Önálló laboratórium

**Gépi tanulás alapú objektum detektálás autonóm beltéri jármű platformon**

Karz Gergely

horizontal line

# Placeholder image

# Téma leírása

A feladat egy, a tanszéken futó kutatási projekthez kapcsolódik, amelyben egy intelligens autó tesztplatform létrehozása a cél. A platform alapja egy csökkentett méretű (kb. 1:3 méretarányú) autómodell, melyet egy távirányítható játékautóból alakítunk át. A tesztplatform létrehozásának célja különböző navigációs algoritmusok fejlesztésének és valós környezetben történő tesztelésének elősegítése.

A sikeres navigáció és az ütközések elkerülése érdekében fontos, hogy a jármű valós időben felismerje az ütközés veszélyt jelentő objektumokat a környezetében, hogy megfelelően tudjon reagálni rájuk. Jelen feladat keretében az autóra szerelt kamera segítségével kell megvalósítani objektumfelismerést gépi tanulás módszer(ek) alkalmazásával.[[1]](#footnote-0)

## Rendszer felépítése

### Tanításhoz használt felépítés

A feladat megoldásához, egy mély neuron hálót használtam. A mély neuronhálók tulajdonsága, hogy jelentős számítási kapacitást igényel a tanításuk, valamint sok tanító adatra van szükség a jó minőségű háló előállításához. A számítási kapacitást a Google Colaboratory szolgáltatással biztosítottam, viszont mivel ide egyenesen feltölteni a tanító adatokat lassú volt, így a saját Google Drive-omra töltöttem fel először, ahonnan sokkal gyorsabban át lehetett őket másolni a Colab virtuális gépére. Ez a megoldás egy újabb problémát vet fel, ugyanis a Google Drive mérete erősen véges (15GB) a tanító adatok mérete pedig ideális esetben ennél sokkal nagyobb (COCO adatbázis: 80GB+[[2]](#footnote-1)). Ezt a problémát, a YOLO honlapján található[[3]](#footnote-2), COCO adatbázison előtanított súlyok felhasználásával oldottam meg. Ez viszont azt jelenti, hogy ebben a rendszerben új háló struktúrát nem tudunk úgy készíteni, hogy annak jobb minősége legyen, mint az előtanított súlyok felhasználásával. A forráskód githubon van[[4]](#footnote-3), ahonnan collabra könnyen lehetett klónozni, viszont az implementáción változtatni nehéz volt, ugyanis minden egyes változtatás után újra kellett pusholni, és klónozni. A Colab további problémája, hogy a tanítás során keletkezett kimeneti log a böngészőben külön soronként kerül megjelenítésre, ami azt eredményezi, hogy néhány ezer sor kimenet (tanítási lépés) után a böngésző jelentősen lelassul, majd végül összeomlik, ezért ezt a kimenetet a tanítás során rendszeresen törölni kell, valamint ennek a javítására, az eredeti darknet implementációból sok fölösleges kiírást eltávolítottam, vagy egy sorba raktam, hogy kevesebb időközönként kelljen törölni. A Colab 12 órás futást engedélyez, miután a virtuális gép újraindul, tehát mielőtt ez bekövetkezik, a tanítást le kell állítani, és a súlyokat el kell menteni. A virtuális géppel való kapcsolat véletlen időközönként megszakad, amit ha nem indítunk újra időben, akkor a gép újraindul, és a haladás elvész, ezért érdemes a tanítások alatt a számítógép közelében maradni.

### Futtatásokhoz használt felépítés



A rendszer három részre bontható: egy Intel NUC beágyazott számítógépen adatgyűjtés és továbbítás, egy kezelő számítógép, amin a fogadó program (FFmpeg dekóder) és a Unity program fut a VR szemüveghez. Jelen feladat keretében az első két egységen dolgoztam (NUC és FFmpeg dekóder). Az Intel NUC számítógépen több konfiguráció mellett mértem a hálózat futási idejét is pontosságát, a méret függvényében. A pontosság futás idő függésének mérése platform függetlenül elvégezhető. Mivel a futási idők mérése azt mutatja, hogy az Intel NUC-on nem reális valós idejű működés elérése, így a detekciók kirajzolását csak a kezelő számítógépen implementáltam. Ezt úgy valósítottam meg, hogy az FFmpeg dekóder programba statikusan linkeltem a darknetet, majd a beérkező képkockákon végrehajtott inferenciákat rajzoltam a továbbítandó képkockákra.

## **Használt neuronháló modell**

A használt neuronháló lényege, hogy konvolúciós szűrüket alkalmazunk a bemeneti képeken, majd a szűrt képek méretét felére csökkentjük, maxpool rétegek használatával. Ezt a két lépést ismételgetve eljutunk a YOLO típusú kimeneti réteghez, ahol a háló kimenetét összehasonlítva a várt kimenettel, tudjuk vissza terjeszteni a hibát a konvolúciós rétegek súlyaira. Ezt az eljárást hívjuk back propagation-nak, melynek segítségével iteratívan javítjuk a konvolúciós szűrők súlyait. A választott modell neve YOLO, lényege hogy nem egy kategorizáló hálót csúsztatunk végig a képeken, hogy az egyes objektumokat detektáljuk, hanem egyetlen egy futás alatt kapjuk meg a felismert objektumokat határoló téglalapokat, így sokkal gyorsabb, akár valós idejű applikációt is csinálhatunk. Ezt a YOLO modell alkotója úgy oldotta meg, hogy a kimeneti réteget négyzetekre osztotta fel, melyekben fix számú objektumot detektál a háló, mindegyiknek a hozzá tartozó négyzetben lévő pozíciójával és méretével, valamint One-Hot kódolt kategóriájával.[[5]](#footnote-4) Ennek a modellnek is a Tiny változatát választottam, mivel ez a háló is jelentős számítási kapacitást igényel.

További előnye a választott modellnek, hogy az eredeti, C implementációban (Darknet), a háló újra tanítás nélkül átméretezhető, azaz a pontosság rovására javítható a futási idő.

A Darknet a hálók struktúrájának és tanítási paramétereinek tárolására .cfg konfigurációs fájlokat használ, melyek struktúrája az INI[[6]](#footnote-5) fájlok struktúráját használja. Ezekben a fájlokban főképp a tanítási paraméterek változtatására van lehetőségünk, ugyanis előtanítot súlyokkal szeretnénk dolgozni, ami azt jelenti, hogy nem változtathatunk a háló azon részén, melyhez súlyok is tartoznak (konvolúciós rétegek, yolo rétegek). A modell eredeti implementációjában sok hasznos eszköz előre implementálva volt (alapvető réteg típusok, optimalizáló algoritmusok, adat augmentálás). Ezeket néhány további funkcióval egészítettem ki. Az adat augmentálást, mivel álló jelenetet felvevő videó folyamon is mozogtak a detekciók határai, zajjal egészítettem ki, valamint mivel kevés tanító adatot tudtam feltölteni, és az adat augmentálás a tanítás közben nem növeli a feltöltendő adathalmaz méretét, csak a tanítási időt, így forgatással is kiegészítettem. Ezt úgy valósítottam meg, hogy a képpel együtt a várt körülhatároló téglalapokat is elforgattam, majd az elforgatott téglalapok sarkaira új téglalapokat illesztettem. Ennek a módszernek egy problémája az, hogy felül becsüli az objektumok valódi méretét, ugyanis ha az objektum nem töltötte ki az eredeti, határoló téglalap sarkait, akkor az ezekre a pontokra illesztett téglalap távolabb lesz az objektumtól, mint ahogy a képen szerepel. Egy lehetséges megoldás erre a problémára a téglalapok területének figyelembe vétele lett volna. Ezt az ötletet nem jártam körbe. Mivel a forgatás implementálása sok próbálgatást vett igénybe, és ezek a próbálgatások sok lépést vettek igénybe, a Colab használata miatt, ezért ezt először a Colab felületén használható Pythonban implementáltam, majd azt az implementációt írtam át, hogy a C-ben megvalósított darknet része legyen. A python kód megtalálható a Colabban használt onlab.ipynb file elején. A zajjal való augmentálást minden pixelhez egyszerű gauss zaj hozzáadásával oldottam meg. Egy részletesebb megoldás az lett volna, ha nem sima Gauss zajt adok a képekhez, hanem valamit, ami jobban hasonlít a kocsi esetében is használt FFmpeg kódolásához. Ez a megoldás, mivel jelentősen bonyolultabb, mint a megvalósított változat, nem került implementálásra.

A darknet implementáció további problémája a regularizációk hiánya volt. A dropout rétegek implementációja nem volt teljesen kész, nem voltak átméretezhetőek, ahogyan a konvolúciós vagy a YOLO rétegek, annak ellenére, hogy ezekhez a rétegekhez nem tartozik saját súly (ez azt is jelenti, hogy használhatóak előtanított súlyokkal). A dropout lényege, hogy a tanító lépések során véletlenszerűen kikapcsoljuk a súlyok egy részét, hogy ne tanuljanak. Ezzel arra kényszerítjük a hálót, hogy minden súlyban próbáljon hasznos információt tárolni. A dropout rétegek implementációját befejeztem, hogy ezek a rétegek is átméretezhetőek legyenek.

## **Tanító adatok**

Mivel a modell autó jelenleg főképp az egyetemi laboratóriumban üzemel, ezért beltéri objektumokról volt érdemes képeket használni a tanításhoz. Ehhez a Princeton által készített SUN RGB-D[[7]](#footnote-6) [[8]](#footnote-7) [[9]](#footnote-8) [[10]](#footnote-9) adatbázisát választottam, ugyanis beltéri jelenetekről tartalmaz több mint tízezer képet. Ebben az adathalmazban viszont aránylag kevés kép van emberekről, így ezt az adathalmazt a PASCAL VOC[[11]](#footnote-10) adatbázissal egészítettem ki. A SUN RGB-D adathalmaz több helyről gyűjtött képeket tartalmaz, amelyeknek mind külön mappájuk van. Ezeken belül, minden képnek is van egy mappája, amiben több almappa és file található. Ezek közül a számunkra érdekes fájlok:

* image/\*.jpg - maga a kép, amihez a címkék tartoznak
* annotation2D3D/index.json - annotációk, json formátumban

Az annotációkat tartalmazó json hasznos részei:

* Gyökér['frames'][0]['polygon'] - lista az objektumokhoz tartozó négyzetekről
  + Gyökér['frames'][0]['polygon'][n]['x'] - a négy sarok x koordinátája
  + Gyökér['frames'][0]['polygon'][n]['y'] - a négy sarok y koordinátája
  + Gyökér['frames'][0]['polygon'][n]['object'] - az objektum azonosítója
* Gyökér['objects'] - lista az objektumokról, ahol az objektumok indexe az előző “objektum azonosítója” mezőnek felel meg
  + Gyökér['objects'][objektum azonosítója]['name'] - kategória neve

Mivel a modellautó számára csak azt kell tudni, hogy van e az adott helyen objektum, így az összes kategóriát egy kategóriába vontam össze. Ezen műveletek elvégzéséhez (adathalmazok összevonása, YOLO formátumba konvertálása, kategóriák szűrése, összevonása) egy PHP szkriptet írtam (merge\_voc\_sunrgbd.php és SUNRGBD2YOLO.php), mivel az otthoni környezetben a PHP nyelv volt kényelmesen elérhető.

A PHP szkriptek, a bennük található változók értékének változtatásával konfigurálhatók. Segítségükkel megadható többek között a forrás és cél adathalmazok elérési útja, a megtartandó kategóriák, az összevonni kívánt kategóriák, valamint a tanítási és a tesztelési vágások mérete. Validációs adathalmaz készítése kézzel történik. A tanítási-tesztelési/validációs vágást 80-20-ra választottam, majd a PHP szkript által generált tesztelési txt-t szövegszerkesztőben megfeleztem és egy új, ‘validációs halmaz’ szöveges dokumentumba másoltam a másik felét. A darknet implementációban helytelenül van elnevezve a teszt adathalmaz, ugyanis a tanítás során, kiértékelésre használt adathalmazt validációs adathalmaznak hívjuk[[12]](#footnote-11). A tanítási halmazzal tanítjuk a hálót, a validációssal követjük nyomon a tanulás alatt a detekciók minőségét, a teszt halmazzal pedig a tanítások után hasonlítottam össze a különböző hiperparaméterekkel tanított hálókat. A list\_images.php segítségével több kép is egyszerre megjeleníthető, ez hasznos lehet tesztelés során. Az összes PHP file a process\_SUNRGBD[[13]](#footnote-12) mappában található.

A SUN RGB-D-t YOLO formátumba konvertáló szkript először kikeresi a képekhez tartozó összes mappát (rekurzív mappa listázás) ezek elérési útját egy tömbben tárolja. A tömbön végig futva, megnyitja az annotációkat tartalmazó json fájlt, összevonja a konfiguráció alapján a kategóriákat, valamint eldobja a tiltott kategóriákat. A képet, és a YOLO formátumú annotációkat tároló txt fájlt elmenti a kiementként konfigurált mappába. Ha hiba lép fel a fájlok megnyitása során, akkor azt a képet átugorja. Mivel a képek a SUN RGB-D adathalmazban viselhetik ugyanazt a nevet és a szkript egy kimeneti mappába másolja át őket, így az esetleges ütközések elkerülése végett, átnevezi őket, de mivel ez nehezíti a hibák keresését, így egy log.txt-t is generál a kimeneti mappába, amiben végig lehet követni a fájlok átnevezését, így hiba esetén vissza lehet keresni a hibás képet az eredeti adathalmazban. A log.txt-n kívül a train és test txt-ket is legenerálja, amik a YOLO számára szükségesek tanítás során, hogy megtalálja a tanító képeket. Az átalakítás után néhány minta képet is megjelenít az egyes kategóriákból, hogy könnyítse a kevés képpel rendelkező kategóriák megtalálását. (Ezek főképp elírt kategória nevek miatt szerepeltek, vagy a kategóriákkal kapcsolatos félreértések miatt keletkeztek, például a “truncated” néhol külön kategóriát kapott, néhol pedig nem).

A YOLO típusú adathalmazok összevonására használt szkript működése során beolvassa a két, összevonni kívánt adathalmaz train és test listáját, amiben a képek fájlnevei szerepelnek sortöréssel elválasztva. A képekhez tartozó annotációkat tartalmazó fájloknak ugyan az a nevük, mint a hozzájuk tartozó képé. Az annotációkat tartalmazó fájlokat beolvassa, majd a konfigurációban megadott kategóriákat összevonja. Az új annotációkat és a hozzájuk tartozó képeket egy, a konfigurációban megadható kimeneti mappába másolja, valamint új train és test listát generál hozzájuk, amikben már az összes kép elérési útvonala szerepel.

Az adathalmazok szerkesztésén kívül lehetőség van a képek és a hozzájuk tartozó objektum határoló téglalapok kirajzolására (SUN RGB-D kép esetén get\_image.php, YOLO kép esetén get\_image\_yolo.php). A feladat megoldása során eredetileg a SUN RGB-D adatbázisban található “fullres” képeket használtam, de mivel később kiderült, hogy az annotációk nem ezekre a képekre vannak megadva, ezért ezt a problémát a vágott képek felhasználásával javítottam. A VOC adatbázis YOLO formátumba konvertálására egy python szkript már elérhető volt a YOLO honlapján[[14]](#footnote-13). A VOC adathalmazt először YOLO formátumba kell konvertálni, majd az így kapott adathalmazt lehet a merge\_voc\_sunrgbd.php-val összevonni a SUNRGBD adathalmazzal. A VOC adathalmaz YOLO formátumba konvertálását az alábbi néhány sor paranccsal lehet elvégezni:

wget https://pjreddie.com/media/files/VOCtrainval\_11-May-2012.tar

wget https://pjreddie.com/media/files/VOCtrainval\_06-Nov-2007.tar

wget https://pjreddie.com/media/files/VOCtest\_06-Nov-2007.tar

tar xf VOCtrainval\_11-May-2012.tar

tar xf VOCtrainval\_06-Nov-2007.tar

tar xf VOCtest\_06-Nov-2007.tar

wget https://pjreddie.com/media/files/voc\_label.py

python voc\_label.py

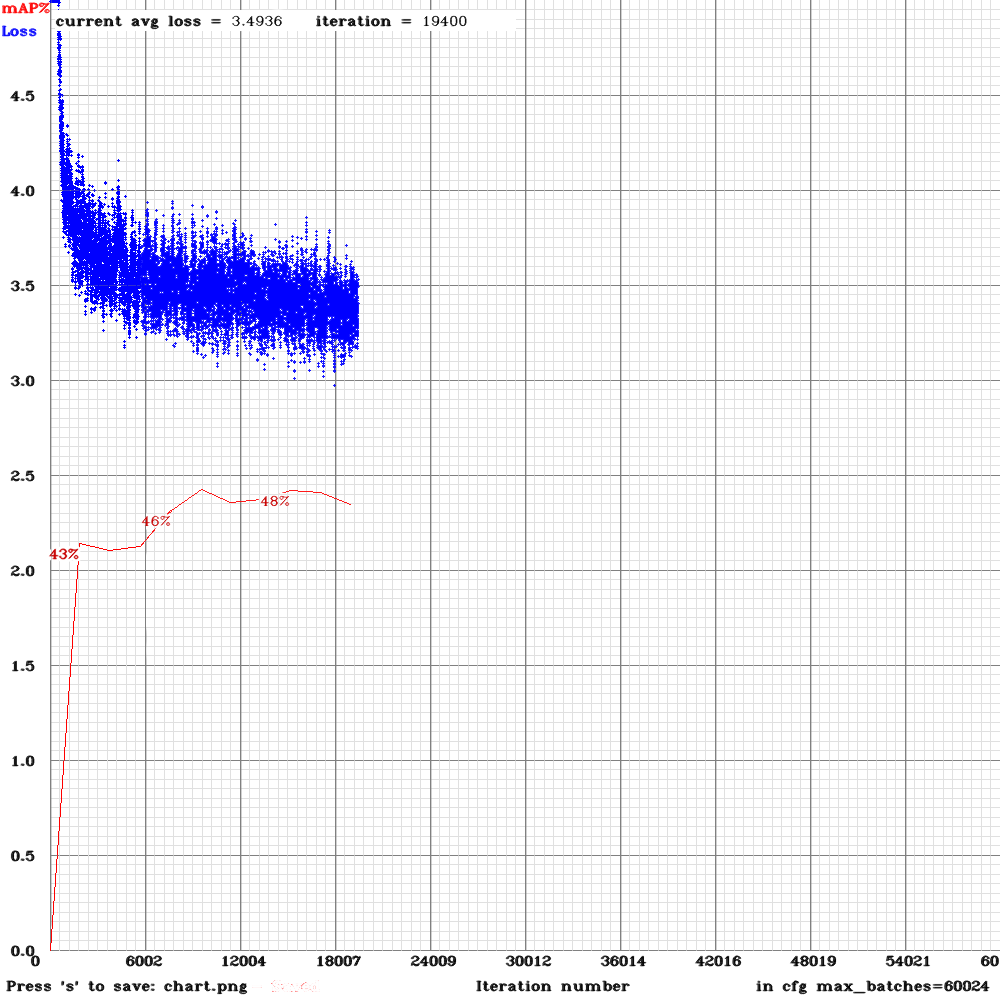
cat 2007\_train.txt 2007\_val.txt 2012\_\*.txt > train.txt

Ezek a parancsok a Colabban használt ipnb-ban is megtalálhatók. Ezek funkcióiról további leírás található a PHP szkriptekben.

**Elvégzett tanítások**

### **Próba tanítások**

A darknettel és az adathalmazokkal való ismerkedés során számos próba tanítást futtattam. Ezek közül az érdekesebbek: egy hosszú tanítás, ahol több mint 12 óráig tanítottam a hálót (A súlyok kimentésével, a Colab újraindításával és a tanítás folytatásával).[[15]](#footnote-14) Mivel a tanítást a Colab újraindításával félbe kellett szakítanom, így két tanítási görbe képet generált a darknet, melyeket utólag, képszerkesztő program segítségével vontam össze:

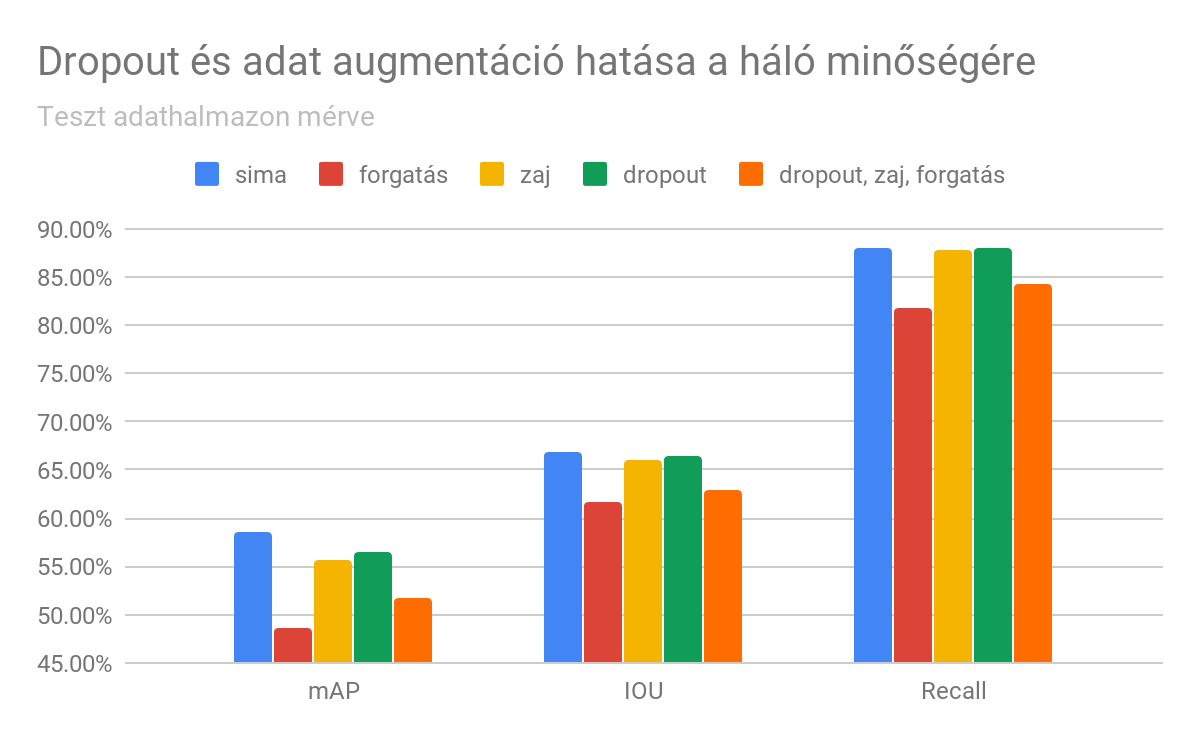


A tanulási görbe és az mAP alakulása alapján azt a következtetést vontam le, hogy míg a loss 12 óra tanítás után tovább csökken, az mAP ekkor már eléri a csúcsát, így ennél tovább nem érdemes tanítani a hálót. Az mAP az átlagos pontosságot méri a validációs adathalmazon, ahol a pontosság a helyes találatok száma osztva az összes találat számával, 50% vagy annál nagyobb detektált biztosságú objektumokat figyelembe véve. A loss a költség függvény értéke az adott kép halmazon (batch), amivel éppen tanítunk.

Három másik próba tanítás során a dropout ésszerű értékét próbáltam meghatározni.[[16]](#footnote-15) [[17]](#footnote-16) [[18]](#footnote-17) Ez az érték adja meg, hogy a neuronok hány százalékát “pihentetjük” a tanítási ciklusok során. Ha ez az érték túl nagy, akkor a hálónak nincs elég neuronja, hogy hasznos információkat tanuljon a tanító adatokból, ha viszont túl kicsi, akkor nem lesz elég erős a regularizációs hatása, tehát elképzelhető, hogy túltanul a háló a tanító adatokon, azaz nem lesz képes új képekre általánosítani. Ezen próba tanítások során, a 30%-os dropout túl erősnek bizonyult. Ezen kívül futtattam egy 128-as batch méretű próba tanítást is, viszont ennek nem volt nagy hatása a tanítás minőségére, viszont lassabbak lettek a tanító lépések (de lépésenként kétszer akkora javulás is volt a detekciók minőségében).[[19]](#footnote-18) Az eredeti optimalizáló helyett megpróbáltam Adam optimalizálót is használni, viszont ez előtanított súlyok használata esetén a második lépés után divergált. Új súlyok használata esetén nagyon lassan tanult, az eredeti, exponenciális jellegű tanulási ráta növelővel szemben.[[20]](#footnote-19)

### **Mérő tanítások**

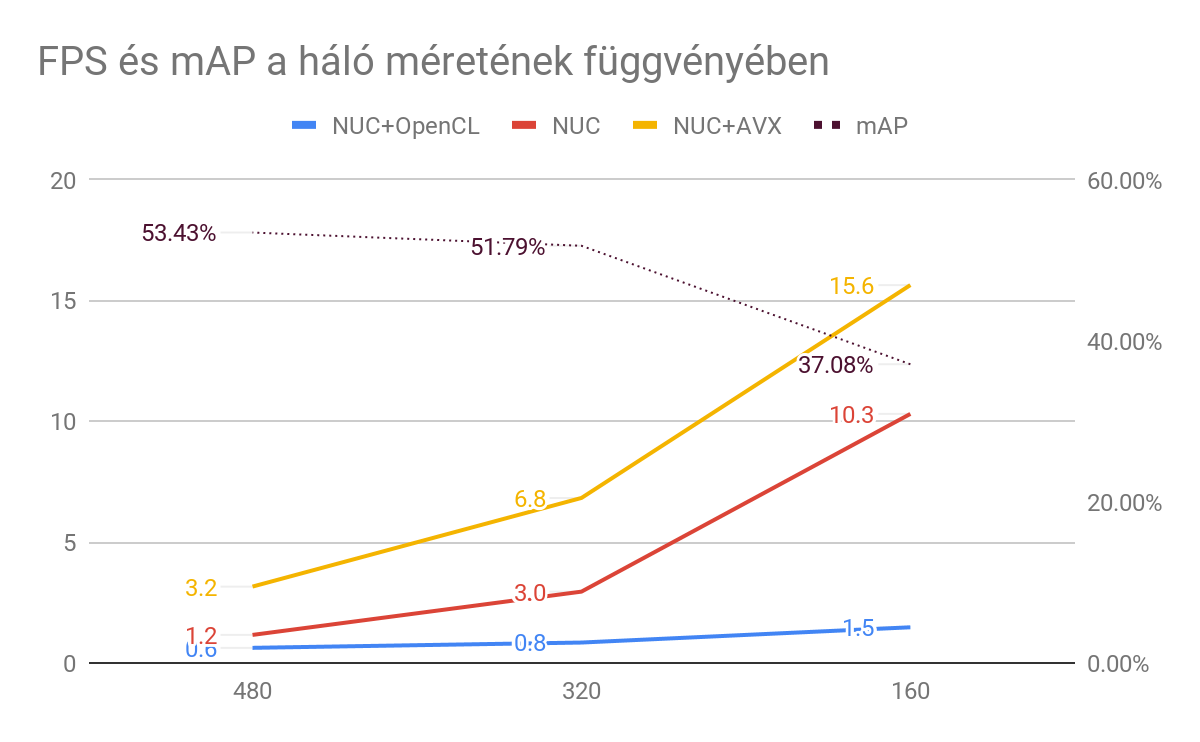
A próba tanítások elvégzése után, mérő tanításokat végeztem, amikkel a dropout, a zaj és a forgatás hatását próbáltam összemérni. Ezekben a zaj maximális értékét 20%-ra a forgatás maximális értékét 0.2 radiánra (kb. 11.5°), a dropout értékét pedig 15%-ra választottam. A tanításokat körülbelül 12 óráig futtattam. A tanítások befejezése után megmértem a hálók végleges mAP, recall és IOU metrikáit, a teszt adathalmazon. A recall metrika a helyes detekciók száma osztva az összes képen lévő objektum számával, 50%-os biztossági detekciók esetében, az IOU pedig a detekciók metszetének területe osztva a az uniók területével. Elmentettem ezen kívül a tanítás logját (azon részét, amit ki tudtam másolni a colabból, jobb megoldás lett volna, ha ezt fájlba írom az implementációban a konzol helyett, erre sajnos csak későn jöttem rá) a beállított paramétereket tároló konfigurációs fájlt és a tanulási görbét. Összesen öt mérő tanítást futtattam. Először egy referencia méréssel kezdtem, ahol az eredeti tiny YOLOv3 konfigurációt használtam, hogy legyen egy viszonyítási alap. Futtattam egy-egy mérést ahol csak a dropout, a forgatás vagy a zaj volt bekapcsolva, valamint futtattam egy tanítást, ahol mindhárom engedélyezve volt egyszerre.



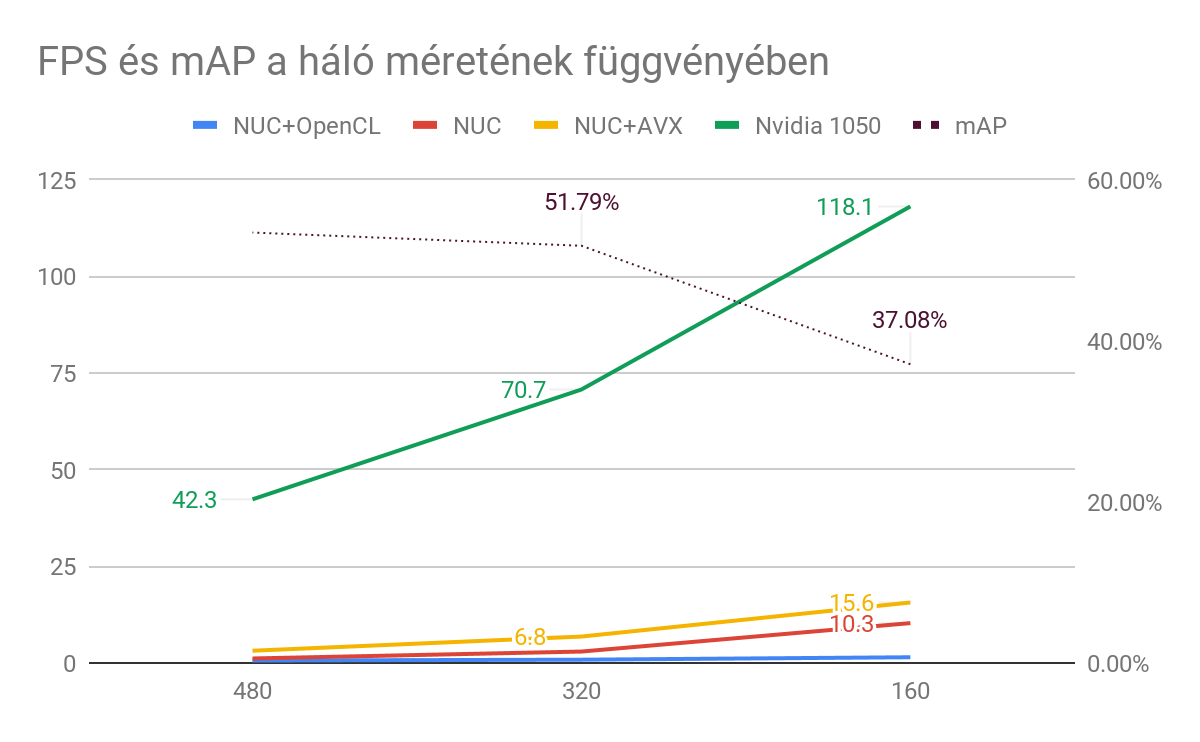
Az oszlopdiagramon látható, hogy a sima, dropout, zaj és forgatás nélküli tanítás teljesít a legjobban. Ez a várt eredménnyel szembe megy, ugyanis mind az adat augmentálás, mind a regularizáció alkalmazásának javítani kellett volna a háló minőségén, mivel mindkét eljárásnak növelnie kéne a háló általánosító képességét. Elképzelhető, hogy a zajjal augmentált képeken tanított modell azért teljesít rosszabbul, mert a zajjal való augmentálást az FFmpeg kódolás miatt okozott zaj miatt implementáltam, ami a teszt adathalmazban lévő statikus képeken nem jelentkezik. A forgatás esetében, az előbbiekben említett felül becsülési probléma okozhatja a romlást. Az összehasonlítás eredményeként ez okozta a legrosszabb romlást, viszont látszik, hogy a másik két eljárás a forgatás rontó hatását kis mértékben, de ellensúlyozza. A dropout alkalmazása rontja a legkisebb mértékben a háló minőségét. Ez valószínűleg azért van, mert a 15%-os dropout is túl erős a hálónak. Konvolúciós rétegek esetén nem annyira fontos a regularizáció alkalmazása, ugyanis a konvolúciós szűrőket amúgyis végig csúsztatjuk a képen, így megosztott, átlagolt súlyaik vannak. Mivel a YOLO modell legkisebb variánsát használtam, így a túltanulás esélye amúgy is kicsinek lehet (kicsi a modell a problémához képest, így nem tudja megtanulni a tanító adatokat, csak a bennük rejlő statisztikai mintákat). Érdekes lett volna egy olyan tanítást futtatni, ahol csak a zaj és a dropout vannak engedélyezve, ugyanis abban a mérésben, ahol mindhárom engedélyezve volt, a zaj és a dropout mintha javítottak volna a forgatás által okozott visszaesésen. Jobb eredményeket eredményezhetett volna, ha több ideig tanítom a modelleket és early stopping jelleggel választom ki a tanítás után a legjobb súlyokat, mivel az augmentált tanítások nem jutottak el ugyan addig a tanítási lépésig, mint a sima tanítás, ugyanis több időbe telltek az egyes tanítási lépések. Ezek kipróbálására viszont idő hiányában sajnos nem jutott sor. Mind a három metrika ugyan úgy változik a különböző modellek esetén.

## **Futási idők összehasonlítása**

A modell autó és a rajta megvalósítandó önvezető algoritmusok szempontjából kritikus az adatfeldolgozás sebessége. A mély neuronháló futtatása jelentős számítási kapacitást vesz igénybe, előny viszont a YOLO modell esetében, hogy a darknet implementációja sok hardveres gyorsító eszközt támogat (OpenCL, CUDA, AVX), valamint hogy átméretezhető rajta a neuron háló. A modell autón használt Intel NUC-nak van egy processzorba épített grafikus vezérlője, melynek van OpenCL támogatása. A kezelő számítógépen pedig az Nvidia 1050-es grafikus kártya támogatja a CUDA-t. Ezek kipróbálásához a hálót teszt képeken futtattam, hogy az egy képkockán vett inferencia sebességét lemérjem.



Három mérés átlagának a reciprokát vettem, így kaptam az ábrán feltűntetett FPS értékeket. Az mAP értékeket minden háló méret esetén csak egyszer mértem le, viszont azt az egész teszt adathalmazon futtattam. Mivel az Nvidia teljesítménye jelentősen felülmúlja a NUC által nyújtott teljesítményt, így az saját ábrát érdemelt ki magának (összenyomta a NUC sebességét mutató görbéket):



Három háló mérettel próbálkoztam: 480x480, 320x30 és 160x160. A háló méretének 32 többszörösének kell lennie. Mivel Windowson a sebesség méréséhez egy python szkriptet használtam, így ott több mintának az átlagát vettem. A NUC-on konfigurációnként háromszor futtattam a modellt parancssorból, és a kiírt sebességeket átlagoltam. A modell futtatásához használt parancs megtalálható az ipynb fájlban.

## **További gondolatok**

Az eddig felsorolt problémákon kívül (forgatás felülbecslése, dropout túl erős, zaj csak sima gauss) további problémák is vannak: a körülhatároló téglalapok kevés információt szolgáltatnak a vezetés problémájának megoldásához, fontos lenne például mélységi információk becslése is (távolság, objektum mégségi kiterjedése). Az adathalmazok, mivel különböző adatbázisok összevonásából származnak, így pontatlanok, ugyanis nem ugyan azok a kategóriák vannak felcímkézve mind a kettőben, valamint a SUN RGB-D adathalmazt kézi címkézéssel készítették, ahol nem előre meghatározott kategóriák alapján címkéztek. Ez azt jelenti, hogy vannak elírások, különböző nevek ugyan azoknak az objektumoknak és más más objektumok vannak felcímkézve.

A modell felhasználhatóságát jelentősen növelné, ha az egymást követő képkockákon egymáshoz lennének rendelve az egyes detekciók, ezáltal akár a hibásan inferált képkockákon néhány hibát is ki lehetne javítani (pl. aluláteresztő szűrővel). Ezt talán a modell struktúrájának változtatásával is meg lehetne valósítani, 3D konvolúciók felhasználásával, vagy rekurrens háló alkalmazásával. Hasznos lenne további algoritmusok szempontjából, ha nem csak határoló téglalapok lennének a háló kimenetei, hanem poligonok, esetleg pixel szintű szegmentálás, valamint ha mélységi információk becslésére is képes lenne. A neuronhálók rugalmasságát felhasználva akár további szenzorok adatait is fel lehetne használni, esetleg egy síkban pásztázó lidar információival kiegészítve a videó folyam adatait, így könnyebben tudna az egész képre mélységi információt becsülni.

1. <https://www.aut.bme.hu/Task/18-19-tavasz/Gepi-tanulas-alapu-objektumdetektalas> [↑](#footnote-ref-0)
2. <http://cocodataset.org/#download> [↑](#footnote-ref-1)
3. <https://pjreddie.com/darknet/yolo/> [↑](#footnote-ref-2)
4. <https://github.com/19greg96/yolo-onlab> [↑](#footnote-ref-3)
5. <https://arxiv.org/pdf/1506.02640.pdf> [↑](#footnote-ref-4)
6. <https://en.wikipedia.org/wiki/INI_file> [↑](#footnote-ref-5)
7. S. Song, S. Lichtenberg, and J. Xiao.

   [SUN RGB-D: A RGB-D Scene Understanding Benchmark Suite](http://rgbd.cs.princeton.edu/paper.pdf)

   Proceedings of 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR2015) [↑](#footnote-ref-6)
8. N. Silberman, D. Hoiem, P. Kohli, R. Fergus. Indoor segmentation and support inference from rgbd images. In ECCV, 2012. [↑](#footnote-ref-7)
9. A. Janoch, S. Karayev, Y. Jia, J. T. Barron, M. Fritz, K. Saenko, and T. Darrell. A category-level 3-d object dataset: Putting the kinect to work. In ICCV Workshop on Consumer Depth Cameras for Computer Vision, 2011. [↑](#footnote-ref-8)
10. J. Xiao, A. Owens, and A. Torralba. SUN3D: A database of big spaces reconstructed using SfM and object labels. In ICCV, 2013 [↑](#footnote-ref-9)
11. <http://host.robots.ox.ac.uk/pascal/VOC/pubs/everingham15.html#bibtex>

    **The PASCAL Visual Object Classes Challenge: A Retrospective**

    Everingham, M., Eslami, S. M. A., Van Gool, L., Williams, C. K. I., Winn, J. and Zisserman, A.

    *International Journal of Computer Vision, 111(1), 98-136, 2015* [↑](#footnote-ref-10)
12. <https://stackoverflow.com/questions/2976452/whats-is-the-difference-between-train-validation-and-test-set-in-neural-netwo> [↑](#footnote-ref-11)
13. <https://github.com/19greg96/yolo-onlab/tree/master/process_SUNRGBD> [↑](#footnote-ref-12)
14. <https://pjreddie.com/media/files/voc_label.py> [↑](#footnote-ref-13)
15. <https://github.com/19greg96/yolo-onlab/tree/master/process_SUNRGBD/weights/VOC_SUNRGBD_3/augmented_data_2> [↑](#footnote-ref-14)
16. <https://github.com/19greg96/yolo-onlab/tree/master/process_SUNRGBD/weights/VOC_SUNRGBD_3/augmented_data_rotate_dropout_01> [↑](#footnote-ref-15)
17. <https://github.com/19greg96/yolo-onlab/tree/master/process_SUNRGBD/weights/VOC_SUNRGBD_3/augmented_data_rotate_dropout_02> [↑](#footnote-ref-16)
18. <https://github.com/19greg96/yolo-onlab/tree/master/process_SUNRGBD/weights/VOC_SUNRGBD_3/augmented_data_rotate_dropout_03> [↑](#footnote-ref-17)
19. <https://github.com/19greg96/yolo-onlab/tree/master/process_SUNRGBD/weights/VOC_SUNRGBD_3/augmented_data_rotate_dropout_01_batch_128> [↑](#footnote-ref-18)
20. <https://github.com/19greg96/yolo-onlab/tree/master/process_SUNRGBD/weights/VOC_SUNRGBD_3/augmented_data_rotate_dropout_adam> [↑](#footnote-ref-19)