Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Gróf Attila Bálint

Vezetést segítő funkciók fejlesztése okostelefonra mély tanulás alapon

KONZULENS

Dr. Gyires-Tóth Bálint Pál

BUDAPEST, 2018

Tartalomjegyzék

Összefoglaló 4
Abstract 5
1 Bevezetés 6
2 Irodalomkutatás 8
2.1 Konvolúciós neurális hálózatok vizualizációja és értelmezése 8
2.2 R-CNN 10
2.3 Yolo (You Only Look Once) hálózat 11
2.4 SSD (Single Shot Multibox Detector) hálózat 13
2.5 Yolo, SSD és R-CNN összehasonlítása 14
2.6 Önvezető autót irányító konvolúciós mély neurális hálózat 15
2.7 Vezetési stílus meghatározása 17
2.8 DeepDriving önvezető rendszer 18
2.9 Android operációs rendszer áttekintése 20
3 Rendszerterv 22
3.1 Okostelefon alkalmazás: DriverPhone 26
3.1.1 Videó kezelő modul 26
3.1.2 Metaadat gyűjtő modul 26
3.1.3 Mély neurális hálózat modul 27
3.1.4 Adatszinkronizációs modul 27
3.1.5 Jogosultság kezelő modul 27
3.2 Python alkalmazás 27
3.2.1 Adatfeldolgozó modul 28
3.2.2 Mély neurális hálózatot tanító modul 28
3.2.3 Adatszinkronizációs modul 28
4 Adatbázisok 29
4.1 COCO (Common Objects In Context) 29
4.2 VOC (The PascalVisual Object Classes Challenge) 2007, 2012 30
4.3 Kidolgozott adatbázis struktúra 30

5 Megvalósítás **34**
5.1 Okostelefon alkalmazás: DriverPhone 34
5.1.1 Felhasználói felület 34
5.1.2 Videó kezelő modul 35
5.1.3 Metaadat gyűjtő modul 37
5.1.4 Mély neurális hálózat modul 38
5.1.5 Adatszinkronizációs modul 43
5.1.6 Jogosultság kezelő modul 44
5.2 Python alkalmazás 46
5.2.1 Adatfeldolgozó modul 46
5.2.2 Mély neurális hálózatot tanító modul 50
5.2.3 Adatszinkronizációs modul 52
5.3 Megvalósítás során felmerülő nehézségek 52
6 Tesztelés és eredmények **53**
7 Kiértékelés **58**
8 Jövőbeli tervek **60**
9 Összefoglalás **61**
10 Irodalom jegyzék **62**

Összefoglaló

Napjainkban a mély neurális hálózatok egyre több szoftver funkciónak a hajtómotorjává válnak, ahogy egyre több területen újabb és jobban teljesítő modelleket alkotnak a kutatók és fejlesztők. A hálózatok erőforrás igényei nem csak tanítási időben magasak, de számos esetben egy komplex, nagy méretű hálózatban a predikció kiszámítása is jelentős számítási kapacitást igényelhet.

Dolgozatomban egy olyan szoftver keretrendszert dolgozok ki, amely egy általános megoldást ad olyan szoftver rendszerek felépítésére, ahol mély neurális hálózat szolgáltat felhasználói funkciókat. A rendszer működésének demonstrálására vezetést segítő funkciót valósítok meg objektum detektálással. Az objektum detektálás és önvezető rendszerek témakörében ismertetem a legfontosabb kutatási és ipari megoldásokat és ezen eredményekből ötletet merítve tervezem meg a rendszer architektúráját. A rendszer biztosítja a nyers adatok gyűjtését, tanító adatok előállítását automatizált módon, mély neurális hálózat tovább tanítását és predikciók számítását.

A megtervezett rendszer működésének bemutatására egy vezetést segítő rendszert építek fel, amely objektumokat detektál. Létrehozok egy Android alkalmazást, amely a kamera képén autókat, buszokat és jelző lámpákat detektál. A detektálásért felelős mély neurális hálózat a készülék processzorán fut. Az alkalmazás emellett képes videót rögzíteni metaadatokkal együtt és feltölteni ezeket egy központi tárhelyre. Az objektum detektálás során futásidőben lehetőség van a hálózat konfigurálására, ezzel biztosítva azt, hogy egy újabb, „okosabb” hálózatot lehessen beállítani. A szerver alkalmazás képes nyers adatot feldolgozni és automatizált módon tanítási adatot előállítani. Az objektum detektálásért felelős mély neurális hálózat a létrehozott tanító adatok segítségével tovább tanítható és az új „okosabb” hálózat egy webszerverről letölthető.

A kutatás és implementálás után az rendszer által elért eredményeket bemutatom és objektíven kiértékelem.

Abstract

Lately, deep neural networks are used to power more and more software features, as researchers and developers come up with new and more advanced models to use. The computational power required to use deep neural networks is very high during the training period, but it can be demanding during the inference phase as well.

In this paper, I design a complex software system that serves as an example architecture to systems, that use deep neural networks. To demonstrate the proper functioning of the system, I implement a deep learning based assistant by detecting objects. First I review all the necessary publications and best practices of the object detection and self-driving systems and with the inspiration, I construct the architecture of the system. The system provides the functionality of collecting raw data, generating training data, retraining the deep neural network and calculating predictions.

To prove the correct functioning of the system, I build a deep learning-based driver assistant system that detects objects. I create an Android application that detects cars, buses and traffic lights in picture of the phone's camera. The deep neural network that computes all the predictions, that run on the CPU of the smartphone. The application is also capable of capturing videos with metadata and uploading them to a remote server. During detection, it is possible to configure the deep neural network in case there is a newer, "smarter" model. The server application can process raw data and generate training data in an automated way. The deep neural network that performs the object detection can be retrained with the data generated by the server application. The retrained, "smarter" model can be downloaded from a webserver.

After the research and implementation, I review and draw conclusions from the results of the system.

1 Bevezetés

Az utóbbi években a telefonok a számítástechnika robbanásszerű fejlődésének köszönhetően teljes átalakuláson mentek keresztül. Ezen belül is az Android operációs rendszert futtató készülékek egyre több funkcióval rendelkeznek és a mindennapjainkba beépültek.

A mély tanulás világa hosszú múltra tekint vissza, azonban az utóbbi néhány évben a kutatások számos áttörést és eredményt értek el. Már a gyakorlati megvalósítást is lehetővé teszik a számítógépek számítási kapacitásának növekedése és grafikus kártyák párhuzamos számításának lehetősége. A képfeldolgozás területén 2012-ben volt egy nagy áttörés, amikor a kanadai kutatók megalkottak egy konvolúciós mély neurális hálózatot [1], amely az akkori megoldásokhoz képest kiemelkedő eredményeket ért el kép osztályozás területén.

Dolgozatom során egy olyan intelligens alkalmazás megvalósítását tűztem ki célul, amely vezetést segítő funkciókkal szolgál a sofőrnek. Egy olyan szoftver rendszert tervezek és valósítok meg, amely egy általános megoldást nyújt olyan rendszerekre, ahol mély neurális hálózatok is része a technológiai csomagnak. Ez jelen esetben autók, buszok és jelző lámpák detektálását jelenti, de a megoldás általános, más területeken is alkalmazható. A most következő években majdnem minden új autó úgy gurul ki a gyárakból, hogy számos érzékelővel rendelkezik, amelyek az önvezető funkció és egyéb kényelmi szolgáltatások megvalósításában elengedhetetlenek. A régi autók jellemzően intelligens szenzorok nélkül közlekednek, ezek működését lehetne segíteni egy olyan okostelefonos alkalmazással, amely vezetést segítő funkciókat lát el. Az okostelefonok ereje a hordozhatóságban van, így bármilyen típusú járművet vezetünk (autó, hajó, bicikli, repülő) az alkalmazás a járműhöz illeszkedő extra információkat szolgáltathat az utazás közben (például: színvakoknak a lámpa színének detektálása, motor hangjából előre jelezni a meghibásodást, beszéd felismerés). Jövőben az okostelefon akár a járművek vezetésében is nagyobb szerepet tölthet be a sok szenzor és erős hardvernek köszönhetően.

Az általam tervezett rendszerben a kamera képének a feldolgozását egy mély neurális hálózat a készüléken dolgozza fel a háttérben. Ennek előnye, hogy nincsen külső adatforgalom, minden adat a készüléken marad és internet elérés nélkül is működik az alkalmazás. Futás közben a felhasználó a kijelzőn kap egy visszajelzést, hogy milyen objektumokat detektál a készülék. Ez a működési mód nagy potenciált hordoz önmagában, mert a mély neurális hálózatok számos területen kiemelkedő eredményt képesek elérni (kép feldolgozás, természetes

nyelv feldolgozás, szenzor adatok értelmezése, felhasználói használatból adódó minták felismerése) és az okostelefonok folyamatosan fejlődő hardvere ezen a számítások elvégzését lehetővé teszik.

Az alkalmazás kiegészítéséhez egy szerver oldali keretrendszer fejlesztését is terveztem, amely az okostelefonos alkalmazásban futó mély neurális hálózat javítását valósítja meg egy távoli szerveren. A szerveren futó alkalmazásnak a feladatai közé tartozik a tanító adatok előállítása és a mély neurális hálózat tovább tanítása [2] (más néven transfer learning). A tovább tanítás lehetőséget nyújt arra, hogy a jövőben a hálózat pontossága javuljon vagy esetleg olyan osztályokra is detektáljon, amely azelőtt még nem volt elérhető.

Az okostelefon alkalmazás és a szerver program között két irányú kapcsolatra van szükség, az egyik az okostelefon által gyűjtött adatok továbbítása a szerver felé és a másik az új hálózatok letöltése a telefon alkalmazásba. Itt érdemes kihangsúlyozni, hogy ha nincs kapcsolat a szerver és alkalmazás között, akkor az okostelefon alkalmazás teljesen funkcionális, nem kapja meg az újabb „okosabb” mély neurális hálózatot és nem tud hozzájárulni a tanításhoz plusz nyers adatokkal.

Az alkalmazás és szerver közötti adat és modell szinkronizáció újabb fejlesztési lehetőségek előtt nyit kaput. A mély neurális hálózatokkal való munka egyik nehéz oldala a tanító adatok beszerzése. Sok adatelemzési területen vannak publikusan elérhető adatbázisok, azonban, ahogy a megoldandó probléma egyre konkrétabb, sok esetben nem áll rendelkezésre publikusan elérhető címkézett adatbázis hozzá. Adatok gyűjtése, címkézése és tanító adattá való alakítása már önmagában egy nagy feladattá képes kinőni magát. Az estek nagy részében a mély tanuló hálózatok csak annyira lehetnek jók, amennyire a tanító adat jó, így a jövőben a hálózatok tovább tanítása új adatokkal elengedhetetlenné válik. Amennyiben a tanító adathalmaz kisméretű vagy nem tartalmaz elég széles skálájú adatokat, a rendszer könnyen tanulhat meg olyan tulajdonságokat, amelyek csak a tanító adathalmaz miatt alakulnak ki.

Az általam tervezett rendszer ezekre a problémákra is mutat egy megoldást. A dolgozatban azt általam megálmodott szoftver rendszernek a felépítését, fejlesztését és kiértékelését mutatom be.

2 Irodalomkutatás

Ebben a fejezetben az önvezető autó és a mély tanulás témakörében ismertetem a munkám szempontjából legfontosabb kutatási és ipari eredményeket. A mély tanulás területéről az objektum detektálásra alkalmas hálózatokat vizsgálom meg, hogy az okostelefonos alkalmazásban az autók, buszok és jelző lámpák detektálása a lehető leghatékonyabb módon történjen. Az önvezető autók területéről olyan megoldásokból próbálok gondolatot meríteni, ahol a jármű irányítás hátterében mély neurális hálózat hozza a döntéseket.

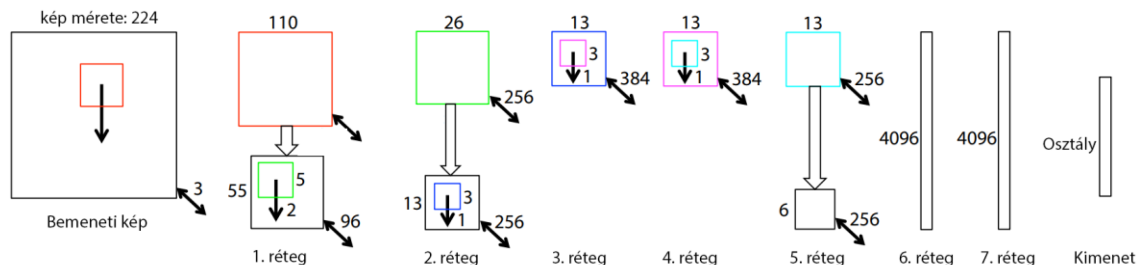
A mély tanulás világában minden hálózatnak megvannak a saját erősségei és gyengeségei is. A szempontok a modell kiválasztásához: a mérete ne legyen 50-100 MB-nál nagyobb, predikció kiszámítása egy mobil készüléken is kevesebb legyen, mint 1 másodperc és ezen feltételek mellett a pontossága a lehető legmagasabb legyen. Az objektum detektálás feladatra a mély neurális hálózatok közül a konvolúciós rétegekből építkező hálózatok bizonyultak eddig a leghatékonyabbnak, a pontosságukat általában az mAP (mean Average Precision) írja le, ezt részletesebben az 5.2.2. fejezetben mutatom be. A kutatások eredményében a mAP lokalizáció 50 % átfedés fölött számít helyesnek, amennyiben nincs más érték jelezve. Annak érdekében, hogy a különböző hálózatokat megértsük, először érdemes a konvolúciós hálózatok alapjait megismerni [1] [3]. A mély neurális hálózatok alapjaira ebben a dolgozatban terjedelmi okok miatt nem térek ki, csak az objektum detektáláshoz szorosan kapcsolódó módszereket, hálózat típusokat, valamint ezek egymáshoz való viszonyát mutatom be.

2.1 Konvolúciós neurális hálózatok vizualizációja és értelmezése

A konvolúciós neurális hálózatok már az 1990-es évek közepén mutatták a bennük rejlő lehetőségeket a kézzel írott számjegyek felismerése terén [4]. Az utóbbi években pedig ezeknél összetettebb feladatokban is kiemelkedő eredményeket értek el. Az R-CNN [8] és Yolo [9] típusú mély neurális hálózatok közel valós időben (számítógép GPU-n), magas pontossággal képesek képeken objektumokat detektálni és osztályozni. A nagyobb adatbázisok (tanító adatok, felcímkézve), a grafikus kártyák fejlődése, és az egyre jobb regularizációs technikák egyaránt hozzájárultak a konvolúciós neurális hálózatok jó eredményeihez. Az egyik elterjedt regularizációs technika a túltanulás ellen a dropout (kiejtés) használata, amely tanítás során ciklusonként valamilyen valószínűséggel egyes csomópontokat és kapcsolatait „kikapcsolja” a hálózatból [5]. Egy másik technika a batch (köteg) normalizálás [6] , amikor az adott ciklusban használt tanító adatok normalizálásra kerülnek és a skálázó és eltoló paramétereket is tanító

adatokká válnak. A hálózatok tanítása során a konvergenciát gyorsítja és a mély hálózatok jobban képesek tanulni. Ezeknek az egyre komplexebb modelleknek a belső működése nehezen érthető és vizualizálható.

A vizualizálás során a használt hálózatnak a bemeneti képek 224x224 pixel méretűek voltak és a mélységük az RGB színskála volt. Ezt követte 5 konvolúciós, 2 előre csatolt, és egy softmax réteg [7]. A hálózat felépítését az 1. ábra mutatja.



1. ábra: Konvolúciós hálózat felépítése (módosítva [7] alapján)

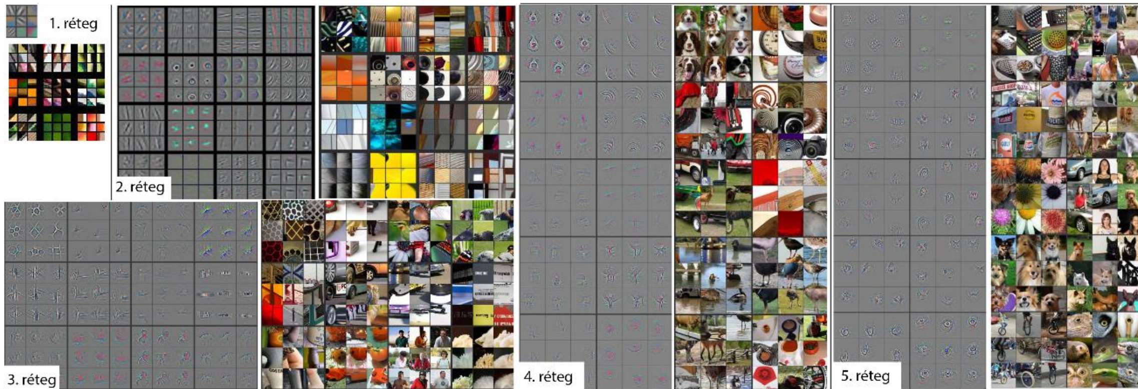
A hálózatot 1000 osztályra tanították (tehát 1000 különféle objektumot képes felismerni), tanítás után a teszt hibaszázaléka 14.8% volt, amely a nem konvolúciós hálózatok hibájának a fele. Minden esetben egy előtte semmilyen plusz információt nem tartalmazó hálózatot tanítottak. A középső két konvolúciós réteg és az előre csatolt rétegek eltávolítása esetében az eredmények drasztikusan romlanak, ez azt jelentheti, hogy a hálózat mélysége fontos a jó eredmény eléréséhez. A középső konvolúciós rétegek méretének növelése az eredmény javulásához vezethet, viszont ebben az esetben az előre csatolt rétegek is nőnek, amitől a hálózat túltanulhat [7].

A konvolúciós neurális hálózatok esetében a jellemzők vizualizációja csak az első réteg esetében triviális, itt a jellemzőket könnyen lehet a pixelekhez kötni. A szerzők egy módszert alakítottak ki arra, hogy milyen módon lehetne a többi rétegeben is megtalálni, hogy melyik pixel melyik jellemzőhöz tartozik. Erre a feladatra egy úgynevezett „dekonvolúciós hálózatot" hoztak létre, amely ugyanazokat az elemeket használja, azonban azok inverz változatát, ezáltal nem a pixeleket rendeli jellemzőkhöz, hanem fordítva [7]. A megoldások, amelyeket alkalmaztak:

- Unpooling: A Max Pooling nem invertálható, azonban a maximális értékek helyének eltárolásával meg lehet oldani ezt a problémát.
- Rectification: Rectifier linear unit (röviden ReLU) aktivációs függvény használata a helyes eredményért.

- Szűrés: A filterezés visszaalakításához az eredeti filterek transzponáltját használták.

A különböző rétegek által megtanult jellemzőket a 2. ábra mutatja.



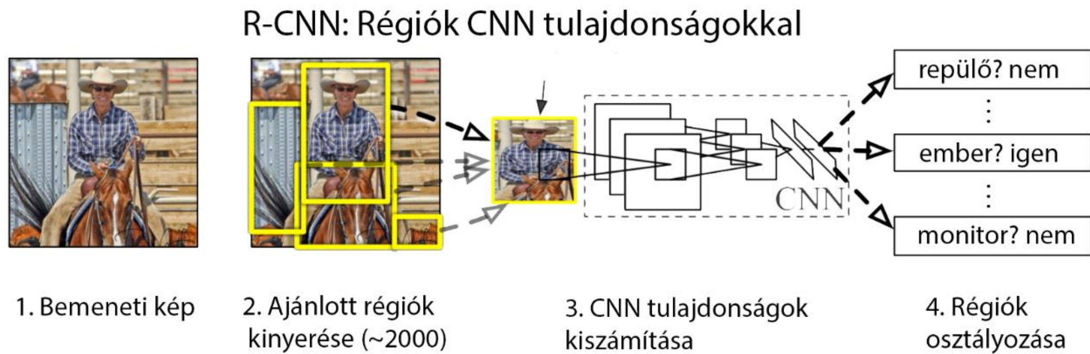
2. ábra: Rétegek által megtanult jellemzők (módosítva [7] alapján)

Ahogy a 2. ábra mutatja, a második réteg a sarkokat,valamint éleket tanulja meg, a harmadik réteg ennél már összetetteb alakzatokat/textúrákat tanul meg. A negyedik rétegben már az osztályoknak megfelelő mintázatok jelennek meg. Az utolsó ötödik rétegben már a felismerhető objektumokat lehet látni. Az alsóbb rétegek a jellemzőket már néhány tanítási iteráció után is megtanulják, azonban a felsőbb rétegeknek az összetettebb jellemzők megtanulásához jelentősen több időre van szüksége.

2.2 R-CNN

Az utóbbi években az objektum detektálásban egyre nagyobb sikerük van az RPM (Region Proposal Methods), valamint az R-CNN (Region Based Convolutional Neural Networks) jellegű hálózatoknak. Az R-CNN a legnagyobb előnye, hogy nagy mértékben párhuzamosítható, így grafikus kártyán közel valós idejű futást tesz lehetővé [8].

Az R-CNN egy olyan hálózat, amely egy képre sok négyszöget próbál illeszteni majd megvizsgálja azokat, hogy melyik helyen lehet objektum. A négyszögek helyét a kép tulajdonságai alapján generálja (pixelek), nagyjából 2000 darab ajánlott régiót generál. Ezek után minden egyes régiót megpróbál osztályozni. Amennyiben azok meghaladnak egy bizonyos határ százalékot, akkor ott egy objektum található [8]. A 3. ábra ezt a működést illusztrálja.



3. ábra: R-CNN működése (módosítva [8] alapján)

A hálózatnak van még egy nagy előnye, miszerint a futási időt nem befolyásolja szignifikáns mértékben az osztályok száma. Ezáltal olyan feladatokra is használható, mint amikor több száz külön osztályú objektumot keresünk egy képen.

Az R-CNN modellel 58.5 mAP-t sikerült elérniük a kutatóknak a VOC 2007 teszt adatbázison úgy, hogy a hálózat elő volt tanítva az ILSVRC 2012-es adatbázison és tovább lett tanítva (transfer learning) a VOC 2007-es adatbázisán. Az alábbi táblázat mutatja az összehasonlítást egy másik modellel, amelyet alapvonalnak tekinthetünk.

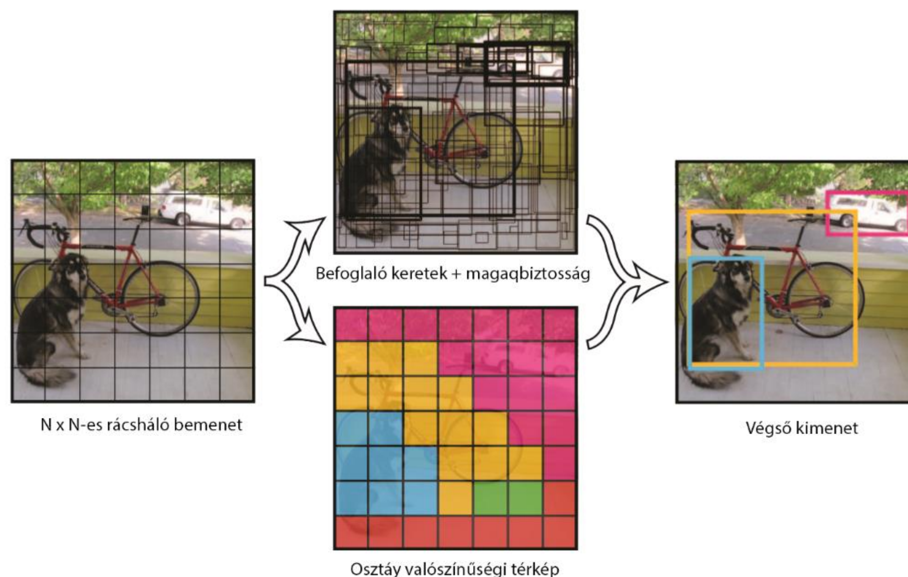
Modell	Adatbázis	mAP
R-CNN FT fc_7 BB	VOC 2007	58.5
DPM(Deformable Part Models) HSC	VOC 2007	34.3

1. táblázat: R-CNN VOC 2007 adatbázison eredmények (módosítva [8] alapján)

Az 1. táblázatból látható, hogy az R-CNN nagyon jó eredményt ér el az alapvonalhoz képest.

2.3 Yolo (You Only Look Once) hálózat

A Yolo egy olyan konvolúciós mély neurális hálózat, amely nem a pontosságot, hanem a sebességet célozta meg fő feladataként. A hálózat, az emberi érzékeléshez hasonlóan adott régióit nézi meg a képnek és ez alapján dönti el, hogy ott található-e objektum. Ezt a műveletet sokszor, különböző méretekkel teszi meg, így képes megtalálni a lehető legtöbb méretű és osztályú objektumot a képen [9]. Abban az esetben amikor több egymást átfedő azonos objektumot talál, akkor ezeket a keretek unióját veszi és az jelöli majd az objektumot. Ezeket a lépéseket az 4. ábra illusztrálja.



4. ábra: Yolo hálózat által detektált keretek összefésülése (módosítva [9] alapján)

A hálózat 448 x 448-as képeket vár és a kimeneten megadja, hogy melyik régióban milyen valószínűséggel talált objektumot, illetve annak a befoglaló keretek a koordinátáit is. A hálózat legnagyobb előnye, hogy az R-CNN hálózatnál akár ~100-szor is gyorsabb lehet. Ez lehetővé teszi a valós idejű objektum detektálást [9]. A sebesség a könnyen a pontosság rovására mehet, az alábbi táblázat a Yolo teljesítményét hasonlítja össze a hasonló tulajdonságú hálózatokkal.

Modell	FPS (Frames Per Second)	mAP
100 Hz DPM	100	16.0
Fast Yolo	155	52.7
Yolo	45	63.4
Fast R-CNN	0.5	70.0
Faster R-CNN VGG-16	7	73.2
Yolo VGG-16	21	66.4

2. táblázat: Yolo összehasonlítása más hálózatokkal (módosítva [9] alapján)

A 2. táblázat mutatja, hogy a Yolo futása megközelíti a valós idejű feldolgozást, azonban az R-CNN pontossága alatt marad. A Yolo és az R-CNN hiba százalékában még arra érdemes említésre kitérni, hogy a Yolo pontosságra majdnem eléri az R-CNN szintjét, viszont a lokalizáció pontossága jóval rosszabb.

A munkám készítése során a Yolo v2 [10] és Yolo v3 [11] is megjelent, s ezek alapján sok területen fejlődést értek el a kutatók a modellel.

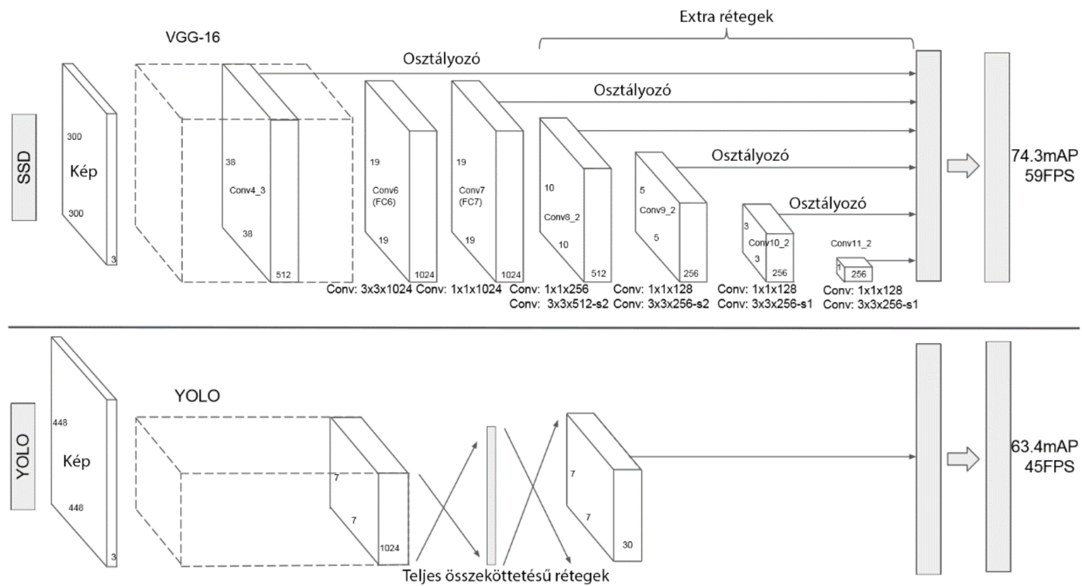
A Yolo v2 hálózat az előző verzióhoz képest a konvolúciós rétegekben kapott normalizálást, illetve a tanítást 2 ciklusban végezték, először az osztályozó hálózatot 224x224-es képekkel, majd az egész objektum detektáló hálózatot 448x448-as képekkel tanították. A befoglaló keretek is a véletlen méretezés helyett egy 5 db előre meghatározott méretarányos keretet kapnak [10].

A Yolo v3 már három különböző szinten valósít meg detektálást, amely segíti a kisebb objektum detektálását is, amely a Yolo v2-nek gyengesége volt. Öt helyett kilenc detektáló keretet használ. A Yolo v2 416x416-os bemenet esetében 845 detektált keret volt képes detektálni, míg a Yolo v3 már a három szintnek köszönhetően már 10 647 darab keretet képes detektálni [11].

2.4 SSD (Single Shot Multibox Detector) hálózat

Az SSD modell [12] alapvetően egy előre csatolt konvolúciós hálózat, amely fix méretű objektum kereteket ad meg, s ezután minden egyes prediktált kerethez egy értéket ad az objektum osztály előfordulásának valószínűségére. Az egymást átlapoló kereteket a hálózat összefésüli egy végleges keretbe, illetve, ha több osztály takarja egymást akkor több végső keret keletkezik.

A hálózat struktúrája egy tradicionális konvolúciós hálózat a kezdete, amit a hálózat alapjának neveznek. Ehhez az alap hálózathoz adnak további rétegeket, melyek méretben jóval kisebbek, ezáltal futásuk is jóval gyorsabb lesz. Ezek a rétegek a detekciót állítják elő különböző méretekben. Emellett minden réteg képes objektumokat detektálni is. Ezt az 5. ábra felső részén, a hálózat tetején láthatjuk.



5. ábra: SSD és Yolo hálózati topológia összehasonlítása (módosítva [12] alapján)

Az SSD 300x300-as méretű képeket vár a Yolo 448x448-as méretéhez képest. Azonban a kisebb méret nem jelenti azt, hogy a pontosság kevésbé lesz jó.

Modell	Adatbázis	mAP
Yolo	VOC 2007 + VOC 2012	57.9
SSD300	VOC 2007 + VOC 2012	72.4
SSD300	VOC 2007 + VOC 2012 + COCO	77.5
SSD512	VOC 2007 + VOC 2012	74.9
SSD512	VOC 2007 + VOC 2012 + COCO	80.0

3. táblázat: Yolo és SSD mAP összehasonlítás (módosítva [12] alapján)

Ahogy a 3. táblázat mutatja az SSD pontosságban jóval felül múlja a Yolo hálózatot.

2.5 Yolo, SSD és R-CNN összehasonlítása

Az eddig bemutatott modellek mind alkalmasak objektum detektálási feladatra. A kérdés az, hogy pontosságban, futási időben és komplexitásban hogyan viszonyulnak egymáshoz. A dolgozatom során egy olyan hálózatra van szükség, amely a lehető legkisebb futási idővel rendelkezik a predikció számításánál, de emellett a pontossága elfogadható. Az összehasonlítást a 4. táblázat szemlélteti.

Modell	FPS	Bemeneti kép felbontása	mAP
Faster R-CNN (VGG-16)	7.0	1000x600	73.2
Fast Yolo	155.0	448x448	52.7
Yolo (VGG16)	66.4	448x448	66.4
SSD300	74.3	300x300	74.3
SSD512	76.8	512x512	76.8

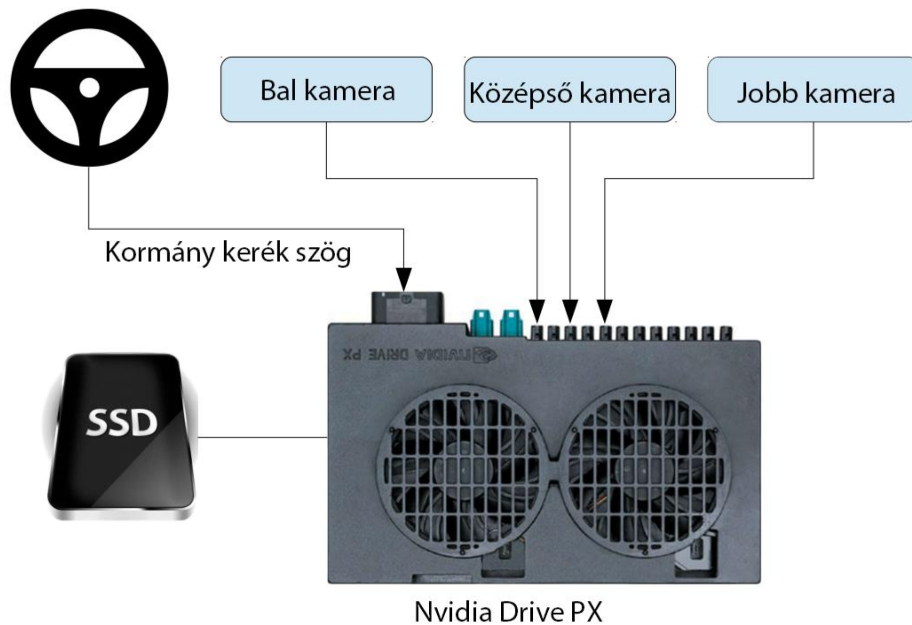
4. táblázat: SSD összehasonlítása a többi modellel (módosítva [12] alapján)

A 4. táblázat mutatja, hogy a leggyorsabb futási idővel a Fast Yolo rendelkezik, azonban a pontossága elmarad a többitől. Az SSD300 futási ideje azonban még mindig 60 FPS fölött van, emellett a pontossága is a legjobbak között van. Továbbá említésre méltó, hogy a cikk írói a méréseket egy Titan X-en futtaták, amely egy átlagos videókártya sebességének a többszörösére képes. A fejlesztési feladatokhoz az SSD 300-as hálózatot fogom használni. A választása oka a gyors futás, a pontossága is elfogadható és emellett a TensorFlow keretrendszerben könnyű ezzel a hálózattal dolgozni.

2.6 Önvezető autót irányító konvolúciós mély neurális hálózat

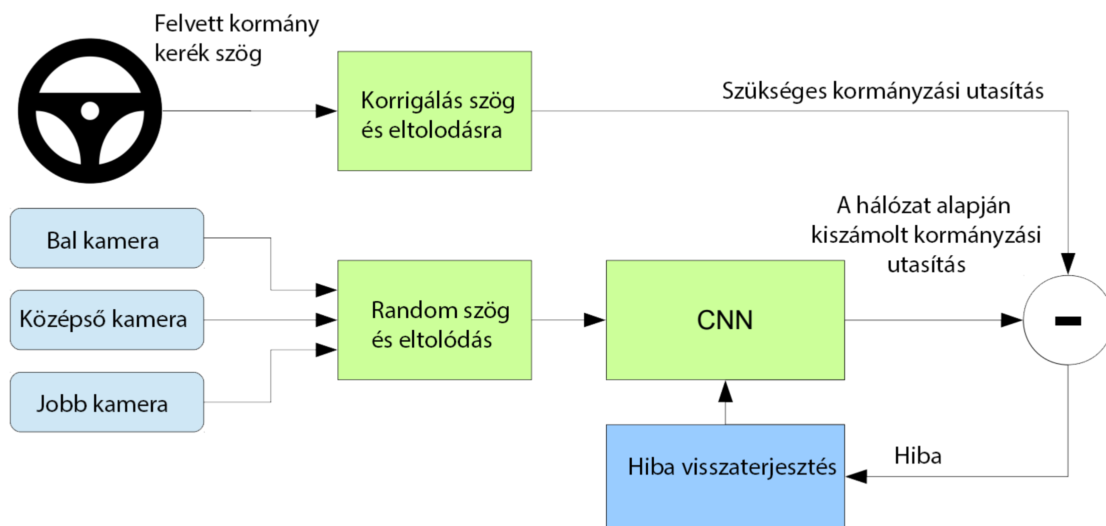
A konvolúciós hálózatok nem csak objektumok detektálására, hanem egy jármű vezetésére is lehet használni. Az Nvidia cég egy olyan megoldást mutatott be, amely egy konvolúciós mély neurális hálózatot használt az autó kormányának irányítására [13]. A konvolúciós hálózat bemenete egy kamera képe volt, amely a járműből előre nézett.

Az általuk megalkotott rendszert három részre lehet felbontani: adatgyűjtő rendszer felépítése, tanítási fázis és a predikciók számolásának fázisa. Az adatgyűjtő fázist a 6. ábra, a tanítást a 7. ábra, valamint a predikció számolását a 8. ábra mutatja be.



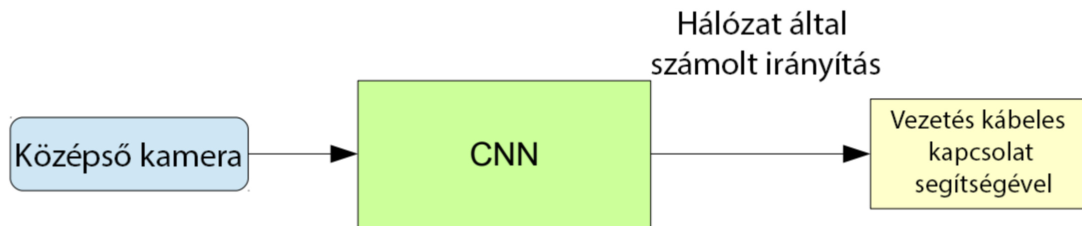
6. ábra: Adat gyűjtés (módosítva [13] alapján)

Ahogyan a 6. ábra mutatja ebben a fázisban az adatok gyűjtése a cél, amely három kamera segítségével és a kormánykerékből kapott szögelfordulásból tevődik össze. Az így nyert adatokat egy Nvidia kártya dolgozza fel, majd menti ki egy SSD háttértárra. A következő lépés a hálózat tanítása.



7. ábra: Tanítás (módosítva [13] alapján)

A 7. ábra mutatja be a tanítás menetét, a tanítás középpontjában egy konvolúciós neurális hálózat van amelynek a bemenetére kerülnek adatok, ezek átmennek a hálózaton majd a kimeneti adat összehasonlításra kerül az elvárt adattal, az eltérésből származó hiba visszaterjesztésre kerül a hálózatba és ilyen módon tanul a hálózat [13].



8. ábra: Predikció számolása (módosítva [13] alapján)

A 8. ábra mutatja a predikció kiszámolását. A konvolúciós hálózatnak egy bemenete van, ami a középső kamerából érkezik, ennek alapján számolja ki kívánt kormánykerék állást. A hálózat eredménye egy vezetékes kábel segítségével jut el az irányításhoz és a kormány állása is módosul [13].

A hálózat hatékonyságát szimulátorban és valódi forgalomban is tesztelték.

9. ábra: Konvolúciós hálózat eredménye a valóssal szemben szimulátor környezetben (módosítva [13] alapján)

A 9. ábra a szimulátorban a valós kormányszög állást és a mély neurális hálózat által prediktált értéket mutatja. A kutatók ~ 15 kilométert mentek az autóval autonóm módon úgy, hogy nem kellett közbeavatkozni egyszer sem [13].

2.7 Vezetési stílus meghatározása

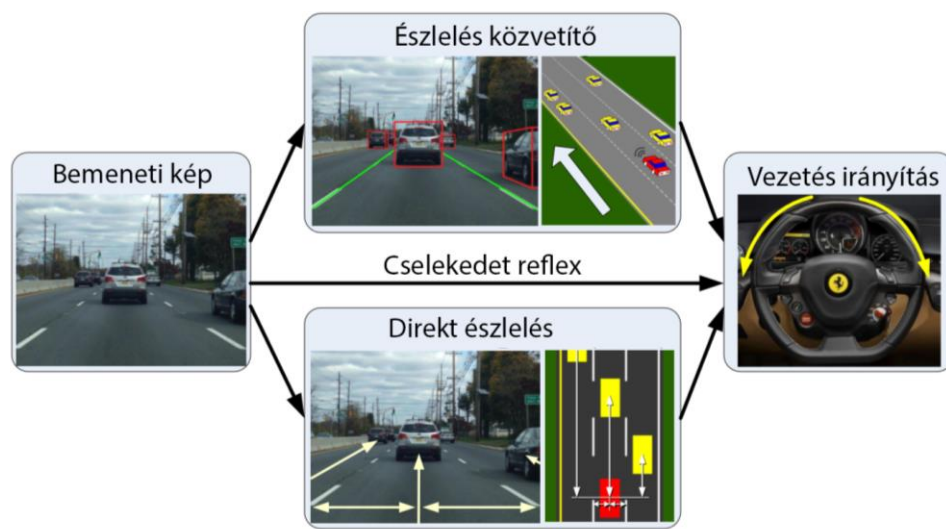
Az utakon rendkívül sok baleset történik és ezeknek visszaszorítása fontos feladat. Egy spanyol egyetem kutatói egy olyan rendszert terveztek meg és implementáltak, amely egy okostelefon alkalmazás segítségével képes osztályozni a sofőr vezetési stílusát [14]. Az alkalmazás számos szenzor adatot gyűjt, majd azt egy távoli szerverre feltölti és egy mély neurális hálózat a kapott adatokból osztályozza a sofőrt. A háttérben az osztályozásra egy előre

csatolt neurális hálózatot használtak. A hálózat bemenetei a sebesség, gyorsulás és a többi szenzorból keletkező adat volt. A sofőröket három osztályba sorolták: csendes, normál és agresszív.

A hálózatot 0.4-es tanítási együtthatóval (learning rate) 5000 cikluson keresztül tanították és a végén 0.43 MSE-t (átlagos négyzetes hiba) értek el. A sikeres tanítás után a hálózatot átkonvertálták C kóddá és beépítették a webalkalmazásban, amely az osztályozást végezte.

2.8 DeepDriving önvezető rendszer

A DeepDriving rendszert [15] amerikai egyetem kutatói alkották meg. Az önvezető autók világában két önvezető vezetés megközelítés volt: a észlelés közvetítő (mediated perception) és a cselekedet reflex (behavior reflex). A kutatók egy harmadik megoldáson dolgoztak, amelyet direkt észlelésnek (direct perception) neveztek. A három megközelítést a 10. ábra mutatja.

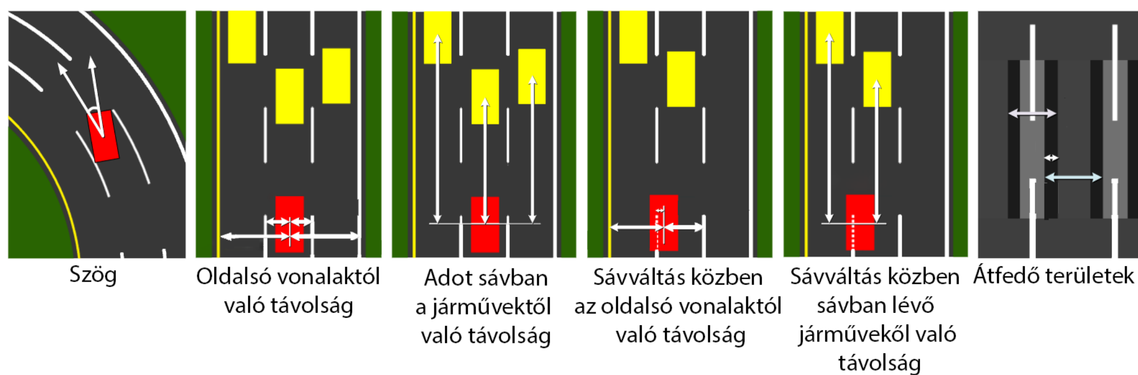


10. ábra: Önvezetés elvi megközelítése (módosítva [15] alapján)

Az észlelés közvetítő megközelítés úgy működik, hogy rendszer komponensei érzékelnek objektumokat, amelyeket hatással lehetnek a járműre (például: autók, jelző lámpák, sávok, stb…). A felismerések egy világ reprezentációba kerülnek beépítésre, ahogy az a 10. ábra felső dobozában látható. Az irányításhoz ezeket az információkat összegyűjtve egy mesterséges intelligencián alapuló program fogja meghozni a döntéseket. Azonban a vezetéshez sokszor elég lenne csak a jármű szempontjából szigorúan szükséges objektumokat figyelembe venni, ezért ez a megoldás sok esetben növeli a döntéshozatal bonyolultságát.

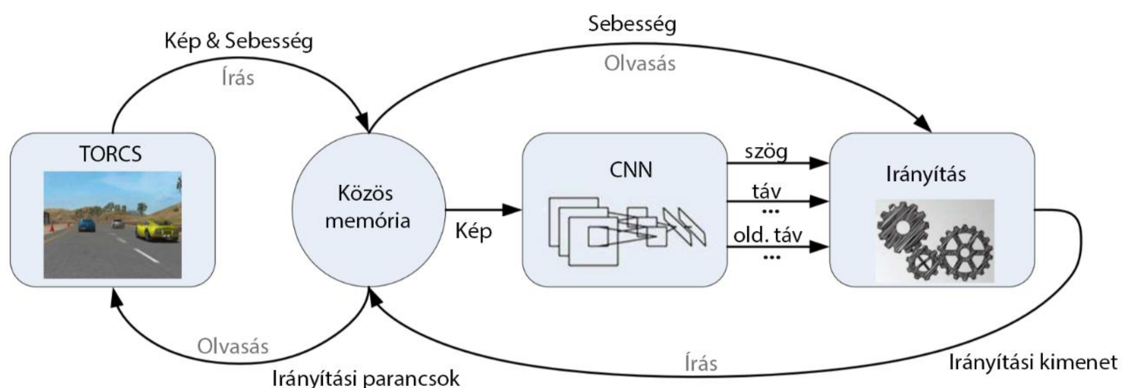
A cselekedet reflex, amely a 10. ábra középen helyezkedik el, egy direkt összeköttetést alkot a szenzorokból és kamerákból kapott adat és a végrehajtott irányítási döntés között. A leképezést mély neurális hálózatokkal oldották meg már a 1980-as évektől kezdve.

A kutatók által megalkotott új megközelítés, a kettő eddig bemutatott megoldások közé esik. Ezeknél a kamera képe alapján nyernek ki olyan észleléshez szükséges indikátorokat, amelyek az úton való elhelyezkedéshez relevánsak. Ilyen indikátor például a jármű szöge az úthoz képest, az elválasztó vonalaktól való távolság, távolság a többi autóhoz képest és ezekhez hasonlóak. Tehát nem a világot próbálják reprezentálni annak egy virtuális leképezésével, hanem a jármű szemszögéből építenek fel egy relatív világot, ezt a 11. ábra mutatja.



11. ábra: Direkt észleléshez szükséges információk (módosítva [15] alapján)

A 11. ábra által bemutatott indikátorok, amelyekre az általuk terveztett rendszernek szüksége van. Szimulálásra a TORCS (The Open Racing Car SWimulator) nevű programot használták. A kép feldolgozására az AlexNet konvolúciós mély neurális hálózatot alkalmazták Caffe keretrendszerben.



12. ábra: DeepDriving architektúra (módosítva [15] alapján)

A 12. ábra mutatja be az általuk megvalósított architektúrát. A folyamat során a konvolúciós hálózat feldolgozza a TORCS-ból kapott képet majd tizenhárom indikátort számol ki. Ezen értékek és a sebesség alapján az irányítás meghozza a döntéseket, amelyek a TORCS felé kerülnek visszaküldésre.

Néhány eredményt a tesztelésből a 5. táblázat szemléltet.

	Szög	Bal oldali sáv távolság	Jobb oldali sáv távolság
Caltech lane	0.048	1.179	1.084
ConvNet full (DeepDriving)	0.025	0.197	0.179

5. táblázat: DeepDriving eredménye az alapvonalhoz képest (módosítva [15] alapján)

Az 5. táblázatban a szög radiánban, a többi érték pedig méterben értetendő és az értékek az átlagos abszolút hibát mutatják. Az eredményekből látszik, hogy a konvolúciós hálózattal megvalósított rendszer az alapvonalhoz képest jobban teljesített, következésképp nagy potenciált rejlik benne.

2.9 Android operációs rendszer áttekintése

Az Android egy mobilkészülékekre készített operációs rendszer, amely a Linux kernel módosított verziójára épül [17]. A rendszert először csupán okostelefonokra tervezték, azonban azóta már kiadták okosórára, autóba, TV-re és tabletre szánt verzióját is. 2005-ben a Google megvette az Android Inc. nevű céget és 2008-ban már saját termékükként adták ki az Android operációs rendszer első, publikus verzióját.

A legelső Android-ot futtató okostelefon a HTC Dream (G1) volt. 2008 óta a rendszer rendkívül nagy ütemben fejlődött és új funkciók százait kapta meg évente. A Google nagyjából évenként adja ki nagyobb frissítését a rendszernek. A legutolsó kiadott verzió az Oreo, de 2018 végén már a Pie becenévre keresztelt verzió is elérhető lesz.

A Pie verzióban a mesterséges intelligencia került a középpontba, a bemutatója során főleg a mesterséges intelligencia által meghajtott funkciókat demonstrálta a Google. Ezek között szerepelt intelligens akkumulátor optimalizálás, proaktív asszisztens funkciók, valamint a kamera kép feldolgozásánál több kamera helyett a szoftverre összpontosítanak. Érdekes módon a Google által fejlesztett TensorFlow gépi tanulásos könyvtár nem kapott natív támogatást a Pie rendszerben. Ezzel szemben az Apple iOS-nek van hivatalos támogatása, amely TensorFlow modelleket az Apple saját Core ML modell formátumra konvertálja.

Az Android operációs rendszer különböző szintekből áll össze. A felépítését a 13. ábra illusztrálja. A Linux kernel a legelső réteg, ez felel az alap funkcionalitásokért. A különböző hardver elemekhez itt találhatóak meg a driver-ek (wifi, kijelző, kamera, stb…) és emellett a készülék erőforrás menedzselésért felelő kódok is itt helyezkednek el. Efölött található a HAL, amely a hardware közeli nyelv és a Java között fordító szerepet tölt be. A natív C/C++ könyvtárak leginkább a készülék grafikus megjelenítésében vesznek részt, ideértve az OpenGl ES, Webkit és egyéb grafikus motorokat. Emellett helyezkedik el az Android Runtime, amely a virtuális gépet biztosítja minden egyes applikációnak. A Java API ezen rétegek fölött helyezkedik el és a rendszer funkcióihoz szolgáltat interfészeket a fejlesztők számára. Végül a legfelső szinten a rendszer alkalmazások és a fejlesztők által írt alkalmazások vannak [16].

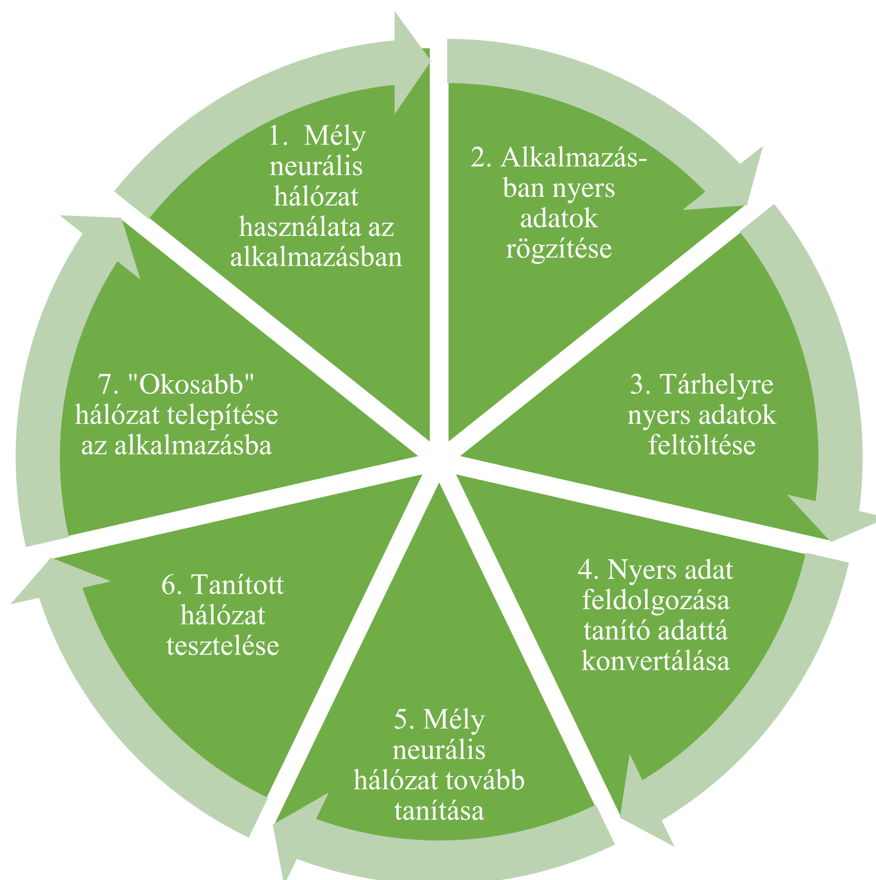
13. ábra: Android operációs rendszerének architektúrája (módosítva [16] alapján)

A gazdag Android API-nak köszönhetően, z Google Play Store boltban már több mint 3.800.000 db alkalmazás érhető el [17].

3 Rendszerterv

Ebben a fejezetben bemutatom az általam tervezett rendszer architektúra tervét és a hozzá tartozó folyamatokat. Először bemutatom a célokat, amelyek az architektúra megalkotását befolyásolták, majd a működést megvalósító modulok szerepét részletezem.

Egy olyan rendszer tervezését tűztem ki célul, amely vezetést segítő funkciókkal szolgál a felhasználóknak. A felhasználói funkciók mellé egy olyan szoftver ökoszisztémát képzeltem el, amely biztosítja, hogy a felhasználó funkciókat meghajtó mély neurális hálózat továbbfejlesztése lehetséges legyen. Ezeket a célokat és kapcsolatukat egy folyamatábrával részletesebben mutatom be.



14. ábra: Rendszer folyamatai

A 14. ábra mutatja be a rendszer folyamatait. A képen látható szeletek tartalmazzák azokat a részletes célokat, amelyeket az általam tervezett rendszernek meg kell valósítania. A célok részletesebb kifejtése alább olvasható.

Mély neurális hálózat használata az alkalmazásban:

- Az alkalmazásnak képesnek kell lennie mély neurális hálózat segítségével képen található objektumok detektálására. A hálózat számításait a készülék processzorán kell elvégezni.

Alkalmazásban nyers adatok rögzítése:

- A készített alkalmazásnak tartalmaznia kell funkciót videó és egyéb metaadatok rögzítésére. A metaadatok között a legfontosabbak a készülék szenzorjai által gyűjtött adatok.

Tárhelyre nyers adatok feltöltése:

- A készüléken gyűjtött nyers adatokat egy távoli tárhelyre fel kell tölteni, mindezt úgy, hogy a háttérben történjen és csak wifi kapcsolaton keresztül, ezzel a mobil internetes forgalmat kímélve.

Nyers adat feldolgozása és tanító adattá konvertálása:

- A mobil készüléken gyűjtött nyers adatot automatizált módszer segítségével fel kell tudni dolgozni és olyan struktúrában tárolni, amelyet később könnyen fel lehet használni. A feldolgozott adatot új tanító adattá alakítani automatizált módon mély neurális hálózat segítségével.

Mély neurális hálózat tovább tanítása:

- A szerver oldalnak lehetőséget kell biztosítania a meglévő mély neurális hálózat tovább tanítására a rendszer által előállított új tanító adatokkal.

Tanított hálózat tesztelése:

- A tovább tanított hálózat teljesítményét tesztelni kell, hogy meg lehessen állapítani, hogy jobb predikciókat számol-e, mint a hálózat előző verziója.

"Okosabb" hálózat telepítése az alkalmazásba:

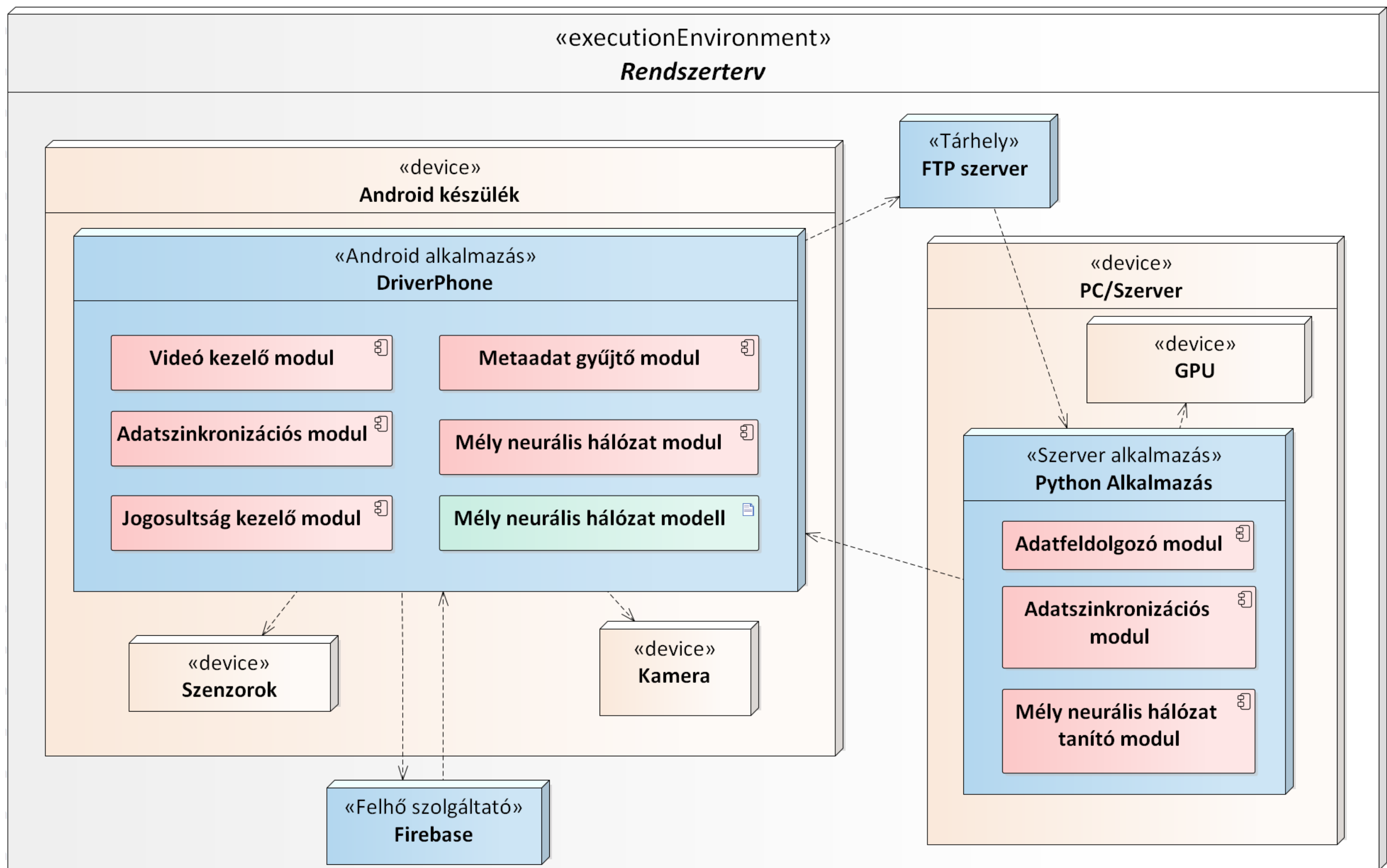
- Az új „okosabb” hálózatot valamilyen módon vissza kell szinkronizálni a már telepített applikációba úgy, hogy azt az alkalmazás használni legyen képes.

Ezeknek a céloknak a megvalósítását lehetővé teszi az az architektúra, melyet a 15. ábra mutat be.

A rendszer tervezésénél figyelembe kellett venni az elérhető erőforrásokat és ezek közötti kompatibilitási kérdéseket. Számomra a készülékek közül egy Nokia 7 Plus és régi iPhone 4s állt rendelkezésre, de Macintosh számítógép hiányában (ami elengedhetetlen iOS fejlesztéshez) az alkalmazás platformjának az Android-ot választottam. Mély neurális hálózatok építésében és futtatásában segítő könyvtáraknak támogatása Linux alapú rendszereken magasabb szintű, mint Windows operációs rendszeren, így a szerver oldali komponensek Linux környezetben fognak futni. A távoli tárhely választásánál az FTP tárhelyet választottam, mert Python és Java kódból is könnyen lehet fájlokat manipulálni rajta és vannak ingyenesen elérhető tárhelyek is. A

jogosultság kezelésre saját megoldás helyett a Google által megvásárolt Firebase szolgáltatót választottam 4 okból: könnyű integrálni Android alkalmazásba, az Android Studio-ba van beépülő plugin-ja, 1000 felhasználóig ingyenes és offline beléptetést is biztosít. A szerver oldalon a mély neurális hálózat tanítását és tesztelését egy Nvidia 940MX segíti. Az előbb említett videó kártya 384 CUDA maggal, 2 GB DDR3 memóriával és 0.9539 TFLOPS (FP32 (float)) számítási kapcaitással rendelkezik. Hasonlításképpen az egyik legelterjedtebb Nvidia kártya mély tanulás területen használt a GeForce GTX 1080 Ti, amely 11.340 TFLOPS (FP32 (float)) számítási kapacitása van. Az rendszerben a 940MX fogja segíteni a tanítást, azonban ez a megoldás skálázható, akár több videókártyát is lehet integrálni a rendszerbe.

A rendszer több komponensből és eszközből épül fel, amelyeknek a kapcsolatát a 15. ábra illusztrálja.



15. ábra: Rendszerterv

A 15. ábra mutatja az általam tervezett rendszer architektúráját. A rendszer 2 nagyobb komponensre osztható fel: Android alkalmazás és a szerver oldal. Az Android applikációt „DriverPhone”-nak neveztem el, ettől kezdve így is fogok hivatkozni rá. A komponensek és a modulok közötti nyilak az adat áramlást vagy használatot reprezentálnak. Az architektúra ábra célja a rendszer felépítésének átláthatósága és az esetleges hibák korai felderítésére szolgált a fejlesztés során. A következő alfejezetben a rendszerben szereplő modulok részletes bemutatása következik.

3.1 Okostelefon alkalmazás: DriverPhone

Több modulból álló Android alkalmazás, amely vezetést segítő funkciókat szolgáltat felhasználóinak. Ezek a funkciók autók, buszok és jelző lámpák detektálást jelentik, amelyeket a felhasználóknak a kijelzőn kell mutatni. Emellett videó rögzítésére, felhasználó beállításokra, adat szinkronizálásokra és mély neurális hálózat konfigurációjára is lehetőséget ad.

3.1.1 Videó kezelő modul

A videó kezelő modulnak elsődleges feladata, hogy a kamera képét élőben mutassa a felhasználónak megfelelően a képernyő méretéhez képest. A kamera képének a tájoláshoz képest megfelelően kell állni, mivel az okostelefonokban sok esetben más rotációval telepítik a kamera modult.

A kamera kép mutatása mellett a modul képes videót rögzíteni legalább 640x480-as felbontásban és a felvétel indítását a felhasználó kezdeményezheti és állíthatja le. A rögzített videó az alkalmazás saját mappájában kerül tárolásra.

3.1.2 Metaadat gyűjtő modul

Az okostelefonok számos szenzorral vannak felszerelve és emellett a beépített hardverekről is sok futás idejű információ beszerezhető. Ennek a modulnak a feladata minél több úgynevezett metaadatot gyűjteni az alkalmazás futása és videó rögzítése alatt. Az alkalmazás futása közben rögzítendő metaadatoknak az alkalmazás teljesítmény kiértékeléséhez szolgálnak statisztikai alapot, míg a videó rögzítése közben gyűjtött adatok a mély neurális hálózat tanításban segíthetnek később.

A kihívás az adatok gyűjtésben Android készüléken az a különböző gyártók különböző szenzoraiból ered. Előfordulnak olyan hardverek, ahol a visszakapott értéket transzformálni

kell, de olyan is megesik, hogy két szenzor teljesen különböző metrikát ad vissza két különböző gyártójú készülék esetén.

Olyan metaadatokat is érdemes gyűjteni, amelyek éppen a fejlesztés pillanatában nem szolgálnak többlet információval, de később értékes lehet új modell tanítása során vagy új funkció fejlesztéséhez.

3.1.3 Mély neurális hálózat modul

Az Android alkalmazással szemben a legfontosabb követelmény, hogy képes legyen olyan mély neurális hálózatot futtatni, amellyel a kamera képét lehet elemezni. Ezek első sorban konvolúciós hálózatokat jelentenek. A hálózatok számításait a készüléken kell elvégezni, hogy az internet kapcsolat nélkül is funkcionális legyen. A mély neurális hálózat által kiszámított predikciókat valamilyen módon a felhasználónak jelezni kell.

3.1.4 Adatszinkronizációs modul

Ennek a modulnak egyik fő feladata a videó kezelő modul és a metaadat gyűjtő modul által létrehozott fájlok egy távoli tárhelyre való feltöltése. Az adatok feltöltésére annyi korlátozás vonatkozik, hogy csak wifi hálózaton keresztül és a háttérben történhet. A másik fontos feladata, hogy előre megadott címről egy mély neurális hálózat modell fájlt képes legyen letölteni és ezt eltárolni a háttértáron.

3.1.5 Jogosultság kezelő modul

A jogosultság kezelő modul a nevéből is kikövetkeztethető módon a felhasználó kezeléssel foglalkozik. A modulnak biztosítani kell lehetőséget új felhasználó regisztrálására, kilépésére és elfelejtett jelszó esetén új jelszó kérésére. Emellett még a bejelentkezett felhasználónak internet elérése nélkül is bejelentkezve kell maradnia.

3.2 Python alkalmazás

A Python alkalmazás egy távoli szerveren fut, Linux operációs rendszer alatt. Az alkalmazás feladata, hogy a felhasználói funkciókat megvalósító mély neurális hálózat tovább tanítása lehetséges legyen. Ehhez szükség van tanító adatok előállítására feltöltött videókból, tanításra és egy webszerverre, ahonnan az alkalmazás az új modellt képes letölteni.

3.2.1 Adatfeldolgozó modul

Az adatfeldolgozó modulnak a feladata a tanító adatok előállítása. A Python alkalmazás nyers videó fájlokat kap, amelyeket át kell alakítani képkockákra. Ezeken a képkockákon meg kell állapítani, hogy hol milyen objektum osztályok találhatóak, majd a tanításhoz szükséges adatokat létrehozni a megfelelő formátumban.

3.2.2 Mély neurális hálózatot tanító modul

A modul alkalmas SSD MobileNet hálózat tanítására, mégpedig úgy, hogy a tanítással kapcsolatos konfigurációs beállítások szabadon szerkeszthetőek. Ez magába foglalja azt is, hogy lehetőség van egy már korábban tanított hálózatot új tanító adatokkal tovább tanítani (transfer learning). Majd az „okosabb" hálózat .pb vagy .tflite típusba exportálni.

3.2.3 Adatszinkronizációs modul

A Python alkalmazásnak képesnek kell lennie fájlokat letölteni egy FTP tárhelyről és ezeket háttértárra elmenteni. Ezen kívül egy webszervert kell biztosítania, ahonnan az Android alkalmazás képes lesz majd letölteni a mély neurális hálózat újabb verzióját.

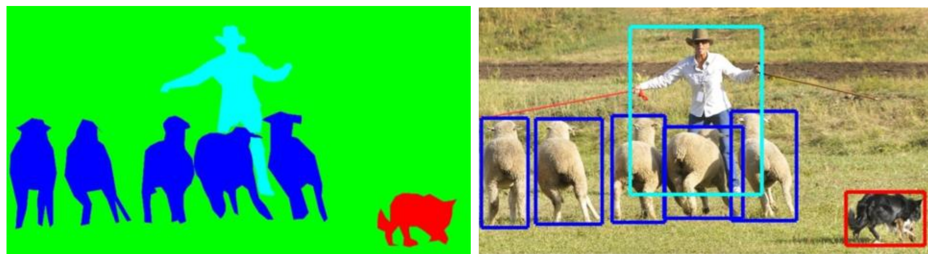
4 Adatbázisok

A COCO és VOC adatbázisokon mérik a kutatások során megalkotott új mély neurális hálózatok teljesítményét az objektum detektálás témakörében. Az általam használt hálózatokat a COCO és VOC adatbázisokon tanították. A SSD MobilNet hálózatot az általam létrehozott adatbázison tanítottam tovább.

4.1 COCO (Common Objects In Context)

A COCO [18] egy nagyméretű adatbázis, amely olyan feladatokra tartalmaz adatokat, mint az objektum detektálás, szegmentáció vagy akár a kulcspont keresés. A COCO nem csak adatbázist, de API-kat is tartalmaz, megkönnyítve ezzel a használatukat. Jelenleg Python, Matlab és Lua környezetekben érhetőek el az API-k.

A legújabb 2017-es COCO adatbázis 123.287 db képet tartalmaz, amelyeken összes 86.284 osztály példány jelenik meg. A rendkívül sok kép közül minden egyes kép fel van címkézve.



16. ábra: Szemantikus szegmentálás (balra) és objektum detektálás (jobbra) [18]

A 16. ábra bal oldalán látható egy példa a szemantikus szegmentálásra. A szemantikus szegmentálás egy olyan feladat, ahol a kép minden pixelét egy osztályba kell sorolni. A lényege a képi tartalom modellezése, a kép a szegmentálás után sokkal könnyebben értelemezhető.

A jobb oldalán az objektum detektálásra látható egy kép. Az objektum detektálás feladata a képen található különböző osztály példányainak meghatározása, azok befoglaló keretével.

A különböző feladatok között a COCO adatbázison való teljesítmény egy nívós mutató a mély neurális hálózatok között.

4.2 VOC (The PascalVisual Object Classes Challenge) 2007, 2012

A VOC [19] kihívás feladata előre nem feldolgozott képeken objektum detektálás. Három részből épül fel ez a kihívás:

- Osztályozás/Detektálás: Az osztályozás esetében egy adott képre meg kell mondani annak osztályát. Objektum detektálás esetében pedig a képen található osztály példányainak a befoglaló téglalapját kell megadni.
- Szegmentálás: Pixelekénti osztályozás, mint a COCO adatbázisban.
- Akció osztályozás: Az adott képen történő esemény/akciónak kell a megmondani az osztályát.

A jelentkezők akár mind a három kihívásra jelentkezhetnek. A kihívás célja, hogy minél több képet címkézzenek föl és ezzel növelni tudják a címkézett képi adatbázisukat. A VOC-n tanított hálózatok eredményei is mérvadók, amikor 2 hálózat teljesítményét hasonlítják össze.

4.3 Kidolgozott adatbázis struktúra

A munkám során a céljaim között volt az is, hogy a magyarországi utakról és közlekedésről egy adatbázist kezdjek el építeni. A dolgozat készítése során nem az adatbázis méretére helyeztem a hangsúlyt, hanem annak struktúrájára és készítésének a folyamatára.

Az nyers és a feldolgozott adatok külön mappastruktúrába kerülnek:

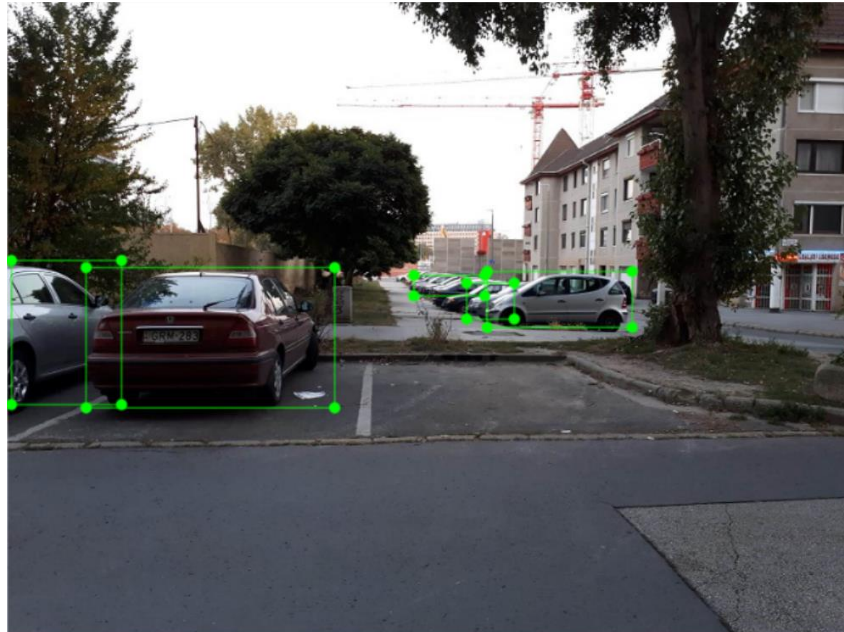
- Eredeti videó és a hozzátartozó metaadatok.
- Képkockák a hozzá tartozó detektált objektumokkal xml fájlban.

Első ránézésre így az adat duplikálva van, azonban mind a két tárolásnak más célja van. Nagy adatmennyiség esetén a feldolgozás hosszú napokat is igénybe vehet, így ezeket az adat feldolgozási lépéseket csak egyszer kellene megtenni.

Az eredeti videó fájlra olyan feladatoknál lehet szükség, ahol egyik képkockáról a másikra valamiféle előrejelzést szeretnénk számolni vagy valamilyen mozgás modellezésére. A metaadatok tartalmazzák a GPS koordinátákat, ennek segítségével a videó készítőjének a mozgását a térképen is lehet követni párhuzamosan a videó képével.

A képkockák a videó fájlokból állnak elő és automatizált módon egy xml-ben a képeken található objektumok helyzete és azok osztályai találhatóak meg. Az xml-ek is egy mély neurális

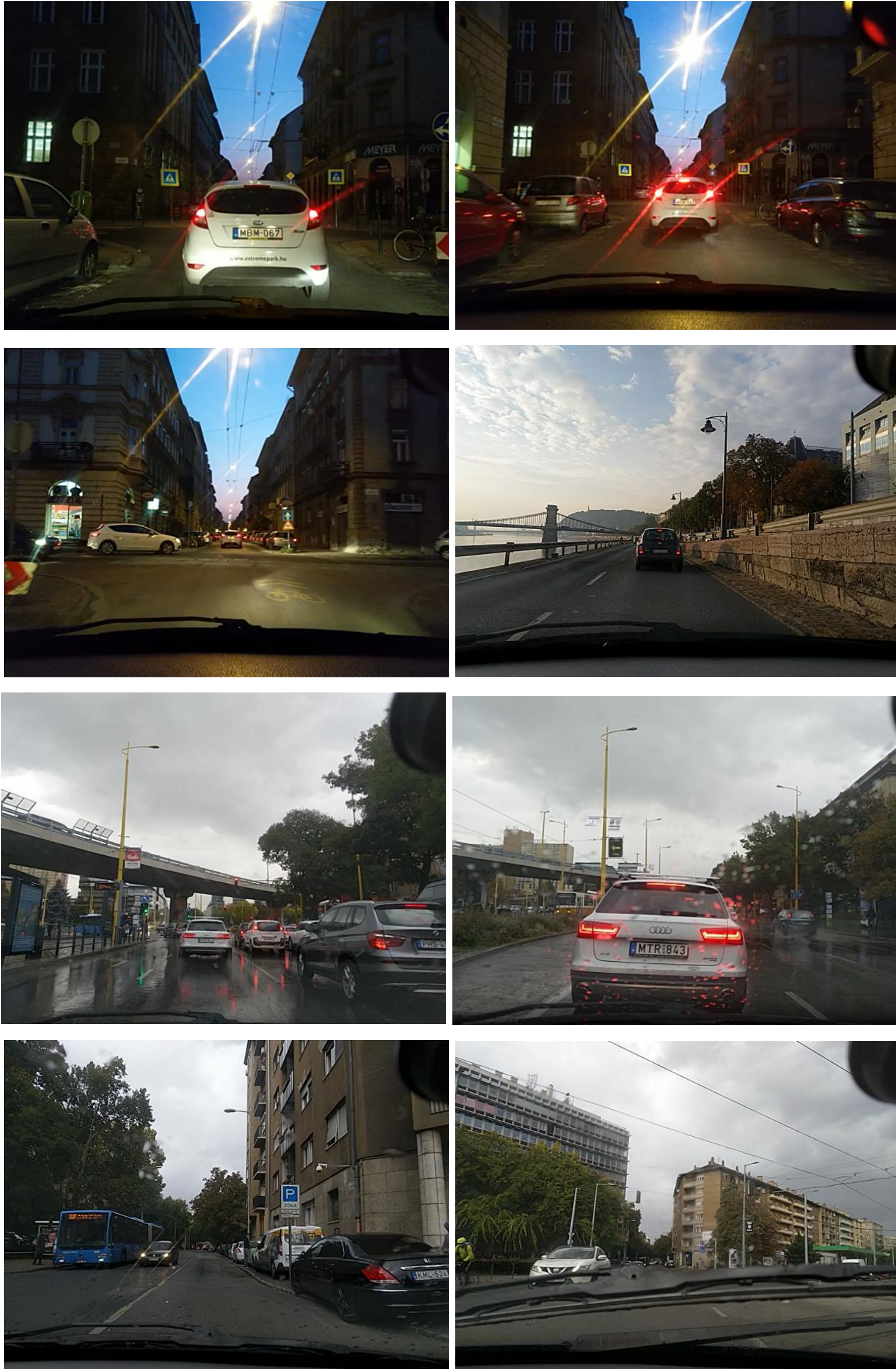
hálózat detektálásából állnak elő, ezen kézzel még van lehetőség később javítani. Ezek a képkockák az xml-el együtt alkalmasak objektum detektáló hálózatok tanító adataként szolgálni.



17. ábra: Általam rögzített videóból egy képkocka

A 17. ábra mutatja a DriverPhone alkalmazás segítségével gyűjtött videóknak az egyik képkockáját. A képen látható objektumok egy részét az automatizált rendszer határozott meg és egy kettő kézzel lett kiegészítve.

A képek és videók 604x480 pixel méretűek, ennél nagyobb felbontás gyűjtése túlságosan leterhelné a mobil készülékeket gyűjtés közben. A videóhoz tartozó metaadatok egy csv fájlban találhatóak, amelynek a neve megegyezik a videó fájl nevével.



18. ábra: Képek az általam gyűjtött videókból

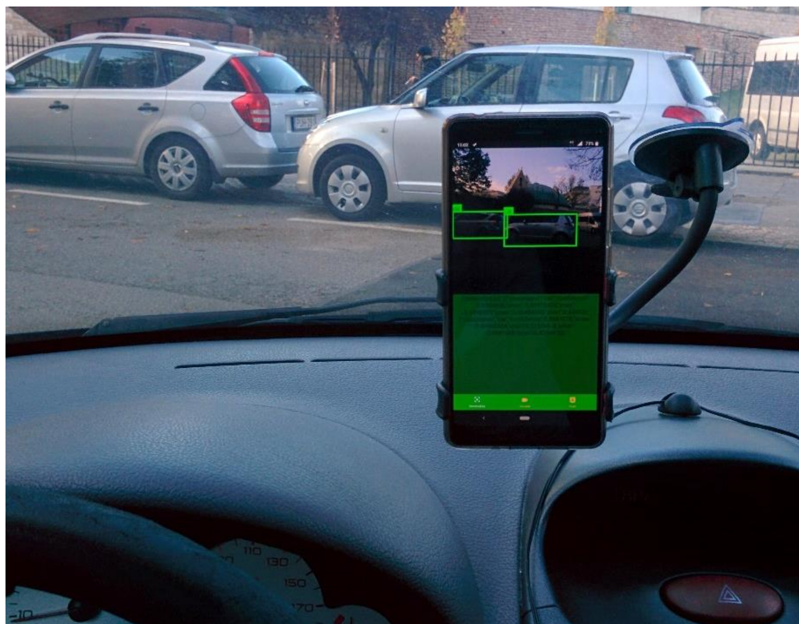
Próbáltam a lehető legkülönbözőbb időjárási körülmények között adatot gyűjteni. A 18. ábra tartalmaz ezekből néhány példát. Számomra a legérdekesebbek az éjszaka készült képek és

az esőben készült képek. Az esős időben két nehezítő tényező is fent áll az objektum detektálásnál. Az egyik, hogy a szélvédő vizes, így a kocsikról érkező fény sok irányba tud torzulni és megtörni. A másik pedig a legutolsó képen látszik, amikor az ablaktörlő éppen törlésben van és ezzel teljesen összezavarhatja a detektálást. Az éjszakai képekben a visszaverődő fények és sötét részek zavarhatják meg a mély neurális hálózatot.

Az adatbázishoz szükséges videók a 19. ábra képén látható módon készültek.

19. ábra: Videó rögzítés folyamatában

A 20. ábra pedig a detektálás futását ábrázolja vezetés közben.



20. ábra: DriverPhone detektálás közben

5 Megvalósítás

Ebben a fejezetben az eddig megtervezett rendszer implementálási lépéseit mutatom be.

Az Android alkalmazás készítésére Android Studio 3.2-t használtam és az alkalmazás Java, C++ és Rust kódokat tartalmaz és támogatja a Kotlin kódot is. Az alkalmazás Java nyelven íródott, egy kép transzformáció lett megvalósítva Rust nyelven, a C++ és a Kotlin támogatása tovább fejlesztési célokból vannak támogatva. A mély neurális hálózat modelleket TensorFlow és Keras gépi tanulásos könyvtár segítségével kezeltem Python 3.7-ben írt kóddal. Python kód fejlesztésére a Jupyter lab és a Visual Studi Code-ot használtam. A fejlesztés Ubuntu 16.04 és Ubuntu 18.04 verziók alatt történt, a munka során verzió frissítés történt. Az általam fejlesztett kódok a fejlesztés során Távközlési és Médiainformatika tanszék által biztosított Github fiókba töltöttem föl. A fejlesztés során sok harmadik féltől származó segédkönyvtárat is használtam, ezeket mindig abban a modulban fogom bemutatni, ahol használatra került.

5.1 Okostelefon alkalmazás: DriverPhone

Az Android alkalmazás futtatásához legalább 6.0 verzió azaz Marshmallow rendszerre van szükség, de a tervezett fordítási API szint az a 27-es, ami Az Oreo névre hallgat. A projekt felépítése lehetővé teszi az alkalmazás fordítását Android Studio-ból.

5.1.1 Felhasználói felület

Az alkalmazás felhasználói felületét én terveztem. A dizájnhoz népszerű alkalmazásokat vettem példának. Az alkalmazás bejelentkezési, regisztrálási és elfelejtett jelszó képernyője ugyanazt a struktúrát követi. Az alkalmazás bejelentkezés utáni képernyője 3 szegmensre bontható: detektáló, rögzítő és profil. A detektálás szegmens végzi az képkockák elemzését mély neurális hálózat segítségével. A rögzítő szegmens új videó rögzítését végzi és a profil szegmens a bejelentkezett felhasználóhoz kapcsolódó funkciókat tárolja. Az alkalmazás az állított és fektetett orientációt is támogatja. Az alkalmazás felhasználói felületéhez az alábbi kiegészítő könyvtárakat használtam:

```
implementation 'com.android.support.constraint:constraint-layout:1.0.2'
implementation 'com.android.support:design:27.0.2'
implementation 'com.android.support:support-v13:27.0.2'
```

5.1.2 Videó kezelő modul

A videó kezelő modulnak a feladata a kamera képet a felhasználói felületen valós időben mutatni. A probléma megoldására több különböző megoldás is létezik. Android API-ban a kamera képének kezelésére meglehetősen bonyolult interfészeket és életciklust terveztek.

Első megoldásomban egy nyílt forráskódú CameraKit-re [https://github.com/CameraKit/camerakit-android] keresztelt könyvtárat használtam, amely elfedi az Android specifikus kamera konfigurációkat. Néhány egyszerű sorral egy videó rögzítését is le lehet programozni:

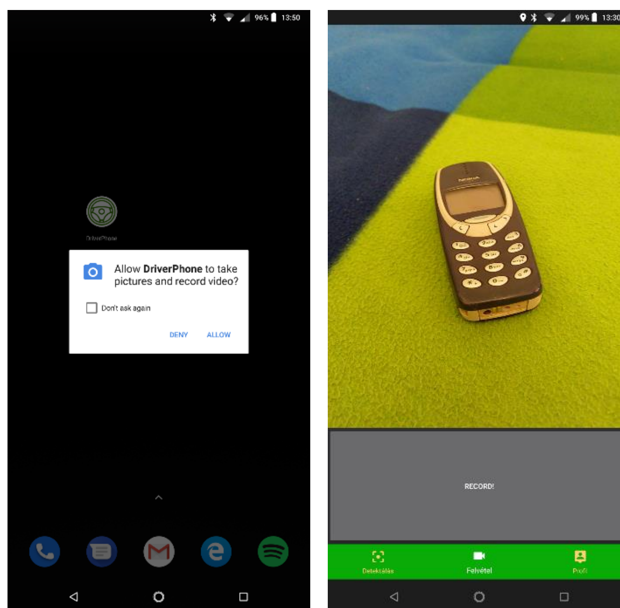
```
camera.setCameraListener(new CameraListener() {
    @Override
    public void onVideoTaken(File video) {
        super.onVideoTaken(video);
        // A fájl paraméter mp4 kiterjesztésű.
    }
});

camera.startRecordingVideo();
camera.postDelayed(new Runnable() {
    @Override
    public void run() {
        camera.stopRecordingVideo();
    }
}, 2500);
```

Az egyszerű használatból azonban hátrányok is származnak. Csak azokhoz a kamera beállítási paraméterekhez lehet hozzáférni, amelyeket a készítői kivezettek a CameraKit interfészére és ez az extra réteg bevezetése egy kép kinyerése esetében nagy (~ 200-300 ms) késleltetés okozott.

Az Android két API-t ad a fejlesztők kezébe, a camera és camera2 API-t. A dolgozat írása során a camera API már érvénytelenített és használata nem garantáltan támogatott, így én a camera2-t választottam a fejlesztésemhez.

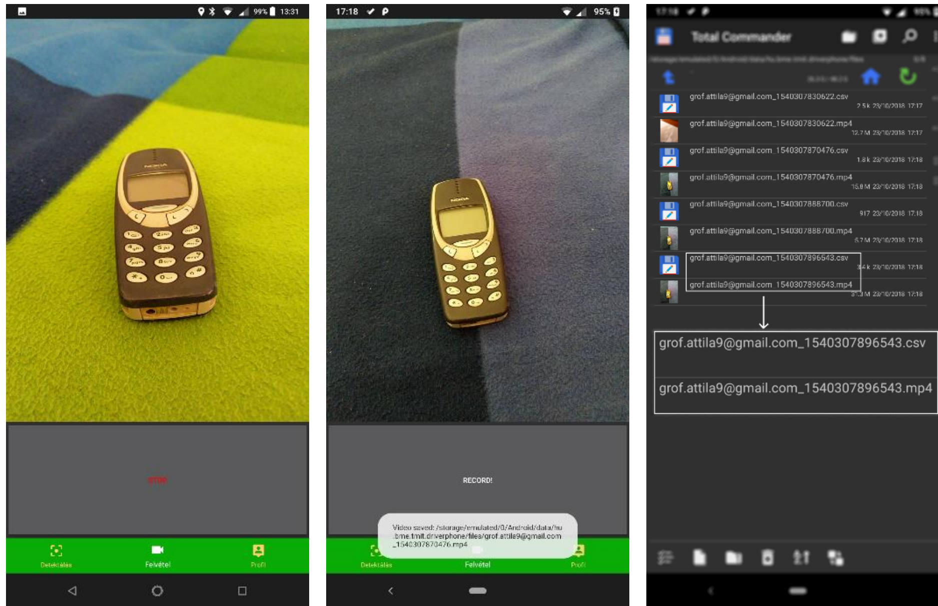
A kamera képét egy fragment-be szerveztem ki, így maga a kamera képe bármilyen képernyőre könnyen beilleszthető. Az alkalmazás indulásakor engedélyt kér a kamerához való hozzáféréshez az alkalmazás.



21. ábra: DriverPhone engedély kérés és kamera képnek megjelenítése

A 21. ábra bal oldalán látható, amikor az alkalmazás indulásakor a kamerához való hozzáférést a felhasználónak engedélyezni kell. A jobb oldalon az alkalmazás futás közben látható, „Record” gomb fölötti rész a kamera élő képét mutatja akkora felbontásban, amekkora az adott kijelzőn arányosan elfér megfelelő rotációval.

A modul videó rögzítésére is a camera2 API-t használja. A rögzített videót mp4 formátumban és 640x480-as felbontásban menti a készülék memóriájába. A 640x480 pixeles felbontás választásának oka leginkább a mentett videó mérete és később a feldolgozás költsége. Egy 20 perces videó ebben a felbontásban ~1.4 GB-ot foglal a háttértáron, ami néhány videó esetén könnyen képes megtelíteni a szabad tárhelyet. A videó felvételt a felhasználó tudja indítani és leállítani is. A videó rögzítésére a háttérben a metaadat gyűjtő modul is munkának lát, majd a rögzítés végeztekor a metaadat gyűjtés is véget ér. A modul a videót a bejelentkezett felhasználó neve és egy időbélyeg konkatenálásával létrehozott néven menti a készüléken a */Android/data/hu.bme.tmit.driverphone/files* mappába.



22. ábra: DriverPhone videó rögzítése

A 22. ábrán látható „Record” gombbal lehet a felvételt elindítani, majd a 22. ábrán látható „Stop” gombbal lehet leállítani a felvételt. A videó sikeres mentése után egy felugró értesítésben jelenik meg a sikeresen mentett videó neve és a mentés helye. A 22. ábrán jobb oldalon Total Commander programban látható, hogy a videó és metaadat fájl ténylegesen létrejött.

5.1.3 Metaadat gyűjtő modul

A metaadat gyűjtő modul feladata a futás idejű adatok gyűjtése. A modulnak két fő feladata van: az egyik a neurális hálózat futása közben a készülék állapotáról teljesítmény adatok gyűjtése és videó rögzítése során a felhasználó helyének és mozgásának adatai.

A videó rögzítése során párhuzamosan a háttérben egy csv fájl készül, amelyben az alábbi mezők szerepelnek:

- Időbélyeg (ms)
- Mobil internet helyzet hosszúsági és szélességi fokban
- GPS helyzet hosszúság és szélességi fokban
- Gyorsulás mérő x,y és z tengely mentén
- Giroszkóp x,y és z tengely mentén

Egy példa csv az alábbi módon néz ki:

```
time;locationNetworkLat;locationNetworkLong;locationGPSLat;locationGP
SLong;accelerometerX;accelerometerY;accelerometerZ;gyroscopeX;gyrosco
peY;gyroscopeZ;
```

```
1540060684684;47.5337027;19.0622581;47.52966641;19.06045393;0.1512298
6;6.652466;6.9555206;-0.0028076172;0.0012512207;-4.4250488E-4
1540060685693;47.5337027;19.0622581;47.52966641;19.06045393;-
0.023345947;6.4713135;7.6416473;0.007736206;-0.008865356;0.002532959
1540060686697;47.5337027;19.0622581;47.52966641;19.06045393;-
2.0831604;4.9767303;7.299881;-0.037857056;-0.18676758;-0.09526062
1540060687702;47.5337027;19.0622581;47.53381461;19.06205087;0.2561798
;6.5702667;6.6269073;0.4253235;-0.2614441;0.22219849
…
```

A videós metaadat gyűjtése a videó rögzítésével egyszerre indul külön szálon és ~1 másodpercenként begyűjti a fentebb bemutatott metaadatokat. Ha valamelyik érték nem érhető el, akkor az előző állapota kerül kiírásra, ha pedig olyan még nem volt, akkor nem kerül érték a helyére.

Emellett a detektálás közben is történik adatgyűjtés, itt azonban ez a csak a predikció kiszámolásának idejét foglalja magában. Az alkalmazásban amikor a detektáló képernyő elindul, akkor egy *usage_data.csv* fájlba kezdi el írni a detektálási időket és az addigi átlagukat. Ez az adatgyűjtés a teljesítmény kiértékelésénél fog fontos szerepet játszani. A detektálás közben az adatgyűjtés 5 másodpercenként történik. Ennek oka, hogy az 5 másodpercenkénti mintavételezés is elég ahhoz, hogy ki lehessen értékelni a detektálás teljesítményét, de nem túl sűrű ahhoz, hogy túlságosan terhelje a készüléket.

A gyűjtött adatok:

- Időbélyeg (másodperc)
- Legutolsó predikció kiszámolásának ideje (ms)
- Átlag predikció számolás ideje(ms)

Egy példa a csv fájlra:

```
Time;Inference Time(ms);Inference Time average(ms)
5;334.0;582.8571
10;345.0;488.0
15;443.0;461.8095
20;438.0;463.77777
25;445.0;459.88235
30;576.0;471.15384
```

5.1.4 Mély neurális hálózat modul

A dolgozat egyik fő célja az objektum detektálás mély neurális hálózat segítségével úgy, hogy az a készüléken fut. Az implementáláshoz a TensorFlow Java API [20] könyvtárt használtam. Az implementálás elején érdemes megemlíteni, hogy a TensorFlow-nak van

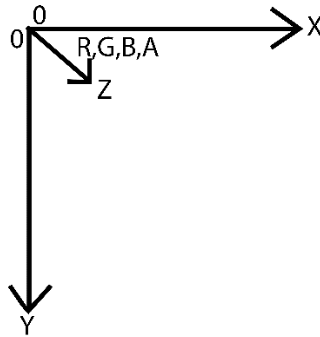
elérhető Python és C++ API-ja, amelyek hivatalosan ki vannak adva és működésük valamilyen szinten támogatott. A Java API egyelőre kísérleti fázisban van, így a működésével kapcsolatban semmilyen garanciát nem vállal a fejlesztő csapat. A csomagban a dolgozat írása pillanatában csak néhány funkció van leimplementálva a többi API-hoz képest. Szerencsére ez ahhoz éppen elég, hogy egy hálózatot képesek legyünk betölteni és futtatni.

A kamera kép indulásával példányosodik egy SsdMobilnet osztály (általam készített osztály) külön szálon, amely elkezdi a mély neurális hálózat futtatásához szükséges komponensek betöltését. Először a futtatáshoz szükséges natív TensorFlow függvények betöltése szükséges:

```
// TensorFlow natív függvények betöltése
static {
    System.loadLibrary("tensorflow_inference");
}
```

Majd a konstruktorban a mély neurális hálózat modellt be kell olvasni byte tömbként és a TensorFlow Graph osztálynak átadni. Amennyiben a modell betöltése sikeres volt, akkor egy új Session osztályt kell példányosítani, amelyben a predikciók számolása futhat. Nagyon fontos, hogy egy közös Session objektum éljen az alkalmazásban Singleton-ként mert, ha minden egyes predikció számolásánál új Session-t kell létrehozni, akkor a futási idő a duplájára is emelkedhet mert egy új Session-t létrehozni költséges művelet. Az inincializáló műveletek lefutása után a külön szál visszajelez a kamerát futtató osztálynak, hogy készen áll a predikciók futtatására. Közben elindul az metaadat gyűjtő modulnak a megfelelő adatgyűjtő háttér folyamata.

Egy predikció számolása a kamera képének elkérésével kezdődik. A kamera kép közvetítését megvalósító osztály (CameraImgeListener) egy Bitmap képet ad, amely ARGB_8888 kódolással adja meg a kép színeit. Az ARGB_8888 típus azt jelenti, hogy minden egyes pixel 4 bájton van tárolva és RGBA színeket foglalja magába. Az SSD hálózatnak a bemenete egy 1x300x300x3-as dimenziójú képet vár RGB színekkel. A CameraImageListener egy négyzet alakú képet ad, az én esetemben egy 2976x2976 méretű képet. Egy skálázás segítségével ebből a képből áll elő a 300x300-as Bitmap kép.



23. ábra: Bitmap kép felépítése

A 23. ábra mutatja a Bitmap kép felépítését. Ezt a 300x300x4 méretű képet először egy 300x300x3-as méretre konvertálom át az „A” eldobásával. Ezután az egész képet egy 1 dimenziós bájt tömbbe konvertálom át úgy, hogy soronként és R,G,B sorrend szerint írom a tömböt. A bájt tömbre azért van szüksége mert a Java-ban nincs Uint8-as típus, ami egy jelöletlen (unsigned) egész szám (integer) típus 8 biten tárolva. Java-ban csak a byte típus van, amelyiket 8 biten tárolnak.

Az 1 dimenziós bájt tömb létrehozás után, a tömbből létre kell hozni egy Tensor típusú objektumot 1x300x300x3-as dimenzió inicializálással. A létrejött Tensor objektum lesz a mély neurális hálózatnak a bemenete. A hálózat futtatásához ismerni kell a bemeneti csomópont nevét, dimenzióját és a kimeneti csomópontok neveit és dimenziójukat. Az SSD MobileNet esetében ezek az alábbiak voltak:

```
private static final long[] INPUT_SIZE = {1, 300, 300, 3};
private static final String SSD_INPUT_NAME = „image_tensor”;
private static final String[] SSD_OUTPUT_NAMES = {„num_detections:0”,
„detection_classes:0”, „detection_scores:0”, „detection_boxes:0”};
```

Ahogy látható ennek a hálózatnak egy bemente van és 4 kimenete. A kimenetek mérete a „num_detections”-től függ.

A Session, Tensor és a kimenetek segítségével egy predikció futtatása aza alábbi módon néz ki:

```
List<Tensor<?>> resultList = session.runner()
        .feed(SSD_INPUT_NAME, image)
        .fetch(SSD_OUTPUT_NAMES[0])
        .fetch(SSD_OUTPUT_NAMES[1])
        .fetch(SSD_OUTPUT_NAMES[2])
        .fetch(SSD_OUTPUT_NAMES[3])
        .run();
```

A futtatás után az eredményt az alábbi módon nyerhetjük ki a hálózatból:

```
//num_detections
```

```
float[] numberOfDetections;
//detection_classes
float[][] detectionClasses;
//detection_scores
float[][] detectionScores;
//detection_boxes
float[][][] detectionBoxes;

numberOfDetections = resultList.get(0).copyTo(new float[1]);
detectionClasses = resultList.get(1).copyTo(new float[1][100]);
detectionScores = resultList.get(2).copyTo(new float[1][100]);
detectionBoxes = resultList.get(3).copyTo(new float[1][100][4]);
```

A kimeneti értékek:

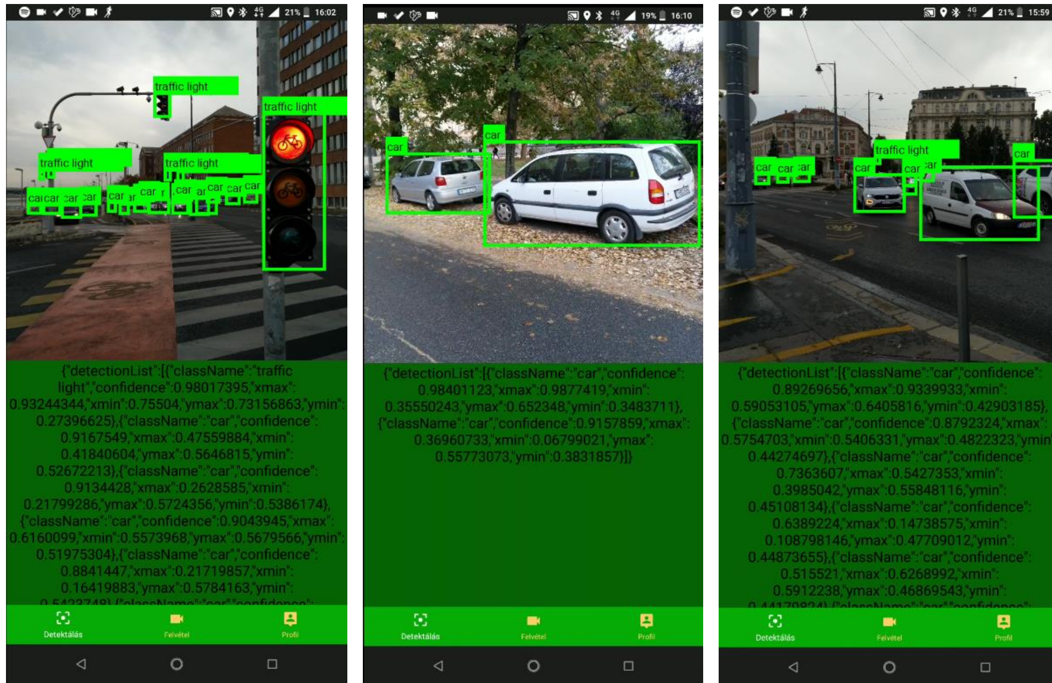
- num_detections: Detektálások száma.
- detection_classes: A detektált osztályok címkéjének kódja.
- detection_scores: A detektált osztályok magabiztossága.
- detection_boxes: A detektált objektum befoglaló keretének koordinátái. (y min, x min, y max, x max)
- detection_masks: A detektált objektum maszkjai, az alkalmazásban ez a kimenet nem releváns így nem is került használatra.

Az eredmények kinyerése után a megkapott értékeket transzformálni kell és objektum orientált módon letárolni, hogy az adatok kezelése könnyebb legyen. A detektálások közül én csak azokat tartottam meg, amelyek a 0.5 magabiztosságot elérték.

Az eredmények átalakítása után a detektált objektumok címkéit egy konfigurációs fájlból kiolvasva állnak elő a detekciók. Egy Android canvas (rajzoló felület) segítségével az adott készülék képernyőjére átskálázva a detektált objektumok befoglaló keretei megjelenítésre kerülnek a címkével együtt.

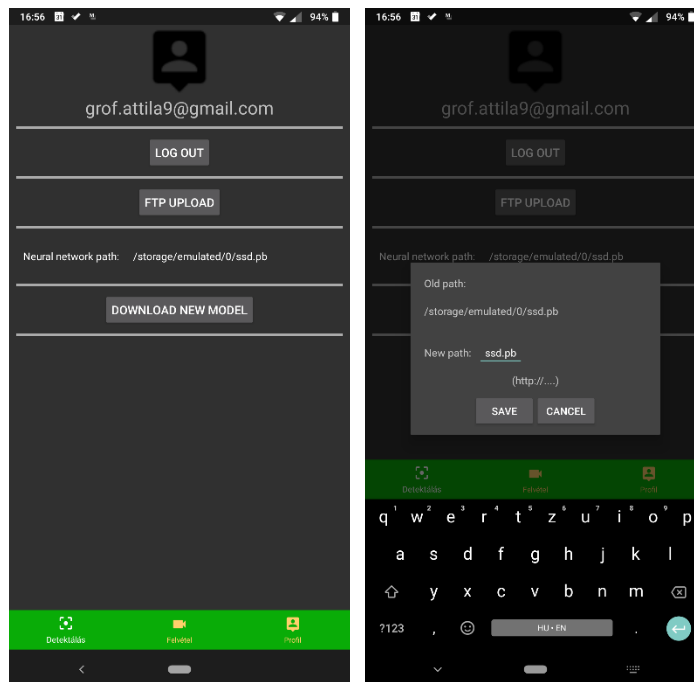
Miután a mély neurális hálózatot futtatását megvalósító osztály a fentebb említett feladatokat elvégezte, kéri a következő képet, amivel a detektálási feladatok újra indulnak. Az Android életciklusok szerint a mély neurális hálózat inicializálása és megsemmisítése is megtörténik, így a készülék elfordítását és egyéb eseteket is képes az alkalmazás kezelni.

Az alkalmazásban alapértelmezetten telepítéskor található egy tanított SSD MobilNet hálózat, azonban van lehetőség ezt konfigurálni. Az adatszinkronizációs modul képes újabb hálózatot letölteni az előre beállított webszerverről, miután a fájl sikeresen letöltődött, annak az elérési címét beállítva lehet lecserélni a háttérben futó mély neurális hálózat modellt.



24. ábra: Objektum detektálás a DriverPhone alkalmazással

A 24. ábra képei mutatják az objektum detektáló képernyőt. A kamera képe alatti terület a hálózat kimenetéből létrehozott objektum json formátumban. A kamera képén láthatóak az objektumokat befoglaló keretek és a keretekhez tartozó címkék is.



25. ábra: Mély neurális hálózat elérési útvonal mentése

A 25. ábra bal oldalon mutatja a profil oldalt. Ezen belül a „Neural network path:" rákattintásával van lehetőség annak módosítására. A rákattintás után egy dialógus ugrik fel, ahol

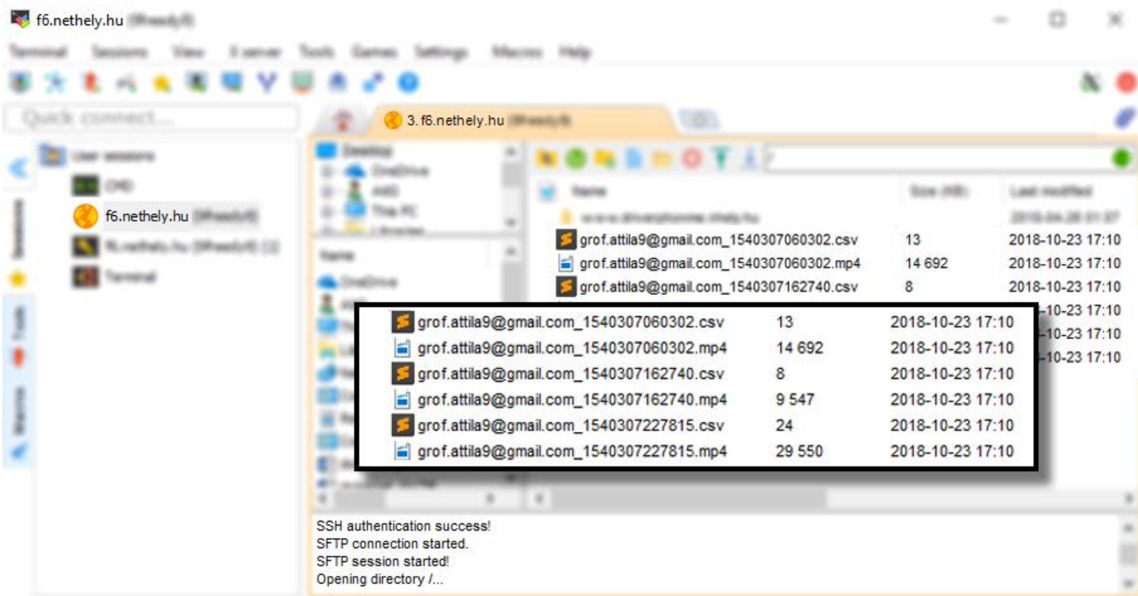
az új elérési útvonalat be lehet állítani. Amennyiben az elérési cím hibás, a detektálás képernyő nem fog lefagyni, a kamera képét így is mutatni fogja csak detektálás nélkül.

A Google által fejlesztett TensorFlow csapat egy TensorFlow Lite-nak [21] keresztelt könyvtár a korlátozott erőforrású készülékekre lett készítve. A DriverPhone alkalmazás képes .tf formátumú hálózatokat is futtatni, de inkább csak tesztelés jelleggel. A TensorFlow Lite-ban kimentett modellek méretben kompaktabbak és futási időben gyorsabb predikció számolás ígérnek, mint a tradicionálisan mentett .pb modellek. Ezeket a javulásokat a súly értékek tömörítésével, hálózat nem aktív részeinek eltávolításával és a számítási gráf optimalizálásával érik el. A könyvtárnak egy hibája van, hogy komplex hálózatok automatikus exportálását még nem támogatja és a Google csak néhány hálózatnak adta ki a .tf verzióját. A meglévő modelleket tovább tanítás után így nem lehet kimenteni futtatható verzióba.

5.1.5 Adatszinkronizációs modul

Az alkalmazás által rögzített videó és CSV fájlok feltöltése a feladat. A modul egy konfigurációs fájlban meghatározott FTP tárhelyre tölti fel az adatokat, majd a sikeres feltöltés után törli azokat. Az adatok feltöltését csak akkor indítja a készülék, amikor wifi kapcsolat áll rendelkezésre. A szinkronizáció elindítására egyelőre kézzel van lehetőség a profil fül alól. A „Log Out" gomb megnyomására indul el a szinkronizáció, amely 25. ábra bal oldalán látható.

A feltöltés a háttérben egy külön szálon történik. és egy felugró szöveg jelzi a feltöltés sikerességét, illetve amennyiben hiba van, annak okát. A 26. ábra mutatja, hogy a tárhelyre való SFTP-s bejelentkezés után a készüléken lévő fájlok a megfelelő struktúrában (1 videó – 1 CSV fájl) megjelennek. Abban az esetben, ha wifi kapcsolat nem elérhető, akkor az alkalmazás egy felugró értesítésben szól, hogy a szinkronizálás nem képes elindulni mert a készülék nincs wifi-hez kapcsolódva.



26. ábra: FTP tárhely

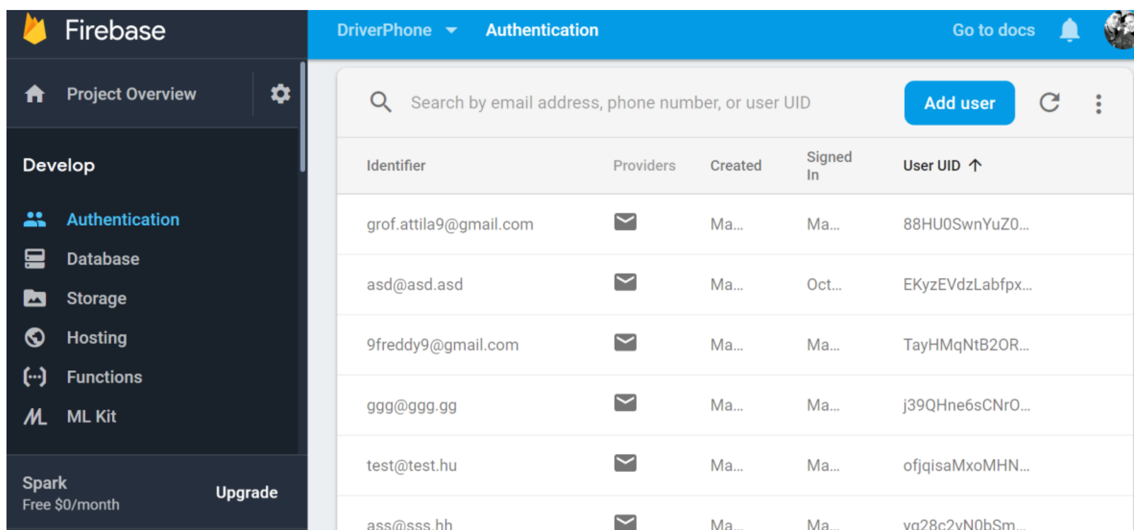
Az adatszinkronizációs modulnak még egy feladata van, ez pedig az „új, okosabb” mély neurális hálózat letöltése. Az alkalmazás konfigurációs fájljában be van állítva egy HTTP cím, ahonnan egy GET HTTP kéréssel leölt egy fájlt és ezt *model.pb* néven elmenti a készülék háttértárára.

5.1.6 Jogosultság kezelő modul

A Firebase nevű céget [22] 2011-ben alapították és céljuk a mobil és webalkalmazások szerver oldali fejlesztését leegyszerűsíteni a fejlesztők számára. 2014-ben a Google megvette és az akvizíciónak köszönhetően egyre jobban próbálja integrálni termékeibe. Az Android alkalmazások sem kivételek ez alól. A szolgáltatások, amelyeket a Firebase szolgáltat:

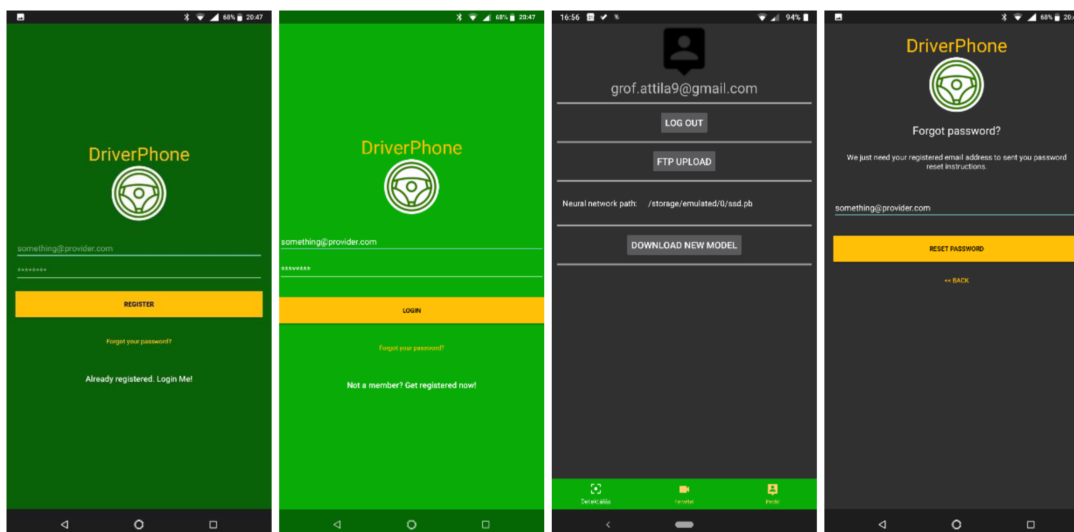
- Authentikáció
- NoSQL Adatbázis
- Tárhely
- Web és mobil alkalmazás hosztolás
- Analitikai eszközök
- stb…

Én munkám során ezek közül az authentikáció szolgáltatását integráltam az alkalmazásba. Az alkalmazás lehetővé teszi új felhasználók regisztrálását e-mail és jelszó megadásával. Emellett elfelejtett jelszó esetén email-en értesíti a felhasználót, aki új jelszót igényelhet. A DriverPhone a bejelentkezés és a kijelentkezés is lehetséges.



27. ábra: Firebase webes adminisztrátori felülete

A 27. ábra mutatja a Firebase adminisztrátori oldalát, ahol a regisztrált felhasználókat láthatjuk. Ezen a felületen van lehetőség a felhasználók blokkolására vagy esetleg új felhasználók felvételére. A többi szolgáltatást is a 27. ábra bal oldalán látható felületről van lehetőség kezelni, de azok egyelőre nincsenek beintegrálva a DriverPhone-ba.



28. ábra: Firebase integrálása DriverPhone alkalmazásba

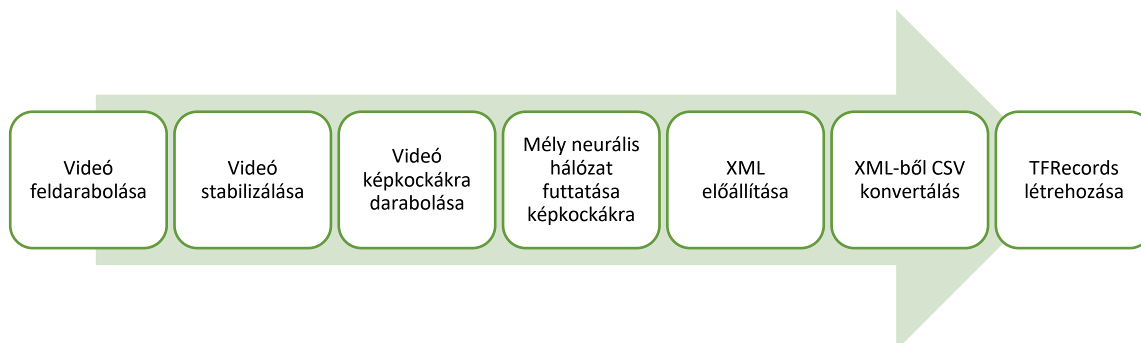
A 28. ábra képein láthatóak a Firebase felhasználói felületei. Bal oldalon a legszélső a regisztráló képernyő, balról a második a bejelentkező, balról a harmadik a kijelentkező és jobboldalon a legszélső az elfelejtett jelszó képernyő.

5.2 Python alkalmazás

A Python alkalmazás tartalmazza azokat a logikákat, amelyek magas számítási kapacitást és nagy tárhelyet igényelnek. Ez magába foglalja a nyers adatok feldolgozását, egy tanító adatbázis építését és mély neurális hálózat újratanítását. Az alkalmazás magába foglal egy feldolgozó scriptet, ami csak adatfeldolgozáskor fut és egy webszervert, amely mindig elérhető és rajta keresztül letölthető a legújabb mély neurális hálózat modell.

5.2.1 Adatfeldolgozó modul

Az adatfeldolgozó modul feladata a nyers videókból egy tanításra alkalmas TFRecords előállítása. A tanítani kívánt hálózat egy objektum detektálására alkalmas SSD MobileNet hálózat, a tanító adatok előállítása ennek fényében történik. Az adatfeldolgozási lépések erőforrás igényes feladatok, nagyobb méretű videók feldolgozási ideje akár a napokat is elérhetik egy átlagos felhasználói gépen. A feldolgozás során keletkezett köztes fájlok nem kerülnek törlésre, így hiba esetén, illetve, ha a folyamatnak egy lépése megváltozik, akkor nem kell a folyamatot az elejétől indítani. A 29. ábra mutatja az általam tervezett 7 lépcsős adatfeldolgozási folyamat lépéseit.



29. ábra: Adatfeldolgozási lépések

A 29. ábra által illusztrált adatfeldolgozási folyamat egyes lépéseit részletesen is bemutatom.

A mobil telefonnal készítettet videó felvételek nagy méretűek és akár 30 perc vagy 1 órásak is lehetnek ezért nem praktikus ezzel a nagy fájllal dolgozni a folyamatban. Első lépésként ezt az eredeti videó fájlt darabolom fel egyenlő hosszúságú darabokra, úgy, hogy ezt a hosszt a futtatás elején konfigurációs paraméterként át lehet adni a script-nek.

A videó stabilizálása az objektum detektálás szempontjából nem feltétlenül szükséges lépés. A stabilizálás akkor szükséges, amikor videóban található egymás után következő képkockák kapcsolatát elemezzük. Ebben az esetben az elmosódás és a gyors kamera mozgás elkerülése a feladat. Az okostelefonnal a járműből készült videók könnyen mosódnak el a sok rázkódásnak köszönhetően. Az általam készített videók egy ablakra rögzíthető telefon tartó segítségével készültek. A telefon tartóból két típussal teszteltem, amelyek a 30. ábra mutat.

30. ábra. Okostelefon tartó eszköz a kocsiban

A stabilizálási lépés megvalósításának oka, hogy a későbbiekben az adatbázis szolgálhasson más mély tanulás alapú feladatokhoz tanítási adatként. A stabilizáláshoz a Vidstab [23] Python nyelven írt csomagot használtam.

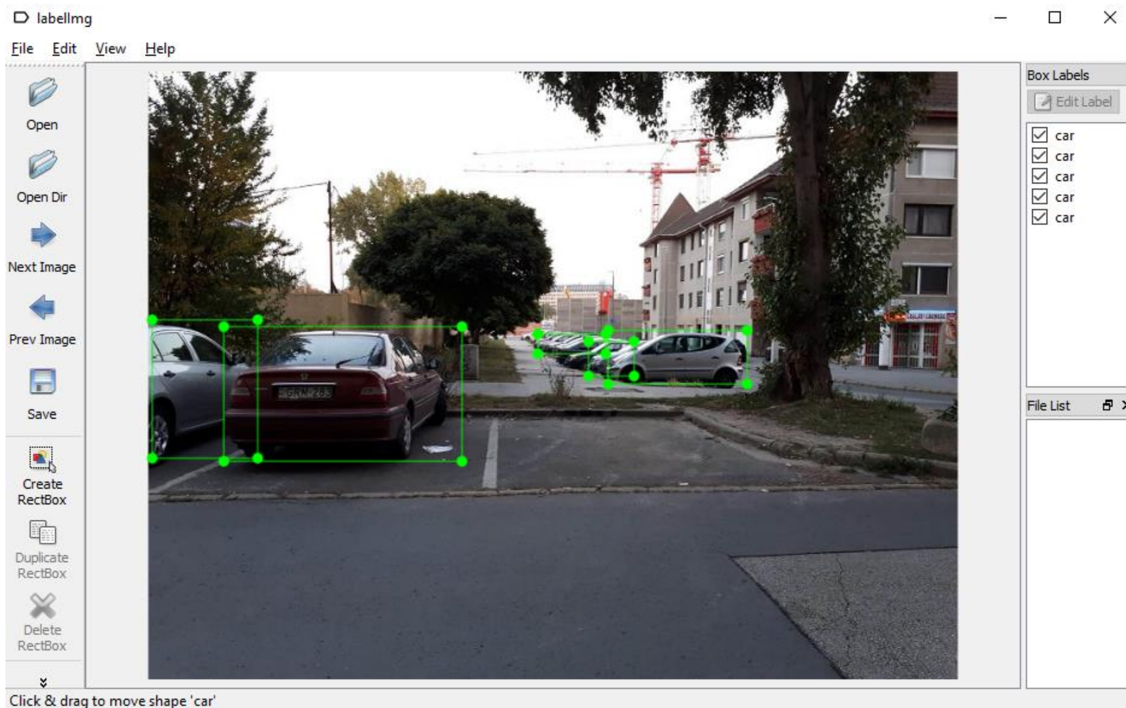
A feldarabolást és stabilizálást követően a videókat képkockákra kell felvágni. A képek a videó nevét viszik tovább és kapnak egy futó sorszámot.

A tanító adatok automatikus előállításához szükség van egy mély neurális hálózatra, amely segít a képeken található objektumok meghatározására, hogy azt ne kelljen kézzel elvégezni. A hálózat valószínűleg nem képes tökéletes detektálásokat jelezni, de az eredményt korrigálni rövidebb idő, mint kézzel bejelölni a detektálásokat. Az okostelefonon az SSD Mobilnet hálózat fut azonban mivel ez az alkalmazás szerver oldalon fut és nem a sebesség, hanem a pontosság a prioritás, így komplexebb modellt is lehet használni GPU gyorsítással. Az általam megírt script támogatja a TensorFlow Object Detection API-ban [24] megtalálható modelleket, amelyeknek a kimenete az objektumokat körbefogó keret. A tanító adatok létrehozásánál futás elején módosítható, hogy milyen hálózattal fusson a kód, sok rendelkezésre álló erőforrás esetén lehet a legpontosabb hálózatot választani, de lehetőség van kisebb erőforrást igénylő hálózat használatára is Jelenleg az SSD típusú hálózatokból elérhető verziókat az 6. táblázat mutatja.

Modell neve	Sebesség (ms)	COCO mAP[^1]
ssd_mobilenet_v1_coco	30	21
ssd_mobilenet_v1_0.75_depth_coco	26	18
ssd_mobilenet_v1_quantized_coco	29	18
ssd_mobilenet_v1_0.75_depth_quantized_coco	29	16
ssd_mobilenet_v1_ppn_coco	26	20
ssd_mobilenet_v1_fpn_coco	56	32
ssd_resnet_50_fpn_coco	76	35
ssd_mobilenet_v2_coco	31	22
ssdlite_mobilenet_v2_coco	27	22
ssd_inception_v2_coco	42	24

6. táblázat: Elérhető SSD hálózatok (módosítva [25] alapján)

A mély neurális hálózat kimenete egy sok dimenziós tömb, amely tartalmazza a detektált objektumok helyét, objektumok maszkjait, magabiztosságát és a detektált objektum osztály kódját. Az előző lépésben létrehozott képkockák közül mindegyikre lefut a bekonfigurált mély neurális hálózat és mindegyikre ad egy detektálást. Ez több óráig is eltarthat, abban az esetben amikor ezt pár ezer képre kell futtatni. Amennyiben egy képen a detektálás nem lett tökéletes, kézzel van lehetőség javítani. A LabelImg program segítségével a képen megjelennek a detektált objektumok és ezek szerkesztésére van lehetőség. Mentésnél a már létrehozott xml fájlba fog írni.



31. ábra: LabelImg működés közben

A 31. ábra a LabelImg programot mutatja futás közben. A téglalapok mozgatásával lehet a mögöttes xml tartalmát módosítani.

Egy példa az xml fájl kinézetére:

```
<?xml version="1.0" ?>
<annotation>
   <folder/>
   <filename>attila_15388170567120_stabilized_frame0.jpg</filename>

<path>/home/atig/test_output/frames/attila_15388170567120_stabilized_
frame0.jpg</path>
   <size>
      <width>640</width>
      <height>480</height>
   </size>
   <object>
      <name>car</name>
      <bndbox>
         <xmin>448</xmin>
         <ymin>161</ymin>
         <xmax>570</xmax>
         <ymax>263</ymax>
      </bndbox>
   </object>
</annotation>
```

Ebből az xml-ből látható, hogy az xml nevével megegyező nevű képen egy autó található a megadott koordináták között.

A következő lépés az összes xml fájl egy csv-re konvertálása. Ez a csv fájl fogja magába foglalni az összes tanító adatot. Magába foglalja a kép nevét, a szélességeét, magasságát, a rajta detektált objektum koordinátáját és osztályát. Ahány objektum van egy képen, a csv fájlban annyi sor fog generálódni. Az említett csv fájl az alábbi módon néz ki:

```
filename,width,height,class,xmin,ymin,xmax,ymax
/home/atig/test_output/frames/asd_15388170567120_stabilized_frame58.j
pg,640,480,car,536,176,640,277
/home/atig/test_output/frames/asd_15388170567121_stabilized_frame19.j
pg,640,480,car,554,225,631,270
/home/atig/test_output/frames/asd_15388170567120_stabilized_frame155.
jpg,640,480,car,83,171,401,468
/home/atig/test_output/frames/asd_15388170567120_stabilized_frame155.
jpg,640,480,car,96,175,390,360
```

A csv létrejötte után a TFRecords létrehozása következik. A konvertálásnál be kell állítani az általunk detektálni kívánt osztályokat, amelyek az én esetemben az alábbiak (autó, busz, jelző lámpa):

```
def class_text_to_int(row_label):
    if row_label == 'car':
        return 1
    elif row_label == 'bus':
        return 2
    elif row_label == 'traffic light':
        return 3
    else:
        print("Error: " + row_label + " received")
        None
```

A script lefuttatásával létrejönnek a TFRecords-ok. A tanító és teszt adatot a TFRecords létrehozás előtt kell még végrehajtani. Így lesz egy *train.record* és egy *test.record* fájljaink, amelyek már a metaadatokat és a képeket is tartalmazzák és alkalmasak tanításra.

5.2.2 Mély neurális hálózatot tanító modul

A tanításhoz az alábbi fájlok szükségesek:

- Mély neurális hálózatot leíró fájl
- Tanító adat
- Tesztelő adat
- Tanítási konfigurációs fájl
- Címke leíró fájl
- Előző tanításból checkpoint file (opcionális)

A mély neurális hálózatot leíró fájl maga a hálózat felépítését írja le, milyen csomópontok milyen más csomópontokhoz kapcsolódnak, az értékeknek milyen típusai vannak és milyen műveletek hajtódnak végre a csomópontokban.

A tanító és tesztelő adat az előző fejezet végén kép, xml és csv-ből generált TFRecords fájlok.

A tanítási konfigurációs fájl tartalmazza az összes tanítással kapcsolatos beállítást. Itt van lehetőség beállítani a 2.4 fejezetben már említett általános keret arányokat (anchor_generator). A dropout értéket is meg lehet adni és ezek mellett még rengeteg tanítást befolyásoló paramétert lehet konfigurálni. Amelyeket kötelező beállítani azok a tanító, tesztelő adatok, osztály címkék és ha már volt korábban tanítás, akkor az ahhoz tartozó checkpoint fájlt.

A tanítás egy train.py script-el indul, aminek a forrás és célmappát meg kell adni. A tanítás előrehaladását a TensorBoard-al lehet monitorozni. A TensorBoard a legfontosabb értékeket a tanítás folyam vizualizálja, hogy a tanítás eredménye és folyamata érthetőbb és átláthatóbb legyen.

A tanítás során az alábbi fájlok jönnek létre:

- model.ckpt.data-00000-of-00001
- model.ckpt.index
- model.ckpt.meta

Ez a három fájl tartalmazza a hálózat által megtanult információt és a modell „befagyasztásához" vagy kimentéséhez ezekre a fájlokra van szükség. Ha több model.ckpt.data van a tanítás után akkor mindig a nagyobb szuffix számút érdemes felhasználni mert az tartalmazza a tanítás későbbi fázisának eredményét. Az általam használt modell a 4.1. és 4.2. fejezetben bemutatott adatbázisokon volt előtanítva.

A modell fagyasztása után létrejön a .pb kiterjesztésű modell, amely már alkalmas arra, hogy más környezetekben predikció kiszámolásra legyen felhasználva. A mentett hálózat ~ 30 MB-ot foglal.

Az objektum detektálás feladatánál egy modell pontosságát az mAP (Mean Average Precision)-vel szokták jelezni.

$$mAP = \frac{\sum AP(q)}{Q} ;$$

$$AP = \frac{Igaz\ pozitív\ (True\ Positive)}{Összes\ pozitív\ (True\ and\ False\ Positive)}\ ;$$

$$Q = összes\ osztály;\ q = egy\ osztályra$$

5.2.3 Adatszinkronizációs modul

A szerver oldali Python alkalmazásnak egy webszervert kell biztosítania, hogy a kliensek képesek legyenek az újabb verziójú mély neurális hálózatot letölteni. A webszerver A Flask nevű Python csomag segítségével került megvalósítására.

A Flask segítségével egy egyszerű webszervert hoztam létre, amelynek egy GET típusú végpontja van. A végpont *<ip cím>/api/file/<fájl név>* címen érhető el és amennyiben a keresett fájl elérhető, annak letöltése elindul.

5.3 Megvalósítás során felmerülő nehézségek

A fejlesztés során a legnagyobb nehézségeket a használt technológiák aktív fejlesztése okozta. A TensorFlow könyvtárt folyamatosan fejlesztik és ebből következően gyakran egy új fejlesztéssel régebbi funkciókat törnek el. A mély neurális hálózat tanítását 3-szor kellett újra tervezni, mert a TensorFlow fejlesztések miatt az addig működő kód, már hibás működést eredményezett.

A legnagyobb kihívást a TensorFlow Java API használata okozta, mivel ez még csak kísérleti fázisban van és emiatt csak minimális szintű dokumentáció érhető el hozzá. A Google által kiadott hivatalos példák a Java API-t még nem is használják. Emellett Python-ban TensorFlow segítségével írt mély neurális hálózat használata Java nyelvben nehézkes a különböző adattípusok miatt.

6 Tesztelés és eredmények

A dolgozat első felében a témához szorosan kapcsolódó technikákat és megvalósításokat feldolgoztam. Ezekből merítve és saját ötletek alapján terveztem egy komplex szoftver rendszert, amely legfontosabb része az objektumokat detektáló mély neurális hálózat. Ezt a komplex szoftver rendszert leimplementáltam.

Az Android alkalmazást Android Studio segítségével írtam és Nokia 7 Plus HDMI Global és Lg Nexus 5 készüléken teszteltem. Az alkalmazás futtatásához szükséges minimális követelmény a 6.0 (Marshmallow) verzió. A szerver oldali Python alkalmazást Visual Studio Code és Jupyter Lab segítségével írtam.

A 15. ábra által leírt architektúra megvalósításra került az összes komponensével együtt. A 14. ábra folyamatai közül a rendszer minden folyamatot megvalósít.

Az Android alkalmazás képes TensorFlow és Keras-ban írt konvolúciós mély neurális hálózatot futtatni a készüléken lokálisan. A tovább tanított SSD MobilNet hálózat futtatásásra is képes. A Nokia készüléken egy predikció átlagos kiszámítási ideje ~470 ms 1000 futás alapján, míg az LG esetében ez 3 másodperc fölött van. Futás közben a készülékek jelentős mértékben melegedtek és a processzor használat is magas volt. A detektálás eredményéről a felhasználó a felületen értesül a kamera képére rajzolva. Az alkalmazásban futás közben lehet cserélni a mély neurális hálózat az új modell elérési címének megadásával.

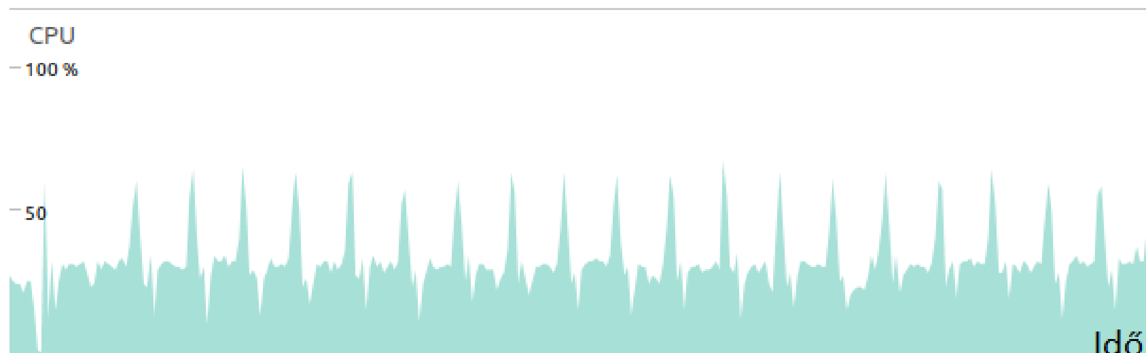
A detektáló funkció tesztelése két készüléken futott:

- LG Nexus 5[1]
- Nokia 7 Plus HDMI Global[2]

A tesztek nagyjából 5 percig futottak a két készüléken és a futás közben az Android Studio Profiler nevű elemzési eszközzel vizsgáltam a két készüléket.

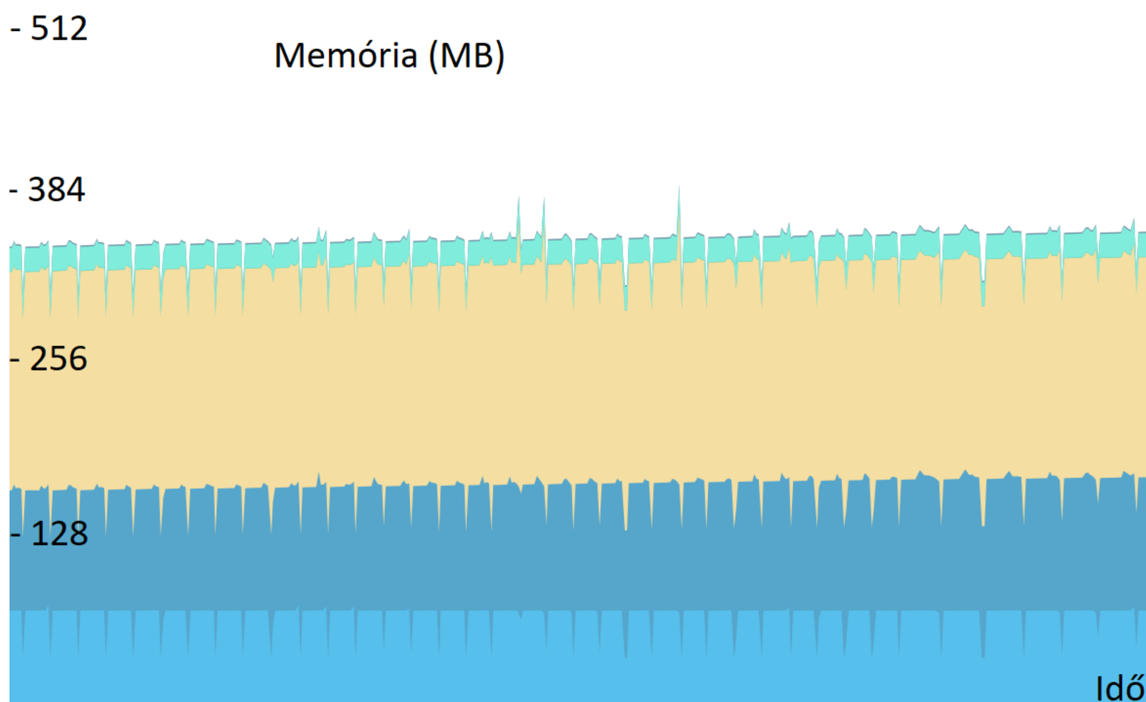
[1] OS: Android 6.0, Processzor: 2.26GHz quad-core (Qualcomm Snapdragon 800), RAM: 2 GB,

[2] OS: Android 9.0, Processzor: 4x2.2GHz + 4x1.8GHz octa-core (Qualcomm Snapdragon 660), RAM: 4 GB



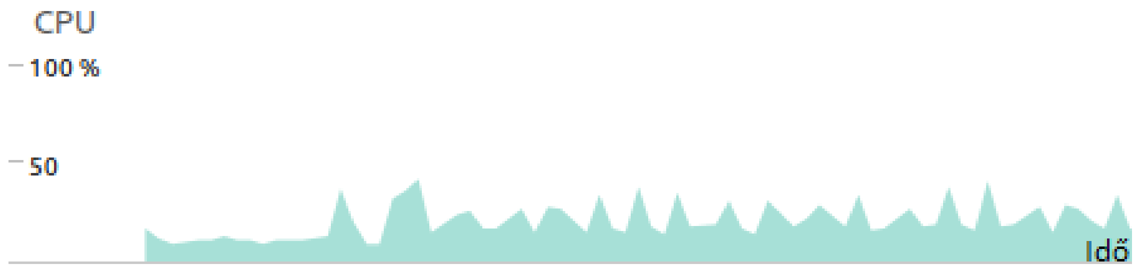
32. ábra: Nexus 5 CPU használata

A 32. ábra A Nexus 5 processzor használatát mutatja. Az átlagos terhelés az 35 – 50 % között mozog. és láthatóan predikció kiszámítási ciklusokat. Magasabb százalékot azért nem tud elérni mert egy többmagos processzorról van szó és a számítások párhuzamosítására nem képes.



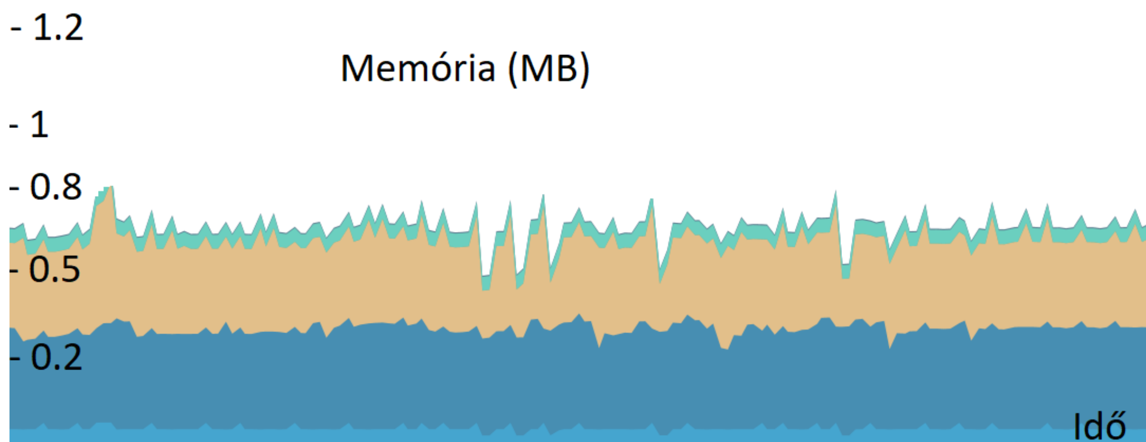
33. ábra: Nexus 5 memória terhelése

A 33. ábra a Nexus 5 memória használatát mutatja. A kék a nativ kód és Java memóriáját jelöli, a sárga a grafikus megjelenítéshez szükséges memória és a türkiz kék a kód mérete a memóriában. A Nexus esetében az alkalmazás összesen ~ 350 MB memóriát foglal.



34. ábra Nokia 7 Plus CPU terhelése

A 34. ábra Nokia 7 processzor terhelését mutatja. A terhelés a Nexus-hoz hasonló módon a 30 - 50 %-os terhelés körül mozog. A Nokia-ban is egy többmagos processzor dolgozik, így 50 % fölé nem is képes menni a terhelés.



35. ábra: Nokia 7 Plus memória terhelése

A 35. ábra a Nokia 7 Plus memória használatát mutatja. A kék a natív kód és Java memóriáját jelöli, a sárga a grafikus megjelenítéshez szükséges memória és a türkizkék a kód mérete a memóriában. A Nokia esetében az egész alkalmazás ~ 700 MB memóriát foglal. A több memória foglalása a képernyő nagyobb felbontása és a jobb kamera modulnak az oka. Mivel nagyobb méretű képekkel dolgozik és a megjelenítéshez is több erőforrásra van szüksége.

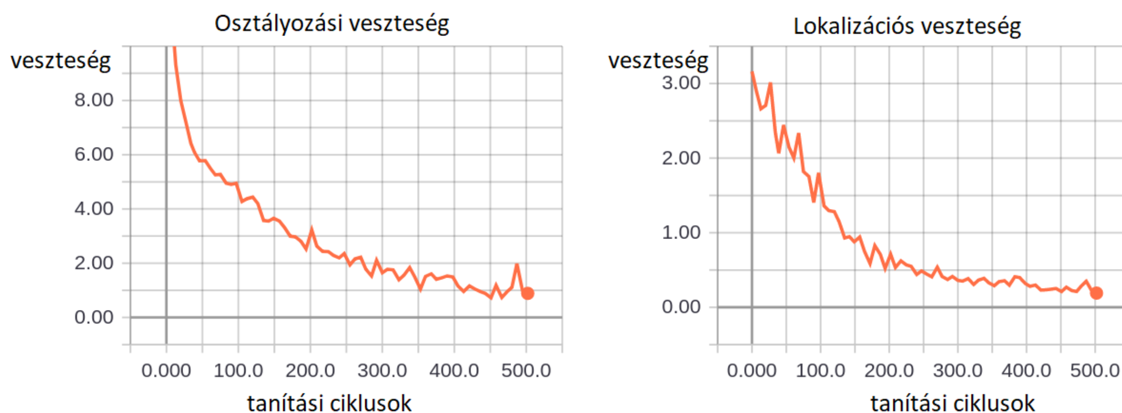
Tesztelés során amikor a detektálás több mint 30 percig futott, akkor a Nokia készülék túlmelegedett és teljesen lefagyott, semmire sem reagált.

Az alkalmazás lehetőséget nyújt új adatok rögzítésére, ami ebben az esetben videó felvételt jelent. A videó felvétel alatt a metaadat gyűjtés is elindul és ezeket a készülék háttértátára menti el. A videó 640x480-as felbontásban kerül rögzítésre és a metaadatok fájlba a készülék fontosabb szenzor adatai kerülnek.

A DriverPhone lehetőséget nyújt az adatok automatizált feltöltésére is, ezt a profil fül alól a „Ftp Upload” gombbal lehet megtenni. Az adatok felöltése után a lokális készülékről azok törlésre kerülnek, hogy ne foglaljon az alkalmazás feleslegesen helyet, illetve a feltöltés csak wifi kapcsolaton keresztül lehetséges.

A feltöltött nyers adatokból adattranszformációs lépések és egy konfigurálható mély neurális hálózat segítségével a rendszer képes előállítani a tanító adatokat. Az előállított adatokban kézzel is lehet javítani, ahogy azt a 31. ábra mutatja. A kulcs ebben a lépésben, hogy amennyiben kézzel nem szeretnénk javítani a szerver oldali szkript a nyers adatokból 1 gombnyomásra előállítja a tanításhoz használt TFRecords-ot.

A mély neurális hálózathoz a konfigurációs fájlokat létrehoztam, a tanító, tesztelő adatok és a hálózat megadásával a hálózat tovább tanítható. A tanítás során 500 cikluson át tanítottam és a veszteség majdnem elérte az 1-et (1.0954). A tanítás több mint 3 óráig tartott és ~ 100MB adattal tanítottam mert a tanításra használt számítógép egy közép kategóriás Lenovo laptop. A 36. ábra bal oldalán látható az osztályozási veszteség és jobb oldalon a lokalizációs veszteség csökkenése. Az osztályozás, amikor kép adott részén megállapítja, hogy milyen osztályú objektum van ott és a lokalizációs hiba magának az objektum helyének a megadása.



36. ábra: SSD MobileNet tanítása

A szerver alkalmazás biztosít egy webszervert, amin keresztül a DriverPhone alkalmazás képes letölteni és elmenti a mély neurális hálózat új verzióját.

Az dolgozat írásának utolsó hetében a Nokia 7 Plus-ra megérkezett az új Android verzió, amely a 9.0 Pie verziót viseli. Az alkalmazás az új verzió alatt is hiba nélkül, helyesen működött.

Az Android alkalmazásban a mély neurális hálózat cserélésének segítségével az újabb hálózatokon is lehetőség van a tesztelésre. A kódbázis módosítása nélkül az újabb SSD

MobilNet2 (ssdlite_mobilenet_v2_coco) hálózat már csak ~20 MB tárhelyet foglal és a Nokia 7 Plus-on a detektálás ~ 250 ms alatt történik meg, ami 4 FPS-t jelent.

7 Kiértékelés

Ebben a fejezetben az eddig bemutatott eredményeket értékelem ki objektíven.

A 15. ábra által illusztrált szoftver architektúra terv megvalósíthatónak bizonyult. Az egyes modulok funkcionalitásai implementálásra kerültek és a szoftver komponensek együttes működése alátámasztja, hogy a tervezett rendszerbe mély neurális hálózatok jól illeszthetőek és egy mély neurális hálózat életciklusát képes vezérelni a rendszer. Emellett a rendszer felépítéséből adódik, hogy más területeken is alkalmazható a struktúra.

A SSD MobilNet mély neurális hálózat típus jó választásnak bizonyult objektum detektálásra, a kis méretnek (~30 MB) és jó pontosságának köszönhetően. A hálózat a kamera képén is pontosan detektálta az autókat, buszokat és jelző lámpákat. Olyan esetekben volt hibás a detektálás, amikor rossz fényviszonyok voltak vagy az objektum nagyon távol volt még.

Az Android alkalmazásban a detektálás helyesen működött a kamera képére, azonban egy futás nagyjából ~400 ms ideig tartott, ami ~ 2-3 FPS-nek felel meg. Ez ahhoz még alacsony, hogy valós idejű visszajelzés kapjon a felhasználó. Detektálás hosszas használat mellett hajlamos a készülék túlmelegedni, bár ez gyártó és készülék specifikus probléma is lehet. A technológia jól működik, de bizonyos optimalizációs eljárások még szükségesek, hogy valós környezetben is használható legyen. Az olyan területeken, ahol nincs szükség valós idejű visszajelzésre, ott már alkalmazható lenne az alkalmazás.

Az alkalmazás rugalmas, támogatja a futás idejű modellcserét, amely újabb lehetőségeket nyit a felhasználási területek előtt. Emellett nem csak detektálásra, hanem adatgyűjtésre is képes a készülék. A gyűjtött adatok távoli tárhelyre való szinkronizálással a tanító adatok gyűjtését nagyban megkönnyíti. Az alkalmazás jelenleg 640x480-as videókat és metaadatok rögzít, jó lenne ezeknek is konfigurálhatóknak lennie, hogy milyen felbontáson, képfrissítésen és milyen sűrű metaadat rögzítéssel kerüljenek mentésre.

Szerver oldalon az automatizált tanító adatok előállítás használható adatokat képes előállítani, ezzel egy manuális lépést meggyorsítva. A rendszer szintén lehetőséget ad a személyes korrigálásra. A tanító adatok előállítása, azaz a videók feldolgozása nagyon magas erőforrás igényű feladat, ezen még lehet optimalizálni. A a metaadatok jelenleg nincsenek felhasználva a tanító adatok létrehozásánál mert az objektum detektálásnál ebben az esetben

nem voltak relevánsok. A képkockát leíró xml fájlba ezeket a metaadatokat érdemes lenne további feldolgozással beépíteni.

A szoftver komponensek megfelelően működtek együtt, a Python webalkalmazásból a jogosultság kezelés még hiányzik, hogy csak beregisztrált felhasználó legyen képes az új hálózatok letöltésére.

A szoftver rendszer a dolgozat elején kitűzött célokat megvalósította, így az eredményeket sikeresnek ítélem meg.

8 Jövőbeli tervek

A mély tanulás és képfeldolgozás területében sok lehetőséget látok. Mind kutatási, mind fejlesztési feladatokat is szeretnék a jövőben végezni.

Kutatási céljaim között szerepel a mély neurális hálózatok optimalizációja, amit egy fontos témának tartok. A korlátozott erőforrású készülékeken a mély neurális hálózatok futtatásához elengedhetetlen, hogy a hálózatok a lehető leghatékonyabbak legyenek.

A TensorFlow Java API továbbfejlesztését és kibővítését is fontos feladatnak tartom. A forráskód publikusan elérhető, így megpróbálnék én is hozzájárulni a fejlesztéshez.

Az Android alkalmazásban azt is konfigurálhatóvá lehetne tenni, hogy SSD Yolo vagy R-CNN típusú hálózatot szeretne futtatni a felhasználó. A TensorFlow Lite könyvtár [21] továbbfejlesztése esetén érdemes lesz majd áttérni rá, ezzel tovább gyorsítva a predikció számolását.

Az adatgyűjtés jelenleg videót vesz fel és mellé metaadatokat, ezt le lehetne cserélni egy erőforrás hatékonyabb módszerre. A háttérben elég lenne adott időközönként fotókat tárolni metaadattal és azt adott időközönként felölteni. Emellett a mikrofon hangjának feldolgozására is lehetne mély tanulás alapon valamilyen funkciót építeni.

Ezek mellett szeretném a forráskódot olyan formán dokumentálni és a kódbázist előkészíteni, hogy azt publikusan is elérhetővé lehessen tenni (természetesen az egyetem és a tanszék beleegyezésével).

9 Összefoglalás

A mély tanulás alapú informatikai megoldások egyre több helyen jelennek meg, okostelefonjaink is számos olyan funkcióval rendelkeznek, amely mögött egy mély neurális hálózat valósítja meg az üzleti logikát.

Ebben a dolgozatban egy olyan szoftver architektúrát mutattam be, amely egy általános megoldás, olyan rendszerekre nézve, ahol intelligens funkciókat mély neurális hálózatok hajtanak meg. A rendszer megvalósítás témájának vezetést támogató rendszert választottam. Ez egy igen fontos terület és a jövőben az okostelefonok sok aspektusban segíthetik a különböző járművekkel való utazást.

A rendszernek része egy Android alkalmazás, egy szerver oldali Python alkalmazás, egy FTP szerver és a Firebase szolgáltatása.

Az Android alkalmazás autók, buszok és jelző lámpák detektálását végzi a kamera képén folyamatosan. A konvolúciós mély neurális, amely a detektálást végezte, teljes egészében a készüléken fut. Másodperceként 2–3-szor volt képes az alkalmazás predikciót számolni. Emellett az alkalmazás képes felhasználó kezelésre, videó rögzítésre metaadatokkal együtt és tartalmaz konfigurálható beállításokat. A rögzített adatok egy kattintásra feltölthetőek egy FTP szerverre.

A szerver oldali alkalmazás a rögzített nyers adatokból tanító adatokat állít elő mély neurális hálózat segítségével automatizált módon és lehetőséget ad az SSD MobilNet konvolúciós mély neurális hálózat tovább tanítására. Az alkalmazás biztosít egy webszervert, ahonnan az újonnan tanított hálózat letölthető.

A rendszer megtervezésekor támasztott követelményeket sikerült megvalósítani. A munka során sok új kihívással álltam szemben, de úgy érzem nagyon sokat tanultam a rendszer megalkotása során és tudásom mind a mély tanulás mind a szoftver fejlesztés témakörében mélyült.

10 Irodalom jegyzék

[1] Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

[2] Bengio, Y., 2012, June. Deep learning of representations for unsupervised and transfer learning. In Proceedings of ICML Workshop on Unsupervised and Transfer Learning (pp. 17-36).

[3] Simard, P.Y., Steinkraus, D. and Platt, J.C., 2003, August. Best practices for convolutional neural networks applied to visual document analysis. In Document Analysis and Recognition, 2003. Proceedings. Seventh International Conference on (pp. 958-963). IEEE.

[4] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. and Jackel, L.D., 1989. Backpropagation applied to handwritten zip code recognition. Neural computation, 1(4), pp.541-551.

[5] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1), pp.1929-1958.

[6] Ioffe, S. and Szegedy, C., 2015, June. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In International Conference on Machine Learning (pp. 448-456).

[7] Zeiler, M.D. and Fergus, R., 2014, September. Visualizing and understanding convolutional networks. In European Conference On Computer Vision (pp. 818-833). Springer, Cham.

[8] Girshick, R., Donahue, J., Darrell, T. and Malik, J., 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In Proceedings of the IEEE Recognition (pp. 580-587).

[9] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A., 2016. You only look once: Unified, real-time object detection. In Proceedings of the IEEE Conference On Computer Vision and Pattern Recognition (pp. 779-788).

[10] Redmon, J. and Farhadi, A., 2017, July. YOLO9000: Better, Faster, Stronger. In 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 6517-6525). IEEE.

[11] Redmon, J. and Farhadi, A., 2018. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767.

[12] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y. and Berg, A.C., 2016, October. Ssd: Single shot multibox detector. In European Conference On Computer Vision (pp. 21-37). Springer, Cham.

[13] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J. and Zhang, X., 2016. End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316.

[14] Meseguer, J.E., Calafate, C.T., Cano, J.C. and Manzoni, P., 2013, July. Drivingstyles: A smartphone application to assess driver behavior. In Computers and Communications (ISCC), 2013 IEEE Symposium on (pp. 000535-000540). IEEE.

[15] Chen, C., Seff, A., Kornhauser, A. and Xiao, J., 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In Proceedings of the IEEE International Conference on Computer Vision (pp. 2722-2730).

[16] Oldalak: 85. oldal: Silberschatz, A., Galvin, P.B. and Gagne, G., 2014. Operating system concepts essentials. John Wiley & Sons, Inc..

[17] Number of apps available in leading app stores as of 1st quarter 2018 - https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores (letöltve: 2018.10.20.)

[18] Lin, T.Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P. and Zitnick, C.L., 2014, September. Microsoft coco: Common objects in context. In European Conference On Computer Vision (pp. 740-755). Springer, Cham.

[19] Everingham, M., Van Gool, L., Williams, C.K., Winn, J. and Zisserman, A., 2010. The pascal visual object classes (voc) challenge. International journal of computer vision, 88(2), pp.303-338.

[20] TensorFlow Java API - https://www.tensorflow.org/api_docs/java/reference/org/tensorflow/package-summary (letöltve: 2018.09.10)

[21] TensorFlow Lite - https://www.tensorflow.org/lite/ (letöltve: 2018.09.10)

[22] Firebase - https://firebase.google.com/ (letöltve: 2018.09.10)

[23] Vidstab - https://pypi.org/project/vidstab/ (letöltve: 2018.09.10)

[24] TensorFlow Object Detection API - https://github.com/tensorflow/models/tree/master/research/object_detection (letöltve: 2018.10.10)

[25] Detection Model Zoo - https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md (letöltve: 2018.10.21)

[26] LabelImg - https://github.com/tzutalin/labelImg (letöltve: 2018.09.10)

[27] TensorBoard - https://www.tensorflow.org/guide/summaries_and_tensorboard (letöltve: 2018.09.10)