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

Dobai Botond

HATÉKONY NEURÁLIS HÁLÓZAT TERVEZÉSE FORGALOMBAN RÉSZT VEVŐ OBJEKTUMOK FELISMERÉSÉRE

KONZULENS

Hadházi Dániel

BUDAPEST, 2018

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Dobai Botond**, 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, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens 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, 2018. 12. 07.

...…….

Dobai Botond

Összefoglaló

Jelenleg egyre nagyobb igény van a járműiparban a hatékony vezetést segítő rendszerekre. Ahogy a hardverek és a technológia fejlődik, úgy egyre nagyobb szerepet kap a neurális háló-alapú objektum detektálás ebben a témakörben. Figyelembe kell venni azt is, hogy egy beágyazott rendszeres környezetben gyakran korlátozott a rendelkezésre álló memória-, illetve számítási kapacitás. Ezért szükséges egy kompromisszumot kötni a teljesítmény és az erőforrásigény között.

Ebben a szakdolgozatban azt vizsgáltam, hogy milyen létező hálóstruktúrát érdemes használni ennek tekintetében, és hogy hogyan lehet egy létező neurális hálót hatékonyabbá tenni. Számos optimalizációs módszer létezik, amelyeknek alapvető célja a memóriaigény csökkentése, az elvégzendő műveletek számának csökkentése, de optimalizációnak nevezhető az is, ha egy adott komplexitású hálóval a korábbinál jobb eredményt érünk el.

Ezekkel a témakörökkel kapcsolatban végeztem irodalomkutatást, illetve próbáltam a szerzett tapasztalatokat a gyakorlatba is átültetni.

Abstract

There is an increasing demand for efficient driver-assistance systems in the vehicle industry. As hardwares and technology have became more advanced, neural network-based solutions in object detection get a highlighted role in this topic. We need to consider that in embedded systems the amount of available memory- and processing capacity might be limited. Because of this reason, we need to make a trade-off between the performance and the hardware-requirements of our solution.

In my current thesis I have examined the existing network-structures according to this problem, furthermore the possibilities of making an existing neural-network more efficient. There are several existing optimalization techniques that aim to reduce the memory-consumption and the number of computing operations. We can even call the intention of getting better results with a given complexity as optimalization.

I have made some literature research in connection with these topics, and I have tried to use the gained knowledge in practice.

1 Bevezetés

1.1 Járművezetőt segítő rendszerek

A szakdolgozat a járművezetőt támogató rendszerek, illetve az autonóm járművek témakörével foglalkozik. Ez egy nagyon tág és szerteágazó terület, amely – különösen az utóbbi évtizedben – nagy figyelmet kapott az autóiparban. A nagy érdeklődés annak köszönhető, hogy olyan újdonságokat vezet be, amelyekben a piac szereplői (vásárlók, autógyártók, tömegközlekedők, biztosítók) erősen érdekeltek.

Ezek az újdonságok részben kényelmi funkciókat jelentenek, mint például egy adaptív sávtartó, vagy egy parkolást segítő rendszer. De akár olyan közvetettebb, kényelmi megoldásokat is ide lehetne sorolni, mint egy adaptív futómű, amely szenzor- és kamera adatokat képes felhasználni.

Ami a vásárlók számára ennél fontosabb felhasználási terület, az az utasbiztonság növelése, illetve a kárcsökkentés. A kockázat csökkentése mind a magánemberek, mind a tömegközlekedést üzemeltető cégek számára nagyon fontos szempont, hiszen jelentős összegek vagy akár emberéletek is foroghatnak kockán.

Ilyen biztonsági funkció lehet egy előzést segítő rendszer, egy vészfékező rendszer, vagy különböző (fél)autonóm járművezető rendszerek [1]. Általánosságban itt a legalapvetőbb célok közé tartozik az emberi figyelmetlenség kiküszöbölése [2], a holttérfigyelés, illetve a rossz látási viszonyok legyőzése különböző szenzoradatok feldolgozása által.

Számos példát lehet hozni mind nemzetközi [3][4], mind hazai [5][6] projektekre, amelyek jelentős tőkeinjekcióval, több autóipari partner támogatásával kutatják ezt a témakört.

Hardvergyártói oldalról is látszik az igény növekedése. A különböző szenzorok egyre olcsóbban beszerezhetőek a tömeggyártásnak hála. A feldolgozó egységek is egyre nagyobb mértékben támogatják azon numerikus módszerek hatékony, párhuzamos végrehajtását, amelyeket például a neurális háló alapú képfeldolgozásban is használnak [7][8][9].

1.2 Forgalomi résztvevők detektálása

Ezen a tág területen belül most csak a forgalomban részt vevő objektumok detektálását tárgyaljuk, és azon belül is a kamera alapú megoldásokról lesz szó.

Példának vegyünk egy ütközés-elkerülő rendszert, hogy megértsük, hogyan is kapcsolódik az objektum-detektálás ehhez a témakörhöz. Képfeldolgozás szinten két fontos részfeladatunk van: Az első az, hogy kialakítsunk egy „veszélyzónát”, azaz határozzuk meg azt az utat a kamera képén, ahol a saját jármű várhatóan haladni fog. A második pedig az, hogy minél pontosabban határozzuk meg az ebben a zónában tartózkodó objektumok helyét, kiterjedését és típusát.

Erre a feladatra már vannak viszonylag kulcsrakész megoldások, azonban mivel a járműiparban tömeggyártott, beágyazott rendszeres megoldásokról kell beszélnünk, így a költségminimalizálás és a hardveres fogyasztáscsökkentés céljából több szempontból is kompromisszumok közé vannak szorítva a lehetséges megvalósítások.

Az első döntést az algoritmus működésével kapcsolatban kell meghozni. A klasszikus képfeldolgozási algoritmusok során különböző prekoncepciókra építve végezünk a képsoron szűréseket, alakzatdetektálásokat, transzformációkat, esetleg geometriai és fizikai kényszereknek feleltetjük meg a kapott eredményt.

Mivel ezek nagy általánosságban összetett algoritmusok különböző kódon belüli függőségekkel, így várhatóan lesznek nehezen párhuzamosítható részei. Illetve, ha az algoritmus nem általánosítható megfelelően bizonyos bemenetekre, az gyakran csak a kód komolyabb átalakításával orvosolható, ami jelentősen növelheti a fejlesztési költségeket.

Ezzel párhuzamban a neurális hálós megközelítések ilyen szempontból szűkebb paradigmákkal operálnak, amik jobban magukban hordozzák az általánosítóképességet, az architektúrák újra-felhasználhatóságát, és a hatékony párhuzamosítás lehetőségét.

A különböző szakmai versenyeken elért eredmények [10] is a neurális háló alapú megoldások rohamos fejlődését mutatják; azonban ezek az architektúrák viszonylag magas számítási igénnyel, és magas memóriaigénnyel rendelkeznek. Ezt szem előtt tartva kompromisszumot kell kötni ahhoz, hogy a háló valós időben fusson az adott hardveren.

Az első kompromisszum az objektumdetektor kimenetével kapcsolatos. Lehetőségünk van olyan hálót használni, amely csak befoglaló dobozokat (bounding-box) ad vissza

lokalizációs eredményként. Olyan hálót is használhatunk, amely pixel-pontosan szegmentálja a képet. Utóbbi megoldás nyilvánvalóan nagyobb pontosságot tesz lehetővé, azonban költségesebb is. Ezért érdemesebb a bounding-box alapú megoldásoknál maradni. mivel várhatóan kevésbé erőforrásigényes. Átmeneti megoldásként pontosíthatjuk úgy az eredményünket, hogy bizonyos detekciókat a bounding-box alapján kivágunk, és a kisebb felbontású, kevésbé változatos képen futtatunk egy hatékony szegmentáló hálót is. Ez a lépés a fals-pozitív (FP) esetek számát hivatott csökkenteni. (Például az ütközés-elkerülő rendszeres példánál maradva ezzel bizonyosodhatunk meg róla, hogy a veszélyzónába éppen belógó bounding-boxok valóban hasznos észlelések-e.)

A következő kihívás, ami felmerül, az a megfelelő háló-architektúra kiválasztása. Nagyságrendnyi különbségek lehetnek a különböző hálók futásidejében, mivel hálóról-hálóra különbözhet a paraméterek száma, illetve az elvégzendő műveletek számítási ideje. Tehát első lépésben összehasonlító vizsgálatok alapján ki kell választani azokat a jelölteket, amelyek az elvárásainknak megfelelő erőforrásigény-teljesítmény arányt hozzák.

1.3 Transfer learning

A hálóarchitektúra betanítása történhet előzetes tanítás nélkül (from-scratch), vagy egy általánosabb, de hasonló feladatra betanított háló transfer learning módszert alkalmazó betanításával a feladatnak megfelelő adathalmaz segítségével. Sokrétegű, komplex hálók esetén gyakran nagyon időigényes a from-scratch tanítás, így ez nem minden esetben járható út.

A transfer learning nem egy triviális feladat az adathalmazok és a cél különbözősége, a nem megfelelő tanítási módszerek, vagy az adathalmazok mérete (változatossága) közti különbségek miatt.

Az egyik alapvető technika a finomhangolás (fine-tuning), ahol egy meglévő (betanított súlyokkal rendelkező) hálót tanítunk más adathalmazon – a cél megváltozása esetén a kimeneti rétegek lecserélésével. Ilyenkor előtérbe kerül a hiperparaméterek megfelelő megválasztása és a megfelelő rétegek lefagyasztása.

Egy másik módszer a tudásdesztilláció. Ebben az esetben rendelkezésünkre áll egy betanított háló, amelynek felhasználjuk a különböző rétegeiben vett kimeneti vektorokat

egy másik háló tanításához. Itt egy tanár-diák kapcsolatot alakítunk ki hálók között. Ha egyszerűsíteni akarunk egy hálót, esetleg mélyebb változásokat végzünk el az eredeti hálón, akkor ez jobb módszer lehet, mint a from-scratch tanítás, mivel felhasználhatjuk a hasonlóságot a két háló részegységei között.

1.4 Háló-optimalizálás

Megfelelő módszerek alkalmazásával optimalizálható egy háló. Ezt legegyszerűbben két teljesítménymutatóval szokták jellemezni: a futásidővel, illetve a háló futásidejű memóriahasználatával. Ezen mérőszámok mögött sok módszer állhat: csökkentjük a súlyok ábrázolásának pontosságát, törlünk kevésbé releváns összeköttetéseket, súlymátrixokat kiváltunk kisebb kiterjedésű súlymátrixokkal, de akár egy egyszerűbb háló újratanítása egy komplexebb háló kimenete alapján is tekinthető háló-optimalizálásnak.

A hálóoptimalizálás a kiválasztott módszerektől függően történhet előre betanított hálón, nem betanított hálón, vagy transfer learning segítségével egy másik hálón.

A cél az, hogy az optimalizált neurális háló közel azonos mértékben alkalmas legyen az adott feladatra, mint az eredeti háló, alacsonyabb erőforrásigény mellett.

2 Követelmények

2.1 Feladatleírás

Az alapfeladat egy neurális háló betanítása, és optimalizálása forgalomban részt vevő objektumok detektálására. A cél az, hogy a végeredmény minél alacsonyabb hardverigény mellett képes legyen elfogadható pontossággal, valós időben futni.

A háló bemenete a kamerakép perspektivikus képkockái. A háló kimenete tartalmazza a detektált objektumok osztályozását, illetve a detektálások lokalizációs adatai. A helymeghatározást befoglaló dobozok (bounding-boxok) formájában adja meg.

Az osztályozás során megkülönböztetünk objektumtípusokat:

- autó
- gyalogos
- teherautó/kisteherautó
- villamos/városi kötöttpályás jármű

A feladat első lépéseként irodalomkutatást kell végezni a követelményeknek megfelelő, annotált adathalmazok, illetve az elvárásoknak megfelelő futásteljesítményű hálóarchitektúrák körében. Ha nincs nyilvánosan elérhető, a specifikációnak már eleget tevő, betanított háló, akkor transfer learning módszerekkel, vagy teljes újratanítás segítségével készítsünk egyet.

Vagy a már betanított hálón, vagy már tanítás közben alkalmazzunk a szakirodalomban ismertetett optimalizációs technikákat, hogy csökkentsük a háló erőforrásigényét elfogadható mértékű teljesítményromlás mellett.

Az eredményeket validálni kell, illetve összesítést kell készíteni a tapasztalatokról.

3 Irodalomkutatás, előzmények

3.1 Adathalmazok

3.1.1 Megfontolások

Egy olyan hálót szeretnénk, amely alkalmas autók, gyalogosok, teherautók/kisteherautók, illetve villamosok/városi kötöttpályás járművek megkülönböztetésére. Ehhez olyan adathalmazt kell felhasználni, amely nagy mennyiségű példát tartalmaz városi környezetben felvett közlekedési szituációkról. További fontos szempont még az objektumtípusok viszonylagos, számbeli arányossága, a képek változatossága, illetve az annotáció minősége.

Mivel a célunk egy olyan objektumdetektor, amely befoglaló dobozokkal lokalizálja a objektumokat, ezért fontos, hogy minél egyszerűbben ki lehessen nyerni az annotációból az ilyen formátumú adatot.

Az általam megvizsgált adathalmazok jellemzően három különböző típusú annotációval lettek felcímkézve:

- Az első típus egy pixel-szintű annotálás, ahol a különböző objektumok vagy az azonos típusú objektumok „bitmap” képeken vannak bejelölve különböző színkódokkal.
- A második típusnál az egyedi objektumok vannak bejelölve. Ez jellemzően zárt sokszögek ponthalmazával van megadva.
- A harmadik típusnál az objektumok befoglaló dobozokkal vannak annotálva.

Általánosságban szemantikus annotációnak nevezzük azt, ahol nem az objektumpéldányok azonosítása az elsődleges cél, hanem a kép részeinek csoportosítása objektumtípusok szerint. Példány szintű annotációról akkor beszélhetünk, ha egyedi példányonként tudunk azonosítani minden egyes annotált objektumot a képeken.

A háló egyedi objektumokat fog detektálni, és kimenetében ezek befoglaló dobozainak a paramétereit adja vissza. Ezért célszerű olyan adathalmazt választani, ahol példány-szinten vannak jelölve az objektumok, lehetőleg valamilyen strukturált, szöveges formátumban (XML, JSON stb.).

3.1.2 Vizsgált adathalmazok

Mapillary Vistas Dataset [13]: Ez egy kutatási célokra ingyenesen elérhető adatbázis, amely 20 000 képet tartalmaz finom felbontású annotációval, változatos környezetben.

Cityscapes Dataset [14]: Elsődlegesen városi forgalomból tartalmaz 3500 finom felbontású, illetve 20 000 durva felbontású annotációval ellátott képet.

CityPersons [15]: A Cityscapes Dataset 5000 képén szereplő emberi forgalmi szereplők annotációját tartalmazza az adatbázis. Amiben többet nyújt az eredeti adathalmaznál az az, hogy tartalmaz minden emberre egy olyan befoglaló dobozt is, amely figyelembe veszi a kitakart részeket is. Ez pozitív hatással lehet a tanításra, mivel konzisztensebb képet ad az objektumtípus várható arányairól ([15] 3.1. bekezdés).

Apollo Scape [17]: 89 500 példány-szinten annotált képet tartalmaz. Jelenleg talán a legösszetettebb, autonóm vezetést segítő adatbázis. Hátrányaként a más kontinensbeli utcai környezet (Kína), illetve a kötöttpályás járművek hiánya vethető fel.

BDD100K [18]: 100 000 képhez tartalmaz befoglaló dobozos annotálást, a feladatra való tekintettel ez lenne a legnagyobb elemszámú adatbázis. A készítés során igyekeztek változatos fény- és időjárási viszonyok között készült felvételeket összeválogatni. Hátrányként itt is az amerikai felvételek túlsúlya, és a kötöttpályás közlekedés alulreprezentáltsága vethető fel.

KITTI [19]: 15 000 képet tartalmazó adathalmaz. Hátránya, hogy a korábban felsorolt adatbázisokhoz képest ez egy objektumszámban ritkásabb adathalmaz.

WildDash [16]: 156 képet tartalmaz különböző fajtájú képi zavarokkal (például alagút, becsillanó szélvédő, rossz időjárási körülmények stb.). Ez önmagában tanításhoz nem képez megfelelően nagy adathalmazt, de érdekes lehet a háló robosztusságának tesztelésekor.

darab	Kitti (box)	Cityscapes (pixel)	BDD100K (box)	ApolloScape (pixel)		
összes kép (x10^4)						
person vehicle	0.6 3.0	2.4 4.1	12.9 110.2	54.3 198.9		
átlag képenként				**e**	**m**	**h**
person vehicle	0,8 4,1	7 11,8	1,3 11	1,1 12,7	6,2 24	16,9 38,1

1. ábra Annotált objektumok száma és átlagos előfordulása ([17], p. 5.)

3.1.2.1 További, említésre méltó jelöltek

TME Motorway Dataset [20]: 30 000 képnyi autópályás felvétel, az autók annotálásával.

Caltech Pedestrian Dataset: 250 000 kép, 350 000 darab gyalogost befoglaló dobozzal. Külön tartalmaz annotálást a gyalogosok látható, és takarásban lévő részével együtt is.

BIT-Vehicle Dataset: 10 000 képet tartalmaz, 6 különböző típusba sorolva a járműveket (busz, mikrobusz, kisteher, szedán, SUV és tehergépjármű).

3.1.3 Osztályozás megválasztása

A cél a forgalomban részt vevő objektumok lokalizációját említi, tehát fontos döntésként jelenik meg az a probléma, hogy hogyan osztályozzuk az objektumokat.

Ha a megvalósításban binárisan osztályozunk (van vagy nincs akadály az adott helyen), akkor jelentéstartam szerint egy egyszerűbb feladatról beszélhetünk. Azonban a jelenlegi implementációban mégis az objektumok osztályozása mellett döntöttem.

Ennek egyik oka az, hogy – termékként tekintve az eredményre – a későbbiekben még felhasználható marad ez az információ. Az osztályozás extra számításigénye megengedhető mértékű. A másik oka az, hogy az osztályoknak vannak erős jellemzői (például egy gyalogos méretarányai), ami által akár még pozitív hatással is lehet az osztályozás az elért teljesítményre.

A választott osztályok:

- **on_rails:** köttöttpályás járművek (vonat, villamos)
- **car**: személygépjárművek
- **person:** gyalogos, ülő emberek stb.
- **person_group:** nehezen szétválasztható embertömeg
- **truck:** kistehergépjárművek, buszok, tehergépjárművek stb.
- **two_wheeler:** kétkerekű járművek, biciklik, motorok (vezetővel együtt)

Ezek az elérhető adathalmazok alapján lettek összeválogatva. A tanítóhalmaz összeállítása során kisebb módosításokat kellett végezni az annotáción, hogy megfeleltethető legyen a felvázolt csoportosításnak.

3.1.4 Adathalmaz megválasztása

Összetétel szempontjából az Apollo Scapes egy jó választásnak tűnik, mivel sok mintát tartalmaz, több szempontból is változatos (időjárás, környezet, járműtípusok). Továbbá a képek más adathalmazokhoz képest sok detekciót tartalmaznak. Ez az előny a gyalogosok és a kétkerekű járművek körében különösen előnyös lehet. Technikai okok miatt azonban nem ezzel az adathalmazzal lett elkezdve a tanítási folyamat. Az Apollo Scapes adathalmazhoz való hozzájutás túl sok időt vett volna igénybe, és valószínűleg hosszas lett volna az előfeldolgozás, ami a használathoz megfelelő állapotba hozta volna. Helyette más jelöltekből lett összeválogatva egy tanítókészlet.

A tanítóhalmaz összeállításakor a BDD100K, a Cityscapes (módosítva a CityPersons annotációit felhasználva), a KITTI egy kis részhalmaza és a Wild Dash lett felhasználva. Méretét tekintve a BDD100K is megfelelne tanítási célokra. A többi adathalmazra két szempont miatt volt szükség: Szerettem volna tovább növelni a minták diverzitását különböző környezetekben felvett adatokkal. Továbbá ami lényegesebb, bizonyos osztályok túlságosan alul reprezentáltak a BDD100K-ban (pl. kötöttpályás járművek, biciklisek).

A tanítás során a hiba-visszaterjesztési lépés mindig adott nagyságú bemeneti kötegek (mini-batch-ek) feldolgozása után fut le. Ezeket a kötegeket érdemes úgy összeállítani, hogy viszonylag kiegyensúlyozott arányban szerepeljenek bennük a feladat megfelelő teljesítéséhez szükséges minták. Ennek az alapelvnek megfelelően próbáltam összeállítani a mini-batch-eket.

A megvalósításban 8-as batchméretet használtam, azaz egy hiba-visszaterjesztési lépésben 8 darab kép eredményeit értékeljük ki. A tanítás során augmentációt alkalmaztam a képeken (függőleges tükrözés, árnyalat, telítettség és világosság módosítása), amitől a modell általánosítóképességének javulását várom.

A ritkásabban előforduló osztályok (on-rails, two-wheeler, truck, person/person_group) rögzített helyet kaptak a mini-batch-ekben (1-1 kép), hogy ne legyenek annyira alul reprezentáltak – ezek vegyesen lettek válogatva, több adathalmazból. Biztosítva van

továbbá az is, hogy a kötegekben mind a BDD100K (1 kép), mind a másik három adathalmaz elemei megosztva szerepeljenek (3 kép).

3.2 Hálóstruktúrák

3.2.1 Célok

Számos publikált hálóarchitektúra érhető el nyilvánosan, azonban ezek különböző erőforrásigényekkel és teljesítménymutatókkal rendelkeznek. Jelenleg minél gyorsabb futást szeretnénk közelíteni minél jobb pontosság mellett.

A vizsgálatom alapja egy olyan publikáció [22] volt, amely összefoglalást nyújt a területen használt, különböző metaarchitektúrákról és feauture-extractorokról, és megvizsgálja őket teljesítmény és erőforrásigény szempontjából.

Egy objektum detektálásra használt konvolúciós hálót általánosságban két virtuális egységre oszthatunk:

A bemeneti oldalon találhatóak a feature-extractor rétegek, amelyek azért felelősek, hogy az alacsony szintűektől kezdve az egyre magasabb szintű képi jellemzőket kinyerjük a bemenetből, amelyeket a későbbi rétegekben majd fel tudunk használni. Hatékony működéshez létfontosságú egy jó általánosítóképességű, hatékonyan tanítható feature-extractor.

A kimeneti oldalon történik a kinyert feature-ök alapján az objektumok lokalizációja (pixel-szintű szegmentálás vagy bounding-box paraméterek), illetve az objektumtípusok osztályozása. Erre különböző metaarchitektúrák léteznek, amelyek más-más elvek alapján képeznek kimenetet.

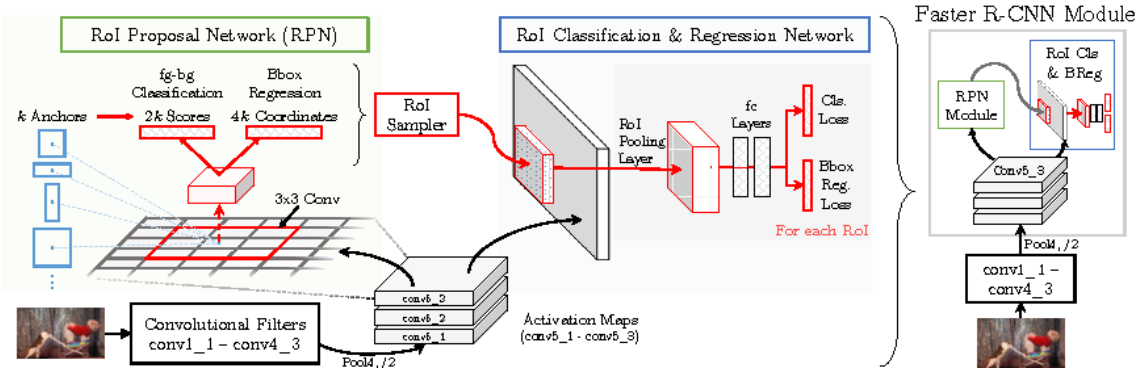
Árnyalja a képet, hogy sok háló felhasználja a feature-extractor különböző szintű rétegeiből nyert feature-öket, így nem különíthető el egyértelműen egymástól a két virtuális egység.

3.2.2 Metaarchitektúrák

3.2.2.1 Faster-RCNN (Region-based Convolutuional Neural Network) [24]

Egy Faster-RCNN háló első része konvolúciós, feature-extractor rétegekből áll. Ennek valamilyen köztes rétegéből vett feature-mapjei egy előzetes feldolgozás részeként keresztül mennek egy „region proposal network” (RPN) hálókomponensen, amelynek a

célja az, hogy becsléseket adjon arra, hogy melyek lesznek a számunkra érdekes területei a feature-mapeknek (lásd: 2. ábra bal fele).



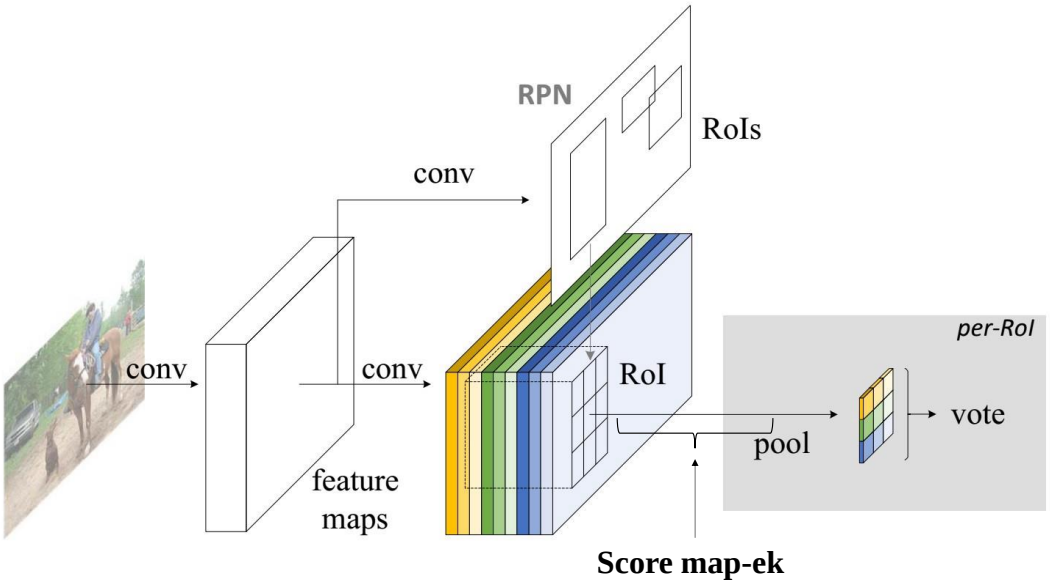
2. ábra Faster-RCNN architektúra ([34], p. 5.)

Ez a gyakorlatban azt jelenti, hogy a kép különböző régióira kifeszítünk egy mátrixot, amely meghatároz régiókat. Minden egyes régióhoz hozzárendelünk előre definiált méretű és méretarányú dobozokat (anchor-boxok). Az RPN első fele bounding-box jelölteket fog kinyerni, amelyek közül kiválasztjuk azokat, amik az anchor-boxokhoz rendelhetőek, és adunk rájuk egy osztályozástól független konfidenciaértéket (objectness confidence). Ez alapján kiválasztunk N darab legnagyobb konfidenciával rendelkező jelöltet (2. ábra, RoI Sampler), majd a feature-extractor háló kimeneti feature-mapjeiből kivágásokat képezünk (region of interest, RoI), és áttranszformáljuk a kivágásokat egy egységes formátumra (2. ábra, RoI Pooling).

Ezt követően fully-connected rétegek segítségével osztályozzuk a kimenetet, és egy pontosított becslést adunk a bounding-boxok paramétereire.

3.2.2.2 R-FCN (Region-based Fully Convolutional Network) [23]

A Faster-RCNN legnagyobb hátránya az, hogy a kivágott RoI-kon a klasszifikáció redundáns módon fut le. Az R-FCN alapkoncepciója az, hogy csökkentsük az osztályozás során fellépő redundanciát.



3. ábra R-FCN architektúra ([23], p. 3.)

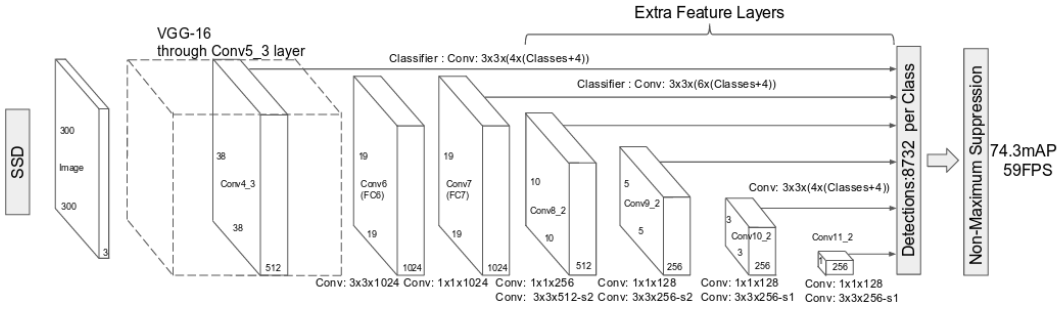
Ezt úgy éri el, hogy veszi a feature-extractor kimeneti feature-mapjeit, és kizárólag konvolúciós rétegek segítségével, párhuzamosan végzi el a RoI-k kijelölését és az osztályozáshoz szükséges előfeldolgozást az egész képen, a kizárólag a RoI-kon végzett műveletek számát pedig minimalizáljuk.

Az előfeldolgozás során tehát konvolúciós rétegek segítségével létrehozunk úgynevezett score-map-eket (3. ábra). A score-map azt az információt hordozza magában, hogy a teljes kép egy-egy régiójában milyen valószínűséggel található meg valamely osztály adott régiója. Az osztályokat is felbontjuk tehát régiókra, és egy-egy score-map egy ilyen osztályrégió detektálásáról ad információt.

Ezzel párhuzamosan egy RPN kijelöli a RoI-kat, amelyeket a score-mapekre vetítve pooling típusú szűrést végzünk el. Mivel ezzel a lépéssel már lényegében megtörtént az osztályokba sorolás és a lokalizáció is, így elhagyhatóvá válnak az Faster-RCNN RoI-nként elvégzett, (fully-connected típusú) rejtett rétegei; egyetlen hátralévő műveletként aggregálni kell a RoI-k által kijelölt, pooling után keletkezett feature-mapeket, és softmax függvénnyel, normalizált formában visszaadni az osztályokba sorolások vélt valószínűségi eloszlását.

3.2.2.3 SSD (Single-Shot Detector) [25]

Az SSD metaarchitektúra mögött az az alapgondolat áll, hogy próbáljuk meg közös lépéssorozaton keresztül megjósolni mind az objektum helyét, mind az osztályát. Tehát egyetlen lépésben végezzük el a RoI-jelöltek kiválasztását, és a klasszifikációt.



4. ábra SSD architektúra ([25], p. 4.)

Hogy ezt elérje, egy olyan kompromisszumot kötünk a kimeneti struktúrában, hogy előre meghatározunk n*n*k kimeneti régiót, ahol az n*n a képre kifeszített régiók számát, a k pedig az előre definiált, régiónként vett anchor-boxok számát jelöli. A kimenet tehát egy ilyen mátrix lesz, hozzáadva további dimenziókat (anchor-boxok eltolása, skálázása, különböző osztályok valószínűségi eloszlása).

Első lépésként itt is vesszük egy feature-extractor háló egyik köztes rétegét, mint a kiindulási feature-mapek forrása (4. ábra bal oldala). Ehhez az SSD extra konvolúciós rétegeket ad hozzá, amelyek piramis-szerű hierarchiában egyre alacsonyabb skálájú feature-öket adnak (4. ábra közepe). Vesszük a feature-piramis különböző rétegeit, és 3*3-as konvolúciókon keresztül ezek fogják adni a detektálási jelölteket (régiónként anchor-box, eltolás, osztály, konfidencia).

Utolsó fázisként a jelöltek közül kiszűrjük a relevánsnak vélt detekciókat (4. ábra, non-maximum suppression). Ez egyrészt azért fontos, mert egyetlen detekció megjelenhet több anchor-boxban is. Másrészt egy detekció megjelenhet a feature-piramis több rétegében is. Átfedő bounding-box jelöltek esetén tehát osztályonként kiválasztjuk a legnagyobb konfidencia-értékkel rendelkező jelölteket. Az ezekkel átfedésben lévő javaslatokat, illetve az alacsony konfidenciával rendelkező jelölteket pedig kiszelektáljuk a detekcióból.

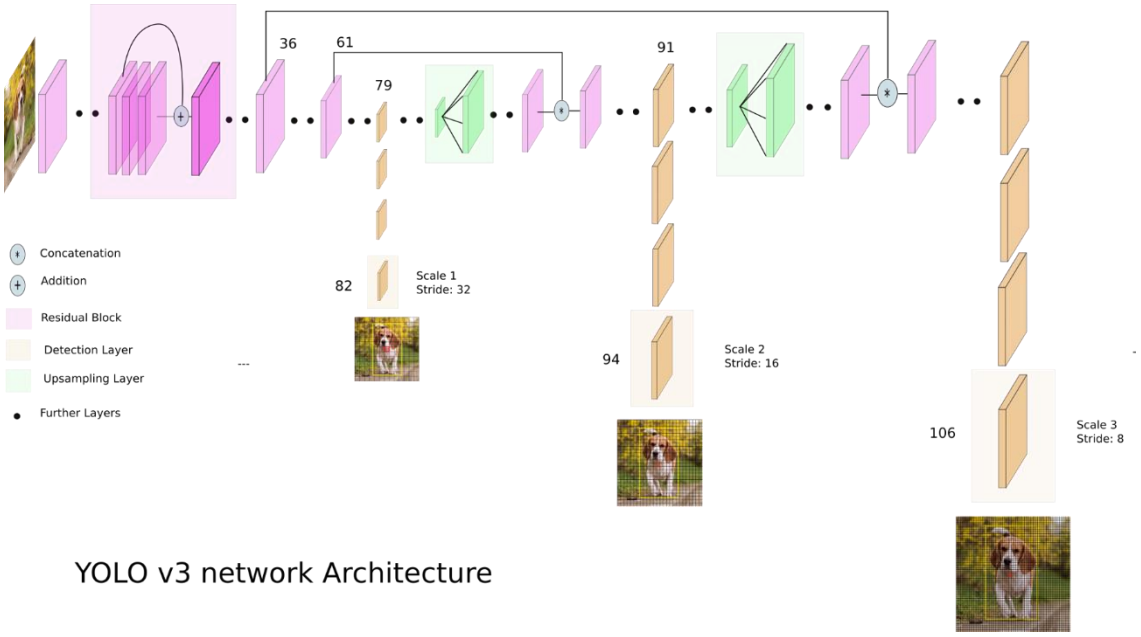
A könnyebb taníthatóság érdekében még egy szűrési módszert alkalmazunk SSD esetén, az úgy nevezett hard negative miningot. Az ok az, hogy a korábbi módszerekkel

ellentétben itt fix számú kimenetünk van, amelyek közül csak kevés fog tartalmazni valós detekciót. Ez azt eredményezheti, hogy a hiba-visszaterjesztés során túlságosan nagy lesz a negatív esetek súlya, így a modell bátrabban fog fals pozitív detekciókat adni a pozitív esetekre jutó túlságosan alacsony büntetési hányad miatt. Ezért tanítás során csak a legnagyobb konfidencia-veszteséget okozó negatív eseteket vesszük figyelembe, és arányosítjuk a számukat a pozitív esetekhez képest.

3.2.2.4 YOLO v3 [26]

Az SSD-hez hasonlóan a YOLO v3 is egyetlen lépésben végzi a RoI-k kiválasztását és a klasszifikációt, viszont van lényegi eltérés az SSD-hez képest ebben az architektúrában.

A leginkább lényegi eltérés az, hogy az SSD feature-piramisa helyett a Feature Pyramid Network (FPN) ötletét [27] vette alapul. Az SSD esetén a feature-extractor különböző felbontású rétegeiből adtunk jelölteket (osztály és bounding-box), majd egy utolsó lépésként ezeket megszűrtük a redundancia és a fals pozitívok számának csökkentése érdekében.

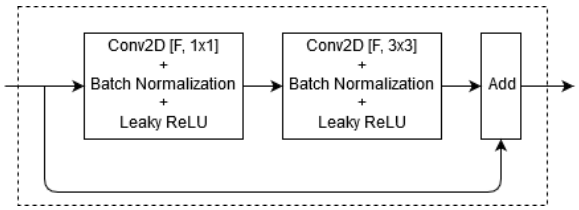


5. ábra YOLO v3 architektúra ([35])

A YOLO v3 hasonlóan működik: a feature-extractor különböző rétegeiből konvolúciós rétegek segítségével adunk jelölteket. A különbség az, hogy a legalsó (legkisebb felbontású) rétegtől kezdve felskálázás után mindig előrecsatoljuk (konkatenálással) az előző kimenet előállításához szükséges információt a következő skálájú kimenet előállításához. Ezzel képes hatékonyabban detektálni azáltal, hogy többletinformációt

kap a durvább felbontású detekciókról. Megoldást jelent például a kis méretű objektumok jobb detektálására.

A YOLO v3 is a nem-maximális értékek elnyomásának módszerével szűri a jelölteket. A klasszifikáció logisztikus regresszióval működik, így egy bounding-boxhoz akár több osztály is társulhat.



6. ábra YOLO v3 reziduális blokk

A YOLO v3 további érdekessége az, hogy egy egyedi feature-extractor struktúrát is terveztek hozzá a publikálók. A Darknet53 egy 53 konvolúciós rétegből álló háló, amely reziduális blokkokat [28] használ. Ez a megoldás kisebb egységekre osztja a hálót (6. ábra), és különböző hosszú, alternatív útvonalakat (shortcutok) hozunk létre vele a hálóban. Ez egy hatékony megoldás mélyebb hálók tanítására.

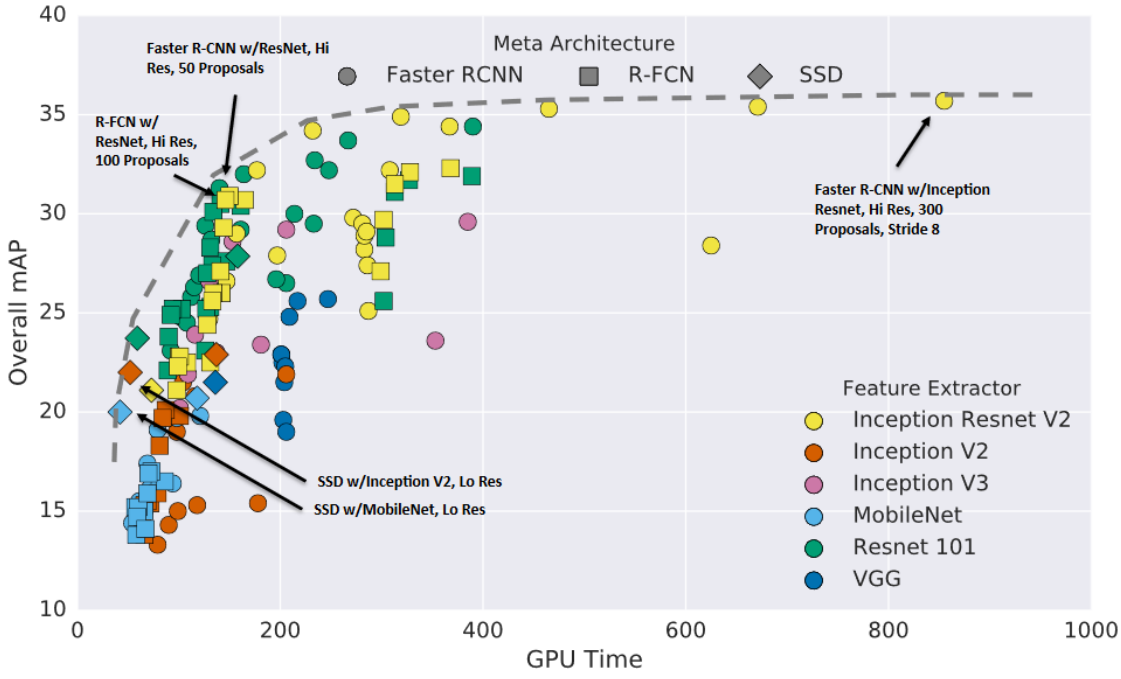
Egyrészt a shortcut csökkenti azt a hatást, hogy a bemenethez közeli rétegek felé haladva a hiba-visszaterjesztés gradiensei aránytalanul kis számok legyenek. A lánc-szabály alkalmazásával látszik, hogy a bemeneti réteg a hiba-visszaterjesztés során kap egy konstans, a reziduális blokktól független tagot. Ez azt garantálja, hogy ha a köztes rétegekre alacsony hibagradiensek is jutnak, akkor sem tűnik el a bemenetre jutó hibagradiens.

Másrészt pedig függetleníti egymástól a rétegek súlyait, ezáltal hasonló diverzifikáló hatást tudunk elérni (implicit módon), mint amit nem-reziduális hálók esetén dropout-stratégiával vagy egész rétegek elhagyásával [29] – ami a dropout stratégia rétegekre való általánosításának tekinthető. Ez annak köszönhető, hogy a shortcutok által számos, különböző mélységű összeköttetést is kialakítunk a rétegek között. Tehát összességében egy robosztusabb, könnyebben tanítható hálóstruktúrát lehet minimális extra költség segítségével (elemenkénti mátrix-összeadások) létrehozni.

3.2.3 Hálók vizsgálata

Összegezve az előző pontban vett betekintést, választani kell egy metaarchitektúrát és egy feature-extractort, amely mérsékelt memória- és számításigényű, mégis elfogadható eredményt ad a bounding-box alapú detekcióhoz.

A döntéshez egy tanulmányt [22] vettem alapul, amely részletes teljesítményelemzést végzett különböző metaarchitektúra- és feature-extractor kombinációkon.

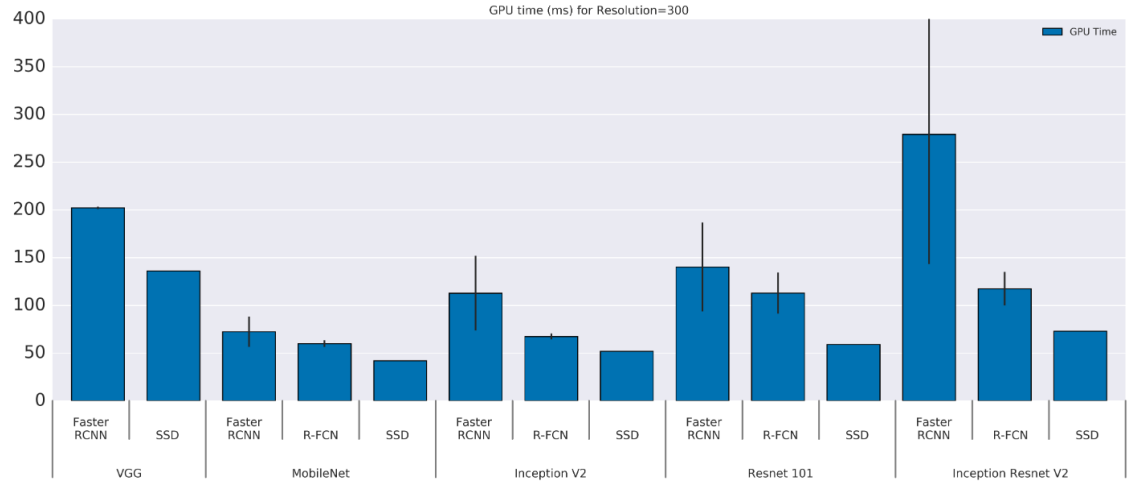


7. ábra Különböző kombinációk átlagos pontossága és átlagos futásideje ([22], p. 8.)

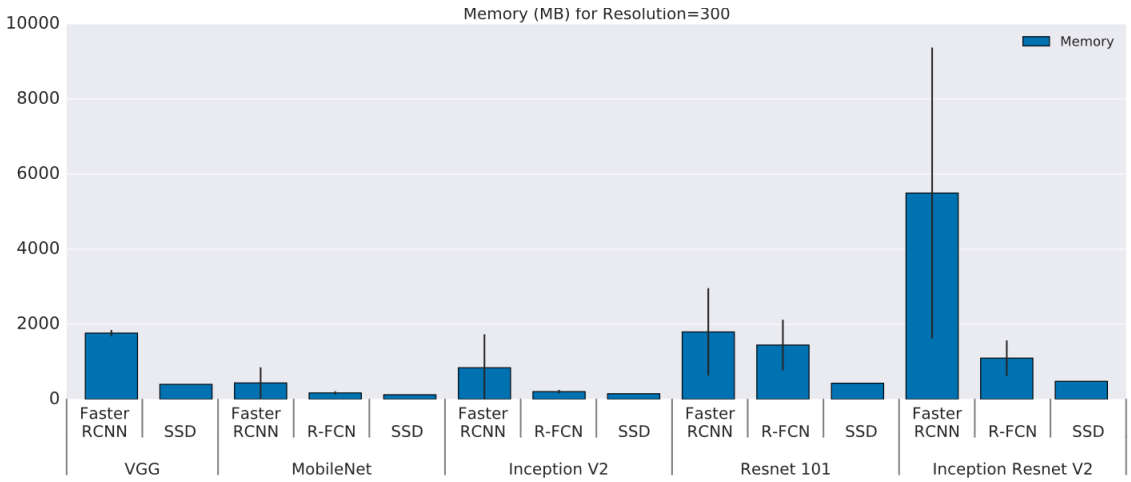
Az 7. ábra jól mutatja azt, hogy az SSD és az R-FCN hálók általánosságban gyorsabbak a Faster-RCNN alapú hálóknál. A Faster-RCNN hálók viszont pontosabb eredményt adnak, és hogy a Faster-RCNN futásideje jelentősen javítható a „jelöltek száma” szűrési

paraméter csökkentésével úgy, hogy a kimenet pontossága versenyképes maradjon az R-FCN hálókéval.

A számunkra fontosabb tanulság viszont az, hogy valósidejű felhasználáshoz (itt: 100ms alatti futásidő) első sorban SSD vagy R-FCN metaarchitektúrák használata ajánlott.

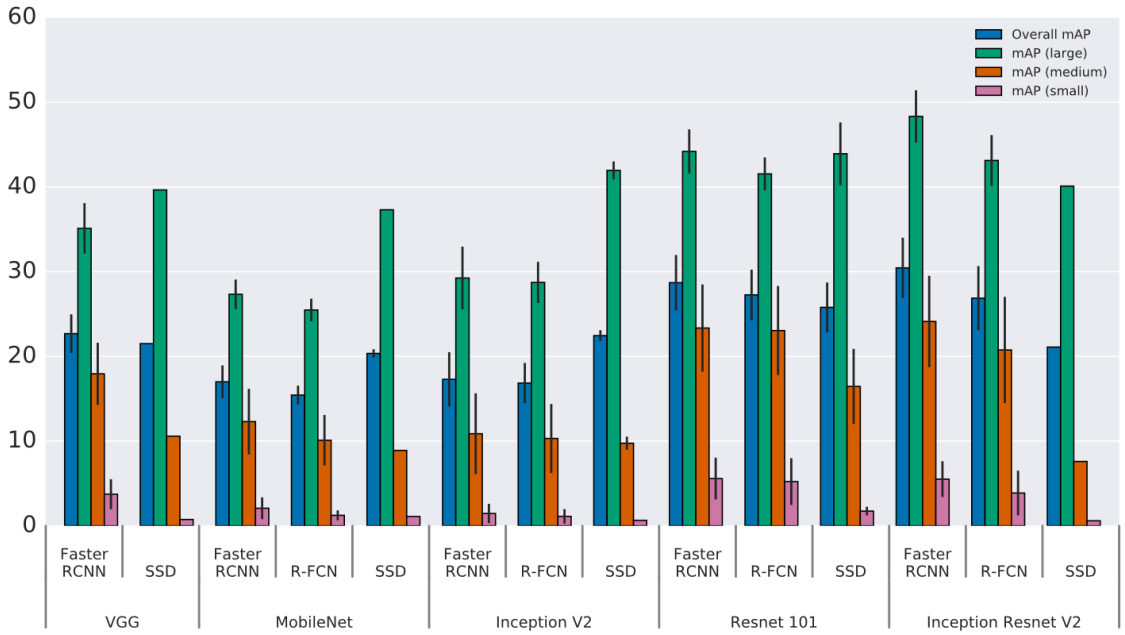


8. ábra Egységes bemeneti méret mellett mért futásidő feature-extractoronként bontva ([22], p. 11.)



9. ábra Egységes bemeneti méret mellett mért memóriaigény feature-extractoronként bontva ([22], p. 12.)

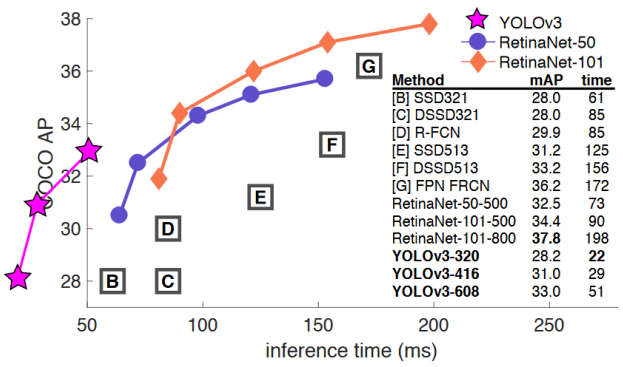
A 8. ábra és a 9. ábra összevetése alapján az látszik, hogy az erősorrend a memóriaigény esetén is megmarad, azaz nem csak a műveletek költsége, de a paraméterek száma is jelentősen eltér az egyes architektúrák között.



10. ábra A tanulmányban vizsgált hálók pontossága objektum méretek szerint bontva ([22], p. 9.)

A 10. ábra azt szemlélteti, hogyan teljesítenek a vizsgált hálók kis, közepes, illetve nagy besorolású objektumokon. Ez már egy komplexebb problémát mutat. Mind a feature-extractor hatékonysága, mind a metaarchitektúra jelentősen befolyásolja azt, hogy a különböző méretű objektumokat mennyire képes felismerni az adott háló. Általánosságban elmondható, hogy az SSD gyengébb teljesítményt nyújt kis méretű objektumok detektálása során. Ez részben az SSD kötött kimeneti struktúrájának köszönhető, részben pedig a metaarchitektúra hiányosságainak köszönhető.

A vizsgálat következő információforrása a YOLO v3 riportja [26], amely a mérési adatokat egy másik hálót - a RetinaNetet - bemutató cikkből [30] van. A RetinaNet – hasonlóan az SSD-hez és a YOLO v3-hoz – is egy lépésben hajtja végre a bounding-boxok és az osztályok becslését. Felépítésben ez is az FPN [27] alapstruktúrájára épít, mint a YOLO v3, és szintén egy nagyon jó teljesítmény-futásidő hányadosú megoldás.



11. ábra COCO Dataseten mért átlagos pontosság és futásidő ([26], p. 1.)

Az 11. ábra egy futásidő összehasonlítás a konkurens hálókkal összevetve. Látható, hogy a YOLO v3 különböző bemeneti méretek esetén nagyjából harmada akkora futási idővel tud azonos, vagy jobb pontosságot elérni, mint a fő konkurens SSD háló (ResNet-101 alappal), ami a korábbi tanulmányban a legjobb kompromisszumként jött ki futásidő szempontjából.

backbone		AP	AP50	AP75	APS	APM	APL
Two-stage methods							
Faster R-CNN+++	ResNet-101-C4	34,9	55,7	37,4	15,6	38,7	50,9
Faster R-CNN w FPN	ResNet-101-FPN	36,2	59,1	39	18,2	39	48,2
Faster R-CNN by G-RMI	Inception-ResNet-v2	34,7	55,5	36,7	13,5	38,1	52
Faster R-CNN w TDM	Inception-ResNet-v2-TDM	36,8	57,7	39,2	16,2	39,8	52,1
One-stage methods							
YOLOv2	DarkNet-19	21,6	44	19,2	5	22,4	35,5
SSD513	ResNet-101-SSD	31,2	50,4	33,3	10,2	34,5	49,8
DSSD513	ResNet-101-DSSD	33,2	53,3	35,2	13	35,4	51,1
RetinaNet	ResNet-101-FPN	39,1	59,1	42,3	21,8	42,7	50,2
RetinaNet	ResNeXt-101-FPN	40,8	61,1	44,1	24,1	44,2	51,2
YOLOv3 608x608	Darknet-53	33	57,9	34,4	18,3	35,4	41,9

12. ábra COCO Dataseten mért pontosság objektum méretek szerint bontva ([26], p. 3.)

A 12. ábra egy részletesebb betekintést nyújt a különböző vizsgált hálók pontosságába. A riportban a szerzők megjegyzik, hogy az AP50-nel jelölt mérőszám (ahol a dobozok közti átfedésre (IoU-ra) adott határérték 0.5) esetén már megközelíti a RetinaNet és a Faster R-CNN alapú hálók pontosságát. Ez implikálja azt is, hogy – hasonlóan az SSD-hez és a YOLO v2-höz – magas pontosság kevésbé várható el tőle, ezt leszámítva viszont egy potens objektum detektor jó futásidővel.

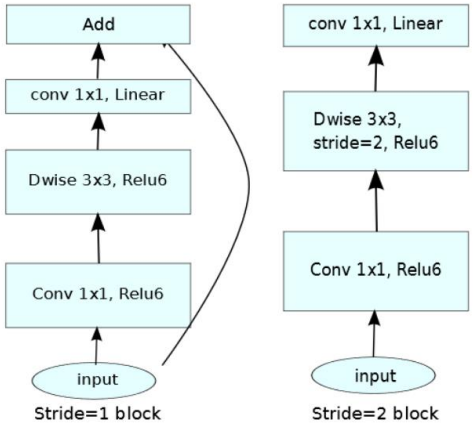
Megfigyelhető az is, hogy elődjéhez képest fejlődött a kis objektumok detektálása, amely az FPN-alapú több skálán történő detektálásnak köszönhető. Ezzel szemben a szerzői

megjegyzés és a táblázat alapján a közepes és nagy méretű objektumok detektálásán szükséges lesz javítani a struktúra egy következő iterációjában.

3.2.3.1 MobileNet v2 [31]

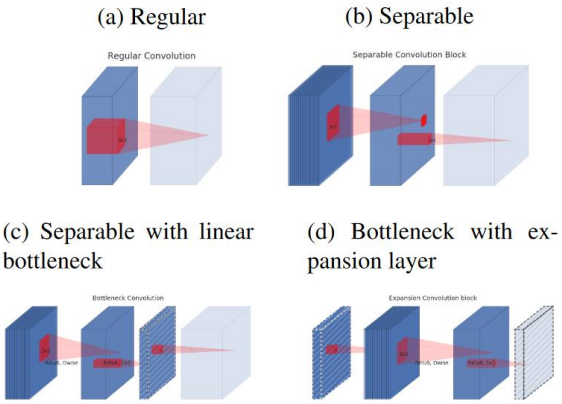
A YOLO v3 rövid bemutatásához hasonlóan itt szintén egy teljesen egyedi architektúráról van szó. A MobileNet v2 elsődleges célja az volt, hogy egy olyan hálót hozzanak létre, amely kis számítás- és memória-igényű, és akár processzoron is futtatható legyen.

A cikk invertált reziduális blokkokként nevezi a hálót felépítő modulokat (13. ábra, bal oldal). Hasonlóan a korábbiakban tárgyalt reziduális blokkokhoz, itt is összeadjuk a blokk bemenetét a blokk kimenetével.



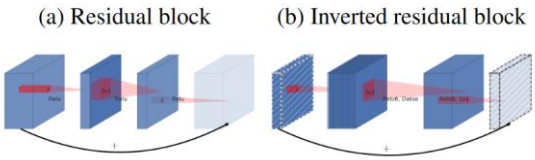
13. ábra MobileNet v2 blokkok felépítése ([31], p. 5.)

Gyakran alkalmaznak a reziduális blokkokban úgynevezett bottleneck rétegeket ([28], pp. 6-7.). Ennek lényege az, hogy 1*1-es konvolúcióval redukáljuk a csatornák számát a költségesebb konvolúciós műveletek előtt, majd a blokk végén ismét visszaállítjuk 1*1-es konvolúció segítségével a csatornák számát. Az elsődleges célja tehát az, hogy a költséges, nagyobb konvolúciós kernellel végzett műveletek költségét minimalizáljuk amellett, hogy az információtartalmat és a háló komplexitását (súlyok száma) nem redukáljuk számottevő mértékben.



14. ábra Invertált reziduális blokk összetevőinek szemléltetése ([31], p. 3.)

A MobileNet v2 invertált reziduális blokkja olyan, mintha egy reziduális blokkban a bottleneck rétegeket lecserélnénk a csatornák számát növelő *1*1*-es konvolúciókra, és a blokk kimenetének aktivációs függvényét lecserélnénk lineáris függvényre. Hogy ne növeljük a műveletvégzések költségét, a blokk köztes rétegeiben lévő konvolúciókat le kell cserélni szeparábilis konvolúciós szűrőkre [32]. Lényegében ez annyit jelent, hogy szétbontjuk a *k*k*m*-es konvolúciónkat egy csatornánként elvégzett *k*k*-s konvolúcióra, és egy normál, *1*1*-es konvolúcióra.



15. ábra Reziduális- és invertált reziduális blokk összehasonlítása ([31], p. 3.)

Mivel a blokkok határai bottleneckek, és a blokk emellett tartalmaz nem-linearitást, ezért fontos az, hogy a kimenet-bemenet közti leképzés során lehetőleg ne legyen információveszteség. ReLU esetén ez azt jelenti, hogy a nem-linearitás miatt a bemenet alapján elvárthoz képest statisztikailag csökken a nem-nulla értékek száma. Ehhez kiszámítható egy arányszám, amely megadja a kiterjesztett réteg és a bottleneck csatornáinak száma között szükséges különbséget.

Fontos még megjegyezni, hogy a cikkben a blokk kimeneteként lineáris bottlenecket használtak, hogy elkerüljék azt, hogy még tovább kelljen növelni a filterek számát a kiterjesztési rétegben.

Háló	mAP	Paraméterek (x10^6)	Műveletek (x10^9)	cpu-idő (ms)

SSD300	23,2	36,1	35,2	-
SSD512	26,8	36,1	99,5	-
YOLOv2	21,6	50,7	17,5	-
MNet V1 + SSDLite	22,2	5,1	1,3	270ms
MNet V2 + SSDLite	22,1	4,3	0,8	200ms

16. ábra MobileNet v2 összehasonlítás, objektum detekció (COCO Dataset) ([31], p. 7.)

A 16. ábra mutatja be a háló COCO Dataseten elért teljesítményét, illetve jól szemlélteti az egy nagyságrendnyi különbséget erőforrásigényben más, viszonylag alacsony erőforrásigényű hálókhoz képest. Az SSDLite metaarchitektúra az SSD-t követi, de a konvolúciós rétegek itt is cserélve lettek a korábban részletezett szeparábilis konvolúciós megoldásra (k*k csatornánként vett, majd 1*1-es konvolúció). Az eredmények alapján ez egy nem túl pontos, de pehelysúlyú háló.

Háló	OS	ASPP	MF	mIOU	Paraméterek (x10^6)	Műveletek (x10^9)
MNet V1	16	X		75,29	11,15	14,25
	8	X	X	78,56	11,15	941,9
MNet V2*	16		X	75,7	4,52	5,8
	8	X	X	78,42	4,52	387
MNet V2*	16			**75,32**	**2,11**	**2,75**
	8		X	77,33	2,11	152,6
ResNet-101	16	X		80,49	58,16	81
	8	X	X	82,7	58,16	4870,6

17. ábra MobileNet v2 összehasonlítás, szegmentálás (PASCAL VOC 2012) ([31], p. 8.)

A 17. ábra egy szegmentáló háló összehasonlítás, ahol a DeepLab v3 ([31], pp. 7-8.) metaarchitektúrát használták fel. Többek között az az érdekessége, hogy egyes rétegekben konvolúció és a méret tömörítése helyett egy speciális konvolúciót használ (atrous convolution [33]), amely rendelkezik egy extra paraméterrel, ami kitágítja a konvolúciós kernelt, ezáltal azonos számú súllyal képes nagyobb tartományban összefüggéseket keresni a feature-mapeken.

A táblázat szerint jól szerepelt ebben a feladatban is a háló.

3.2.3.2 Összegzés

Az objektumdetektálási feladatra a YOLO v3-at választottam, mint optimális kompromisszum. A témakörben olvasott szakirodalom még segíthet a háló-optimalizálási feladatomban is.

Felmerült kiegészítő feladatként, hogy a detekciók egy részére érdemes lenne utólagosan, egy szegmentáló háló által finomított eredményt is adni. Ez olyan kontextusban merült fel, hogy sávdetektálóval együtt használva fontos feladat lenne kiszűrni az olyan fals pozitív eseteket, ahol a bounding-box sarka, vagy oldala tévesen lóg át másik sávba.

Erre a részfeladatra vagy egy, a MobileNet v2 cikkben leírt hálót lehetne használni kizárólag bizonyos kivágott képeken, vagy pedig a meglévő hálót kellene kiegészíteni olyam kimeneti paraméterekkel, amelyek jelzik a bounding-box kitöltöttségét (például saroktól való távolság).

3.3 Optimalizáció

Az háló-optimalizációs lehetőségeket jól összefoglalja a [37] és a [38] tanulmány, ezek adták a kutatómunkám alapját.

3.3.1 Paraméternyesés (pruning)

A betanított modellek gyakran tartalmaznak redundáns, elhagyható kapcsolatokat. A paraméternyesés során megpróbáljuk feltérképezni ezeket kapcsolatokat, majd elhagyni a felesleges paramétereket – számottevő teljesítményromlás nélkül.

A fully connected rétegekhez képest a konvolúciós rétegek nyesése kevésbé triviális, mivel itt nem egyedülálló súlyok vannak, hanem filterenként tudjuk elvégezni a nyesést.

Első lépésként szükségünk van egy előtanított hálóra, amelyen a nyesést el tudjuk végezni. Ezután valamilyen algoritmus alapján rangsorolni kell a filtereket. Egy egyszerű megoldás lehet az, hogy a rangsoroláshoz vesszük a filtereket L1 normáját, azaz vesszük a súlymátrixok abszolút-összegeit. Az n darab legalacsonyabb rangsorolású filtert töröljük, majd egy finomhangolás és egy kiértékelés után elölről kezdhető az iteráció.

Az eredmény jelentősen javítható összetettebb rangsorolási algoritmusokkal (lásd: [39]).

3.3.2 Paraméterkvantálás (quantization)

Egy másik megközelítés lehet az, hogy kevesebb biten reprezentáljuk a súlyokat (lásd: [38], p. 23.). Gyakori megoldás a 8 bites, fixpontos, illetve a 116 bites, lebegőpontos ábrázolás, amelyhez már létezik széleskörű hardveres támogatás. Léteznek olyan kísérletek is, ahol bináris illetve ternáris számításokat alkalmaztak a súlyoknál, aktivációknál. Ez egy érdekes témakör, azonban ezt az előnyt nem feltétlen tudná a teszthardver kihasználni, továbbá a teljesítménycsökkenés is jelentős lehet.

3.3.3 Transferred konvolúciós filterek

Megfigyelések alapján különböző szimmetriák fordulnak elő a filterek között. Ezt a tulajdonságot kihasználhatjuk úgy, hogy kiválogatjuk a súlyok egy szűkebb körét, amelyeknek az egyszerű transzformáltjait használjuk fel a hálóban. Ezzel egyszerűsíthető a tanítás, és a súlyok memóriaigénye is csökkenthető. Empirikus tapasztalatok alapján ez a módszer regularizáló hatással is bír, azaz növelheti a modell általánosítóképességét ([37], pp. 4-5.).

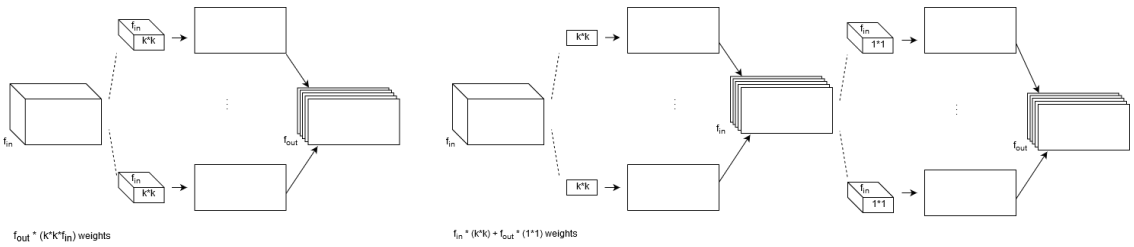
3.3.4 Konvolúciós műveletek dekompozíciója

Jelentősen csökkenthetjük a konvolúciós rétegek erőforrásigényét, ha dekomponáljuk azt több, kisebb méretű filterekkel végzett műveletre.

A low-rank factorization ([37], p. 4.) módszer lényege az, hogy egy tömörebb reprezentációba foglaljuk a lényeges paramétereket. Konvolúciós súlyok esetén ezt úgy tudjuk elérni, hogy a súlymátrixokat közelítjük több, kisebb méretű mátrix szorzataként.

Egy, hogy egy *m*n*-es mátrixot közelítünk egy *m*k*-s és egy *k*n*-es mátrix szorzatával (spatial separable convolution). Ekkor *k* egy tetszőleges szám lehet. Ha k nem túlságosan alacsony szám, akkor kis információveszteséggel közelíthető az eredeti mátrix. erre léteznek numerikus számítási módszerek.

Másik bevett módszer az, hogy egy rétegben csatornánként végezünk el egy *m*n*-es kernellel a konvolúciót, majd egy következő lépésben *1*1*-es konvolúciós kernelekkel végzünk még egy konvolúciót az összes csatornán (depthwise separable convolution) [40]. Sok csatorna esetén ez a módszer látványos csökkenéseket eredményezhet mind a paraméterszámban, mind a műveletek költségében.



18. ábra Konvolúciós réteg és depthwise separable konvolúciós réteg struktúrája

Gyakorlatban kis konvolúciós kernelekkel fogunk dolgozni, vegyük azt az esetet, amikor *3*3*-as konvolúciós kerneleket használunk. Ekkor egy hagyományos, konvolúciós réteg *3*3*f* súlyt fog tartalmazni, ahol az *f* a bemeneti csatornák (filterek) számát jelöli.

Az első módszert alkalmazva helyettesíthetjük a *3*3*-as kerneleket *1*3*-as, illetve *3*1*-es kernelekkel. Ekkor a *9*f* súlyt *6*f*-re tudjuk redukálni. Ez a módszer kis kernelméret esetén nem jelent olyan látványos javulást, mint a második módszer.

A második módszert alkalmazva drasztikusan csökkenthető az erőforrásigény (18. ábra). Ahogy az ábrán is látszik, a kimeneti csatornák számától függően nagyságrendekkel kevesebb paraméter szükséges. *3*3*-as kernel esetén nagyjából 4,5-ször kevesebb műveletet kell elvégezni a dekompozíciónak köszönhetően.

3.3.5 Tudásdesztilláció

A tudásdesztilláció során komplexebb hálókat használunk fel kisebb komplexitású hálók from-scratch tanításához ([37], pp. 5-6.). A tanuló háló esetén felhasználjuk a másik háló kimeneteit vagy akár köztes rétegekből kapott feature-mapeket is felhasználhatunk ahhoz, hogy utánozzuk a kisebb hálóval a komplexebb háló viselkedését.

3.3.6 Stochastic depth

Ez a módszer a dropout-stratégiának egy általánosítása rétegekre, célja pedig az, hogy gyorsabban lehessen elvégezni a mély hálók tanítását azáltal, hogy elhagyunk rétegeket, vagy kicseréljük őket identitásfüggvényes kapcsolatokra ([37], p. 6.). Ez analóg azzal, mintha több, különböző mélységű modellt tanítanánk be egyetlen mély háló helyett.

A dropout-stratégiához hasonlóan azt várjuk el a modelltől, hogy csökken az összeköttetések közti függőség, és nő az egyes kapcsolatok önálló egységként vett kifejezőereje ([41], p. 4.).

4 Tervezés

4.1 Környezet

4.1.1 Felhasznált hardver

Számítógép specifikációja:

- Operációs rendszer: Ubuntu 18.x 64 bit
- Processzor: Intel Xeon 1230 v2
- Memória: 2x4GB 1600MHz
- Videokártya: GeForce 1050Ti 4GB VRAM

4.1.2 Szoftveres környezet

A feladat gyakorlati megvalósításához Python 3.6 alapú futtatási környezetet használtam.

Számos, neurális háló alapú szoftverfejlesztést támogató API érhető el publikusan (például Caffe, PyTorch, Theano, TensorFlow). Ezek közül a TensorFlow-t [11] választottam Keras-szal [12]. A tensorflow egy viszonylag jól dokumentált, egyszerűen használható API Pythonhoz, amely CUDA hardveres gyorsítást is biztosít. A Keras egy wrapper API, amely TensorFlow-t hassználva valósít meg magasabb szintű, kulcsrakész funkciókat az egyszerűbb használat érdekében.

Az adatokat reprezentáló mátrixokat NumPy segítségével kezeltem és transzformáltam. Adatok szerializálásához Pickle-t használtam. A képeket OpenCV segítségével kezeltem és transzformáltam.

A TensorFlow tartalmaz egy TensorBoard nevű eszközt, amely lehetőséget nyújt adatvizualizációhoz.

4.1.3 Yolo v3 implementáció

A Yolo v3 rendelkezik saját implementációs platformmal (Darknet), azonban én nem ezt, hanem egy szabad licensz alatt álló, Kerast használó megvalósítással [21] dolgoztam. Több implementációt is megvizsgáltam, és ez egy szélesebb körben is használt, viszonylag jó kódminőségű változatnak tűnt.

A megvalósítás dokumentációjában utalnak rá, hogy teljesítményben lehet némi eltérés az eredeti implementációhoz képest. A tesztelés és a felhasználói visszajelzések igazolták ezt az eltérést, de a feladat szempontjából elfogadhatónak találtam a különbséget.

Teljesítménybeli különbségről is írnak a visszajelzésekben, ami valószínűleg az implementációs platformok közti eltéréseknek és a felhasznált képkezelési segédkönyvtárakból fakad. Mivel ismert a Yolo v3 riportjából [26] egy jól behatárolható futásteljesítmény, és mivel jelenleg csak relatív sebességnövekedés mérése a cél, így ez is egy vállalható kompromisszum a feladat kezdeti szakaszában.

A hálóhoz elérhetőek (bináris formátumban) COCO Datasetre és ImageNetre előretanított súlyok. Transfer learning technikák segítségével ezt a súlykészletet felhasználva tanítottam tovább egy saját adathalmazon.

4.1.3.1 Változtatások

Az implementáció kódját átnézve nem akadtam komolyabb hibákra. Egyedül a hiba-visszaterjesztéshez tartozó függvényben voltak a specifikációhoz képest eltérések:

```
# yolo_loss(args, anchors, num_classes, ignore_thresh=.5, print_loss=False)

obj_scale = 5
xywh_scale = 0.5

xy_loss = object_mask * box_loss_scale * K.binary_crossentropy(raw_true_xy,
raw_pred[...,0:2], from_logits=True)
xy_loss = xywh_scale * object_mask * box_loss_scale * K.square(raw_true_xy -
raw_pred[..., 0:2])

wh_loss = xywh_scale * object_mask * box_loss_scale * 0.5 * K.square(
        raw_true_wh-raw_pred[...,2:4])

confidence_loss = obj_scale * object_mask * K.binary_crossentropy(object_mask,
        raw_pred[...,4:5], from_logits=True) + (1-object_mask) * \
        K.binary_crossentropy(object_mask, raw_pred[...,4:5],
        from_logits=True) * ignore_mask

class_loss = object_mask * K.binary_crossentropy(true_class_probs,
        raw_pred[...,5:], from_logits=True)
```

A bounding-box paramétereinek és az 'objectness' konfidenciaértékeinek a hibatagja meg lett szorozva 1-1 skaláris szorzóval. Továbbá az xy_loss tagban ki lett cserélve a bináris keresztentrópia-alapú hibaszámítás hibanégyzetes összegre.

A megvalósítás Github oldalán javasolták a Pythonos képkezelésre használt Pillow könyvtár eljárásainak az OpenCV-s megfelelőire cserélését. Ezt a módosítást elvégeztem.

4.2 Előkészületek az adathalmazon

Az adathalmazokat le kellett tölteni, majd közös (JSON-alapú) formátumra hozni. Rövid tervezés után elkészítettem egy megfeleltetési táblázatot, ami alapján kijelöltem egy minimális számú közös osztályt a használandó adathalmazhoz. Ahol nem volt egyértelmű

a megfeleltetés, ott vagy kihagytam osztályokat, vagy átalakítottam az annotációt (például, hogy ne legyen külön annotálva egy motor és a vezetője).

A CityScapes adatbázis emberi annotációit lecseréltem a CityPersonsére, mivel jobb minőségű, illetve tartalmaz bounding-boxot a takarásban lévő részre is.

Készítettem a ritkábban előforduló osztályokhoz külön annotációs fájlokat.

A képeket egységes méretarányra hoztam levágással és skálázással, majd ehhez igazítottam az annotációt is.

4.3 Tanítás

4.3.1 Tanítás: részletes előkészület

A tanítás mini-batchekkel történt. Úgy próbáltam kialakítani ezeket, hogy kiegyensúlyozottabbak legyenek. Az adathalmaok közül a BDD100K, a CityScapes + CityPersons, a KITTI, illetve a WildDash lett felhasználva. A mini-batchek 8-as egységekben vannak összeállítva. Ebből 1-1 rögzített helye van az „on-rails", a „two-wheeler", a „truck", illetve a person/person_group" objektumokat tartalmazó képeknek. A maradék négy helyre pedig véletlenszerűen készít elő képeket a három kisebb, illetve a BDD100K adathalmazból.

A validáció a BDD100K validációs célra elkülönített részén történik.

A tanítás jellegétől függően vagy a teljes háló volt tanítva, vagy kizárólag a felső rétegek, majd finomhangolásként alacsony tanulási tényezővel a háló többi része.

4.3.1.1 Bounding-boxok szűrése

Az eredményként kapott bounding-box jelöltek határértékekkel és nem-maximális értékek elnyomása módszerrel vannak szűrve. Ennek során a Score metrikát használjuk, amelyet a detekció konfidencia-értékei határozzák meg. Az *objectness confidence* osztály-független módon ad becslést a detekció helyességére, a *class confidence* pedig azt becsli meg, hogy a detektálás melyik osztályba tartozik. Első lépésként egy Score-határérték alatt kidobunk minden jelöltet.

$$threshold_{Score} < confidence_{objectness}(pred_i) * confidence_{class}(pred_i)$$

A kapott részhalmazt a Score értékek alapján rendezzük csökkenő sorrendben. Második lépésként mohó módszerrel beválogatja az algoritmus az elfogadott bounding-boxokat

úgy, hogy ne legyen egy határérték feletti átfedés (intersection over union, IoU) a halmazon belül.

$$IoU(pred1, pred2\ ) = \frac{A_{intersection}}{A_{union}}$$

A következő függvényt használtam a tanítás során:

$$loss_{xy} = scale_{xywh} * \sum_{\substack{\forall\,(gt_{xy}, pred_{xy}) \\ \in obj}} (gt_{xy} - pred_{xy})^2$$

$$loss_{wh} = scale_{xywh} * \sum_{\substack{\forall\,(gt_{wh}, pred_{wh}) \\ \in obj}} (gt_{wh} - pred_{wh})^2$$

$$binary\ crossentropy(p, q) = -gt_{conf} * \log(sigmoid(pred_{conf})) - (1 - gt_{conf}) * \log(sigmoid(1 - pred_{conf}))$$

$$loss_{conf} = scale_{noobj} * \sum_{\substack{\forall\,(gt_{conf}, pred_{conf}) \\ \in noobj}} (binary\ crossentropy(gt_{conf}, pred_{conf})) + \sum_{\substack{\forall\,(gt_{conf}, pred_{conf}) \\ \in obj}} ((binary\ crossentropy(gt_{conf}, pred_{conf})))$$

$$loss_{class\ conf} = \sum_{\substack{\forall\,(gt_{conf}, pred_{conf}) \\ \in obj}} (binary\ crossentropy(gt_{class\ conf}, pred_{class\ conf})))$$

$$loss = \frac{loss_{xy} + loss_{wh} + loss_{conf} + loss_{class\ conf}}{batch\ size}$$

Hogy hatékonyabban felhasználja a háló azt a tulajdonságát, hogy három különböző skálájú kimenete van, így készítettem egy olyan függvényt, amely a fals pozitív detekciókat vizsgálja az alapján, hogy a doboz méretei alapján várhatóan melyik kimeneti rétegben kéne megjelenniük.

$$\hat{A} = pred_w * pred_h$$

$$f = \frac{log(\hat{A}) - log(A_{anchor_{2,0}})}{log(A_{anchor_{0,2}}) - log(A_{anchor_{2,0}})} * 2$$

$$for\ layer\ idx = 0..2: loss_{outp_{layer\ idx}} = \sum_{\substack{\forall pred \\ \in preds^{false\ positive}_{layer\ idx}}} |f - layer\ idx| * c$$

$$pred \in preds^{false\ positive}_{layer\ idx} \leftrightarrow IoU(pred, gt) < threshold_{IoU}$$

Ezt a függvényt sikeresen implementáltam.

4.4 Hálók kiértékelése

A kiértékelés során fontos mérőszám lehet a paraméterek száma, az átlagos futási idő, az átlagos pontosság, illetve az átlagos visszahívás. Érdemes lehet mintavételezetten tesztképeket gyűjteni a kimenetről.

A validációs halmaz ugyanazon 10 000 képén folyt a kiértékelés.

Az alábbi mutatókat használtam fel:

1 Paraméterek száma [db]

2 Átlagos kiértékelési idő képenként [ms]

3 Átlagos pontosság (mean average precision):

$$mAP = \Sigma_{images} \frac{true\ positives}{true\ positives\ +\ false\ positives} \Big/ n_{images}$$

4 Átlagos felidézés (mean average recall):

$$mAR = \Sigma_{images} \frac{true\ positives}{true\ positives + false\ negatives} \Big/ n_{images}$$

A paraméterek száma és az átlagos kiértékelési idő a háló erőforrásigényére ad betekintést.

Az átlagos pontosság azt mutatja meg, hogy a detektálások hánya része volt helyes. Akkor tekintünk helyesnek egy detektálást, ha létezik olyan azonos osztályba tartozó ground truth, amellyel egy határértéket meghaladó IoU típusú átfedése van.

Az átlagos felidézés azt mutatja meg, hogy a ground truth adatok hányad részét sikerült detektálni. Itt is az előző feltételek adottak.

4.5 Kísérletek

4.5.1 Tanítás

A YOLO v3 felépítése megtalálható a függelékben.

A tanítás során augmentáltam a képhalmazt (függőleges tükrözés, árnyalat, telítettség és világosság módosításával), amitől a modell általánosítóképességének javulását várom. A képek előzetesen egységes képarányra lettek vágva, a tanítás során 1:1-es képarányba lettek kinyújtva a bemeneti képek, egységesen 608*608 pixeles méretben.

Egy COCO Dataseten betanított, nyilvánosan elérhető súlykészletből indultam ki. Első lépésben a saját adathalmazomon a hálót. A COCO Datasethez képest 6 osztályra redukálódott a kimenet. A három kimeneti rétegen kívül a háló többi része be volt fagyasztva, hogy először a megváltozott kimeneti formátumhoz legyen hangolva a modell. A COCO Dataset is tartalmaz járműveket és embereket, így már ezzel a lépéssel is szemléletes eredményt lehetett elérni.

A következő konfigurációt használtam:

Adam optimizer, 10^-3 tanulási tényező, 8-as batch méret, augmentáció, 800*50 batch.

A következő lépésben a kapott súlyokat finomhangoltam alacsonyabb tanulási tényezővel és korai leállási (early stopping) feltétellel. amely leállítja a tanítási iterációt, ha tartósan (*n* epochon át) nem változik a validációs halmazon számított hiba.

Az egész hálót csak 1-es mini-batch mérettel tudtam volna tanítani, ami túlságosan lassú lett volna, így inkább befagyasztottam a hálónak az első felét, azaz az összes reziduális blokkot. Ezzel nagyjából a súlyok harmada maradt tanítható, viszont megmaradt a 8-as mini-batch méret.

mean_avg_precision: 0.730790157199792

mean_avg_recall: 0.36599476132287084

avg_duration: 0.17160535294556176 ms

Pár véletlenszerűen válogatott példakimenetek a függelékben megtalálható.

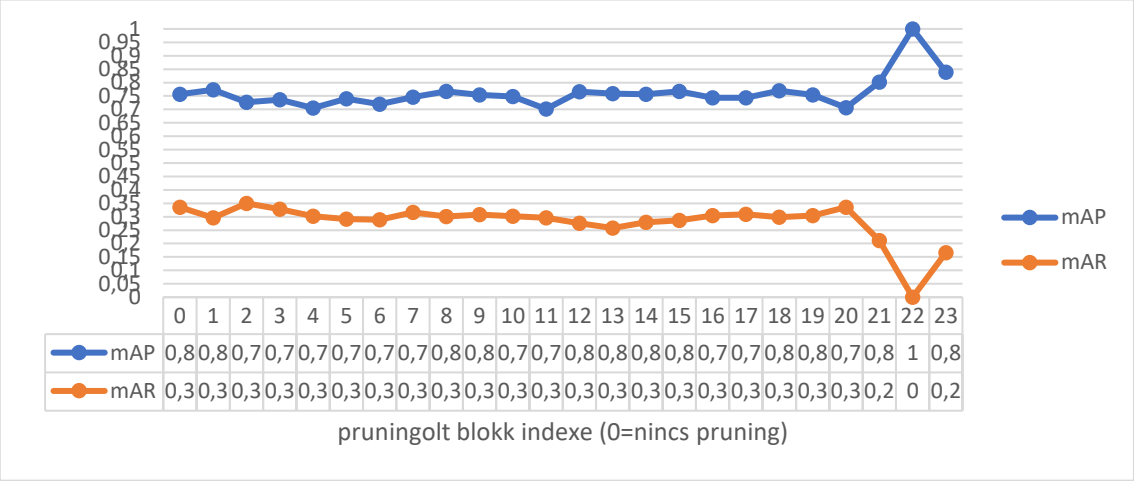
A tanulási görbékből azt a következtetést vontam le, hogy valami probléma lépett elő. Mind a tanító-, mind a validációs halmazon ugráltak a hibaértékek, és az eredmény is romlott az első lépésben kapott modellhez képest. Ezt okozhatta valamilyen egyszerűbb hiba a kódban vagy a megvalósításban, az augmentáció, a nem megfelelő tanulás tényező, vagy nem futott kellő ideig a tanítás.

4.5.2 Reziduális blokkok pruningja

A korábban már említett reziduális blokkokkal kapcsolatban találtam egy érdekes tanulmányt, amely a reziduális hálókkal és a shortcutok szerepével foglalkozik. Ebben szerepelt néhány teszt, amely ezeknek a robosztusságát vizsgálja ([42], pp. 5-6.). A teszt konklúziója szerint a reziduális hálókban az alternatív útvonalak miatt kevésbé függenek

egymástól a rétegek, így elhagyhatóak akár egész rétegek is a háló teljesítményének drasztikus romlása nélkül.

Én az egyszerűség kedvéért csak azt vizsgáltam meg, hogy az egy-egy reziduális blokkot elhagyva mekkora változás figyelhető meg a teljesítményben. A teszt során a blokkok utolsó konvolúciós rétegének a súlyait nullára inicializáltam, illetve befagyasztottam ezeknek a taníthatóságát. Ez a gyakorlatban megfelel annak, mintha törölném a blokkokat alkotó rétegeket a modellből.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
mAP	0,8	0,8	0,7	0,7	0,7	0,7	0,7	0,7	0,8	0,8	0,7	0,7	0,8	0,8	0,8	0,8	0,7	0,7	0,8	0,8	0,7	0,8	1	0,8
mAR	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,3	0,2	0	0,2

Ahogy látható az adatsoron, az utolsó három blokk kivételén kívül elmondható, hogy mérsékelt befolyással volt a hálóra a beavatkozás.

4.5.3 Háló módosítása invertált reziduális blokkokkal

A MobileNet v2 cikkjében [31] szerepelt az invertált reziduális blokként megnevezett struktúra. A tanulmány szerint kisebb komplexitás mellett képes elérni a normál reziduális blokkokhoz hasonló eredményt.

Ez a kísérlet tehát arról szól, hogy megfeleltetjük egymásnak a kétfajta blokkot, és egyenként lecserélve újratanítjuk a hálót.

Az egyik megközelítés az implementációban az volt, hogy az első blokktól kezdve le kell cserélni egyenként a blokkokat, betölteni az előző iteráció súlykészletét, majd minden más réteget be kell fagyasztani a tanítási ciklus során.

A másik megközelítés során készítünk egy modellt, amelynél a közös bemenetről ágaznak el az eredeti- és a módosított háló „csonkjai” – azaz a háló elvágva az éppen tanítandó blokknál. A két kimeneti réteget azonos méretűre hozzuk egy 1*1-es konvolúció

segítségével, majd hibát számolunk a két különböző kimenet összehasonlításából (például az abszolút különbségek összegeként).

Ez a megközelítés közelebb áll a tudásdesztilláció alapgondolatához, mivel a továbbiakban azt várjuk el, hogy adott bemenetre a módosított háló adott blokkja az eredeti háló megfeleltetett blokkjához hasonló kimenetet adjon.

Ezzel a paraméterek száma körülbelül 2/3-szorosa lesz az eredetihez képest.

Az első megközelítés implementációja működőképes állapotban van ehhez a teszthez, viszont egyelőre nem sikerült értékelhető eredményt előállítani.

4.5.4 Filter-szintű pruning

Vesszük a filterek L1 normáját (súlyok abszolútértékének összege), majd ez alapján fordított rangsort állítunk. Iteráció során mindig kiválasztjuk az első *n* darabot ebből a rangsorból, amíg elfogadhatónak találjuk a hiba változását.

Ehhez jelenleg még csak félkész állapotban van megvalósítás.

5 További lehetőségek

5.1 Eddigi próbálkozások befejezése

Az idő hiánya és a különböző implementációs nehézséges miatt nem tekinthetőek befejezettnek a kísérleti eredmények. Ezeket mindenképpen érdemes lenne folytatni, befejezni a feladat szempontjából.

5.2 Paraméter-megosztás, transferred konvolúciós filterek

A Korábban megemlített módszert is érdemes mélyebben megvizsgálni.

5.3 Pruning finomítása

A pruning tesztek esetén érdemes finomhangolási lépéseket beiktatni, hogy jobb eredményeket érjünk el a megváltozott hálóstruktúrával. A pruning-jelöltek kiválasztásán is lehet finomítani (lásd: [39]).

5.4 16 bites lebegőpontos számok használata

A keretrendszer lehetőséget arra, hogy az alapértelmezett, 32 bites lebegőpontos számtípus helyett 16 biteset használjunk. Ezt a módosítást a jelenlegi hardver (1050Ti videokártya) nem tudja kihasználni, viszont az újabb videokártyák (például Nvidia RTX 2xxx-es sorozat) már igen. A memóriaigény a hardver kialakítása miatt (32 bites regisztерméret) nem csökken, azonban a számítások gyorsasága közel kétszeresére is növelhető a specifikáció alapján [36]. (Ez a módosítás Keras esetén egy konfigurációs paraméter megadása, de elképzelhető, hogy további változtatásokra is szükség lehet.)

5.5 Bemeneti méret hatásának vizsgálata

Jelenleg 608*608-as képeket kap a háló. A YOLO v3 tanulmány eredményei alapján kisebb képmérettel akár felére is csökkenthető a futási idő. Emiatt érdemes lehet bővebben is tesztelni az optimális bemeneti méretet.

Irodalomjegyzék

[1] Jools, Ipon.hu: „Mennyire önállóak az önvezető autók?”, 2016.07.10.; https://ipon.hu/magazin/cikk/mennyire-onalloak-az-onvezeto-autok. [2018. 11. 22.]

[2] Wikipedia contributors: „List of self-driving car fatalities”, Wikipedia, 2018.11.03.; https://en.wikipedia.org/w/index.php?title=List_of_self-driving_car_fatalities&oldid=867022821. [2018. 11. 22.]

[3] Hlács Ferenc, Hwsw.hu: „Hetek óta üres vezetőüléssel cirkálnak a Waymo önvezető autói”, 2017.11.08.; https://www.hwsw.hu/hirek/58024/waymo-onvezeto-auto-chrysler-arizona-phoenix.html. [2018. 11. 22.]

[4] Hlács Ferenc, Hwsw.hu: „Az FCA is beszáll a BMW-Intel-Mobileye önvezető autós projektjébe”, 2017.08.16.; https://www.hwsw.hu/hirek/57663/chrysler-fiat-onvezeto-auto-intel-bmw-mobileye.html. [2018. 11. 22.]

[5] Habók Lilla, Hwsw.hu: „38 millió dolláros befektetést kapott az AImotive”, 2018.01.05.; https://www.hwsw.hu/hirek/58274/aimotive-onvezeto-auto-startup-befektetes-tamogatas.html. [2018. 11. 22.]

[6] Papp Tibor, Totalcar.hu: „Íme a magyar fejlesztésű, önmagát vezető kamion”, 2018.09.27.; https://totalcar.hu/magazin/technika/2018/09/27/ime_a_magyar_fejlesztesu_onmagat_vezeto_kamion. [2018. 11. 22.]

[7] Abu85, Prohardver.hu: „Több DRIVE platform alapja lesz az NVIDIA Xavier SoC”, 2018.01.11.; https://prohardver.hu/hir/nvidia_xavier_tobb_drive_platform_alapja_lesz.html. [2018. 11. 22.]

[8] Abu85, Prohardver.hu: „Betekintés az NVIDIA Volta architektúra képességeibe”, 2017.05.12.; https://prohardver.hu/teszt/nvidia_volta_architektura_gv100_tesla_v100/gepi_tanulas_mindenek_felett.html. [2018. 11. 22.]

[9] Asztalos Olivér, Hwsw.hu: „Jövőre debütálhat a Tesla saját fejlesztésű AI-gyorsítója”, 2018.08.06.; https://www.hwsw.hu/hirek/59202/tesla-ai-gyorsito-chip-processzor-platform-nvidia-drive-px-onvezeto-auto.html. [2018. 11. 22.]

[10] Wikipedia contributors:ImageNet, ImageNet Challenge, Wikipedia, 2018.10.24.; https://en.wikipedia.org/w/index.php?title=ImageNet&oldid=865534026#ImageNet_Challenge. [2018. 11. 22.]

[11] Tensorflow https://www.tensorflow.org. [2018. 11. 22.]

[12] Documentation: Keras: The Python Deep Learning library; https://keras.io. [2018. 11. 22.]

[13] Gerhard Neuhold, Tobias Ollmann, Samuel Rota Bulò, Peter Kontschieder: „The Mapillary Vistas Dataset for Semantic Understanding of Street Scenes", 2017; DOI: 10.1109/ICCV.2017.534

[14] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth: "The Cityscapes Dataset for Semantic Urban Scene Understanding", 2016; arXiv:1604.01685.

[15] Shanshan Zhang, Rodrigo Benenson: "CityPersons: A Diverse Dataset for Pedestrian Detection", 2017; arXiv:1702.05693.

[16] Oliver Zendel, Katrin Honauer, Markus Murschitz, Daniel Steininger, Gustavo Fernandez Dominguez: „WildDash - Creating Hazard-Aware Benchmarks", The European Conference on Computer Vision (ECCV), 2018, pp. 402-416

[17] Xinyu Huang, Peng Wang, Xinjing Cheng, Dingfu Zhou, Qichuan Geng: "The ApolloScape Open Dataset for Autonomous Driving and its Application", 2018; arXiv:1803.06184.

[18] Fisher Yu, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, Vashisht Madhavan: "BDD100K: A Diverse Driving Video Database with Scalable Annotation Tooling", 2018; arXiv:1805.04687.

[19] Andreas Geiger, Philip Lenz, Raquel Urtasun: „Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite", 2012; DOI: 10.1109/CVPR.2012.6248074 http://www.cvlibs.net/publications/Geiger2012CVPR.pdf

[20] Claudio Caraffi, Tomáš Vojíř, Jiří Trefný, Jan Šochman, Jiří Matas: „A System for Real-time Detection and Tracking of Vehicles from a Single Car-mounted Camera", 2012; DOI: 10.1109/ITSC.2012.6338748

[21] Github contributors: „A Keras implementation of YOLOv3 (Tensorflow backend)", commit 2018.07.31. e6598d1, qqwweee/keras-yolo3

[22] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama: "Speed/accuracy trade-offs for modern convolutional object detectors", 2016; arXiv:1611.10012.

[23] Jifeng Dai, Yi Li, Kaiming He: "R-FCN: Object Detection via Region-based Fully Convolutional Networks", 2016; arXiv:1605.06409.

[24] Shaoqing Ren, Kaiming He, Ross Girshick: "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", 2015; arXiv:1506.01497.

[25] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu: "SSD: Single Shot MultiBox Detector", 2015; arXiv:1512.02325.

[26] Joseph Redmon: "YOLOv3: An Incremental Improvement", 2018; arXiv:1804.02767.

[27] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan: "Feature Pyramid Networks for Object Detection", 2016; arXiv:1612.03144.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren: "Deep Residual Learning for Image Recognition", 2015; arXiv:1512.03385.

[29] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra: "Deep Networks with Stochastic Depth", 2016; arXiv:1603.09382.

[30] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He: "Focal Loss for Dense Object Detection", 2017; arXiv:1708.02002.

[31] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov: "MobileNetV2: Inverted Residuals and Linear Bottlenecks", 2018; arXiv:1801.04381.

[32] Yusuke Uchida: „Why MobileNet and Its Variants (e.g. ShuffleNet) Are Fast", 2018.04.23.; https://medium.com/@yu4u/why-mobilenet-and-its-variants-e-g-shufflenet-are-fast-1c7048b9618d

[33] Liang-Chieh Chen, George Papandreou, Florian Schroff: "Rethinking Atrous Convolution for Semantic Image Segmentation", 2017; arXiv:1706.05587.

[34] Shrivastava, Abhinav and Abhinav Gupta: „Contextual Priming and Feedback for Faster R-CNN.", ECCV, 2016; DOI:10.1007/978-3-319-46448-0_20.

[35] Ayoosh Kathuria, Towardsdatascience.com, „What's new in YOLO v3?", 2018.04.13., https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b. [2018. 11. 22.]

[36] Wikipedia contributors: „List of self-driving car fatalities", Wikipedia, 2018.12.05.;https://en.wikipedia.org/w/index.php?title=GeForce_20_series&oldid=872098575#Chipset_table. [2018. 12. 05.]

[37] Yu Cheng and Duo Wang and Pan Zhou and Tao Zhang: „A Survey of Model Compression and Acceleration for Deep Neural Networks", 2017; arXiv:1710.09282.

[38] Vivienne Sze and Yu-Hsin Chen and Tien-Ju Yang and Joel Emer: „Efficient Processing of Deep Neural Networks: A Tutorial and Survey", 2017; arXiv:1703.09039.

[39] Jacob Gildenblat: „Pruning deep neural networks to make them fast and smal",2017; https://jacobgil.github.io/deeplearning/pruning-deep-learning. [2018.12.05.]

[40] Atul Pandey, Medium.com: „Depth-wise Convolution and Depth-wise Separable Convolution", 2018; https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec. [2018.12.05.]

[41] Gao Huang and Yu Sun and Zhuang Liu and Daniel Sedra and Kilian Weinberger: „Deep Networks with Stochastic Depth", 2016; arXiv:1603.09382.

[42] Andreas Veit and Michael Wilber and Serge Belongie: „Residual Networks Behave Like Ensembles of Relatively Shallow Networks", 2016; arXiv:1605.06431.

Köszönetnyilvánítás

Abban, hogy eljutottam idáig, bizonyára szerepet játszik a Jóisten, a szerencse, és a MIT-tanszék. Továbbá szeretnék megköszönni mindent annak a lánynak, aki ezt olvassa.

Függelék

A szakdolgozathoz tartozó programkód megtalálható az alábbi linken:

https://github.com/boti996/szdoga

Yolo v3 felépítése:

Darknet-53:

Type	Filters	Size / Stride
Conv2D*	32	3*3
Conv2D*	64	3*3 / 2
1x Residual block	64	
Conv2D*	128	3*3 / 2
2x Residual	128	
Conv2D*	256	3*3 / 2
8x Residual block	256	
[out1]		
Conv2D*	512	3*3 / 2
8x Residual block	512	
[out2]		
Conv2D*	1024	3*3 / 2
4x Residual block	1024	

Residual block:

Type	Filters	Size / Stride
[inp]		
Conv2D*	f	1*1
Conv2D*	2*f	3*3
Add, [inp]		

Last layers<n>:

Type	Filters	Size / Stride
Conv2D*	f	1*1
Conv2D*	2*f	3*3
Conv2D*	f	1*1
Conv2D*	2*f	3*3
Conv2D*	f	1*1
[x<n>]		
Conv2D*	2*f	3*3
Conv2D	out_filters	1*1
Output [y<n>]		

Yolo body:

Type	Out
Darknet-53	
Last layers 3	-> Output [y3]
Conv2D* [x3]	
Upsampling2D, 2x	
Concatenate, [out2]	
Last layers 2	-> Output [y2]
Conv2D*, [x2]	
Upsampling2D, 2x	
Concatenate, [out1]	
Last layers 1	-> Output [y1]

*Conv2D + Batch Normalization + Leaky ReLU

Néhány kimeneti eredmény: