

UNIVERSITATEA „SAPIENTIA” DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,
TÎRGU-MUREŞ
SPECIALIZAREA CALCULATOARE

**RECUNOAȘTEREA SEMINȚELOR
DE PLANTE ȘI DETERMINAREA
PROPRIETĂȚILOR FIZICE**

Coordonator științific:

Conf. dr. ing. Brassai Sándor Tihamér

Absolvent:

Bíró Apor

2024

UNIVERSITATEA „SAPIENTIA” din CLUJ-NAPOCA
Facultatea de Științe Tehnice și Umaniste din Târgu Mureș
Specializarea: **Calculatoare**

Viza facultății:

LUCRARE DE DIPLOMĂ

Coordonator științific:
Conf. dr. ing. Brassai Sándor Tihamér

Candidat: **Bíró Apor**
Anul absolvirii: **2024**

a) Tema lucrării de licență:

RECUNOAȘTEREA SEMINTELOR DE PLANTE ȘI DETERMINAREA PROPRIETĂȚILOR FIZICE

b) Problemele principale tratate:

- Studiu bibliografic privind algoritmele de recunoasterea obiectelor si privind aplicatiile Android utilizand acestea
- Realizarea unei set de date despre 21 de specii de seminte de plante
- Antrenarea si validarea ale 6 tipuri de retele neuronale
- Utilizarea procesarii imaginilor pentru a realiza zona, diametrul si masa semintelor
- Realizarea unei aplicatii Android pentru recunoasterea si determinarea proprietatilor fizice al semintelor de plante

c) Desene obligatorii:

- Schema bloc al aplicației
- Diagrame UML privind software-ul realizat.

d) Softuri obligatorii:

- Aplicație Android pentru recunoasterea si determinarea proprietatilor fizice al semintelor de plante

e) Bibliografia recomandată:

- Jiang, Peiyuan, et al. "A Review of Yolo algorithm developments." *Procedia Computer Science* 199 (2022): 1066-1073.
- Karar, Mohamed Esmail, et al. "A new mobile application of agricultural pests recognition using deep learning in cloud computing system." *Alexandria Engineering Journal* 60.5 (2021): 4423-4432.

f) Termene obligatorii de consultații: săptămânal

g) Locul și durata practicii: Universitatea „Sapientia” din Cluj-Napoca,

Facultatea de Științe Tehnice și Umaniste din Târgu Mureș

Primit tema la data de: 31.03.2020

Termen de predare: 27.06.2021

Semnătura Director Departament

Semnătura coordonatorului

Semnătura responsabilului
programului de studiu

Semnătura candidatului

Declarație

Subsemnata/ul , absolvant(ă) al/a specializării , promoția..... cunoscând prevederile Legii Educației Naționale 1/2011 și a Codului de etică și deontologie profesională a Universității Sapientia cu privire la furt intelectual declar pe propria răspundere că prezenta lucrare de licență/proiect de diplomă/disertație se bazează pe activitatea personală, cercetarea/proiectarea este efectuată de mine, informațiile și datele preluate din literatura de specialitate sunt citate în mod corespunzător.

Localitatea,

Data:

Absolvant

Semnătura.....

Declarație

Subsemnata/Subsemnatul , funcția..... ,
titlul științific..... declar pe propria răspundere că
..... , absolvent al specializăriiconform
HG..... a întocmit prezenta lucrare sub îndrumarea mea.

În urma verificării formei finale constat că lucrarea de licență/proiectul de diplomă/disertația corespunde cerințelor de formă și conținut aprobată de Consiliul Facultății de Științe Tehnice și Umaniste din Târgu Mureș în baza reglementărilor Universității Sapientia. Luând în considerare și Raportul generat din aplicația antiplagiat „Turnitin” consider că sunt îndeplinite cerințele referitoare la originalitatea lucrării impuse de Legea educației naționale nr. 1/2011 și de Codul de etică și deontologie profesională a Universității Sapientia, și ca atare sunt de acord cu prezentarea și susținerea lucrării în fața comisiei de examen de licență/diplomă/disertație.

Localitatea,

Data:

Semnătura îndrumătorului

TURNITIN

Extras

Cuvinte-cheie: CNN, Deep learning, Detectarea obiectelor, Android, Semințe de plante

In vremea prezentă, Inteligența Artificială se bucură de multă atenție nu numai în Știința Calculatoarelor, ci și în multe alte discipline diferite. Câteva dintre acestea sunt științele sociale, medicina sau chiar științele agricole. În zilele noastre, smartphone-urile noastre petrec mai mult timp în mâini decât în buzunare. Idei noi, soluții noi și inovații noi se nasc în mod constant pentru a combina științele agricole și tehnologia computerelor. În această lucrare de diplomă, am vrut să încorporez diferite algoritme pentru detectarea obiectelor, bazate pe rețele neuronale artificiale conoluționale, într-o aplicație Android care recunoaște diferite semințe de plante cultivate prin camera dispozitivului în timp real. Pe lângă detectarea obiectelor, efectuează și determinarea zonei, masei și dimensiunii al semințelor cu o valoare de referință. Cele 21 tipuri de semințe de plante antrenate includ, de exemplu, trei tipuri de măzăre, două tipuri de porumb, floarea soarelui, grâu, orz, secără, bob etc. Cu setul de date pregătit de mine și cu câteva ajutor, și care conține aproape 50.000 de mostre, YOLO-urile și MobileNet-urile au reușit să atingă precizii de aproape 80%. Din cauza erorilor de compatibilitate, în cele din urmă nu am reușit să integrez doar algoritmele YOLO în aplicație. Scopul declarat al acestei teze a fost o aplicație de recunoaștere a semințelor de plante ușor de utilizat, precisă, care funcționează și în condiții de câmp. În final, am reușit să creez o aplicație care funcționează în condiții de laborator, care poate fi potrivită pentru utilizare în condiții agricole cu posibile dezvoltări ulterioare.

**SAPIENTIA ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR
SZÁMÍTÁSTECHNIKA SZAK**

**NÖVÉNYMAGOK FELISMERÉSE
ÉS FIZIKAI TULAJDONSAKAIK
MEGHATÁROZÁSA**

Témavezető:

Dr. Brassai Sándor Tihamér
egyetemi docens

Végzős hallgató:

Bíró Apor

2024

Kivonat

Kulcsszavak: CNN, Deep learning, Objektumdetektálás, Android, Növénymagok

Korunkban a Mesterséges Intelligencia nem csak a Számítástechnikában, hanem sok más különböző tudományában is rengeteg figyelemnek örvend. Ezen tudományok közül egy párat hogy felsorolják például a társadalomtudományokban, az orvostudományokban vagy építészetben az agrártudományokban. Okostelefonjaink manapság pedig már legtöbb esetben több időt töltenek a kezünkben, mint a zsebünkben. Az agrártudományok és a számítástechnika ötvözésére folyamatosan új ötletek, új megoldások, új innovációk születnek. Ebben a szakdolgozatban különböző konvolúciós mesterséges neurális hálózatokon alapuló algoritmusokat szerettem volna beépíteni egy Android alkalmazásba, amely kamerán keresztül, valós időben ismer fel különböző kultúrnövénymagokat. Objektumdetektálás mellett pedig referenciaértékkel történő számításokat végez ezek területéről, tömegéről, illetve méretéről. A tanított 21 növénymagfaj közé tartozik például három típusú Borsó, két típusú Kukorica, Napraforgó, Búza, Árpa, Rozs, Lóbab stb. Az általam, illetve némi segítséggel készített, majdnem 50.000 mintát tartalmazó tanítóhalmazzal csaknem 80%-os pontosságot tudtak elérni ugyanúgy a YOLO-k, mint a MobileNet-ek. Kompatibilitási hibák miatt nem sikerült végül csak a YOLO algoritmusokat beépítenem az alkalmazásba. A kitűzött célja ennek a dolgozatnak egy egyszerűen használható, pontos, mezei körülmények között is működő növénymagfelismerő alkalmazás volt. Végül sikerült elkészítenem egy labor körülmények között működő alkalmazást, amely esetleges továbbfejlesztéssel terepen történő használatra is alkalmas lehet.

Abstract

Keywords: CNN, Deep learning, Object detection, Android, Plant Seeds

In the present, Artificial Intelligence enjoys a lot of attention not only in Computer Science, but also in many other sciences. A few examples out of these sciences are social sciences, medicine or even agricultural sciences. Nowadays our smartphones spend more time in our hands than in our pockets. New ideas, new solutions, and new innovations are constantly born to combine agricultural sciences and computer technology. In this diploma work I wanted to incorporate different CNN based object detection algorithms into an Android application that recognizes different cultivated plant seeds through the camera of the smartphone, in real-time. In addition to object detection, it performs calculations of the seeds area, mass, and size with a reference value. The 21 types of plant seeds which were trained include, for example, three types of peas, two types of corn, sunflower, wheat, barley, rye, faba bean etc. With a training dataset containing almost 50,000 samples created by me with some help, YOLOs and MobileNets were able to achieve an accuracy of almost 80%. Due to compatibility errors, I only managed to integrate the YOLO algorithms into the application. The stated goal of this thesis was an easy-to-use, accurate plant seed recognition application that works even at the fields. Finally, I managed to create an application that works in laboratory conditions, which with possible further development could also be suitable for use at the fields.

Tartalomjegyzék

1	Bevezetés.....	14
2	Elméleti megalapozás és szakirodalmi tanulmány	15
2.1	<i>Konvolúciós neurális hálózatok.....</i>	15
2.2	<i>YOLO objektumdetektáló algoritmusok.....</i>	18
2.3	<i>Egyéb objektumdetektáló algoritmusok.....</i>	25
2.4	<i>Ismert hasonló alkalmazások.....</i>	27
2.5	<i>Felhasznált technológiák</i>	28
3	Részletes tervezés.....	30
3.1	<i>Első fázis: Tanítóhalmaz</i>	30
3.2	<i>Második fázis: Tanítókörnyezet kialakítása</i>	34
3.3	<i>Harmadik fázis: Neuronhálók tanítása</i>	35
3.4	<i>Negyedik fázis: Az alkalmazás implementálása.....</i>	42
3.4.1	<i>Felhasználói felület (UI)</i>	42
3.4.2	<i>A YOLO-k beillesztése az Androidos alkalmazásba.....</i>	43
3.4.3	<i>A valós idejű kamera megjelenítése és a bounding boxok kirajzolása</i>	46
3.4.4	<i>Fizikai tulajdonságok kiszámolása.....</i>	46
3.4.5	<i>A MobileNet-ek Androidos beillesztésének kísérletei</i>	49
4	A rendszer specifikációi és architektúrája	50
4.1	<i>Az Android alkalmazás architektúrája</i>	50
4.2	<i>Az MI működése az applikáción belül.....</i>	56
4.3	<i>Verziókövetés.....</i>	57
5	Üzembe helyezés és kísérleti eredmények	58
5.1	<i>Üzembe helyezési lépések</i>	58
5.2	<i>Felmerült problémák és megoldásai</i>	58
5.3	<i>Kísérleti eredmények, mérések.....</i>	59
5.3.1	<i>Neuronhálók sebességei.....</i>	59
5.3.2	<i>Osztályozás tesztelése</i>	59
5.3.3	<i>Tömeg számítások tesztelése</i>	61
6	Következtetések	61

6.1	<i>Megvalósítások és a rendszer felhasználása.....</i>	61
6.2	<i>Hasonló rendszerekkel való összehasonlítás</i>	62
6.3	<i>Továbbfejlesztési lehetőségek.....</i>	62

Ábrajegyzék

Táblázatjegyzék

3.1.1. TÁBLÁZAT A NÖVÉNYMAGFAJOK MINTÁINAK SZÁMAI	34
3.3.1. TÁBLÁZAT A YOLO HÁLÓK SPECIFIKÁCIÓI	36
3.3.2. TÁBLÁZAT A MOBILENET HÁLÓK SPECIFIKÁCIÓI	40
3.4.1. TÁBLÁZAT A HUAWEI P40 PRO SPECIFIKÁCIÓI	42
3.4.2. TÁBLÁZAT A LÓFOGÚ KUKORICA TÖMEGMÉRÉSEI	48
5.3.1. TÁBLÁZAT A MODELLEK PROGRAM FUTÁSA KÖZBENI ÁTLAGSEBESSÉGEK EGY KÉPKOCKÁRA.....	59
5.3.2. TÁBLÁZAT AZ OSZTÁLYOZÁSOK TESZTELÉSEI	60
5.3.3. TÁBLÁZAT TÖMEG MÉRÉSEK	61

1 Bevezetés

A Számítástechnikában, a Mesterséges Intelligencia korunkban hatalmas figyelemnek Őrvend. Ugyebár az MI ezen a szakterületen már elég régóta jelen van, úgymond itt "született", de mégis csak az utóbbi pár évben ért el olyan áttörések - gondolok akár a GPT nyelvi modellekre, kép- és formaalkotó algoritmusokra, illetve pontos és könnyen használható objektumfelismerő algoritmusokra, mint például a YOLO - amelyek már nem csak a számítógéptudósok és -mérnökök figyelmét, hanem a más szakterületekben, mint például orvostudományokban, társadalomtudományokban vagy éppenséggel agrártudományokban munkálkodók figyelmét is felkeltették.

A cél egy olyan mesterséges neurális hálózatokat alkalmazó szoftver megvalósítása volt, amely valós időben, egy okostelefon kameráján keresztül képes felismerni egy képkockán lévő különböző magfajtát, ezeket osztályozni, ezek után pedig képes legyen becslést végezni ezeknek a területére, méretére, tömegére, illetve ezen szempontok szerint, az egymás közötti arányaikra. Ezt az applikációt felhasználási szempontból majd alkalmazni lehessen a mezőgazdaságban, olyas értelemben például, hogy elkülönítsen több magfajtát, így ki tudja szűrni a nem egy kívánt fajhoz tartozó magokat vagy például mérleg nélkül tudjon tömeget becsülni.

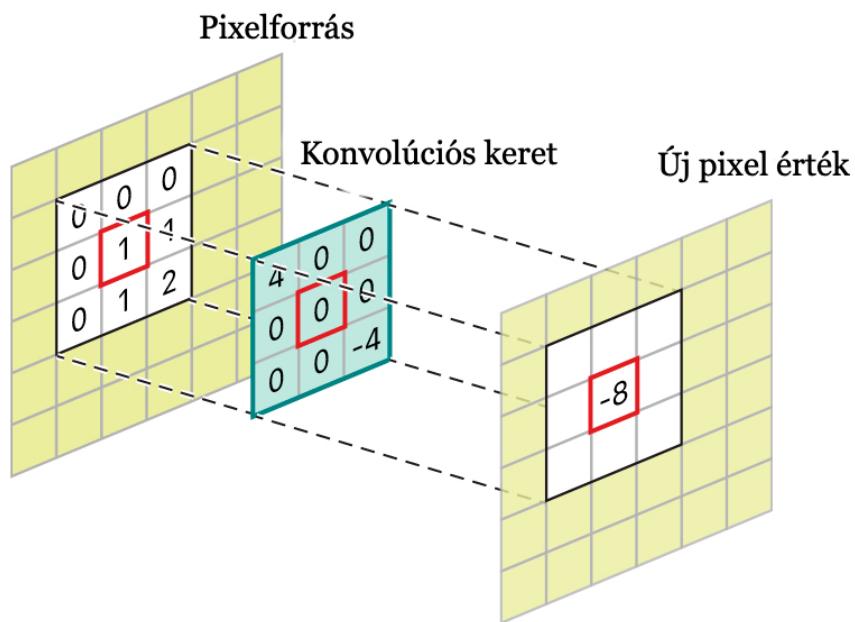
Azért ezt a MI-hoz kapcsolódó témát választottam, mert ami óta a Mesterséges Intelligencia tantárgy keretein belül mélyebb betekintést nyertem ennek a működésébe, történetébe, felhasználási területeibe, és a szerteágazó sok lehetőségeibe, elkezdett kifejezetten érdekelni. Hirtelenjében még nem tudtam magamtól elképzelni, hogy milyen témát keressek, amely érdekelne is, meg lenne benne potenciális felhasználási lehetőség, ezért témavezetőmtől kértem segítséget, aki MI gyakorlat tanárom is volt. Ő tartotta a kapcsolatot a Kertésmérnöki tanszék tanáraival és egyeztetett velük, hogy milyen projekteket lehetne közösen kezdeményezni, amelyekben ők is asszisztálhatnának. Az egyik ötlet egy növényfelismerés alapján működő gyomláló-robot, a másik ötlet pedig ez a magfelismerő szoftver volt. Eredetileg a robotot akartuk megcsinálni, a tavalyi évben azt a dolgozatot is választottuk, de mivel nem sikerült elég időt szentelni azon a nyáron a tanítóhalmaz gyűjtésnek, ezért inkább ennek a projektnek fogtam neki.

Hatalmas köszönetet szeretném mondani ezúton is vezető tanáromnak, Dr. Brassai Sándor Tihamér tanár úrnak a szakmai segítségért és útmutatásért, akinek volt türelme hozzá, akkor is, amikor néha nehezen munkálkodtam.

2 Elméleti megalapozás és szakirodalmi tanulmány

2.1 Konvolúciós neurális hálózatok

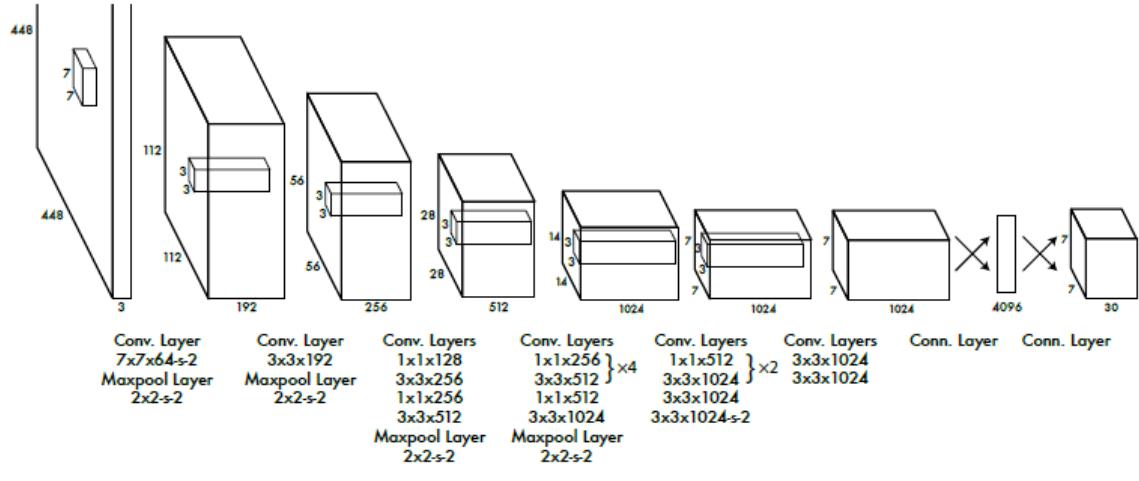
A dolgozatban az összes felhasznált objektumfelismerési modell konvolúciós neurális hálózatokra (CNN) alapszik. A CNN hálóknak az a legfontosabb ismertetőjegyük, hogy a konvolúció művelet segítségével nyernek ki tulajdonságokat (*feature*) a fényképekből. A konvolúció, mint művelet végül is csak összeadás és szorzás. Azt jelenti, hogy egy bizonyos keretet (kernel) a bemeneti fényképnek mondjuk a bal felső sarkából elindulva egy bizonyos lépésszámmal (stride) végig csúsztatunk a képen. A keret az éppen alatta lévő megegyező méretű fénykép részével konvolúciós műveletet hajt végre, az eredménye egy új pixel érték lesz. Az alábbi 2.1.1.-es ábrán látható egy konvolúciós művelet végrehajtása.



2.1.1. ábra Konvolúció szemléltetése

Látható, hogy a képen lévő pixelek a konvolúciós keret megegyező helyén lévő értékével szorzódnak, majd összeadódnak, az új eredmény egy új pixel érték. A konvolúciókat a bemeneti képen a konvolúciós rétegek végzik. Ezen a típusú rétegen kívül még két fontosabb rétegük van a CNN hálóknak. Az egyik a *Max Pooling*, melynek lényege, hogy szintén egy üres kerettel végighalad egy fényképen, az eredménye a keretben jelenleg legnagyobb értékkel rendelkező pixel értéke. A másik a teljesen kapcsolt réteg (*fully-connected layer*), amelyek a kimeneteket, vagyis az osztályozásokat végzik a konvolúciós és max pooling rétegek által előállított featurek-ből vagyis tulajdonságokból. Ezekről tanultam a Mesterséges Intelligencia tantárgy keretein belül.

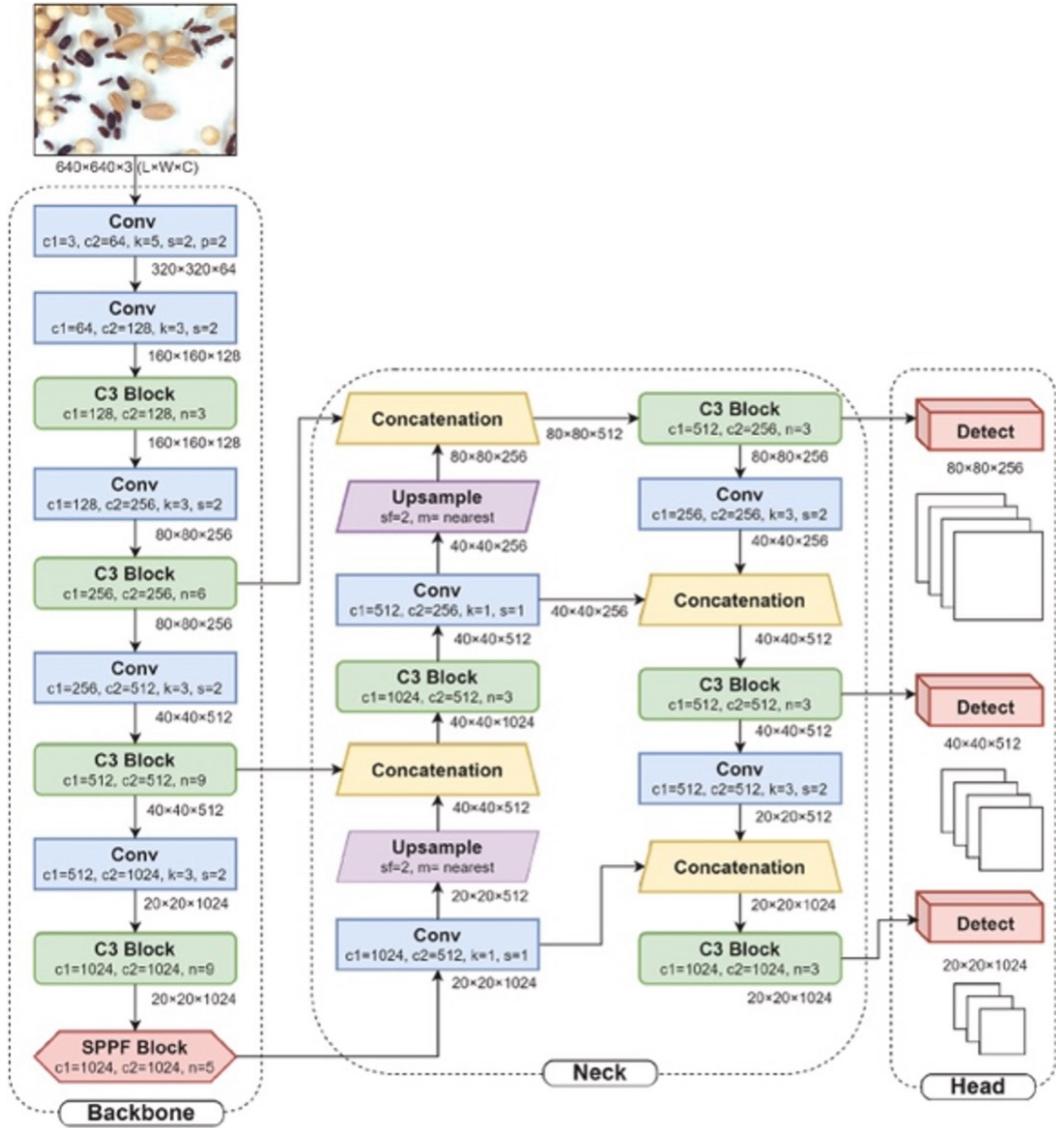
A következő 2.1.2-es ábra szemlélteti az első generációs YOLO CNN-ének a felépítését, működését. Megfigyelhető, hogy a kezdeti 448x448 méretű kép mérete folyamatosan csökken, csatornái viszont nőnek, a végül a teljesen kapcsolt rétegek után a kimenet egy tensor, amelynek értékétől függ, hogy milyen osztályt prédiktált a háló.



2.1.2. ábra A YOLOv1 CNN-e

A CNN hálók azért ennyire hatásosak, mert a rétegeik műveleteinek segítségével jól elkülöníthető közös tulajdonságokat tudnak találni az egy bizonyos, előre meghatározott osztályba tartozó fényképek között. A CNN hálók felügyelt tanítási módszert alkalmaznak, ami azt jelenti, hogy az osztályok tudottak és az értékek előre fel vannak osztva az osztályok között. minden tanítási ciklus végén az eredmények függvényében (pld. *loss*, amely meghatározza, hogy hány bemenet került a helyes kimenetre) módosulnak a súlyok (*weights*) a pontosabb prediktálás érdekében (*backpropagation*), tehát háló tanul. Tanul, mivel a súlyok megváltoztatásával a neuronok kimeneteinek az értékei megváltoznak, a bementek osztályozása ennek függvényében megváltozik, ideális esetben a jó irányba, vagyis a következő ciklusban már a bemenetek pontosabb osztályozásra kerülnek.

A YOLO egyszerű CNN architektúrája a negyedik verzióról változott meg [4]. Az Ultralytics az V5 kifejlesztéséhez is alkalmazta ezt az architektúrát. Egy rovardetektáló alkalmazás dokumentációjából [20], a következő oldalon található 2.1.3-as ábrán szépen ábrázolva van ez az architektúra. Megfigyelhetők az architektúra részei, amelyek a *Backbone*, *Neck*, *Head*. Ezeken a részeket belül pedig a modulok, amelyek több neuronréteget tartalmaznak. A Backbone (gerinc) "sima" konvoluciós neuronháló modulokat és CSP (Cross Stage Partial) modulokat (C3, ezek is tartalmaznak 3 konvoluciós réteget) tartalmaznak, amelyeknek a feladata, hogy minél nagyobb *feature map*-et (tulajdonság térképet, gyűjteményt) nyerjenek ki a bemeneti fényképekből, a kép csatornáinak a megnövelésével [20]. A CSP technológia a YOLOv4-nél jelent meg azzal a céllal, hogy csökkentse a számítások komplexitását.



2.1.3. ábra A YOLOv5 architektúrája [20]

tását, ezzel növelve a pontosságot és a sebességet [21]. Működése szerint felossza a feature map-et két részre, majd ezeket átmenetek (*transition*) segítségével, összefűzést (*concatenation*) használva újra összevonja [21]. Ezenkívül a Backbone része még egy gyors pooling modul, az SPPF (*Spatial Pyramide Pooling Fast*) modul.

A Neck (nyak) a következő része az architektúrának, ez a rész konvolúciós rétegeket, feature piramid hálókat és pooling rétegeket tartalmaz [20]. Célja finomítani (*refine*) és összesíteni (*aggregate*) a tulajdonságokat (*feature*) [20]. A végső része az architektúrának a Head (fej), amely a képek osztályozásáért, a bounding box-ok (objektumot körülhatároló keret) koordinátáinak meghatározásáért (detektálás), továbbá ugyanezek konfidenzia értékének a meghatározásáért felel [20]. A Head rész végén hajtódnak végre a végső műveletek, mint például az egymáson elhelyezkedő, vagy kis konfidenzia értékkel rendelkező bounding box-okat törlő NMS (*non-max. suppression*).

A YOLOv5 verzióknál (már a v2 és v3-nál is) már konvolúciós rétegek osztályoztak a teljesen kapcsolt rétegek helyett. A bounding boxok helyeinek meghatározására *anchor box*-oknak (horgony doboz) nevezett technológiát használtak, amely tartalmazta az detektált objektumok formáját és méretét [20]. A YOLOv8-ban már nem használtak anchor box-okat. Egy másik különbség a YOLOv8 architektúrája és e között, hogy a CSP moduljai már csak 2 konvolúciós réteget használtak.

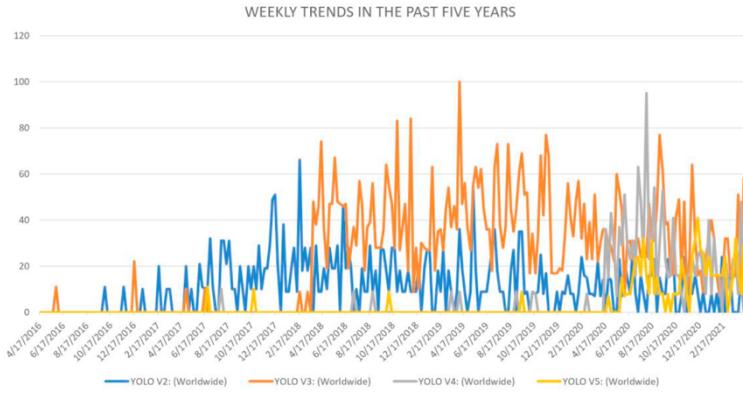
Az SSD, illetve MobileNet hálók is hasonlóképpen működnek, éppen nem követik ezt az architektúrát, de hasonló CNN-eket használnak. Példa különbségeknek okáért az első verziós YOLO-val egyidőben kiadott SSD sem használt már teljesen kapcsolt réteget a kisebb számítási igény érdekében, a YOLOv2 innen inspirálódott. A MobileNet a kisebb erőforrásigény érdekében kevesebb réteggel rendelkezik, viszont emiatt pontosságot veszít. Bővebben [15][16][17][18]. Összességében ezek a technológiák mind ugyanúgy a CNN-ek deriválásaik, viszont mindegyikük más-más megközelítésekkel próbálja meg kihozni a legjobb eredményt.

2.2 YOLO objektumdetektáló algoritmusok

Szakirodalmi tanulmányozást legelőször a nyári gyakorlatom alatt kezdtem el, ekkor még az terv az volt, hogy egy gyomnövény felismerő programot kell készítsek. Ekkor ismerkedtem meg a YOLO (*You Only Look Once*) algoritmussal. A YOLO az egyik legnépszerűbb objektumfelismerő algoritmus, amint a legtöbb ehhez hasonló, CNN-ek (Konvolúciós Neurális Hálózat) felhasználásával működik. Egyszerű a használata, a tanítása, az alkalmazása. Emellett az algoritmus gyors, valós idejű működést tesz lehetővé, ami azt jelenti, hogy képes az objektumok felismerésére szinte azonnali. Készítettem erre még akkor egy egyszerűbb kis programot, mely videóról tudta felismerni a Százszorszép virágait, ez bizonyította az előbb említett tulajdonságait.

Amikor végleg eldöntöttem, hogy a jelenlegi projekttel fogok foglalkozni, a képfelismerő algoritmusok magokon való alkalmazásainak tanulmányozására fektettem legelőször időt. A tanulmányok között szintén rengeteg YOLO-s projekt volt. Konkrétan a rengeteg YOLO-t alkalmazó dolgozat közül alig lehetett találni más technológiákra összpontosító tanulmányokat. Arra a következtetésre jutottam, hogy mivel már amikor legelőször kipróbáltam a YOLO-t és saját szememmel láttam, hogy milyen pontos muszáj szerepeljen a dolgozatomban. A következőkben részletesebb kutatást végeztem magáról a YOLO algoritmusról, illetve alkalmazásáról a magfelismerés terén.

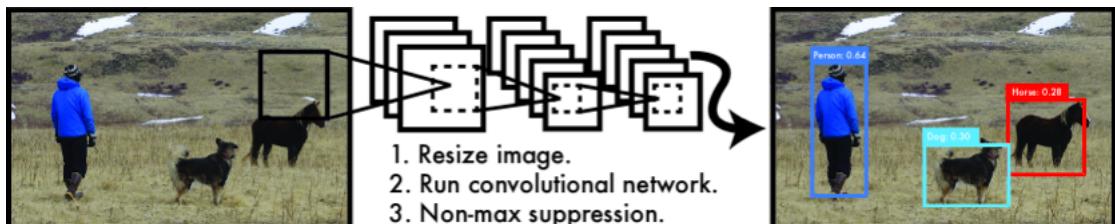
Az első YOLO 2015-ben jelent meg. Ez a verzió még nem ért el akkora ismeretséget és felhasználást, mint az utódai. Az első verzióban munkálkodó CNN 24 konvolúciós réteget és utána 2 teljesen kapcsolt réteget tartalmazott, ennek volt két hibája: a pontatlan pozícionálás, illetve más akkori módszerekkel összehasonlítva az alacsonyabb visszahívási arány, a második verzióban ezeket a hibákat kijavították [1]. Ezeknek ellenére kezdete volt egy “dinasztiának”, amely a mai napig uralja a képfelismerés iránt érdeklődők figyelmét. A 2.2.1-es ábrán látható, hogy 2016-tól 2021-ig hogyan tornászta fel magát ez az algoritmus a csúcsra a népszerűséget tekintve [1].



2.2.1. ábra A YOLO verziók népszerűsége [1]

Mint általánosan a neuronhálók tanításakor az adatok átméretezésre, normalizálásra kerülnek, majd ismétlődő tanítási ciklusok során a súlyok módosulnak a pontosabb prediktálás eléréséért [2]. A YOLO esetében a gyors és hatékony objektumdetektáláson van a hangsúly, ugyanis a You Only Look Once mondat egyszerűen annyit jelent, hogy egyszeri megnézése, egyszeri prediktálása egy bizonyos képkockának, ez jelentősen gyorsabb felismerési sebességhez vezet.

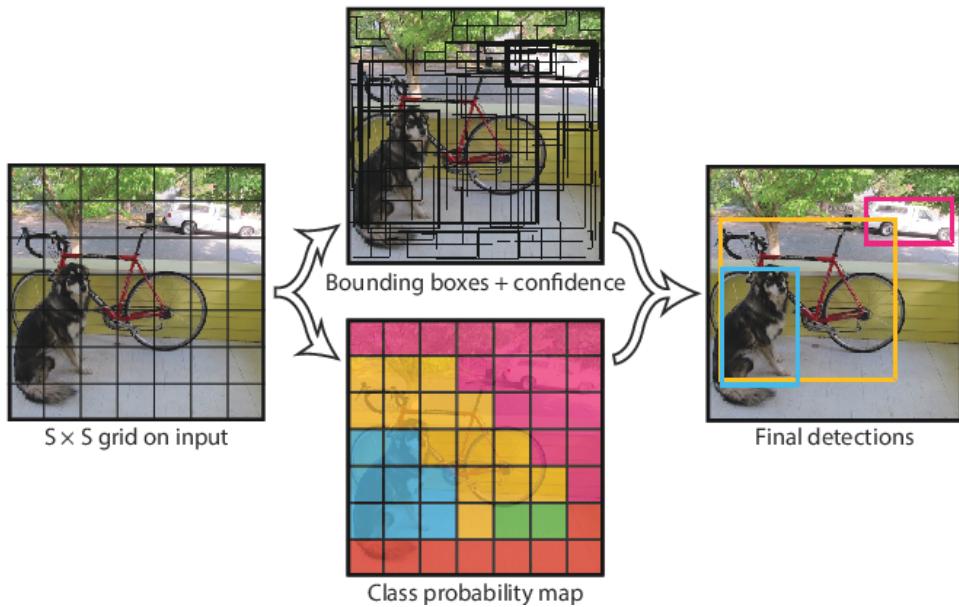
Az alábbi 2.2.2-es ábra szemléltet egy *bounding box* prediktálást. A YOLO végig megy a képkockákon, ezek átméreteződnek, végig haladnak egyetlen CNN-en, legvégén pedig egy *non-maximum suppression* (NMS) algoritmus, amelynek a lényege, hogy a duplikált, egymáson elhelyezkedő detektálásokat törli, csak a legvalószínűbbet tartja meg [2]. Eredményként pedig visszatérít detektálásonként egy bounding boxot (határoló doboz, keret), illetve hozzájuk tartozó osztályt és a detektálás konfidenciáját (Pontosságának arányát) [2].



2.2.2. ábra A YOLO képkockánkénti működése [2]

A következő 2.2.3-as ábrán látható, hogy mi történik egyetlen fotóval, amin objektumokat kell detektálni. A YOLO felossza a bemeneti képet SxS méretű gridekre (rácsokra), minden gridben lévő képkockára prediktál B bounding boxot, azoknak a konfidencia értékeit (azaz tartalmaz-e objektumot vagy sem), továbbá C darab osztály perditcióját (vagyis az a bizonyos objektum, ha van, akkor milyen valószínűséggel tartozik egyenként az osztályokhoz), ez a *grid-based* algoritmus [2]. Ezek a predikciók egy tensor-ban lesznek eltárolva, aminek a méretét a következő képlettel lehet kiszámolni [2].

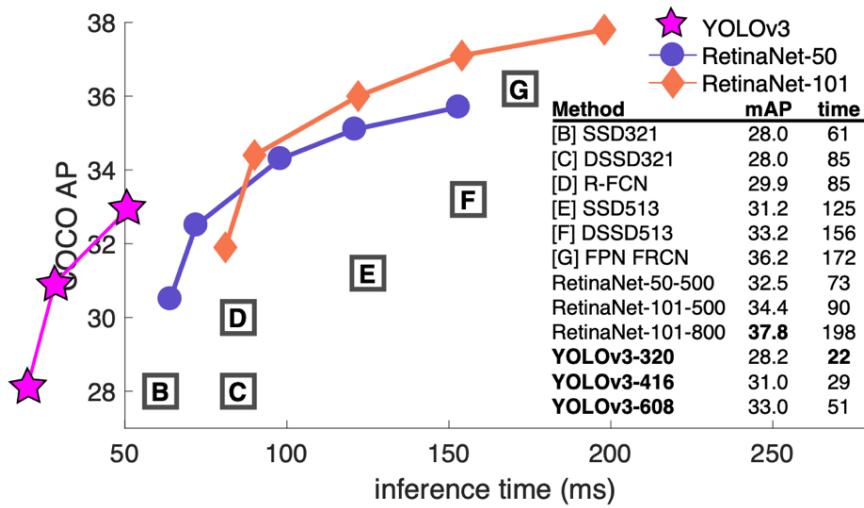
$$S * S * (B * 5 + C)$$



2.2.3. ábra A YOLO képfeldolgozása [2]

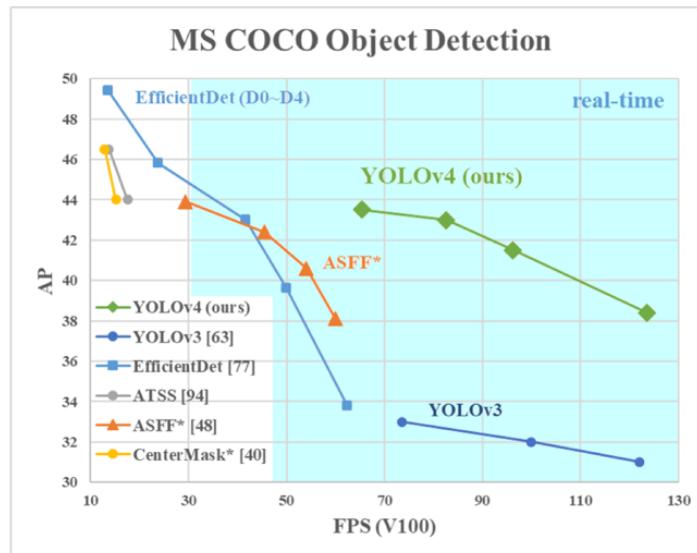
Az egyik legsikeresebb verziója a harmadik verzió (YOLO V3), amely már a fennebbi népszerűségi grafikonon is látszik, hogy kiemelkedik a többi közül. Ez a verzió nagy lépés volt az előző kettőhöz képest. Ámbár kicsit mélyebb hálót alkalmaztak ennél a verziónál, sikerült fejleszteni az alacsonyabb erőforrásigényű és hatékonyabb működést, ezek mellett ráadásul az eddigi leggyorsabb detektálási idővel rendelkezik és sokkal pontosabb az elődeihez képest [3]. A következő oldalon található 2.2.4.-ik ábrán egy grafikon látható, amelyen látszik, hogy a YOLO V3 elhagyja detektálási sebességen a 2018-as kortárs hálóit, hasonló átlag pontossági metrika mellett. Látható, hogy a “legkisebb” 320x320-as harmadik generációs YOLO háló a leggyorsabb 22 milliszekundumos sebességgel, viszonylag kicsi 28.2 mAP-s (*Mean Average Precision*) pontossággal [3]. A legnagyobb 608x608-as modell pedig ámbár nagyon jó 33.0 mAP-s pontosságot produkált, mégis a töredék idejét futja a nála pontosabb vagy nagyjából megegyező pontosságú, de egyébként nagyobb RetinaNet, SSD, DSSD és FPN hálóknál [3].

A következő verzió a YOLO V4, amely már nem volt akkora változás a harmadik verzióhoz képest, viszont eredményekben igencsak előrelépett. Lénygében egy Alexey Bochkovskiy nevezetű fejlesztőnek és csapatának az implementációja a YOLOv3-ról. Kevés dolgot változtattak meg, inkább kiegészítettek részeket, például a YOLO V3 CNN-ét, a Darknet-53-at (53 konvolúciós réteg) megörökölte, épp a CSP-t (*Cross Stage Partial*) vezették be mellé, amely hatékonyabban kezeli az információ áramlását a hálózatban, így lett a háló neve CSPDarknet-53 [1].



2.2.4. ábra YOLO V3 összehasonlítása [3]

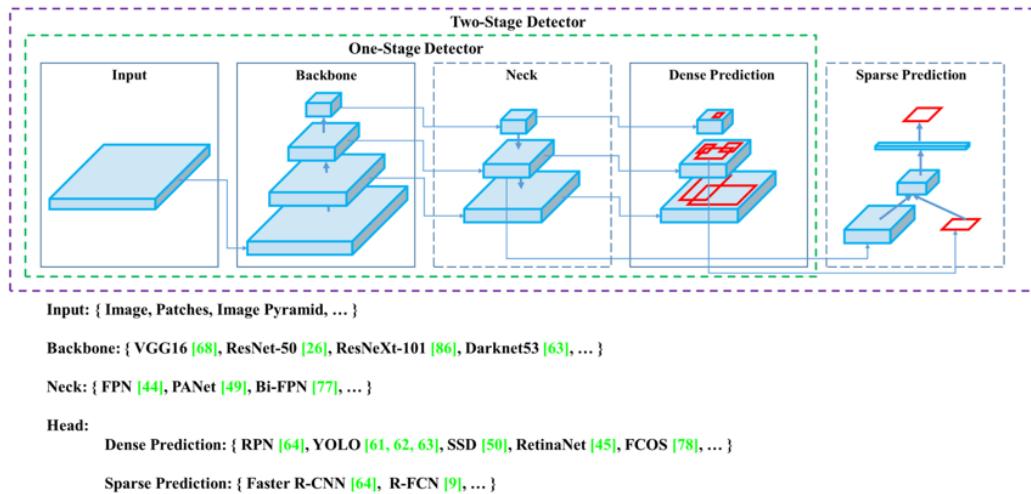
A YOLO V4 kb. 65 FPS-es (*Framerates Per Second*, Képkocka másodpercenként) sebességű valós idejű képfeldolgozással, egy NVIDIA Tesla V100-as grafikus gyorsítóval 43.5% AP-s (*Average Precision*) pontosságot tudott elérni [4]. A következő 2.2.5.-ik ábrán az



2.2.5. ábra YOLO V4 összehasonlítása [3]

figyelhető meg, hogy a YOLO V4 kétszer gyorsabban fut (FPS), mint a EfficientDet nevű háló, mindenmellett hasonló pontossággal [4]. Ez a verzió az előző harmadik verzióhoz képest 10% javulást hozott az AP (Avarage Precision, átlag pontosság) és 12% javulást az FPS szempontjából [4].

A modell szoftverarchitektúrája is megváltozott, ugyanis itt már beszélünk *Input* (bemenet), *Backbone* (gerinc), *Neck* (nyak) és *Head* (fej) részkről, ezek a részek egyenként magukba foglalják a hálózat bizonyos rétegeit, továbbá különböző feladatokat látnak el a bemenet feldolgozása és prediktálása, illetve a tanulás során [4]. Az Input felel a kép vagy képek helyes bemenetéért, a Backbone felel a képek jellemzőinek kinyeréséért, a Neck a Backbone egy kiegészítő része, amely a PAN (*Path Aggregation Network*) segítségével összefűzi a különböző mélységű feature map-eket a Backbone-ból, végül a Head pedig a predikciók készítéséért felel [4]. Az alábbi 2.2.6.-ik ábrán láthatóak a YOLO V4 modellnek a részei, megfigyelhető, hogy ezeknek a részeknek az együtteséből, ezeknek sorozatának, együttes munkájuknak köszönhetően jöhetsz létre a *Dense Prediction* (sűrű predikció), majd a *Dense Prediction*-ből pedig a *Sparse Prediction* (ritka predikció), amelyek együttesen kétrépcsős detektálást tesznek lehetővé (Two-Stage Detector) [4].



2.2.6. ábra YOLO V4 architektúrája [4]

A következő verzió a YOLO V5, 2020-ban jelent meg [5]. Az ötödik verziónál több hangsúlyt fektettek az erőforrás optimalizációra, mint az előző verzióknál [1]. Itt adtak ki legelőször több méretű modellt, ezek lettek a *nano*, a *small*, a *medium*, a *large* és az *xlarge*, ennek következtében a modell méretének a beállítása flexibilissé vált [1]. Következtethető, hogy a nevek azt tükrözik, hogy a méretek a könnyű (*lightweight*) nagyságtól (nano, small) a hatalmas, 493 réteggel és majdnem 100 millió paraméterrel rendelkező *xlarge*-ig terjednek [5]. Ez volt az első verzió, amelyet az Ultralytics nevű csapat fejlesztett, javarészt Glenn Jocher, a

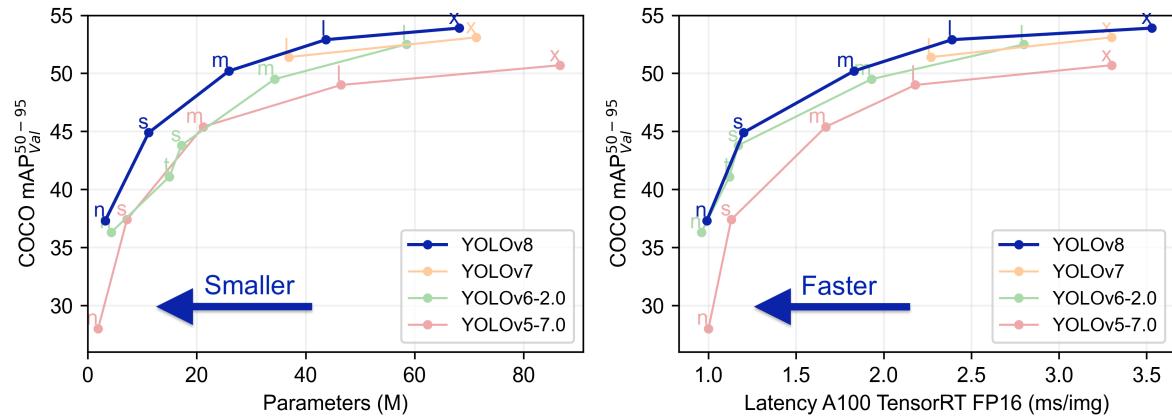
csapat alapítója. Ez a csapat kifejlesztett egy keretrendszer is, amelyről a **Felhasznált technológiák** alfejezet alatt lesz szó. Az Ultralytics által hozott újdonságok közé tartozik a már említett keretrendszer (amely CLI-ként és Python SDK-ként is elérhető), a méret szerinti skálázás, de továbbá a teljesítményt és sebességet növelő új funkciók is, mint például a horgonydobozok (*anchor boxes*) eltávolítása [5]. Fontos még megemlíteni azt, hogy ez volt az első verzió, ahol már előre-tanított (*pre-trained*) modelleket is lehetet használni [5]. Ezek lényegében finomhangolt (*fine-tuned*) modellek, ami azt jelenti, hogy a súlyokat nem a nulláról kell tanítani, ezzel a tanítás gyorsabb, a tanítás eredménye jobb. Ez a verzió volt az addigi, és minden napig az egyik legtöbb helyen megjelenő detektáló algoritmus, legyen szó telefonos applikációkról vagy más szoftverekről. Ezt a keretrendszer által nyújtott könnyű kezelhetőség és persze a sebesség és pontosság tette lehetővé.

A következő 6-os verziót a Meituan nevezetű kínai internetes vállalatnak a Gépi látás osztálya (Vision AI Department) fejlesztette ki ipari alkalmazások céljából 2022-ben [6]. A fejlesztések közül, hogy egy párat megemlítsék: A Bidirectional Concatenation Module (BiC), magyarul a „Kétirányú összefűzés modul” a *neck* részben helyezkedik le, felerősítve a lokalizációs jeleket, segítségével jobb teljesítmény érhető el elhanyagolható idő degradációval [6]. Példaként egy másik újítás az új Self-Distillation Strategy, vagyis az „Öndesztillációs stratégia” [6]. Ez különösen a könnyebb hálókat segíti, amelyeket én is tanítok, mivel segíti ezek teljesítményét és sebességét azzal, hogy fokozza a regressziós ágakat a lelassulás elkerülése érdekében.

A hetedik verziót ugyanúgy a már említett Alexey Bochkovskiy és csapata fejlesztette ki szintén 2022-ben, a YOLOv5 fejlesztéseként, viszont mivel a már említett Meituan verziót egy hónappal korábban kiadták, ennek már a hetes verziónevét adták. Fontos újítások voltak a re-parameterization (újra-paraméterezés), amely több számítási modult képes egyesíteni, így magasabb sebességre tesz szert [7]. Továbbá egy másik újítás volt a *coarse-to-fine lead guided label assignment* (durvából-finommá irányított címke bejelölő), amely megoldotta a dinamikus címke bejelölés problémáját (*Dynamic Label Assignment issue*) [7].

Mivel mind a hatodik és hetedik verzió a YOLOv5-nek a továbbfejlesztett verziói más-más csapatok által, így az eredmények is nagyjából fej-fej mellett vannak. A YOLOv6 2023-as legfrissebb dokumentációjában [6] ugyan jobban teljesít a hatodik verzió a hetediknél, viszont teszem én azt hogyha a hetedik verzió egy újabb dokumentációjában is történt volna egy összehasonlítás a hatodikkal, szerintem ott pedig a YOLOv7 jött volna ki a nyerőnek. Viccet félretéve az Ultralytics tesztelései alatt [8][9] a hálók hasonló pontosságokat és időket értek el.

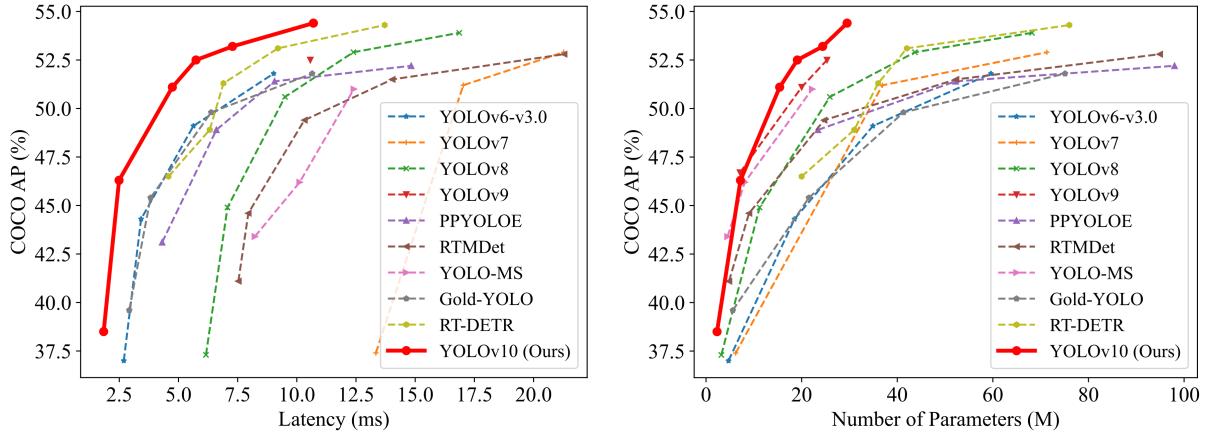
Mindezek után az Ultralytics továbbfejlesztett verziója következett, amely nem más, mint a YOLOv8, amelyet már 2023 elején adtak ki. Ez a verzió lett az addigi legjobb YOLO modell, mivel a fejlesztő csapat már ismerte az ötödik verzió két eddigi továbbfejlesztési próbálkozását. Egy minden eddiginél gyorsabb és pontosabb modell lett. Átdolgozták az architektúrát, mint a *backbone* és *neck* részeket, optimalizálták a sebesség pontosság fordított arányosságát (ezalatt azt értem, hogy minél nagyobb a sebesség annál kisebb a pontosság és fordítva) [10]. Továbbá egy minden eddiginél nagyobb előre-tanított (pre-trained) készletet hoztak létre, amely már nem csak objektum detektáló algoritmus modelleket tartalmazott YOLO név alatt, hanem megjelentek más feladatokat ellátó modellek is, mint például az osztályozás vagy a szegmentálás. A következő oldalon található 2.2.7.-ik ábrán látható a YOLOv8 összehasonlítása az előjeivel, látható a sebesség növekedés, illetve a *nano* változatok megközelítőleg ugyanannyi paraméterrel rendelkeznek.



2.2.7. ábra YOLO V8 összevetése [10]

A következő verziók már idén jelentek meg, amelyek nem mások, mint a kilenc és tizedik verziók. A YOLOv9 egy kísérleti verzió, amely a PGI és GELAN technológiákat implementálta [11]. Ez a verzió még mindig fejlesztés alatt áll, továbbá a könnyű modell méretek nem voltak elérhetők a tanítás alatt.

A YOLOv10 a legfrissebb elfogadott verziója a YOLO-nak, amelyet a már említett Ultralytics alapító Glenn Jocher a kínai Tsinghua egyetem hallgatói kutatócsoportjának segítségével fejlesztett. Fontosabb újítások például a *non-max suppression* (NMS) algoritmus kitörlése, amely már az első verzió óta a YOLO része, vagy akár a teljesítmény növelése érdekében bevezetett *large-kernel convolutions* (nagy rácsú konvolúció) és a *partial self-attention* (PSA), amely optimalizálja a számítási komplexitást és csökkenti a memóriahasználatot [12]. következő oldalon a 2.2.8.-ik ábra szemlélteti az összehasonlítást.



2.2.8. ábra YOLO V10 összevetése [12]

2.3 Egyéb objektumdetektáló algoritmusok

Miután befejeztem a YOLO utáni kutatást elkezdtem keresgálni más potenciális objektumdetektáló algoritmusok után. Felfedeztem a SSD-t, a Faster R-CNN-t, a MobileNet-et, az EfficientDet-et, illetve a régebbi ShuffleNet-et és SqueezeNet-et. Ezek közül a Faster R-CNN-nel készült dolgozatot sikerült találnom, ez ma is az egyik leg pontosabb modell, de egy lassabb, nagyobb erőforrású igényű, tehát nem ideális valós idejű felhasználásra, inkább már elkészült képről történő felismerő alkalmazásokra használják [13]. A SqueezeNet-et, ShuffleNet-et, EfficientDet, MobileNet hálók gyorsak, viszont a pontosságuk a modern YOLO mellett eltörpül. Ezeken kívül még felfedeztem az RT-DETR-t [14], amelyet dedikáltan valós idejű objektumfelismerésre fejlesztették, de nem ajánlott telefonon használni, mivel több rétegű háló, így nagyobb erőforrás igényű.

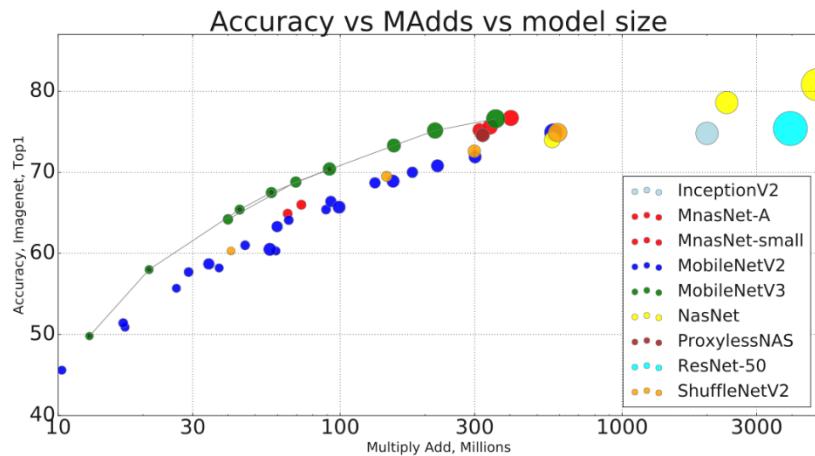
Mégis annak érdekében, hogy legyen amivel összevessem a YOLO hálókat úgy döntöttem, hogy két másik háló típust fogok tanítani. Hogy miért pont ezeket, azt a következőkben megindokolom: Az egyik a MobileNet lesz, mivel hogy a Google fejlesztette telefonokra és ugyebár Android (szintén a Google-é) okostelefonos szoftver a végeredmény. A másik pedig az SSD lesz, mivel ez az egyik legrégebbi objektumdetektáló algoritmus, amely jelentős hatást gyakorolt a mai objektumfelismerésre. Azért szeretném, hogy benne legyen, mivel ez egy viszonyítási pont, hogy valójában mennyit is fejlődött ez a tudományág.

Akkor kezdeném az idősebbel. Az SSD (Single Shot MultiBox Detector) egyidőben jelent az első generációs YOLO hálóval. Ahogyan a neve is tükrözi, a lényege, hogy egyetlen alkalommal halad végig a bemenetként adott fényképen (single shot), illetve a kép fölé több dobozt helyez el (multibox), amelyek körül határolhatják az objektumokat [15]. A YOLO-hoz hasonlóan ez az algoritmus is NMS-t (Non Max Supression) használ az egymás felett

elhelyezkedő dobozok kiiktatására [15]. Mivel egyetlen egyszer halad végig a bemeneten, ezért gyors, a pontossága az a tényező, amelyre kíváncsi leszek.

A MobileNet hálózatokat a Google fejleszti és határozottan okostelefonon történő gépi-látásra vannak tervezve. Az első MobileNet-et 2017-ben adták ki, állításuk szerint amiatt, mert egy egyre csak növekvő igény van jelenleg kisebb és hatékony neurális hálózatok fejlesztésére [16]. Az akkor “*trendek*” azok voltak, hogy minél komplikáltabb és mélyebb hálókra összpontosítani a nagyobb pontosság elérése érdekében, viszont ezek egyre és egyre erőforrás-igényesebbek is lettek [16]. A legelső hálókhöz, vagyis a MobileNetV1-hez könnyű (*lightweight*) CNN-eket használtak, az ezeken belüli konvolúciós rétegek pedig mélységi konvolúciós műveleteket (*depthwise convolution*) hajtottak végre [16]. A mélységi konvolúció röviden annyit jelent, hogy a bemenet minden csatornájára külön-külön konvolúciós műveleteket végzünk, ellenben a hagyományos párhuzamos konvolúcióval, ahol ugye egy meghatározott szűrő halad végig a képen. Egy másik változtatás, amely elősegítette az erőforrásokkal hatékonyabban “*bánást*” az a hiperparaméterek (*hyper-parameters*) - mint például a tanulási ráta, batch méret, tanulási ciklusok stb. - csökkentése volt [16].

A második verziója a MobileNet-nek két fontos újítást hozott, ez a *Linear Bottlenecks* és az *Inverted Residuals*, ezek az információáramlásban és a hatékonyabb erőforrás kezelésben segítettek [17].



2.3.1. ábra MobileNetV3 összehasonlítása [18]

A legutóbbi verzió a harmadik [18], fontosabb újítás volt, hogy megörökölte a MobileNetV2 alapján megtervezett MnasNet fejlesztéseit, mint például a *lightweight attention* (könnysűsűlyú figyelő) modulokat, illetve a *Squeeze-and-Excite* figyelő mechanizmust [18]. Ezenkívül az egyik legfontosabb újítás az volt, hogy a MobileNetV3 már nemlineáris *Swish* aktivációs függvényt használ [18]. A fennebbi 2.3.1.-ik ábra azt mutatja, hogy a MobileNetV3

mennyivel lett jobb, mint például a második verziós MobileNetV2, a MnasNet, a ShuffleNetV2 stb. [18].

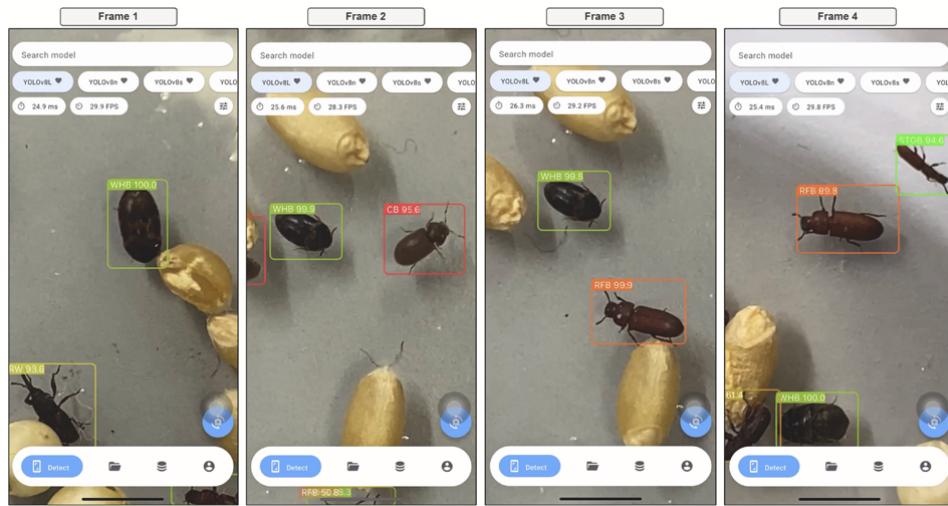
2.4 Ismert hasonló alkalmazások

Az egyetlen hasonló alkalmazás, amelyről szót szeretnék ejteni az Ultralytics okostelefonokra fejlesztett applikációja, melynek a neve Ultralytics HUB [19]. Ennek az applikációnak a lényege az, hogy ki lehet próbálni benne a csapat által fejlesztett verzióknak a legtöbb méretét, tehát a YOLOv5-öt és YOLOv8-at a *nano*, *small*, *medium* stb. méretekben. Ezenkívül pedig a saját általunk tanított YOLO hálókat is importálni lehet és futtatni. Ez az applikáció úgy van fejlesztve, hogy fel tudja használni mind az okostelefon processzorának, mind a grafikus kártyájának az erőforrásait (illetve Snapdragon csipek esetén a Hexagon gyorsítót), ezáltal növelve a hatékonyságot. Ezt az applikációt letöltöttem az okostelefonra, amelyen az én szoftveremet is fejleszteni fogom. Ez a telefon egy Huawei P40 Pro, amelynek a specifikációit a **Részletes tervezés** fejezet alatt fogom kifejteni. Béta verzió ellenére szerintem nagyon jól futott, az alábbi 2.4.1-es ábrán egy képernyőfotó látható az alkalmazás működéséről, ahogy a YOLOv8n, tehát *nano* méretű hálóval majdnem 20 FPS-es képfrissítéssel, körülbelül 40 ms-os sebességgel ismert fel különböző objektumokat.



2.4.1. ábra Ultralytics HUB [19]

Egy megvalósítás a fenti programmal egy kutatás [20] céljából rovarok felismerésére tanított YOLOv5 és YOLOv8 modellek importálása az Ultralytics HUB-ra. Az eredmények ez az alkalmazás esetén kiválóak lettek. 6 rovarfajra átlagos 0.5 mAP pontosságok sikerült elérni a minden verzióból a *small*, *medium*, *large* méretek tanításainál. Az iPhone 11-es (nagyjából az én Huawei eszközömmel megegyező erősségű hardver) telefonon átlag 30 FPS képförissítéssel futott. Az alábbi 2.4.2-es ábrán látható ennek az alkalmazásnak a felülete.



2.4.2. ábra Rovarfelismerő hálók importálva az Ultralytics HUB-ra [20]

2.5 Felhasznált technológiák

Az első lépés a program megvalósításához a címkézés volt. Címkézéshez a CVAT (Computer Vision Annotation Tool) online címkéző programot használtam [22]. Egy modern felülettel rendelkező, egyszerű használatú, magas minőségű programról beszélünk, amelynek bő eszközkészlete van kezdve a szegmentálástól, több alakú keretezésekig egészen addig, hogy a tanítóhalmazokat több típusként lehet exportálni (COLO, YOLO stb.).

A tanításokhoz két keretrendszer használtam. Elsőként az a már említett Ultralytics keretrendszer [23] parancssoros felületén (*CLI, Command Line Interface*) kezdtem kísérletezni a tanításokkal. Ezt a felületet a Python csomagkezelővel, a pip-el (*Preferred Installer Program, Preferált Telepítő Program*) lehet telepíteni az Unix alapú rendszerekre, mint a Linux és a MacOS. Használata rendkívül egyszerű, egyetlen parancsra van szükség, amelynek paraméterekként kell megadni a tanítani kívánt háló típusát, a tanítóhalmaz konfigurációját, valamint a további hiperparaméterek (pld. tanítási ciklusok száma, batch méret, tanulási ráta).

A másik tanításhoz használt keretrendszer az OpenVINO training extensions. Az OpenVINO [24] az Intel vállalat által fejlesztett szoftver, egy eszközkészlet, amelyet azzal a céllal hoztak

létre, hogy a *deep learning* (mély gépi tanulás) modelleket hatékonyabban lehessen optimalizálni és telepíteni (*deploying*). 2021-ben ennek a szoftvernek külön kiadtak egy tanítási eszközöket, tanítási programokat tartalmazó bővítményt, ez egyszerűen a *training extensions* [25] nevet kapta, amely annyit angolul, hogy tanítási bővítmények. Telepítésével voltak némi bajlódások, problémák, már nem ment annyira gördülékenyen, mint az Ultralytics-é, ezeket a **Részletes tervezés** fejezet alatt jobban kifejtem. Miután sikeres lett a telepítés, a CLI használata már hasonló, mint az Ultralytics-é.

A tanítás után a neuronhálók kezelésére, kimeneti tenorjaiknak feldolgozására, illetve formátumok közti konvertálásához a PyTorch-ot használtam [31]. A PyTorch egy optimalizált tenzor könyvtár, lényegében egy eszközkészlet, amely a *deep learning* neuronhálók tervezésére, létrehozására, tesztelésére stb. nyújt egy átláthatóbb és egyszerűbben kezelhető keretrendszeret Python nyelv alatt. Mind az Ultralytics és OpenVINO keretrendszerek erre a technológiára épülnek. Az Android fejlesztés során is a PyTorch Mobile disztribúcióját használtam [32].

Tanítani a témavezetőm magas teljesítményű laborgépein tanítottunk, Dockert [26] használtam a tanítási környezet konfigurálásához és mozgatásához a rendszerek között. A Docker egy szoftver, amely a konténerizációs technológiára alapszik (*containerization technology*). Ez a technológia lehetővé teszi, hogy konténereknek nevezett környezetekbe szoftvereket tudunk konfigurálni, amelyeket majd operációs rendszertől és más rendszerspecifikációtól függetlenül több gépen is futtatni lehet.

Az okostelefonos applikáció az Android Studio [27] fejlesztői környezetben lett megírva. Ezt a környezetet a JetBrains orosz csapat fejlesztette, ugyanúgy, mint a hozzá tartozó Android SDK (*Software Development Kit*) szoftver fejlesztői készletet, és a működéslogikát leíró Kotlin [28] nyelvet is. Az Android SDK lényegében egy keretrendszer, amely összezárja a logikát a grafikus felülettel. A grafikus felület dizájnleíró nyelve az XML, ám a dizájnelemeket az Android SDK könyvtárai biztosítják. A Kotlin egy modern nyelv, nagyobb hatással rá a Java és a C# nyelvek voltak. A Kotlin Java bájtkódra (vagy JavaScript) fordul, mivel pedig az Android operációs rendszer támogatja a JVM-et (*Java Virtual Machine*) vagyis a Java virtuális gépet (ez futtatja a bájtkódot), így Androidon lefut a program. Azért döntöttem úgy, hogy Android Stúdiós applikációt készítetek, mivel ezen év első félévében erről részletesebben tanultam, projektet is kellett készítsek belőle.

Az magok méreteinek kiszámítására referencia objektumokat fogok tanítani. Ez a referenciaobjektum egy 2.5cm x 2.5cm-es szürkére satírozott négyzet. Elméleti szinten ugyebár a négyzet területe 6.25 cm^2 lesz, amely a fényképen le fog fedni egy bizonyos pixel

számot. A magok fehér hátteren szintén le fognak fedni egy bizonyos pixel számot. Ebből először is viszonyítással ki fog lehetni számolni, hogy mekkora területet foglal el egy mag. A magok méreteiről, tömegükiről, illetve térfogatukról előzetes méréseket fogok végezni, illetve az ezekből a mérésekből majd arányszámokat fogok számolni fajonként, abból kiindulva, hogy területfoglalásukhoz képest milyen arányban változnak ezek a tulajdonságaik. A **Részletes tervezés** fejezetnél a gyakorlati formát bővebben tárgyalni fogom.

3 Részletes tervezés

3.1 Első fázis: Tanítóhalmaz

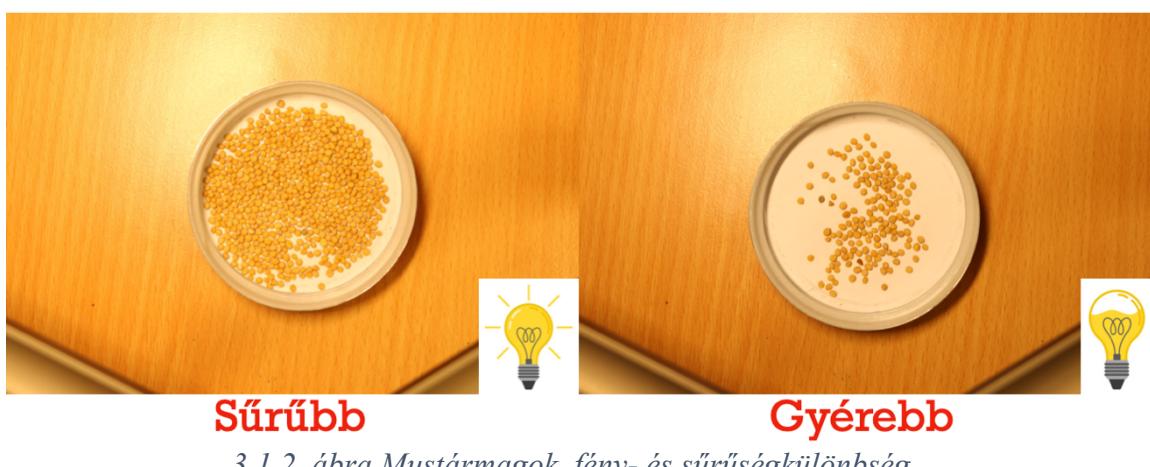
A tanítóhalmazhoz a kultúrnövénymagokat a Sepsiszentgyörgyi tanulmányi központtól kaptuk. 50 magot kaptunk, ebből választottunk ki Kelemen Tamás Kertésmérnök szakos hallgatóval egyetemben a legismertebb és leggyakrabban előforduló fajokat. Eredetileg 25 fajt választottunk ki és fényképeztünk le. A fényképezéshez egy Canon EOS 70D [29] fényképezőgépet használtunk a magas minőségű fényképek érdekében. Összesen 1522 fénykép készült. A fényképezésnél három paramétert vettünk figyelembe: a fókusztávolságot, a fényviszonyokat, illetve a magok sűrűségét a területen. A cél az volt, hogy fajonként készítsünk kb. 60 fényképet, ezekből 20-20-20 darabot fókusztávolságoként, a 20 fényképen belül pedig 10 darabot sűrűbb magszámmal a területen, 10 darabot pedig gyerebb magszámmal a területen. Három fókusztávolságon készítettünk fényképeket: 5.6, 4.5 és 3.5. A fényképeket két különböző helyen készítettük: az egyetem 308-as laborjában, illetve a bentlakás tanulóiban. A laborban készült fényképek 5.6f fókusztávolságon, vagyis maximális közelítésen készültek minden a 25 fajról. Ezek a fényképek még 3648×2432 -es “közepes” felbontáson készültek. A másik két fókusztávolsággal a tanulókban készítettük a képeket, ezeknél már a fényképfelbontást felhúztam a legmagasabb 5472×3648 -ra. Az alábbi 3.1.1-es ábrán a tavaszi árpáról szemléltetek három képet, a három fókusztávolsággal.



3.1.1. ábra Tavaszi árpa három távolságban

A különböző fényviszonyok eléréséhez egy egyszerű asztali lámpát használtunk, amelynek 3 különböző fényerősségbeállítása volt. Egyetlen probléma, ami adódott az a kamera képfeldolgozó szoftvere volt, amely minden módon javítani próbálta a fényképezésnél lévő fényviszonyokat. Ez azt eredményezte, hogy függetlenül attól, hogy milyen fényerősséget állítottam be a lámpának, ez a szoftver úgyis ugyanolyan színeket egyensúlyozott ki. Kikapcsolni ezt csak a fényképezőgép manuális módjában lehetett és viszont akkor pedig az autófókusz sem működött, amely ahhoz, hogy hamarabb haladjunk, ne legyenek homályos képek, elengedhetetlen volt. Ennek ellenére sikerült más-más képviszonyú képeket készíteni, valamilyen okból kifolyólag amikor kevesebb magot helyeztünk el fényképezésre a képek sötétebbek lettek.

Két különböző mennyiségű magot helyeztünk el a kicsi fehér hátterű tálacskába. Először sűrűn raktuk a magokat, arra gondolván, hogy majd a program az egymáshoz közeli, esetleg egymáson lévő magokat is külön-külön fel tudja ismerni (persze helyes címkézés következetében). Másodjára pedig gyerebben helyeztünk el magokat, hogy a háttéren egyedülálló példányokról is legyenek minták. A következő 3.1.2-es számú ábrán a mustármagról látható két fénykép, a két sűrűségről és két fényviszonyról.



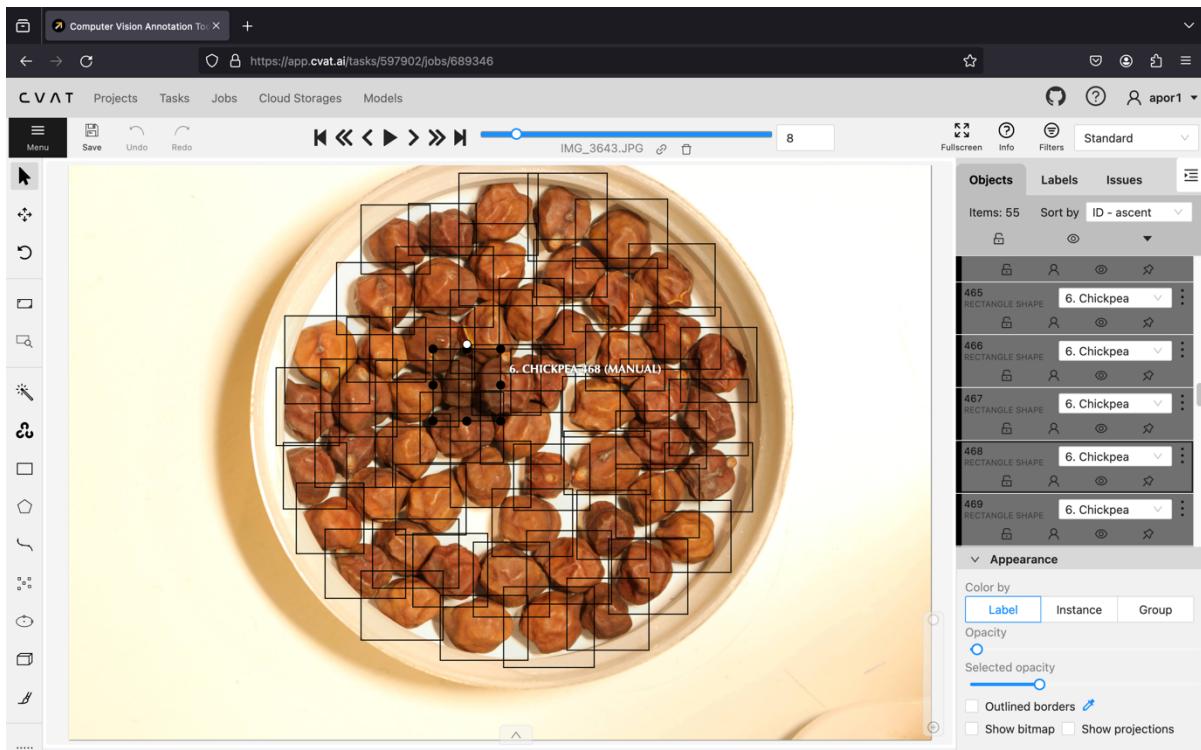
3.1.2. ábra Mustármagok, fény- és sűrűsékgükönbség

Amint a fényképezést befejeztük, a rendezésük következett. Átnéztem az összes fényképet, a duplikátumokat töröltem. Ezután fajok szerint, fajokon belül pedig fókusztávolság szerint rendszereztem a képeket.

Miután ez megvolt, a címkézés következett. Tisztában voltam, hogy az egyetemnek van egy saját licenszű CVAT programja, eleinte ezt próbáltam. Az volt ezzel a baj, hogy felhasználóként csak nagyon kevés adatot lehetet feltölteni, ekkor váltottam a közösségi verziójára (*Community version*), amely ingyenes volt. A CVAT-ról a **Felhasznált technológiák** alfejezetnél már beszéltem. Az egyetlen bökkenő ennél a verziótól viszont az volt,

felhasználónként csak maximum 2 projektet lehetett létrehozni, projektenként pedig 5 taszkot, vagyis összesen 10 taszkot. Mivel, hogy végül 22 osztályom volt, így kénytelen voltam három felhasználót létrehozni, hogy a biztonság érdekében ne kelljen címkézéseket töröljek, a címkék a CVAT adatbázisában is maradjanak meg másolatként.

Végül 21 magfajt címkéztem fel és tanítottam, plusz a referencianégyzetet. Amiatt csak ennyit a 25-ből, mivel 4 pár magfajta nagyon hasonlított egymásra. Utólagosan jöttem rá, hogy szabad szemmel nem lehet őket megkülönböztetni, így a neuronháló is összetévesztette volna őket, mivel a fényképek között se talált volna különböző tulajdonságokat. A magfajok, amelyeket végül felhasználtam (zárójelekben melyik hasonlásait nem): búza (tritikálét nem), bíborhere (mohar nem), tavaszi árpa (őszi árpa nem), mustár (sárga magvú köles nem). A következő 3.1.3-as ábrán 5.6 fókusztávolságú, magas példányszámú csicseriborsókról látható egy felcímkezett fénykép a CVAT környezetben, Mozilla Firefox böngésző alatt.



3.1.3. ábra Csicseriborsó címkézése

Megfigyelhető továbbá az előző ábrán, hogy a kurzor jelenleg egy olyan címkén található, amelyben a mag egy másik mag alatt van, nem teljesen látszik. Majdnem mindegyik fajnál címkéztem ilyen eseteket is.

Rengeteg időt töltöttem azzal, hogy minél több képet felcímkezzek, hogy minél több mintám legyen. Fajonként körülbelül két órát vett igénybe, hogy hozzávetőlegesen 2000 példányt felcímkezzek, nem beszélve az exportálásról, javításról. Az exportálást YOLO-ban végeztem, viszont mivel ezt csak CVAT projektenként lehetett, így a címkék indexei nullától

ötig lettek beállítva minden fajra. Egy Python szkripttel javítottam, hogy minden osztály nullától huszonegyig a megfelelő indexet kapja meg. Ezután a tanítási és validációs adatok különböző választása következett. Fajonként kb. 300 validációs mintát céloztam meg, de mivel a címkék fényképenként voltak állományokba szervezve, így van amelyik fajnál kevesebbet, van amelyik fajnál többet sikerült elhatárolnom.

A legelső faj, amelyet felcímkeztettem a bíborhere volt, ekkor még azon az elven voltam, hogy minden egyes magot a fényképen fel akarok címkézni. Körülbelül hat órát töltöttem ennek a fajnak a címkézésével, több mint hétezer mintám gyűlt belőle. Következőkben már kiszabtam egy hozzávetőleg 2000 minta per faj határt. A másik “kakukktojás” a lóbab, ez volt a méretben legnagyobb magfaj, amelyről fényképeket készítettünk. Mintáinak száma csak 725 lett mind 60 róla készült fénykép felcímkezésével. A többi maggal igyekeztem tartani a 2000-2500 közötti mintaszámot. A következő *3.1.1-es táblázatban* a tanítóhalmaz fajai láthatók neveikkel és mintaszámaikkal, amelyek külön-külön tanítási és validációs mintaszámokra is le lettek bontva.

Név	Összes minta	Tanítási	Validációs
00. Referencia	35	35	a tanítási adatok
01. Bíborhere	7015	6088	927
02. Kifejtő borsó	2809	2390	419
03. Takarmány borsó	3654	3232	422
04. Velős borsó	2231	1891	340
05. Cukorrépa	3320	2892	428
06. Csicseriborsó	2371	2015	356
07. Hajdina	3079	2682	397
08. Lófogú kukorica	1519	1277	242
09. Puhaszemű kukorica	1758	1545	213
10. Lóbab	725	657	68
11. Szója	2400	2082	318
12. Őszi búza	2463	2115	348
13. Szöszös bükköny	2301	1906	395
14. Napraforgó	2524	2193	331
15. Zab	2116	1814	302
16. Murok (Sárgarépa)	1493	1300	193

17. Mustár	2064	1762	302
18. Olajretek	2064	1732	332
19. Rozs	2079	1744	335
20. Tavaszi árpa	2013	1699	314
21. Szudáni cirokfű	2126	1815	311
Összesen	52159	44866	7293

3.1.1. táblázat A növénymagfajok mintáinak számai

3.2 Második fázis: Tanítókörnyezet kialakítása

Az elsődleges cél az volt, hogy minél nagyobb sebeséggel tudjon a tanítás lefolyni, ehhez viszont mindenkor szükség volt arra, hogy videókártyán fusson (GPU). Az első probléma az volt, hogy az én Apple MacBook Air laptopom nem rendelkezik dedikált videókártyával. Megpróbálkoztam az Ultralytics által támogatott PyTorch MPS-el (*Metal Performance Shaders*: egy API, amely lehetővé teszi, hogy feladatok fussanak le az Apple chipek beépített videókártyáján), de ez a sima processzoron (CPU) való futástól nem mutatott hatalmas fejlődést a sebességen, a dedikált videókártyán történő futás sebességétől még nagyon le volt maradva, ráadásul nagyon felmelegítette a laptopomat, amely mellesleg aktív hűtéssel nem rendelkezik, kitudja mennyire volt közel a katasztrófa.

Alternatíva után kutakodva először a Google Colab felületén próbáltam meg tanítani, amely ugyan egy Jupiter Notebook szolgáltatás, vagyis jegyzetkészítő, rendelkezik Python kompilátorral és kód szerkesztővel, ráadásul hozzáférést biztosít GPU-s számítógépes erőforrásokhoz ingyenesen. Nem is akármilyen számítógépek szolgáltatják ezeket az erőforrásokat, hanem olyanok, amelyek NVIDIA Tesla T4-es *deep learning* gyorsítókkal rendelkeznek. Feltöltöttem a tanítóhalmazt, bekonfiguráltam egy YOLO tanítást a Colab-ban, valóban a tanítási sebesség a leggyorsabb volt, amelyet eddig láttam. A csattanó az egészben az volt, hogy az ingyenes szó alatt azt értették, hogy 100 “számítási egység” ingyenes, utána viszont a továbbiakért fizetni szükséges. 100 számítási egység alatt a YOLOv5 esetében a 50 ezres tanítóhalmazzal 13 ciklusnak sikerült lefutnia, ami előtt az ingyenes erőforrásszolgáltatás határt átléptem és a Colab leállt.

Végül a laborgépeken történt a tanítást, itt Dockert használtam egy egységes tanítókörnyezet kialakításához. Az első próbálkozás az MacBook-on történt. Lefuttattam egy konténert a sima ubuntu22-es image fájljal. Erre feltelepítettem a Pythonot, a PyTorch-ot és az

Ultralytics-et. Amikor viszont az Openvino került volna feltelepítésre, kompatibilitási hiba történt. A gépben egy Apple M1-es ARM architektúrájú chip dolgozik, az Openvino viszont csak a hagyományos x86-os architektúrát támogatja. Néhány próbálkozás után ennek a kikerülésére abbahagytam a laptopomon a munkát, helyette a 309-es labor egyik számítógépén folytattam. Ekkor jöttem rá, hogy a Docker-en a Linux olyan processzorarchitektúrára konfigurálódik, amelyen futtatjuk, tehát ha Macen sikerült is volna lefuttatni az Openvino-t, az image akkor sem futott volna a x86-os architektúrájú gépeken.

A laborgépen már sikeresen fel lehetett telepíteni az Openvino-t is, viszont szintén kompatibilitási hibák keletkeztek a Docker-en történő futtatás miatt. Végül az Openvino-nak a Dockerre készített ubuntu22-es fejlesztői image-ét futtattuk (ubuntu22_dev), beállított videókártya hozzáéréssel, amelyre már előre fel volt telepítve az Openvino és konfigurálva volt a Docker környezetéhez. A training extensions-t még külön fel kellett telepíteni, néhány Linux csomag hiányzott telepítéskor, ezeket miután kipótoltam és elindult a tanítás. Erre ezek után feltelepítettem még külön az Ultralytics-et. Az Ultralytics feltelepítése után a YOLO hálók tanítása sikeresen megindult a videókártyán, viszont az Openvino elhasadt. A probléma abból adódott, hogy mivel az Ultralytics a feltelepítésnél frissítette a CUDA (*Compute Unified Device Architecture*: eszközcsomag, amely lehetővé teszi az NVIDIA videókártyákon a történő programfuttatást) eszközökészletet és a PyTorch-ot, a CUDA és PyTorch legfrissebb verziója pedig már nem volt kompatibilis az Openvino-val. A problémát a régebbi verziók újra telepítésével kellett megoldani. Ezekután már mind a két keretrendszeren sikeresen lehetetett tanítani.

3.3 Harmadik fázis: Neuronhálók tanítása

Mivelhogy valós idejű felismerés volt a cél okostelefonon, tanításra a YOLO-k esetében az ötödik verziótól errefelé a nano modelleket használtam, mivel ezek a leggyorsabbak és a legerőforráskímélőbbek. Ezenkívül az Openvino tanítási keretrendszerből (*training extensions*) pedig az eredeti MobilNetV2 háló, illetve az SSD alapú MobileNetV2 háló került tanításra. A MobileNet hálókból amiatt a második verziót tanítottam, mert az Openvino által támogatott Model Zoo-nak nevezett modellkönyvtárból csak ez a verzió volt elérhető objektumdetektálásra.

Próbálkoztam Alexey Bochkovskiy Darknet keretrendszerét is feltelepíteni, hogy a YOLOv4 és YOLOv7 modelleket is tanítsam, viszont folyton hibákat dobott telepítésnél. Nagyon sok időt vett igénybe utánajárni a hibák megoldásának, 2 napot töltöttem vele, a legfelzaklatóbb dolog pedig az volt, hogy amint egy hibát sikerült kiküszöbölnöm, újabb dobta

fel a fejét, amely az előző hibának a szöges ellentéte. Valószínűleg a Docker volt ezért is a felelős.

A YOLOv6 esetében pedig letöltöttem a Meituan keretrendszerét a GitHubról, sikeresen telepíteni is tudtam. A YOLOv6, mint ahogy már szó volt róla, nem kifejezetten telefonokra volt fejlesztve, viszont a Meituan git repository átböngészésével találtam egy pár hálókonfigurációt, amelyek a YOLOv6Lite nevet viselték. Ez a Lite disztribúció már kifejezetten telefonos alkalmazásra volt fejlesztve, egy próbatanítást el is indítottam a YOLOv6Lite-S mérettel. A tanítási ciklusok rendkívül lassan indultak el és futottak le videókártyán, talán lassabban, mint a Mac-en. Az S (small) legkisebb modellméreten is egy éjszaka alatt 30 tanítási ciklus futott le, ezután pedig semmilyen hibakezelési üzenet nélkül leállt a tanítás.

Végül úgy döntöttem, hogy csak az Ultralytics és OpenVino keretrendszereket fogom használni. Összeségében a tanított modellek tehát az Ultralytics keretrendszer alatt: YOLOv3u (Ultralytics implementáció), YOLOv5n, YOLOv6n (Ultralytics implementáció), YOLOv8; Az OpenVino keretrendszer alatt pedig: ATTS_MobileNet_V2 és SSD_MobileNet_V2. A következő 3.3.1.-es táblán felsoroltam a használt YOLO-kat és ezek tulajdonságait.

Modell neve	Bemenet mérete	mAP	Sebesség GPU (ms)	Paraméter szám (millió)
YOLOv3u	640x640	43.3	4.8	61.9
YOLOv5nu	640x640	34.3	1.06	2.6
YOLOv6n	640x640	37.5	0.84	4.7
YOLOv8n	640x640	37.3	0.99	8.7

3.3.1. táblázat A YOLO hálók specifikációi

Megfigyelhető a fenti táblázatban, hogy a YOLOv3-nak a legnagyobb az mAP (*Mean Average Precision*, Átlagos közép-pontosság, az abszolút P pontosság és az osztályok számának a hányadosa) metrikája, vagyis ez a legfontosabb modell. De mivel ez a legnagyobb (több, mint tízszer több paraméterrel rendelkezik átlagosan), ezért majdnem ötször lassabb sebességet mértek rajta egy NVIDIA A100-as videókártyával, mint a többin.

A tanítást a YOLO-kal kezdtük. A hatodik verziót kívül az összes verziót finomhangolt súlyokkal tanítottam, amelyeket már előre szolgáltat a keretrendszer. A finomhangolás azt jelenti, hogy a tanítandó neuronháló már előre tanítva van egy általánosabb tanítóhalmazra, így a súlyok nem “nulláról” kezdik a tanulást, hanem már az előre betanított háló súlyait használják

kiindulópontnak. Ezáltal a tanítás erőforráskímélőbb is lesz, továbbá nem vesz annyi időt igénybe, hogy a háló egy bizonyos pontosságot elérjen, tehát az elején gyorsabban is tanul. A tanítási parancs a YOLO esetében az Ultralytics csomag pip-el történő feltelepítése után a *yolo train*. Ezt a parancsot többszörösen szervezem bele sorban egy *bash* fájlba, hogy egymás után az összes verzió tanuljon. A lentebb található 3.3.1.-es ábra szemléltet egy részletet ebből a *bash* fájlból.

Megfigyelhetők a paraméterek. A *model* paraméter megadja, hogy milyen modellt szeretnénk tanítani, ebben az esetben éppen a harmadik verziós YOLO-t. A fájlnév, amit megadtam *.pt* kiterjesztésű, ez a PyTorch-nak a súlyállomány kiterjesztése, tehát azt indikálja, hogy a modell már tanított, a *yolov3u.pt* a v3-as hálókonfigurációján kívül tartalmazza a finomhangolt súlyok értékeit is. A v6 verzió esetében, mivel ennek nincs finomhangolt verziója, így a megadott fájl *.yaml* kiterjesztésű, amely csak a neuronháló konfigurációját tartalmazza. A második paraméter a *data*, a paraméterként megadott fájl pedig a tanítóhalmaz konfigurációs fájl. A *config.yaml* állomány tartalmazza a tanítóhalmaz útvonalát, elválassza külön a tanítási és validációs adatokat, illetve kilistázza az osztályok neveit, YOLO esetében

```

1 #!/bin/bash
2
3 # Train parameters
4
5 trainingPath="/home/data/train/netframeworks/ultralytics/training"
6 outputPath="/home/data/train/netframeworks/ultralytics/runs"
7 imageSize=640
8 epochsEach=150
9 deviceTrain="0"
10 batchSize=16
11
12 # YOLOv3u.pt (ultralytics implementation pre-trained) training
13
14 yolo train model=$trainingPath/yolov3u.pt \
15           data=$trainingPath/config.yaml \
16           imgsz=$imageSize \
17           epochs=$epochsEach \
18           batch=$batchSize \
19           device=$deviceTrain \
20           project=$outputPath
21

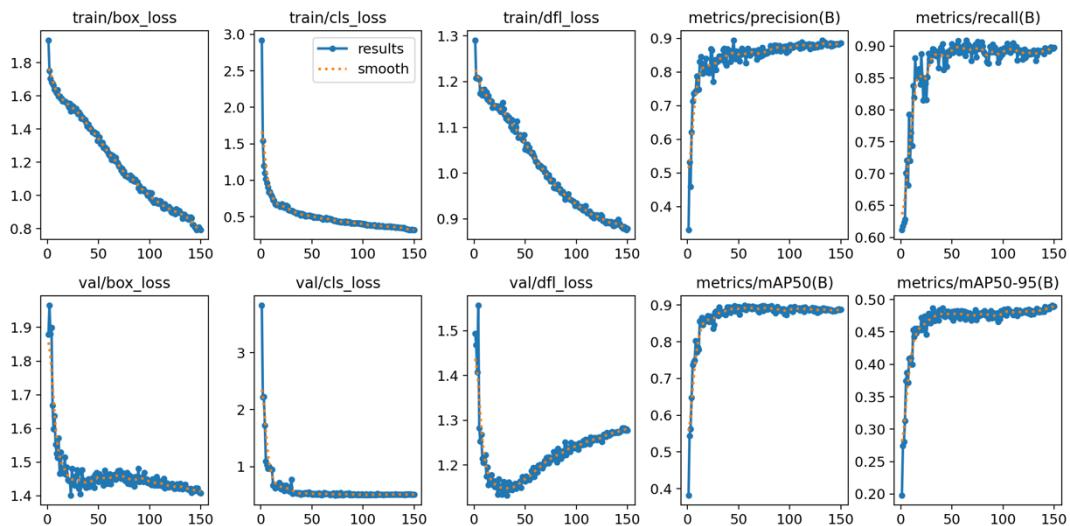
```

3.3.1. ábra A *trainyolos.sh* részlete VS Code alatt

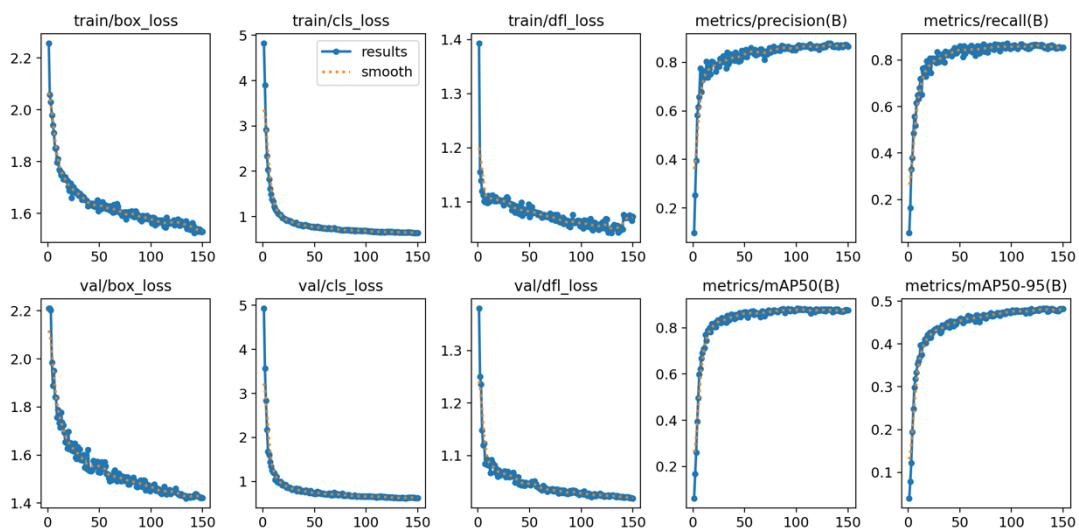
az osztályindexek szerint. A következő paraméter az *imgsz* megadja a bemeneti fénykép méretét, amely szerint a bemeneti tenzor mérete fog alakulni, a YOLO-k esetében az alapértelmezett 640x640 pixel a képarány. A következő paraméter az *epochs*, a tanítási ciklusok számának beállítására használjuk. Az összes YOLO 150 cikluson keresztül lett tanítva. A *device* paraméter megadja, hogy milyen eszközön fog folyni a tanítás, ha az értéke szám az azt jelenti, hogy videókártyán tanul, ha az értéke “*cpu*”, akkor pedig processzoron tanul. Ennek az értéke jelen esetben 0, ami azt jelenti, hogy a tanítás a 0-ás számú videókártyán fog futni. Az utolsó előtti paraméter a *batch*, ahol a *batch* méretet adjuk meg, mivelhogy nagy

tanítóhalmazzal folyik a tanítás a batch mérete nagyban befolyásolja a tanítás sebességét. Az utolsó használt paraméter a *project*, ez megadja a tanításnak, hogy milyen útvonalra mentse az eredményeket.

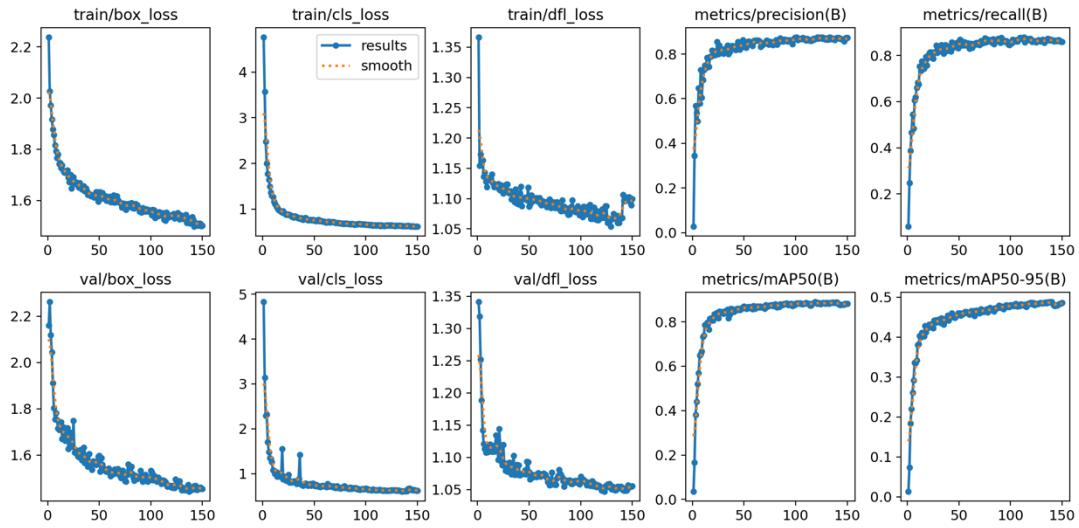
A tanítások verzióinként körülbelül 3-4 órát futottak 150 ciklust, a tanítások végére pedig a hatodik verziótól kívül sikerült elérni a 85-90%-os pontosságot. Mivel a YOLOv6 konfiguráció nem volt finomhangolva, ezért az első tanítási ciklusból csak 54%-os pontosságot sikerült elérnie. Ennek az eredménynek a súlyaival tanítottunk tovább még 150 ciklust, így már a YOLOv6-nak is sikerült elérni egy kb. 80%-os pontosságot. Az alábbi 3.3.2.-es, 3.3.3-as, 3.3.4.-es számú ábrákon láthatóak a tanítások során alakuló metrikák diagrammjaik a v3, a v5 és a v8 esetében. Mivelhogy a hatos verzió kétszer tanult 150 ciklust, így minden tanítás pontosságának alakulásának a 3.3.5-ös ábrán szemléltetem.



3.3.2. ábra A YOLOv3 metrikáinak az alakulásai

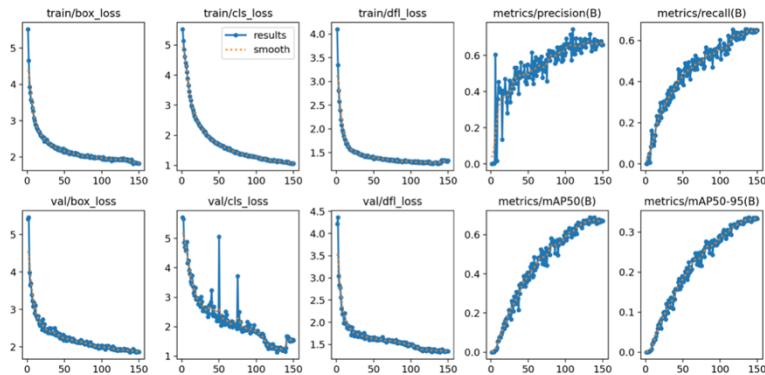


3.3.3. ábra A YOLOv5 metrikáinak az alakulásai

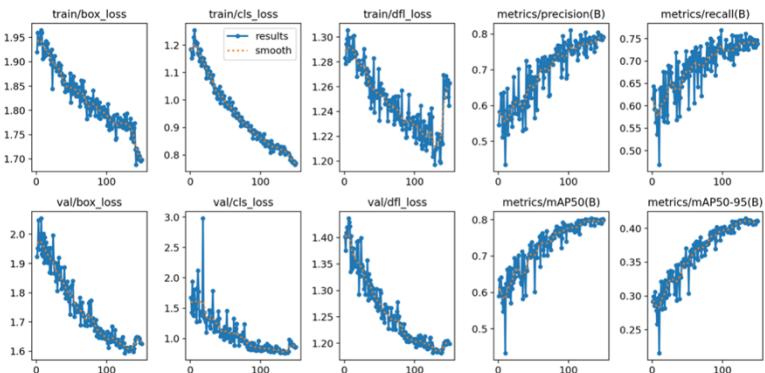


3.3.4. ábra A YOLOv8 metrikáinak az alakulásai

1. Tanítás



2. Tanítás



3.3.5. ábra A YOLOv6 metrikáinak alakulásai két tanítás során

A legjobb százalékos pontosságok (*Precision(B)*) verzióinként:

- YOLOv3: 89.484%
- YOLOv5: 87.915%
- YOLOv6: 81.104%
- YOLOv8: 87.431%

Az Openvino keretrendszer esetében a tanítás kicsivel nehezebben indult el. Ahogy már említettem, probléma akadt az instalálással, utána az tanítóhalmaz konfigurációját teljesen át kellett konvertáljam COCO formátumra. Ezek csak a kisebb problémák voltak, a nagyobbak a keretrendszer rossz dokumentációjából adódtak. Nem voltak egyértelműek a tanítóhalmaz konfigurációs fájlok, a model konfigurációk, de végük sikerült minden megoldani.

Ahogyan már említettem az Openvino tanítási bővítményéből, a *training extensions*-ből két modellt tanítottam végül. Az egyik az eredeti MobileNetv2 háló a másik az SSD-s MobileNetv2. A következő 3.3.2.-es számú táblázatban a két MobilNetv2 specifikációi vannak megjelenítve.

Modell neve	Bemenet mérete	mAP	Sebesség CPU (ms)	Paraméter szám (millió)
MobileNetv2	932x736	22.1	200	4.3
SSD_MobileNetv2	864x864	Nincs infó	Nincs infó	4.475

3.3.2. táblázat A MobileNet hálók specifikációi

Az MobileNetv2-SSD annyiban különbözik a “sima” MobileNetv2 hálótól, hogy az utolsó fénykép osztályozó réteg az SSD hálóból van átvéve. Ebben a hálóban úgymond egyesül a kellemes a hasznossal, mivel ugyebár tudott, hogy az SSD az egyik leggyorsabb algoritmus, viszont a nagyobb erőforrásigényé miatt nem ideális mobiltelefonon használni valós idejű felismerésre. Viszont így, hogy a direkt erőforráskímélésre fejlesztett MobileNet-el össze van építve igencsak sok esélyét látom annak, hogy sebességen ez lesz a nyerő. A hátránya viszont az SSD-nek a pontosság, ez már a tanításnál is meglátszott, amelynek eredményeit kissé lentebb szemléltetem.

A fenti táblázatban sajnos az SSD-s változatról nem igen találtam pontossági, illetve sebességi fejlesztői eredményeket. Az alap verzió esetében pedig meg kell jegyeznem, hogy a fejlesztők dokumentációjából vett eredmények közül a sebességet processzoron mérték (CPU), illetve a pontosság is nem IoU (*Intersection of Union*, a valós doboznak, illetve a prediktált doboznak egy arányszáma, amely megadja, hogy mennyire pontos a prediktált doboz) érték felett megadott mAP (mint a YOLO-knál, ugye 50-95), hanem általános, így kisebb. Ezt amiatt mondok el, hogy mivel más mértékegységek szerintiek a specifikációk, összehasonlítni őket majd csak a tesztelési eredményeknél fogom.

Tanítani először az alap MobileNetv2 modellt tanítottuk. A következő 3.3.6.-ik ábrán látható a tanítási *bash* fájlon belüli *otx train* parancs és a megadott paraméterek. Nem szeretném

```
1 #!/bin/bash
2
3 epochs="150"
4 data_path="/home/data/train/datasets/coco_dataset/"
5 mobilenet_path="/home/openvino/training_extensions/src/otx/recipe/detection/atss_mobilenetv2.yaml"
6 ssd_path="/home/openvino/training_extensions/src/otx/recipe/detection/ssd_mobilenetv2.yaml"
7 gpu_id='1'
8
9 # ATSS MobileNetv2 train
10
11 otx train --data_root $data_path \
12     --config $mobilenet_path \
13     --max_epochs $epochs \
14     --gpus $gpu_id
15
```

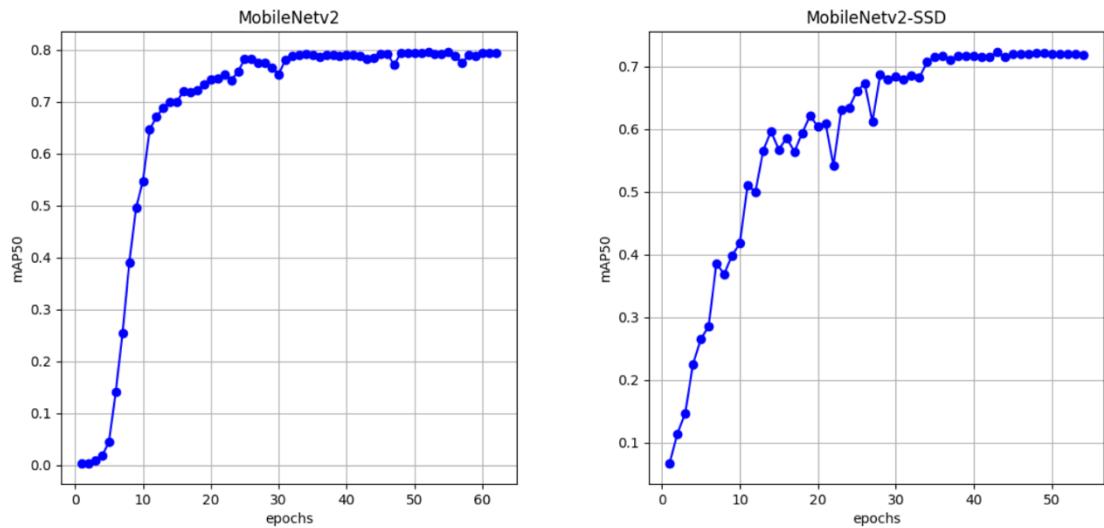
3.3.6. ábra A *trainopenvino.sh* részlete VS Code alatt

hosszasan részletezni újra a paramétereket, ugyanazon paramétereket adtam meg, mint a YOLO-nál, annyi különbséggel, hogy a *batch* mérete itt alapértelmezetten 8, illetve a bemente mérete is más mind a két modellnél. A tanítást ugyanúgy 150 ciklusra indítottuk el, viszont ez automatikusan leállt 41 ciklus után. Először azt gyanítottuk, hogy valami hiba miatt állt le a tanítás, ezért a már eddig tanított *checkpoint* súlyértékekkel elindítottunk egy tovább tanítást. Ez a tovább tanítás is 21 ciklus után leállt. Az eredményeket végig nézve jöttem rá, hogy valójában egy pontossági határt ért le kb. 80%-nál, így a keretrendszer mivel észlelte, hogy a pontosság nem ér el nagyobb értékeket leállította a tanítást.

Az SSD-s esetében pedig az első tanításnál maximum 72.329% pontosságot ért el, az 52-ik ciklusnál szintén leállt a tanítás. Ezt az alacsonyabb pontosságot furcsálva egy újabb tanítást indítottunk az SSD-vel. Ez a tanítás már 64 ciklust futott, viszont ekkor is a maximális pontosság még 0.1%-al kevesebb lett, mint az első tanításnál, tehát végül az első tanítás eredményeit használtam.

Az Openvino keretrendszer esetében nem lett mérve ciklusonként az abszolút pontosság (P , a helyes predikcióknak és az összes predikciónak a hányadosa), mint az Ultralytics esetében, illetve diagrammok sem készültek a metrikákról. Talán amelyik metrika értéke a legközebb áll a P -hez az az **map50** nevezetű metrika. Az **map50** az az Átlagos középpontosság érték, amely az 50% feletti IoU (*Intersection of Union*) értékekkel rendelkező detektálásokból számolódik. Ezek az értékeket a *matplotlib* nevű Python csomag segítségével rajzoltam ki diagramba. Mivel az alap MobileNetv2 tovább lett tanítva (0.5%-al pontosabb is lett ennek következtében), ezért a két tanítás ciklusainak és **map50** értékeit összevontam, az

SSD-nél viszont csak az első tanítás eredményeit foglaltam diagrammba. A következő 3.3.7.-ik ábrán a MobileNetv2 és a MobileNetv2-SSD **mAP50** pontosságának a változása látható.



3.3.7. ábra A MobileNetv2 hálók mAP50 metrikájának az alakulása

A legjobb százalékos mAP50 a két verzióból:

- MobileNetv2: 79.563%
- MobileNetv2-SSD: 72.329%

3.4 Negyedik fázis: Az alkalmazás implementálása

Az eszköz, amelyen az alkalmazást fejlesztettem, illetve teszteltem egy 2020-ban gyártott csúcstelefon, a Huawei P40 Pro. Az okostelefon fontosabb specifikációit a következő 3.4.1.-es táblán szemléltetem.

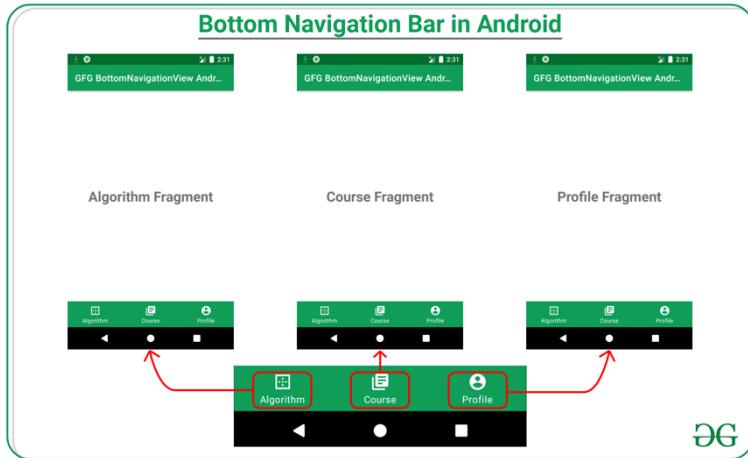
Huawei P40 Pro	
Chipset	Kirin 990 5G Octa-core (7 nm+)
RAM	8 GB
OS	Android 12 alapú EMUI 12.0
Kamera	Leica optics 50 MP, f/1.9, 23mm, 1/1.28", 1.22µm, dual pixel PDAF, OIS

3.4.1. táblázat A Huawei P40 Pro specifikációi

3.4.1 Felhasználói felület (UI)

Az alkalmazás implementálását a *UI pattern* (minta) kiválasztásával kezdtem az Android Studio-ban. Az alkalmazásomat egy *Bottom Navigation Menu Activity*-re építettem fel, amelyhez az Android Studio a projekt létrehozásánál generál egy sablont. Android Studio-

ban általában az *Activity*-k a főbb UI alkotóelemek, ezek pedig összezárhatnak *Fragment*-eket, amelyek angolul töredéket jelentenek, tehát UI töredékek. A *Bottom Navigation Menu Activity* (Alsó navigációs menü *Activity*) egy olyan összetett felületi elem, amelynek a lényege, hogy a képernyő alsó felében folyton látható opciói általában egy-egy *Fragment*-re irányítanak át. A következő 3.4.1.-es ábrán egy példát szemléltetek egy ilyen *Activity*-re.



3.4.1. ábra Bottom Navigation Menu Activity

Első lépések kímélyítettem az alkalmazás színét, illetve opcióként csak kettőt hagytam meg. A Camera opciót, illetve a Details (részletek) opciót, amelyeknek külön ikonokat állítottam be. Ez a két opció átirányít a nekik megfelelő *Fragment*-ekre.

3.4.2 A YOLO-k beillesztése az Androidos alkalmazásba

Legeslegelőször az ONNX Runtime nevezetű keretrendszernek Androidos verziójával próbálkoztam meg, mivel tudtam, hogy a YOLO modellek súlyfájljait könnyen át lehet erre a formátumra konvertálni. Ezenkívül az ONNX-nek van az egyik legjobb modelloptimalizálása, ugyanis ezen formátum alatt akar háromszoros CPU sebességet is el lehet élni az Ultralytics adatai szerint a YOLO-kal.

Első lépésként tehát átkonvertáltam a *.pt* kiterjesztésű tanítás utáni súlyfájlokat *.onnx* formátumba. Ehhez az Ultralytics keretrendszer *yolo export* parancsát használtam, amelynek csak meg kell adni a konvertálandó súlyfájlt, illetve a formátumot, amelybe konvertálni szeretnénk. Ekkor hiba adódott a konvertáló által használt *2onnx* pip csomaggal, amelynek a megoldásával egy pár órát megszenvedtem, de végül meglettek az ONNX formátumú YOLO modelljeim. Ezekután sikeresen leimplementáltam az ONNX Runtime felhasználásával egy függvényt, amely megnyit egy ilyen ONNX modellt, illetve elhelyez a bemeneti tenzor méretű képet (esetünkben 640x640), a függvény viszont hibásan futott le. Kiderült, hogy a felhasznált példa Android program, amelyet az ONNX Runtime oldalán adtak meg valójában

osztályozásra (Image Classification) és nem objektumdetektálásra adott példa. Androidon történő objektumdetektálásra sem az ONNX oldalán, sem más forrásokból sajnos példát nem találtam.

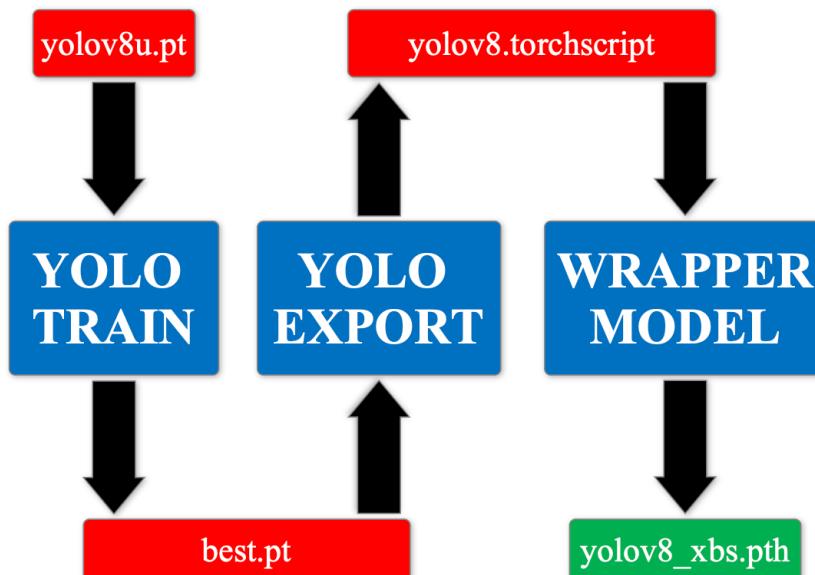
Más Androidon futó neuronhálókezelő keretrendszer után kellett néznem. Ekkor tudatosult bennem, hogy mind az Ultralytics, mind az OpenVINO modelljei PyTorch-ban vannak felépítve. Utánakeresés után fedeztem fel a PyTorch Mobile és PyTorch Lite [32] nevezetű okostelefonra fejlesztett keretrendszereket. A PyTorch Lite-nak telefonra optimalizált (PyTorch függvény) *.ptl* kiterjesztésű súlyfájlra volt szüksége. Ezt a *.ptl* kiterjesztést egy *.torchscript* formátumból lehetett átalakítani egy pár PyTorch-os függvény használatával Python nyelvben, *.torchscript* formátumra való átalakítást pedig a *yolo export* nevű parancs támogatja. A PyTorch Mobile-nak viszont már volt példa programja Androidos Objektum Detektálásra [33], eszerint implementáltam le a bemeneti fénykép átalakítását bemeneti tenzornak, illetve ennek a feldolgozását.

A kimeneti tenzort végül egy modul átkonvertálja egy ugyanakkora méretű tömbbé, az adatokat pedig feldolgozza, amely annyit tesz, hogy a bounding box-okat az osztálymutatóval (*class index*) és az osztály konfidencia (*class probability*) egy *data class* adatstruktúrában eltárolja, majd visszatéríti. A problémák a futtatásnál keletkeztek, amikor is a bounding boxok a fénykép főátlójára rajzolódtak ki csakis, vagyis a kimeneti tenzor feldolgozása helytelen volt. Debuggolás közben akadt meg a szemem, hogy a kimeneti tenzor mérete nem akkora, mint amekkorát a tenzorfeldolgozó modul vár. Megváltoztattam a méretet, amelyre a modul számítson, a feldolgozásnál a dobozok szintén a fénykép főátlójára kerültek csak. A probléma a gyökerétől eredt a programnak, ugyanis a példaként megadott program adatfeldolgozása egy teljesen más kimeneti tenzor struktúrájú YOLO hálóra volt elkészítve.

Ezekután keresgélni kezdtem, hogy hogyan lehet az Ultralytics típusú YOLO-kat elhelyezni PyTorch Mobile segítségével egy Androidos programba. Hosszas keresgélés után találtam rá Stephen Cow Chau cikkére a medium.com weboldalon [34], amely meghozta a “megváltást”. Eme cikk szerint a *yolo export* parancs az Ultralytics tanítás utáni *.pt* kiterjesztésű modellfájlok ból *.torchscript* kiterjesztésre konvertálása során “levágja” a *Post-Process* (utólagos-feldolgozás) műveleteket, köztük a *Non-Max Supression* műveletet, illetve a tipikus **{left, top, right, bottom, score, class}** formájú tenzorrá való átalakítás műveletét. Ennek eredményeként kimenetként az összes bounding box kikerül, konfidenciától függetlenül, egy úgynevezett *pre-NMS* (Non-Max Supression előtti) formájú tenzor alakjában.

A cikk szerzője több lehetséges megoldást kipróbált a Post-Process “levágás” megelőzése érdekében, kezdve az eredeti Ultralytics-es *.pt* formátumú modell azonnali

PyTorch Mobile által támogatott formátummá való konvertálásától egészen egy külön *Non-Max Supression* implementálásáig és hozzárendeléséhez a modellhez, amely remélhetőleg nem vágódik le, de ezek minden hibákat generáltak a futtatásoknál. Végül a bevált módszer az lett, hogy a *Post-Process* műveleteket a *.torchscript* formátumhoz rendelte hozzá, egy kiegészítő *Wrapper* (csomagoló) modell létrehozásával. Ennek a *Wrapper* modellnek a lényege az, hogy bemeneti tenzorként megkapja a *torchscript* modell *pre-NMS* formátumú kimeneti tenzorát, majd ezen végrehajtja a *Non-Max Supression*-t, illetve a `{left, top, right, bottom, score, class}` formátummá való alakítást. Ezeket az értékeket pedig egyetlen `x` tömb típusú változóba téri ki vissza (mellette a **boxes**, illetve **scores** változókkal, de ezeket a változókat nem használtam), ezek a visszatérített értékek lesznek a *Wrapper* modellnek a kimeneti tenzorában. A folyamatnak a könnyebb átláthatósága érdekében a következő 3.4.2.-es ábrán tömbvázlatban is ábrázoltam a YOLOv8 esetében.



3.4.2. ábra Modellformátum átalakulási folyamata

Ez Wrapper model így háti tartalmazza a YOLO modellt, továbbá ennek kimeneteit utólagosan feldolgozza (*Post-Process*). Ez a Wrapper modell kerül exportálásra Python-ban *.pth* kiterjesztésben. A *.pth* kiterjesztésről eddig nem került szó, ez szintén a PyTorch egyik modellformátuma, amelynek kimeneti tenzora speciálisan beállított alakokat is felvehet, esetünkben ugyebár három, a bemenetek függvényében változó dimenziójú tömböt (**x**, **boxes**, **scores**). A *.pth* kiterjesztést a PyTorch Mobile (nem PyTorch Lite) értelmezni tudja, a három tömböt Kotlin-ba egy az egyben, feldolgozás nélkül megkapjuk. Az így megkapott bounding boxok végre helyesek lettek, helyesen rajzolódtak ki az objektumokra. A Kotlin oldalon a

feldolgozásnál ezeket az adatokat az PyTorch Mobile Objektum felismerés példájából vett adatsuktúrába rendeztem be, amely megkönnyítette a dobozok ábrázolását.

3.4.3 A valós idejű kamera megjelenítése és a bounding boxok kirajzolása

A példa applikációban találkoztam először a CameraX [35] nevű könyvtárral, amelynek a célja könnyebbé tenni a kamera használatát az Android fejlesztésben. Ennek a lényege az, hogy egy *Layout*-hoz (*Activity*-hez vagy *Fragment*-hez tartozó kimeneti XML fájl) tartozó *TextureView*-ra játssza át a folyamatos kameraképet. Az átátszott képkockákat (*frame*) figyelni (*listening*) lehet, illetve kezelni (*handling*). A CameraX képkocka kezelését (*frame handler*) az **ImageAnalysis** (fénykép analízis) nevű objektum biztosítja. Ennek az objektum lekéri a jelenlegi képkockát a megadott felbontással, majd ennek a kezelését az objektum **setAnalyzer()** metódusával lehet végrehajtani, a képkocka **ImageProxy** típusú. Ebben a metódusban hajtódnak végre képkocka átadása a neuronháló bemenetére, illetve a kimeneti adatok felrajzolása a *Layout*-ra (bounding box, osztály azonosító).

Több idő telt el a bounding box rajzolásoknak a megoldásával, mint a CameraX beüzemelésével, ugyanis a *TextureView*-ra rajzolni nem lehetett, illetve az **ImageAnalysis**-nek a módosított képkockáját nem lehetett kicserélni a CameraX *Preview* felületbiztosító elemében (azaz a *TextureView*-ban). Sok kutakodás és tesztelgetés után végül egy stackoverflow.com-os kérdésnek [36] válaszaként kaptam meg a megoldást. A megoldás egy *ImageView* elem elhelyezése volt a *TextureView* elem felett. Mivelhogy az *ImageView* széllességét és magasságát *wrap_content* értéknek állítottam be (akkora amekkora az elem, amelyet tartalmaz), így csak egy, a jelenlegi képkocka méretű üres fényépet kellet létrehozni, arra rajzolni, végül ezt átadni az *ImageView*-nak. Ez a megoldás tökéletesen működött, létrehoztam egy üres képkocka méretű **Bitmap** típusú változót, erre **Canvas** segítségével a kimenti bounding box-ok pixelei szerint rajzoltam a négyzeteket. Ezután a bitmap-ot elhelyeztem a *ImageView*-ra, amely értelmezni tudta és megjelenítette.

3.4.4 Fizikai tulajdonságok kiszámolása

Ugyebár a neuronhálóknak tanításra került egy szürke referencia négyzet is, amely felismeréskor elfoglal egy bizonyos számú pixelt. Ezt a négyzetet én tanításkor 2.5 centiméternek, teszteléskor pedig 1 centiméternek rajzoltam meg. Ugyebár felismeréskor, ha ennek a referencia bounding box-nak összeszorozzuk a két oldalát (vagy egyetlen oldalát négyzetre emeljük), akkor megkapjuk, hogy a négyzet hány pixelnyi területet foglal el. Ezekután, ha ezzel a pixelszámmal elosszuk az egy centiméteres négyzet területét (100 négyzetmilliméter), akkor kapunk egy arányszámot. Ezzel az arányszámmal, ha majd

beszorozzuk későbbiekben egy mag pixel területét, akkor hozzávetőlegesen megkapjuk, hogy a mag milyen területet foglal el négyzetmilliméterben. A következőkben szemléltetem ennek a számításnak a levezetését. A 3.4.3.-es ábrán a magnak, referenciajának, illetve ezek bounding box-ainak egy tömbvázlatát szemléltetem.



, ahol:

- **MagPA** - Mag pixelterülete
- **RefA** - Ref. területe (mm^2)
- **MagBBox** - Mag bbox-ja
- **RefBBox** - Ref. bbox-ja

3.4.3. ábra A mag, a referencia és bbox-ai tömbvázlata

$$\mathbf{RefPA} = (\text{OldalA}_{\text{RefBBox}})^2 \text{ px} \quad \text{vagy} \quad \mathbf{RefPA} = (\text{OldalA}_{\text{RefBBox}} * \text{OldalB}_{\text{RefBBox}}) \text{ px}$$

$$\mathbf{Ar} = \frac{\text{RefA}}{\text{RefPA}}$$

$$\mathbf{MagA} = (\text{MagPA} * \text{Ar}) \text{ mm}^2$$

, ahol:

- **RefPA** – A referencia pixelterületének KÖZELÍTÉSE (a bounding box és a referencianégyzet közötti rés valójában a detektálásnál nem lesz ennyire széles)
- **OldalARefBBox, OldalBRefBBox** – A referencia bounding box-nak az oldalai
- **Ar** – A terület arányszám (az angol *Area ratio*-ból)
- **MagA** – A mag területének a KÖZELÍTÉSE négyzetmilliméterben

A magok pixelterületét úgy számolja ki a program, hogy megszámolja a nem háttér színű pixelek számát a detektált bounding box-ban. A háttér lehetséges színeinek skálájának a felállításához a MacOS Digitális Színmérő szoftverét használtam. Készítettem néhány képernyőképet az alkalmazáson belüli kamera fényviszonyaival készülő képkockákról amint a háttér veszik. Ezeken a képeken látható hátteren mentem végig a színmérővel, ugyanúgy a sötétebb, mint a világos részeken. Mindeközben odafigyeltem arra is, hogy nehogy véletlenül valamelyik mag színe beletartozzon ebbe a skálába. A végleges háttér skála végül a szürke RGB(150, 150, 150)-től a fehér RGB(240, 240, 240)-ig terjed el az alkalmazásba.

Amint leimplementáltam a terület számítást, következett a magok osztályonkénti (fajonkénti) csoportosításának az implementálása, majd ezután az osztályonkénti területszámítás a Details (részletek) Fragment alatt.

A következőkben a tömeg számítása következett a kiszámított magok területe alapján. Méréseket végeztem a magok tömegeire grammban, egy három tizedesnyi pontosággal (milligrammos nagyságrend) mérő mérlegén. minden egyes magfajra 10 mérést végeztem, 10 különböző méretű (területű) példányon. Kivételt három magfaj képezett az apróságukból kifolyólag. A bíborhere, a mustár, illetve a murok magoknak méreteikből adódóan annyira kicsi a tömegük, hogy ezzel a mérleggel nem lehetet pontosan mérni őket példányonként (mikrogrammos nagyságrendűek). Akadtak még kisebb tömegű magok, amelyeknek az átlagos példányaik nem haladták meg a 100 milligrammot, például cukorrépa (átlag 32.2 mg), hajdina (átlag 21.5 mg) vagy az olajrétek (átlag 11.9 mg), de az ilyen magoknál esetlegesen nagyobb hiba a tesztelés során fog kiderülni. A terület arányszámot (Ar) ezeknél a méréseknél 10 referencianégyzet detektálásból átlagából számoltam ki a program, a pontosabb mag terület érdekében. A következő 3.4.2.-es táblázatban példákat szemléltetek ezekről a mérésekről a lófogú kukorica esetében.

Mag neve és mérés száma	Terület (mm ²)	Tömeg (gramm)
8. Lófogú kukorica 1	40.23889	0.211
8. Lófogú kukorica 2	53.38909	0.17
...
8. Lófogú kukorica 10	35.60658	0.186
Átlag	54.068868	0.2102

3.4.2. táblázat A lófogú kukorica tömegmérései

Kétféleképpen implementáltam a tömegbecslést. Az első módszer a mérésekből származó átlag terület, illetve tömegből származó arányszám használata. A második módszerben pedig egy terület szerinti növekvő skálát állítottam fel a mérésekből, majd az arányszámot abból a területből, illetve a hozzátartozó tömegből számoltam, amelyhez a detektált mérés területe a legközelebb esett. A második módszer könnyebb megértése érdekében vegyük a fenti táblázatot. Ha most például a detektált mag területe 50.5 mm² lenne, akkor az 53.38909 területet, illetve a hozzátartozó 0.17 tömeget venném, az arányszám pedig (0.17 / 53.38909) lenne, amelyet ha megszorzok a detektált mag területével megkapom a detektált mag tömegét (közelítés).

Utolsóként implementálta a magok méretének számítását a fénykép keresztátlói (függőleges, vízszintes) mentén. Ehhez a detektált referencianégyzet két oldalának az átlagát vettem megfeleltetve 1 cm-nek (10 mm), ez lett a hosszúság arányiszáma (**L_r**, az angol length ratio-ból). Az átlók mentén megszámoltam a nem háttér színű pixeleket és megfeleltettem ezt összeszoroztam az **L_r-rel**. Ezeket amikor meg szeretnénk tekinteni, akkor kirajzolódik a detektált mag fényképe, illetve az említett átlók a fényképen.

3.4.5 A MobileNet-ek Androidos beillesztésének kísérletei

Az Openvino a tanított modellek súlyértékeit *.ckpt* (PyTorch checkpoint) kiterjesztésű fájlba menti. Ez a fájltípus különbözik az Ultralytics esetében kimentett *.pt* fájltól, tehát nem lehet konvertálni *torchscript* formátumba, onnan pedig *.pth* formátumba, hogy Kotlin-ban a PyTorch Mobile számára értelmezhető legyen. A checkpoint fájl nem tartalmazza a modell konfigurációját, csak a súlyértékeket. Az Openvino tanítási bővítmény keretrendszer (*training extensions*) rendelkezik egy *export* parancssal viszont ez csak Openvino IR formátumba tudja konvertálni a *.ckpt* fájlt, így is csak megadott Openvino típusú modellkonfigurációval. A probléma ezzel a formátummal az, hogy csak szigorúan az Openvino keretrendszer ismeri fel, illetve semmilyen más formátumba (gondolok ONNX, PyTorch, Tensorflow stb.) hivatalosan nem lehet konvertálni.

Megoldást keresve találtam rá az *openvino2tensorflow pip* csomagra, amely elméletben lehetővé teszi Openvino IR formátum átalakítását Tensorflow formátumba. Az elméletem az volt, hogy hogyha már Tensorflow-ba sikerül átalakítani ezt a zárt formátumot, akkor a Tensorflow formátumú modellt már át lehet alakítani ONNX formátumba, az ONNX formátum pedig egyfajta univerzális formátum, vagyis PyTorch formátumba is tovább lehet konvertálni. Sajnos többszöri próbálkozás után ezzel a parancssal nem sikerült konvertálnom Tensorflow-ba. Először az Openvino IR formátumú *.xml* fájlba adott hibát a modell rétegjeinek és kimenetének a felépítésében, miután pedig ezt sikerült kijavítanom, a saját forráskódjából dobott kompatibilitási hibákat (megpróbáltam az IR *.bin* fájlt is konvertálni de az sem működött).

A második lehetséges megoldás az volt, hogyha a *.ckpt* fájlokat sikerülne manuálisan a modell konfigurációjához hozzárendelnem (egy Python szkriptben PyTorch-ot használva), akkor azt le lehetne menteni *.pt* vagy *.torchscript* formátumba. A probléma az volt, hogy az Openvino és az Openvino **Training Extension** modellkonfigurációs fájloknak a formátuma és bennük a modellek felépítési architektúrák különböznek. Az Openvino modellkészletéből (*.xml* kiterjesztés, amelyeket meg lehet nyitni PyTorch API-val) ugyanezek a *training extension*-ben

megtalálható MobileNetv2 és SSD_MobileNetv2 modellek konfigurációi hiányoztak, helyettük csak egy SSlite_MobileNet nevű modellkonfiguráció volt meg, amely ugyebár különböző. Megpróbáltam az *training extension* modellkonfigurációs fájlokat (*.yaml* kiterjesztés, amelyekkel a tanítás folyt) PyTorch-al megnyitni, viszont kompatibilitási hiba történt.

Sokáig kerestem még további megoldások után, mind sikertelenül. Sajnos mivel nem tudtam az Openvino training extension keretrendszerében tanított modell formátumokat átalakítani semmilyen más formátumba, így nem tudtam PyTorch Mobile alatt lefuttatni. A MobileNetv2 modelleket végül nem illesztettem be az alkalmazásba. A tanítás teszt eredményeit fentebb szemléltettem, továbbá írtam a Python szkriptet, amelyben az Openvino API-val megnyitottam ezeket az Openvino IR formátumú modelleket. Ebben a szkriptben egy fényképre is teszteltem a MobileNet-ek helyes működését.

4 A rendszer specifikációi és architektúrája

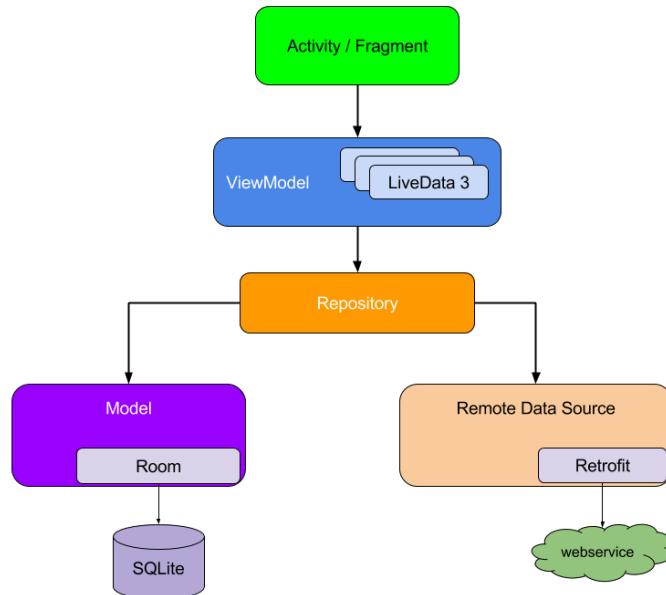
4.1 Az Android alkalmazás architektúrája

Androidos alkalmazáshoz híven, a rendszer objektumelvű architektúrára épül. Az osztályokat funkcióik alapján több típusba sorolják be. Az alkalmazásomhoz én alapjáraton a következő osztálytípusokat használtam:

- **Activity** – a fő UI kontroller elem, teljes képernyőt elfoglal, kezeli a felhasználó utasításait
- **Fragment** – a másodlagos UI kontroller elem, többször felhasználható töredék, mindenig egy Activity-hez tartozik
- **Adapter** – összetettebb struktúrájú UI elemek (pld. RecyclerView) működését kezelő osztály, feladatai közé tartozik például az adatot megjeleníteni az elemeken
- **Data Transfer Object**– célja adatstruktúrák leírása és adatok eltárolása
- **Repository** – az adatokkal és adatstruktúrákat feldolgozó műveletek, függvények az ilyen típusú osztályokban vannak megvalósítva
- **ViewModel** – ennek az osztálytípusnak a szerepe összekötni az adatokat és az adatfeldolgozást a UI-al
- **Util** – segédfüggvényeket tartalmazó osztály, általában *object*, vagyis nem példányosítható

A rendszer architektúrája nagyjából követi “klasszikus” Kotlin-ban írt Android applikációk architektúráját. Az általános architektúra szerint a fő UI oldalak az Activity-k, ezek összezártják

a Fragment-eket, illetve navigálnak közöttük. Az feldolgozásra az adatok ugyebár általában egy adatbázisból vannak lekérdezve, konfigurálva, feldolgozva. A Repository-ba történő feldolgozás után az adatok a ViewModel-en keresztül eljutnak a UI-hoz (Activity, Fragment), ahol megjelenítésre kerülnek. A következő 4.1.1.-es ábrán látható a “klasszikus” Kotlin Android alkalmazásarchitektúra.



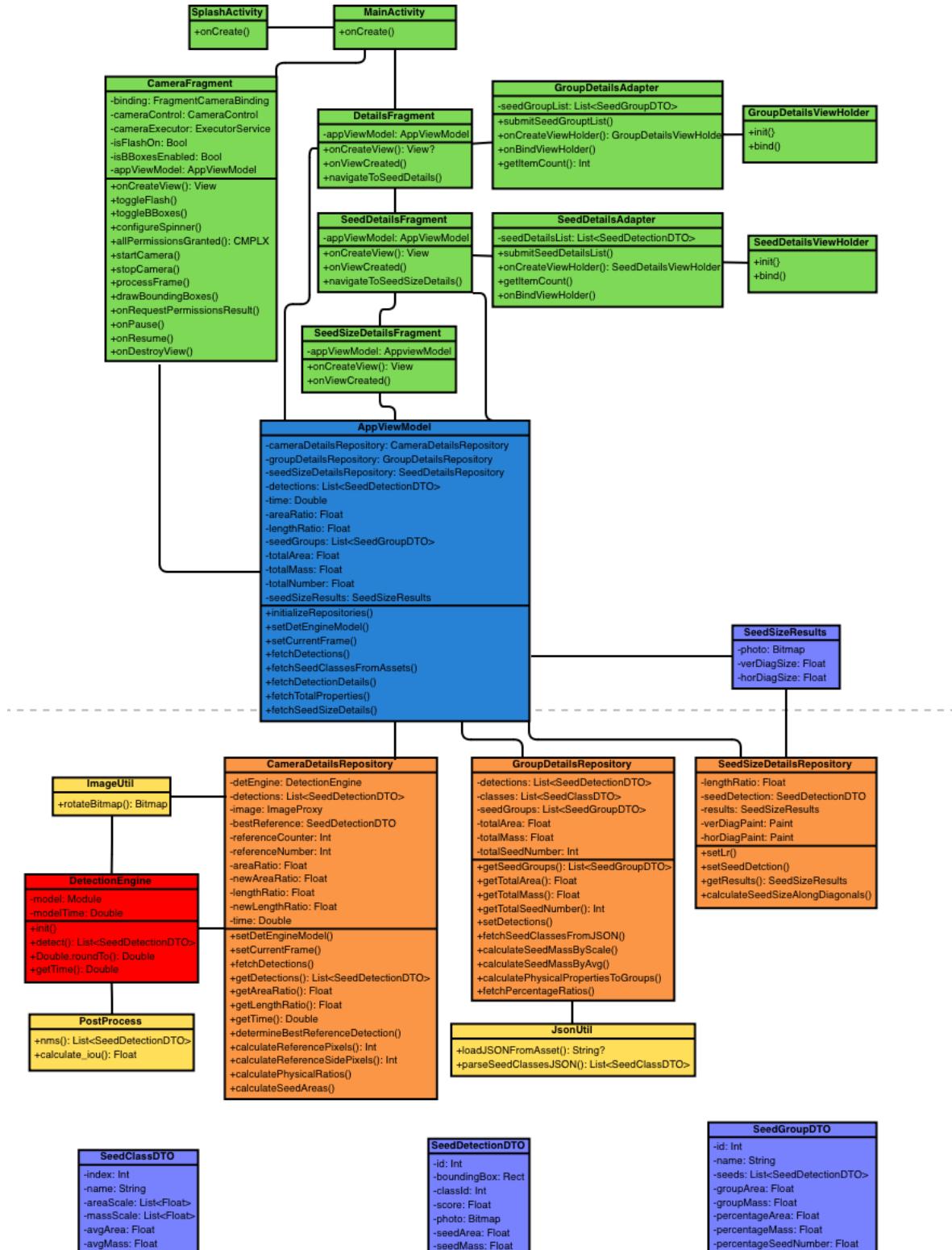
4.1.1. ábra Az Kotlin általános architektúrája

A lényegesebb különbség a fenti architektúra, illetve az én alkalmazásomnak az architektúrája között, hogy én nem használok semmiben adatbázist, sem webservice-t (habár továbbfejlesztésként be lehetne iktatni). Az adatokat emiatt nincs szükség aszinkron módon feldolgozni, tehát a ViewModel-ben sem LiveData típusokkal dolgoztam. Az én alkalmazásomban minden adat lokálisan jön létre, tárolódik el, illetve kerül feldolgozásra.

A következőkben a klasszikus Android applikáció architektúra hierarchiája szerint fentről lefelé fogom bemutatni az osztályok, illetve függvényeik működéseit, céljait. Azért nem haladok logikusan lentről felfele a magyarázatokkal, mivel az adat (képkocka) ugyebár a kamerán keresztül érkezik, vagyis a UI rétegbe (*Activity, Fragment*), így lényegében a program funkciója innen “indul”. A következő oldalon található 4.1.2.-es ábrán található az alkalmazás UML osztálydiagramja az általános architektúra szerinti hierarchiába rendezve (ezt az ábrát a GitHub repository-ban magasabb felbontásban is meg lehet tekinteni). Az applikáció osztályai az architektúra csoportjai szerint tehát a következők:

- **UI főelemek:** SplashActivity, MainActivity

- UI másodlagos elemek:** CameraFragment, DetailsFragment, SeedDetailsFragment, SeedSizeDetailsFragment
- UI elem adapterek:** GroupDetailsAdapter, SeedDetailsAdapter



4.1.2. ábra UML osztálydiagramm (magasabb felb. git repo)

- **Adat átvitel:** AppViewModel
- **Adatfeldolgozás:** CameraDetailsRepository, GroupDetailsRepository, SeedSizeDetailsRepository
- **Adatstruktúrák:** SeedClassDTO, SeedDetectionDTO, SeedGroupDTO
- **Mesterséges intelligenciát kezelő osztályok:** DetectionEngine, PostProcess
- **Util:** ImageUtil, JsonUtil

A *SplashActivity* az első képernyő, amely az alkalmazás indításakor jelenik meg két másodpercre. Ez egy loccsanás (*Splash*) animáció, amely megjeleníti az alkalmazás logóját egy különféle magokról készült fényképpel a háttérben. Miután a két másodpercnyi idő lejárt a MainActivity-re irányít át.

A *MainActivity* a főképernyő, megjelenít egy folyamatos navigációs menüt a Camera, illetve Details fragmentekre. Ezekhez az egyetlen **onCreate()** lifecycle (életciklus) függvénynek bővítésében tartalmazza a navigációs kontrollert. Az alapértelmezett fragment, amelyet megjelenít a CameraFragment.

A *CameraFragment* megjeleníti a folyamatos kameraképet egy 1080x1088-as TextureView elemben. Az **onCreateView()** fragment életciklus függvényben inicializálásra kerülnek az ehhez tartozó UI elemek, illetve ezeknek a működési logikáik. Megemlítendő függvény még a **startCamera()** függvény, amely működteti a kamerát, figyeli a képkockákat az ImageAnalysis segítségével, majd meghívja rájuk a megfelelő **processFrame()**, illetve innen a **drawBoundingBoxes()** függvényeket. A **processFrame()** az AppViewModel osztályal dolgozik, tehát itt kerül feldolgozásra az adat.

A *DetailsFragment* megjeleníti a fajonként csoportba szedett detektálásokat egy görgethető ablakban (*RecyclerView*), kiírva ezek számát, tömegét, illetve elfoglalt területét mind mértékegységekben, mind egymás közötti százalékos arányaikban. Ezenkívül fajtól függetlenül az összes detektált magot, a magok össztömegét, illetve összterületét is megjeleníti. A RecyclerView-ban bármelyik csoportnak a rákattintásakor a SeedDetailsFragment-be navigál át a program.

A *SeedDetailsFragment* megkapja a faj csoport azonosítóját, amelyre rákattintottunk. A teljes képernyőn egy RecyclerView van elhelyezve, amely megjeleníti a csoport összes detektálását, illetve így egyedenként a mag területét, illetve tömegét. Ha ezen a RecyclerView-n kattintunk rá akármelyik elemre, akkor navigálás történik a SeedSizeDetailsFragment-re.

A *SeedSizeDetailsFragment* megkapja a detektálás azonosítóját, eszerint pedig a detektálás bounding box képkivágását. Ez a képkocka feldolgozás után kirajzolódik, rajta egy

vízszintes és egy függőleges középátlóval, amelyeknek különböző színeik vannak. A fénykép alatt különböző színekkel megjelenő számok pedig a mag méretét írják ki milliméterben az átlók mentén.

Az *AppViewModel* ugyebár az adatátmeneti osztály, ez csoportosítja a Repository-k metódusait fragmentenként fetch függvényekbe. Itt tárolódnak el adatfeldolgozások eredményei, amelyek így egyenesen elérhetőek lesznek a UI-nak.

A *CameraDetailsRepository* felel az adatok feldolgozásáért, amelyek a CameraFragment-en jelenítődnek meg. A legfontosabb művelet itt hajtódik végre, amely nem más, mint a detektálás. Ennek a Repository-nak van egyedül kapcsolata a DetectionEngine osztályal, továbbá a referencia arányszámok is itt kerülnek kiszámításra.

Pontosabban a *DetectionEngine* osztályba vannak leimplementálva a neuronhálókezelő függvények, amelyek között olyan műveletek találhatóak meg, mint a súlyfájlok (.pth) megnyitása, majd betöltése PyTorch Mobile modellként, bemeneti tenzor létrehozása a képkockából, majd prediktálás és ezek feldolgozása. A DetectionEngine használja a *PostProcess* objektumot (*object*), amely tartalmazz még egy Non-Max Supression és IoU (*Intersection over Union*) függvényt, az *nms()* továbbá feldolgozza az adatokat és SeedDetectionDTO adatstruktúrájú listává szervezi őket, majd ezt a listát teríti vissza.

Értelemszerűen a *GroupDetailsRepository* a DetailsFragment-en és a SeedDetailsFragment-en megjelenő adatok feldolgozásáért felel, itt határozza meg a program a területet és a tömeget, majd ezeket SeedGroupDTO szerinti listába szervezi.

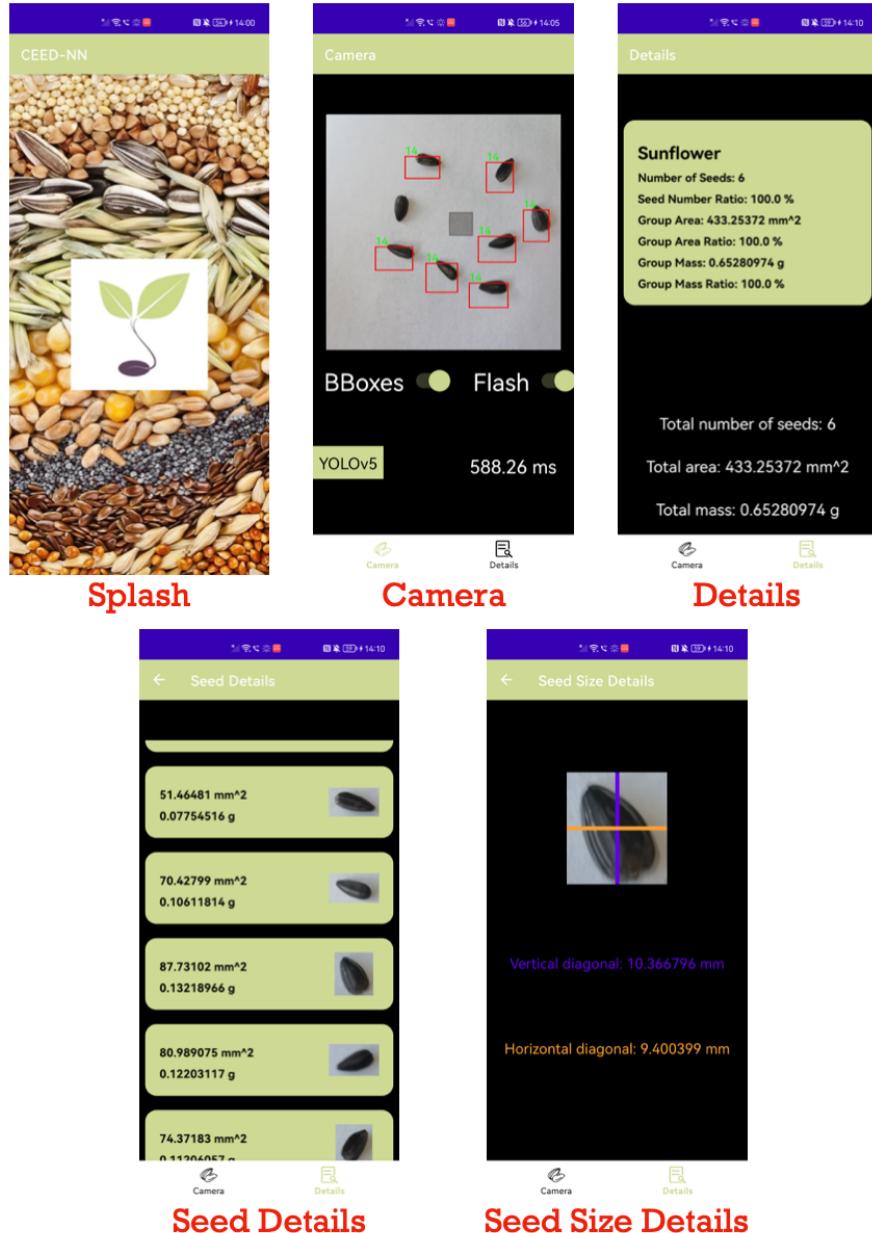
A *SeedSizeDetailsRepository* felel a mag méretének a meghatározásáért, vagyis a SeedSizeDetailsFragment-en megjelenő adatok feldolgozásáért.

A *SeedDetectionDTO*, *SeedClassDTO*, *SeedGroupDTO* információkat tartalmaznak a detektált magokról, mag csoportokról. Eddig még csak a SeedClassDTO-ról nem volt szó, ilyen adatstruktúrában tárolódnak el az osztályazonosítókhoz tartozó osztálynevek, illetve a viszont tömegek merésekor számolt eredmények és az eredmények átlagaik.

Ezek az adatok egy JSON fájlba vannak eltárolva, amelyet a *JsonUtil* objektumban lévő függvények dolgoznak fel. A másik *ImageUtil* pedig a Bitmap forgatására tartalmaz egyetlen függvényt, melyet a DetectionEngine és a CameraDetailsRepository is használ.

Az adapterek, mint a *GroupDetailsAdapter* és a *SeedDetailsAdapter* a nekik megfelelő fragmenteken lévő Recycler View-knak a működési logikáit leíró osztályok. Ezek a fragmentból lesznek konfigurálva, megkapják a ViewModel-ből az lista formájú adatokat, majd ezeket a megfelelő UI elemekre írják.

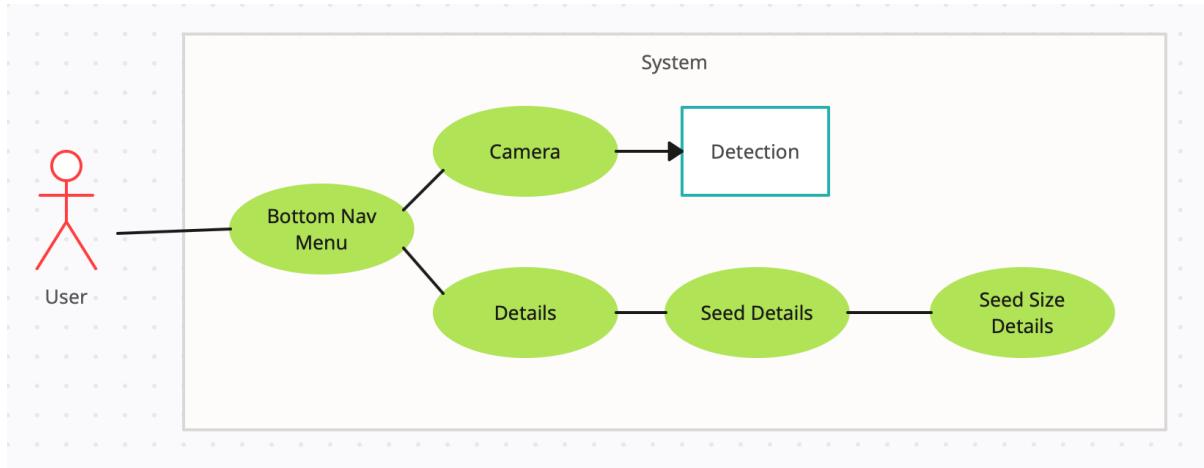
Az alkalmazás fejlesztésénél nem volt játszott központi szerepet a UI dizájnja, arra törekedtem, hogy a funkcionálitások minél jobban működjenek, elsődleges prioritás az alkalmazás működése mintsem a kinézete volt. A következő 4.1.3.-ik ábrán látható az alkalmazás öt képernyője.



4.1.3. ábra Az alkalmazás oldalai

A fenti ábrán megjegyezném, hogy a bounding boxok nincsnek valójában ennyire elcsúszva kirajzoláskor, csupán amikor a képernyőképeket készítettem, akkor a gombok lenyomásával hirtelen ennyire elmozdult a telefon. Továbbá az ábrán lévő *Details* nem az ábrán lévő *Camera* detektálásait mutatja. A vaku használatával, a jobb fényviszonyok miatt sokkal pontosabban detektál a program, de erről az **Üzembe helyezés** fejezetnél bővebben fogok beszélni.

Amiről még nem esett szó talán, megfigyelhető a *Camera* képernyőn, hogy a bounding boxokat, illetve a vakut ki be lehet kapcsolni egy *toggle switch* segítségével, továbbá egy *spinner* (legördülő opciók) segítségével ki lehet választani, hogy most éppen melyik modellt szeretnénk használni. A következő 4.1.4.-ik ábrán az alkalmazás Use-Case diagrammja látható, amely a felhasználó szemszögéből mutatja be az alkalmazást.



4.1.4. ábra Az alkalmazás Use-Case diagrammja

4.2 Az MI működése az applikáción belül

Ahogyan azt már említem, a *DetectionEngine* osztályban vannak leimplementálva a neuronhálók működéséhez szükséges függvények. Ennek az alkalmazásnak a neuronháló kezelése a PyTorch Mobile 1.13.1-es verziója szerint lett felépítve. Már a program indításakor a *spinner* (legördülő opciók) alapértelmezett opciójában (YOLOv8) szereplő modul konfigurációs fájlja (.pth) megnyitásra kerül ebben a modulban az *assets* mappából. A következő 4.2.1.-ik ábrán egy kódrészlet látható a *detect()* függvényből, amely megkapja az **ImageProxy** típusú fényképet az **ImageAnalysis**-től, átalakítja **Bitmap** típussá, elfordítja, átméretezi, majd bemeneti tenzorrá alakítja.

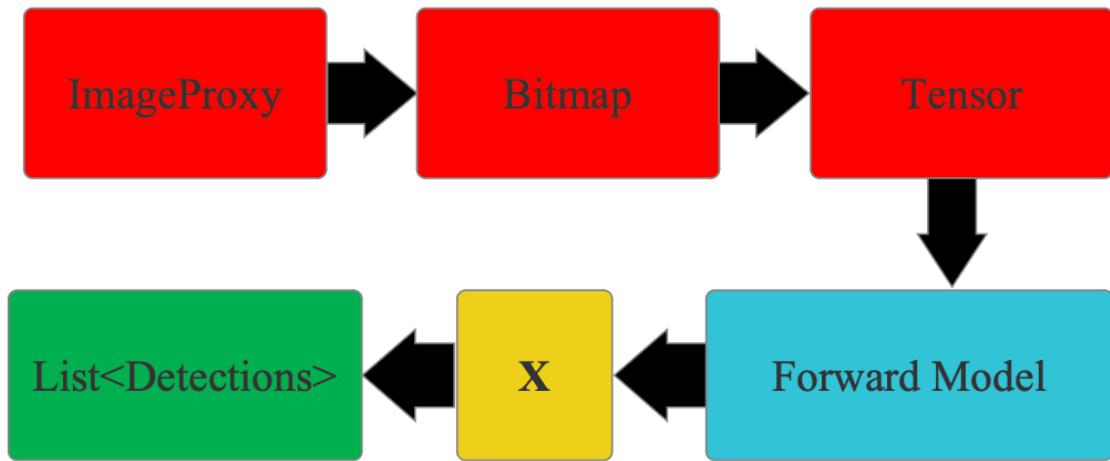
```

54     fun detect (imageProxy: ImageProxy) : List<SeedDetectionDTO>{ ▪ BApot *
55         val bitmap: Bitmap = ImageUtil.rotateBitmap(imageProxy.toBitmap(), angle: 90f)
56
57         val resizedBitmap = Bitmap.createScaledBitmap(
58             bitmap,
59             dstWidth: 640,
60             dstHeight: 640,
61             filter: true
62         )
63
64         val inputTensor = TensorImageUtils.bitmapToFloat32Tensor(
65             resizedBitmap,
66             PrePostProcessor.NO_MEAN_RGB,
67             PrePostProcessor.NO_STD_RGB
68         )
69     }

```

4.2.1. ábra Kódrészlet a képkockák kezeléséről

Ezekután erre a bemenetre lefut a neuronháló, amely ugyebár kimenetként három változót ad, ebből csak az **x** lesz felhasználva. Ugyebár az **x** változó egy tömb amely tartalmazza detektálásokat, ezek bounding box-ainak koordinátáit, a score-ját, illetve az osztályazonosítót. Ez az **x** adódik át a után-feldolgozásra (*Post Process*), hajtódik végre rajta NMS (*Non-Max Supression*), illetve képződik belőle egy könnyebben használható adatstruktúra (*SeedDetectionDTO*). A következő 4.2.2.-es ábra szemlélteti ezt a folyamatot blokkséma szinten.



4.2.2. ábra Képkocka feldolgozása

4.3 Verziókövetés

Verziókövetéshez a GitHub-ot használtam. Létrehoztam a repository-ban külön mappákat minden mesterséges intelligencia résznek, az Android résznek, illetve a segéd szkripteknek. A mesterséges intelligencia része a projektnek az **ai** mappában található. Ezen a mappán belül szintén található egy **ai/tools** mappa, amely tartalmazza a neuronhálókon végzett konvertálásoknak, konfigurálásoknak a forráskódjait. Ezenkívül itt találhatóak meg a modellek konfigurációs fájljaik több kiterjesztésben (**.pt**, **.torchscript**, **.pth** stb.). Az **android** mappa tartalmaz minden kódöt, amely az Androidos applikációhoz szükséges. Ez Android Studio projekt formátumban van. A **tools** mappa tartalmazza a további segéd Python szkripteket, mint például a tanítóhalmazokhoz használt szkripteket, diagram kirajzoló programokat stb. Próbáltam minél gyakrabban commitolni, viszont sajnos túl sokszor vagy belegabalyodtam, vagy túlságosan elmélyültem valamibe, és amiatt felejtettem el, de minden nagyobb változtatás vagy rész befejezése után igyekeztem commitolni. Egyetlen problémám csak a nagy fájlok feltöltésével akadt a YOLOv3 esetében, de a Git Large File Storage segítségével sikerült ezt a problémát is megoldani.

5 Üzembe helyezés és kísérleti eredmények

5.1 Üzembe helyezési lépések

Az alkalmazás funkcionalitásait már a fejlesztés ideje alatt is folyamatosan teszteltem. A tanítóhalmazban lévő magokról készült minták is folyamatos fehér háttér felett voltak címkézve. Szükségem volt továbbá ugyebár egy szürke referencianégyzetre is. A tesztelőkörnyezetem lényegében egy fehér A4-es lap lett egy szürke nyomtatott 1cm x 1cm-es négyzettel a közepében. Ezenkívül ugyebár újból alkalommal elkértem a grammot 3 tizedes pontossággal mérő mérleget a tömeg viszonyításának érdekében.

5.2 Felmerült problémák és megoldásaik

Az eszköz kamerája igencsak sötét képet alkotott, továbbá a fényviszonyok is nagyon instabilak voltak. Az első probléma, amelyre felfigyeltem az volt, hogy a fényviszonyok változására a neuronhálók nagyon hajlamosak összekeverni a detektálások osztályait, sőt még kevesebb példányt is detektálnak rossz fényviszonyokkal rendelkező, például sötét képeken. Ennek a problémának az orvoslására implementáltam a programba egy vaku kapcsolót. A vaku segítségével egyértelműen jobb eredmények születtek, mivel stabilizálta a fényviszonyokat és világosabb képet alkotott.

A következő probléma a kamera és felismerendő objektumok közötti távolság miatt adódott. A tanított fajok közül vannak külsőleg hasonlító, de viszont méretekben teljesen eltérő magfajok (pld. Szöszös bükköny és Olajrétek). Ha a telefon kameráját minél magasabbra tartottam a tesztelési háttéről, az egy bizonyos magfajnak az apróbb hasonlását ismerte fel, ha a legmagasabbra tartottam akkor pedig minden a legapróbb magfajnak a Bíborherének ismert fel. A probléma okára nem sikerült rájönnöm. Gondoltam a tanítóhalmaz helytelenségére, viszont ebben a tanítóhalmazban három különböző távolságban készültek a fényképek. Gondoltam továbbá az alacsony bemeneti felbontásra, amely legalább négyszer alacsonyabb, mind a valódi felbontása a képeknek, így sok tulajdonság és részlet veszik el az újraméretezésnél. A módszer amelyet kipróbáltam ennek a problémának az orvoslására az volt, hogy megpróbáltam a képkockákat negyedelni, vagyis négy egyforma méretű részre vágni a középátlóknál, majd ezeket így külön lefuttatni a neuronhálón, majd összerakni őket és kirajzolni a prédkálásokat. Ennek a megoldásnak a hátulütője az volt, hogy mivel a középátlóknál osztottam fel a képeket, így az éppen ott lévő magok két (vagy esetleg négy) rész külön lett detektálva, külön bounding box-ok lettek a felekre rajzolva, ez a fizikai tulajdonságok számolását is nagyban megnehezítette volna. Továbbá módszer nem hozta meg

az megváltást, ugyanis csak egy pár milliméternél több távolságot tudtam tartani, mielőtt a predikciók elvesztették volna az osztálypontosságukat.

A következő probléma már a tulajdonságok számításánál jött elő, ugyanis a referenciakockából számolt arányszám folytonos megváltozása miatt egy ugyanazon mag folytonosan újra számolt területe is igen nagy intervallumban változott. Erre megoldásként bevezettem egy átlag számítást 10 külön referencianégyzet detektálásból, hogy sokkal stabilabb értékeket kapjak. Ez bevált, ugyanazon példányok külön-külön detektálásaiból a fizikai tulajdonságok már nem lettek annyira eltérőek.

5.3 Kísérleti eredmények, mérések

5.3.1 Neuronhálók sebességei

A neuronhálók programon belüli sebességét a 10 detektálás sebeségének átlagából számoltam ki. Ezt a 10 detektálási sebességet modellenként, a program futása közben a konzolra írottam ki, a referenciának az átlagolását pedig kikapcsoltam. A következő 5.3.1.-es táblán a négy YOLO modell programfutás közbeni átlagsebessége van megjelenítve milliszekundumban.

Modell neve	Átlagsebesség (ms)
YOLOv3	4857,707
YOLOv5	198,602
YOLOv6	169,889
YOLOv8	220,007

5.3.1. táblázat A modellek program futása közbeni átlagsebességek egy képkockára

Érhető okokból lett harmadik verzió a leglassabb, hiszen a legtöbb réteggel és paraméterrel rendelkező modell. Továbbiakban viszont érdekes, hogy nem az Ultralytics által fejlesztett hálók érték el a legnagyobb sebességet, mondjuk az a 20 milliszekundumnyi különbség jelen helyzetben eltörpül.

5.3.2 Osztályozás tesztelése

Az applikáció dokumentációba való tesztelése során hat növénymagfaj osztályozását teszteltem. Mind az hat növénymagfajra, mind a négy modellel végeztem egy-egy osztályozást, illetve ezek eredményeit lejegyeztem. A következőkben felsorolom ezen magok neveit, illetve, hogy miért pont ezekre esett a választás:

- **Bíborhere:** Ez a legkisebb mérettel rendelkező mag, illetve ebből készült a legtöbb, mint a tanítóhalmazban

- **Kifejtő borsó:** Mivel három másik borsófélle is tanításra került kíváncsi voltam az eredményekre
- **Csicsceriborsó:** A színe nagyon hasonlít a legnagyobb “magra”, a Lóbabra, kíváncsi voltam, hogy mennyire sikerül elhatárolni a kettőt
- **Lófogú kukorica:** Mivel még van egy kukoricaféle tanítva
- **Őszi búza:** Mivel még három gabonaféle található
- **Napraforgó:** Mivel gyanúm szerint az ekkor a teszteléskor lévő sötét fényviszonyok zavartak be a tervezés során sokkal pontosabban és jobban működő osztályozáskor

A következő 5.3.2.-ik táblázatban szemléltetem a mérések Excel táblázat béli struktúráját, ez az Excel táblázat a projekt GitHub Repository-ában is megtalálható lesz a **docs** mappában.

Magfaj	Modell	Magszám	Felismert	Téves	Nem detektált
1. Bíborhere	YOLOv3	53	45	0	8
	YOLOv5		49	0	4
	YOLOv6		41	8	3
	YOLOv8		49	0	4
...

5.3.2. táblázat Az osztályozások tesztelései

A tesztelések során éppen sötétebb idő volt kint, így ennek következtében az élénk színű magok, amelyek eddig tulajdonságaik nagyrészét ebből nyerik elsötétülnek és színük a sötétebb magokéra kezd hasonlítani. Ez történt az Őszi búza esetében, amelynek piros színe barnára sötétült, ugyanis a tesztelés során a legtöbb detektálás a Lóbab osztályhoz került. Ugyanez történt a Csicsceriborsó esetében is, ugyanis a tesztelés ideje alatt a detektálások a sötét színű Takarmányborsó és Lóbab osztályokhoz kerültek. A tesztelés során az elmélet bebizonyításához a Napraforgó magot használtam, mivelhogy ez fekete, így nem kellene annyira befolyásolja a fényviszonyok változása. Így is történt, a Napraforgó magok nem lettek helytelen osztályokba sorolva.

A leggyorsabb, de ezzel egyidőben legpontatlanabbról osztályozó modell a YOLOv6 lett. Sok esetben még a referencianégyzetet is alig ismerte fel, ha mégis akkor pedig legtöbb esetben helytelenül keretezte vagy a napraforgó osztályba sorolta. A leg pontosabb modellek helyzettől függően a YOLOv5 és YOLOv8 verziók lettek.

5.3.3 Tömeg számítások tesztelése

A méretek meghatározásait ugyanezknél a fajknál, ugyanebben a tesztelési fázisban mértem. A referencia átlag számítását a tesztelésnél kikapcsoltam. A méretek pontosságai nagyban függetek a referencianégyzet pontos detektálásától, így az átlag tömeg hiba intervalluma egészen 33%-ig terjedt. A következő 5.3.3.-ik táblázatban a tömeg mérések részlete látható.

Magfaj	Modell	Magszám	Példány valódi tömege	Példány számolt tömege	Tömeg hiba
2. Kifejtő borsó	YOLOv3	17	0,166	0,166	0
	YOLOv5		0,142	0,21	32,38095238
	YOLOv6		0,219	0,219	0
	YOLOv8		0,2	0,194	3
...

5.3.3. táblázat Tömeg mérések

6 Következtetések

6.1 Megvalósítások és a rendszer felhasználása

A kezdeti célok közül sikerült a legtöbb funkcionálitást implementálnom, a program megfelelő kezeléssel sikeresen megkülönböztet magokat, viszonylag pontosan becsül tömeget és méretet. Ahhoz viszont, hogy a mezőgazdaságban is lehessen használni szükség van még néhány fejlesztésre, változtatásra.

A rendszert jelenlegi állapotban felhasználásra labor körülmények között lehetséges, vagyis szükség van kiegészítő hátrére, illetve referenciára, amely szerint a fizikai tulajdonságokat kissámolja. Labor körülmények között könnyebben (mint a mérleg vagy a tolómérce) használható tömeg, illetve méretbecslésre, abban az esetben, ha nincs szükség csak egy viszonylagos (nem több tizedesnyi) pontosságra. Mivel valós időben számol, így viszonylag kevés továbbfejlesztéssel, éppen, hogy a méréseket lokálisan vagy egy adatbázisban tárolja el, fel lehet használni a rendszert. Ha mondjuk alatta folyamatosan egy szalagon görgetjük, vagy keretbe rakjuk a különböző kevert magtípusokat, ennek a végén az eredményeket összesíteni lehet, mind példányszámok, mind tömeg szempontjából.

6.2 Hasonló rendszerekkel való összehasonlítás

Az Ultralytics HUB [19] applikációval hasonlítottam össze az alkalmazásomat. A legszembetűnőbb különbség talán az volt, hogy az Ultralytics applikációjában a modellek 10 szer gyorsabban futottak (20-30 ms), mivel ugyebár az Ultralytics HUB a modelleket GPU-ra optimalizálja. Ezenkívül természetesen a felhasználói felület is szebben ki lett van dolgozva, a program gördülékenyebb, de viszont egy egész csapat dolgozott rajta. Következőként a rovarfelismerésre tanított YOLO-k [20] tanítási eredményeivel is összevettem az én tanítási eredményemet. Ezek tanításánál az adathalmaz mérete, a magfelismerő modelleket tanítására használt adathalmaznak a harmada volt, továbbá 22 osztály helyett is csak 6 osztály lett tanítva. A rovarfelismerő applikáció YOLO hálóinak a pontossága 95% felettire sikeredett 140 ciklusnyi tanítások után, de majdnem negyszer több osztállyal és háromszor nagyobb tanítóhalmazzal 150 ciklusra 80%-os pontoság jó eredmény.

6.3 Továbbfejlesztési lehetőségek

Véleményem szerint a neuronhálók pontosabb detektálási képességéhez a következő fejlesztéseket lehetne végezni:

- **Tanítóhalmaz fejlesztése:** Több háttér alkalmazása, bővebb tanítóhalmaz gyűjtése, minél több fényviszony alkalmazása a fényképek készítésénél, továbbá minél több távolságból készíteni a fényképeket.
- **Nagyobb bemenettel rendelkező hálók alkalmazása:** Például a YOLO-k esetében vannak 1280 x 1280 méretű bemenettel rendelkező modellek is, viszont ezek természetesen lassabbak.
- **Nagyobb modellek alkalmazása:** A YOLO-k nagyobb méretű modelljeinek a tanítása és beillesztése az alkalmazásba, szintén kisebb sebességhoz vezet.

A neuronhálók sebességének a növeléséhez a következő fejlesztési lehetőségeket kell figyelembe venni:

- **PyTorch Lite használata:** A PyTorch Lite állítólag jobban optimalizálja és kezeli a neuronhálókat, mint a PyTorch Mobile. Ezáltal jobb erőforráskezeléshez, és így sebességhoz vezethet. Ennek a csomagnak a használatához viszont szükséges a Wrapper modellt Kotlin-ban implementálni, illetve további bővítésekkel végezni.
- **GPU-ra való optimalizálás:** Az okostelefonok grafikus feldolgozóegységének használata, mint ahogy az Ultralytics HUB esetében is észrevehető, többszörösen megnöveli az erőforrásokat.

A fizikai tulajdonságok pontosabb meghatározásához a következő fejlesztési lehetőségeket kellene szempontként venni:

- **Távolság szerinti méretmeghatározás:** A referencianégyzet használatának elvetéséhez például ez az egyik lehetséges megoldás. A tervezésnél felfigyeltem erre a módszerre, de mivel ennek a megvalósítására két kamera működtetése szükséges, illetve szögek meghatározása és komplex matematikai egyenletek, így féltem, hogy a teljesítményből húzna le, amely a neuronhálók lelassulásához vezetne.
- **Szegmentálás bevezetése:** A szegmentációs algoritmusok lényegében nem csak négy pontból álló mintákat tudnak bemenetként kezelni, illetve megtanulni. Szegmentációs modellek, illetve szegmentációs tanítóhalmaz összerakásával anélkül lehetne a fizikai tulajdonságokat kiszámítani, hogy aggódni kellene a háttér pixelek kiiktatásával.

Bibliográfia és egyéb hivatkozások

1. Jiang, Peiyuan, et al. "A Review of Yolo algorithm developments." *Procedia Computer Science* 199 (2022): 1066-1073.
2. Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.
3. Redmon, Joseph, and Ali Farhadi. "Yolov3: An incremental improvement." *arXiv preprint arXiv:1804.02767* (2018).
4. Bochkovskiy, Alexey, Chien-Yao Wang, and Hong-Yuan Mark Liao. "Yolov4: Optimal speed and accuracy of object detection." *arXiv preprint arXiv:2004.10934* (2020).
5. <https://docs.ultralytics.com/models/yolov5/>
6. Li, Chuyi, et al. "Yolov6 v3. 0: A full-scale reloading." *arXiv preprint arXiv:2301.05586* (2023). Howard, Andrew G., et al. "MobileneTs: Efficient convolutional neural networks for mobile vision applications." *arXiv preprint arXiv:1704.04861* (2017).
7. Wang, Chien-Yao, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors." *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2023.
8. <https://docs.ultralytics.com/models/yolov6/>
9. <https://docs.ultralytics.com/models/yolov7/>
10. <https://docs.ultralytics.com/models/yolov8/>
11. Wang, Chien-Yao, I-Hau Yeh, and Hong-Yuan Mark Liao. "YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information." *arXiv preprint arXiv:2402.13616* (2024).
12. Wang, Ao, et al. "YOLOv10: Real-Time End-to-End Object Detection." *arXiv preprint arXiv:2405.14458* (2024).
13. Karar, Mohamed Esmail, et al. "A new mobile application of agricultural pests recognition using deep learning in cloud computing system." *Alexandria Engineering Journal* 60.5 (2021): 4423-4432.
14. <https://docs.ultralytics.com/models/rtdetr/>
15. Liu, Wei, et al. "Ssd: Single shot multibox detector." *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I* 14. Springer International Publishing, 2016.
16. Howard, Andrew G., et al. "MobileneTs: Efficient convolutional neural networks for mobile vision applications." *arXiv preprint arXiv:1704.04861* (2017).
17. Sandler, Mark, et al. "MobileneTv2: Inverted residuals and linear bottlenecks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.
18. Howard, Andrew, et al. "Searching for mobilenetv3." *Proceedings of the IEEE/CVF international conference on computer vision*. 2019.
19. <https://docs.ultralytics.com/hub/app/android/>
20. Badgujar, Chetan M., et al. "Real-time stored product insect detection and identification using deep learning: System integration and extensibility to mobile platforms." *Journal of Stored Products Research* 104 (2023): 102196.
21. Parico, Addie Ira Borja, and Tofael Ahamed. "Real time pear fruit detection and counting using YOLOv4 models and deep SORT." *Sensors* 21.14 (2021): 4803.

22. <https://www.cvat.ai>
23. <https://docs.ultralytics.com>
24. <https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html>
25. https://github.com/openvinotoolkit/training_extensions
26. <https://docs.docker.com/engine/>
27. <https://developer.android.com/studio>
28. <https://kotlinlang.org/docs>
29. https://www.canon.hu/for_home/product_finder/cameras/digital_slr/eos_70d/specifications/
30. <https://openvinotoolkit.github.io/datumaro>

Kódsegítség

31. <https://pytorch.org/docs/>
32. <https://pytorch.org/mobile/android/>
33. <https://github.com/pytorch/android-demo-app/tree/master/ObjectDetection>
34. <https://stephencowchau.medium.com/exploring-using-yolov8n-with-pytorch-mobile-android-479b0b866a3d>
35. <https://developer.android.com/media/camera/camerax>
36. <https://stackoverflow.com/questions/58080669/how-to-draw-rectangle-at-camerax-textureview>

UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE, TÎRGU-MUREŞ
SPECIALIZAREA AUTOMATICĂ ȘI INFORMATICĂ APLICATĂ

Vizat decan

Conf. dr. ing. Domokos József

Vizat director departament

S.1. dr. ing Szabó László Zsolt