



TANSZÉKVEZETŐ

DIPLOMATERVEZÉSI FELADAT

Bujtor Bálint

szigorló villamosmérnök hallgató részére

Objektumdetektálás megvalósítása autonóm járművekhez szintetikus képek segítségével

A gépi tanulás új módszerei az intelligens érzékelés számos területét forradalmasították az elmúlt évtizedben. Ezen módszerek közül külön figyelmet érdemel a mély tanulás (Deep Learning), amely a gépi tanulás legtöbb területén state of the art megoldásnak számít. A mély tanulás egyik legfontosabb alkalmazása a vizuális objektumdetektálás területe, amely a számítógépes látás témakörének egy alapvető feladata.

Az utóbbi néhány évben a mély tanulás kutatásának egyik fontos fókuszja a beágyazott rendszerekben történő alkalmazása (pl.: mobil robotok, okos eszközök), ami a mély neurális hálók hatalmas számításgigénye miatt nagy kihívást jelent. Ezek közül kiemelhető az autonóm járművek területe, ahol a környezetben előforduló objektumok valósidejű detektálása alapvető fontosságú.

A diplomatervezés során a hallgató feladata egy olyan algoritmus készítése, amely képes különböző a közutakon gyakorta előforduló objektumok pontos detektálására, valamint képes mesterségesen előállított tanító adatok alapján valós képekre általánosítani.

A hallgató feladatának a következőkre kell kiterjednie:

- Tanulmányozza át a téma releváns szakirodalmát. Vizsgálja meg, hogy más műhelyek milyen megoldásokat alkalmaznak.
- Készítsen rendszertervet egy megoldásra, amely képes közúti objektumok felismerésére.
- Végezze el az algoritmus fejlesztését és tanítását. Vizsgálja a szintetikus képek és a valós jelenetek közötti általánosítás minőségét.
- Tesztelje a megoldás pontosságát és hatékonyságát, valamint végezze el a tanuló algoritmus validációját.
- Vizsgálja a megoldást valósidejűség szempontjából.

Tanszéki konzulens: Szemenyei Márton

Budapest, 2020.09.08

Dr. Kiss Bálint
egyetemi docens
tanszékvezető

(Tanszéki levélpapír hivatalos lábrésze)



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Bujtor Bálint

OBJEKTUMDETEKTÁLÁS MEGVALÓSÍTÁSA AUTONÓM JÁRMŰVEKHEZ SZINTETIKUS KÉPEK SEGÍTSÉGÉVEL

KONZULENS

Szemenyei Márton

BUDAPEST, 2020

Tartalomjegyzék

Összefoglaló	4
Abstract.....	5
1 Bevezetés:.....	6
2 Irodalmi áttekintés.....	8
2.1 Intelligens látórendszerek	8
2.2 Felügyelt tanulás	19
2.3 Objektum detektálás	22
3 Specifikáció és tervezés.....	27
3.1 Feladatspecifikáció	27
3.2 Háló struktúra választása	29
4 Fejlesztés	30
4.1 Fejlesztőkörnyezet	30
4.2 Tanító adatbázis létrehozása	33
4.3 Tesztadatbázis létrehozása	43
4.4 Kiegészítő osztályozás	45
5 A tanítás módszertana és eredményei.....	46
5.1 Az értékeléshez használt mutatók	46
5.2 A tanítás módszertana	47
5.3 A táblaosztályozó neurális háló tanítása	56
5.4 Eredmények a tesztadatbázison	61
6 Összefoglalás, az eredmények értékelése:.....	66
6.1 Lehetőségek az eredmények további javítására	66
7 Irodalomjegyzék.....	68

HALLGATÓI NYILATKOZAT

Alulírott **Bujtor Bálint**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2020. 12. 09.

.....
Bujtor Bálint

Összefoglaló

Szakedolgozatomat a mesterséges intelligencia területéről, azon belül a mélytanulással kapcsolatban írtam. Feladatnak azt választottam, hogy lehetséges-e egy objektum detektáló algoritmust mesterségesen szintetizált képeken tanítani, majd pedig a betanított algoritmust valós környezetben sikeresen futtatni, alkalmazni.

A mesterséges intelligencia napjaink egyik igen felkapott témája, rengeteg kutatás történik ezen a tématerületen, számos területen alkalmaznak sikeresen mesterséges intelligencia alapú algoritmusokat, szolgáltatásokat. A mesterséges intelligencia egyre gyakrabban jelenik meg a mindennapi életben is, gondoljunk a részben vagy teljesen autonóm járművekre. Ezek mindennapi jelentősége hatalmas. Egy biztonságosan működő autonóm jármű esetén a közúti balesetek száma jelentős mértékben csökkenthető, hiszen a számítógép vezérlése által elimináljuk az emberi mulasztásból, hibából fakadó baleseteket.

Feladatom egy mesterséges intelligencia alapú algoritmus megalkotása volt, amely képes valós időben objektumokat detektálni. A mesterséges intelligencián alapuló algoritmusok sikeres betanításának alapja a megfelelő minőségű adatbázis megalkotása. Ebből kifolyólag egy új feladat során az adatbázis előállításával telik el a legtöbb idő. Az én feladatom az adatbázis előállításával eltelt időt eliminálja azáltal, hogy olyan szintetikus generált képeken tanítja be az algoritmust, aminek segítségével az adatbázis generálás automatikusan, emberi felügyelet nélkül történik meg, ami nagy mértékben csökkenti a fejlesztési időt. A feladatom során valós időben kell detektálnom embereket, járműveket, közúti táblákat, illetve jelzőlámpákat.

A szakedolgozatom során kielégítő mértékben sikerült betanítani a neurális hálót szintetikus generált képeken, a validációs eredmények egy ilyen adatbázis esetén megfelelőnek mondhatók. Mindazonáltal, ha csak tisztán szintetikus generált képeken tanítom be az algoritmust, akkor valós képeken nem fog tudni megfelelő pontossággal objektumokat detektálni a program.

Abstract

I wrote my dissertation on the field of artificial intelligence, including deep learning. My task was to analyze whether it is possible to teach an object detection algorithm on artificially synthesized images, and then to successfully run and apply the taught algorithm in a real-world environment.

Artificial intelligence is one of the bleeding edge topics of today, a lot of research is being done in this field. Artificial intelligence-based algorithms and services are successfully used in many fields. Artificial intelligence is increasingly appearing in everyday life as well, think of partially or completely autonomous vehicles. Their everyday significance is enormous. In the case of a safe autonomous vehicle, the number of road accidents can be significantly reduced, as we eliminate accidents caused by human negligence and errors by controlling the vehicle by the computer.

My task was to create an artificial intelligence-based algorithm capable of detecting objects in real time. The basis for the successful training of artificial intelligence-based algorithms is creating a database of appropriate quality. Because of that, the creation of the database takes most of the time. My task eliminates the time required to generate the database by teaching the algorithm on synthetically generated images. This allows the database to be generated automatically, without human supervision, which greatly reduces development time. During my task, I have to detect people, vehicles, road signs and traffic lights in real time.

During my dissertation I was able to train the neural network satisfactorily on synthetically generated images, the validation results are adequate for such a database. However, if I train the algorithm only on purely synthetically generated images, the program will not be able to detect objects with sufficient accuracy on real images.

1 Bevezetés:

A szakdolgozatom témája a felügyelt tanulás köre a mélytanuláson belül. Mint azt az előző hasábokon említettem, a felügyelt tanítás egyik nagy hátránya, hogy a tanító adatbázis előállítása és felcímkézése egy rendkívül költséges, időigényes folyamat.

Az én munkám erre a problémára kínál egy megoldást. A szakdolgozatom során ezt a tanító adatbázist szintetikus, számítógépesen generált képekből hozom létre, valamint az egyes képek felcímkézését is automatikusan végzem el. A cél egy olyan algoritmus létrehozása, amely képes egy járművön futó beágyazott környezetben, valós időben objektumokat detektálni.

A közúti környezetben használt mélytanuló algoritmusokat leginkább olyan problémák megoldására szokták alkalmazni, amelyek valamilyen módon javítják a vezető komfortérzetét, növelik a biztonságot vagy terhet vesznek le a vezetőről. Ilyen probléma lehet az automatikus vészfékezés, ha például egy gyalogos vagy biciklis halad át a jármű előtt. További példa lehet az automatikus távolságkövetés, illetve a tábla- és jelzőlámpa felismerés és ezáltal a közúti szabályok betartása. Éppen ezért munkám során azokat az objektumokat igyekeztem betanítani az algoritmusomnak, amelyek a fent bemutatott problémák szempontjából elengedhetetlenek. A mélytanuló algoritmusomat a következő objektumok felismerésére tanítottam be: gyalogos, jármű közlekedési tábla és jelzőlámpa.

Fontos szempont volt a feladatot és az algoritmust tekintve, hogy valós időben fusson ezáltal a mindennapi környezetben, közúton is használni lehessen. Olyan algoritmust kellett választanom és használnom, ami valós időben dolgozza fel a beérkező adatokat. Enélkül nem lehetne garantálni éles környezetben, hogy megfelelő időn belül képes lefutni és döntést hozni az algoritmus.

Többször említettem azt is, hogy az algoritmusnak a valós életben is meg kell állnia a helyét. Ezért azt is meg fogom vizsgálni a munkám során, hogy a betanított algoritmus, hogyan teljesít valós környezetben készült képeken. Ezért egy olyan adatbázison fogom tesztelni az algoritmusomat, amely ilyen képeket tartalmaz.

A bevezetés lezárásaként be szeretném összefoglalni, hogy a következő fejezetekben miket fogok bemutatni. Elsőként részletesen be fogok számolni a munkám tématerületéről, ami a mesterséges intelligencia és a számítógépes látás. Be fogok számolni arról is, hogy a mesterséges intelligenciának milyen módszereit használom a munkámhoz. Kitérek ennek a területnek a bemutatására, a jelenlegi kutatásokra, a legújabban használt metódusokra és módszerekre.

Miután részletesen bemutattam a tématerületemet rátérek a saját munkám részletezésére. Elsőként specifikálom a pontos feladatot és bemutatom a tervezés egyes lépéseit. Ezt követően rátérek a fejlesztésre, részletesen bemutatom az egyes lépéseket, az esetlegesen felmerült problémákat és a megoldásukat. Végül pedig bemutatom az eredményeket.

2 Irodalmi áttekintés

Jelen fejezetben részletesen bemutatom azt a szakterületet és tématerületet, amit a szakdolgozatom és diplomamunkám végzése során megismertem. Elsőként a szélesebb, általánosabb bemutatót adok a témáról, majd rátérek arra tématerületre, ami szorosabban kapcsolódik a szakdolgozatom munkámhoz.

2.1 Intelligens látórendszerek

Mint ahogy azt már említettem az eddigiekben, a szakdolgozatom témájának gerincét a számítógépes látás és a mesterséges intelligencia alkotja. Az én feladatom ennek a két tématerületnek a mezsgyéjén helyezkedik el, ezért mindkét tudományterületet meg kellett ismernem és először ezt a két területet fogom ismertetni.

2.1.1 Számítógépes látás

Elsőként a számítógépes látás témáját szeretném röviden bemutatni. A számítógépes látás olyan feladatokkal, problémákkal foglalkozik, amelyek során valamilyen vizuális, képi adatból szeretnénk információt kinyerni és ezt felhasználni. Ez az információ sokféle lehet. Egy adott képből kinyerhetünk távolság adatokat, megszámlálhatjuk, hogy egy objektumból mennyi van az adott képen, megszámlálhatjuk, hogy hányféle objektum van. A lehetőségek gyakorlatilag korlátlanok. A számítógépes látás célja, hogy ezeket az adatokat képekből, képsorozatokból algoritmikus eszközök segítségével kinyerjük egy másik döntéshozó alkalmazás vagy személy számára. A számítógépes látás egyik legnagyobb kihívása az, hogy óriási (akár milliós) adathalmazzal dolgozik és ezeknek az adatoknak gyakorlatilag végtelen féle kombinációja állhat elő. Éppen ezért ezeknek a problémáknak a megoldásához hatékony algoritmusokra van szükség. Sokáig a hatékonyság, illetve a nem megfelelő számítási kapacitás szabott gátat a számítógépes látórendszerek széleskörű elterjedésének.

A számítógépes látás további nehézsége, hogy mi emberek egy látott kép alapján könnyen tudunk információt kinyerni, ugyanakkor az a folyamat, amelynek során ezt az információt kinyerjük a képből részben vagy akár teljesen tudat alatt történik. Emiatt ezeket a képességeket nehéz algoritmikus lépések sorozatává konvertálni. Ezen felül sokszor más érzékszerveinket is felhasználjuk a döntésünk meghozatalához, amelyet

egy számítógépes látórendszer nem tud igénybe venni. Ezen problémák miatt a számítógépes látás területén sokszor használunk heurisztikus megoldásokat, gépi tanulásra vagy optimalizálásra épülő megoldásokat.

A számítógépes látó algoritmusokat sokféleképp szokás csoportosítani, mindazonáltal a legelterjedtebb megoldás az eljárások kimenete alapján történik. Eszerint megkülönböztethetünk képfeldolgozó (image processing) valamint a számítógépes látás (computer vision) algoritmusait. Előbbi esetében a kimenet egy olyan módon feldolgozott kép, ami a felhasználó szempontjából előnyösebb tulajdonságokkal rendelkezik. Ezen algoritmusok kimenete általában valamilyen további algoritmus bemenetét szolgáltatják, esetleg az emberi láthatóságot javítják. Utóbbi esetben olyan kimenetet kapunk, amely magasabb absztrakciós szinttel rendelkező információt szolgáltat. Ilyen információ lehet a képen lévő objektumok száma, színe mérete, de bármi egyéb is kinyerhető, amit szeretnénk.

A számítógépes látáson belül legtöbbször két csoportot lehet megkülönböztetni, attól függően, hogy milyen módszereket használ fel az információ kinyerésére. Megkülönböztethetjük azokat a módszereket, amelyek a gépi tanuláson (machine learning) alapszanak, valamint azokat a módszereket, amelyek nem használnak gépi tanuló algoritmusokat. Ezeket hagyományos látó algoritmusoknak nevezzük. A következőkben a gépi tanulásra alapuló algoritmusokat fogom bemutatni, mert ilyen algoritmusokat alkalmaztam a munkám során én is.

2.1.2 Gépi tanulás

A gépi tanulás a számítógépes algoritmusoknak olyan vizsgálata, kutatása, amelynek célja az, hogy olyan algoritmusok jöjjenek létre, amelyek képesek a tapasztalás segítségével fejlődni. [1] A gépi tanulást szokás a mesterséges intelligencia egyik részhalmazának is tekinteni. A gépi tanuló algoritmusok egy adott adathalmaz alapján készítik el az adott problémát megoldó modellt. Ezt az adathalmaz tanító adatbázisnak is szokás nevezni. Ezek a modellek olyan bemenetek alapján is tudnak döntést hozni, előrejelezni, amely bemenetre azelőtt nem voltak felkészítve. A gépi tanulást számos területen tudják használni, mint például az e-mailek szűrésénél vagy pedig a számítógépes látás területén is, mint ahogy azt később be is fogom mutatni.

A gépi tanulás segítségével olyan komplex problémák megoldása vált lehetségessé, amelyeket a hagyományos számítógépes látáson alapuló programokkal

egyáltalán nem, vagy csak óriási erőfeszítések árán sikerült volna megoldani. Az ilyen algoritmusok előnye, hogy segítségükkel olyan problémákat is meg lehet oldani, amelyeknek megoldását magunk sem ismerjük. Ezek a problémák általában magas absztrakciós szintűek. Ezen programok hátránya, hogy maga az algoritmus egyfajta feketedobozként funkcionál, nem teljesen tudjuk, hogy pontosan milyen folyamatok játszódnak le benne, hogyan jutott el az algoritmus oda, hogy megfelelő konfidenciával megoldjon egy adott feladatot. A programok rendszeresen alkalmaznak numerikus optimalizálást vagy statisztikai módszereket, emiatt nem lehetnek teljesen pontosak.

A gépi tanuló algoritmusokat többféleképpen lehet csoportosítani. Az egyik legelterjedtebb felosztás szerint megkülönböztethetünk felügyelt tanítást, felügyelet nélküli tanulást, valamint megerősítéses tanulást. Mindegyik csoportnak megvannak az előnyei és hátrányai is, éppen ezért más-más területeken terjedtek el sikeresen.

A gépi tanulás legegyszerűbb fajtája a felügyelt tanulás, amelynek során minden egyes bemenet mellé tartozik egy elvárt kimenet is. Ilyenkor azt várjuk el, hogy az algoritmus minél nagyobb arányban, minél pontosabban tanulja meg, hogy egy adott bemenethez mi a megfelelő, elvárt kimenet. Ennek a tanulási módszernek azonban számos korlátja, gátja van. A felcímkézett tanító adatbázis előállítása rendkívül költséges, időigényes feladat, minősége pedig nagymértékben meghatározza a tanítás sikerességét is. Másik hátránya pedig az, hogy csak olyan problémákat tudunk ezekkel az algoritmusokkal megoldani, amelyeknek a megoldását mi magunk is ismerjük.

Létezik felügyelet nélküli tanulás is, amelynek során a tanuló algoritmus számára csak bemeneti adatokat szolgáltatunk, elvárt kimeneteket nem adunk meg. Ilyenkor a cél az, hogy az algoritmus valamilyen séma alapján maga alkossa meg a kimenetet, valamilyen kompakt módon. Előnye ennek a módszernek, hogy az adatbázist rendkívül olcsón, akár automatikusan is elő lehet állítani, valamint olyan problémák is megoldhatók vele, amelyeknek megoldását mi sem ismerjük [22]. Emellett azonban ezek az algoritmusok lényegesen bonyolultabbak a felügyelt változatokhoz képest.

A gépi tanulás harmadik fajtája a megerősítéses tanulás (reinforcement learning). Az előbbi két módszertől ez több dologban is különbözik. A megerősítéses tanítás során az esetek döntő többségében összefüggő döntéseket, döntéssorozatot kell meghozni és a döntések helyességéről nem minden esetben kap visszajelzést. A másik különbség a visszajelzés mibenlétében rejlik. Ez a visszajelzés csak azt mondja meg, hogy mennyire volt jó az adott döntés, azt nem mondja meg, hogy mi lett volna a helyes

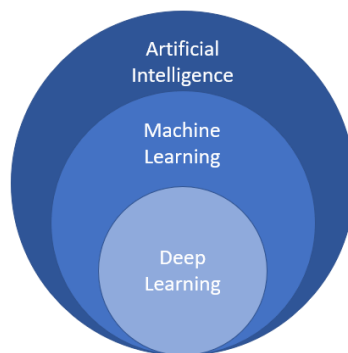
döntés [3]. A megerősítéses tanulás alkalmazási területére egy remek példák a különféle játékok (sakk, számítógépes játékok) vagy a járműirányítási feladatok.

A gépi tanulást, mint fogalmat először az IBM egyik munkatársa, Arthur Samuel használta, aki maga is egyik úttörője volt a témának. [4] Az érdeklődés a mintázatok felismerésének kutatásában az 1960-as években is folytatódott. Ekkor jelent meg Nilsson Learning Machines [5] című könyve, ami az időszak reprezentatív munkája volt és főleg a gépi tanulás mintázat felismerési lehetőségeit kutatta. A kutatás az 1970-es években is tovább folytatódott, ekkor jelent meg Duda és Hart [6] mintázat felismeréssel foglalkozó munkája, valamint a következő évtized elején odáig is eljutottka a kutatók, hogy képesek voltak felismerni egy 40 karakterből álló adatkészletet [7].

A gépi tanulás, mint tudományos kutatási terület, a mesterséges intelligencia utáni kutatás során alakult ki. A mesterséges intelligencia kutatásának kezdetén néhány kutató arra volt kíváncsi, hogy a gépek hogyan képesek adatok alapján tanulni. A kísérletek során számos irányból próbálták megközelíteni a problémát. Olyan szimbolikus módszereket dolgoztak ki, amit később neurális hálóknak kezdtek el hívni, ezeket leginkább perceptronok [55] és más modellek alkották.

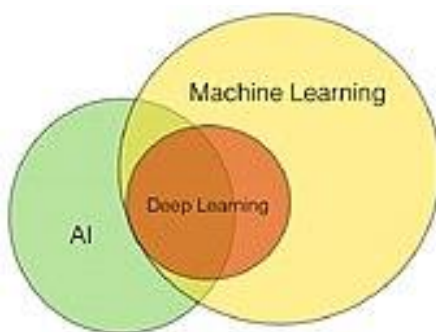
Mindazonáltal a mesterséges intelligencia kutatása egyre inkább a tudás és logika alapú megközelítés felé tolódott, ami egy törést, távolodást jelentett a gépi tanulás módszereitől. A valószínűségi modellek nagy hibája volt mind elméleti, mind gyakorlati szinten az adatok reprezentálásának és összegyűjtésének kérdése [8]. Többek között ez is volt az oka annak, hogy mind a mesterséges intelligencia kutatások, mind a számítástudomány felhagyott a neurális hálók alkalmazásával.

Ezen történések együttesen vezettek oda, hogy az 1990-es években a gépi tanulás, mint önálló tudományág elkezdjen növekedni és fejlődni. Mindinkább az lett a terület célja, hogy a mesterséges intelligencia elérése helyett olyan problémákkal, kérdésekkel foglalkozzon, hogy hogyan lehet adott problémákat természetes, praktikus módokon megoldani. Ezek hatására kezdett el a gépi tanulás olyan módszereket alkalmazni, mint a statisztika és a valószínűségszámítás [9].



ábra 1: a gépi tanulás a mesterséges intelligencia része, saját kép

Napjainkban találkozhatunk azzal az állítással is, hogy a gépi tanulás továbbra is a mesterséges intelligencia egyik részhalmaza [11], ugyanakkor olyannal is, hogy a gépi tanulásnak vannak olyan területei is, amelyek nem tartoznak bele a mesterséges intelligencia tudományterületébe [12].



ábra 2: a gépi tanulás és a mesterséges intelligencia csak részben fedik egymást, forrás: [10]

A mesterséges intelligencia és a gépi tanulás közötti különbséget legjobban talán Judea Pearl fogalmazt meg *The Book of Why* [13] című könyvében. Ebben azt mondja, hogy míg a gépi tanulás passzív megfigyelések alapján tanul és jósol, addig a mesterséges intelligencia valamilyen ügynököt használ arra, hogy interakcióba léphessen a környezetével, tanulhasson, akciókat hajthasson végre, hogy a célját minél nagyobb arányban érje el.

2.1.3 Deep learning

Ezen a ponton térnék rá a mélytanulásra, mint témakörre. Az előző pont ábráin láthattuk, hogy a mélytanuló algoritmusok mind mesterséges intelligencia alapú, mind gépi tanulásra alapuló algoritmusokat, módszereket alkalmaznak (ábra 1) (ábra 2). Mint

látható a mélytanulás mindkét tudományterületből merít ötleteket, metódusokat, amelyet olyan sikeresen tesz, hogy a mélytanuló algoritmusok napjainkban a vezető módszert jelentik a gépi tanulás területén [2]. A technika lényege, hogy tanítópéldák segítségével tanítjuk meg a számítógépnek. A mélytanulás olyan technológiák alapja, mint az önvezető autók, hangfelismerő rendszerek (Shazam) vagy akár tőzsdei elemzőprogramok.

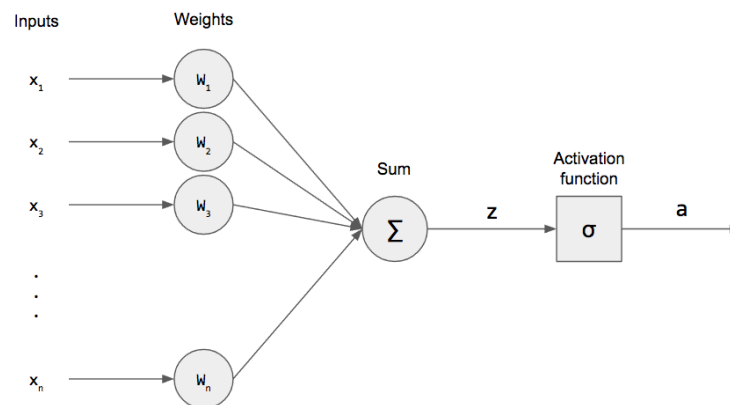
Annak ellenére, hogy a deep learning népszerűsége csak az utóbbi néhány évben kezdett meredek felívelésbe, maga a téma már régebb óta ismert. Az első működő, felügyelt, több rétegű perceptront 1967-ben publikálta Alexey Ivakhnenko és Valentin Lapa [14][16]. A mélytanulás fogalmát 1986-ban Rina Dechter [15][16] vezette be a mesterséges intelligenciával foglalkozó kutatások körébe. A technológia elterjedésének ekkor még elsősorban a rendkívüli adatmennyiség tárolásához és feldolgozásához szükséges tárolási és számítási kapacitás hiánya szabott gátat. Az utóbbi néhány év, évtized robbanásszerű fejlődése az informatikában, a számítási és tárolási kapacitások megnövekedése és az egyre több tárolt adat utat nyitott a deep learning térhódításának.

A deep learning alapjául szolgáló mesterséges neurális hálók ötletét a különböző biológiai rendszerekben megtalálható elosztott kommunikációra és adatfeldolgozásra képes sejtek, pontok inspirálták; gondoljunk csak az emberi agy neuronjaira. Ugyanakkor a szigorúan vett biológiai agytól néhány tulajdonságban különbözik is. Míg egy biológiai agy egy élő, változó organizmus, addig egy mesterséges neurális háló statikus és szimbolikus fogalom, amely a működésbeli hasonlóságokra fókuszál.

A mélytanulás kifejezésében szereplő mély melléknév arra utal, hogy egy ilyen struktúrában több, hasonló réteget használunk fel, ezek a rétegek az úgynevezett perceptronok. Egy perceptron képes eldönteni, hogy a számára megadott számokból álló vektor egy adott osztályhoz, csoporthoz tartozik-e vagy sem, amelyet a kimenetén egy bináris számmal jelez (ábra 3). A korai kutatások megmutatták, hogy egy lineáris perceptron nem képes univerzális osztályozásra. Továbbá azt is, ha az ilyen lineáris rétegekből álló hálót ellátjuk egy nemlineáris, aktivációs függvényvel, akkor az így kapott perceptron már bármilyen kompakt halmazon folytonos függvényt képes tetszőleges pontossággal megközelíteni, feltéve, ha nem korlátozzuk a rejtett réteg szélességét. Ezt nevezik univerzális approximációnak [56].

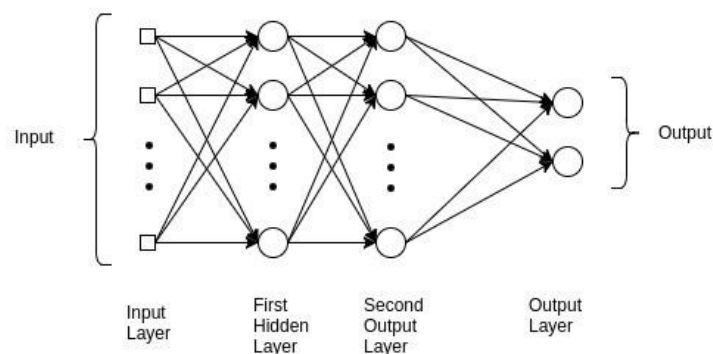
A perceptron egy olyan algoritmus, amelyet a felügyelt tanulás területén alkalmaznak bináris osztályozásra. A bináris osztályozók képesek eldönteni azt, hogy

egy bemenet, amit általában egy vektor reprezentál, egy adott osztályba tartozik-e. A perceptron tulajdonképpen egy egyrétegű neurális háló, amelyet négy főbb érsz alkot. Ezek a bemeneti értékek, a súlyok és ofszet, az összegzés, valamint az aktivációs függvény. A folyamat azzal kezdődik, hogy a bemeneti vektorértékeket megszorozzuk a hozzájuk tartozó súlyokkal, majd pedig összeadjuk ezeket a szorzatokat. Itt opcionálisan lehetőség van az ofszetet is hozzáadni, amivel valamilyen irányba mozdíthatjuk el az összeget. Ezt követi az aktivációs függvény, ami esszenciális a kimenet megfelelő értéktartományba konvertálásához (0, 1), (-1, 1). (ábra 3)



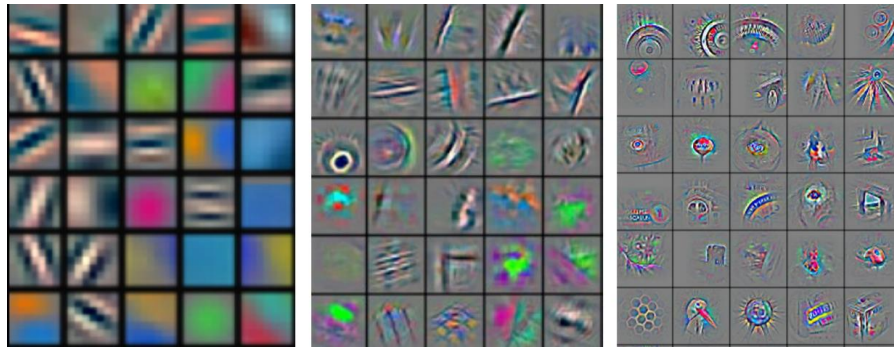
ábra 3: Egy perceptron felépítése, az aktivációs függvény a nemlinearitást hivatott reprezentálni, forrás: [40]

Ahhoz, hogy ne csak bináris értékeket tudjunk osztályozni nem elég egy darab perceptron. Ahhoz, hogy egy n különböző csoportból, osztályból álló adathalmazt fel tudjunk ismerni n darab bináris perceptron egymás mellé helyezésére van szükségünk, amelyek közül mindegyik egy-egy osztály, csoport felismeréséért felel (ábra 4).



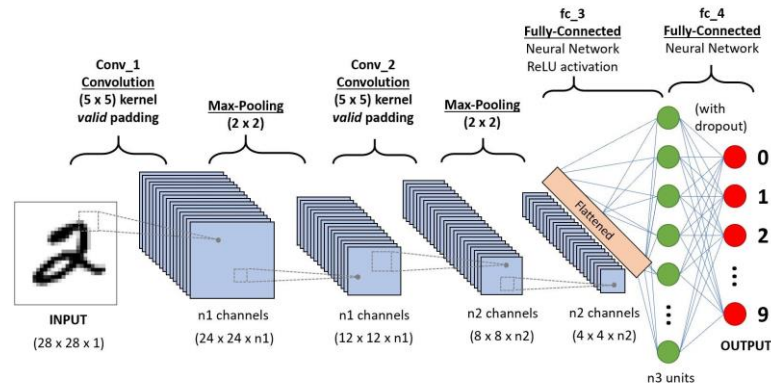
ábra 4: Egy többretegű perceptron felépítése, forrás: [41]

Az ilyen perceptron rétegekből felépülő neurális háló működését szokás úgy bemutatni, mint hierarchikusan egymásra épülő, egyre komplexebb tulajdonságok kinyerésére képes rétegeket (ábra 4). Például képfeldolgozás esetén az alsóbb rétegek felelnek vonalak, görbék felismeréséért, az azt követő réteg ezekből a formákból alkot alakzatokat (kör, négyzet háromszög). Az ezt követő réteg pedig ezekből a formákból alkotja meg a felismerni kívánt alakzatot, és végül dönti el, hogy az megtalálható-e a bemenetként megadott képen (ábra 5).



ábra 5: A különböző absztrakciós szinteket reprezentáló képek deep learning esetén (balról jobbra egyre növekvő absztrakciós szintek) [17]

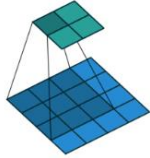
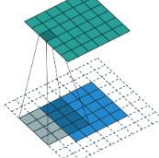
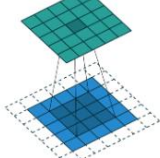
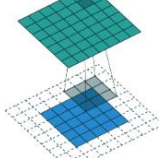
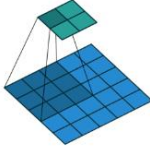
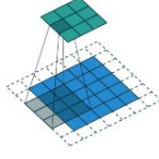
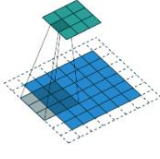
Az előző bekezdésekben röviden bemutatam a perceptron modellt és azt, hogy ebből az alap építőköből hogyan és milyen struktúrákat lehet felépíteni. Mindazonáltal egy ilyen többrétegű perceptron modellel nehézkes lenne olyan képosztályozást vagy objektumdetektálást végrehajtani, amilyen feladatokkal a mai neurális hálók szembekerülnek. A képfeldolgozás témakörén belül, a konvolúciós neurális hálók váltak rendkívül elterjedté. A CNN-ek (Convolutional Neural Networks) [18] a konvolúció műveletével nyerik ki az információt a feldolgozandó képekből. Az első konvolúciós neurális hálót Kunihiko Fukushima alkotta meg 1980-ban [18]. A kép egy adott részletén a konvolúciót elvégezve, abból magasabb szintű információ nyerhető ki, ezzel pedig hierarchikus szinteket lehet felépíteni, hogy valamilyen absztrakt eredményt érthessünk el (ábra 6).



ábra 6: Konvolúciós neurális háló struktúra, forrás: [42]

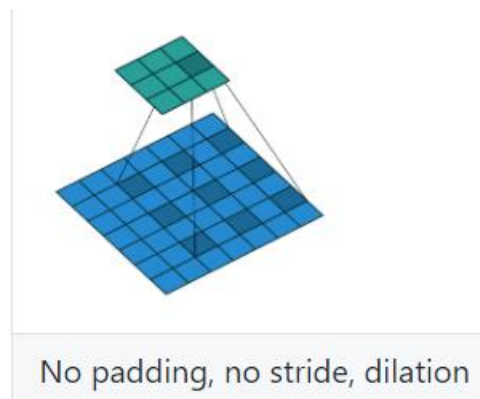
A konvolúciós architektúra előnye egy lineáris neurális hálóval szemben, hogy az egyes vektorelemek (pixelértékek) egymáshoz képesti pozícióját képes felhasználni, míg ezt egy lineáris réteg nem volt képes kezelni, hiszen az egy egyszerű sorvektorként dolgozta azt fel. Az architektúra további előnye egyéb algoritmusokhoz képest, hogy az egyes képeket relatíve kicsi előfeldolgozással tudja fogadni. A konvolúciós rétegek fontos hátránya azonban, hogy az egyes részletek egymáshoz képesti helyzetét még így sem tudja teljesen jól kezelni (például egy arc esetén a szem, száj, orr egymáshoz képesti helyzetét nem tudja jól felismerni). A neurális hálók hátránya általánosságban, hogy a tanítás numerikusan instabil lehet és a betanított háló csak olyan objektumokat képes felismerni, amelyre a tanítás során megtanítottuk (ez minden felügyelt tanulás alapú neurális hálóra igaz). Egy felügyelt tanulásra alapuló módszer nem képes tehát saját döntéseket, saját címkéket generálni, olyan információt alkotni, amelyet a megalkotói ne tudtak volna.

Egy ilyen konvolúciós réteg a konvolúciós szűrőkhöz hasonlóan működik. Ez a réteg a bemeneti képen, ami általában szürkeárnyaltos vagy színes, N darab különböző konvolúciós szűrőt futtat végig, amelynek eredményeképpen egy N csatornás képet kapunk. Az ezt követő konvolúciós réteg pedig már ezzel az N csatornával dolgozik tovább. Az, hogy mekkora legyen egy ilyen konvolúciós szűrő és hogy hány csatornája legyen az az adott feladattól függő, úgynevezett hiperparaméter. A hiperparaméterek optimalizálásának segítségével tudjuk a tanulást gyorsítani és a háló pontosságát növelni.

			
No padding, no strides	Arbitrary padding, no strides	Half padding, no strides	Full padding, no strides
			
No padding, strides	Padding, strides	Padding, strides (odd)	

ábra 7: konvolúció különböző paraméterek mellett, forrás: [19]

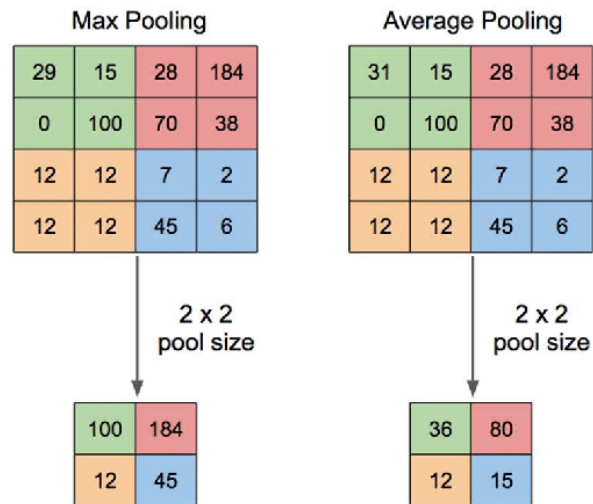
Mint ahogy az a fenti ábrán is látszik, a konvolúciónak számos paramétere van. Ezek közül röviden bemutatnám a legfontosabbakat. A keret (padding) segítségével megadható, hogy legyen-e a kép körül egy üres pixelekből álló keret, ami a konvolúcióba beleszámít-e vagy sem. A lépésközzel (stride) azt lehet megadni, hogy a két konvolúció között hány pixel különbség legyen. Egyes lépésközzel értelemeszerűen egy pixelt lépünk odébb, míg kettővel kettő pixelyit és így tovább (ábra 7). Az utolsó fontos paraméter a dilatació (ábra 8), ami azt adja meg, hogy a konvolúció széthúzza-e a pixeleket úgy, hogy nem szomszédos pixeleket vesz be az adott konvolúciós ablakba.



ábra 8: dilatació konvolúciós szűrő esetén, forrás: [19]

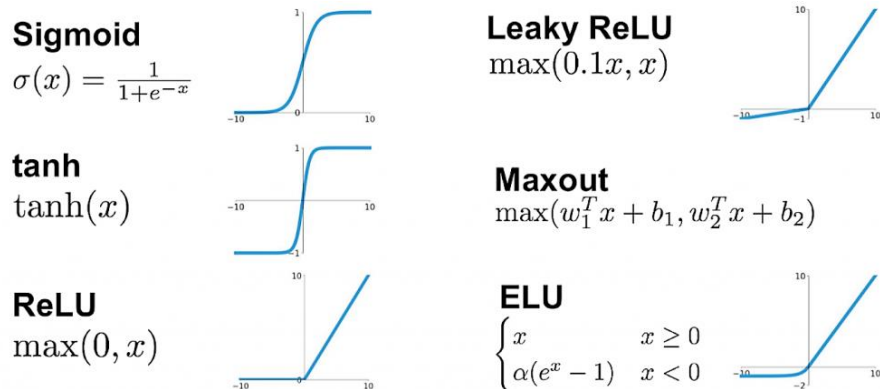
A konvolúció, mint művelet tehát számos hasznos tulajdonsággal bír, különösen olyan térbeli információk kinyerésére, mint amilyenekkel a képfeldolgozás témakörében találkozhatunk. Mindazonáltal nem érdemes csak konvolúciós rétegeket alkalmazni, ha egy modellt szeretnénk felépíteni, ugyanis minél több ilyen konvolúciós réteget helyezünk egymás mögé, annál nagyobb lesz a konvolúció kimenetén kapott struktúra

mérete. Ennek elkerülése végett érdemes néhány rétegenként csökkenteni a tömb méretét. Ezt megtehetjük úgy, hogy a konvolúció lépésközét növeljük, vagy beiktatunk egy tömörítő, úgynevezett pooling réteget is. Egy ilyen művelet során valamilyen módszer szerint a tömb aktuális részét egy adott számmal helyettesíti. Ez a módszer legtöbbször vagy az ablakban szereplő értékek átlagának számítása, vagy a maximális érték választása (ábra 9).



ábra 9: max (balra) és átlagoló (jobbra) pooling, forrás: [20]

Az utolsó fontos komponens a konvolúciós neurális háló struktúrákban az aktivációs réteg. Ilyen aktivációs rétegek általában egy konvolúciós és lineáris réteget követően helyezkednek el, ugyanis mind a konvolúció mind a lineáris réteg lineáris műveleteket alkalmaznak, ezért fontos beékelni nemlinearitást a perceptron bemutatásánál említett okokból. Amikor elkezdtek konvolúciós neurális rétegeket alkalmazni előszeretettel használtak szigmoid vagy hiperbolikus tangenst (ábra 10). Azonban ezek deriváltja az értelmezési tartományuk szélén alacsony, ami ahhoz vezet, hogy a deriváltak nullához fognak konvergálni emiatt pedig elrontják a tanítást. Emiatt mindinkább a ReLU (Rectified Linear Unit) kezdett elterjedni (ábra 10), amely kiküszöbölte ezt a problémát és a deriváltjának számítása is gyorsabb. Azóta a ReLU aktivációt tovább finomították, hogy még hatékonyabban működhessen (ábra 10).



ábra 10: különböző aktivációs függvények – balról lefelé, oszloponként: szigmoid, hiperbolikus tangens, ReLU, Leaky ReLU, Maxout és ELU, forrás: [21]

Az eddig bemutatott rétegek felhasználásával már fel lehet építeni egy olyan konvolúciós neurális háló struktúrát, amivel már nagy pontossággal lehet megoldani könnyebb feladatokat. A következő alfejezetben be fogom mutatni, hogy jelenleg milyen objektumdetektáló neurális hálókat használnak, hogy néz ki ezek szerkezete, milyen előnyeik és hátrányaik vannak.

2.2 Felügyelt tanulás

A diplomamunkám során én is felügyelt tanulást hajtok végre, ezért a következőkben ezt fogom bemutatni. A felügyelt tanulás a gépi tanuló algoritmusok egyik megközelítése egyik, amelynek során az algoritmust egy nagy adatbázis segítségével tanítjuk meg adott feladatra. Az adatbázis minden egyes eleméhez tartozik egy címke, amin azt jelezzük a neurális hálónak, algoritmusnak, hogy mi az adott bementre elvárt, a háló által adandó kimenet. Az algoritmus arra lesz képes, hogy meghatározza, hogy a tanítás során megadott osztályok közül jelen van-e a képen valamelyik és ezt minél nagyobb pontossággal tegye meg. A következőkben röviden összefoglalom, hogy hogyan működik a legtöbb objektum osztályozó algoritmus, ezután pedig bemutatok néhány előszeretettel használt neurális háló struktúrát.

A megerősítéses tanulás elvén működő gépi tanuló algoritmusok célja az, hogy az elvárt kimenetek függvényében a neurális háló minél nagyobb pontossággal tudja eltalálni azt, hogy az aktuálisan a bemenetére adott kép melyik osztályba tartozik. A háló kezdetben valamilyen véletlen kimenetet ad, hiszen nincs megtanítva a helyes kimenetre. Ezt valamilyen tanító függvény segítségével lehet megtenni.

A tanítás során az elvárt kimenet és a háló által adott kimenet között valamilyen hibafüggvény segítségével adjuk meg (például keresztentropia vagy átlagos négyzetes hiba) [23], hogy mekkora hibát vét a háló. Ez a hiba a háló belső változói alapján (súlyok, ofszetek) kerül kiszámításra. Számos ilyen hibafüggvény létezik, mindegyik másféle feladattípus esetén hatékony. A cél az, hogy ezt a hibafüggvényt minimalizáljuk, aminek következtében a háló pontossága nőni fog.

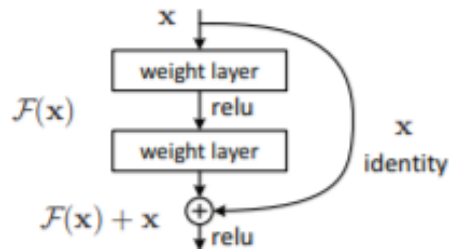
Azt, hogy milyen mértékben és hogyan változtassuk a háló súlyait és ofszetjeit a gradiens módszer segítségével lehet kiszámolni. A gradiens módszer során a neurális háló súlyainak és ofszetjeinek parciális deriváltjait számítjuk ki a láncszabály segítségével, a háló kimenetétől a bemenet felé haladva. Ezt a metódust backpropagation-nek is nevezik.

Miután kiszámoltuk a gradienst, már csak azt kell megadni, hogy melyik irányba és mekkora mértékben mozduljunk el valamilyen optimalizáló algoritmus segítségével tudjuk megtenni. Ezek az optimalizáló algoritmusok egy függvényt maximalizálnak, esetünkben ez a függvény a hibafüggvény. Mivel a hibafüggvény súlyok szerinti gradiense azt adja meg, hogy melyik irányban növekszik a hiba a leggyorsabban, ezért a maximális gradienshez képest pont ellentétes irányba lépünk el, ezzel pedig épp minimalizálni fogjuk a hibafüggvényt. Ahogy hibafüggvényekből, úgy optimalizáló függvényekből is sokféle létezik, a YOLOv3 például Adam optimalizálót [24] használ.

A félév során végzett munkám keretében számos modern neurális háló struktúrával is megismerkedtem, néhányat ezek közül ki is próbáltam, mielőtt a végleges algoritmust elkezdtem volna használni és megérteni. A megismert struktúrák között szerepel a ResNet [25] és az InceptionNet [26][27], melyek közül a ResNettel foglalkoztam részletesebben.

2.2.1 ResNet

Az általános egymás után helyezett konvolúciós rétegekkel ellentétben a ResNet rendelkezik úgynevezett reziduális kapcsolattal, amely egy előre csatoló ágot jelent a struktúrában (ábra 11). Az előre csatolások révén elérhető, hogy nagyon mély neurális háló struktúrákat lehessen alkotni úgy, hogy azok megőrizték a numerikus stabilitásukat. Így sokkal mélyebb struktúrákat lehetett alkotni az eddigi, reziduális réteggel nem rendelkező megoldásokhoz képest.



ábra 11: Reziduális építőegység, forrás: [25]

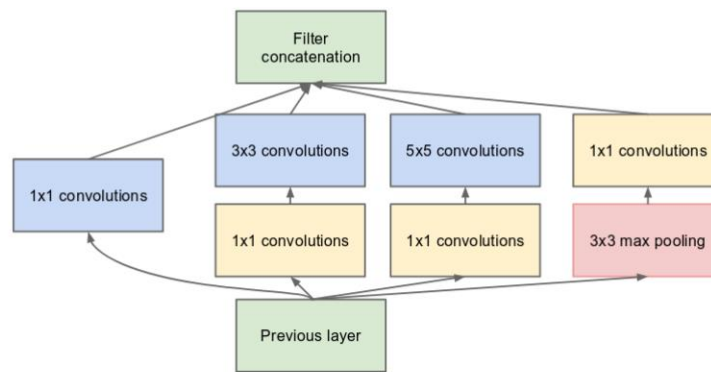
2.2.2 Inception háló

A másik modern konvolúciós neurális háló struktúra, amivel tüzetesebben foglalkoztam az Inception Net (GoogleNet). Egy képen szereplő osztály detektálásakor a számunkra fontos képrészlet rendkívül változatos lehet a méretét tekintve (ábra 12). Éppen ezért a megfelelő kernelméret kiválasztása rendkívül nehéz lehet. Továbbá a rendkívül mély neurális hálók hajlamosabbak az overfitting jelenségére (a tanító adatsoron remek pontosság, de a validációs és tesztadatokon rossz eredmény), valamint a számítási kapacitást tekintve is drágábbnak minősülnek.



ábra 12: A kép különböző részét kitöltő, ámde ugyanazt az osztályt (kutya) ábrázoló képek

Az Inception Net struktúra erre kínál megoldást. Az elgondolás alapja az, hogy egy adott szinten nem egy konvolúciót hajt végre, hanem egyazon szinten több különböző kernel mérettel is elvégzi a konvolúciót (továbbá egy max poolingot is végrehajt), majd az elvégzett konvolúciók kimenetét összefűzi és ezt adja tovább a következő ilyen rétegnek. Ahhoz, hogy a számítási kapacitást csökkentse, a réteg egy extra 1x1-es konvolúciót is végrehajt a rétegen belül, ezzel limitálva a bemeneti csatornaszámot (ábra 13). Noha ez ellentmondásosnak tűnhet, hiszen pluszműveletet jelent, egy 1x1-es konvolúciót sokkal kevésbé költséges elvégezni, mint egy 5x5-s konvolúciót.



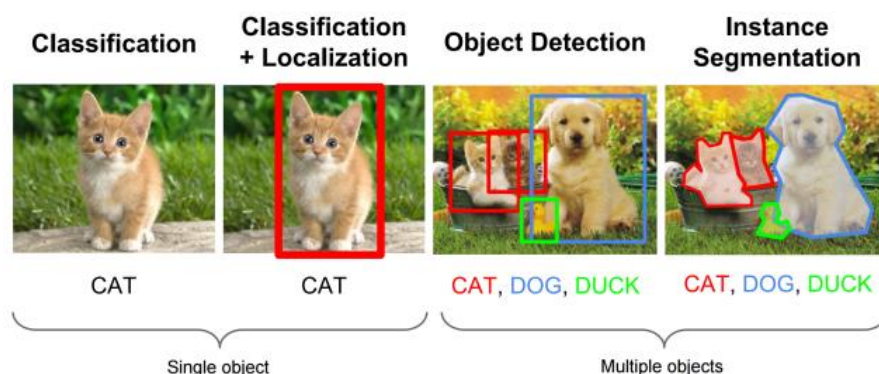
(b) Inception module with dimension reductions

ábra 13: Az inception háló (Inception v1 – GoogleNet) építőegysége dimenzióredukcióval ellátva, forrás: [23]

A teljes neurális háló ilyen építőegységekből épül fel, természetesen további kiegészítésekkel, amelyekre most nem térek ki, mert nem képezik szerves részét a beszámolómnak. Az struktúrát a későbbiekben tovább is bővítették, finomították, így született meg az Inception Net v2 [37] és v3 [38].

2.3 Objektum detektálás

Az előző bekezdések során ismertetett neurális hálók objektum osztályozó neurális hálók voltak, ami azt jelenti, hogy rendkívül jó pontossággal képesek megmondani, hogy egy adott osztály van-e a képen vagy sem. Azonban vannak jelentős hátrányaik, amelyek az én témám esetében nagy jelentőséggel bírnak. Nem képesek kezelni azt, ha egy adott képen több osztályból is szerepelnek objektumok, valamint arra sem képesek, hogy a képen belül meghatározzák, hogy hol helyezkednek el ezek az objektumok (ábra 14).



ábra 14: Különböző objektum meghatározó módszerek. Balról jobbra: osztályozás, osztályozás + lokalizálás, detektálás, szegmentálás. Forrás: [39]

A témámhoz tehát egy olyan neurális hálóra struktúrára volt szükségem, amely képes egyszerre több objektumot felismerni és azok helyzetét meghatározni. A kiválasztott algoritmus a YOLO (You Only Look Once) [28][29] objektum detektálásra képes struktúra volt, ugyanakkor a félév során más, hasonlóan működő struktúrákkal is megismerkedtem. A következőkben bemutatom az egyes objektum detektáló módszereket, néhány példát, ezután pedig a YOLO architektúrát és működését fogom bemutatni részletesebben.

A különféle objektum detektáló algoritmusokat két csoportba lehet sorolni. Az első csoport a régiójavaslat alapú algoritmusok. Ezeket az algoritmusokat működésük szempontjából két lépésre lehet bontani. Első lépésként kiválasztanak a képen több, az algoritmus alapján érdekesnek talált régiót, második lépésként pedig ezeket a régiókat osztályozzák egyesével valamilyen CNN segítségével. Ez a módszer nem kifejezetten gyors, hiszen minden kiválasztott régión le kell futtatni az osztályozó algoritmust, ugyanakkor pontosabbak lehetnek, mint a másik alcsoport. Ehhez a kategóriához tartozik például a régió alapú neurális háló (Region-based convolutional neural network (RCNN)) [31] és ennek a továbbfejlesztett verziói, a Fast-RCNN [32], Faster-RCNN [33], valamint a legújabb Mask-RCNN [36]. Egy másik példa erre a típusra a RetinaNet [34].

A másik csoport a regresszió alapú algoritmusok. Ezek az algoritmusok ahelyett, hogy kiválasztanák a kép egy adott részét és azon végeznék el az osztályozást, az egész képen próbálják meg előrejelezni az objektumosztályokat és az azt övező bounding box-okat, mindezt egy egyszeri futtatás során. A két leginkább elterjedt algoritmus ebben a kategóriában a YOLO (You Only Look Once) [28][29] algoritmus, illetve az SSD (Single Shot Multibox Detector) [35]. Ezeket az algoritmusokat elsősorban valós idejű alkalmazások során használják, hiszen a pontosságbeli hátrányukat a sokkal gyorsabb futási sebességükkel kompenzálják.

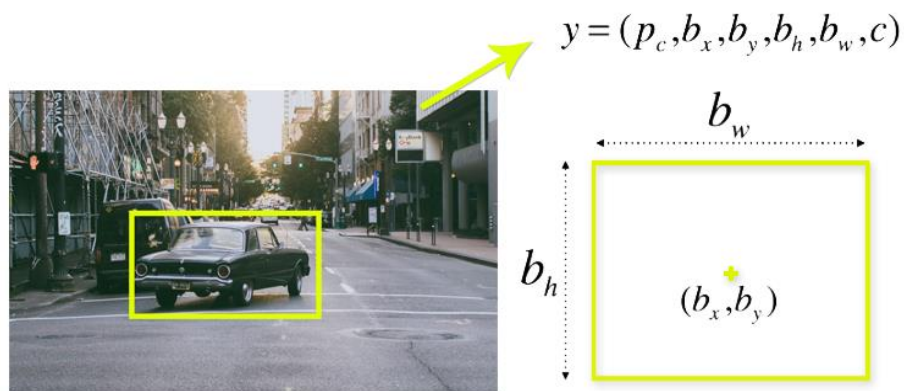
A fenti megfontolások alapján tehát egy regresszió alapú, valós időben futtatható algoritmusra volt szükségünk, a választásunk pedig a YOLO algoritmusra esett. A YOLO algoritmus egy hatékony, valós időben futó objektum detektáló algoritmus, amelynek első verzióját Joseph Redmon és társai publikáltak 2015-ben. [28] Az algoritmus egy objektum detektálásra képes metódus, tehát egy képen több osztály több objektumát is képes észlelni, és azok köré egy úgynevezett bounding box-ot (körülvevő keretet/ dobozt) helyezni. Később több, további verzióját, továbbfejlesztését

is publikálták az algoritmusnak, a legutóbbi a YOLOv3 [29] és YOLOv4 [57], amit 2018-ban illetve 2020-ban publikáltak. A szakdolgozatomhoz a v3-as verziót használtam, így a következőkben ennek a működését fogom részletesebben bemutatni.

Mint a legtöbb modern mély neurális háló, a YOLO is konvolúciós neurális háló, ezen belül egy teljesen konvolúciós neurális háló (FCN - fully convolutional neural network), ami azt jelenti, hogy csak konvolúciós rétegekből épül fel. Összesen 53 konvolúciós réteget tartalmaz, skálázott (strided) rétegekkel és előrecsatoló (reziduális) kapcsolatokkal.

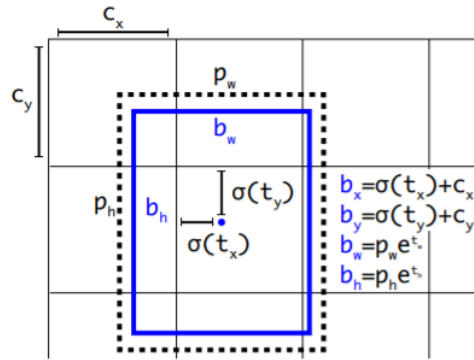
Az eddig megismert struktúrák esetén, a konvolúciós rétegek által megtanult kimeneti valószínűség vektort valamilyen lineáris réteg adja ki. A YOLO esetén ez nem így van, a detektálást egy 1x1-es konvolúciós réteg végzi el, tehát ugyanakkora lesz a kimeneti aktivációs tömb, mint az azt megelőző réteg. Ennek a kimeneti képnek a csatornaszáma összesen $(B * (5 + C))$, ahol B az adott cella által meghatározott bounding box-ok száma (YOLOv3 esetén ez a szám 3), C pedig az adatbázisban szereplő osztályok száma. Több bounding box-ot azért használunk egy cellán belül, hogy több objektumot is észlelni tudjunk, amelyek ugyanabba a cellába esnek.

A végső célunk az, hogy egy adott osztályba tartozó objektumot észleljünk, továbbá azt is meghatározzuk, hogy az hol helyezkedik el a képen. Ezt a már említett bounding box-ok segítségével tehetjük meg, amit négy paraméter segítségével lehet leírni. Ezek a bounding box szélessége (b_w), magassága (b_h), valamint a bounding box középpontjának koordinátái (b_x, b_y). Az algoritmus ezen felül kimenetként megadja, hogy mekkora konfidencia mellett van jelen egy objektum az adott bounding box-ban, illetve, hogy az mekkora valószínűséggel egy adott osztály tagja (ábra 15).



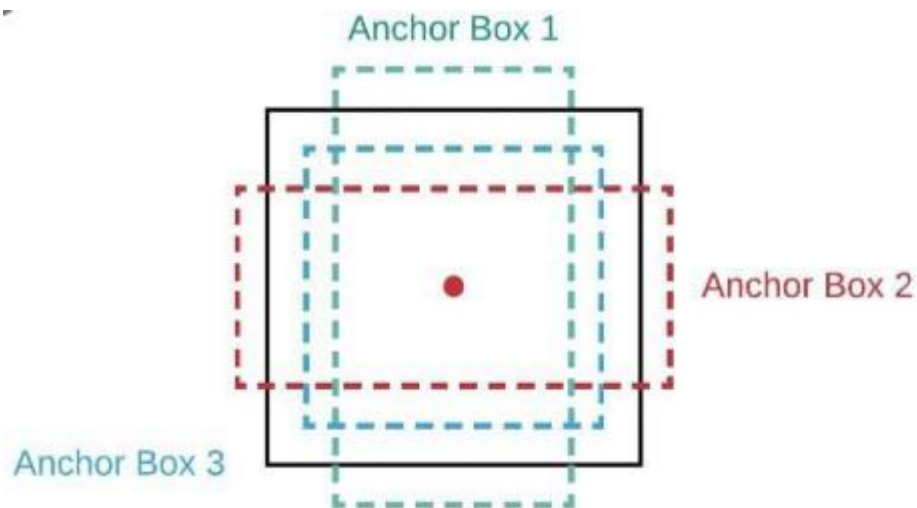
ábra 15: A YOLO algoritmus bounding box-ának paraméterei, valamint a kimeneti vektor elemei, forrás: [43]

Az algoritmus a képet azonos méretű cellákra osztja (a képtől és a leskálázástól függően), és minden egyes cellára meghatározza a fent említett kimenetet. Azt várjuk el, hogy minden egyes cella detektáljon egy ilyen objektumot abban az esetben, ha a bounding box-ának középpontja az adott cellába esik. Annak elkerülésének érdekében, hogy az adott koordináta ne lóghasson ki a cellából, a bounding box-ot meghatározó paramétereken egy transzformációt hajtanak végre (ábra 16).



ábra 16: A YOLO algoritmus bounding box leképezés transzformációja, forrás: [30]

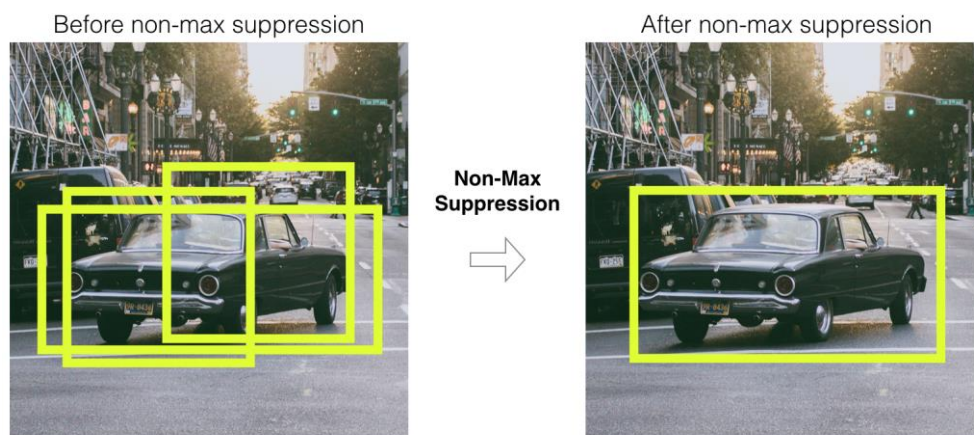
A fenti kép alapján tehát a háló a t_x , t_y , t_w és t_h paramétereket határozza meg és a fent látható leképezés segítségével kapjuk meg a végleges, fent részletezett paramétereket. Egy adott objektumhoz számos ilyen bounding boxot készít az algoritmus. De hogyan tudja az algoritmus eldönteni, hogy melyik az igazi objektum? A YOLO algoritmus egyik legfontosabb paraméterei az anchor box-ok (ábra 17). Az anchor box-ok olyan előre definiált bounding box-ok, amelyek segítségével megadhatjuk, hogy milyen jellemző alakú objektumok lehetnek a képeken. Például egy gyalogoshoz tartozó anchor box magas és keskeny, ugyanakkor egy járműhöz tartozó az lapos és széles. Ezeknek az anchor box-oknak az alakján kívül a méretét is lehet változtatni a háló által becsült t_w és t_h paraméterek alapján. Például egy jelzőtáblát övező keret legtöbbször kisebb, mint egy járművet övező. A futás során (ahogy a következő bekezdésben azt bemutatom) az algoritmus számos ilyen bounding boxot predikál, amelyek közül utólag kell eldönteni, hogy melyik övez valódi objektumot. Ezt többek között az anchor boxok segítségével dönti el.



ábra 17: különböző alakú és méretű anchor boxok, forrás: [58]

A YOLOv3 algoritmus 3 különböző méretű képen végzi el a detekciókat. Az algoritmus futása során a különböző szintű konvolúciók elvégzése során teszi ezt meg. Nézzük a következő példát, ahol a bemeneti kép 416x416 pixeles, és a három leskálázó réteg méretei: 32, 16 és 8. Ez azt jelenti, hogy a kimeneten megkapott képek cellaszámai: 13x13, 26x26 és 52x52. Az algoritmus minden egyes szinten 3 bounding boxot határoz meg minden egyes cellához, ami azt jelenti, hogy összesen $((52 \times 52) + (26 \times 26) + (13 \times 13)) \times 3 = 10647$ bounding box-ot határoz meg.

Ez a szám rendkívül nagy, természetesen nem lesz ennyi objektum a képen. Ezt a számot az algoritmus két lépésben csökkenti le. Elsőként egy bizonyos küszöbszint alatti konfidenciával rendelkező detekciókat törli, majd ezt követően Non-Maximum Suppression-t (NMS) (ábra 18) hajt végre és így kapjuk meg a végleges, az adott objektumhoz tartozó detekciót.



ábra 18: NMS a YOLO algoritmus során, forrás: [43]

3 Specifikáció és tervezés

Jelen fejezetben ismertetem, hogy mi volt pontosan a feladatom a szakdolgozatom során. Bemutatom, hogy mik határozták meg a feladatot, milyen korlátokat szabtak ezek a körülmények. Ezt követően ismertetem azt is, hogy milyen követelményeket támasztottunk előzetesen a feladat felé. Ezek a szempontok együttesen határozták meg azt, hogy végül melyik algoritmusra, neurális háló struktúrára esett a választásom. Miután kiválasztottuk az algoritmust, azt kellett meghatároznunk, hogy milyen keretrendszer segítségével állunk neki megvalósítani azt.

3.1 Feladatspecifikáció

A következő alfejezet során részletesen bemutatom azt, hogy mi volt a feladatom a szakdolgozat során. A félév kezdetén egy olyan feladatot szeretnénk volna választani, ami valamiféleképpen folytatása az önálló laboratóriumi munkámnak, vagy ha nem is, akkor minél szorosabban kapcsolódjon ahhoz. Az önálló laboratóriumi munkám során a következő feladatot határoztuk meg. Egy önvezető autót szeretnénk volna kialakítani a tanszéken található egyik távirányítás versenyautóból, amelynek az objektumfelismerését végeztem volna én.

A koronavírus sajnos nem tette lehetővé, hogy ezt a feladatot pontosan úgy tudjam folytatni, ahogyan azt elterveztük az önálló laboratórium során. A távoktatás miatt úgy módosítottuk a feladatot, hogy minél inkább szoftvercentrikusabb legyen a téma, de identitásában lehetőleg hasonlítson az eredeti tervre. Tehát az algoritmus elkészülése után is lehetőség legyen arra, hogy azt akár az önvezető autóra szerelt beágyazott GPU rendszeren futtatni lehessen. Mindemelett fontosnak tartottuk, hogy az eredeti feladat megmaradjon, tehát egy olyan objektum detektáló algoritmust fejlesszünk, ami képes járművek, személyek, közlekedési táblák és lámpák detektálására.

Eredetileg olyan adatbázist szeretnénk volna előállítani, amellyel az önvezető autó működése közben is találkozhat. Ehhez szeretnénk volna gyalogos, autó és közlekedési tábla modelleket befotózni, majd ezekből a képekből manuálisan előállítani az adathalmazt. Azonban erről a tervről hamar le kellett mondanunk, ugyanis erre távoktatásban nem volt lehetőség.

Ekkor váltottunk megközelítési módot. A manuálisan készített adatbázis helyett egy mesterségesen generált adatbázison tanítottuk be a neurális hálót. A kérdés, amire a választ kerestük pedig az volt, hogy vajon egy ilyen mesterséges képekből álló adatbázison betanított neurális háló képes-e egy valós képekből álló adatbázison is kielégítő pontossággal működni.

Miután pontosan meghatároztuk a feladatot, elkezdtük keresni, hogy hogyan tudjuk megvalósítani azt. Elsőként azt kellett kiválasztani, hogy milyen módon tudjuk legegyszerűbben és legpontosabban létrehozni a számítógép által generált adatbázist. Erre a CARLA Szimulátort [44] választottuk. Ez egy olyan open source szoftver, ami kifejezetten az autonóm járművek kutatásának segítségére lett kifejlesztve. A program működését a következő fejezetek során részletesebben is be fogom mutatni.

Emellett azt is meg kellett határoznunk, hogy milyen adathalmazon fogjuk tesztelni a betanított hálót. Ehhez a COCO adatbázist [45] használtuk fel, amelyen némi módosítást is eszközöltünk, hogy jobban illeszkedjen a szakdolgozatom témájába.

Miután meghatároztuk, hogy milyen adatbázisokat fogunk előállítani, azt kellett eldöntenünk, hogy ehhez a feladathoz milyen algoritmust, neurális háló struktúrát tudunk legeredményesebben használni. Fontos szempont volt az algoritmus kiválasztása során, hogy ez az algoritmus valós időben tudjon futni. Erre azért van szükség, hogy igény szerint ezt a megírt algoritmust alkalmazni is lehessen éles környezetben, például valamilyen jármű ECU-ján. A beágyazott rendszerekben futó alkalmazások irányába támasztott egyik legfontosabb elvárás az, hogy az adott alkalmazás elérjen a beágyazott GPU rendszerben, hiszen, ha ezt a feltételt nem tudjuk teljesíteni, akkor mit sem ér az algoritmusunk. A valós időben történő futtathatóság mellett ez volt tehát a másik fontos szempontunk.

Ahogy már említettem, a munkám során két különböző adatbázist használok. A neurális hálót a CARLA szimulációból kinyert képeken tanítom be, majd pedig a betanított neurális hálót a COCO adatbázis módosított képein fogom tesztelni. Sajnos a két adatbázisban nem lehetett minden egyes objektumot pontosan megfeleltetni egymással, ugyanis a CARLA szimulátor csak járműveket és embereket különböztet meg, míg a COCO adatbázis a járműveken belül megkülönböztet például autót, buszt és teherautót is. Ez a feladat során nem jelentett problémát, mert a COCO adatbázisban az egyes képeken lévő annotációkat úgy módosítottam, hogy minden autó, busz, teherautó

stb. jármű címkét kapjon. Emiatt azonban sajnos nem lehetett úgy diverzifikálni az eredményeket, mint ha ezeket az objektumokat a CARLA is megkülönböztette volna.

3.2 Háló struktúra választása

Az előző alfejezetben bemutatott szempontok alapján kellett meghatároznunk azt a rendszert, amit használni szeretnék a munkám során. Az irodalomkutatásról szóló fejezetekben már bemutattam azokat a lehetséges neurális háló struktúrákat, amelyek közül választottunk.

Az egyik, már korábban bemutatott szempont az volt, hogy a háló a valós idejűséget biztosítani tudja. Ezért a kevésbé gyors struktúrák, mint a különböző régió alapú konvolúciós hálók, mint az RCNN, Fast RCNN és a Faster RCNN hátrányból indultak. A gyorsaságát tekintve ideális jelölt volt még az SSD struktúra is.

Továbbá fontos volt a választás során, hogy egyszerűen implementálható és használható legyen a struktúra. A fenti megfontolások alapján a YOLO algoritmus mellett döntöttünk. A döntést végül az is elősegítette, hogy szemben az SSD algoritmussal a YOLO sokkal jobban elterjedt, mint az SSD és számos implementációja is megtalálható az interneten, ami a felhasználhatóságát javította.

4 Fejlesztés

A következő fejezetben szeretném bemutatni a szakdolgozat során elvégzett munkámat, kezdve azzal, hogy milyen programozási nyelvet, keretrendszert használtam. Ezután azt fogom bemutatni, hogy hogyan készítettem el a tanító adatbázist, milyen algoritmusokat használtam ehhez. Ezután bemutatom azokat a modulokat, amelyek a kinyert nyers képekből a használható képadatbázist megalkotják.

Az adatbázis ismertetése után kitérek arra, hogy pontosan hogyan épül fel a neurális háló struktúra, végül pedig azt is bemutatom, hogy milyen módszerekkel, kísérletekkel értük el a legjobb eredményt.

4.1 Fejlesztőkörnyezet

Mielőtt a feladatomnak nekiláttam volna, el kellett döntenem, hogy milyen programozási nyelvet, keretrendszert és fejlesztőkörnyezetet érdemes használnom a fejlesztéshez, mi az, ami a legkifizetődőbb hosszú távon.

4.1.1 Programozási nyelv

A gépi tanulásról és mélytanulásról, mint tudományterületekről egyaránt elmondható, hogy magas absztrakciós szintű feladatokkal foglalkoznak. Ezért tehát egy olyan programozási nyelvet kellett választanunk, amelyben minél egyszerűbb a komplex programozási feladatok megoldása. Az, hogy a nyelv mennyire legyen hatékony és mennyire optimalizáljon jól mind futási idő, mind memóriakapacitás szempontjából csak másodlagos szempont volt. Ezek miatt az olyan alacsony szintű programozási nyelvek, mint a C szóba sem jöttek.

Magasabb szintű programozási nyelveken belül számos lehetőség közül tudtunk választani, mint a Java, a C++ vagy a Python. Ezekről egyaránt elmondható, hogy támogatják az objektum orientált szemléletet és könnyebben kezelhetőek velük komplexebb feladatok. A Java-t hamar elvetettük mint lehetséges opciót, mert egyetemi tanulmányaim során nem tanultam ezen a nyelven programozni.

A két megmaradt programozási nyelv között már kissé nehezebben sikerült választani. A C++ nyelv mellett szólt az, hogy ezt a programozási nyelvet már jól ismertem, hiszen az alapozó tárgyaink keretében már megtanultam ebben a nyelvben

programozni és használni azt. További pozitívum, hogy a nyelvben elérhető egy már kész implementációja a YOLO algoritmusnak (az implementáció C nyelvben lett megírva, de a C++ képes a C-ben megírt kódok futtatására) [46]. Hátránya, hogy a nyelv erősen típusos, erre jobban oda kell figyelni a fejlesztés közben. További hátrány, hogy minden memóriakezelésért a fejlesztő felel, nem menedzselt környezetben fut a program, mint a Python esetében, nincs garbage-collector sem.

A Python nyelv szintén egy magas szintű programozási nyelv, amely nem típusos, tehát nem kell figyelni arra, hogy milyen típust adunk át egy változónak, ezt akár futás közben is lehetőség van megváltoztatni. Továbbá a memóriakezelésre sem kell figyelni, a program ugyanis egy úgynevezett menedzselt környezetben fut, egy garbage-collector felelős a lefoglalt memória felszabadításáért. Hátránya viszont, a menedzselt környezetnek és a nem típusos tulajdonságnak, hogy fordítási időben nem kapunk visszajelzést a programunk milyenségét illetően, ellenben a C++-al, ami tájékoztat, ezekről az információkról. Éppen ezért Pythonban nehezebben is lehet megtalálni az ilyen jellegű hibák forrását.

A Python mellett szólt az, hogy számos bővítmény és keretrendszer elérhető, amelyek nagy mértékben könnyítik meg a gépi tanulással és képfeldolgozással kapcsolatos algoritmusok fejlesztését. További érv volt a nyelv mellett, hogy ebben szintén implementálva van a YOLO algoritmus [28][29][57]. Az viszont a nyelv ellen szólt, hogy a témám megkezdése előtt semmilyen formában nem találkoztam még ezzel a programozási nyelvvel, tehát meg kellett tanulnom ebben programozni.

A fent bemutatott szempontok alapján végül a Python nyelv mellett döntöttünk, mivel a nyelv megtanulása nem jelent különösen nagyobb problémát, a hibák nehezebb megtalálhatóságát pedig könnyen orvosolni lehet egy modern fejlesztőkörnyezettel.

4.1.2 Fejlesztőkörnyezet/IDE

A nyelv kiválasztását követően le kellett töltenünk magát a Python nyelvet futtató interpretert. Ez Linux környezeten alapértelmezetten elérhető, Windows rendszerre viszont le kell tölteni. Eerre számos lehetőség van, az egyik legelterjedtebben használt telepítő és keretrendszer kezelő az Anaconda [49]. Az Anaconda rendelkezik grafikus kezelőfelülettel, amelyben könnyedén tudjuk kezelni a szükséges modulok verzióját, letöltését és törlését is. Arra is lehetőségünk van, hogy projekttől függően több környezetet is létrehozzunk, amiben az egyes csomagok más verziójuk lehetnek.

A nyelv kiválasztása után azt kellett eldöntenünk, hogy milyen fejlesztőkörnyezetben szeretnénk dolgozni. Itt is több lehetőség állt rendelkezésemre. A két lehetséges választás a Microsoft Visual Studio Code-ja [47] volt, illetve a JetBrains PyCharm környezete [48]. Mindkét fejlesztőkörnyezet rendelkezik ingyenes, community kiadvánnyal, valamint mindkét környezet script alapú, tehát nem projekteket hozunk benne létre, hanem a megírt scriptet önállóan tudjuk futtatni az éppen jelen lévő fordítóval, környezettel. Ez a környezet Windows rendszeren az Anaconda környezet segítségével változtatható, míg Linux rendszeren a beépített terminálból könnyen futtatható.

Mindkét környezet fejlett debuggolást támogató funkciókkal van felszerelve, lehetőség van feltételes breakpointok lehelyezésére, a változók átírására, megtekintésére, ha megállítottuk a program futását. Ez nagyban megkönnyíti a fejlesztést. Mindkét programban támogatja a különböző kód formátáló kiegészítőket, a PyCharm az autopep-et használja, VS Code-ban pedig kiterjesztésként letölthető a kívánt formátáló. A PyCharm környezet alapértelmezetten támogatja a Python nyelvet, sőt ez a környezet kimondottan ehhez a nyelvhez lett fejlesztve, míg a Visual Studio Code-hoz le kell tölteni a Python támogatást egy bővítmény formájában.

A fenti szempontok alapján a választás a PyCharm Editorra esett, mivel ez kifejezetten a Python nyelvben történő programozásra lett kifejlesztve, míg a VS Code egy általánosabb fejlesztőkörnyezet. Továbbá a tanszéki gépeken is PyCharm fut.

4.1.3 Keretrendszer

A fejlesztőkörnyezet után azt is ki kellett választani, hogy milyen keretrendszert fogunk használni a feladatok megvalósításához. A gépi tanuláshoz Python nyelven számos keretrendszer íródott, amelyekben egyaránt megvalósítható lett volna a feladat. Mind a Tensorflow [50], mind a Keras [51], mind a PyTorch [52] keretrendszerek rendelkeznek azokkal az esszenciális modulokkal és függvényekkel, amelyek a mélytanuló algoritmusok megalkotásához szükségesek.

A választásunk a PyTorch keretrendszerre esett, a választást több érv is támogatja. Talán a legfontosabb érv az, hogy az a YOLO implementáció, amit használni fogok [53] is a PyTorch keretrendszert használja, ez pedig nagy mértékben megkönnyíti a módosítások eszközését a programban. Továbbá a keretrendszer honlapján számos oktatóanyag található, ami megkönnyíti az alkalmazást, valamint az egyes modulok,

függvények részletesen vannak dokumentálva, működésük pedig példákkal van szemléltetve.

4.2 Tanító adatbázis létrehozása

Elsőként az adatbázis megalkotását mutatom be, hiszen ez a szakdolgozatom egyik legfontosabb eleme, így a munkámat ennek az adatbázisnak a megalkotásával kezdtem. Ahogy már azt bemutattam az előző fejezetekben a szakdolgozatomban két adatbázist használok. A tanító adatbázisban mesterségesen generált képeket használok, míg a tesztadatbázist valós fényképek alkotják. Ebben az alfejezetben a tanító adatbázis létrehozását fogom bemutatni, a tesztadatbázist megalkotását pedig a következő fejezetben mutatom be.

4.2.1 CARLA Szimulátor

A szintetikus képeket a CARLA szimulátort [44] használva hoztam létre. Ez egy nyílt forráskódú program, amely kifejezetten az autonóm rendszerek fejlesztésére, tanítására és validálására hoztak létre. A szimulátor számos nyílt világot szolgáltat, amelyben szabadon mozoghat a felhasználó egy jármű segítségével vagy akár anélkül. Ezek a világok nagy mértékben módosíthatóak, hogy az minél inkább megfeleljen a fejlesztő igényeinek. Lehetőség van megadni a világban szereplő járművek, gyalogosok számát, az időjárást tetszés szerint változtatni, a saját jármű paramétereit igény szerint állítani.

A létrehozott járműveket egy úgynevezett forgalom menedzserben lehet irányítani. A forgalom menedzserben számos további paramétert lehet állítani, mint például a jelzőlámpák állítását vagy a közvilágítást. Számos környezetérzékelő szenzort is el lehet helyezni egy adott járművön, mint például színes vagy mélységi kamera, LIDAR szenzor. Ezek a szenzorok a hatékony autonóm rendszerek fejlesztése szempontjából elengedtetlenek, magam is számos ilyen szenzort alkalmaztam a szakdolgozatom során.

Az ilyen szenzorok révén számos eseményt, szituációt lehet megfigyelni, ezért az, hogy széles spektrumból választhat a fejlesztő nagyban megkönnyíti a munkát. Többek között lehetőség van radar, lidar szenzorok alkalmazására is, de olyan speciálisabb szenzorok is rendelkezésre állnak, mint a sávelhagyás érzékelő vagy az ütközés érzékelő. Ugyanakkor azok az alapvető szenzorok is megtalálhatóak, amik az

egyes pedálok vagy a kormány állását érzékelik. A szimulátorhoz számos kiegészítő script is tartozik, amelyekkel az egyes funkciókat lehet futtatni, továbbá demó script is található benne, amelyet igény szerint át lehet formálni.

4.2.2 Képgenerálás

A CARLA Szimulátor segítségével olyan képeket kellett kinyerjek, amik egy jármű szemszögéből mutatják a világot, oly módon, hogy a generált képek minél változatosabbak, sokszínűbbek legyenek. Fontos volt továbbá, hogy minden olyan információt kinyerjek a kép készítésének pillanatában, amely ahhoz szükséges, hogy később az adott képhez el tudjam készíteni a hozzá tartozó annotációs listát.

Ahhoz, hogy a képgenerálást el lehessen indítani mindenekelőtt magát a szimulátort kell elindítani. Ekkor a szimulátor leteszi az ágenszt a kezdőpozícióba a harmas számú, alapértelmezett városba (ábra 19). Ilyenkor szabadon mozoghatunk a városban egy felülnézeti módban.

Város	Összefoglalás
Town01	Egyszerű városi kialakítás, „T kereszteződésekkel”
Town02	A Town01-hez hasonló, csak kisebb.
Town03	A legösszetettebb város, 5 sávós kereszteződéssel, körforgalommal, alagúttal és sok egyébvel.
Town04	Egy végtelenül ismétlődő körpálya sztrádával és egy kisvárossal.
Town05	Klasszikus kereszteződésekkel teli város, többsávós utakkal. Ideális sávtartó és sávváltó programokhoz.
Town06	Hosszú sztrádák, sok fel- és lehajtóval.
Town07	Mezőgazdasági környezet szűk utakkal kevés jelzőlámpával.
Town10HD	Nagyvárosi környezet, sugárúttal, felhőkarcolókkal, promenáddal, részletesebb textúrával.

táblázat 1: a CARLA Szimulátorban található városok leírása, forrás: [54]

Egy tanító adatbázisnál rendkívül fontos, hogy minél változatosabb környezetben, körülményekben készítsünk képeket, hogy minél inkább csökkentsek a mélytanuló algoritmusok tanítása során jelentkező túlillesztés lehetőségét. Ez a

számunkra előnytelen jelenség azt jelenti, hogy a neurális háló nem általánosan tanulja meg felismerni az objektumokat, hanem megjegyzi az összes képet, „bemagolja” azokat. Ennek a hátulütője az, hogy a validációs és tesztadatbázisokon, valamint a tényleges működés közben rossz eredményt fogunk elérni.

A változatos környezetet és körülményeket a következőképpen értem el. A CARLA Szimulátor alapváltozata összesen 8 darab várost tartalmaz, amelyek mindegyike más és más tulajdonságokkal rendelkezik (táblázat 1).



ábra 19: a Town03 térképe, forrás: [54]

A szimulátor további beépített lehetőséget biztosít a környezet dinamikus változtatására, mégpedig az időjárás változtatásának segítségével. Ezt szintén egy beépített script segítségével van lehetőségünk módosítani. A dinamikus időjárás bekapcsolása után lesz olyan időszak, amikor süt a nap, esik, vagy köd van. A napszakok is változni fognak, lesz lehetőség éjszaka, alkonyatkor és napkeltekor is képeket készíteni (ábra 20).



ábra 20: esős körülmények alkonyatkor, Town01-ben

Azonban a körülmények változatossá tétele mit sem ér, ha nem helyezünk el olyan objektumokat, amelyeket a neurális hálónkkal fel szeretnénk ismertetni. A szimulátor alapértelmezetten üres városba helyezi a felhasználót, ahol a számomra fontos objektumok közül csak jelzőlámpák és közlekedési táblák vannak. Járműveket és gyalogosokat egy script segítségével tudunk lehelyezni, amiket aztán a forgalom menedzser fog kezelni. A script futtatásakor lehetőség van megadni, hogy járműből és gyalogosból pontosan hány darabot tegyen le a térképre. Ez a szám csak akkor nem fog teljesülni, ha az adott kezdeti pozícióban már van egy jármű.

A következő feladat ahhoz, hogy képeket tudjunk generálni a szimulátorral az volt, hogy le kellett tenni a térképen egy olyan járművet, amivel a képkészítést lehet majd elvégezni. A dinamikus időjárás miatt sokszor előfordulhat az, hogy az autó sötétben közlekedik, ezért be kellett állítani azt, hogy a napsugarak bizonyos beesési szöge alatt a jármű fényszórói bekapcsoljanak, ha pedig világos van, akkor kikapcsoljanak.

Az olyan feladatok elvégzésére, ahol egy szereplőnek valamilyen szempontból kiemelt szerepe van és a szimuláció többi szereplője mellékes, a szimulátor biztosít egy úgynevezett szinkronizált módot. Ennek során a kiemelt szereplő és az őt futtató script szerint szinkronizál. Ez a szinkronizálás olyan esetekben esszenciális, amikor a szimuláció szempontjából sokáig tartó folyamatok zajlanak le, mint például képek kimentése. Ha nem szinkronizálnánk és több képet szeretnénk lementeni egy adott pillanatban, akkor előfordulhatnak a járművek elmozdulásából származó

inkonzisztenciák, amik az annotációk és ezáltal az adatbázis minőségének romlásához vezetnének. A szinkronizált módban addig nem folytatódik a szimuláció, amíg a szinkron módot futtató script minden feladatot meg nem csinált, ez esetben a képek elmentése.

Most, hogy már minden feltétel adott ahhoz, hogy megfelelően történjen meg a kép kimentése, rátérek arra, hogy ezt hogyan valósítottam meg. Szinkronizált módban tehát lehelyezek a térképre egy ego járművet. Ezen jármű kezeléséért a forgalom menedzser lesz a felelős csakúgy, mint a többi jármű irányításáért. A megfelelő annotációk elkészítéséhez kétféle kamerára mindenképp szükség van, egy további kamerát az illusztrációkhoz és ellenőrzéshez helyeztem el a járművön, egy negyediket pedig arra az esetre, ha javítani szeretnénk az adatbázis és az annotációk minőségén. A kamerák a következők.

Az egyik legfontosabb és mindenképp szükséges szenzor a klasszikus RGB kamera (ábra 21). Az RGB kamerával felvett képeken fogjuk betanítani a hálót, ezen kell majd felismerni az objektumokat.



ábra 21: CARLA-ban készített RGB kép, járműre helyezve

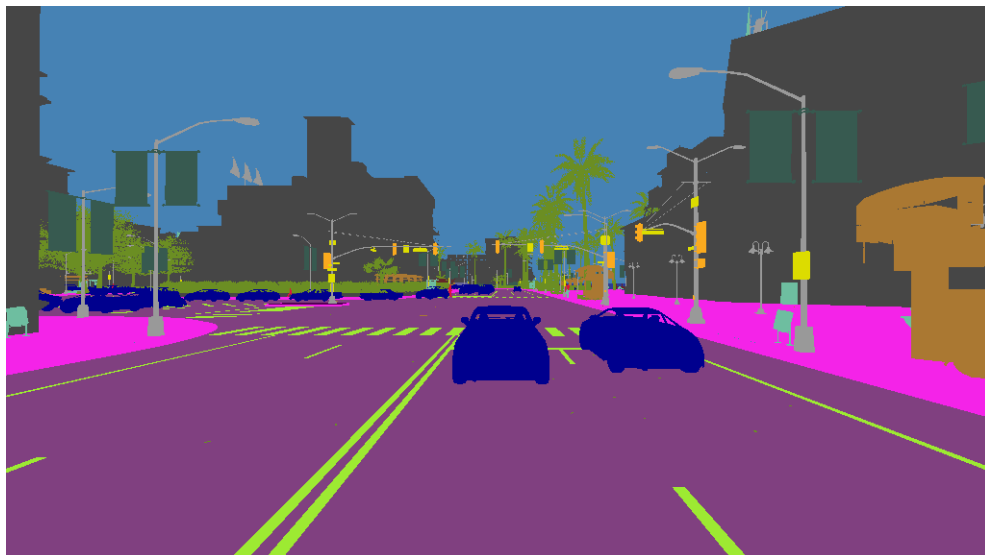
A másik elengedhetetlen kamera, amit mindenképpen el kell helyezni az autón a szemantikus képet készítő kamera. Ez a kamera nem ad látványos képet, ám az annál hasznosabb; Az így kapott színes kép a piros csatornát kivéve mindenhol nullát tartalmaz, a piros csatornában pedig az egyes osztályok kódjai vannak tárolva egy-egy numerikus értékkel (táblázat 2). Ezáltal lehetőségünk van minden egyes objektum

pozícióját kinyerni a képből. Én a későbbiekben ezt az információt a befoglaló téglalapok (bounding box-ok) elkészítéséhez fogom használni.

Érték	Címke	Konvertált szín
4	Gyalogos	(220, 20, 60)
10	Jármű	(0, 0, 142)
12	Közlekedési tábla	(220, 220, 0)
18	Jelzőlámpa	(250, 170, 30)

táblázat 2: A feladathoz használt pixelértékek a piros csatornában, a hozzájuk tartozó címével

Ahhoz, hogy az emberi szem számára jobban értelmezhető legyen a szemantikus szegmentációból származó információ elhelyeztem egy olyan kamerát is, amely ezt az információt alakítja át a fenti transzformáció szerint (táblázat 2). Az alábbi képen (ábra 22) már jól láthatóak az egyes objektumok, amiket a szemantikus kamerával nyertünk ki.



ábra 22: konvertált szemantikus szegmentáció kép

Végül elhelyeztem egy mélységérzékelő kamerát is arra az esetre, ha a mélységinformációk alapján tovább szeretném vagy tovább kell finomítani az annotációk pontosságát (ábra 23). Ez a kép szintén nem olyan látványos, mert a közeli tárgyak pixelértékei csak kis mértékben térnek el.



ábra 23: mélységinformációt tartalmazó kép

Miután felkerültek a szenzorok a járműre, minden készen állt az adatbázist elkészítéséhez. Az adatbázist a következőképpen generáltam: Minden egyes városban elhelyeztem megfelelő számú járművet (100-200) és gyalogost (25-75), majd beállítottam, hogy dinamikusan változzon az időjárás. Ezt követően elindítottam a szinkronizált módú scriptet, ami elhelyezi az ego járművet a térképen, átadja az irányítását a forgalom menedzsernek és elkészíti a képeket. A képek készítésénél arra kellett figyelni, hogy ugyanabból a szögből, helyzetből ne készüljön több kép. Elhelyeztem egy olyan feltételvizsgálatot, ami csak akkor engedélyezi új kép készítését, ha az előző kép készítésének pozíciójához képest már megtettünk egy bizonyos távolságot (15-20m) légvonalban. Minden városban 1024 képet készítettem, mindegyik kamerával, így összesen 8192 képből áll a tanító adatbázisom.

4.2.3 COCO annotáció generálás

A képek generálása utáni következő feladat a hozzájuk tartozó annotációk elkészítése volt. Az annotációk tartalmazzák az adott képen szereplő összes objektumot, ezek tehát szintén esszenciális részét képezik egy adatbázisnak

Az annotációkat A COCO adatbázis szerkezete szerint építettem fel, azért, mert a tesztadatbázisom a COCO adatbázis módosított változata, s így ugyanazzal az algoritmussal tudom majd átalakítani a YOLO által megkívánt formátumra. A COCO adatbázis annotációs listája json formátumban van tárolva, s a következőképpen épül fel. Öt főbb része az info, a licenses, az images, a categories, valamint az annotations.

Elsőként az egyszerűbbeket ismertetném, majd pedig a fontosabb részeket. Az információs részben találhatóak a metaadatok, mint hogy mikor és ki készítette, hol elérhető, illetve hányadik verziójú. A licenses részben különböző szerzői jogi információk találhatóak, ez esetben nem került kitöltésre, mert minden kép a CARLA szimulátorból származik. A categories részben találhatóak a képeken szereplő osztályok leírása, ezek esetében az alábbiak:

```
"categories": [  
  {"supercategory": "person", "id": 1, "name": "person"},  
  {"supercategory": "vehicle", "id": 2, "name": "vehicle"},  
  {"supercategory": "outdoor", "id": 3, "name": "traffic sign"},  
  {"supercategory": "outdoor", "id": 4, "name": "traffic light"}  
],
```

Az images szekcióban találhatóak az egyes képek leírásai, amelyekről a következő adatokat tárolom. Minden képhez eltárolom a fájlnevét, az adott mappában, a dimenzióját, valamint egy egyedi azonosítót is, ami minden kép esetén más.

```
new_im_dict = {  
  "license": "",  
  "file_name": str(rgb_files[i]),  
  "coco_url": "",  
  "height": 1024,  
  "width": 576,  
  "date_captured": "",  
  "flickr_url": "",  
  "id": IM_ID  
}
```

A legfontosabb rész az annotációk tárolása, amelyekhez el kell tárolni a bounding box területét, a hozzátartozó kép egyedi azonosítóját, az adott bounding box szélességét, magasságát, illetve a bal felső sarok pozícióját (a kép bal felső sarkához képest), továbbá az objektum azonosítóját és az adott címke egyedi azonosítóját. A felépítéshez még a szegmentáció is hozzátartozik, de ez nem került kitöltésre esetemben, mivel ezt nem fogom használni.

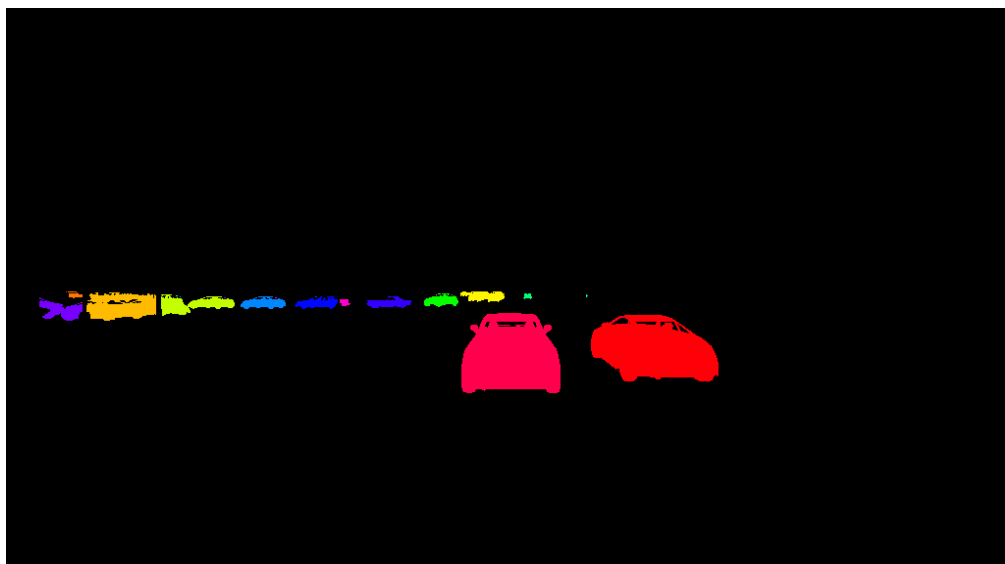
```
new_ann_dict = {'segmentation': [[]],  
  'area': area,  
  'iscrowd': 0,  
  'image_id': IM_ID,  
  'bbox': bounding_boxes[j],  
  'category_id': cat_id,  
  'id': cur_ann  
}
```

Most, hogy ismertettem az annotációs lista felépítését, bemutatom, hogy a képekből, hogyan nyerek ki a bounding box-okat. Miután ellenőriztem, hogy minden

szükséges kép megvan, beolvasom az első képet és kitöltöm az kép adatait. Itt az egyetlen említésre érdemes rész a képhez tartozó egyedi azonosító. A képek kinyerésekor minden kép filenevében eltároltam, hogy melyik városban készült és hogy hányadik kép volt az. Ezek alapján alkotom meg az egyedi azonosítót, ahol a TOWN_NO 1 és 10 közötti érték lehet:

$$IM_ID = TOWN_NO * 10000 + IM_NO$$

Ezt követően az annotációk létrehozása következik. Az adott színes kép mellé beolvasom a hozzá tartozó szegmentációs képet, amely az objektuminformációkat tartalmazza. A szegmentált képből ezután az osztályok szerint kinyerem a bounding boxokat. A szegmentált képnek csak a piros csatornája tartalmaz információt, így csak azt az egy csatornát tartom meg. Ezután ott, ahol az adott pixelérték megegyezik az aktuális osztály azonosítójával, azt egyessé konvertálom, a többit pedig nullává, ezt követően az egyes értékeket pedig 255-ös, maximális értékke alakítom. Ezután pedig az OpenCV [59] könyvtár `connectedcomponents` függvényével megszámlolom az egyes komponenseket. Ez a függvény a komponensek számát adja vissza, valamint egy olyan képet, ahol minden komponenst egy egyedi szám reprezentál 1-től kezdődően (ábra 24). A komponenseken egyesével végig haladva a bounding box-ok kiszámolhatók, amiket egy listába gyűjtök. Ezután már csak az van hátra, hogy az információk alapján kitöltssem az annotációs könyvtárt.



ábra 24: komponensek a jármű osztály esetén

4.2.4 COCO-YOLO konverzió

Az a nyílt forráskódú implementációja a YOLOv3-nak, amit használok más annotációs formátumot használ, ezért a COCO-s formátumot át kell alakítanom. Az implementációhoz a json formátumból minden egyes képhez egy hozzá tartozó annotációs szöveges fájlt kell létrehozni. A szövegfájl minden sora egy-egy objektumhoz tartozó információt tartalmaz. A sorban összesen 5 darab szám van, az első szám az adott annotációhoz tartozó osztály sorszáma, ez azonban nullával kezdődik nem eggyel. A következő két szám a kép középpontjának szélessége és magassága 0 és 1 közé normálva. Az utolsó számpár pedig a bounding box szélessége és magassága szintén 0 és 1 közé normálva. A normalizálás a bounding box szélességének és magasságának esetében úgy történik, hogy a két értéket rendre el kell osztani a kép szélességével, illetve magasságával, így kapjuk meg a 0 és 1 közé eső értéket. A bounding box középpontját hasonlóképpen az adott dimenziókkal való osztással lehet 0 és 1 közé normálni.

További előnye a COCO formátumnak és a YOLOv3 implementációnak, amit használok, hogy mindkettő rendkívül népszerű, ezért számos kiegészítő alkalmazás és forráskód elérhető hozzá, többek között olyan modul is létezik, ami a COCO formátumból a YOLO v3 programhoz szükséges formátummá konvertálja, az annotációs fájlt [60]. Ehhez nem kell mást megadni, mint az annotációs fájlt, az osztályok nevét, valamint az annotációkhoz tartozó képeket.

4.2.5 Tapasztalatok az adatbázis megalkotása során

Az adatbázis megalkotása és természetesen az egész szakdolgozat elkészítése során számos dolgot tanultam, kódolástechnikától kezdve, tesztelésen és debugoláson át egészen a képfeldolgozás megfelelő alkalmazásáig bezárólag. Ezek közül egyet szeretnék most bemutatni, ami különösen tanulságos volt számomra és megmutatta, hogy milyen fontos odafigyelni a részletekre egy adott téma esetén, majd azt megfelelően alkalmazni.

A tanító adatbázis megalkotása során az objektumok bounding box-át a szegmentációs kép alapján alkottam meg. Ezeken a képeken az egyes pontos értéke hordozza az információt, azok bármilyen módosulása az információ jelentős veszteségével jár, ezért rendkívül fontos arra odafigyelni, hogy ez ne sérüljön. A CARLA Szimulátorban többféle formátumban van lehetőségünk elmenteni a képeket

többek között .png és .jpg formátumban. Én először a szegmentációs képeket is .jpg formátumban mentettem el, nem figyelve arra, hogy a .jpg formátum tömöríti az adatokat, ami információvesztéssel is jár. Emiatt amikor az annotációk generálását végeztem meglepődve tapasztaltam, hogy a várt objektumszámhoz képest sokkal több bounding box-ot generál az ellenőrzöten jól működő algoritmus. Csak hosszas keresés és konzulensi segítséggel jöttem rá, hogy a hiba okát a .jpg tömörítésből adódó információvesztés okozza. Emiatt a teljes képadatbázist újra kellett generálnom, hogy megfelelő legyen az adatbázisom.



ábra 25: a .jpg formátum tömörítéséből adódó hibás detekciók jármű osztály esetén

Ahogy az a fenti képen (ábra 25) is látható különösen objektumhatároknál (épület és ég határa, középen fent) keletkeztek hamis pozitív osztály érzékelések .jpg formátum esetén. Ezekben az esetekben a tömörítés miatt a jpg kép a két szomszédos pixelt átlagolta, így olyan érték is létrejöhetett, ami éppen a detektált osztály pixelértékének felel meg.

4.3 Tesztadatbázis létrehozása

Ahogy az a kivonatban és a bevezetőben is szerepel, a feladatom során arra a kérdésre keresem a választ, hogy sikeresen alkalmazható-e valós képeken egy olyan algoritmus, amit kizárólag szintetikus generált képeken tanítottunk be. Ehhez egy olyan tesztadatbázist kellett alkotnom, amin ezt le lehet ellenőrizni.

Tesztadatbázisnak a népszerű COCO adatbázist választottam és ezen hajtottam végre módosításokat. A COCO adatbázis egy rendkívül nagy adatbázis, több tízezer képpel és összesen 90 osztállyal. Egy olyan alkalmazáshoz, ahol a tanító adatbázis is

csak 8000 képből és négy osztályból áll ez az adatmennyiség jelentősen nagyobb a szükségesnél. Ezért olyan módon kellett szűrni a képeket, hogy csak azok maradjanak meg, amelyek a feladat szempontjából relevánsak. A feladat során olyan képekkel tanítom a neurális hálót, amely kizárólag közlekedési szituációkat ábrázol. Ezért a COCO adatbázisból is azokat a képeket kellett kinyernem, amelyek ilyen eseteket ábrázolnak. Ezt úgy oldottam meg, hogy azokat a képeket választottam ki az adatbázisból, amelyeken egy jelzőlámpa és egy másik objektum is látható a következő osztályokból: személy, bicikli, autó, motorbicikli, busz, teherautó. Ezzel a szűréssel sikerült elérnem, hogy a képek relevánsak legyenek a feladat szempontjából.

A másik módosítás, amit végrehajtottam, az az volt, hogy különböző közlekedési lámpákat helyeztem el a tesztadatbázison. Ennek az oka az volt, hogy a munkám során készítettem egy táblafelismerő neurális hálót, amellyel a szakdolgozatom során a felismert táblákat tudom majd osztályozni. Mielőtt a táblákat elhelyeztem volna a tesztképeken, azokon az alábbi augmentációkat hajtottam végre, hogy javítsam a képek változatosságát. A képeken véletlenszerűen hajtottam végre fényesség, kontraszt, szaturáció, élesség, skála, rotációs és perspektív transzformációkat és csak ezután helyeztem el az eredeti COCO-s képen, valamint a bounding boxot is a transzformált képhez számítottam ki. Egy képre 70% eséllyel kerül 1 tábla 30% eséllyel pedig 2 (ábra 26).



ábra 26: módosított tesztkép két beillesztett táblával

Ahhoz, hogy a tanítást el lehessen kezdeni a YOLO implementációban néhány előzetes lépést el kell végezni, ami az implementáció Github oldalán ismertetve van.

Elsőként le kell tölteni a szükséges Python modulokat és telepíteni kell azokat. Opcionálisan lehetőség van letölteni olyan súlyokat, amelyekkel már sikeresen működött a neurális háló struktúra egy bizonyos feladat megoldása során. Ahhoz, hogy saját adatbázison is tanítani lehessen létre kell hozni egy saját modellt, ahol paraméterként a saját osztályaink számát kell megadni. Emellett egy szöveg fájlban soronként meg kell adni az osztályaink nevét. A képeket és az annotációkat a megfelelő mappába kell mozgatni. Utolsó lépésként pedig fel kell osztani az adatbázist tanító és validáló részekre, ezt én 80%-20% arányban osztottam fel. Fontos megjegyezni, hogy az implementáció nem képes olyan képeket kezelni, amin nincs egy darab objektum sem, ezért ezeket előzetesen el kell távolítani az adatbázisból.

4.4 Kiegészítő osztályozás

A neurális háló tanításának tervezése során úgy döntöttünk, hogy a táblák osztályozásához egy külön osztályozó neurális hálót fogunk felhasználni. Ennek legfőbb előnye az objektumdetektáló YOLO hálózat tehermentesítése, hogy annak ne kelljen több, mint 50 további osztály között különbséget tennie. Többek között ez is volt az oka annak, hogy a tesztadatbázisra ennyi táblát helyeztünk el. Jelen alfejezetben ennek az osztályozó neurális hálónak a struktúráját szeretném röviden bemutatni.

Ez a neurális háló egy teljesen konvolúciós neurális háló, ami a tesztadatbázishoz felhasznált közlekedési táblákat képes osztályozni. Ebben az adatbázisban összesen 55 osztály van. A létrehozott neurális háló hiperparaméterei állíthatóak, ezeken Bayes-i optimalizálást [61] is végrehajtottam, hogy a pontosságot maximalizálni tudjam. A háló struktúrájában lehetőség van hangolni az osztályok számát, a háló szintjeinek számát, az egyes szinteken található rétegek számát. További állítható paramétere a hálónak, hogy mekkora legyen az első konvolúciós réteg kimeneti csatornaszáma, hogy reziduálisak legyenek-e az egyes rétegek, a kernelméret, a linearitás típusa, hogy legyen-e batch normalizálás, valamint, hogy az osztályozó réteget megelőző dropOut réteg bemeneti valószínűsége mekkora legyen.

Ezeket a paramétereket a Bayes-i optimalizálás során hangoltam be, hogy a táblafelismerés pontossága ideális legyen.

5 A tanítás módszertana és eredményei

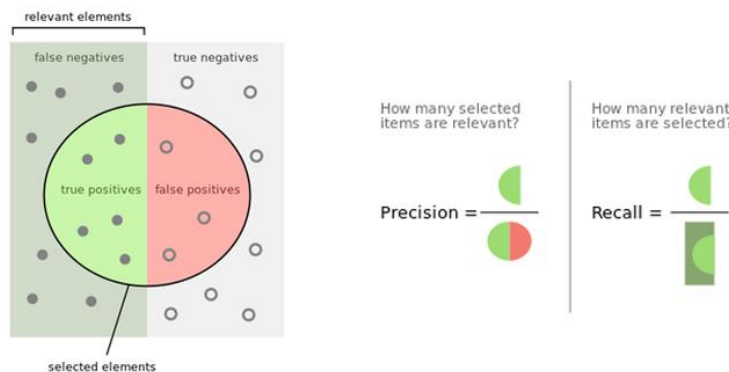
Ebben a fejezetben azt mutatom be, hogy az elkészült adatbázison hogyan tanítottuk be a neurális hálót, az eredmények kiértékeléséhez milyen metrikákat és mutatókat használtunk, és ezek alapján milyen paramétereket és hogyan módosítottunk.

5.1 Az értékeléshez használt mutatók

Az eredmények kiértékeléséhez két főbb mutatót alkalmaztunk, amit mind a tanító, mind a validációs adathalmazon kiértékelünk.

Az első mutató a hibafüggvény eredménye. A hibafüggvény a YOLO adatbázis esetén, az egyes YOLO rétegek hibáinak összege. Az eredeti struktúrában összesen három darab ilyen réteg van, ezek hibafüggvényét pedig több komponens összege alkotja, az osztályozás pontatlanságából fakadó hiba, a lokalizálásból adódó hiba, illetve a konfidenciából fakadó hiba. Ahhoz, hogy a háló minél pontosabb legyen, ezt az értéket minél jobban csökkenteni kell. Fontos, hogy mind a tanító, mind a validációs adatbázison egyaránt alacsony legyen ez az érték, hogy elkerüljük a túlillesztés (amikor a tanító adatbázison alacsony a hiba, de a validációs adatbázison nagy), illetve alul illesztés (a hiba mindkét adatbázison nagy) jelenségét.

A másik használt metrika az egyes osztályokra adott átlagos pontosságok számtani közepe (mAP - mean Average Precision) volt. E mutató megértéséhez két másik mutatót is be kell vezetnem, ezek a precision és a recall. A precision azt mutatja meg, hogy az összes detektált objektum közül, mennyi a releváns, a „true positive”. A recall azt mutatja meg, hogy az összes releváns objektum közül mennyit talált meg az algoritmus. Az alábbi ábra (ábra 27) szemléletesen ábrázolja ezeket a mutatókat. A mAP ebből a két értékből van származtatva. A mAP annak a függvénynek a grafikon alatti területe, amelynek két tengelye az előbb bemutatott precision és recall.



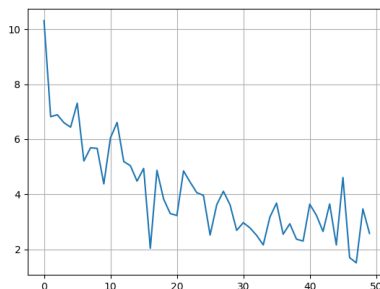
ábra 27: precision és recall szemléltetése, forrás: [62]

5.2 A tanítás módszertana

Az első futás során beállítottam egy kezdeti paraméterhalmazt, amivel megvizsgáltam, hogy milyen eredményt ér el a háló. Majd a kezdeti futás alapján módosítottam a háló struktúráját, hangoltam a hiperparamétereket vagy adtam hozzá funkciókat. A tanítások során állítottam a batch méretet, az epochok számát, kísérleteztem gradiens összegzéssel, alkalmaztam tanulási ráta ütemezőt, adataugmentációt, valamint töröltem felskálázó YOLO réteget. A tanításokhoz a DarkNet architektúra COCO adatbázison előre betanított súlyait használtam.

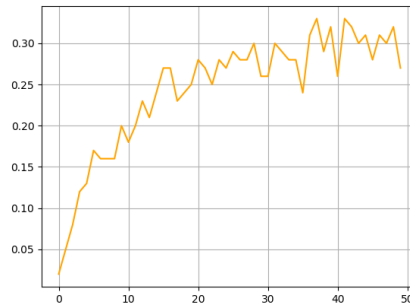
5.2.1 Első tanítás

Az első tanítás során csak arra voltam kíváncsi, hogy milyen eredményt ér el az algoritmus alapbeállítások segítségével. Ehhez 50 epochon át futtattam a tanítást, 8-as batch mérettel. Az alábbi két képen még csak a tanítási hiba és a validációs pontosság látható. Az első grafikonon megfigyelhető, hogy nagyon jelentős mértékben változik a tanítási hiba (ábra 28). Ennek oka az, hogy ekkor még nem átlagoltam az egyes batch-ekre számolt hibát minden epoch végén, hanem csak az utolsó batch hibáját mentettem el.



ábra 28: tanítási hiba, minden epoch utolsó batch-ére, nem átlagolt

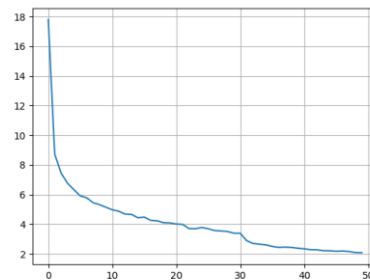
Az mAP grafikonján (ábra 29) az látható az első futás után, hogy nem véletlenszerűen tippelget a háló (25%-os pontosság), hanem valamit már megtanult. Ugyanakkor ez az eredményt még elmarad az elfogadható szinttől.



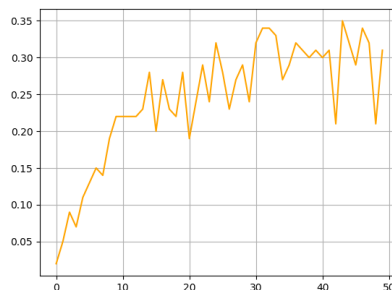
ábra 29: mAP az első futás után

5.2.2 Második tanítás

A második futás során már átlagoltam a tanítási hibát, ezáltal sokkal pontosabb képet kaptam arról, hogy hogyan teljesít a háló. E futás során is 50 epochon keresztül tanítottam, emellett egy tanulási ráta ütemezőt is elhelyeztem az algoritmusban. Ez az ütemező minden 30. epoch után tizedére csökkenti a tanulási rátát, ennek eredménye jól látható a grafikonon is (ábra 30).



ábra 30: tanítási hiba, a 30. epochnál jól láthatóan csökken a hiba a ráta csökkentése miatt

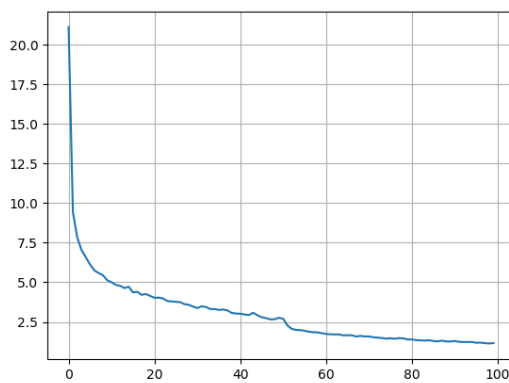


ábra 31: validációs mAP

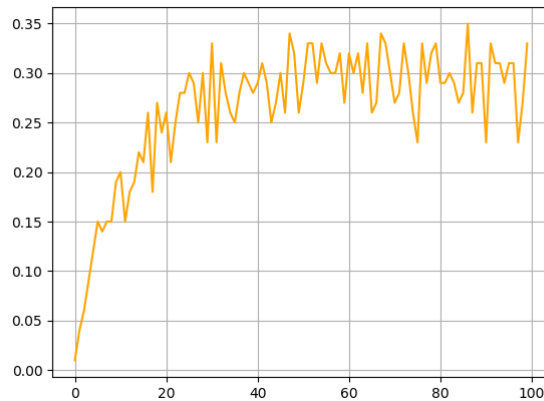
A validációs mAP-n jól látható a tanulási ráta ütemező jótékony hatása, a legjobb átlagos pontosság 35% (ábra 31), ami számottevő javulás az első tanításhoz képest.

5.2.3 Harmadik tanítás

A harmadik futás során elindítása előtt megnöveltem a batch számot 12-re, ennyi kép még befér a GPU memóriájába, az epochok számát felemeltem 100-ra és az 50. epochnál a tanulási rátát pedig tizedére csökkentettem.



ábra 32: tanulási hiba

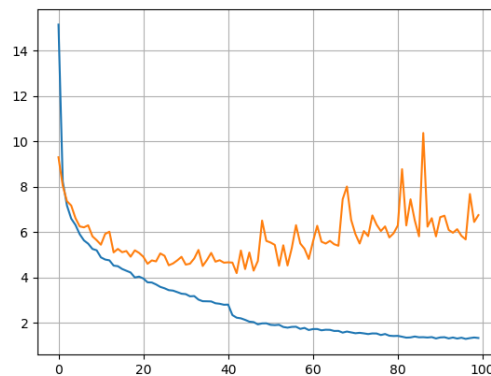


ábra 33: validációs mAP

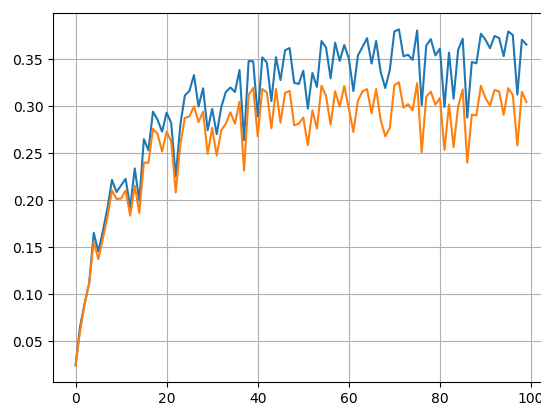
Az eredményeken (ábra 32), (ábra 33) számottevő javulás nem látható, ugyanis a mAP érték megint 35% környékén veszi fel a maximumát, mindazonáltal bizakodásra ad okot, hogy a validációs pontosság sem csökkent.

5.2.4 Negyedik tanítás

Az negyedik futás során a gradiens összegzést növeltem. Az eddig futások során a gradiens összegzések alapbeállítása 2 volt, ezt emeltem fel 4-re. A gradiens összegzést olyan esetekben érdemes alkalmazni, amikor a GPU memóriájába nem fér be elegendően nagy batchméret, emiatt pedig az egyes batch-ek nem lesznek reprezentatívak. A gradiens összegzés során gyakorlatilag virtuálisan növeljük a batch-ek méretet, viszont a memória használat nem növekedik. Ezzel a módszerrel a tanulási ráta nagyobb lehet, illetve a tanulás is stabilizálódik a gradiensek zajának csökkenése miatt. Ezt úgy lehet elérni, hogy az optimalizáló függvényével nem minden epoch után lépünk, csak minden n. batch esetén (n tipikusan 2 valamilyen hatványa) eközben pedig a gradienseket összegezzük minden ilyen mini batch-re. A tanítás 100 epochig tartott, 8-as batch size-al, 4 szerez gradiens összegzéssel, minden 40. és 80. epoch során pedig tizedére csökkentettem a tanulási rátát.



ábra 34: hibafüggvény, késsel a tanító, sárgával a validációs adatokon

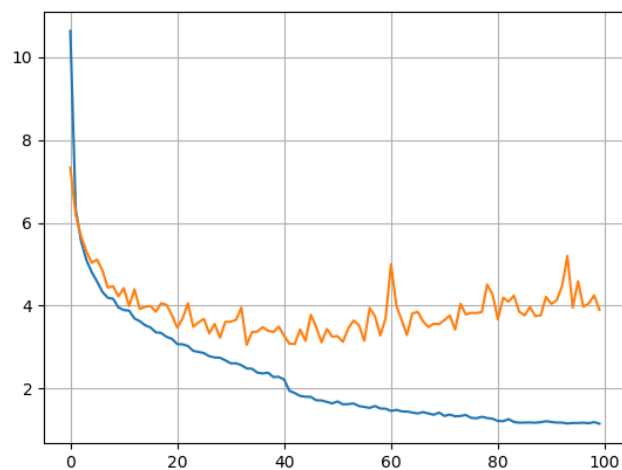


ábra 35: mAP, késsel a tanító, sárgával a validációs adatokon

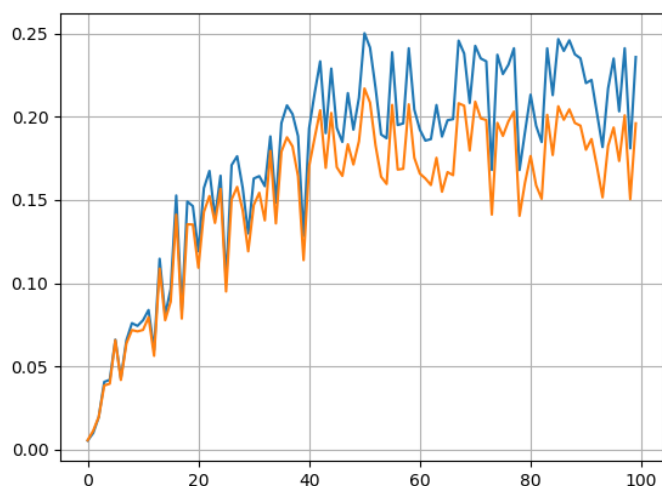
A tanítás során látható eredmények közül a hibafüggvények egymáshoz képesti változása igazán szemléletes (ábra 34), ezen jó látszik, hogy a 40. epoch után, ahol a tizedére csökkent a tanulási ráta, a tanítási hiba tovább csökkent, a validációs hiba viszont stagnálni, majd emelkedni kezdett. Ez valószínűleg azt jelenti, hogy hogy túlillesztettük a neurális hálót, mivel a tanítási hiba a tanítás végére alacsonyabb lett. De ezt nehéz megmondani, mert az előző esetben nem mentettem el a mAP változását.

5.2.5 Ötödik tanítás

Az ötödik futtatás során azzal próbálkoztam, hogy a tanító képeken adataugmentációt alkalmaztam, hogy változatosabb adatbázist hozzak létre. Minden egyes képen véletlen mértékben állítottam a szaturációt, a hue-t, a fényességet és a kontrasztot. Emellett töröltem a harmadik felskálázó YOLO réteget és az előtte található összes réteget, a második YOLO rétegig bezárólag. Ezzel az volt a célom, hogy elimináljam a hamis pozitív detekciókat, ugyanis a legnagyobb felbontáson működő kimenet hajlamosabb erre. Sajnos azonban az előző futtatáshoz képest rosszabb eredményt kaptam, és mivel két helyen is módosítottam az algoritmust, ezért egy újabb futtatás volt szükséges ahhoz, hogy kiderítsem, hogy melyik okozza a hibát. A futás során 8-as batch méretet, négyszeres gradiens összegzést alkalmaztam, 100 epochon át tanítottam és minden 40. epoch után tizedére csökkentettem a tanulási rátát.



ábra 36: hibafüggvény, késsel a tanító adatok, sárgával a validációs adatok

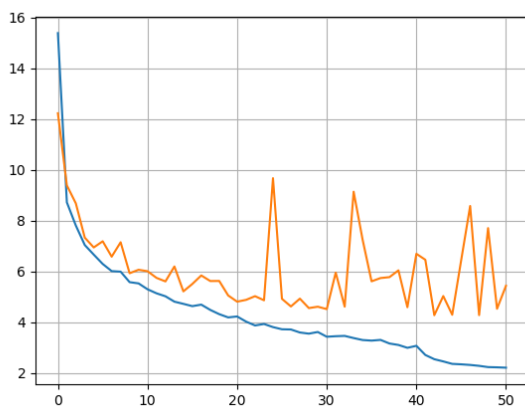


ábra 37: mAP, késsel a tanító, sárgával a validációs adatokon

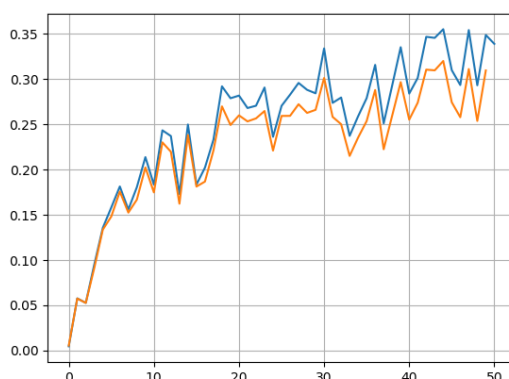
A grafikonokon (ábra 36), (ábra 37) látható, hogy az előző futáshoz képest 10%-kal rosszabb mAP értéket értünk el, a validációs hiba pedig a 40. epoch után elkezdett emelkedni, ami túlillesztést jelent.

5.2.6 Hatodik tanítás

A hatodik tanítás során azt derítettem ki, hogy melyik paraméter okozza az eredmények romlását, ezért visszatettem a harmadik YOLO réteget az augmentációt viszont benne hagytam. Most csak 50 epoch-on keresztül tanítottam az algoritmust, hiszen az ennyiből is megmondható, hogy a harmadik YOLO réteg hiánya okozta-e a hibát.



ábra 38: tanítási és validációs hiba a 8. futás során, késsel a tanítási, sárgával a validációs hiba

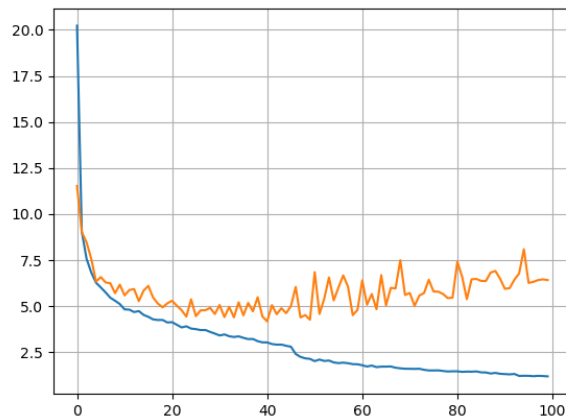


ábra 39: tanítási és validációs mAP a 8. futás során, késsel a tanítási, sárgával a validációs adatokon

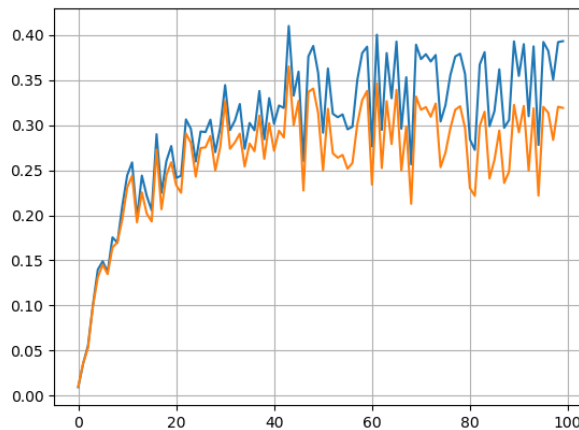
A fenti ábrákon látható (ábra 38) (ábra 39), hogy a mAP megint a maximális pontosság környékén van, tehát elmondható, hogy a harmadik YOLO réteg hiánya okozta a hibát. Az is látható, hogy az adat augmentáció nem okozott javulást a képeken. Ez nem feltétlen jelent rosszat, hiszen, ha változatosabb képeken is ugyanazt a teljesítményt produkálja a háló, akkor a teszt adatbázison is jobb eredményt érhet el. Mindezek ellenére az augmentáció mértékét csökkentettük.

5.2.7 Hetedik tanítás

A hetedik tanítás során megpróbáltam úgy megválasztani a paraméterek értékeit az eddigi tapasztalatok alapján, hogy a tanítás minél jobb legyen. A tanítás során 12-es batch méretet választottam, kétszeres gradiens összegzéssel. A tanulási ráta ütemezőt továbbra is alkalmaztam, viszont most minden 45. epoch után csökkentem tizedére a tanulási rátát. A tanítás során mindhárom YOLO layert alkalmaztam, valamint a képeken is véletlenszerűen végrehajtok augmentációs műveleteket, az eddigiekhez képest azonban kevésbé jelentős mértékben. A tanítást 100 epochon keresztül tartott.



ábra 40: tanítási (kék) és validációs (sárga) hiba



ábra 41: tanítási (kék) és validációs (sárga) mAP

A fenti (ábra 40) (ábra 41) eredményeken látható, hogy a paraméterek észszerű megválasztásával sikerült növelni a neurális háló teljesítményét, a legmagasabb validációs mAP a tanítás során 36.5%-nak adódott, ami az eddigi legjobb eredmény.

5.2.8 A validációs mAP kiértékelése

Az eddigiek során úgy próbáltam meg növelni a pontosságot, hogy a főleg olyan paramétereket változtattam, amelyek a tanító adatokra vannak hatással. Az eddigi fejezetekben ezeket a próbálkozásokat mutattam be. Mint az látható volt, egy bizonyos szinten túl nem voltam képes növelni a háló teljesítményét, ezért fenti próbálkozások után azzal kísérleteztem, hogy az eddigi tanítások legjobb modelljeire milyen hatással van az, hogy ha a validációs paramétereken változtatok. Az objektumaink kicsik, kicsi bounding boxokkal viszont nehéz pontos IoU-t csinálni, ezért kicsit enyhítem ezt a

követelményt. Így jelentősen javulhat az eredmény, amiből látszik, hogy a háló sok esetben megtalálja ugyan az objektumokat, csak nem elég pontos a lokalizáció.

Az IoU azt mutatja meg, hogy mekkora két bounding box átfedése, mennyire átlapoltak. Ezt úgy lehet megkapni, hogy ha elosztjuk a két bounding box metszetének a területét az uniójuk területével. E program esetén ezzel azt lehet megmondani, hogy mekkora legyen az a minimális átlapolódás a háló által detektált bounding box és a helyes bounding box között, ami fölött elfogadottnak tekintsük a detekciót. Ennek a paraméternek a csökkentésével akkor lehet jelentős javulást elérni a validációs során, ha sok az apró, széles és lapos vagy keskeny és magas objektum az adatbázisban. Az én adatbázisomra is igaz ez a kijelentés, mivel mind a gyalogos, mind a tábla, mind a jelzőlámpa osztályra elmondható, hogy kicsi és/vagy keskeny.

Ezért azt vizsgáltam meg, hogy hogyan változik a validációs mAP a fenti tanítások során kapott legjobb modellek esetén, hogy ha az IoU követelményt relaxálom. Az IoU paraméter mellett a másik hangolt változó a konfidencia küszöb volt. Ez azt mondja meg a hálónak, hogy milyen minimális konfidenciaszint felett mondja azt egy detekcióra, hogy az valóban objektum-e. A validációs script-ben ennek a paraméternek az alapértelmezett értéke 0.001 volt. Ez az érték alapvetően helytelen, hiszen ilyen paraméter érték mellett gyakorlatilag bármilyen detekcióra azt mondhatjuk, hogy objektum. Erre azonban csak az első IoU számítások után jöttem rá. Mindazonáltal e paraméter hangolásával is számos információ kiderülhet az adatbázisról. Az IoU változását így 0.001 és 0.5-ös konfidenciaszint mellett vizsgáltam meg.

táblázat 3: validációs mAP különböző IoU értékek esetén, 0.001-es konfidencia küszöb mellett

Conf threshold = 0.001	Második tanítás	Harmadik tanítás	Ötödik tanítás	Hatodik tanítás	Hetedik tanítás
IoU = 0.5	0.358	0.363	0.336	0.327	0.373
IoU = 0.25	0.537	0.521	0.505	0.508	0.550
IoU = 0.1	0.579	0.561	0.553	0.559	0.592
Precision (IoU=0.1)	0.041	0.138	0.108	0.050	0.032

A (táblázat 3) alapján látható, hogy az IoU paraméter hangolásával jelentős mértékben lehetett növelni az mAP értékét. A fenti értékek még 0.001-es konfidencia küszöb mellett lettek kiszámítva, ami miatt a precision metrika értéke még a legjobb, 0.1-es IoU érték mellett is rendkívül alacsony. Ezt orvosolandó elvégeztem ezeket a számításokat megemelt, 0.5-ös konfidencia küszöböt alkalmazva is.

táblázat 4: validációs mAP különböző IoU értékek esetén, 0.5-ös konfidencia küszöb mellett

Conf threshold = 0.5	Második tanítás	Harmadik tanítás	Ötödik tanítás	Hatodik tanítás	Hetedik tanítás
IoU = 0.5	0.350	0.345	0.325	0.319	0.365
IoU = 0.25	0.517	0.481	0.476	0.488	0.532
IoU = 0.1	0.555	0.513	0.519	0.534	0.570
Precision (IoU=0.1)	0.326	0.459	0.386	0.323	0.316

A (táblázat 4) alapján látható, hogy minimális mAP csökkenés mellett a precision értékét jelentős mértékben lehetett növelni, ami az algoritmus minőségét tekintve előnyös. A tanítás során ezek voltak az utolsó paraméterek, amelyek hangolásával igyekeztem javítani az algoritmus teljesítményét. Ezt követően rátértem a program tesztelésére, valamint a táblaosztályozásra is.

5.3 A táblaosztályozó neurális háló tanítása

A táblaosztályozó neurális hálót eltérő módszertan szerint tanítottam. Ahogy azt a fejlesztés részénél említettem, az algoritmus összes főbb változóját paraméterezhetőnek választottam meg. Ennek az volt az célja, hogy a tanítás során Bayes-i optimalizálást tudjak végrehajtani. Az optimalizálás egy megadott függvényt igyekszik maximalizálni, ezt a paraméterek változtatásával éri el. Ezen paraméterek határait nekem kell megadni, észszerű választások alapján. Fontos megjegyezni, hogy a maximalizáló függvény csak folytonos paramétereket alkalmaz, ezért bármilyen nem folytonos értéket nekem kell átkonvertálni. Az optimalizáció folytonos értékeket ad át a

függvénynek, ezért, ha diszkrét paraméterek is vannak a függvényben, akkor azt a megfelelő módon kell folytonos tartományból diszkrétre leképezni.

Az optimalizáció során megadható, hogy hány random lépést hajtson végre mielőtt elkezdené optimalizálni az értékeket (legyen ennek a száma x). Az optimalizálandó paraméterek száma legyen n ; ez az n paraméter meghatároz egy n változós függvényt, amelynek a maximumpontját igyekszik meghatározni. Az iterálás kezdetén x alkalommal ennek az n paraméternek véletlen értékeket ad a függvény. A függvény minden egyes paraméterkombinációra ad egy kimenetet, ezekre az optimalizáló algoritmus illeszt egy közelítő görbét. A következő iteráció során az illesztett görbe maximuma, valamint az ismeretlen kombinációk (pontok, tartományok) alapján határozza meg a következő paraméterkombinációt, amelyet be fog illeszteni a függvénybe. Az n ismétlés elvégzésével visszaadja azt a paraméterkombinációt, amellyel a legnagyobb kimenetet érte el. N iteráció után tehát ezek az értékek fogják a legjobb pontosságot adni, nekünk ezeket kell behelyettesíteni a függvényünkbe. Természetesen a diszkrét paramétereket konvertálni kell.

5.3.1 Tanítás megvalósítása

A megvalósításhoz létrehoztam egy függvényt, ami az optimalizáló függvény folytonos paramétereit a megfelelő módon diszkrét értékke konvertálja. Általános esetben a folytonos értékeket egész számokat egész típusú konvertáltam. Ez alól kivétel az első kimeneti csatorna mérete és batch méret paraméter, ahol kettő hatványára emelem a paramétereket, ezzel egyszerűsítve, pontosítva az algoritmust (kisebb tartományon könnyebben tud optimalizálni), valamint ezáltal a feladat specifikációjának is biztosan megfelelek. A paraméterek konvertálása után meghívom az eredeti tanító függvényt.

Következő lépésként minden egyes paraméterhez megadom azokat a határokat, amelyeken belül azok mozoghatnak. A maximalizáló függvény ezeket is meg fogja kapni. A függvény kimenete a validációs pontosság a Bayes-i optimalizálás ezt az értéket fogja maximalizálni a paraméterek hangolásával. Ezután már csak az optimalizálás meghívására van szükség, valamint a program sikeres futtatására.

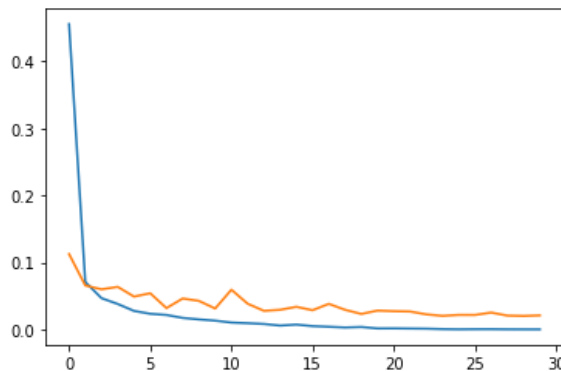
5.3.1.1 Optimalizálatlan tanítás

Miután teszteltem az eddig megírt programok helyességét elvégeztem a tanítást egy számomra megfelelőnek gondolt, de még nem optimalizált paraméterkombinációra. A kapott eredmények alább láthatóak, a paraméterek pontos értéke:

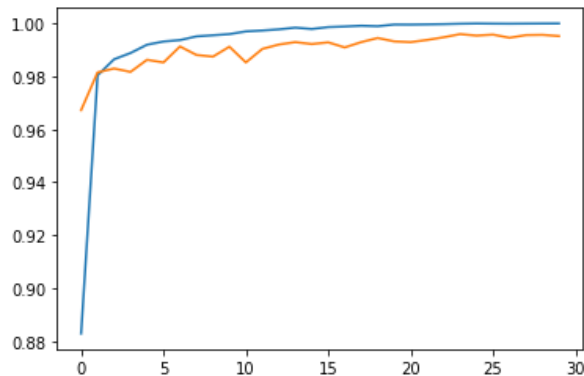
táblázat 5: optimalizálatlan paraméterek

Batchsize	Decay	Dropout	Kernel	Layers pe rlevel	LR	LR ratio	Nfeat	Nlevels	Res
64	1e-5	0.2	5	3	1e-3	0.1	8	3	1

Az eredményeken jól látható (ábra 42) (ábra 43), hogy megfelelően történik a tanítás, a veszteség az epochok során csökken, amelyet a validációs veszteség követ, nem tapasztalható overfitting. Ugyanez elmondható a pontosságról is.



ábra 42: Tanító és validációs veszteség az epochok függvényében (30 epoch). Késsel a tanítási, sárgával a validációs veszteség van jelölve.



ábra 43: Tanító és validációs pontosság az epochok függvényében (30 epoch). Késsel a tanítási, sárgával a validációs pontosság van jelölve.

A függvényünk a tanítás során megjeleníti a legjobb validációs veszteség és pontosság értékeket is. Ezen számértékek mögöttes tartalmára érdemes további figyelmet fordítani.

Legjobb validációs veszteség: 0.020

Legjobb validációs pontosság: 0.996

A két adat közül a validációs pontosság a beszédesebb, ezért ezt fogom részletesebben elemezni. A validációs pontosság megmutatja, hogy olyan adatokon, amelyekkel a tanítás során nem találkozott a neurális háló (tehát nem látta még az adott képet), milyen pontossággal ismeri fel. A tanítási adatbázisom közlekedési táblákat tartalmaz a feladat során cél, hogy ezeket a közlekedési táblákat minél jobban felismerje a neurális háló.

Az egyes adatsorokon kapott validációs pontosság azonban önmagában nem értelmezhető, tudnunk kell azt is, hogy az adott adatsor mennyire könnyen tanulható, milyen könnyen lehet elérni rajta magas validációs pontosságot. A grafikon ábráján láthatjuk, hogy ez az adatsor viszonylag könnyen tanulható, hiszen a neurális háló már az első epoch után is 97% körüli pontossággal rendelkezett. Ezek alapján megállapíthatjuk, hogy a kapott 99.6%-os eredmény egyáltalán nem kiemelkedő ezen az adatsoron.

5.3.2 Eredmények optimalizált tanítás mellett

A Bayes-i optimalizálás futtatása után legjobb eredmény paraméterei az alábbiak (táblázat 6) (táblázat 7):

táblázat 6: Folytonos paraméter értékek

Target	Batchsize	Decay	Dropout	Kernel	Layersperlevel	LR	LR_ratio	Nfeat	Nlvs	Res
0.98	7.24	1.0E-07	0.39	0.52	2.93	0.02	0.03	4.51	2.52	0.84

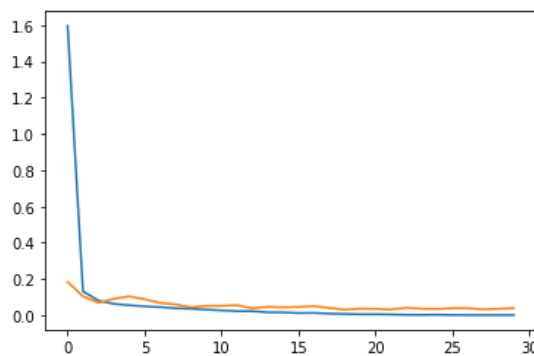
táblázat 7: diszkrét tartományba konvertált paraméter értékek

Target	Batchsize	Decay	Dropout	Kernel	Layersperlevel	LR	LR_ratio	Nfeat	Nlvs	Res
0.98	128.00	1.0E-07	0.39	3.00	3.00	0.02	0.03	32.00	3.00	1.00

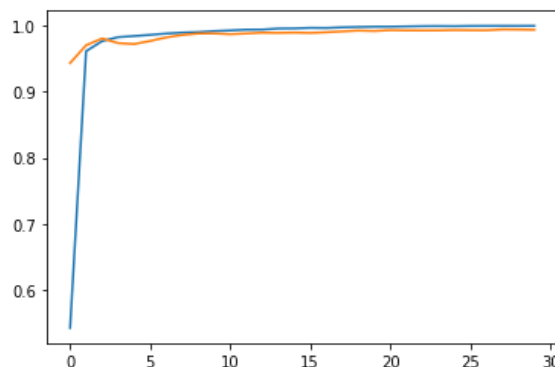
Ezen paraméterek mellett futtatott tanítás után a validációs pontosság 99.5%-os lett. Ezen eredmény mellett is érdemes elidőzni egy kicsit. Ugyan ez az érték kisebb,

mint az optimalizálatlan tanítás során kapott 99.6%-os validációs pontosság ugyanakkor érdemes megjegyezni, hogy az optimalizálás során csak az adatbázis 20%-a volt felhasználva, valamint 30 epoch helyett csak 20 epochon át tanítottam (ábra 44) (ábra 45). Ennek oka az volt optimalizálást Google Colaboratory-ban végeztem, a Google e célból fenntartott GPU egységein. Így az optimalizálást sajnos nem tudtam volna elvégezni a teljes adatsoron, mivel a szolgáltatás egyszerre 12 óra folytonos GPU használatot engedélyez és a csökkentett adatsoron való optimalizálás is 8 óráig futott. Ennek az információnak a birtokában már érthető, hogy miért nem kaptam jobb eredményt, mint nem optimalizált esetben.

Legjobb validációs veszteség optimalizált hiperparaméterek esetén: 0.030
Legjobb validációs pontosság optimalizált hiperparaméterek esetén: 0.995



ábra 44: Tanítói és validációs veszteség optimalizált hiperparaméterek mellett, az epochok függvényében (30 epoch). Kékkel a tanítási, sárgával a validációs veszteség van jelölve.



ábra 45: Tanítói és validációs pontosság optimalizált hiperparaméterek mellett, az epochok függvényében (30 epoch). Kékkel a tanítási, sárgával a validációs pontosság van jelölve.

A neurális háló pontossága tovább lehet javítható, ha a Bayesian Optimization maximalizáló függvényében, a hangolni kívánt paramétereket több körben szűkítjük, valamint, ha az egész adatsort használjuk, egy olyan GPU-n, ami ezzel a feladattal belátható időn belül végez.

5.4 Eredmények a tesztadatbázison

A tanítás során több módszert kipróbáltam, számos paraméter hangolásával igyekeztem növelni a validációs mAP értékét. A tanítások elvégzését követően, a legjobb mAP értéket szolgáltató modell súlyait felhasználva elvégeztem a validálást a valós képekből álló tesztadatbázison is.

Ehhez az algoritmust tesztmódba kapcsoltam, ami azt jelenti, hogy ezeken az adatokon nem tanítom a programot. A tesztelés gyakorlatilag megegyezik a validálással a módszert tekintve, azzal a jelentős különbséggel, hogy más adatbázison dolgozik az algoritmus. A tesztadatbázison történő validálást követően az alábbi eredményeket kaptam.

A (táblázat 8) adatai alapján látható, hogy a háló jelentősen rosszabbul teljesít a tesztadatbázison, összevetve a legjobb validációs pontossággal, ami 57%. Az AP értékek közül is különösen a „person” osztály esetén volt rossz az algoritmus, itt mindössze valamivel több mint 1%-os eredményt ért el.

táblázat 8: Eredmények a tesztadatbázison

	AP	Recall	Precision
Class 0 - person	0.0127	0.0382	0.2109
Class 1 - vehicle	0.0645	0.1353	0.2876
Class 2 – traffic sign	0.0813	0.2607	0.1712
Class 3 – traffic light	0.0754	0.1506	0.2671
Átlag	0.0585 (mAP)	0.1462	0.2342

A jelentős teljesítménybeli különbségnek a validációs és tesztadatbázisok között számos oka lehet, amelyek közül néhányat be is szeretnék most mutatni. A különbség forrását nem feltétlen a tanítás hibája, rossz módszere jelenti, hiszen a validációs

eredmények alapján látható, hogy a háló meg tudja tanulni detektálni a tanító adatbázisban szereplő objektumokat. Természetesen a tanítás paramétereinek hangolásával további javulást lehetne elérni, de azt gondolom, hogy ez nem lenne jelentős. A romlás okát sokkal inkább a két adatbázis közötti különbségek okozzák.

Összevetve például az alábbi (ábra 46) teszt képet egy, a tanító adatbázisban szereplőével (ábra 21) látható, hogy sokkal mozgalmasabb képet tartalmaz a tesztadatbázis. Egy adott képen sokkal több, sokkal változatosabb objektum(ka)t kell felismernie az algoritmusnak, mint amilyenekkel a tanítás során találkozott. Ezzel a feladattal pedig - mint látható - nem tudott megbirkózni. További ok lehet az, hogy sokkal részletgazdagabbak a valós környezetben készült képek, mint a szintetikusán generált társai. A „person” osztály különösen rossz eredményének oka lehet az is, hogy a tesztadatbázisban számos olyan kép szerepel, amelyek nagy embertömegeket tartalmaznak.



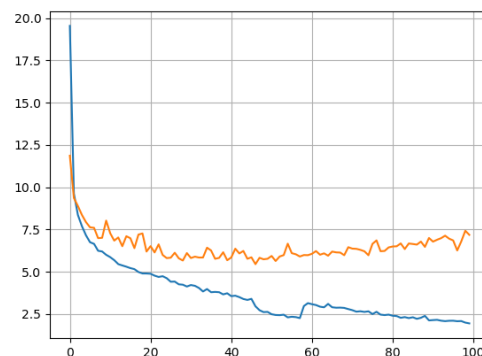
ábra 46: kép a tesztadatbázisból

5.4.1 Eredmények az adatbázisok módosításával

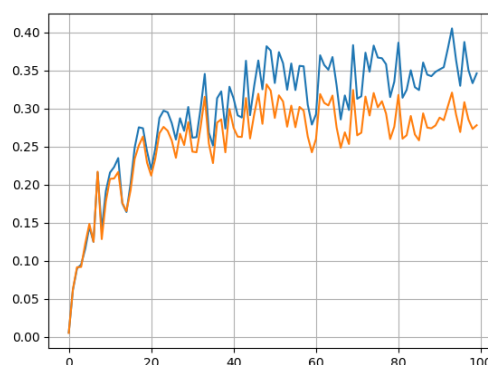
Ezért úgy döntöttem, hogy megvizsgálom azt is, hogy valóban a tanítás hibája-e teszt adatbázison produkált rossz teljesítmény, vagy inkább a két adatbázis közötti különbségek okozzák ezt a jelentős eltérést. Ezt úgy vizsgáltam meg, hogy mind a tanító, mind a validációs adatbázisba beletettem a COCO adatbázisból származó módosított képeket, amiket eredetileg csak a tesztadatbázishoz használnék. Itt fontosnak tartom megjegyezni azt a tényt, hogy nem a tesztadatbázisból tettem át képeket a tanító

és validációs adatbázisba, hanem teljesen új képeket használtam fel a COCO adatbázisból.

Az eddigiekben a tesztadatbázisomhoz összesen 1024 COCO-s képet generáltam, amire még ráhelyeztem a plusz közlekedési táblákat. A mostani esetben kétszer ennyi, 2048 képet generáltam, amelynek a fele továbbra is a tesztadatbázist alkotja. A fentmaradó 1024 kép 60%-át a tanító adatbázisban, a maradék 40%-ot pedig a validációs adatbázisban helyeztem el. A COCO-s adatbázis generálása során még annyit módosítottam, hogy kivettem az „iscrowd” paraméterrel ellátott annotációkat. Ezek olyan bounding box-okat tartalmaznak, amelyek tömörülő példányokat tartalmaznak az adott osztályból (például elefántcsorda, egy csoport ember, vagy dugóban álló autók). Az „iscrowd” komponenssel ellátott annotációk a tanító adatbázis connected components metodikával generált bounding box-ok limitációja miatt okozhatnak problémát, ezt is orvosolja, hogy ezeket az annotációkat kivettem. Ezt követően indítottam egy új tanítást, a 7. tanítással megegyező paraméterek mellett.



ábra 47: tanító (kék) és validációs (sárga) hiba a módosított adatbázison



ábra 48: tanító (kék) és validációs (sárga) mAP a módosított adatbázison

Az eredményeken (ábra 47) (ábra 48) látható, hogy a módosított adatbázison is hasonló validációs mAP (32%) maximumot tud elérni az algoritmus, mint az eredeti adatbázisok használatával. A hibafüggvényben látható megemelkedő hibának az 58. epochnál az oka az, hogy nem tudtam egyben lefuttatni a tanítást, csak az 58. epochig jutottam el, itt mentette utoljára a program a modellt. Ezért amikor folytattam a tanítást ezt a modellt töltöttem be és innen futtattam tovább a tanítást, ez okozza az inkonzisztenciát.

A fenti adatok alapján elmondható, hogy a módosított adatbázisok alkalmazásával is hasonló pontosságot lehet elérni az algoritmussal, a sokkal inkább meglepő eredményeket a tesztadatbázison mért eredmények fogják szolgáltatni. A tanítás befejezését követően ezt vizsgáltam meg. A validálást 0.5-ös konfidencia és 0.1-es IoU értékek mellett végeztem el.

táblázat 9: teszteredmények a módosított adatbázison betanított algoritmussal

	AP	Recall	Precision
Class 0 - person	0.4172	0.7596	0.2399
Class 1 - vehicle	0.4074	0.6936	0.2416
Class 2 – traffic sign	0.9083	0.9425	0.7164
Class 3 – traffic light	0.3710	0.5376	0.3085
Átlag	0.5260 (mAP)	0.7333	0.3766

Az eredmények elsőre meghökkentő mértékű javulást mutatnak, az előző, alig 6%-os mAP-hoz képest most 52.6%-ra volt képes az algoritmus. Ennek egyik oka a tanító és validációs adatbázisban elhelyezett valós képek, ami bizonyítja azt is, hogy nem a tanításban volt valamilyen hiba, egyszerűen a két adatbázis ennyire különbözik. A másik rendkívül érdekes adat a „traffic sign” osztályon elért kiemelkedően magas, 90%-os AP. Ennek oka az, hogy a közlekedési táblák túlnyomó részét mindhárom adatbázisban (tanító, validáló, tesztelő) a COCO-s képekre ráillesztett saját közlekedési táblák alkotják. Erről a közlekedési tábla adatbázisról tudni kell, hogy egy könnyen

tanulható adatbázis, amit az is mutat, hogy a táblaosztályozó neurális hálóm 95-97%-os pontossággal tudja osztályozni ezeket a táblákat. Ez az eredmény pedig a jelentős mértékben növeli meg a négy osztály átlagára számított eredményeket. Ha kivennénk ezt az osztályt, akkor a következő átlagértékeket kapnánk (táblázat 10):

táblázat 10: Teszteredmények a "traffic sign" osztály nélkül

	mAP	Recall	Precision
Átlag	0.3985	0.6636	0.2633

Jól látható, hogy igen jelentős mértékben emeli meg a „traffic sign” osztály az összes osztályra számolt átlagértékeket. Azonban, ha ettől az osztálytól eltekintünk még úgy is számottevő mértékben, 33%-kal emelkedett meg az mAP értéke a tisztán szintetikus tanító és validációs adatbázisokon betanított háló eredményéhez képest.

6 Összefoglalás, az eredmények értékelése:

Az algoritmus a tesztadatbázisokon kielégítő teljesítményt nyújtott. A tanító paraméterek hangolásával el tudtam érni azt, hogy 30%-os kezdeti mAP értékről 37%-ig emeljem a validációs mAP eredményt, ezt követően pedig az IoU küszöb enyhítésével könnyítettem meg az algoritmus számára a követelményeket, így 57%-ra is képes volt.

Abban az esetben, hogy ha szigorúan csak szintetikusán generált képeken tanítom és validálom a neurális hálót, akkor csak nagyon kis mértékben tud általánosítani két adatbázis között. Kijelenthető az is, hogy ezzel a módszerrel nem képes arra az algoritmus, hogy valós környezetben is megfelelő minőségű működést szolgáltatson, a 6%-os mAP nagyon kevés egy ilyen alkalmazás esetén.

Azonban, ha megengedjük azt, hogy a tanító és validáló adatbázisok ne csak tisztán szintetikus képekből álljanak, hanem egy része (10-20%) valós képekből álljon, akkor a neurális háló az előző esethez képest sokkal jobb, 50%-ot meghaladó eredményre képes, ez pedig a validációs eredmények tükrében kielégítő.

Valósídejűség szempontjából is megfelelő sebességgel tud futni az algoritmus. A validációs adatokon az algoritmus másodpercenként 2-3 iterációt képes végrehajtani, ami azt jelenti, hogy 1-es batch méret esetén 1 másodperc alatt 2-3 képet képes feldolgozni, ez pedig egy mozgóképek FPS szintjével egy nagyságrendbe esik, de annál valamivel kisebb. Ez a sebesség természetesen egy átlagérték, ami jelentős mértékben függ attól, hogy az adott képen hány objektum található. Mozgalmasabb képek esetén lassabb a feldolgozás, kevesebb objektumot tartalmazó képek esetén viszont gyorsabban dolgozik a program.

6.1 Lehetőségek az eredmények további javítására

Természetesen az eredmények további javítására még számos lehetőség van. Először is, lehetőség van az adatbázisban generált képek javítására is. Jelenleg a CARLA Szimulátor 8 várost és nagyjából 10 különböző járműtípust tartalmaz. Több város és járműtípus bevezetésével további javulásra lehet számítani. Továbbá a Szimulátorban generált képek minőségének, részletgazdagságának, valószerűségének növelésével is javítani lehetne az eredményeket. Azzal is tudnék még kísérletezni, hogy

az összes kép mekkora százaléka készül sötétben és mennyi világosban. Ennek meghatározására meg lehetne vizsgálni azt, hogy a tesztadatbázisban szereplő képek mekkora hányad készült világosban és mekkora sötétben.

További lehetőség a teljesítmény és pontosság növelésére az, hogy ha a képekhez generált annotációkat, bounding box-okat finomítom. Ennek érdekében generáltam a mélységinformációkat tartalmazó képet is. Ezt felhasználva felbonthatóak lennének az összenőtt objektumok (például kocsisorok), amik az algoritmus pontosságát csökkentik. Azzal is lehet kísérletezni, hogy mi legyen az a minimális objektum méret, ami alatt már nem veszem fel az adott objektumot az annotációk közé.

A tanítás minőségét is lehet természetesen javítani. Tovább lehetne kísérletezni azzal, hogy milyen paraméterkombináció az, amely mellett a legjobb pontosságra képes a neurális háló. Ha rendelkezésre állna megfelelően nagy teljesítményű hardver, akkor akár ezen a neurális hálón is lehetne Bayes-i optimalizálást alkalmazni, hogy meghatározzam az ideális paraméter értékeket.

További javítási lehetőség a táblaosztályozó algoritmus integrálása a YOLO detektáló programba. Jelenleg a táblaosztályozó attól függetlenül fut, így e feladat elvégzésével is lehet javítani a programot.

Összességében tehát elmondható, hogy a jelenlegi eredmények kielégítőek, de természetesen számos mód áll rendelkezésre ahhoz, hogy a neurális háló teljesítményét tovább lehessen javítani.

7 Irodalomjegyzék

- [1] Tom Mitchell: Machine Learning (1997): ISBN 978-0-07-042807-2.
<http://www.cs.cmu.edu/~tom/mlbook.html>
- [2] Ethem Alpaydin (2020). Introduction to Machine Learning (Fourth ed.).
<https://www.springer.com/gp/book/9780387310732>
- [3] Bishop, C. M. (2006), Pattern Recognition and Machine Learning, Springer, ISBN 978-0-387-31073-2
- [4] Samuel, Arthur (1959). "Some Studies in Machine Learning Using the Game of Checkers". *IBM Journal of Research and Development*. 3 (3): 210–229
<https://ieeexplore.ieee.org/document/5392560>
- [5] Nilsson N. Learning Machines, McGraw Hill, 1965.
<https://psycnet.apa.org/record/1965-35030-000>
- [6] Duda, R., Hart P. Pattern Classification and Scene Analysis, Wiley Interscience, 1973
<https://pdfs.semanticscholar.org/b07c/e649d6f6eb636872527104b0209d3edc8188.pdf>
- [7] S. Bozinovski "Teaching space: A representation concept for adaptive pattern classification" COINS Technical Report No. 81-28, Computer and Information Science Department, University of Massachusetts at Amherst, MA, 1981.
<https://web.cs.umass.edu/publication/docs/1981/UM-CS-1981-028.pdf>
- [8] Russell, Stuart; Norvig, Peter (2003) [1995]. Artificial Intelligence: A Modern Approach (2nd ed.). Prentice Hall. ISBN 978-0137903955.
<https://www.cin.ufpe.br/~tfl2/artificial-intelligence-modern-approach.9780131038059.25368.pdf>
- [9] Langley, Pat (2011). "The changing science of machine learning". Machine Learning.
https://www.researchgate.net/publication/220343986_The_changing_science_of_machine_learning
- [10] A gépi tanulás és a mesterségs intelligencia kapcsolata: "rasbt/stat453-deep-learning-ss20"
- [11] Garbade, Dr Michael J. (14 September 2018). "Clearing the Confusion: AI vs Machine Learning vs Deep Learning Differences". <https://towardsdatascience.com/clearing-the-confusion-ai-vs-machine-learning-vs-deep-learning-differences-fce69b21d5eb>
- [12] Dr. Sebastian Raschka. "Chapter 1: Introduction to Machine Learning and Deep Learning". <https://sebastianraschka.com/blog/2020/intro-to-dl-ch01.html>

- [13] Pearl, Judea; Mackenzie, Dana. The Book of Why: The New Science of Cause and Effect (2018 ed.). Basic Books. ISBN 9780465097609.
<https://www.basicbooks.com/titles/judea-pearl/the-book-of-why/9780465097609/>
- [14] A. G. Ivakhnenko, Valentin Grigor'evich Lapa: Cybernetics and forecasting techniques (1967)
https://books.google.hu/books?id=rGFgAAAAMAAJ&redir_esc=y
- [15] Rina Dechter: Learning While Searching in Constraint-Satisfaction-Problems (1986):
https://www.researchgate.net/publication/221605378_Learning_While_Searching_in_Constraint-Satisfaction-Problems
- [16] Juergen Schmidhuber: Deep Learning (2015):
http://www.scholarpedia.org/article/Deep_Learning
- [17] Matthew D Zeiler, Rob Fergus: Visualizing and Understanding Convolutional Networks (2013): <https://arxiv.org/abs/1311.2901>
- [18] Kunihiro Fukushima: Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position (1980):
<https://www.cs.princeton.edu/courses/archive/spr08/cos598B/Readings/Fukushima1980.pdf>
- [19] Konvolúció aritmetika:
https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md
- [20] Maximum pooling és átlagos pooling:
https://www.researchgate.net/figure/Illustration-of-Max-Pooling-and-Average-Pooling-Figure-2-above-shows-an-example-of-max_fig2_333593451
- [21] Aktiváció függvények: <https://7-hiddenlayers.com/deep-learning-2/>
- [22] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio: Generative Adversarial Nets (2014): <https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>
- [23] Cross Entropy Loss, MSE Loss: <https://pytorch.org/docs/stable/nn.html#loss-functions>
- [24] Diederik P. Kingma, Jimmy Ba: Adam: A Method for Stochastic Optimization (2017): <https://arxiv.org/abs/1412.6980>
- [25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun: Deep Residual Learning for Image Recognition (2015): <https://arxiv.org/pdf/1512.03385.pdf>
- [26] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich: Going deeper with convolutions (2014): <https://arxiv.org/pdf/1409.4842v1.pdf>

- [27] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna: Rethinking the Inception Architecture for Computer Vision (2015): <https://arxiv.org/pdf/1512.00567v3.pdf>
- [28] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi: You Only Look Once: Unified, Real-Time Object Detection (2015): <https://arxiv.org/pdf/1506.02640.pdf>
- [29] Joseph Redmon, Ali Farhadi: YOLOv3: An Incremental Improvement (2018): <https://pjreddie.com/media/files/papers/YOLOv3.pdf>
- [30] Bounding box leképezés. <https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py>
- [31] Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik: Rich feature hierarchies for accurate object detection and semantic segmentation (2014): <https://arxiv.org/pdf/1311.2524.pdf>
- [32] Ross Girshick: Fast R-CNN (2015): <https://arxiv.org/pdf/1504.08083.pdf>
- [33] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks (2016): <https://arxiv.org/pdf/1506.01497.pdf>
- [34] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollár: Focal Loss for Dense Object Detection (2018): <https://arxiv.org/abs/1708.02002>
- [35] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg: SSD: Single Shot MultiBox Detector (2015): <https://arxiv.org/abs/1512.02325>
- [36] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick: Mask R-CNN (2017): <https://arxiv.org/abs/1703.06870>
- [37] Inception V2 Deep Convolutional Architecture: https://www.researchgate.net/publication/337294435_An_Efficient_Inception_V2_based_Deep_Convolutional_Neural_Network_for_Real-Time_Hand_Action_Recognition
- [38] Inception V3 Deep Convolutional Architecture <https://software.intel.com/content/www/us/en/develop/articles/inception-v3-deep-convolutional-architecture-for-classifying-acute-myeloidlymphoblastic.html>
- [39] Objektum meghatározási módszerek: <https://medium.com/zylapp/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852>
- [40] Perceptron modell: <https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/>
- [41] Többretegű perceptron modell: https://medium.com/@AI_with_Kain/understanding-of-multilayer-perceptron-mlp-8f179c4a135f

- [42] Konvolúciós neurális háló struktúra: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>
- [43] A YOLO algoritmus működése: <https://appsilon.com/object-detection-yolo-algorithm/>
- [44] CARLA Szimulátor: <https://carla.org/>
- [45] COCO adatbázis: <https://cocodataset.org/#home>
- [46] A DarkNet architektúra: <https://pjreddie.com/darknet/>
- [47] Visual Studio Code: <https://code.visualstudio.com/>
- [48] PyCharm Editor: <https://www.jetbrains.com/pycharm/>
- [49] Anaconda: <https://www.anaconda.com/>
- [50] TensorFlow: <https://www.tensorflow.org/>
- [51] Keras: <https://keras.io/>
- [52] PyTorch: <https://pytorch.org/>
- [53] YOLOv3 implementáció a PyTorch keretrendszerrel használva: <https://github.com/eriklindernoren/PyTorch-YOLOv3>
- [54] CARLA város és térképleírás: https://carla.readthedocs.io/en/latest/core_map/
- [55] Freund, Y.; Schapire, R. E. (1999). "Large margin classification using the perceptron algorithm" <https://cseweb.ucsd.edu/~yfreund/papers/LargeMarginsUsingPerceptron.pdf>
- [56] David A. Winkler (2016) Performance of Deep and Shallow Neural Networks, the Universal Approximation Theorem, Activity Cliffs, and QSAR <https://onlinelibrary.wiley.com/doi/abs/10.1002/minf.201600118>
- [57] YOLOv4: <https://arxiv.org/abs/2004.10934>
- [58] Anchor boxok: <https://towardsdatascience.com/yolo2-walkthrough-with-examples-e40452ca265f>
- [59] OpenCV: <https://opencv.org/>
- [60] COCO-YOLO átalakító: <https://github.com/ssaru/convert2Yolo>
- [61] Bayes-i optimalizálás: <https://github.com/fmfn/BayesianOptimization>
- [62] Precision and recall: <https://towardsdatascience.com/whats-the-deal-with-accuracy-precision-recall-and-f1-f5d8b4db1021>