

Flowium Dokumentáció

py-snake

2025. december 7.

Kivonat

Ez a dokumentum a Flowium projekt részletes dokumentációja. A Flowium fantázianévre hallgató projekt egy olyan forgalomfigyelő és előrejelző rendszer, ami többféle docker konténer szolgáltatására épül. Maga a program Pythonban készült, használ hozzá Dockert és több nyílt forráskódú könyvtárat is, mint például yt-dlp, ffmpeg, opencv2, yolov8. A rendszer képes élő videófolyamot rögzíteni, fel is ismeri, illetve követni tudja a járművek mozgását. Az adatokat sqlite adatbázisban eltárolja, és gépi tanulás segítségével próbálja megjósolni, hogy milyen lesz a forgalom a későbbiekbén.

Tartalomjegyzék

1. A Projekt Kiindulópontja és Ötlete	3
1.1. Bevezető	3
1.2. Élő kamerakép	3
1.3. Kamerakép előfeldolgozása	3
1.4. Járműdetektálás az előfeldolgozott képeken	3
1.5. Adatrögzítés, mentés	4
1.6. Valós idejű tanulás	4
1.7. Kiegészítő adatok, például időjárás és környezeti tényezők	4
1.8. Futtatási környezet, rendszerek összekapcsolása	4
1.9. Használni tervezett rendszerek	5
1.10. Mintaképek	5
2. Rendszerarchitektúra	7
2.1. Docker Compose áttekintés	7
3. Alapvető algoritmusok	9
3.1. Járműkövető (<i>tracker.py</i>)	9
3.2. Atomi írási mechanizmus (<i>stream-capture</i>)	9
3.3. Adaptív előfeldolgozás	9
3.4. Online tanulási folyamat	10
4. Szolgáltatások	10
4.1. Stream Capture (<i>stream-capture</i>)	10
4.1.1. Cél	10
4.1.2. Belső logika	10
4.1.3. Függőségek	11
4.1.4. Konfiguráció	11
4.2. Preprocessing (<i>preprocessing</i>)	11
4.2.1. Cél	11
4.2.2. Belső logika	11
4.2.3. Függőségek	11
4.2.4. Konfiguráció	12
4.3. YOLO Detector (<i>yolo-detector</i>)	12
4.3.1. Cél	12
4.3.2. Belső logika	12
4.3.3. Függőségek	12
4.3.4. Konfiguráció	12

4.4. Data Manager (<code>data-manager</code>)	12
4.4.1. Cél	12
4.4.2. Belső logika	13
4.4.3. Függőségek	13
4.4.4. Konfiguráció	13
4.5. Online Learner (<code>online-learner</code>)	13
4.5.1. Cél	13
4.5.2. Belső logika	13
4.5.3. Függőségek	13
4.5.4. Konfiguráció	14
4.6. Weather Service (<code>weather-service</code>)	14
4.6.1. Cél	14
4.6.2. Belső logika	14
4.6.3. Függőségek	14
4.6.4. Konfiguráció	14
4.7. Web UI (<code>web-ui</code>)	15
4.7.1. Cél	15
4.7.2. Belső logika	15
4.7.3. Függőségek	15
4.7.4. Konfiguráció	15
5. API és Adatmodellek	15
5.1. Data Manager	15
5.1.1. POST /detections	15
5.1.2. POST /weather	16
5.2. Preprocessing	16
5.2.1. POST /preprocess	16
5.3. Online Learner	16
5.3.1. POST /train	16
5.3.2. POST /predict	16
6. Vizualizációk	17
6.1. Rendszerarchitektúra Diagram	17
7. Konfiguráció	17
8. Telepítés és Üzemeltetés	17
8.1. Helyi Telepítés	17
8.1.1. Előfeltételek	18
8.1.2. Telepítési Lépések	18
9. Továbbfejlesztési Lehetőségek és Kiegészítések	18
9.1. YOLO Modell Optimalizáció	18
9.1.1. ONNX Export	18
9.1.2. Teljesítmény Összehasonlítás	18
9.2. Web UI Kiegészítések	19
9.2.1. Analytics Tab - Legutóbbi Detekciók	19
9.2.2. Predictions Tab	19
10. Képek a Web-es felületről	19

1. A Projekt Kiindulópontja és Ötlete

1.1. Bevezető

BSc tanulmányaim során mások projektjeit, szakdolgozatát figyelte felkeltette az érdeklődésem az objektum detektálás képeken, valósidejű videón. Ugyan érdeklődésem megvolt a téma iránt, de soha nem volt szükségem ilyen jellegű projektre a hétköznapokban. A féléves feladat kapcsán ötletelem, átgondoltam hogy lehetne összekötni ezt az évek óta meglévő kíváncsiságomat és a tantárgy követelményeit. Egy barátom által fejlesztett rendszerben megláttam a valósidőben tanuló neurális hálózatok szépségét, és a Tanár úr által mondott órai példa, jármű detektálás alapján jött a következő ötlet: járműdetektálás valós időben elő kamera képről és valósidőben tanuló neurális hálózat tanítása forgalmi adatok becslésére, időpont és akár időjárási körülmények figyelemben.

A tervem egy forgalombecső alkalmazás elkészítése, amely egy közúti kamera képe alapján tanul. Sok online, előben elérhető közterületet vagy közutat figyelő kamerát átnézem, de legnagyobb bánatomra ezek annyira voltak elők, hogy percenként 1 képet rögzítettek. Jobb esetben 10-15 másodpercenként érkezett egy kép. Ezek a célra egyáltalán nem felelnek meg, ennyi idő alatt számtalan jármű elhaladhat alattuk. Végül az összes magyar kamerás oldal átnézése után eszembe jutott a YouTube elők átnézése mint opció, és találtam is egy 2023 óta folyamatosan közvetítő Bajai kamera képet, amely egy parkra és közútra néz, 1920x1080 pixel felbontással és 30fps képkockasebességgel rendelkezik. Ez a forrás véleményem szerint megfelelő a feladatra. Amennyiben nem sikerül használható minőséget kapni róla, külföldi kamerára vagy rögzített videó fájra való átállás lesz szükséges.

A következő fejezetekben a technikai megvalósítás lehetségeit megközelítéseit vázolom, ami persze előretekintve jó ötletnek tűnik, de a megvalósítás során változhat.

1.2. Elő kamerakép

A bevezetőben ismertetettem, YouTube-on elérhető Bajai Sugovica-sétányra néző kamera a tervezett videó forrás. Technikai adatai közé tartozik az 1920x1080 pixel képfelbontás és 30fps képkockasebesség. Az adás élő, 7/24 közvetített. A látott kép 3 fő szekcióra osztható: égbolt (időjárási viszonyok, napszak, természeti jellemzők), gyalogos sétány (szerencsés esetben követhető lenne a gyalogos forgalom, de jelenlegi feltételezésem alapján túl apró a kép hozzá), közút (sajnos az út egy részét fák takarják, de egy szakaszon tökéletes rálátás van a közlekedő járművekre). A kamera nappali fényviszonyok mellett kiválasztott közvetít, az éjszaka során az erős kivilágításnak köszönhetően sincs túlságosan sok zaj a képen.

Az elő streamet yt-dlp, ffmpeg és opencv segítségével tudnám betölteni és hasznosítani.

1.3. Kamerakép előfeldolgozása

Mint korábban említettem, a kamera nagy területet fog be, ezeket különbözően lehetne hasznosítani. Elsősorban a járművek detektálására szeretnék koncentrálni, ehhez egy körülbelül 800x400 pixel területre kéne koncentrálni.

A hardveres erőforrásaim erősen korlátozottak, GPU egyáltalán nem áll rendelkezésre, tehát a lehető leghatékonyabban kell elkészíteni a programot. A feldolgozásra vagy laptop processzor (Intel(R) Core(TM) i5-8350U CPU @ 1.70GHz, 4C/8T) vagy bérelt vps dedikált 4 magja (AMD EPYC 9634 84-Core Processor @ 2.20GHz, 4C/4T) áll rendelkezésre.

A beérkező 1920x1080 képből első lépésként a közutat rögzítő, körülbelül 800x400 pixeles kép kivágását kell megtenni, majd esetlegesen napszaktól, időjárási viszonyuktól függően opencv segítségével hiszogram kiegyenlítést, kontrasztfokozást, zajszűrést végezni. A bejövő 30fps helyett valószínűleg 4-8fps közötti feldolgozási sebesség lesz reális a rendelkezésre álló hardveren, tehát minden 4-7. frame fog feldolgozásra kerülni, a többi eldobásra.

1.4. Járműdetektálás az előfeldolgozott képeken

A kamerakép oldalnézetből mutatja a közutat, de még így is látható az elhaladó járművek alakja, mozgása. A detektálásukhoz YOLOv5 vagy YOLOv8 alapú megközelítést tartok reálisnak. Ezzel a módszerrel a tervezek szerint felismerhető és követhető lesz a járművek mozgása valósidőben. A detektálás körülmenyeit, mint dátum, időpont, (időjárási körülmények) egy csv állományban vagy esetleg adatbázisban tárolnám, valamint átadnám a valósidőben tanuló neurális hálónak. A rögzített adatok később jó dataset-ként szolgálhatnának, mivel valós környezetben, több paraméter feljegyzésével lennének tárolva. Ebből később

más neurális hálókat is lehetne tanítani, esetleg teljesítményt összehasonlítani. Forgalmat becsülni a jövőre nézve.

A mozgást ugyan lehetne opencv alapokon detektálni, de tartok tőle hogy az oldalnézet, távoli kamerakép miatt nem birkózna meg a feladattal az elvárt pontossággal neurális háló bevetése nélkül.

1.5. Adatrögzítés, mentés

A detektált objektumokról készülő feljegyzéseket egyszerű csv szöveges állományban is lehetne tárolni, de ha hosszútávú üzemeltetés a cél, jobb lenne valamilyen adatbázist használni, például egyszerű helyi SQLite alapú tárolást. Ez a megközelítés jelentősen gyorsítaná nagy mennyiséget adat tárolását, és nem járna nagy többlet erőforrás igénygel. A feldolgozás jelentősen gyorsul, lekérdezések, keresések, statisztikák, exportálás, biztonsági mentés könnyedén végezhető.

A tárolni kívánt adatmezőket a későbbiekben szeretném pontosan meghatározni attól függően, hogy milyen adatok lesznek elérhetők (például az időbeli esetleg autó típus méret alapján, ideiglenes egyedi azonosító a járműre, konfidenca). Amennyiben időjárásadatok és esetleg más, a forgalmat befolyásoló tényezők tárolása bevezetésre kerülne, további adatmezőket kell felvenni.

1.6. Valós idejű tanulás

A terv egy online tanuló rendszer elkészítése, amely a beérkező adatokat feldolgozva folyamatosan képes a tanulásra. A megvalósításhoz a River python csomag illeszkedne, mivel online tanuló algoritmusok vannak benne implementálva. A módszer előnye, hogy nem lenne időszakosan kimagsoló erőforrásigénye, hanem egy folyamatos egyenletes terhelést jelentene a rendszer számára. Ezzel a megközelítéssel nem lenne szükség a régi adatok újra tanulására, hanem a modell folyamatosan frissítésre kerülne.

1.7. Kiegészítő adatok, például időjárás és környezeti tényezők

A forgalmat és a járműdetektálás sikerességét egyaránt befolyásolhatják az időjárási tényezők, ezért véleményem szerint érdemes lenne feljegyezni ezeket az adatokat is. Az időjárási viszonyok hatást gyakorolhatnak a forgalom sűrűségére, így ezek rögzítésével lehetne elemzni az összefüggéseket, és esetleg pontosabb becslést adni a felépített modellel. A rossz látási viszonyok a járműdetektálást is negatívan befolyásolhatják, mivel esős viharos időben jelentősen romlik a kamera által rögzített kép. Ekkor elkerülhető lehet, hogy ugyan a valóságban megnő a forgalom, de a rossz képminőség miatt csak a járművek töredéke lesz feljegyezve.

Az időjárásadatok forrása lehetne maga a kamerakép, valamilyen módon felismerni a környezeti tényezőket valós időben. Ez nagy erőforrásigénnel járna és korlátozottan lehetne eljárni. Ezzel ellentétes megközelítés egy időjárás API használata, vagy időjárás weboldalról legyűjtött adatok használata. Ennek előnye, hogy jóval kevesebb erőforrásra van szükség hozzá, több környezeti jellemzőhöz tényezőkkel hozzá lehet férfi (hőmérséklet, páratartalom, eső vagy napsütés, napfelkelte, napnyugta, szélérősséggel), viszont hátránya, hogy bizonyos esetekben pontatlan lehet. Pontatlanságot okozhat, ha az online előrejelzés szerint az adott időszakban esőzés van a területen, ám a kamera képe szerint csak felhős az idő, de a valóságban nem esik eső. Ennek kiküszöbölésére hibrid megközelítést lehetne alkalmazni, de ez jelenleg túlmutat a projekt keretein.

A rögzített időjárási adatokkal elemelhetővé válik a forgalmi minta és az időjárás összefüggése, vagy a detektálási hiba alakulása nem várt időjárási körülmények közt.

1.8. Futtatási környezet, rendszerek összekapcsolása

A projekt megvalósítása Python programozási nyelven tervezett, különböző Python könyvtárak felhasználásával. Az egységbe csomagolás, könnyű telepítés és hordozhatóság érdekében a teljes rendszert szerepkörök alapján felbontva, docker alapú konténerekbe szervezve, API alapú kommunikációval tervezem megvalósítani. A tervezett szerepkörök körülbelül megfelelnek a felsorolt pontoknak: élő kamerakép fogadása (yt-dlp), kép előfeldolgozása (ffmpeg, opencv), járműdetektálás (YOLO), adatrögzítés (SQL vagy SQLite), online tanulás (River), kiegészítő szolgáltatások, például időjárás API.

Kategória	Technológia	Felhasználás
Programozási nyelv	Python	Fejlesztési környezet
Videó stream	yt-dlp	YouTube stream letöltése
Videó feldolgozás	Ffmpeg	Videó feldolgozása
Gépi látás	OpenCV	Videó és kép feldolgozása
Objektum detektálás	YOLOv5 vagy YOLOv8	Járműdetektálás és követés
Adatbázis	SQLite	Adattárolás
Online tanulás	River	Valós idejű modell létrehozása
Adatmanipuláció	Pandas	Adatfeldolgozás
Adatmanipuláció	NumPy	Adatfeldolgozás
Webkliens	Requests	API kommunikáció
Webkliens	curl	Web és API kommunikáció
Vizualizáció	Matplotlib	Grafikonok
Vizualizáció	Seaborn	Statisztikai ábrák
API keretrendszer	FastAPI	REST API végpontok
Machine learning	Scikit-learn	Modellek
Konténerizáció	Docker	Környezetek futtatása
Verziókezelés	Git	Fejlesztés követése

1. táblázat. A projekt megvalósításához tervezett technológiai stack

1.9. Használni tervezett rendszerek

1.10. Mintaképek

Nappali és éjszakai mintakép. Jól látható, hogy szép időjárás és megfelelő fényviszonyok között elég jó a képminőség, a feladatnak megfelelő lesz a forrás. Mintakép a járműdetektálás szempontjából releváns területről. Az eltérő időpontok kissé eltérő színvilágú képeket eredményeznek.



1. ábra. Mintakép 1



2. ábra. Mintakép 2



3. ábra. Mintakép 3

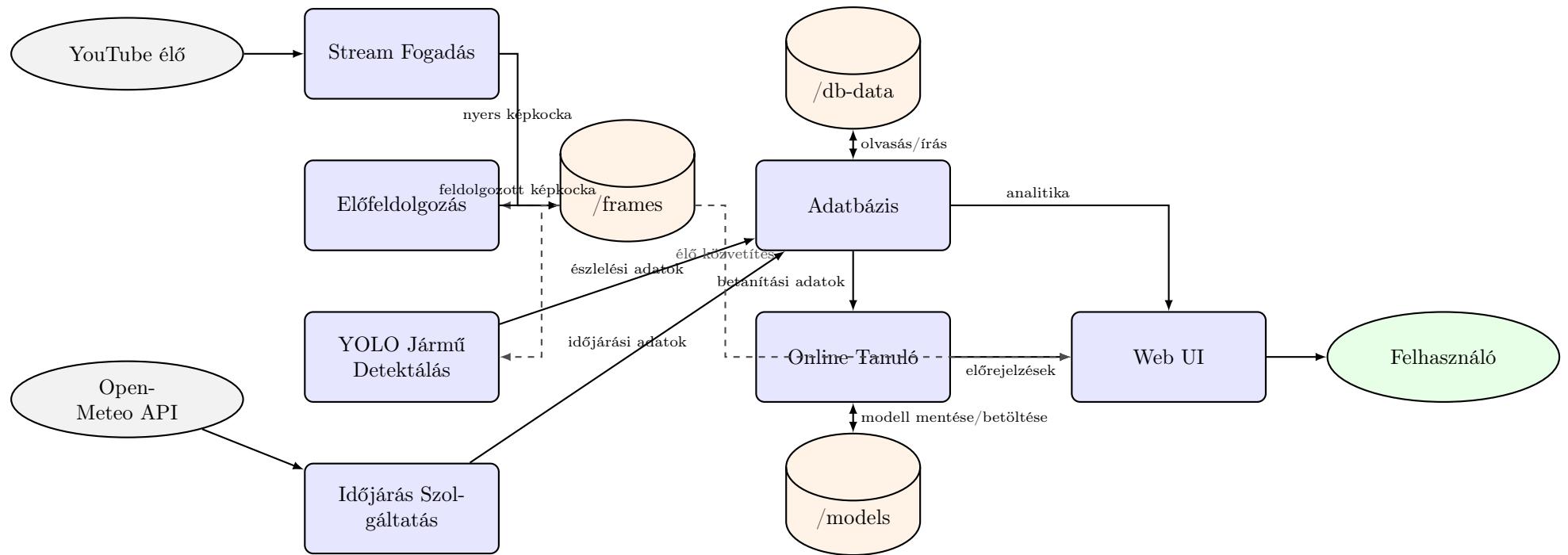
2. Rendszerarchitektúra

A Flowium projekt mikroszolgáltatás elven épül fel. Az egyes szolgáltatások Docker konténerekben futnak, ezek indítását és kezelését pedig a Docker Compose végzi. Ez a megoldás azért előnyös, mert a rendszer így jó modularitással rendelkezik, könnyen bővíthető vagy karbantartható, és a skálázhatóság is biztosított.

2.1. Docker Compose áttekintés

A rendszer működésének központja a `docker-compose.yml` fájl, amelyben pontosan szerepel, hogy melyik szolgáltatás mit csinál, és hogyan kapcsolódna egymáshoz. Összesen hétfő szolgáltatás tartozik a rendszerhez, ezek egy közös, `flowium-network` nevű Docker hálózaton keresztül kommunikálnak. Az adatok tárolásához Docker volume-ok, vagyis virtuális kötetek kerülnek felhasználásra.

- **stream-capture:** Ez a szolgáltatás a YouTube-ról érkező élő videófolyam fogadásáért felel.
- **preprocessing:** Itt zajlik a bejövő videó képkockáinak előfeldolgozása.
- **yolo-detector:** Ebben a lépésben történik a járművek felismerése az előfeldolgozott képeken.
- **data-manager:** Ez a modul tartalmazza az adatbázist és tárolja az összes begyűjtött adatot.
- **online-learner:** Itt valósul meg a gépi tanulás, ami a forgalmi minták előrejelzésére szolgál.
- **weather-service:** Ez a szolgáltatás szerzi be az aktuális időjárási adatokat.
- **web-ui:** A rendszerhez tartozó webes felület, amelyen keresztül minden funkció elérhető és követhető.



4. ábra. Részletes Flowium Rendszerarchitektúra, amely bemutatja a szolgáltatásokat, az adatmennyiségeket és az adatfolyamot.

3. Alapvető algoritmusok

Ebben a részben bemutatásra kerülnek azok a kulcsfontosságú algoritmusok, amelyek a rendszer működését biztosítják.

3.1. Járműkövető (tracker.py)

A járműkövető komponens alapvető szerepet tölt be a pontos járműszámok meghatározásában, illetve az adatredundancia elkerülésében. A megvalósítás során egy úgynevezett mohó illesztési algoritmus alkalmazása történik, amely az Intersection over Union (IoU) és a centroid távolság együttes pontszámán alapul.

- **Illesztés:** minden egyes új képkockához a rendszer minden létező nyomvonalhoz kiszámít egy illesztési pontszámot az újonnan észlelt objektumokra vonatkozóan. Az algoritmus az IoU értéket (ami a térbeli átfedést méri) és a normalizált centroid távolságot (ami a közeliséget reprezentálja) kombinálja. Ezáltal lehetővé válik a gyorsan mozgó objektumok hatékony követése.
- **Mohó hozzárendelés:** Az illesztések a pontszámok alapján kerülnek sorba rendezésre, és minden legjobb pontszámú párok kerülnek először hozzárendelésre. Ez a módszer gyors és hatékony alternatívát nyújt a bonyolultabb, optimális hozzárendelési algoritmusokkal szemben.
- **Nyomvonal életciklusa:**
 - **Létrehozás:** minden olyan észlelés, amelyhez nem tartozik illesztett nyomvonal, új nyomvonalat hoz létre.
 - **Frissítés:** Ha illesztés történik, az adott nyomvonal pozíciója frissítésre kerül, és a `hits` számláló értéke növekszik.
 - **Törlés:** Azok a nyomvonalak, amelyeket a `max_age` paraméterben meghatározott számú képkockán keresztül nem észlel a rendszer, törlésre kerülnek.
- **Megerősítés és hatékonyság:** Egy nyomvonal csak akkor tekinthető megerősítettnek, ha azt legalább `min_hits` képkockán keresztül folyamatosan észleli a rendszer. A megoldás lényege, hogy a jármű adatai csak akkor kerülnek egyszer az adatbázisba, amikor a nyomvonal megerősítetté válik. Ezt a folyamatot a nyomvonal adataiban található `stored` jelző, valamint a `get_confirmed_new_tracks` metódus kezeli, így az adatbázisba történő írások száma jelentősen csökkenhető.

3.2. Atomi írási mechanizmus (stream-capture)

Ebben a részben bemutatásra kerül, hogy a `stream-capture` és a `preprocessing` szolgáltatások hogyan alkalmaznak atomi írási mechanizmust annak érdekében, hogy az alsóbb rétegbeli szolgáltatások ne olvassanak be részben megírt vagy sérült képfájlokat.

A projekt készítése közben felmerült a probléma, hogy az egyik konténer által még csak részlegesen mentett képfájlokat próbált beolvasni egy másik konténer és emiatt hibás adatok készültek.

1. Először egy folyamat (például az `ffmpeg`) egy ideiglenes fájlba írja a képadatokat (`latest_writing.jpg` néven).
2. Egy külön szál folyamatosan figyeli ezt az ideiglenes fájlt, és ellenőrzi annak módosítási idejét, illetve méretét.
3. Amint a fájl egy rövid ideig stabilan változatlan marad, az írás befejezettnek minősül.
4. Ezt követően a figyelő szál egy atomi `os.rename` művelettel átnevezi az ideiglenes fájlt a végleges helyre (például `latest.jpg`-ra). Ezután már használhatóvá válik a teljes fájl.

3.3. Adaptív előfeldolgozás

Ebben a részben bemutatásra kerül, hogy a `preprocessing` szolgáltatás hogyan alkalmaz adaptív eljárásokat a képek optimalizálására, amely különösen fontos a YOLO észlelő számára eltérő fényviszonyok mellett.

- **Fényerő észlelése:** A képek először HSV színtérre kerülnek átalakításra. Ezután a V (érték/-fényerő) csatorna átlagos értéke kerül kiszámításra annak eldöntésére, hogy a jelenet nappali vagy éjszakai.
- **Éjszakai javítás:** Amennyiben sötét jelenetről van szó, a rendszer gamma korrekciót alkalmaz a fényerő növelése érdekében, majd CLAHE (kontrasztkorlátozott adaptív hisztogramkiegyenlítés) kerül felhasználásra magas klipkorláttal, így a helyi kontraszt is javításra kerül, és előtérbe kerülnek az árnyékos részek részletei.
- **Nappali javítás:** Világos jelenetek esetén enyhébb CLAHE művelet zajlik le, amely segíti a kontraszt javítását, ugyanakkor elkerülhető a túlexponálás.
- **Háttérkivonás:** A szolgáltatás lehetőséget biztosít az OpenCV `createBackgroundSubtractorMOG2` funkciójának alkalmazására, amely során a rendszer statikus háttér statisztikai modelljét hozza létre. Amennyiben ez a funkció engedélyezve van, a nem mozgó objektumok (például fák, oszlopok, épületek) eltávolításra kerülnek a képkockákból, így a járművek izolálhatók és kevesebb zaj jut el az észlelőhöz. A funkció jó, nappali fényviszonyok mellett hatékonynak bizonyult, de sötétedéskor már nem nyújtotta a megfelelő teljesítményt. Az autók a gyors mozgásuk miatt kissé elmosódva, a gyenge fény miatt pedig homályosan látszódtak a funkció használata mellett.

3.4. Online tanulási folyamat

Ebben a részben bemutatásra kerül, hogy az `online-learner` szolgáltatás miként alkalmazza a `river` könyvtárat egy folyamatosan tanuló gépi tanulási modell kialakítására.

1. **Folyamat:** A modell egy `river.compose.Pipeline` objektum formájában valósul meg, amely két fő lépést kapcsol össze.
2. **Standard skálázó:** Az első lépésben a `preprocessing.StandardScaler` működése során a beimeneti jellemzők (például idő, hőmérséklet) normalizálása történik meg az átlag kivonásával és egységnyi szórásra való skálázással. A `river` skálázó folyamatosan frissíti ezeket a statisztikákat az új adatok beérkezésekor.
3. **Lineáris regresszió:** A második lépésben a `linear_model.LinearRegression` használata történik. Ez a modell a skálázott jellemzőkhöz tartozó súlyokat tanulja meg a járműszámok előrejelzése céljából. A súlyok frissítése minden egyes új adatpont esetén az online optimalizálási algoritmusok alkalmazásával valósul meg.
4. **Folyamatos betanítás:** Egy háttérfolyamat rendszeresen lekéri az `data-manager` szolgáltatástól a legfrissebb összesített adatokat, és ezeket folyamatosan betaplálja a modellbe. Ezáltal biztosított, hogy a modell képes legyen alkalmazkodni az újonnan megjelenő forgalmi mintákhoz.

4. Szolgáltatások

4.1. Stream Capture (stream-capture)

4.1.1. Cél

Ebben a részben bemutatásra kerül a `stream-capture` szolgáltatás fő feladata, amely az élő videófolyam fogadását és a képkockák megosztott kötetre történő írását foglalja magában.

4.1.2. Belső logika

A szolgáltatás működése az alábbi lépésekben foglalható össze:

1. **Elő videó forrás:** A rendszer a YouTube elő közvetítését használja adatforrásként, amelynek címe a `YOUTUBE_URL` környezeti változóban kerül megadásra.
2. **Képkocka rögzítése:** A képkockák folyamatos rögzítését az `ffmpeg` és `cv2` könyvtárak segítségével végzi, amelyek lehetővé teszik a stream megnyitását és a képkockák kinyerését.

3. **Proxy:** A YouTube az adatközponti `datacenter` IP-ket általában bot-nak jelöli, ezért szükséges `http proxy` és bejelentkezett Google fiókból kinyert `cookie`-k használata.
4. **Atomi írás:** Az elkeszült képkockákat a rendszer atomi írási eljárással menti a `/frames` megosztott kötetre. Ennek célja, hogy a `preprocessing` szolgáltatás minden épsgben lévő, teljes képkockákat tudjon beolvasni.
5. **Sebesség korlátozás:** A képkockák rögzítési gyakoriságát az FPS környezeti változó szabályozza. Mivel csak CPU-val rendelkező, GPU nélküli gép állt rendelkezésre a megvalósításhoz, tesztek után 5 FPS került meghatározásra felső korlátként, így még lépést tud tartani a CPU a feldolgozással.

4.1.3. Függőségek

- **Bemenet:** Elő videófolyam a YouTube csatornáról.
- **Kimenet:** Képkockafájlok a `/frames` megosztott kötetre.
- **Szolgáltatás függőségek:** Ennél a szolgáltatásnál közvetlen szolgáltatás-függőség nem jelenik meg.

4.1.4. Konfiguráció

Az alábbi környezeti változók alkalmazásával történik a szolgáltatás beállítása:

```
1 # A YouTube elő közvetítésének URL-je.  
2 YOUTUBE_URL  
3  
4 # Frames per second (FPS) korlát. Alapértelmezetten: 5  
5 FPS
```

Environment Variables

4.2. Preprocessing (preprocessing)

4.2.1. Cél

Ebben a részben bemutatásra kerül a `preprocessing` szolgáltatás célja, amely a nyers képkockák fogadását, azok adaptív javítását és végül a képkockák továbbítását végzi a `yolo-detector` részére.

4.2.2. Belső logika

A szolgáltatás működése az alábbi lépésekben összegezhető:

1. **Képkocka olvasás:** A rendszer folyamatosan beolvassa a legfrissebb képkockát a `/frames` megosztott kötetről.
2. **Adaptív javítás:** Az adaptív előfeldolgozási algoritmus kerül alkalmazásra minden képkockán, hogy az aktuális fényviszonyokhoz igazítsa a képet.
3. **Képkocka továbbítás:** Az előfeldolgozott képkockák továbbítása a `yolo-detector` szolgáltatás felé történik, HTTP POST kérések segítségével.
4. **API:** A szolgáltatás egy egyszerű HTTP API-t is kínál, amely lehetővé teszi a konfiguráció futásidőben történő módosítását.

4.2.3. Függőségek

- **Bemenet:** Képkockafájlok a `/frames` megosztott kötetről.
- **Kimenet:** Feldolgozott képkockák a `yolo-detector` szolgáltatás felé.
- **Szolgáltatásfüggőségek:** Közvetlenül a `yolo-detector` szolgáltatáshoz kapcsolódik.

4.2.4. Konfiguráció

A szolgáltatás az alábbi környezeti változóval konfigurálható:

```
1 # A yolo-detector szolgáltatás URL-je. Alapértelmezett: http://yolo-detector:8000
2 YOLO_DETECTOR_URL
```

Environment Variables

4.3. YOLO Detector (yolo-detector)

4.3.1. Cél

Ebben a részben kerül bemutatásra a `yolo-detector` szolgáltatás fő feladata, amely a járművek észleléését végzi a beérkező képkockákon, illetve a nyomvonalak kezelését is ellátja a `tracker.py` segítségével.

4.3.2. Belső logika

A szolgáltatás működése az alábbi lépésekben foglalható össze:

1. **YOLO modell:** A rendszer a YOLOv8 modellt használja arra, hogy észlelj a különböző járműtípusokat (autókat, teherautókat, buszokat, motorokat) az egyes képkockákon.
2. **Nyomvonal kezelés:** minden észlelt objektum információja átadásra kerül a `tracker.py` járműkövetőnek, amely a járművek mozgásának folyamatos követését végzi.
3. **Adatküldés:** A követő által megerősített új járművekről (például észlelési adatok, járműszám) szóló információk HTTP POST kéréseken keresztül jutnak el a `data-manager` adatbázis szolgáltatáshoz.
4. **Képkocka mentés:** A szolgáltatás elmenti a legfrissebb, észlelekkel és nyomvonalakkal ellátott képkockát a `/frames` megosztott kötetre, hogy a `web-ui` számára elérhető legyen az élő nézethez.

4.3.3. Függőségek

- **Bemenet:** Feldolgozott képkockák a `preprocessing` szolgáltatástól.
- **Kimenet:** Észlelési adatok a `data-manager` szolgáltatás felé.
- **Szolgáltatásfüggőségek:** Közvetlen kapcsolatban áll a `data-manager` szolgáltatással.

4.3.4. Konfiguráció

A szolgáltatás működése az alábbi környezeti változók segítségével testreszabható:

```
1 # A data-manager szolgáltatás URL-je. Alapértelmezett: http://data-manager:8000
2 DATA_MANAGER_URL
3
4 # YOLO detektálás bizalmi küszöbértéke. Alapértelmezett: 0.5
5 CONFIDENCE_THRESHOLD
6
7 # Egy nyomvonal maximális kora (képkockában), mielőtt törlésre kerülne. Alapértelmezett:
8     30
9 MAX_AGE
10
11 # Egy nyomvonal megerősítéséhez szükséges minimális találatok száma. Alapértelmezett: 3
12 MIN_HITS
```

Environment Variables

4.4. Data Manager (data-manager)

4.4.1. Cél

Ebben a részben kerül bemutatásra a `data-manager` szolgáltatás célja. Ez a rendszer központi adatbázisaként működik, mely képes fogadni, eltárolni és aggregálni az összes jármű- és időjárási adatot.

4.4.2. Belső logika

A szolgáltatás működése az alábbi pontokban foglalható össze:

1. **Adatbázis:** A rendszer egy SQLite adatbázist használ, amely a `/db-data` megosztott köteten kerül tárolásra.
2. **API:** REST API-t biztosít az adatok fogadására (`/detections`, `/weather`) és lekérdezésére (`/analytics`, `/historical`).
3. **Adat validáció:** A beérkező adatok érvényességét Pydantic modellekkel ellenőrzi.
4. **Aggregáció:** Egy háttérfolyamat óránként aggregálja a beérkezett járműszám adatokat, ezzel biztosítva a szükséges tanítóadatokat az `online-learner` szolgáltatás számára.

4.4.3. Függőségek

- **Bemenet:** Észlelési adatok a `yolo-detector` szolgáltatástól, valamint időjárási adatok a `weather-service` szolgáltatástól.
- **Kimenet:** Történelmi adatok az `online-learner` és a `web-ui` számára.
- **Szolgáltatásfüggőségek:** Közvetlen szolgáltatásfüggőséggel nem rendelkezik.

4.4.4. Konfiguráció

```
1 # Path to the SQLite database file. Default: /db-data/flowium.db
2 DATABASE_PATH
```

Environment Variables

4.5. Online Learner (`online-learner`)

4.5.1. Cél

Ebben a szakaszban bemutatásra kerül az `online-learner` szolgáltatás fő célja, amely egy online gépi tanulási modell futtatása a jövőbeli forgalmi minták előrejelzésére.

4.5.2. Belső logika

A szolgáltatás működésének főbb lépései a következők:

1. **Modell:** A szolgáltatás központi eleme egy `river.compose.Pipeline` objektum, ami egy `StandardScaler`-ből és egy `LinearRegression` modellből tevődik össze. Maga a teljes tanulási folyamat egy fájlban van tárolva, így az aktuális modellállapot megőrizhető újraindítás után is.
2. **Online tanulás:** Az aktuális forgalmi adatpontokat a `/train` végponton keresztül kapja meg a modell, amely ezeket a `learn_one` metódussal dolgozza fel. Ez lehetővé teszi, hogy a modell valós időben alkalmazkodjon az aktuális forgalmi trendekhez.
3. **Háttér betanítás:** Egy háttérfolyamat a `data-manager`-től származó legfeljebb 7 napos történelmi adatokkal felfrissíti a modellt, ezt nevezük "felmelegítésnek". Ezután a rendszer 10 percenként lekéri az elmúlt 24 óra adatait, hogy a modell mindenkor naprakész legyen.
4. **Előrejelzés:** A szolgáltatás a `/predictions/next-hours` végponton keresztül biztosít előrejelzéseket: minden jövőbeli órára kiszámolja a szükséges jellemzőket (óra, hét napja, időjárási adatok) és a `predict_one` metódussal adja meg a forgalmi előrejelzéseket.

4.5.3. Függőségek

- **Bemenet:** Történelmi forgalmi és időjárási adatok lekérdezése HTTP GET kéréssel a `data-manager` szolgáltatástól.
- **Kimenet:** JSON formátumú forgalmi előrejelzések a `web-ui` számára.
- **Szolgáltatásfüggőségek:** Közvetlenül kapcsolódik a `data-manager` szolgáltatáshoz.

4.5.4. Konfiguráció

Az alábbi környezeti változók használhatók a szolgáltatás beállításához:

```

1 # A data-manager szolgáltatás URL-je. Alapértelmezett: http://data-manager:8000
2 DATA_MANAGER_URL
3
4 # A tanult modell mentésének helye. Alapértelmezett: /models/river_model.pkl
5 MODEL_PATH

```

Environment Variables

4.6. Weather Service (weather-service)

4.6.1. Cél

Ebben a részben bemutatásra kerül a `weather-service` szolgáltatás célja. Ez a komponens adatgazdagító szerepet tölt be a rendszerben, mivel valós idejű időjárási adatokat szerez be egy külső API-ból, majd ezeket továbbítja a `data-manager` részére. Az időjárási adatok egy további jellemzőt képeznek az `online-learner` forgalom-előrejelző modell számára.

4.6.2. Belső logika

A `weather-service` egy könnyű, teljesen aszinkron FastAPI alapú alkalmazás, amelynek működése a következő fő lépésekkel áll:

- Adatforrás:** Az időjárási információkat az ingyenesen, regisztráció és API kulcs nélkül használható Open-Meteo API-ból szerzi be.
- Aszinkron műveletek:** A gyors és hatékony adatlekérés érdekében a szolgáltatás a `httpx` könyvtárat használja, amely nem blokkoló HTTP hívásokat készít.
- Háttérfeladat:** Egy beállított időközönként (alapértelmezésként 10 percenként) induló háttérfolyamat lekéri a kiválasztott földrajzi hely aktuális időjárását, majd POST kéréssel küldi el azt a `data-manager` szolgáltatásnak.

4.6.3. Függőségek

- Bemenet:** Időjárási adatok a külső Open-Meteo API-ból.
- Kimenet:** Időjárási adatok HTTP POST-tal a `data-manager` számára.
- Szolgáltatásfüggőségek:** Közvetlenül kapcsolódik a `data-manager` szolgáltatáshoz.

4.6.4. Konfiguráció

A szolgáltatás az alábbi környezeti változókkal konfigurálható:

```

1 # Ember által olvasható helységnév. Alapértelmezett: Baja,HU
2 LOCATION
3
4 # Szélességi/hosszúsági értékek az időjárási lekérdezéshez.
5 LATITUDE
6 LONGITUDE
7
8 # Időjárási lekérdezések intervalluma másodpercben. Alapértelmezett: 600
9 WEATHER_FETCH_INTERVAL
10
11 # A data-manager szolgáltatás URL-je. Alapértelmezett: http://data-manager:8000
12 DATA_MANAGER_URL

```

Environment Variables

4.7. Web UI (web-ui)

4.7.1. Cél

Ebben a részben bemutatásra kerül a web-ui szolgáltatás célja. Ez a komponens nyújtja a webes felületet a felhasználó számára. A dashboard lehetővé teszi az adatfolyam valós idejű vizualizálását, a historikus adatok metekintését, valamint a gépi tanulási modell által készített előrejelzések áttekintését.

4.7.2. Belső logika

A szolgáltatás alapja egy Streamlit alkalmazás, amely a következő fő elemekből épül fel:

1. **Dashboard szerkezete:** A teljes alkalmazás többoldalas felépítésű. A főnézetben megtalálhatók az élő videók, járműdetektálás valamint külön elemzési lap is rendelkezésre áll, amelyet az `analytics_tab.py` fájl definiál. Itt jelennek meg a korábbi adatok diagramjai és az adatelemzést segítő statisztikák.
2. **Adatlekérés:** Az UI kizárolag az adatfogyasztással foglalkozik, vagyis folyamatosan HTTP kéréseket küld a háttérszolgáltatásoknak, hogy letöltsse a szükséges adatokat a különböző nézetekhez. A képeket például a `yolo-detector`-tól, a statisztikákat a `data-manager`-tól, míg az előrejelzéseket az `online-learner`-tól szerzi be.

4.7.3. Függőségek

- **Bemenet:** Adatokat szinte az összes többi szolgáltatástól, azok REST API-jain keresztül.
- **Kimenet:** Webes felületet biztosít a felhasználó számára.
- **Szolgáltatásfüggőségek:** Közvetlenül kapcsolódik a következő szolgáltatásokhoz: `preprocessing`, `yolo-detector`, `data-manager`, `online-learner`, `weather-service`.

4.7.4. Konfiguráció

A szolgáltatás működéséhez a következő környezeti változók használhatók:

```

1 # Az összes backend szolgáltatás URL-je.
2 DATA_MANAGER_URL
3 YOLO_DETECTOR_URL
4 ONLINE_LEARNER_URL
5 WEATHER_SERVICE_URL
6 PREPROCESSING_URL
7
8 # Az indítandó belépési pont szkript neve. Alapértelmezett: app_simple.py
9 STREAMLIT_APP
10
11 # Kivágási régió koordinátái (a vágás overlay megjelenítéséhez).
12 CROP_X
13 CROP_Y
14 CROP_WIDTH
15 CROP_HEIGHT

```

Environment Variables

5. API és Adatmodellek

Ebben a fejezetben bemutatásra kerülnek azok a Pydantic adatmodellek, amelyek az egyes API-kérések során kerülnek felhasználásra.

5.1. Data Manager

5.1.1. POST /detections

Ez a végpont több észlelési adatot vár egyetlen kérésben. Az elfogadott adatok szerkezetét az alábbi modellek határozzák meg:

```

1 class DetectionCreate(BaseModel):
2     class_id: int
3     class_name: str
4     confidence: float
5     bbox: List[float] # [x1, y1, x2, y2]
6     timestamp: Optional[str] = None
7
8 class DetectionBatch(BaseModel):
9     detections: List[DetectionCreate]

```

DetectionBatch Model

5.1.2. POST /weather

Ez a végpont egyetlen időjárási adatpontot fogad el. Az elvárt adatszerkezet a következő modell szerint alakul:

```

1 class WeatherCreate(BaseModel):
2     temperature: float
3     humidity: float
4     weather_condition: str
5     precipitation: float = 0.0
6     wind_speed: float = 0.0

```

WeatherCreate Model

5.2. Preprocessing

5.2.1. POST /preprocess

Ez a végpont egy konfigurációs objektumot vár, amely egyetlen futtatásra szabja testre az előfeldolgozási lépéseket.

```

1 class ProcessingConfig(BaseModel):
2     crop: bool = True
3     adaptive_enhancement: bool = True
4     denoising_level: int = 5 # 0=off, 1-20
5     sharpness_amount: float = 0.5 # 0.0=off, 0.5=moderate, 1.0=strong
6     background_subtraction: bool = False

```

ProcessingConfig Model

5.3. Online Learner

5.3.1. POST /train

A tanítási adatokat ezen a végponton keresztül lehet elküldeni egyesével a modell betanításához. Az elvárt szerkezet az alábbi:

```

1 class TrainingData(BaseModel):
2     hour: int
3     day_of_week: int
4     temperature: Optional[float] = 20.0
5     humidity: Optional[float] = 50.0
6     precipitation: Optional[float] = 0.0
7     vehicle_count: int

```

TrainingData Model

5.3.2. POST /predict

Ez a végpont egy előrejelzéshez szükséges jellemzőkészletet vár. Az adatszerkezet az alábbi modell alapján épül fel:

```

1 class PredictionRequest(BaseModel):
2     hour: int
3     day_of_week: int
4     temperature: Optional[float] = 20.0
5     humidity: Optional[float] = 50.0

```

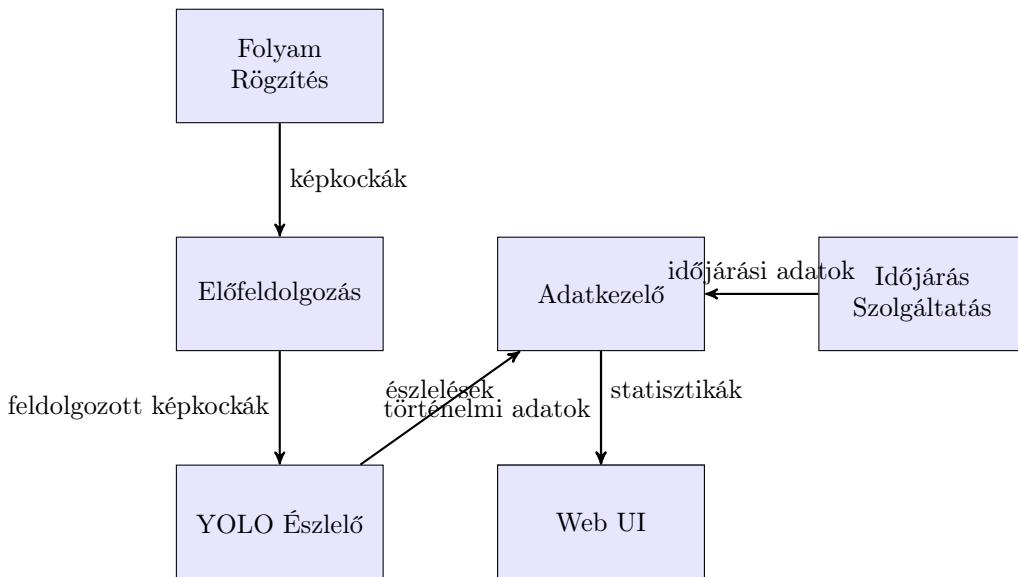
```
6 precipitation: Optional[float] = 0.0
PredictionRequest Model
```

6. Vizualizációk

Ebben a fejezetben bemutatásra kerülnek a rendszer különböző vizuális ábrázolásai, amelyek segítik az architektúra és a működés jobb megértését. A következőkben szemléltető példákat, valamint architektúra-diagram található.

6.1. Rendszerarchitektúra Diagram

Az alábbi ábra szemlélteti a Flowium projekt mikro-szolgáltatásos architektúráját. A diagramon jól láthatók a fő komponensek és az adatfolyamok iránya a rendszerben.



5. ábra. A Flowium rendszer mikro-szolgáltatásos architektúrája

7. Konfiguráció

Ebben a szakaszban ismertetésre kerül a rendszer konfigurációja. A Flowium rendszer beállításai főként környezeti változókkal történnek, amelyekhez egy `.env.example` nevű sablonfájl ad segítséget. Az alábbiakban felsorolásra kerülnek a legfontosabb változók:

- `YOUTUBE_URL`: A megfigyelt YouTube élő közvetítés URL-je.
- `LOCATION`, `LATITUDE`, `LONGITUDE`: Az időjárási szolgáltatás helyadatai.
- `CROP_X`, `CROP_Y`, `CROP_WIDTH`, `CROP_HEIGHT`: A videó kivágási régiójára vonatkozó koordináták.

8. Telepítés és Üzemeltetés

8.1. Helyi Telepítés

Ebben a részben bemutatásra kerül a rendszer helyi gépen történő telepítésének folyamata.

8.1.1. Előfeltételek

A rendszer futtatásához az alábbi szoftverekre van szükség:

- Docker Engine
- Docker Compose

8.1.2. Telepítési Lépések

A telepítés az alábbi lépésekben történik:

- 1. Forrás fájlok kicsomagolása:** A projekt forráskódjának kibontása a zip fájlból, ami a docker fájlokat és a python kódokat tartalmazza.
- 2. Környezeti változók beállítása:** A `.env.example` fájl másolása és szerkesztése szükséges.

```
1 cp .env.example .env
2 nano .env
```

- 3. Szolgáltatások építése és indítása:** A Docker Compose segítségével az összes konténer felépítése és indítása megtörténik.

```
1 docker-compose up -d --build
2
```

- 4. Logok ellenőrzése:** A szolgáltatások működésének figyelése a következő parancssal történhet:

```
1 docker-compose logs -f
2
```

- 5. Web felület elérése:** A böngészőben a <http://localhost:8501> címen érhető el a Streamlit dashboard.

9. Továbbfejlesztési Lehetőségek és Kiegészítések

9.1. YOLO Modell Optimalizáció

Ebben a részben bemutatásra kerül a YOLOv8 modell ONNX formátumra történő konvertálásának előnyei.

9.1.1. ONNX Export

Az ONNX (Open Neural Network Exchange) formátum használata jelentős teljesítménynövekedést eredményez CPU-alapú rendszereken. A projekt tartalmaz egy `export_model.py` szkriptet, amely elvégzi a konverziót:

```
1 python services/yolo-detector/export_model.py
Modell konverzió
```

9.1.2. Teljesítmény Összehasonlítás

Formátum	Fájlméret	Inferencia idő (CPU)
PyTorch (.pt)	6.2 MB	~200 ms
ONNX (.onnx)	12.2 MB	~80 ms

2. táblázat. YOLO Modell Teljesítmény Összehasonlítás

A táblázatból látható, hogy bár az ONNX fájl mérete közel kétszerese a PyTorch modellnek, a CPU-alapú inferencia sebessége több mint kétszeresére növekszik.

9.2. Web UI Kiegészítések

Ebben a részben bemutatásra kerülnek a webes felület funkciói.

9.2.1. Analytics Tab - Legutóbbi Detekciók

Az Analytics lapon megjelenik egy új szekció, amely valós időben jeleníti meg az utolsó 10 detekciót. A megjelenítés táblázatos formában történik, amely a következő oszlopokat tartalmazza:

- Időpont (óra:perc:másodperc)
- Dátum (év-hónap-nap)
- Jármű típusa
- Konfidencia érték
- Detekció azonosító

9.2.2. Predictions Tab

Ebben a részben bemutatásra kerül a rendszer negyedik lapja, amely a gépi tanulással kapcsolatos funkciókat tartalmazza.

Modell Teljesítmény Mutatók: A lap tetején négy oszlopban jelennek meg a fő metrikák:

- **Tanított minták száma:** Az eddigi betanítások összesített mennyisége
- **Mean Absolute Error (MAE):** A modell átlagos abszolút hibája
- **Modell státusz:** Ready (kész) vagy Training (tanulás alatt)
- **Átlagos hiba:** Járművek számában kifejezve

24 Órás Előrejelzés: Egy interaktív Plotly diagram jeleníti meg a következő 24 óra várható forgalmát. A diagram x tengelyén az idő, y tengelyén pedig a becsült járműszám látható.

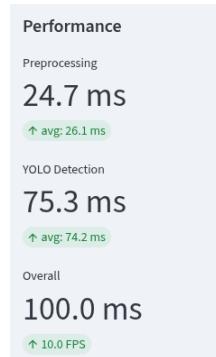
Tanítási Történet: Az utolsó 20 tanítási minta megjelenítése történik egy kombinált line chart-on, amely összeveti az előrejelzett és a valós értékeket. Ez lehetővé teszi a modell pontosságának vizuális ellenőrzését.

Hiba Eloszlás: Egy hisztogram ábra mutatja a predikciós hibák eloszlását, amely segít megérteni, hogy a modell mennyire konziszensen jósol.

10. Képek a Web-es felületről



6. ábra. Kép a folyamatról: bejövő videó, előfeldolgozás (vágás, fényerő, gamma, kontraszt), detektált kép



7. ábra. Feldolgozási sebesség

Preview with Selected Crop Region (Red Rectangle)



8. ábra. Teljes kép releváns része, kivágott terület



9. ábra. Feldolgozott kép