Önálló laboratórium beszámoló

Távközlési és Médiainformatikai Tanszék

Készítette:	**Faluvégi Ármin**
Neptun-kód:	**WYL4R7**
Ágazat:	**Infokommunikációs hálózatok és alkalmazások**
E-mail cím:	**Faluvegi.armin@gmail.com**
Konzulens:	**Dr. Varga Pál**
E-mail címe:	**pvarga@tmit.bme.hu**

Téma címe: YOLO alapú több osztályos kép szegmentáció, Jenkins alapú CI/CD megoldással

Feladat

A feladatom a félév során a YOLO modell tanításának és használatának megismerése, és ez alapján egy működő modell létrehozása, valamint a Jenkins nevezetű CI/CD platform megismerése volt, amibe az előzőleg létrehozott YOLO modellt kellett átültetnem és ebből automatikusan kellett a feltöltött adatokkal lefuttatott tanítást tartalmazó, használható felületet létrehoznom, ami már a jenkinstől függetlenül fut, annak leállása után is.

2023/2024. 2. félév

1. A laboratóriumi munka környezetének ismertetése, a munka előzményei és kiindulási állapota

1.1 Bevezető

A feladat motivációja egy olyan rendszer létrehozása volt, ami képes valós időben megkülönböztetni több előre meghatározott objektum-osztályt egy bejövő képfolyamból. Ehhez a mély neurális hálózat alapú tanulásos rendszerekre épülő megoldást választottam alapul, nevezetesen a You Only Look Once (YOLO) elnevezésű konvolúciós rétegekre építő valós idejű objektum detekciós és kép szegmentációs rendszert.

Maguk a mély tanulásra épülő első objektum detektálásra készült modellek a 2010-es évek közepén kezdtek el megjelenni [1]. Akkor még eléggé kezdetlegesen működtek ezek a modellek, mind számítási kapacitás, mind pedig az algoritmusok kiforratlansága miatt. Ez mára már jelentősen megváltozott, köszönhetően a videókártyákra történő implementálásnak, a videókártyák megnövekedett számítási kapacitásának, és a sokadik iterációból következő már eléggé jól kiforrott modell felépítéseknek. Ezek közül is kiemelkedően jól használhatók az objektum detektálásra, „befoglaló doboz” /” bounding box” ba való foglalásukra és szegmentálásra, vagyis az objektum képen való pixelszintű megjelölésére alkalmas modellek. Ilyenek többek között a YOLOv8, Mask R-CNN, Panoptic FPN és a YOLACT++ [2] nevezetű modellek, ezek közül én a YOLOv8 szegmentációra készült modelljét használtam.

A modellt egy valódi környezetben több esetben nem csak magában működő rendszerként használnák, emiatt a projektemben én is beépítettem a modell létrehozásához elkészült kódot egy folyamatos integráció/folyamatos leszállítást (Continuous Integration/Continuous Deployment) lehetővé tévő környezetbe, a Jenkinsbe. Ez a program lehetővé teszi, hogy különböző általunk definiált eseményre a kód újra leforduljon. Esetemben bármilyen frissülésre, legyen az új adat, vagy változott kód, a rendszer újra lefuttatja az egész kódot és az elkészült rendszert egy virtuális konténerben. A modellt már a Jenkinstől függetlenül futtatja, ez az elkészült konténer a Docker nevezetű virtualizációs környezetet használja. Az elkészült kód egy kezdetleges weboldalon keresztül ad egy kezelőfelületet a felhasználónak, ahol lehetősége van képek feltöltésére, amiket a program lefuttat az elkészült modellen, így adva egy szegmentált eredményt.

1.2 Elméleti összefoglaló

1.2.1 YOLO

A You Only Look Once (YOLO) egy olyan, több modellt magába foglaló modellszéria, ami az utóbbi években a mély tanulásra épülő képi objektum-detekció meghatározó szereplője lett. Az elterjedtségének az oka a pontossága mellett az, hogy már egy kommerciális videókártyán is képes tanításra, így széles fejlesztőréteg számára elérhető. Az első változata 2015-ben jelent meg, ez a változata még a Darknet nevezetű C-alapú mély tanulásos keretrendszerre épült, a v3as modell már a pytorch nevezetű Python-alapú, facebook által létrehozott mély tanulásos keretrendszerre épült. Ez ma is egy meghatározó keretrendszer, a YOLOv8 is erre épül. Jelenlegi feladatomhoz is ezt használtam.

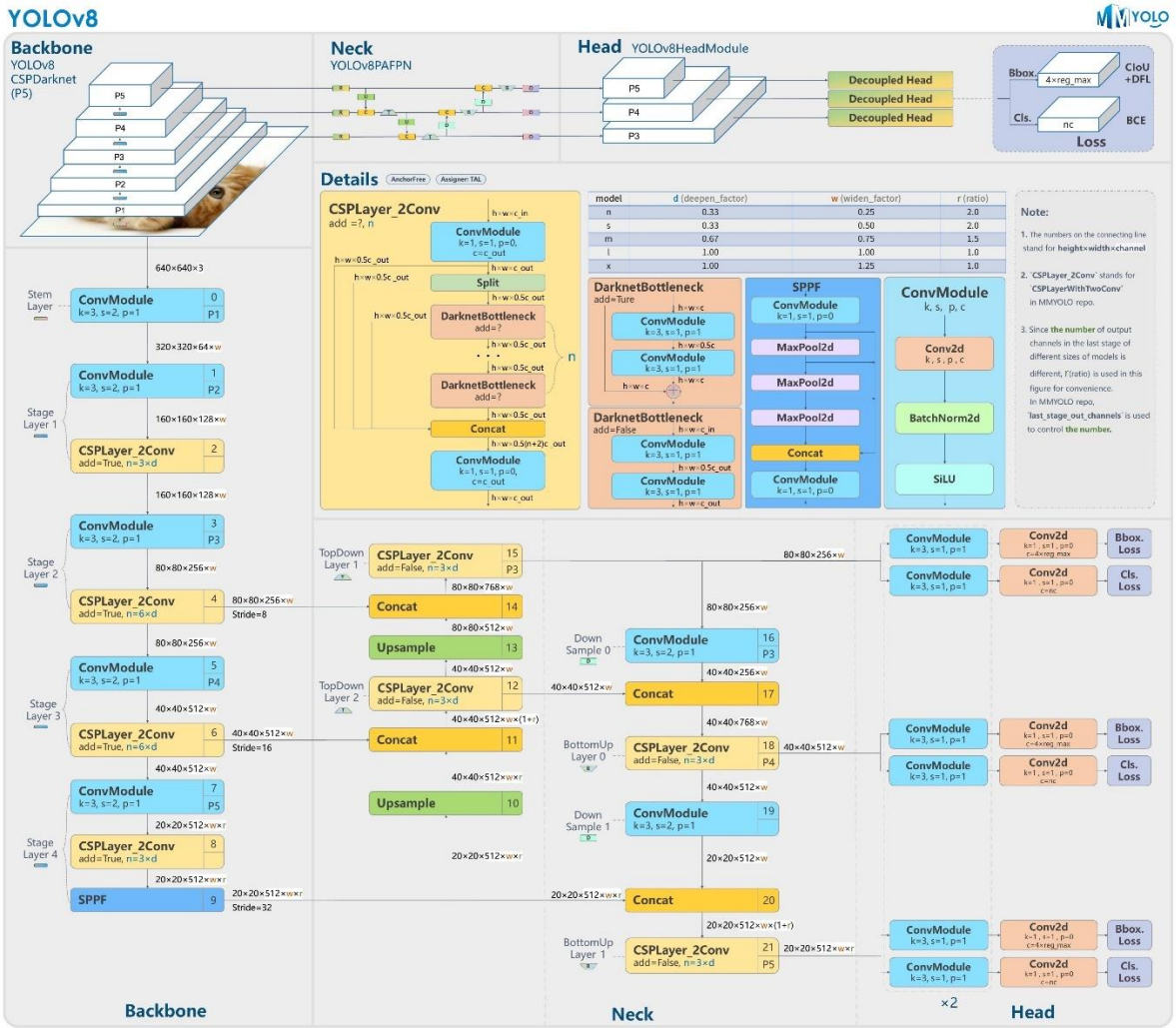
A YOLO főként egy objektum detekcióra és bounding boxba foglalásra készült modell. A szegmentációt is lehetővé tevő első változata a YOLOv7 volt [3] [4] [5], de ennek a modellnek a fejlesztése már 2 éve befejeződött.

Mint minden Deep learning modell, a YOLO is különböző rétegeken keresztül futtatja az adatait, így létrehozva a nyers bemenő adatokból egy számunkra hasznos kimenetet. A YOLO v8-as változatának a rétegei láthatóak az 1. ábrán.

Az általam is használt YOLOv8, az Ultralytics gondozásában álló, jelenleg is fejlesztés

alatti változatának a szegmentációs modell már sokkal integráltabb része, mint a YOLOv7-nek. Több különböző paraméterszámmal rendelkező modell is elérhető, amiből a fejlesztő a szükséges pontosságnak és az elérhető erőforrásoknak megfelelően maga választhat egy neki megfelelő modellt [6].

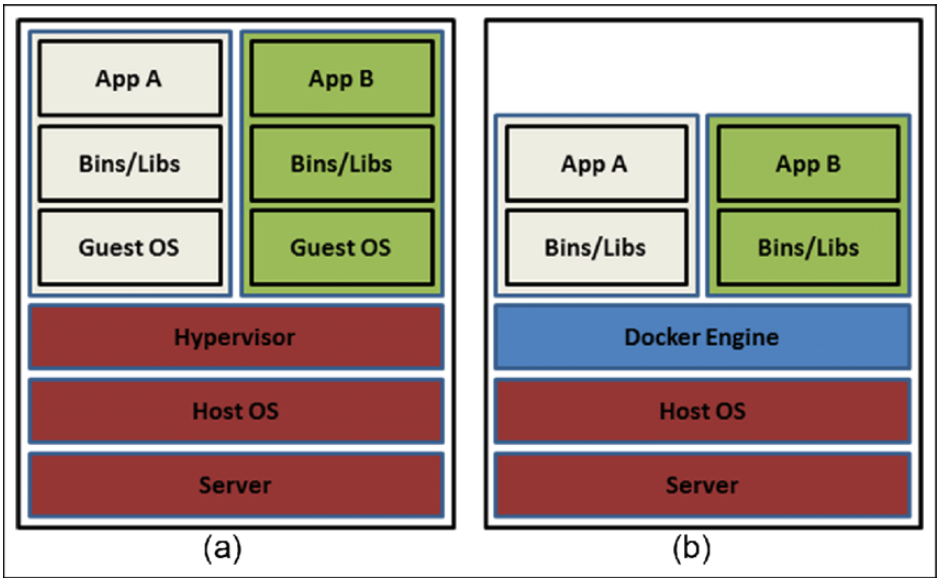
A YOLOv8as modell rendelkezik egy úgynevezett hangolás (tune) funkcióval, aminek a segítségével próbatanításokat lehet végezni automatikus módon, ami alapján könnyebben megválaszthatóak az optimális hiperparaméterek. A hiperparaméterek általánosságban a mély tanulásos modellek különböző beállításait jelentik, például batch size, epoch szám, learning rate. A batch size az egyszerre a modell által tanuláshoz felhasznált adatcsomag, például ha van 100 adatunk és a batch size 20, akkor a modellünk 5 alkalommal tanít a különböző 20 darabos batchekkel/csomagokkal, az epoch szám az egész adathalmazon átfuttatott tanítások számát adja meg, a learning rate pedig a tanítási folyamatban használt lépés méretét határozza meg. Minél kisebb, annál lassabb, de pontosabb a tanítás. A YOLO-ban van lehetőség folyamatosan csökkenő learning rate (tanítási ráta) megválasztására, így a tanítás kezdetekor még a folyamat gyorsan tanul, de később a csökkenő learning rate miatt ez nem fog a pontosságban problémát okozni. Részletesen a tune funkcióról és a hiperparaméterek pontos leírásáról az ultralytics weboldalán (is) van lehetőség tájékozódni [7].



1. ábra: A YOLOv8 rétegei

1.2.2 Docker

A Docker egy virtualizációs környezet, ami operációs rendszer szintű virtualizációra képes. Ez a klasszikus hardver virtualizációval ellentétben csak operációs rendszer szintig történő szimulációt hajt végre, a kernelt már nem példányosítja. Ezt szemlélteti a 2. ábra is. Az így létrejött úgynevezett konténerek, amik ezeket a virtualizált példányokat tartalmazzák, egy közös kernelen futnak, ezáltal kevesebb erőforrás árán is képes egymástól elszigetelt rendszerek futtatására ugyanazon a hardveren.



2. ábra: teljes és Docker féle konténer virtualizáció

A másik, ennél jelentősebb előnye a Docker környezetnek az, hogy az elkészült virtuális konténerekből könnyen készíthetőek a hardveres szimulációban is ismert képek, angolul image-k. Ezeket az imageket a Docker egyik alapvető funkciójával egy általunk választott hubra feltölthetjük és később más környezetben használhatjuk. Ez a hub lehet akár privát, vagy a Docker által fenntartott dockerhub nevű szerver is, ahonnan más által elkészített és feltöltött image is letölthető számunkra, így sok időt megspórolva egy jól kiválasztott image-el, amin már a számunkra szükséges környezet előre be lett állítva és installálva, például már egy előre telepített Linux környezet.

A Docker képes úgynevezett swarm módra, ennek a segítségével lehetséges több, egymástól független Docker hoszt egyként való kezelése. Rendelkezik beépített load balancing, vagyis terhelés elosztás funkcióval, ami automatikusan elosztja a hosztok között a konténereket, így azok egyenletesen terhelik a swarm hálózatot, képes automatikusan növelni vagy csökkenteni a használt szolgáltatásokat CPU terhelés alapján, így skálázva azokat. Ezen kívül képes a fokozatos frissítésre (angolul rolling updates), így nem egyszerre áll le egy szolgáltatás összes szervere/konténere, hanem folyamatosan lesz elérhető kiszolgáló, így a felhasználó nem vesz a frissítésből észre semmi különöset. A különböző hosztokat node-oknak nevezzük; létezik worker és manager node. A worker nodeon futnak a különböző, épp használatban levő konténerek, ezeket a manager node-ok irányítják, szervezik. Több manager node is létrehozható, így redundanciát teremtve, aminek segítségével a hibatűrést növelhetjük és „öngyógyító” hálózatot hozhatunk létre. A swarm Transport Layer Security (TLS) protokollt használ az authentikációhoz és titkosításhoz, így már kezdetekben is egy biztonságos kommunikációt nyújt a különböző nodeok és saját maga között.

A Docker képes a különböző egymástól függetlenül futó konténereket több különböző

módon hálózatba kapcsolni. Ezek a következőek: Bridge, Host, Overlay, Macvlan, Ipvlan, None.

- Bridge: A bridge az alapértelmezett hálózati beállítás a konténereken, ilyenkor a létrejött konténer rákerül egy szoftveres bridge hálózatra, amin keresztül csak az ugyanezen a bridge hálózaton található konténerekkel lesz képes kommunikálni, így izolálva egymástól különböző konténercsoportokat, amik nem akarjuk, hogy hozzáférjenek egymáshoz közvetlenül.
- Host: A host beállítás esetében a konténerünk hozzáfér a gazda számítógép teljes interfészéhez, ilyenkor nincs semmilyen izoláció, alapvetően ez a funkció hozzáférést ad a hoszt gép interfészéhez, így olyankor lehet hasznos, amikor sok portot szeretnénk a konténerrel használni, vagy nagy forgalom kiszolgálása a cél.
- Overlay: Az overlay különböző hosztokon futó konténerek összekötésére alkalmas, gyakran a swarm beállítással együtt használják, így összekötve a swarmban résztvevő hosztok hálózatait.
- Macvlan: Macvlan esetében a konténer egyedi interfészt és MAC azonosítót kap, ahol a DHCP is működik, így a konténer számára nagy szabadságot biztosít hálózati beállítások szempontjából.
- Ipvlan: Ipvlan esetében a hoszt a konténer számára hozzáférést ad az IP alapú címzéshez, vagyis minden konténer egyedi IP címet kap a hálózaton, így a konténer a bridge beállításhoz képest sokkal nagyobb szabadságot kap, de nem kap egyedi MAC címet, úgyhogy ez egy köztes út a bridge és Macvlan között.
- None: Ebben az esetben nincs semmilyen hálózati hozzáférés.

A Dockerrel kapcsolatosan további információ a Docker weboldalán található dokumentációban elérhető [8].

1.2.3 Jenkins

A Jenkins egy úgynevezett CI/CD szolgáltatást nyújtó alkalmazás. A CI/CD kifejtve: Continuous Integration and Continuous Deployment/Delivery, ami magyarul: folyamatos integráció és folyamatos leszállítás.

A folyamatos integráció alatt azt a gyakorlatot értjük, hogy a fejlesztés közben megírt kódot gyakran lefordítjuk és teszteljük, így megbizonyosodva róla, hogy az eddig elkészült programunk szintaktikailag jó, és teszt alapján megfelelően működik. A folyamatos leszállítás (vagy a Wikipédia magyar cikkében használt fordítás: Rövid ciklusú fejlesztési eljárás) alatt a fejlesztés gyorsabb ciklusokon keresztül történik és a lefejlesztett alkalmazást ezeknek a ciklusoknak a végén már kiadják, például naponta frissülő alkalmazás.

Ennek a CI/CD elvnek az automatizációjához egyik elterjedtebben használt szoftver a Jenkins. A segítségével lehetséges az elkészült kód automatikus fordítása és tesztelése, ehhez az egyik használható funkció a pipeline, aminek a segítségével egy általunk meghatározott folyamat zajlik le, például egy megírt kódot először lefordít, tesztel, majd ebből egy futtatható docker fájlt készít vagy akár azonnal elindítja egy, a Jenkins szervertől független szerveren. A pipeline-on kívül még több másik munka definiálás lehetséges - például az egyszerűbb Freestyle elnevezésű folyamat, ami a Jenkins megismerésekor egy könnyített lehetőséget biztosít egy működő munkafolyamat létrehozására.

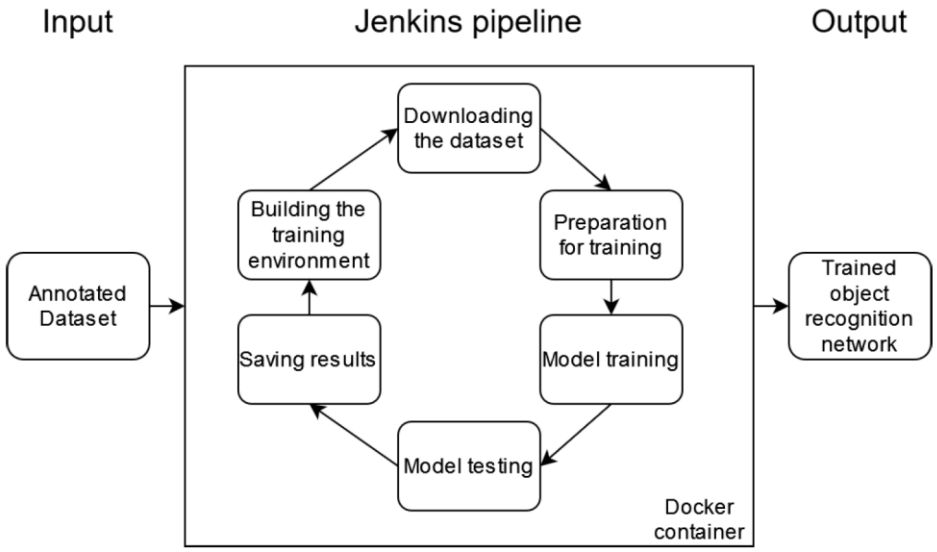
A Jenkins szerveren található kód frissítése többféleképpen is lehetséges, akár kézzel, akár egy úgynevezett Poll Source Control Management (SCM) beállítással, ahol meg tudjuk határozni, hogy egy általunk meghatározott git repository-t milyen gyakran ellenőrizzen új változások/commitok után és ha új commitot észlel, akkor automatikusan letölti azt, és elindít egy új pipeline futtatást/jobot.

A Jenkins minden elindított munkát képes a saját szerverén futtatni, de különböző plugin-ek

segítségével akár egy külső Docker szerveren is el lehet indítani és futtatni a különböző munkákat.

1.2.4 A Jenkins által vezérelt folyamat

A 3. ábra azt a jellegzetes Jenkins pipeline-t mutatja, ahogyan a címkézett adattömeg alapján az objektum-detekciós modell betanításra kerül, és a kimeneten előáll a Docker-ként csomagolt modell.



3. ábra: Jenkins pipeline alkalmazása gépi tanulási modellek folyamatos integrációjára [15]

1.3 A munka állapota, készültségi foka a félév elején

Közvetlenül a feladathoz nem volt elkészült kódom vagy leírásom, de előző félévben a Deep Learning a gyakorlatban Python és LUA alapon nevű tárgyban hallottakat és az arra a tárgyra készült kódot alapként felhasználtam az önálló laboratóriumi munkához, valamint az előző félévben már megismertem a témalaboratórium keretében a Docker és Linux működésének az alapjait. Ezeken kívül nem volt más előismeretem a munkához kötődő technológiákhoz, a Jenkinst az önálló laboratórium keretén belül ismertem meg.

2. Az elvégzett munka és eredmények ismertetése

2.1 A tanításhoz használt adatcsomag kiválasztása

A félév elején a munkához először egy szegmentációhoz használható adatcsomagot kellett találnom, ehhez a kaggle nevezetű, Deep Learninges versenyeket is szervező weboldalon kerestem elsődlegesen számomra megfelelő adatcsomagokat. Fontos tényező volt, hogy a szegmentációhoz szükséges előfeldolgozási munkák már készen legyenek, vagyis legyen valamilyen szegmentációs maszk definiálva a képekhez. Emiatt már az elérhető adatcsomagok egy jelentős része nem volt megfelelő, mivel, vagy csak bounding boxok koordinátáit tartalmazta, vagy egyáltalán nem volt a csoportosításon kívül semmilyen előmunka elvégezve. Ezekkel a szempontokkal több adatcsomagot is megvizsgáltam [9] [10] [11] [12]. Ezek közül a [9] autó alkatrészeket tartalmazó adatcsomag először tűnt csak egy jó választásnak, de a képek csak az objektumot tartalmazták, és nem volt semmilyen szegmentáció, így ezzel többet nem is foglalkoztam. A [12] ipari hibákról készült adatcsomag alapvetően a szegmentációs követelményt teljesíti, de mivel csak ipari környezetben készült hibás és jó termékeket tartalmaz, ezért végül nem választottam, mivel a többi lehetőség sokkal diverzebb csoportokat tartalmazott. A [11] drónnal készített képcsomag alapvetően szegmentációval is rendelkezik, emiatt egy megfelelő választás lehetne, de a képek perspektívája miatt nem használtam mégsem ezt, mivel csak felülről készült képeket tartalmaz, így a tanított modell nem lenne képes emberi magasságból megfelelő szegmentációra és identifikációra. Végül a [10] PASCAL VOC0712 adatcsomagra esett a választásom, bár nem mindegyik képhez tartozik szegmentáció, de így is elegendő számú képhez tartalmaz szegmentációs maszkot.

2.2 Adatok előkészítése a tanításhoz és a tanítás

A feladatot először több jól elkülöníthető részfeladatra bontottam, amiket egy Jupyter notebook nevű formátumban különálló blokkokban valósítottam meg, ezt később már elkészült állapotában különálló fájlokba is átmásoltam, az automatizálás megvalósításához.

A tanításhoz először elő kellett készítenem az adatokat. Ehhez a beolvasott képekhez tartozó szegmentációs bitmaszkoknak leellenőriztem a kódolását, mivel észrevettem, hogy nem a klasszikus RGB színkódolást használta, hanem egy P nevezetű egyszerűsített színpalettát, amiben minden elérhető színhez egy számérték van rendelve. Emiatt egy rövid kóddal kiírattam a bitmaszk által használt P színpaletta színeihez tartozó pixelszámokat, mivel a maszk nagy része fekete, ezért ebből könnyen meghatározható, hogy a 0 számú színhez a fekete tartozik, ami számunkra egy később kihagyandó szín. Ezen kívül a szegmentációkat egy azonos színű edge/”szél” vette körül, amik még nem az objektumot jelölték, csak a szegmentáció végét, így ezt is kihagytam a későbbi feldolgozáskor, ennek pedig a 255 szám volt az azonosítója.

Miután megállapítottam a színek kódjait, beolvastam a képekhez tartozó xml fájlokat, amik a képen található szegmentáció azonosításához szükséges bounding boxok koordinátáit, és a hozzájuk tartozó osztályokat tartalmazták egyéb számomra irreleváns információk mellett.

Ezután az osztályokat, maszkokat és a bounding boxokat felhasználva átalakítottam a bitmaszkokat egy YOLO által használt szegmentációs formátumra, ami a következőképp épül fel: „osztály név, x1, y1, x2, y2, x3, y3 … xn, yn” Egy sor egy teljes zárt szegmentációt kell tartalmazzon. Ennek az átalakításnak a folyamata a következő:

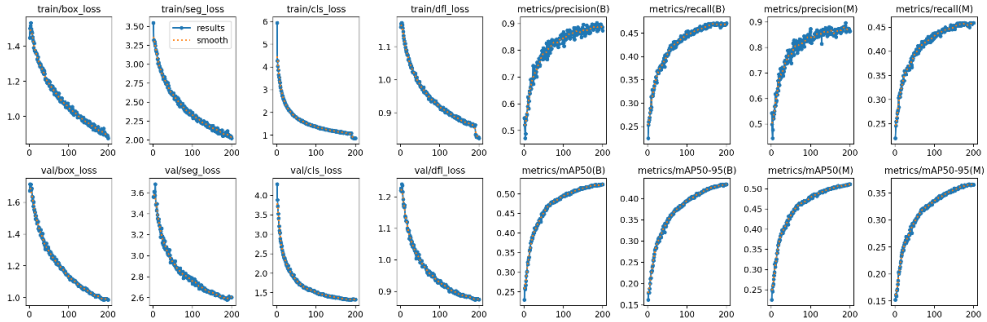
Mielőtt beolvastam egy képet, még létrehoztam egy 1 pixel széles paddinget / keretet a kép körül, mert enélkül az alkalmazott könyvtár nem tudta kezelni a kép szélein található szegmentációkat. A feldolgozás során a bounding boxok segítségével megállapítottam a

szegmentációt tartalmazó régióját a képnek, és az ott legtöbb nem fekete és edge színű (0 és 255 azonosítójú színek) pixelcsoport kerülete mentén óra forgásának megfelelő irányban beolvastam, a kiválasztott csoportot a pixelcsoportok területe alapján és a köréjük rajzolt bounding box xml fájlban található bounding boxszal való összehasonlításával állapítottam meg. Ezt követően a létrejött adatot képenként egy szöveges fájlba írtam már a YOLOnak megfelelő előbb ismertetett formátumban. A következő lépés előtt leellenőriztem a létrejött maszkokat, az alapként használt maszkokkal.

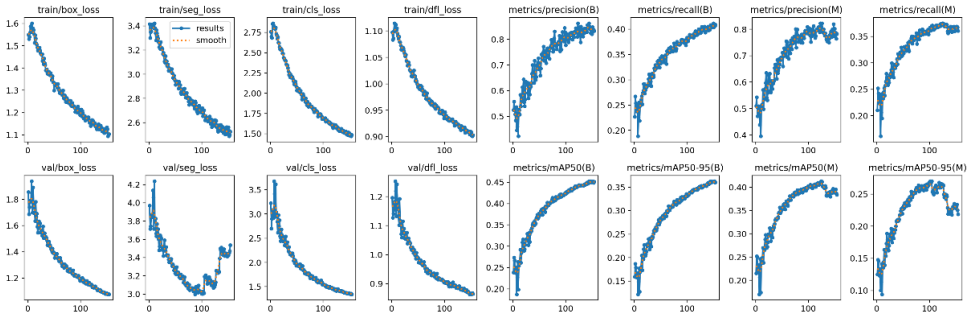
A következő lépésben a generált címkéket, vagyis a maszkok koordinátáit tartalmazó text fájlokat és a hozzájuk tartozó képeket 70,15,15 százalékos arányban a train, validation és test mappába másoltam, amiknek az útvonalát a YOLO YAML fájlában ennek megfelelően definiáltam a csoport nevekkel együtt.

Az előkészitett mappák és YAML fájl segítségével már elindítható egy tanítás, de mielőtt ezt megtennénk, van lehetőség a YOLO tune funkciójának a segítségével egy optimális hiperparaméter keresésre. Ekkor a YOLO az általunk megszabott epoch és iteráció szám alapján futtatt több tanítást változó hiperparaméterekkel, és a legjobb modellhez használt hiperparamétereket megadja. A tune funkcióhoz tudunk a normál tanításhoz hasonlóan megszabni hiperparamétereket, én az epoch és iteráció számon kívül egy worker számot (4) és a batch -1 es, vagyis automatikus batch szám választást állítottam be.

A tune elvégzése után egy 200 epochból álló tanítást indítottam több hiperparaméter beállítással is. Ennek oka, hogy az első esetben még nem ismertem a tune funkciót, így az alap hiperparamétereket használtam. A második tanitás esetében az auto módra állított optimizáló miatt a tune alapján beállított hiperparaméterek egy részét ignorálta a modell. Harmadik esetben már beállítottam az AdamW optimizálót, így ekkor már az összes hiperparaméter megfelelően működött, később ezt a tanítást tovább tanítottam egy 30as patience beállítással, ami azt jelenti, hogy ha nem lát javulást a tanításban 30 epoch óta, akkor leállítja azt. A tanítási eredmények az auto optimalizáló beállítással, de tune hiperparaméterekkel a 4. ábrán láthatóak. Az ábrából, amelyen az x tengely az epochok, az y pedig a loss esetén a hibás érzékeléseket, a metrikák pedig a különböző pontosságokat jelölik, látható, hogy a 200 epoch végén még minimálisan, de javult a loss arány minden kategóriában, úgyhogy még folytatható lenne a tanítás, de ezzel a beállításokkal nem futtattam több epochot. A dokumentum végén található GitHub linken elérhető a többi modell hasonló statisztikái, Ezen kívül az AdamW optimalizálót használó tanítást tovább tanítottam a patience beállítással, aminek a segítségével megállítható a tanítás, amikor az már nem javul, erre mutat példát az 5.ábra, ahol látszik, hogy 150 epoch körül leáll, mivel nem javul az eredmény. A többi eset megtalálható a GitHubon található repository leírásában. A 6. ábrán egy kép szegmentációja látható a különböző modellek használatával. Itt látható, hogy bár arra számítanánk, hogy a tune hiperparamétereit használó tanítás adja a legjobb eredményt, mégis a patience 30as beállítást tartalmazó tovább tanítás nem ismert fel a háttérben egy üveget. de az ember szegmentációját jobban végezte, mint a többi modell.



4. ábra: YOLO tanítás Auto optimalizálóval és beállított hiperparaméterekkel



5. ábra: AdamW tovább tanítás patience 30 beállítással

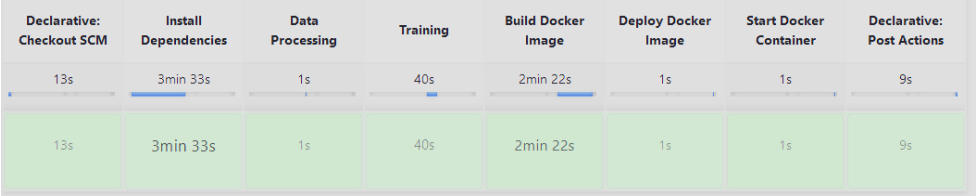
6. ábra: A 4 modell szegmentációja, 1 a tanításban használt képen

2.3 A kész tanítás átültetése Jenkins rendszerbe

Miután elkészültem a YOLO tanítással, a témavezetővel megbeszéltek alapján elkezdtem a kész kódot egy általam létrehozott Jenkins rendszerbe átültetni. Mivel eddig nem ismertem a Jenkinst, ezért először egy YouTube-on megtalálható [13] kb. 1 órás angol nyelvű oktató videót néztem meg, ami alapján már volt egy alap szintű tudásom a Jenkins működésével és létrehozásával kapcsolatban, ezen kívül a Jenkins honlapján található dokumentációt és ott található szöveges tutorial lapokat használtam fel [14].

Ezek alapján egy Dockerhub-ról letöltött Docker képből létrehoztam egy konténert, amiben a BlueOcean kezelőfelületet is tartalmazó Jenkins szerver futtatható volt. A megfelelő (8080 és 50000) portok kinyitásával és az általam létrehozott bridge hálózatra csatlakoztatással elindítottam a konténert. Ezután létrehoztam egy másik docker konténert, aminek a segítségével a hoszt gépen található Dockeren tudtam a Jenkinsen keresztül új konténereket létrehozni, így szimulálva egy, a Jenkins szervertől független docker szervert. Ezen kívül még létrehoztam egy saját magam által készített docker képet, amiben egy alap Debian Linuxot tartalmazó képre telepítettem egy Dockert, Pythont és minden, a tanításhoz szükséges Python csomagot (például numpy, ultralytics stb.). A Jenkins folyamat végén létrehozott Docker képhez még készítettem erre a Python csomagra épülő képet, amiben még telepítettem egy Python Flask csomagot.

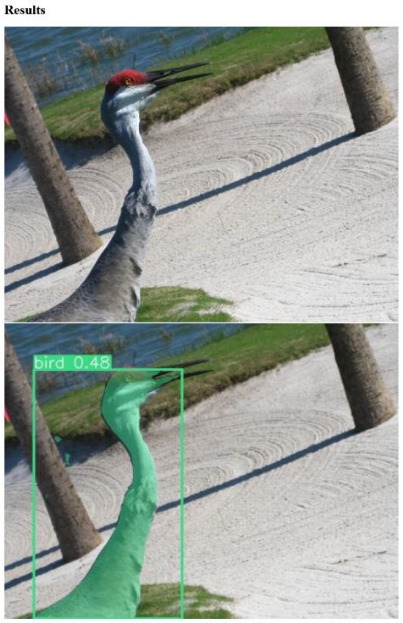
A Jenkins szerveren beállítottam a home könyvtár útvonalát és a docker hoszt szerver IP címét, amihez előzőleg leírt konténert használtam. Ezt követően beállítottam egy Docker agent mintát, amin a későbbi munkák futnak. A pipeline munka létrehozásánál több dolgot is beállítottam, ezek a következőek: Csak az előző 5 fordítást tartja meg, ezeknél régebbieket törli. A munka kódját a GitHub repositorymból tölti le, és az SCM: H * * * * beállítás miatt minden órában ellenőrzi, hogy történt-e változtatás a repositoryn, ha igen újra fordítja a kódot, a H egy hashelést jelent, aminek a segítségével egyenletesen eloszthatóa terhelés. A SCM karakterei a következő beállításokat takarják 1.Perc, 2. Óra, 3.Nap, 4.Hónap,5.Hét napjai. Ez a változás lehet egy változás a kódban, vagy új / törölt fájl, így új képek esetén is újra fordul, és így új tanítást végez. A pipeline a 7. ábrán látható részek/ stagekből áll, ezt a repositoryban megtalálható jenkinsfile definiálja.



7. ábra: Jenkins Pipeline egy sikeres futás után

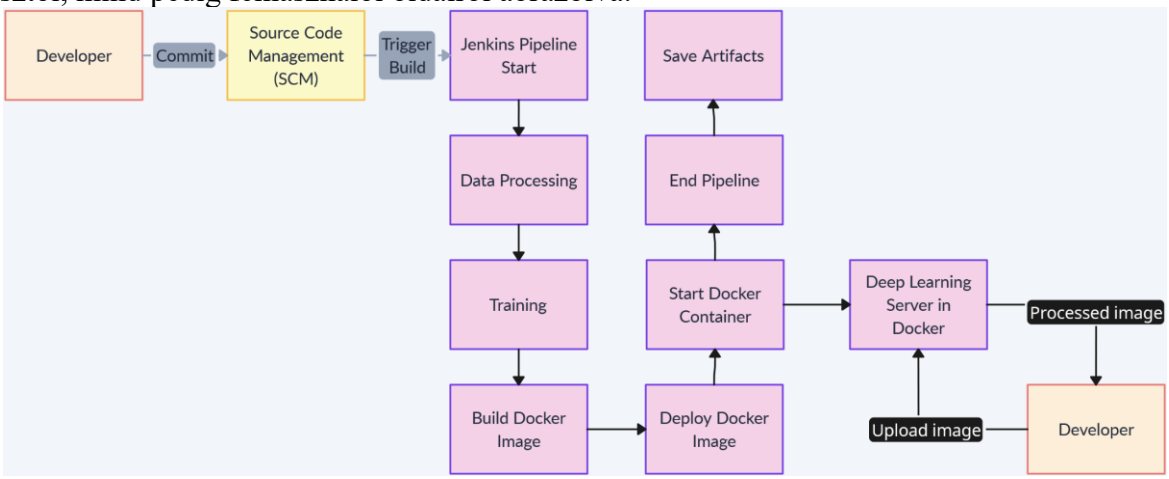
Mint a képen is látható, a tanítást felbontottam több különálló folyamatra, amik külön Python fájlokban találhatóak, így az adatfeldolgozás és a tanítás egymástól elválasztva, de egymás után történik. A kód lefutása előtt még egyszer megpróbálja telepíteni a kódhoz használt csomagokat, amiket a requirements.txt fájlból ér el, így, ha később új csomagra lenne szükség, nem kellene új Docker képet létrehozni, elég lenne a text fájl módosítása. A tanítás után pedig létrehoz egy docker képet, amiben már a kész modell található egy predict.py kóddal társítva, aminek a segítségével futtatható egy Flask csomag alapú webszerver, amin a modell már használható, ez látható a 8. és 9. ábrán. Az itt látható weboldal egy nagyon egyszerű HTML kódon alapul, mivel a feladatnak nem ez volt a fókusza.

8. ábra: fájlfeltöltés a modell használatához



9. ábra: A weblapon keresztüli szegmentáció

A példában az előzőleg betanított modellt használja a predict.py, mivel a Jenkins pipelineon akár kb.10 óráig is eltartana egy hasonló minőségű tanítás, míg a 7. ábrán látható 40 másodperc esetén az elkészült modell nem képes semmilyen detekcióra. Az elkészült Docker képet ezután feltölti a Dockerhubra, ahonnan a következő stagen a szerver elindítja azt, és ekkor már a Jenkins szervertől független módon fut. Az utolsó fázisban a pipeline elmenti a runs mappa tartalmát, így megőrizve a tanítás eredményét, minden mást pedig töröl. A teljes Jenkins-es folyamat a 10. ábrán látható, mind fejlesztői, mind pedig felhasználói oldalról ábrázolva.



10. ábra: A program teljes futásának és használatának bemutatása

2.4 Összefoglalás

A félévi munka alatt sikerült elmélyítenem mind a Linuxal, Dockerrel, Pythonnal és YOLO-val kapcsolatos tudásomat. Ezen kívül megismertem a Jenkins rendszert és megtanultam a CI/CD elven alapuló fejlesztést. Számomra még a Git/GitHub használata is új volt, a Jenkins-re történő kódírás már teljes mértékben a GitHubra történő commitolásokkal történt, előtte csak egyszerűen feltöltöttem a kódot, de a félév végére elsajátítottam a Git féle szemlélet alapjait.

A kész feladat önmagában már egy működő rendszer, egy megfelelő hardverrel felszerelt szerveren már feltehetőleg jól használható, ennek a tesztelésére nincs eszközöm.

A lehetséges továbbfejlesztési irány lehet akár egy jobb weboldal készítése, egy jobban optimalizált modell vagy videó adatfolyam kezelése. Erre a YOLO ad lehetőséget a stream paraméter használatával, de ehhez véleményem szerint már az általam használt hardver nem elegendő.

Összeségében véleményem szerint a félév alatt sikerült sok új a gyakorlatban is használt technológiát megismernem, és megtapasztalnom milyen önképző módon új rendszerek használatát elsajátítani.

Irodalom, és csatlakozó dokumentumok jegyzéke

2.5 A tanulmányozott irodalom jegyzéke:

3 Irodalomjegyzék

[1] "Revolution of object detection," [Online]. Available: https://medium.com/analytics-vidhya/evolution-of-object-detection-582259d2aa9b. [Accessed 07 05 2024].

[2] „Top models for instance segmentation," [Online]. Available: https://keylabs.ai/blog/top-models-for-instance-segmentation-reviewed/. [Hozzáférés dátuma: 07 05 2024].

[3] Wong Kin-Yiu, „YOLOv7 Repository," [Online]. Available: https://github.com/WongKinYiu/yolov7. [Hozzáférés dátuma: 07 05 2024].

[4] Chien-Yao Wang, Alexey Bochkovskiy, Hong-Yuan Mark Liao „YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," 2022. [Online]. Available: https://arxiv.org/abs/2207.02696. [Hozzáférés dátuma: 07 05 2024].

[5] „Roboflow Yolov7 Breakdown," [Online]. Available: https://blog.roboflow.com/yolov7-breakdown/. [Hozzáférés dátuma: 07 05 2024].

[6] „Ultralytics YOLOv8 segmentation models," [Online]. Available: https://docs.ultralytics.com/tasks/segment/#models.

[7] „Ultralytics hyperparameters," [Online]. Available: https://docs.ultralytics.com/guides/hyperparameter-tuning/#what-are-hyperparameters. [Hozzáférés dátuma: 08 05 2024].

[8] „Docker Dokumentáció," [Online]. Available: https://docs.docker.com/engine/. [Hozzáférés dátuma: 08 05 2024].

[9] „Kaggle Car Parts," [Online]. Available: https://www.kaggle.com/datasets/gpiosenka/car-parts-40-classes?resource=download.

[10] „Kaggle PASCAL," [Online]. Available: https://www.kaggle.com/datasets/bardiaardakanian/voc0712. [Hozzáférés dátuma: 09 05 2024].

[11] „Kaggle Drone Dataset," [Online]. Available: https://www.kaggle.com/datasets/santurini/semantic-segmentation-drone-dataset/data. [Hozzáférés dátuma: 09 05 2024].

[12] „Kaggle Industrial Defect," [Online]. Available: https://www.kaggle.com/datasets/aryashah2k/few-shot-industrial-defect-detection. [Hozzáférés dátuma: 09 05 2024].

[13] D. Journey, „Learn Jenkins! Complete Jenkins Course - Zero to Hero," [Online]. Available: https://www.youtube.com/watch?v=6YZvp2GwT0A. [Hozzáférés dátuma: 10 05 2024].

[14] „Jenkins Documentation," [Online]. Available: https://www.jenkins.io/doc/. [Hozzáférés dátuma: 10 05 2024].

[15] P. Varga, A. Kővári, M. Herkules, Cs. Hegedűs, „MLOps in CPS – a use-case for image recognition in changing industrial settings", IEEE/IFIP Network Operations and

Management Symposium, Seoul, Korea, 2024.

3.1 Csatlakozó egyéb elkészült dokumentációk / fájlok / stb. jegyzéke:

Az elkészült forráskód: https://github.com/arminfal/Onlab_YOLO_PASCAL