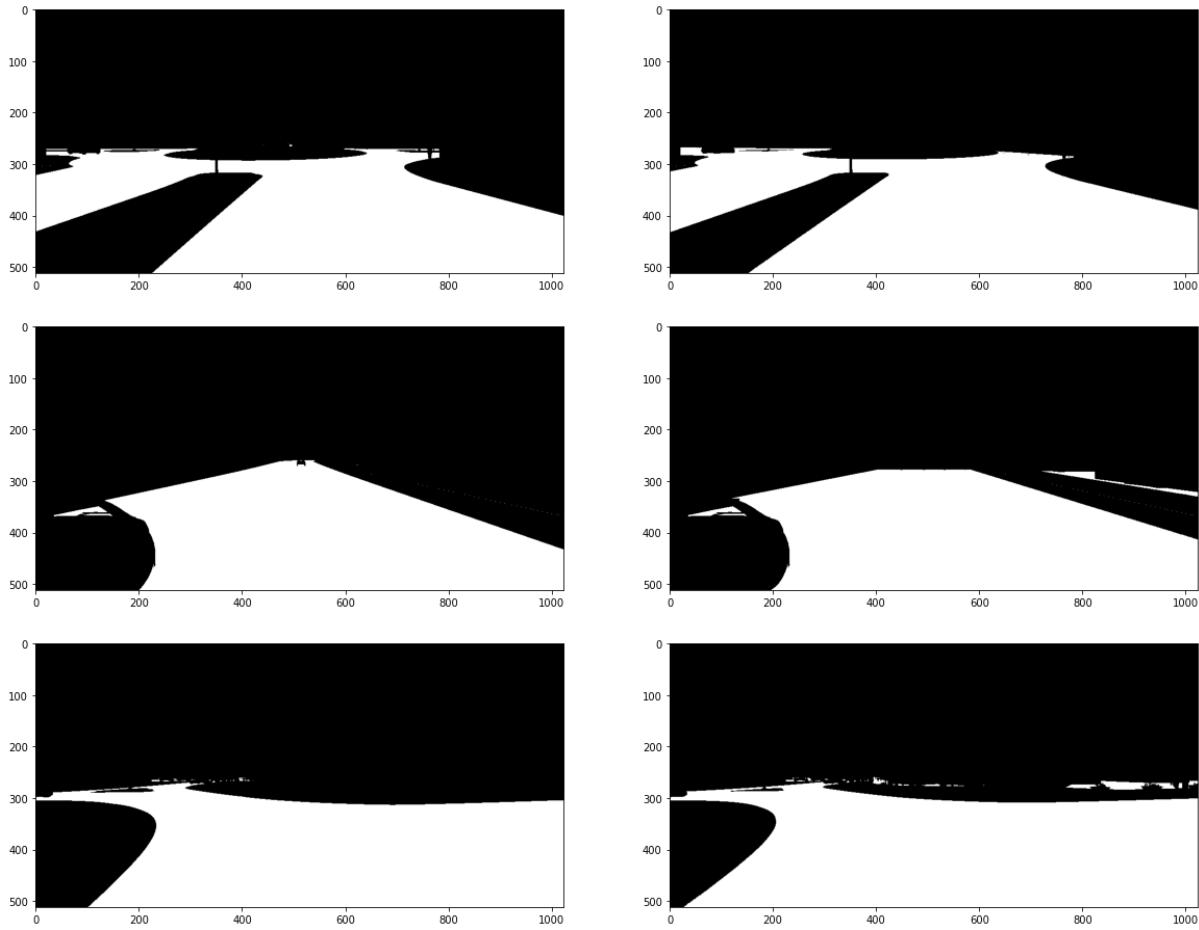
Az algoritmus mind a 1237 képen 42 másodperc alatt fut le. Ez azt jelenti, hogy átlagosan 34ms egy képen az út detektálása.

Az elvárt kimenet és a kapott kimenet egymás mellett:



Házi feladat dokumentáció





További fejlesztési lehetőségek a pontosság növelésére:

- Az egymás utáni szegmentált út képek összehasonlítása. Egyrészt ha a két kép között az úthoz tartozó pixelek száma jelentősen eltér, akkor lehet tudni, hogy nem sikerült jó síkot illeszteni. Ekkor lehet esetleg megpróbálni új síkot keresni. Egy másik lehetőség, hogy egy már kész út képen megbecsüljük mekkora valószínűsséggel lesz az a pixel a következő képen is az út része. Például ha egy adott pixel bizonyos sugarú környezetében csak út pixelek vannak, akkor nagy valószínűsséggel a következő képen is az lesz. Ehhez akár a jármű mozgás állapotát is fel lehet használni, abból nézni hogyan mozdulhatnak el az útpontok.
- RGB képeken végzett szegmentálással kombinálható, úgy valószínűleg még pontosabb eredményt adna.

Példa egy rossz képre, azonban ezek száma minimális:

Házi feladat dokumentáció

