



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatikai Kar

Algoritmusok és Alkalmazásaik Tanszék

# Közúti forgalomelemzés neurális hálózatokkal

Hajder Levente

docens

Fenyvesi Attila

programtervező informatikus BSc

Budapest, 2022

# Tartalomjegyzék

Bevezetés .....	3
Képkockáktól a járművekig .....	3
You Only <del>Live</del> Look Once (YOLO) .....	7
A modell szerkezete .....	8
A magyar módszer .....	22
Kálmán-szűrő .....	25
Az algoritmus működése .....	27
Paraméterek beállítása .....	34
Rauch–Tung–Striebel simító algoritmus .....	42
Felhasználói dokumentáció .....	44
Áttekintés .....	44
A kezdőképernyő felépítése .....	44
Program használatának lépései .....	45
Videófájl betöltése .....	45
Cache betöltése .....	46
Forgalomszámláló kapuk megadása .....	47
Kamera kalibrálása .....	48
Videó elemzése .....	49
Statisztikák megjelenítése .....	51
Fejlesztői dokumentáció .....	53
Feladat .....	53
A program felépítése .....	53
Osztályok leírása .....	54
MediaPlayer .....	56

VideoFilter .....	57
VideoFilterRunnable.....	58
CameraCalibration.....	59
Tracker.....	61
TrackerWorker .....	63
VehicleModel .....	64
GateModel.....	66
StatModel.....	68
Detection .....	70
Vehicle .....	71
Tesztek.....	73
További fejlesztési lehetőségek .....	77
Zárószó és köszönetnyilvánítás .....	78
Irodalomjegyzék .....	79

# Bevezetés

A közúti forgalomban résztvevő gépjárművek számlálásának és követésének kiemelten fontos szerepe van a forgalomtechnikában. Az idők folyamán erre számtalan módszert dolgoztak ki, kezdve a „kézi” forgalomszámlálástól, a különféle útburkolatba épített mágneses- vagy nyomásérzékeny szenzorokon át a rádiós- vagy optikai elven működő jeladó/jelfogó kapukig. Az előbbi a mai napig az egyik legpontosabb módszernek tekinthető és jól skálázható, viszont rendkívül nagy az emberi erőforrás igénye, ami miatt hosszú távon rendkívül költségessé válik. A különféle szenzorokkal történő gépjármű detektálásnak a hátránya pedig a monolitikus felépítéséből adódik: nehezen skálázható, hiszen fixen telepített, sokszor az útburkolatba épített, kizárólag erre a célra tervezett érzékelőkkel történik a járművek érzékelése.

A gépi látás ezen problémákra tud egy átfogó megoldást nyújtani. Nincs szükség speciális hardverre, csupán a vizsgálni kívánt csomópontokba telepített kamerákra van szükség, de akár a már meglévő térfigyelő kamerahálózat is bevonható. Alapos tervezéssel pedig rendkívül jó pontosság érhető el.

A szakdolgozatom célja egy olyan közúti forgalom elemzésére alkalmas szoftvernek az elkészítése, amely nemcsak a járművek követésére és számlálásra képes, hanem a mesterséges intelligencia eszköztárát is felhasználva egyéb információkat (pl. típusuk) is képes szolgáltatni a forgalomban résztvevő gépjárművekről.

## Képkockáktól a járművekig

Ahhoz, hogy a videófolyamból értékes információkat nyerhessünk, először is fel kell ismerni a gépjárműveket a képen, majd ezeket a járműveket képkockáról-képkockára követni kell. Érdemes megjegyezni, hogy minket csak is a gépjárművek érdekelnek, így az egyéb mozgó objektumokat (pl. gyalogosokat) ki kell szűrni.

Mozgóképeken az objektumok követésére számtalan algoritmus létezik [1]:

1. Optikai áramlás:
  - a. **Dense Optical Flow**: egy videófolyam minden képpontjára meghatározza annak az elmozdulásvektorát.
  - b. **Sparse Optical Flow**: néhány, ún. *feature point* elmozdulását követi a videófolyamban.
2. Egy meghatározott objektum követésére szolgáló algoritmusok. Ezeket az algoritmusokat inicializálni kell, azaz a legelső képkockán meg kell határozni egy téglalap alakú területet, amely a követendő objektumot tartalmazza. Ilyen algoritmusok pl.: **BOOSTING, MIL, KCF, TLD, MEDIANFLOW, GOTURN, MOSSE** és a **CSRT**.
3. Több objektum együttes követése: Amennyiben minden egyes képkockára meg tudjuk határozni az objektumok helyzetét (pl. egy *object detector*tal), akkor az ugyanahhoz az objektumhoz tartozó pozíciókat összeköthetjük egy folytonos útvonallá.

Mivel szeretném a követett gépjárműveket kategóriákba sorolni, így mindenképpen alkalmaznom kell egy CNN<sup>1</sup> alapú *object detector*ot, így kézenfekvő a 3., azaz a „több objektum együttes követése” módszernek az alkalmazása. A módszer a következő lépésekből áll:

1. **Detektálás**: Egy betanított *object detector*tal minden egyes képkockán meghatározom a gépjárművek helyzetét.
2. **Találatok szűrése**: A detektor találatai közül kiszűröm az alacsony valószínűségi pontszámmal rendelkezőket és a duplikátumokat.
3. **IOU<sup>2</sup>-alapú követés**: Minden *bounding box*hoz<sup>3</sup> meghatározom a követett járművek utolsó detektációjához viszonyított IOU értékét, majd a *magyar módszerrel*<sup>4</sup> megkeresem az összetartozó párokat.

---

<sup>1</sup> **CNN (Convolutional Neural Network)**: A mély neuronhálók egy speciális, a képanalitikában használt fajtája. Filtereket képesek megtanulni, és rendkívül jól teljesítenek osztályozási problémák megoldásánál képek esetében.

<sup>2</sup> **IOU (Intersection Over Union)**: Két téglalap metszetének és uniójának az arányát kifejező szám. 0 és 1 közötti értéket vehet fel.

<sup>3</sup> **bounding box**: A képeken a detektált objektumokat körülhatároló téglalap.

Egy *object detector* sem képes 100%-os pontossággal detektálni a betanított objektumokat és lehetnek olyan képkockák, amelyeken egy-egy járművet nem ismer fel. Ezek a vakfoltok azt eredményezik, hogy a járművek követése az utoljára detektált képkockán véget ér, majd miután újra detektáljuk ugyanazt a járművet, az új objektumként kerül felismerésre. Ennek kivédésére két módszer létezik:

1. Egy Kálmán-szűrővel megpróbáljuk megbecsülni a jármű helyzetét a hiányzó képkockákon. [2] Ennek a módszernek az előnye, hogy gyors és teljes takarásban lévő objektumok esetén is (a helyzettől függően) viszonylag nagy pontossággal képes megbecsülni a követett objektum helyzetét.
2. Egy másik módszer, hogy a hiányzó képkockákon egy optikai elven működő *tracker* (pl. KCF) használunk. [3] Hátránya, hogy pl. pontatlan detektor esetén a sok fals negatív eredmény miatt sokszor kell inicializálni a *tracker*-t, ami jelentősen lelassíthatja az algoritmust, illetve jelentős kitakarás esetén nem képesek tovább követni az objektumot. Viszont sokkal pontosabb és képes reagálni a követett objektum sebességének és irányának a változására is.

Én az utóbbi megoldást választottam, mert a forgalomban résztvevő járművek sebessége (és sokszor az iránya is) szinte folyamatosan változik, így ez egy kardinális szempont az értékelhető eredmény tekintetében, illetve a járművek teljes kitakarására is ritkán kerül sor, mivel a forgalomfigyelő kamerák jellemzően fentről rögzítik a forgalmat.

A járművek követése akkor ér véget, ha az *object detector* az optikai *tracker* egy  $N$  számú követése után sem ad újabb jelzést, vagy ha a *tracker* hibával leáll (a jármű kitakarásra kerül vagy kimegy a képből).

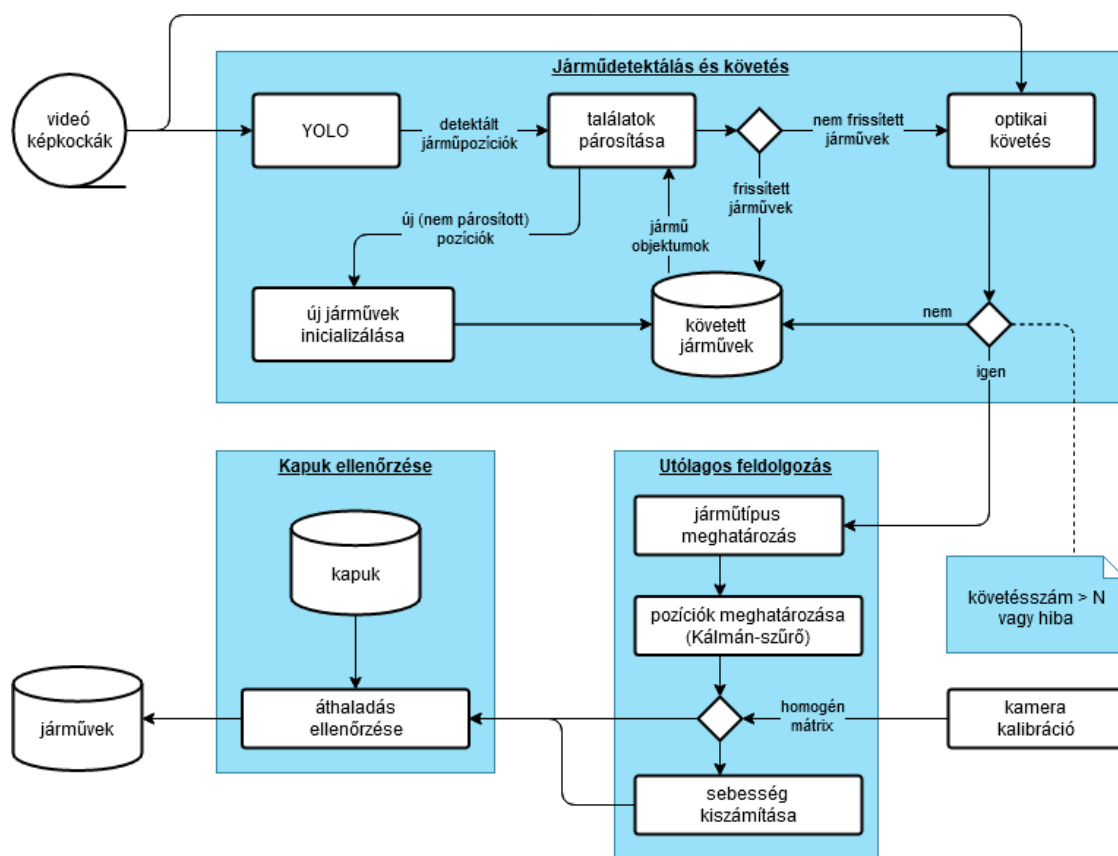
A rögzített adatok feldolgozását egy többlépcsős *pipeline*<sup>5</sup> végzi, melynek bizonyos lépései a gyorsabb végrehajtás érdekében párhuzamosítva vannak, azaz néhány lépést egy különálló szál végez. Miután egy jármű követése véget ért, a járműobjektum továbbadásra kerül utólagos feldolgozásra. Az utólagos feldolgozás során meghatározásra kerül a jármű típusa

---

<sup>4</sup> **magyar módszer:** Egy algoritmus, amely segítségével páros gráfokban lehet maximális elemszámú párosítást keresni polinom időben. Harold Kuhn dolgozta ki az eljárást König Dénes és Egerváry Jenő munkája nyomán. Tiszteletükre magyar módszernek nevezte el.

<sup>5</sup> **pipeline (csővezeték):** Adatok több lépésben történő feldolgozásának a láncolata. Minden egyes lépés kimenete egyben a következő lépés bemenete is. A lépések között pufferek biztosítják az eltérő végrehajtási sebességekből adódó torlódások feloldását.

(személyautó, teherautó, busz, motorkerékpár vagy kerékpár), a végleges pozíciója és – amennyiben a kamera kalibrálása megtörtént, vagyis a homogén leképezési mátrix ki lett számítva – a sebessége is. Mivel a detektor által szolgáltatott járműpozíciók elég zajosnak tekinthetők, így azok a véglegesítés során még egy Kálmán-szűrőn átmennek, ezáltal egy zajmentes, egyenletes pályagörbét kapunk minden járműhöz. Ennek főként a sebességek meghatározásánál van fontos szerepe. Legutolsó lépésben pedig az előzetesen megadott virtuális forgalomszámláló kapukon való áthaladás ellenőrzése történik.



1. ábra - Járművek detektálásának és feldolgozásának a folyamata

Azt azért érdemes megjegyezni, hogy habár az adatok feldolgozása párhuzamosítva van, a legszámításigényesebb művelet mégis maga az objektumdetektor, így jelentős sebességnövekedésre a párhuzamosítás miatt nem számíthatunk.

## You Only ~~Live~~ Look Once (YOLO)

Az objektum detektálás az informatikán belül a gépi látás témakörébe tartozó technológia, amelynek a célja meghatározott objektumok felismerése képeken és azok kategóriákba sorolása (*image classification*), és adott esetben azok képen belüli helyzetének a meghatározása (*object detection*). Az objektum felismerő algoritmusok jellemzően a gépi tanulás, azon belül is legfőképpen mélytanulás (*deep learning*) eszköztárát használják. Az elmúlt évtizedben a mesterséges intelligencián belül a mély neurális hálók (DNN) rohamos fejlődése számtalan objektum felismerő algoritmus kidolgozását tette lehetővé:

- Region Proposals (R-CNN [4], Fast R-CNN [5], Faster R-CNN [6], Libra R-CNN [7])
- Single Shot MultiBox Detector (SSD) [8]
- You Only Look Once (YOLO) [9] [10] [11] [12]
- Single-Shot Refinement Neural Network for Object Detection (RefineDet) [13]
- Retina-Net [14]

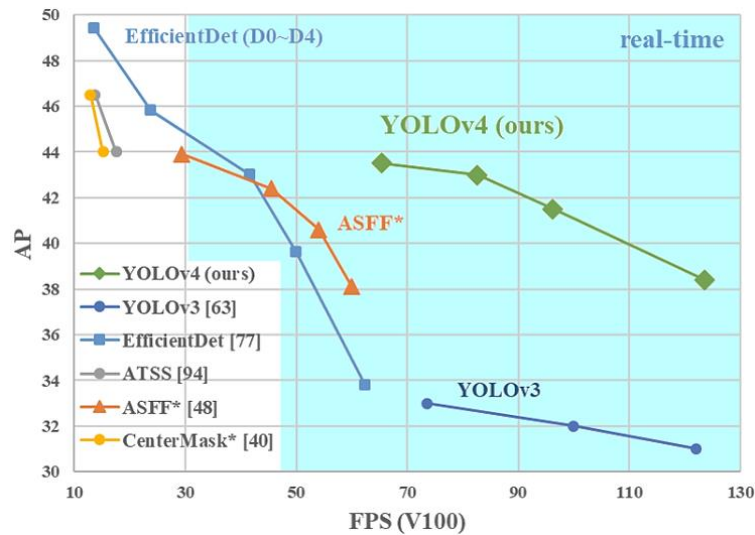
Ezek közül mérföldkőnek számít a 2016-ban bemutatott YOLO modell. Míg a legtöbb *object detector* ún. *two-stage* megközelítést alkalmaz (pl. a különféle R-CNN variánsok), addig a YOLO egy *one-stage* architektúra, melynek köszönhetően az *MS-COCO*<sup>6</sup> adatkészlet 43,5%-os AP<sup>7</sup> (65,7% AP<sub>50</sub>) érték mellett 65 FPS sebességgel képes objektumok detektálására egy Tesla V100 architektúrájú GPU-n.

---

<sup>6</sup> **MS-COCO (Microsoft Common Objects in Context):** Objektumfelismerő algoritmusok tanítására és összehasonlítására szolgáló adathalmaz. Mára *de facto* benchmark eszköznek tekinthető a gépi látás világában. Bővebben: [25]

<sup>7</sup> **AP (Average Precision):** Az objektum felismerő algoritmusok pontosságának mérésére szolgáló érték. Az értéket úgy kapjuk, hogy a találatok arányát elosztjuk az adathalmazban szereplő összes objektumnak és a megtalált objektumok számának az arányával. Találatnak számít, ha a detektor által jelzett *bounding box* IOU értéke egy meghatározott érték felett van (jellemzően 0.5) a tanításhoz használt (az adathalmazban rögzített) *bounding box*okhoz képest.





2. ábra - YOLOv4 teljesítménye az MS-COCO adathalmazon, más object detectorokhoz viszonyítva. Forrás: [12]

A YOLO modell rohamos fejlődését mi sem bizonyítja jobban, hogy szakdolgozatomat legelőször a YOLOv3-mal kezdtem el megírni, majd ezt lecseréltem az idő közben bemutatott YOLOv4-re, de ezen sorok írásakor már publikálták a YOLOv5-öt is.

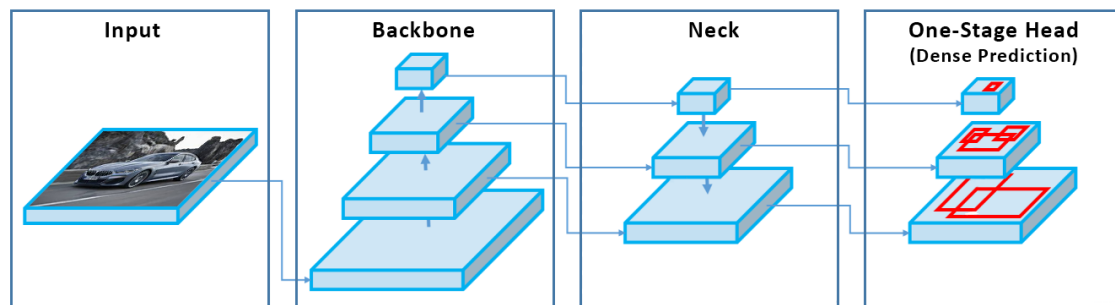
## A modell szerkezete

A modern *object detector*ok általában két részből állnak: **backbone** és **head**. A *backbone*-t alkotó konvolúciós hálózatok már „elő vannak tanítva” valamilyen *image classification* adathalmazon (pl. ImageNet), majd a tényleges tanítás során finom hangolva vannak az adott *image detection* adatokon. Lényegében ez a rész felelős az objektumok „jellegzetességeinek a felismeréséért” (*feature map extraction*).

A *head* feladata az objektumok kategóriájának (*object class prediction*) és a képen belüli helyzetének (*bounding box prediction*) a megadása. Ezeket két kategóriába lehet sorolni: *one-stage (dense prediction)* és *two-stage (sparse prediction) detector*. A *two-stage detector*ok pontosabbak, viszont rendkívül számításigényesek és emiatt lassúak. Ezekkel szemben a *one-stage detector*okkal szignifikáns gyorsulás érhető el, viszonylag csekély pontatlanságnövekedés árán. A YOLO architektúra – ahogy az már fentebb említésre került – a *one-stage* megközelítést alkalmazza.

Az elmúlt években kifejlesztett detektoroknál a *backbone* és a *head* közé további köztes rétegeket is beépítettek, amelyek a **neck** nevet kapták. Ezen rétegek feladata a *backbone*

különböző rétegeiből különböző „felbontású” *feature map*ek leválasztása, amely nagyban segíti az eltérő méretű objektumok pontosabb detektálását.



3. ábra - A YOLOv4 felépítése. Forrás: [12] (szerkesztett)

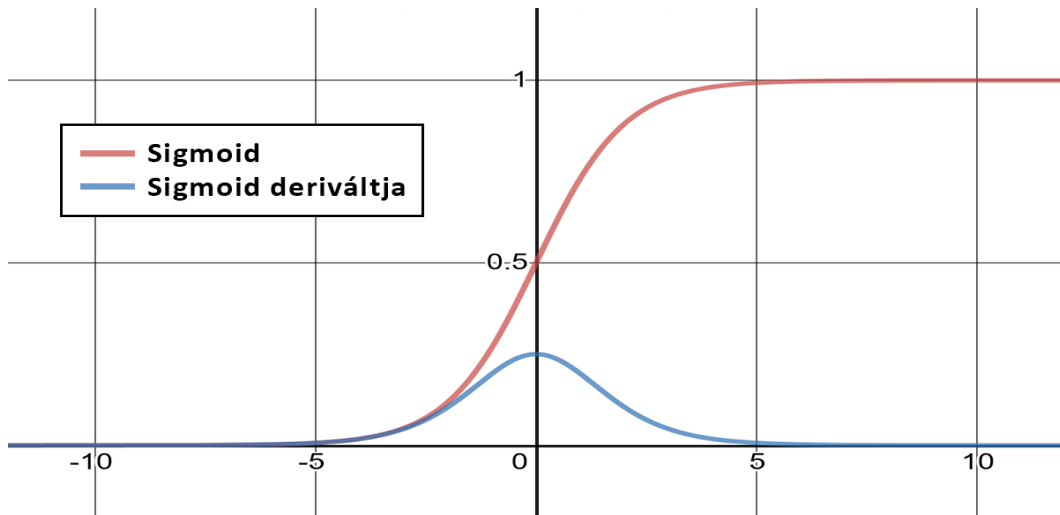
Összefoglalva a YOLOv4 felépítése a következő:

- Input réteg
- Backbone: CSPDarknet53 [15]
- Neck: SPP [16], PAN [17]
- Head: YOLOv3 [11]

## Backbone

Ha növelni szeretnénk a neurális hálózatunk által felismerhető minták számát (vagy pontosságát), akkor kézenfekvő megoldás a háló mélységének a növelése. A modell komplexitásának növelése viszont számtalan problémát eredményezhet:

- **Overfitting:** A feladat- és a modell komplexitása kéz a kézben jár. Ha túl nagy a modellünk, akkor a tanító adatokat tökéletesen (nagyon kis hibával) megtanulja, viszont a tesztadatokon (és a gyakorlati, éles adatokon) nagyon nagy hibát produkál.
- **Vanishing gradients:** Bizonyos aktivációs függvényeknek (mint pl. a népszerű sigmoidnak) a parciális deriváltja az értelmezési tartományuk nagy részén tart a nullához.



A gradiens alapú tanító algoritmusok a háló súlyait az aktivációs függvények deriváltjával arányosan változtatják. Ez kevés réteget (kevés aktivációs függvényt) tartalmazó hálók esetén még nem okoz gondot, de mély neurális hálóknál a gradiens értéke olyan kicsire csökkenhet, hogy a tanítás folyamata nagyon lelassulhat, akár meg is akadhat.

Ezekre a problémákra ad megoldást a **DenseNet** [18] struktúra.

### Dense Block és DenseNet

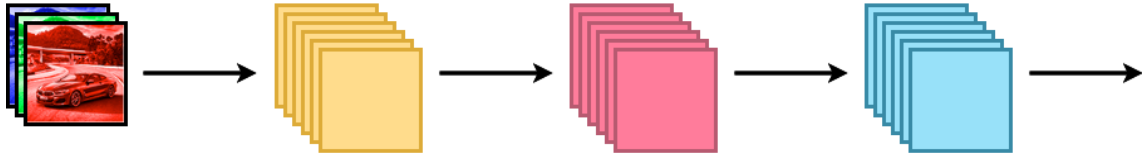
Képzeljük el, hogy egy neurális háló minden rétegét egy  $H_i$  nemlineáris transzformáció ír le. Ez a  $H_i$  transzformáció megadható akár egy összetett függvénnyel is, amely tartalmazhat aktivációs függvényt, Batch Normalization<sup>8</sup>, Pooling<sup>9</sup> és konvolúciós lépéseket is.

Egy hagyományos konvolúciós háló a következőképpen írható fel:

$$x_i = H(x_{i-1}) \quad (1)$$

<sup>8</sup> **Batch Normalization:** A mesterséges neurális hálók stabilitását szolgáló lépés, amely normalizálja az előző réteg aktivációs függvényének kimenetét azáltal, hogy kivonja az elemekből az átlagukat, majd elosztja a szórásukkal. Bővebben: [26]

<sup>9</sup> **Pooling:** A konvolúciós rétegek a megtanult mintázatokat csak egy meghatározott helyen képesek felismerni. A *pooling* rétegek közbeiktatásával a konvolúciós rétegek által kiszámolt *feature map*ek robusztusságát lehet növelni azáltal, hogy csökkentjük a méretüket átlagolással (Average Pooling) vagy maximum kiválasztással (Max Pooling). Bővebben: [27]

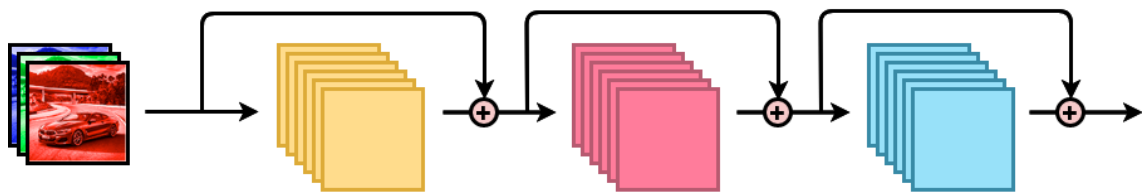


4. ábra - Hagományos CNN felépítése

A ResNet beiktat egy párhuzamos „skip-connection” lépést is, ami kihagyja a  $H_i$  transzformációt, és végül hozzáadja az előző réteg eredményét a  $H_i$  eredményéhez:

$$x_i = H(x_{i-1}) + x_{i-1} \quad (2)$$

Ennek következtében a gradiens szabadon áramolhat vissza a legelső rétegek felé a tanítás során, és nem akad el még mély hálózatok esetén sem.

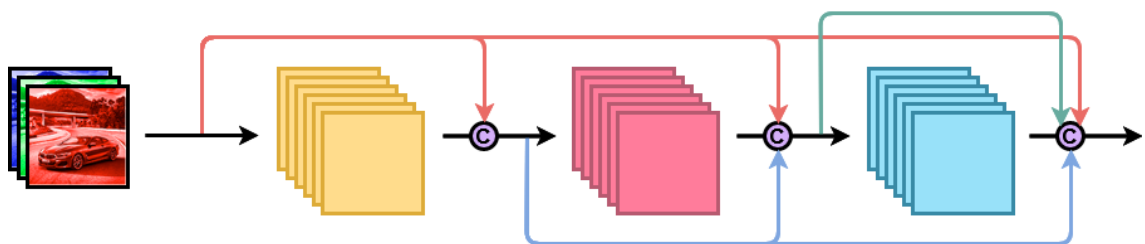


5. ábra – ResNet felépítése

A DenseNet két további változtatással segíti elő a még jobb rétegek közti információ áramlást: az  $i$ -edik réteg nem csak az  $(i - 1)$ -edik, hanem az összes előző réteg kimenetét megkapja, illetve ezeket a *feature map*eket nem összeadja, hanem konkatenálja:

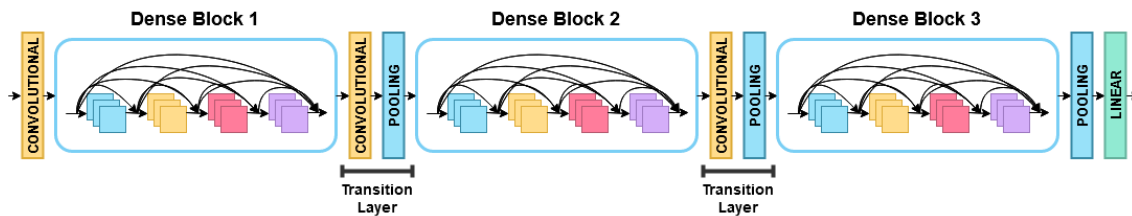
$$x_i = H([x_1, \dots, x_{i-1}]) \quad (3)$$

A konkatenáció miatt az egyes  $H_i$  transzformációk során létrejövő *feature map*eknek azonos méretűeknek kell lennie.



6. ábra - Dense block felépítése

Egy ilyen transzformációs sorozatot nevezünk *Dense block*-nak. Két egymást követő *Dense block*-ot az úgynevezett *transition layer* köt össze. Ez a réteg felelős a *feature map*ek felbontásának csökkentéséért (*downsampling*) is.



7. ábra - DenseNet szerkezete

A legutolsó *Dense block*-ot egy *global average pooling* réteg követ, amihez egy *linear softmax classifier* van kapcsolva. A *Dense block*ok sorozatának *transition layer*ekkel történő összekapcsolását nevezzük DenseNet struktúrának.

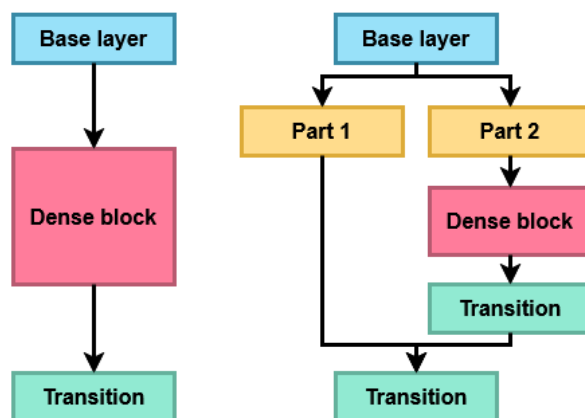
### Cross-Stage-Partial-connections (CSP)

A mesterséges neurális hálózatok méretének növelésével egy másik problémával is szembe kellett néznie a kutatóknak, mégpedig az egyre nagyobb számítási kapacitás (és memória) igényével. Habár léteznek *real-time object detector*ok, ezek valós idejű futása csak egy rendkívül drága GPU-n (pl. az NVidia Tesla V100-as architektúrája) teljesül, a CPU-ra vagy mobil GPU-ra optimalizált változataiknál pedig a pontatlanság számottevő növekedése tapasztalható, egyéb célhardvereken (*ASIC - Application-Specific Integrated Circuit*) történő valós idejű alkalmazásuk pedig gyakorlatilag elképzelhetetlen.

A **Cross Stage Partial Network (CSPNet)** [15] felépítés a következő problémák orvoslása során lehet a segítségünkre:

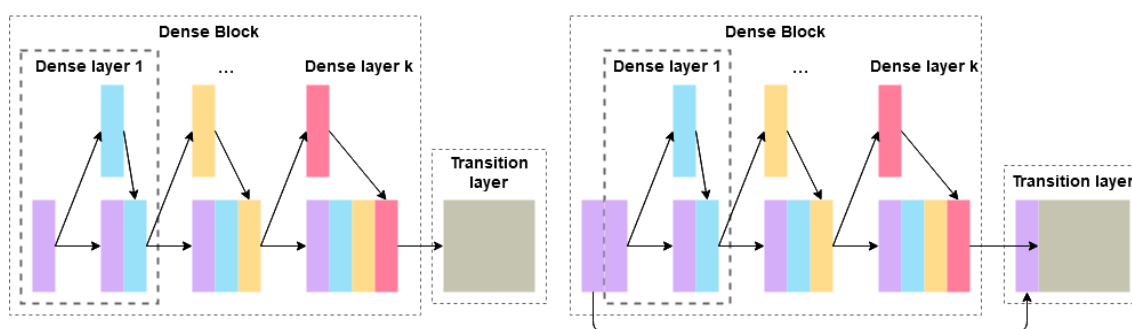
- A CNN tanulási képességeinek javítása
- Szűk számítási keresztmetszet feloldása
- Memória szükséglet csökkentése

A módszer abból áll, hogy egy *Dense block* bemenetétől szolgáló *feature map*et szétválaszt két részre. Az első fele – kikerülve a *Dense block*-ot – hozzá lesz konkaténálva a következő bemenetéhez, és csak a másik fele halad át a *Dense block*-on.



8. ábra - DenseNet (balra) és CSPDenseNet (jobbra). Forrás: [15] (szerkesztett)

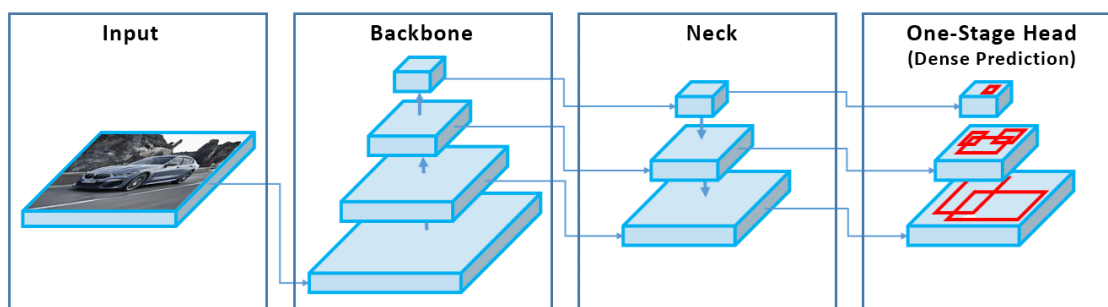
A *CSPNet* felépítést számtalan backend esetén alkalmazni lehet. A YOLOv4 esetén a YOLOv3-ban is használt DarkNet53-mal kombinált változatot alkalmazták, ami a *CSPDarkNet53* nevet kapta.



9. ábra - DenseNet (balra) és CSPDenseNet (jobbra). Forrás: [15] (szerkesztett)

## Neck

Ahogy az már említésre került, az objektum detektorok egyrészt állnak egy *backbone* részből, amely a betanított objektumok „jellegzetességeit” ismeri fel a képen (*feature extraction*), és áll egy *head* részből, amely konkrétan az objektumok detektálását végzi. Ahhoz, hogy a képen a különböző méretű objektumokat is megfelelő pontossággal fel tudjuk ismerni, egy köztes, hierarchikus struktúra is beépítésre került, amely eltérő felbontású *feature map*eket szolgáltat a *head* számára.

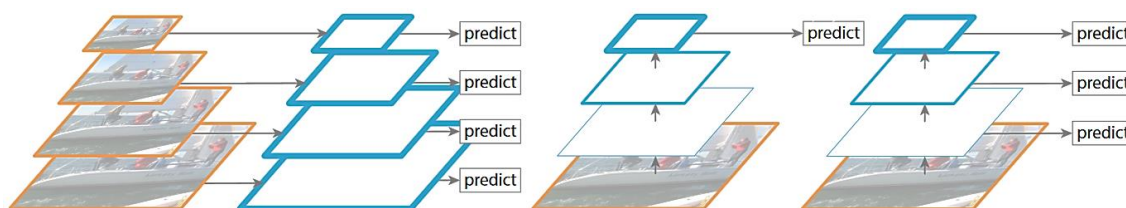


10. ábra - A YOLOv4 felépítése. Forrás: [12] (szerkesztett)

Ahhoz, hogy a *head* számára átadott adat információtartalmát növeljük, a *backbone* meghatározott rétegeiből érkező *feature maps*et hozzáadjuk (vagy konkatenáljuk) a *neck* szomszédos rétegével. Mivel a *backbone* és a *neck* rétegeiben az információ áramlása ellentétes irányú (lásd 10. ábra), így a *head* bemenete tartalmazni fog térbeli információ tartalmat a *backbone* lentől-felfelé tartó áramából, és szemantikai információ tartalmat is a *neck* fentről-lefelé tartó áramából.

### Feature Pyramid Networks (FPN)

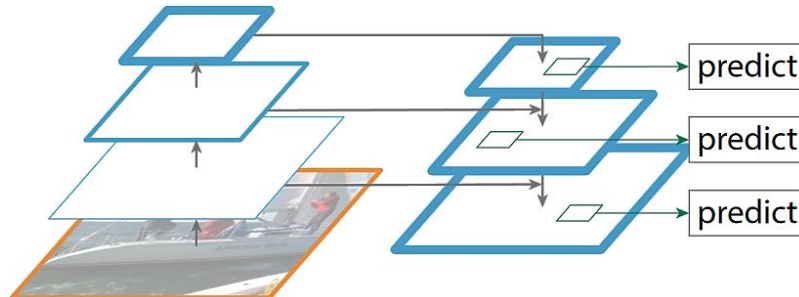
A *feature map* piramisok (*feature pyramids*) alapvető komponenseknek számítanak egy objektumfelismerő rendszerben, amennyiben különböző méretű objektumokat is fel akarunk ismerni. Viszont rendkívül memóriaigényes megoldásnak számítanak, ezért sok objektumfelismerő rendszerben nem alkalmazzák őket.



11. ábra - Különféle feature piramis megoldások. Forrás: [19]

A fenti ábrán a bal oldali megoldás értelmében ugyanazt a képet többször különböző felbontásokon átengedjük ugyanazon a hálón. Ez nyilvánvalóan rendkívül lassú. Éppen ezért sok rendszer a sebesség érdekében csak egy adott felbontású *feature map*et használ a detektáláshoz (középső ábra). Alternatív megoldásnak számít a *feature maps*ek „újrahasznosítása”, így több különböző felbontású rétegből kapunk információt a detektáláshoz (bal oldali ábra).

Az **FPN (Feature Pyramid Networks)** [19] néven bemutatott módszer majdnem olyan gyors, mint az előbbi, „újrahasznosított” megoldás, viszont sokkal pontosabb detektálást tesz lehetővé.



12. ábra - Az FPN (Feature Pyramid Networks). Forrás: [19]

A YOLOv3 (nem elírás, a YOLOv4 is a v3 head-jét használja, kisebb módosításokkal) az FPN-hez hasonló megközelítést alkalmaz abban a tekintetben, hogy az objektumok detektálást különböző felbontású rétegeken külön-külön elvégzi. Vagyis a „fentről-lefelé áramló” *feature map*eket (jobb oldali piramis) minden lépésben kétszeresére felskálázza, majd a „lentől-felfelé áramló” szomszédos (ugyanakkora méretű) *feature map*et hozzáadja. Az így kapott eredményt még egy  $(3 \times 3)$ -as konvolúciós szűrőn átengedi, hogy a felskálázáskor létrejövő torzulásokat valamelyest csillapítsa. Aztán az így kapott *feature map*et adja át a *head*nek.

### **SPP (Spatial Pyramid Pooling)**

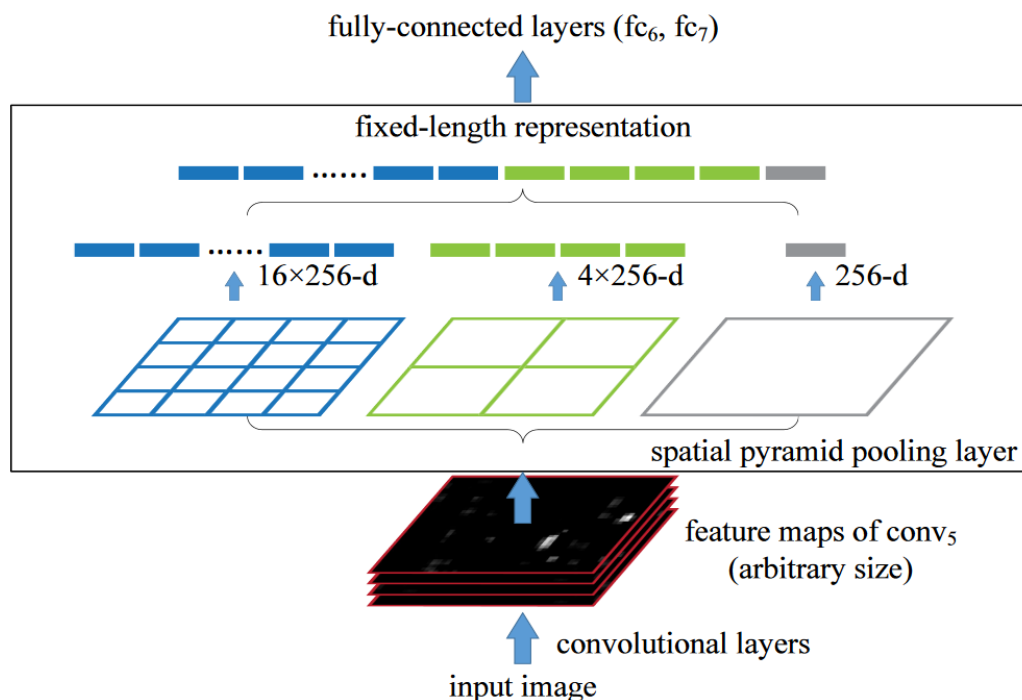
Az objektumfelismerő rendszereknél egy alapvető technikai hibával is találkozunk, mégpedig azzal, hogy bemenetként egy fixen meghatározott felbontású (általában  $m \times m$ , azaz négyzet alakú) képet várnak. Ez nagyban korlátozza a bemeneti képek képarányát. A bemenetként átadott képeket általában átméretezik a kívánt felbontásra, ez viszont eltérő képarányú (álló vagy fekvő) képek esetén olyan mértékű torzuláshoz vezethet, hogy a detektor nem lesz képes felismerni a képen található objektumot. A képek széleinek levágásával (*cropping*) pedig értékes információt veszthetünk.

A CNN hálózatok alapjában véve két részből állnak: konvolúciós rétegekből, majd az ezeket követő FC (*Fully Connected layer*, azaz elemi perceptronok sorozata) rétegekből. A konvolúciós rétegeknek bemenetként nincs szüksége fix méretű képekre, gyakorlatilag akármekkora felbontású *feature map*eket képesek generálni. Az FC rétegek viszont definíciójukból adódóan fix méretűek. Vagyis fix méret megkötése a hálózaton belül



mélyebben fekvő FC rétegek miatt van. A legutolsó konvolúciós réteg után beépített **Spatial Pyramid Pooling (SPP)** [16] réteg ezt a fix méretre vonatkozó megkötést hivatott feloldani.

Ahhoz, hogy ezt a méretbeli megkötést feloldjuk, az utolsó konvolúciós réteg után következő *pooling* réteget kicseréljük egy *spatial pyramid pooling* rétegre (lásd: 13. ábra). Ebben a rétegben az összes filter által kiszámolt *feature map*et egyesével több ( $m \times m$ )-es területre osztjuk. Majd az egyes területeknek valamilyen *pooling* (*max* vagy *average*) eljárással meghatározott értékét egy vektorba rendezve átadjuk az első FC rétegnek. (Alapjában véve *max*- vagy *average pooling*ot végzünk úgy, hogy az ablakméretet akkorának választjuk meg, hogy mindig ugyanannyi darab ablakunk legyen.)



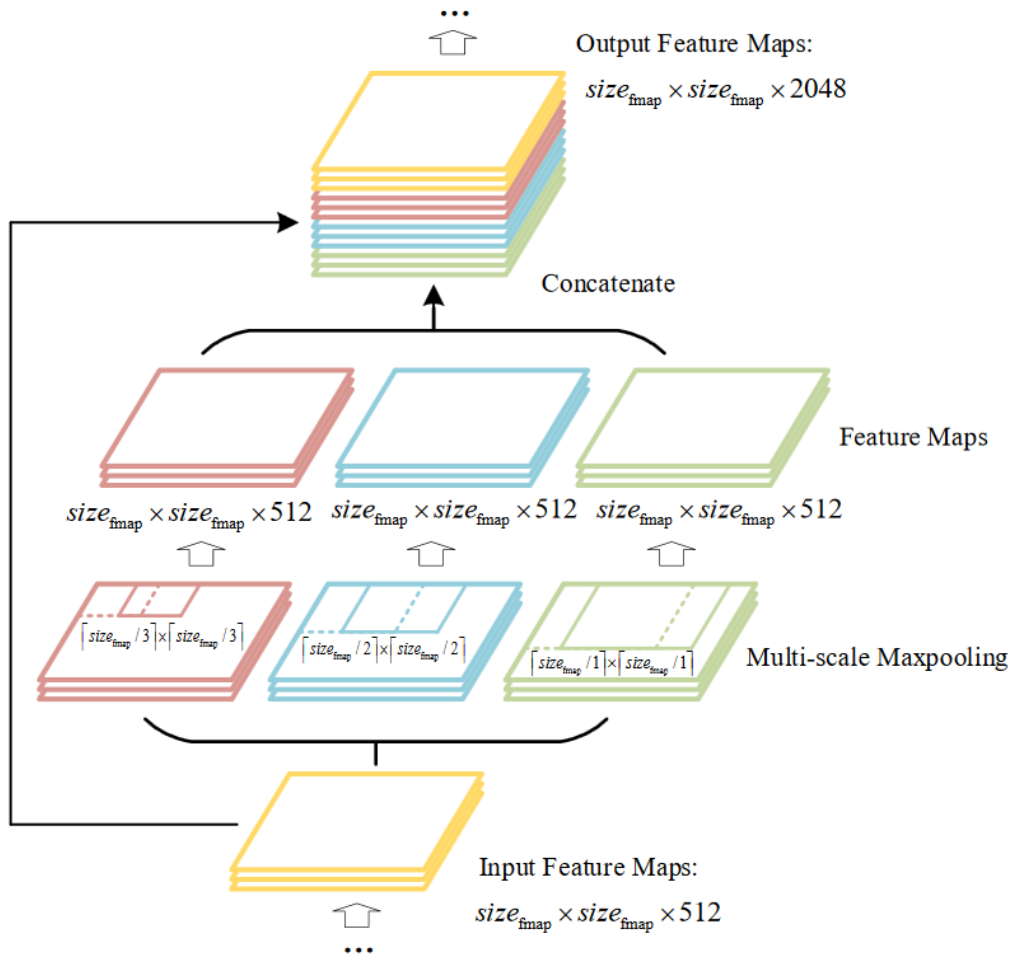
13. ábra - Az SPP réteg szerkezete. Forrás: [16]

Ahogy az az eljárásból is látható, akármekkora a méretű is a *feature map*, mindig ugyanakkora elemszámú vektort fog eredményezni az SPP réteg, mivel mindig ugyanannyi darabra osztjuk fel.

A YOLO viszont az SPP-nek egy átdolgozott változatát használja. [20] Először egy  $(1 \times 1)$ -es konvolúciós lépést alkalmaz, hogy lecsökkentse a filterek számát 1024-ről 512-re. Ezek után a *feature map*eken 3 különböző ablakméretű *maxpooling*ot hajt végre, ahol az ablakméretek a következők:

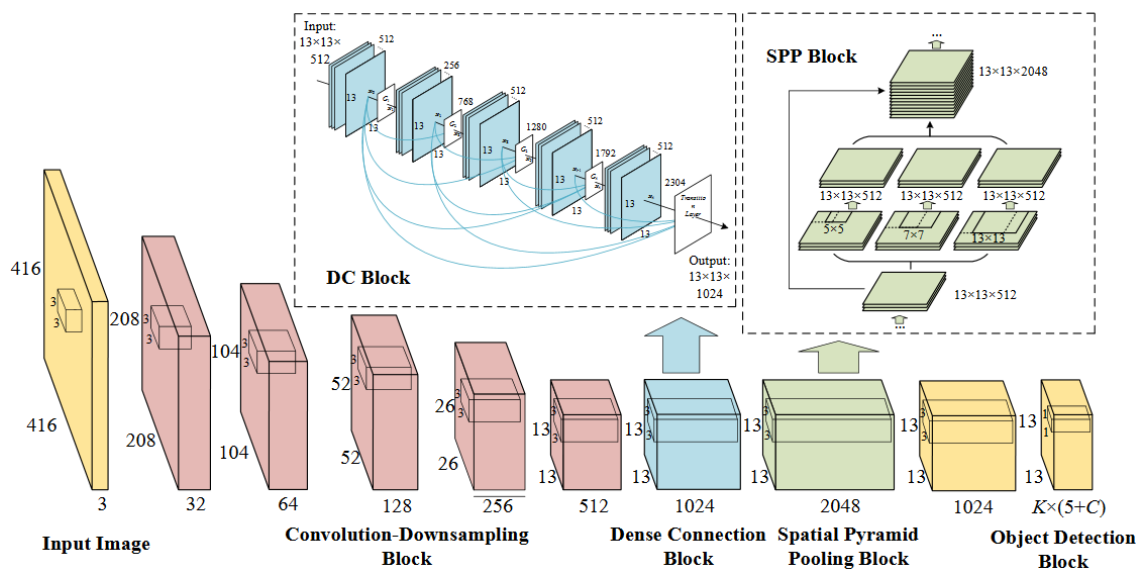
$$MÉRET_{MPOOL} = \left\lceil \frac{MÉRET_{FMAP}}{i} \right\rceil, (i = 1, 2, 3)$$

A lépések mérete (*stride*) 1, a *padding* értéke pedig úgy van meghatározva, hogy az eredmény mérete az eredeti *feature map* méretével azonos maradjon. Erre azért van szükség, hogy az így kapott 3 *feature map*et az eredetivel együtt összekonkatenálja, és így egy 2048 filterből álló tömböt kapunk. (3 · 512 a három *pooling* lépés eredményeként, és az eredeti az eredeti 512 filter.)



14. ábra - A YOLO átdolgozott SPP rétege. Forrás: [20]

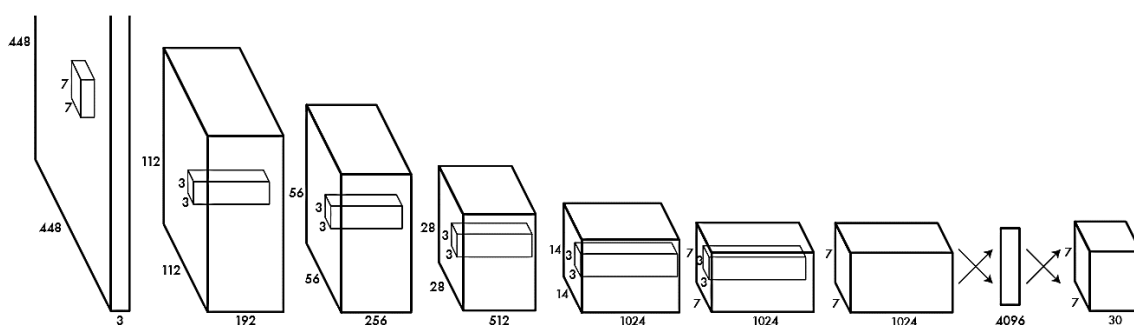
Az alábbi ábra mutatja, hogy az SPP réteg hogyan lett beillesztve a YOLO modellbe:



15. ábra - Az SPP réteg a YOLO modellen belül. Forrás: [20]

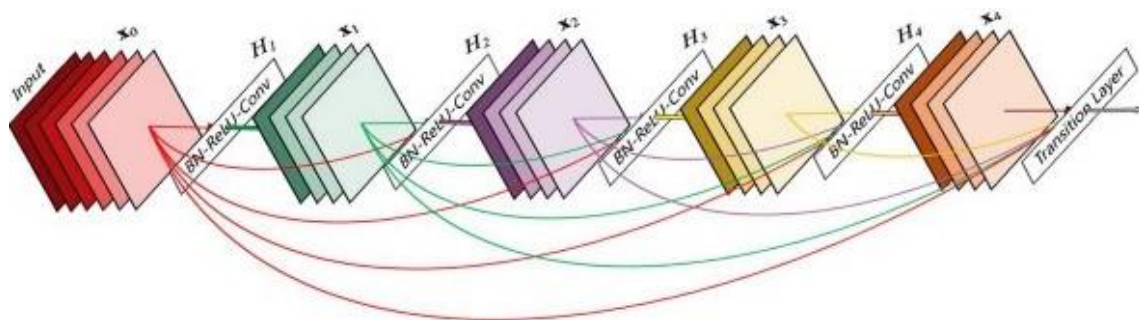
### Path Aggregation Network (PAN)

A korai *deep learning* modellek szerkezete viszonylag egyszerű volt. Minden réteg a bemenetét az előző rétegtől kapta. A legelső rétegek a képen belül meghatározott helyeken kerestek mintázatok, hogy ezekből szemantikus információkat nyerjenek a későbbi FC rétegek számára. Viszont a modell finomhangolásához szükséges kisebb részletinformációk elveszthettek ahogy az adat egyre több rétegen haladt át a modellben.



16. ábra - A YOLOv1 felépítése: példa a korai deep learning modellek egyszerű szerkezetére. Forrás: [9]

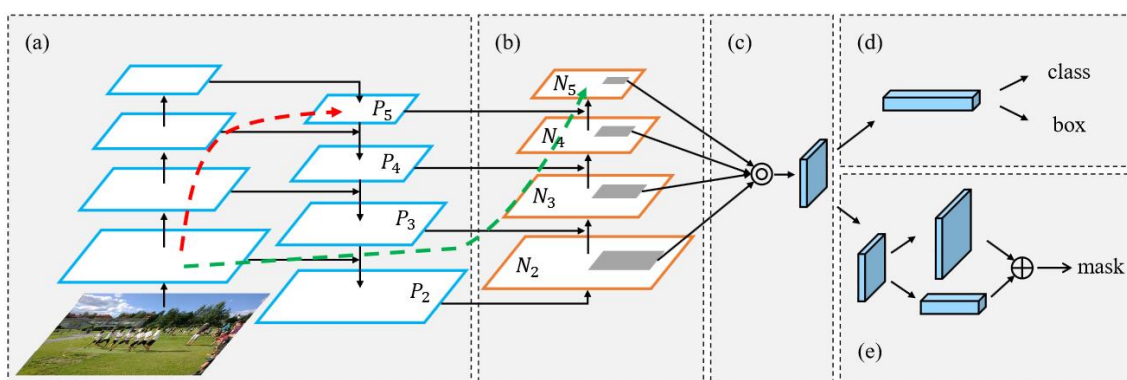
A *deep learning* későbbi fejlődésével a rétegek összekapcsolódásának és az információ párhuzamos áramlásának egyre bonyolultabb és bonyolultabb módjai jelentek meg. A *DenseNet* (lásd fentebb) ékes példája ennek, ahol a minden réteg az összes előzővel össze van kapcsolva.



17. ábra - Egy Dense Block rétegeinek összekapcsolódása a DenseNet struktúrán belül. Forrás: [18]

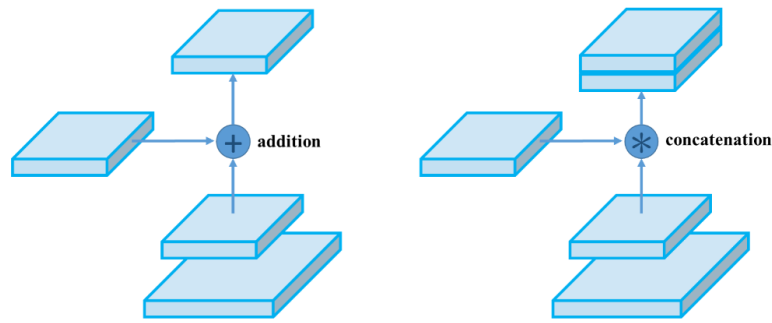
A rétegek közti információáramlás kulcskérdéssé vált a modellek szerkezetének megtervezésekor. Erre egyrészt a *vanishing gradients* nevű „betegség” kivédése miatt van szükség, másrészt ezáltal a modell első rétegeiben megjelenő *spatial information* elérhetővé válik a modell mélyebb rétegeiben is, ezáltal javítva a modell pontosságát.

A **PAN (Path Aggregation Network)** [17] struktúrában (lásd: 18. ábra) bevezetésre került egy újabb, köztes, „lentről-felfelé tartó” útvonal, amely megkönnyíti az alsóbb *feature map*ekben található információ felfelé áramlását. Az FPN-ben az információ az ábrán a **piros** nyíllal ábrázolt útvonalon halad a felső rétegek felé, és ez – habár ez az ábrán nem látszik – akár több mint 100 réteget is jelenthet. Ezzel szemben PAN-ban bevezetett kerülőút (zöld nyíl az ábrán) csak kb. 10 réteget jelent. Ez a koncepció lehetővé teszi, hogy az alsóbb rétegekben található részletes, „nagy felbontású” információ is rendelkezésre álljon a felsőbb rétegekben, jelentősebb torzulás nélkül.



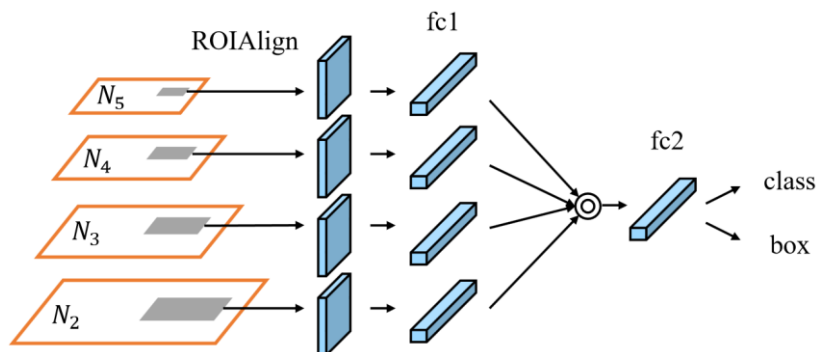
18. ábra - A PAN keretrendszer szerkezete: (a) FPN backbone; (b) „lentről-felfelé” történő információáramlást segítő extra útvonal; (c) adaptív „feature pooling” réteg; (d/e) bounding boxok és maszkok meghatározása. Forrás: [17]

A YOLOv4-ben viszont egy apró változtatást eszközöltek az eredeti PAN architektúrához képest, mégpedig a szomszédos *feature map*eket nem összeadják, hanem konkatenálják:



19. ábra - Balra: eredeti PAN; Jobbra: módosított PAN a YOLOv4-ben. Forrás: [12]

Az FPN architektúrában az objektumok detektálására a különböző felbontású rétegeken belül egymástól függetlenül került sor. Ez egyrészt ugyanazon objektumok többszöri detektálását eredményezheti, másrészt a detektálások során nem fér hozzá más felbontású *feature map*ekből származó információkhoz. A PAN ezzel szemben az összes *feature map*et az *adaptive feature pooling* nevű eljárással összefűzi: először minden rétegből létrehoz egy ugyanakkora méretű *fully connected* réteget, majd ezeket az FC rétegeket az elemenkénti maximum kiválasztás (*max-pooling*) módszerével összefűzi egy újabb FC rétegbe.



20. ábra - Az „adaptive feature pooling” megvalósítása a PAN architektúrában. Forrás: [17]

### **Spatial Attention Module (SAM)**

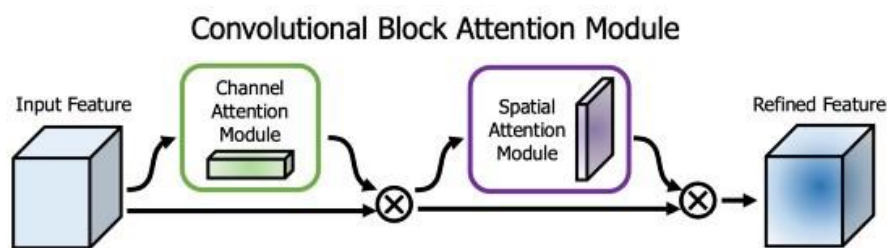
Az ún. *attention* modulokat legelőször a *Seq2Seq* (sequence-to-sequence) modelleket használó nyelvi fordító algoritmusokhoz fejlesztették ki. Az attention modulok felelősek egy architektúrában:

1. a be- és kimeneti elemek között fennálló (*General Attention*)
2. vagy a bemeneti elemek közti (*Self-Attention*)

kapcsolatok mértékének meghatározásáért és kezeléséért. Mára széles körben elterjedtek, többek között a képi feldolgozást végző CNN modellek is előszeretettel alkalmazzák őket.

Egy hasonlattal élve: Az emberi látás során – habár érzékeljük a perifériás látómezőnkben történő dolgokat is – a figyelmünk a látómezőnk közepére irányul. Az *attention* modulok is próbálják meghatározni, hogy a hálózat további rétegeinek is mely *feature*-ökre kell nagyobb hangsúlyt fektetniük. Teszik ezt azért, hogy a *feature map* „érdekesebbnek ítélt” *feature*-jeit nagyobb értékkel szorozzák meg (ezáltal kiemelve azokat), míg a többit egy alacsonyabb értékkel (vagyis elnyomva azokat). Talán innen is ered az elnevezés: *attention*, azaz figyelem.

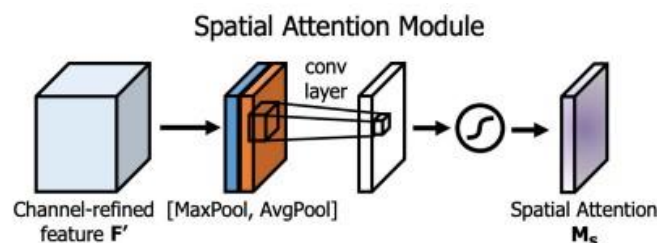
Az egyik ilyen, kifejezetten konvolúciós hálózatokhoz kifejlesztett *attention* modul a **CBAM**<sup>10</sup>, amely két almodulból épül fel: *Channel*- és *Spatial Attention Module*.



21. ábra - A CBAM két almoduljának kapcsolata. A  $\otimes$  az elemenkénti mátrixszorzást jelenti. Forrás: [21]

Alapjában véve a bemeneti *feature map* minden egyes csatornája egy-egy mintafelismerő filternek (*feature detector*) felel meg. A *Channel Attention Module* lényegében azt határozza meg (vagy legalábbis próbálja megjósolni), hogy mely filter tartalmazhat releváns információt, míg a *Spatial Attention Module* pedig arra fókuszál, hogy a képen belül hol található az értékes információ.

A SAM-en belül a bejövő *feature map*-ekre külön-külön egy *maximum pool* és egy *average pool* lépést alkalmaznak. Az így kapott *feature map*-eket összeillesztik, átengedik egy konvolúciós rétegen, majd végül sigmoid függvényen<sup>11</sup> is.

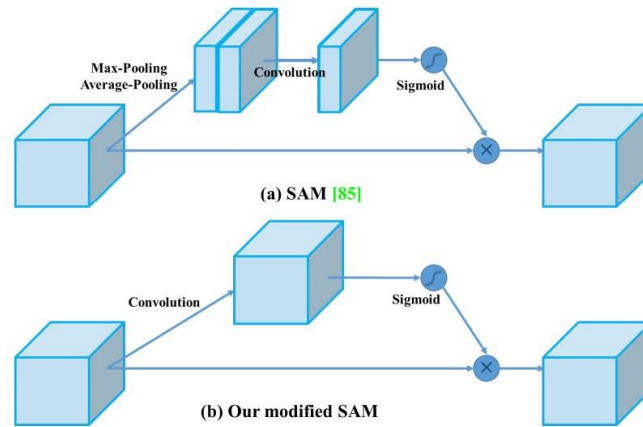


22. ábra - A SAM almodul felépítése a CBAM-en belül. Forrás: [21]

<sup>10</sup> **CBAM**: Convolutional Block Attention Module. Bővebben: [21]

<sup>11</sup> **Sigmoid függvények**: Matematikai függvények, amelyeket az „S” betűre hasonlító alakjukról kapták. A neurális hálózatokban gyakran alkalmazott változat az  $f(x) = \frac{1}{1+e^{-x}}$  függvény.

A YOLOv4-ben a SAM-nek egy módosított változatát alkalmazzák. A változtatás nem mindennapi: egészen egyszerűen kihagyják a MaxPool és az AvgPool lépéseket.



23. ábra - Az eredeti SAM (fent), és a YOLOv4-ben alkalmazott módosított SAM (lent). Forrás: [12]

## A magyar módszer

Miután a járművek detektálva lettek egy adott képkockán, össze kell őket párosítani a jelenleg követett gépjármű objektumokkal. De hogyan rendeljük hozzá az új detektálásokat a jelenleg követett járművekhez? Itt jön képbe a magyar módszer.

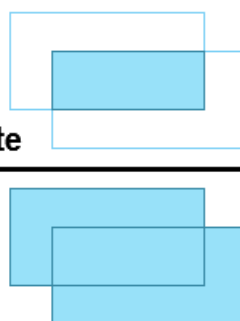
Páros gráfokban a javító út keresésének (és így a maximális elemszámú párosítás keresésének) algoritmusát Kőnig Dénes és Egerváry Jenő dolgozták ki. Az ő munkájuk nyomán Harold Kuhn 1955-ben kidolgozta a hozzárendelési problémát [22], amelyet a tiszteletükre *magyar módszernek* nevezett el. A módszer lényege az, hogy négyzetes költségmátrixokban lehet a sorok és oszlopok között párosításokat keresni úgy, hogy a sorok és oszlopok által a mátrixban kijelölt költségek összege minimális legyen:

$$C(x) = \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} \rightarrow \min \quad (4)$$

Esetünkben a költségeket a detektor által szolgáltatott *bounding box*ok és a követett járműobjektumok utolsó pozíciója (úgyisntén *bounding box*okról van szó) között kell számolni. Erre többféle költségfüggvényt is alkalmazhatunk:

- **IOU (Intersection Over Union):** A téglalapok metszetének és uniójának az aránya. 0 és 1 közötti értéket vehet fel.
- **Alaki- és méreti eltérésből számolt költség:** Minél nagyobb az alaki- (oldalak aránya) és méretbeli eltérés, annál nagyobb értéket rendelünk hozzá.
- **Konvolúciós érték:** A *bounding box*ok által meghatározott képrészleteket átengedjük egy CNN hálón, majd összehasonlítjuk az előző képkockán kapott értékekkel. Azonos *feature set* esetén (majdnem) azonos értékeket kell kapnunk.

A legegyszerűbb és legelterjedtebb megoldás az IOU értékek számolása, így én is ezt a módszert választottam.

$$\text{IOU} = \frac{\text{Metszet területe}}{\text{Unió területe}}$$


24. ábra - IOU számítása

A költségtáblázat megadásához a táblázat sorait megfeleltetjük az éppen követett járműveknek, az oszlopait pedig az *object detectortól* kapott *boundig box*oknak. Ezek után pedig kitöltjük a táblázatot úgy, hogy kiszámoljuk minden, az adott sorhoz tartozó járműobjektumnak az utolsó (előző képkockán detektált) pozíciójának és az adott oszlophoz tartozó, a jelenlegi képkockán detektált *boundig box*nak az IOU értékét.

	DET A	DET B	DET C
JÁRMŰ 1	IOU = 0	IOU = 0,56	IOU = 0
JÁRMŰ 2	IOU = 0	IOU = 0	IOU = 0,77
JÁRMŰ 3	IOU = 0	IOU = 0	IOU = 0

1. táblázat - Példa IOU költségtáblázatra

Az IOU értékekre meghatározhatunk egy küszöbértéket, vagyis, ha ez az érték alatt van az IOU, akkor nullát írunk a táblázatba, afelett pedig rendesen beírjuk a kiszámolt értéket. Ezzel



csökkenthetjük annak az esélyét, hogy egy kimaradt detektálás esetén tévesen az adott jármű mellett lévő másik jármű detektálását rendeljük hozzá.

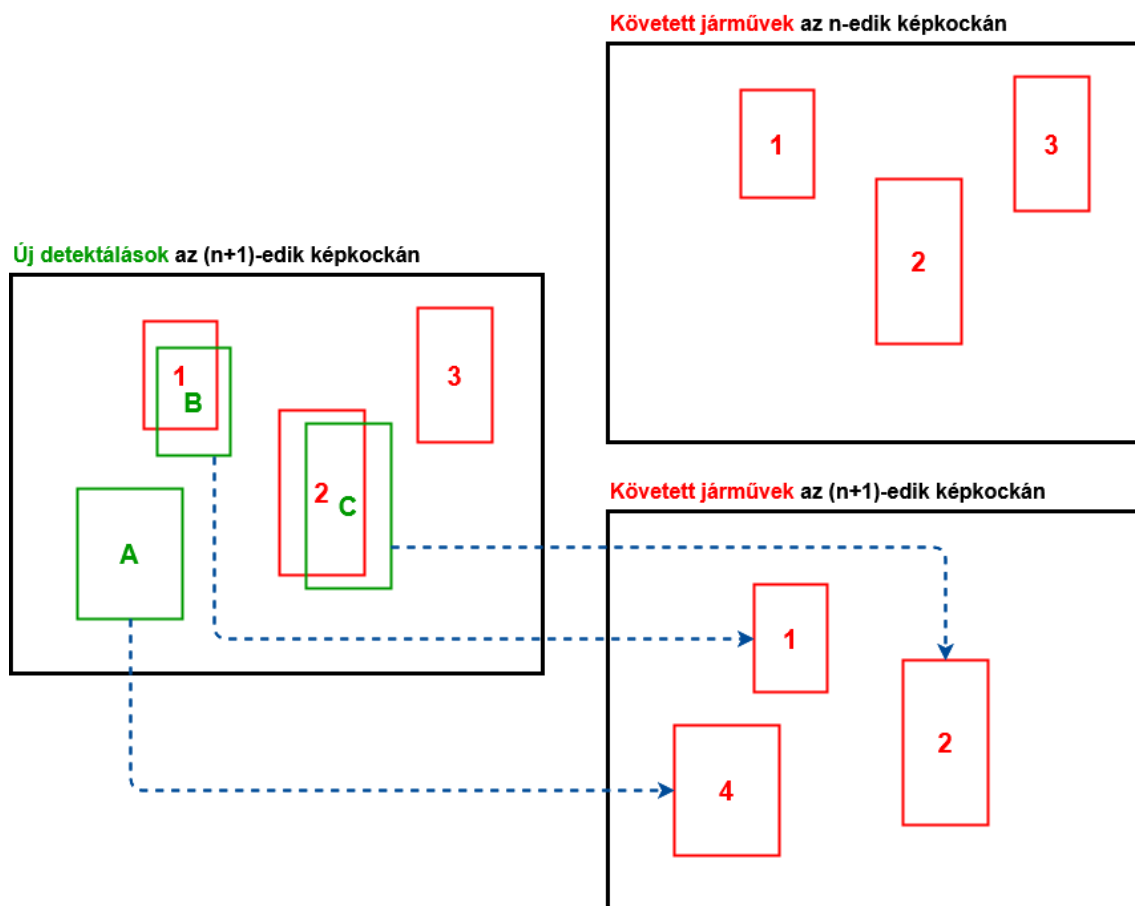
Mivel az algoritmus minimális költségű párokat keres, és mi minél nagyobb IOU értékű párokat szeretnénk találni, így valójában a  $(-IOU)$ , vagy az  $(1 - IOU)$  értékeket kell felvenni a táblázatba.

Az algoritmus eredményeként egy két oszlopból álló asszociációs táblázatot kapunk, amelynek az egyik oszlopában a járművek azonosítói (a költség táblázat hányadik sora), míg a másik oszlopában a hozzájuk rendelt új detektálások azonosítói (a költség táblázat hányadik oszlopa) található.

JÁRMŰ azonosító	DET azonosító
1	B
2	C

2. táblázat - Példa az asszociációs táblázatra

Ezek alapján az aktívan követett járművekhez hozzá lehet adni az adott képkockán detektált pozíciójukat. Azon járművek esetén, amelyeknek az azonosítója nem szerepel az asszociációs táblázatban, egy meghatározott N lépésig a fentebb említett optikai elven működő *tracking* veszi kezdetét, és azzal próbáljuk meghatározni a jármű helyzetét. Illetve az új járműdetektálásokhoz, amelyeknek az azonosítója nem szerepel a táblázat második oszlopában, új járműobjektumot hozunk létre.



25. ábra - Detektálások hozzárendelése a követett járművekhez

## Kálmán-szűrő

A Kálmán-szűrőt Kálmán Rudolf matematikus fejlesztette ki 1960-ban. Az algoritmus az állapotméréseket felhasználva komplex, változó rendszerek belső állapotáról ad optimális becslést, figyelembe véve a mérések pontatlanságát (zajokat, mérések bizonytalanságát).

Esetünkben a mérési bizonytalanságot a gépjárművek pozíciójának meghatározásakor a YOLO detektor adja. Mivel egy *object detector* sem képes 100%-os pontossággal meghatározni egy adott objektum pontos helyzetét, így az adatsorokat egyfajta zajszűrésnek kell alávetni, és ennek egy tökéletes eszköze a Kálmán-szűrő.

A Kálmán-szűrő egy rekurzív algoritmus, amely adott időpillanatban ( $k$ -edik iterációs lépésben) a rendszer belső állapotát a következőképpen definiálja:

$$x_k = Fx_{k-1} + Bu_{k-1} + w_{k-1} \quad (5)$$

ahol  $F$  az ún. állapot-átmenet mátrix,  $B$  a *control-input* mátrix, amivel a rendszer bemenetét leíró  $u_{k-1}$  vektort szorozzuk balról, míg a  $w_{k-1}$  a rendszerben található zajt reprezentáló vektor, amiről azt feltételezzük, hogy 0 átlagú Gauss-eloszlása van egy  $Q$  kovarianciamátrixszal<sup>12</sup>, azaz  $w_{k-1} \sim \mathcal{N}(0, Q)$ .

A rendszer belső állapota és a mérések közti kapcsolatot a következő egyenlet határozza meg:

$$z_k = Hx_k + v_k \quad (6)$$

ahol  $z_k$  a méréseket jelentő vektor,  $H$  a mérési mátrix, és  $v_k$  a mérési zaj, amiről – az előbbiekhez hasonlóan – ugyancsak azt feltételezzük, hogy 0 átlagú Gauss-eloszlása van, ez esetben egy  $R$  kovarianciamátrixszal, azaz  $v_k \sim \mathcal{N}(0, R)$ .

A Kálmán-szűrő a működése során az  $x_0$  kezdőállapotból kiindulva, minden  $k$ -adik lépésben megpróbál egy pontos közelítést adni az  $x_k$  állapotvektorra, ehhez felhasználva a rendszert leíró  $F, B, H, Q$  és  $R$  mátrixokat.

A Kálmán-szűrőnek amúgy következő három típusa létezik:

- **KF (Kalman Filter):** Ez az eredeti – Kálmán Rudolf által publikált – változat, amely lineáris rendszerekhez lett kifejlesztve.
- **EKF (Extended Kalman Filter):** Ezt követte a kibővített változat, amelyet már nem lineáris rendszerekhez fejlesztettek. Ennél a típusnál az  $F$  és a  $H$  mátrixokat egy  $f$  és  $h$  függvényekre cserélték, amelyeket minden egyes lépésben a pillanatnyi érték körül lineárisan közelítik (Taylor-sorfejtéssel). Hátránya, hogy a különböző valószínűségi változók a transzformáció után már nem Gauss-eloszlásúak, így a becslés nem optimális, illetve a Jacobi-mátrix előállítása eléggé számításigényes.

---

<sup>12</sup> **kovarianciamátrix:** A valószínűségszámítás és a statisztika tárgykörébe tartozó, két egymástól különböző változó mennyiség együtt mozgását kifejező mennyiséget kovarianciának nevezzük. A kis érték gyenge-, a nagy érték erős lineáris összefüggésre utal. A kovarianciamátrix vektorok együtt mozgását írja le, ahol a sorok az egyik, míg az oszlopok a másik vektor egyes elemeit jelentik. A mátrix egyes elemei pedig az adott sor és oszlop (a két vektor egy-egy adott eleme) közti kovarianciát tartalmazza.

- **UKF (Unscented Kalman Filter):** Az ezt követő „szagtalan” változatot ugyanúgy nem lineáris rendszerekhez fejlesztették, és alapjában véve az EKF hibáit hivatott orvosolni. (Kicsit gyorsabb, kicsit pontosabb.)

Habár egy gépjármű mozgása minden, de nem lineáris, én mégis az eredeti, lineáris rendszerekhez fejlesztett változatot fogom használni. Egyrészt mert könnyű implementálni magát a szűrőt, másrészt még annál is könnyebb hozzácsapni egy RTS-simítót<sup>13</sup>, és a kettő együtt majdnem tökéletes eredményt tud produkálni. Akkor nézzük meg, hogyan is működik a lineáris változat.

## Az algoritmus működése

A Kálmán-szűrő algoritmus két iteratív lépésből áll:

- **Állapotbecslés:**
  - becsült állapotvektor:  $\hat{x}_k^- = F\hat{x}_{k-1}^+ + Bu_k$
  - becsült hiba-kovarianciamátrix:  $P_k^- = FP_{k-1}^+F^T + Q$
- **Állapot frissítés:**
  - Kálmán-nyereség:  $K_k = P_k^-H^T(HP_k^-H^T + R)^{-1}$
  - frissített állapotvektor:  $\hat{x}_k^+ = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-)$
  - frissített hiba-kovarianciamátrix:  $P_k^+ = (I - K_kH)P_k^-$

A változók felső indexbeli – és + jelölése rendre az előzetesen becsült (*prior*) és a mérések utáni pontosított (*posterior*) állapotot jelenti, míg az állapotvektor felett a ^ jelölés arra utal, hogy a változó nem a rendszer pontos állapotát írja le, hanem csak egy közelített értéket jelent.



26. ábra - Kálmán-szűrő iterációs lépésének körforgása

<sup>13</sup> Rauch–Tung–Striebel

## Állapotbecslés

A mi gépjárművek helyzetét modellező  $\hat{x}_k$  állapotvektorunk és a hozzá tartozó  $P_k$  kovarianciamátrix a következő lesz:

$$\hat{x}_k = \begin{bmatrix} \text{pozíció} \\ \text{sebesség} \end{bmatrix} \quad (7)$$

$$P_k = \begin{bmatrix} \Sigma_{pp} & \Sigma_{ps} \\ \Sigma_{sp} & \Sigma_{ss} \end{bmatrix} \quad (8)$$

(Természetesen a *pozíció* és a *sebesség* is egy kételemű vektor ( $x$  és  $y$  koordináták), így az  $\hat{x}_k$  egy négyelemű vektor lesz, és ennek megfelelően a  $P_k$  is egy  $4 \times 4$ -es mátrix, de most az egyszerűség kedvéért csak az előbbi vektort és mátrixot fogom használni az algoritmus bemutatására.)

A továbbiakban a *pozíciót* és a *sebességet* rendre jelölje  $p$  és  $v$ .

Ahhoz, hogy egy adott  $k$ -adik lépésben megkapjuk a jármű pozícióját, az előző ( $k - 1$ ) pozíciójához hozzá kell adni az utolsó ismert sebességének és az eltelt időnek a szorzatát.

Azaz:

$$p_k = p_{k-1} + \Delta t \cdot v_{k-1} \quad (9)$$

$$v_k = v_{k-1} \quad (10)$$

Ezeket visszaírva az  $\hat{x}_k$ -s mátrix alakba:

$$\hat{x}_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \hat{x}_{k-1} = F \cdot \hat{x}_{k-1} \quad (11)$$

Ezt az  $F$  mátrixot nevezzük az állapot-átmenet mátrixnak.

Mivel egy-egy lépés egy-egy képkockát jelent a videófolyamban, vagyis mindig ugyanannyi idő telik el két lépés között, így a  $\Delta t$ -t egységnyiinek veszem, és a sebesség pedig jelölje a jármű két képkocka között megtett távolságát.

Ahhoz, hogy a  $P$  kovarianciamátrixot is frissíteni tudjuk, egy lemmát hívunk segítségül:

**LEMMA:**

Legyen  $A \in \mathbb{R}^{n \times n}$  mátrix,  $X \in \mathbb{R}^n$  valószínűségi vektorváltozó. Ekkor:

$$\text{cov}(AX) = A \cdot \text{cov}(X) \cdot A^T$$

**Bizonyítás:**

$$\begin{aligned} \text{cov}(AX) &= \mathbb{E}[(AX - \mathbb{E}[AX])(AX - \mathbb{E}[AX])^T] \\ &= \mathbb{E}[(AX - A\mathbb{E}[X])(AX - A\mathbb{E}[X])^T] \\ &= \mathbb{E}[A(X - \mathbb{E}[X])(X - \mathbb{E}[X])^T A^T] \\ &= A \underbrace{\mathbb{E}[(X - \mathbb{E}[X])(X - \mathbb{E}[X])^T]}_{\text{cov}(X)} A^T \end{aligned}$$

Azt már tudjuk, hogy  $\hat{x}_k = F \cdot \hat{x}_{k-1}$ . Ha erre alkalmazzuk az előbbi lemmát, akkor azt kapjuk, hogy:

$$P_k = \text{cov}(\hat{x}_k) = \text{cov}(F \cdot \hat{x}_{k-1}) = F \cdot \underbrace{\text{cov}(\hat{x}_{k-1})}_{P_{k-1}} \cdot F^T = F P_{k-1} F^T \quad (12)$$

Lehetnek olyan a rendszerünket befolyásoló külső hatások is, amelyek az állapotvektor által nincsenek reprezentálva, mégis hatnak a rendszerre. Esetünkben például az állapotvektor leírja egy gépjármű mozgását (pozícióját és sebességét egy adott időpillanatban), viszont a közlekedésben résztvevő járműveknek szinte folyamatosan változik a sebessége. Vagy gyorsítanak, vagy fékeznek. A gyorsulás mértékét a következőképpen adhatjuk hozzá a rendszerünket leíró, már az előbbiekben felírt egyenletekhez:

$$p_k = p_{k-1} + \Delta t \cdot v_{k-1} + \frac{1}{2} a \cdot \Delta t^2 \quad (13)$$

$$v_k = v_{k-1} + a \cdot \Delta t \quad (14)$$

Mátrix alakban:

$$\hat{x}_k = F \cdot \hat{x}_{k-1} + \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} a = F \cdot \hat{x}_{k-1} + Bu_k \quad (15)$$

Ahogy az már korábban említésre került, ezt a  $B$  mátrixot control mátrixnak nevezzük, az  $u$  vektor pedig control vektornak.

Ezen kívül a rendszerünkre hathatnak még egyéb külső tényezők, amelyek mértékét nem tudjuk meghatározni. Mozgó gépjárművek esetén ilyen lehet pl. az útburkolat egyenletlenségéből adódó sebesség- és irányváltozások. Ezek a rendszerünk bizonytalanságát növelik, amit a  $P$  kovarianciamátrix ad meg. Ezek reprezentálásához egész egyszerűen adjunk hozzá egy  $Q$  mátrixot a  $P$  kovarianciamátrix már eddig kiszámított értékéhez.

És ezzel meg is kaptuk az algoritmus első iterációs lépésének két egyenletét:

$$\hat{x}_k = F \cdot \hat{x}_{k-1} + Bu_k \quad (16)$$

$$P_k = FP_{k-1}F^T + Q \quad (17)$$

## Állapot frissítés

Most, hogy van egy „durva” becslésünk a rendszerünk belső állapotáról, érdemes ezt pontosítani néhány mérési adattal. A való életben a különféle szenzorok által szolgáltatott adatok zajosak, akárcsak a mi esetünkben a YOLO detektor által szolgáltatott, a jármű pozíciójára vonatkozó információk. Szerencsére a Kálmán-szűrő egyik legnagyobb erőssége pont abban rejlik, hogy rendkívül jól tudja kezelni a mérési zajokat, és egy jól inicializált szűrő gyakorlatilag már egy pár iterációs lépés után elég jól konvergál az általa reprezentált rendszer valódi állapotához.

Ahogy az sejthető, a különféle szenzorokból érkező adatok nem feltétlen kompatibilisek sem darabszámban, sem pedig a mértékegységük tekintetében az állapotvektorunkban tárolt adatokkal. Lehetséges, hogy pl. egy távolságérzékelő inchben adja meg a távolságot, mi viszont mm-ben kezeljük azt, illetve az esetünkben a YOLO detektortól csak a pozícióra

vonatkozó információt kapunk, sebességre vonatkozót nem, pedig az állapotvektorunkban mindkét adat szerepel. Éppen ezért a mérési adatainkat egy  $H$  transzformációs mátrixszal fogjuk modellezni, vagyis a rendszerállapot várható értéke és a hozzá tartozó kovarianciamátrix rendre a következők lesznek:

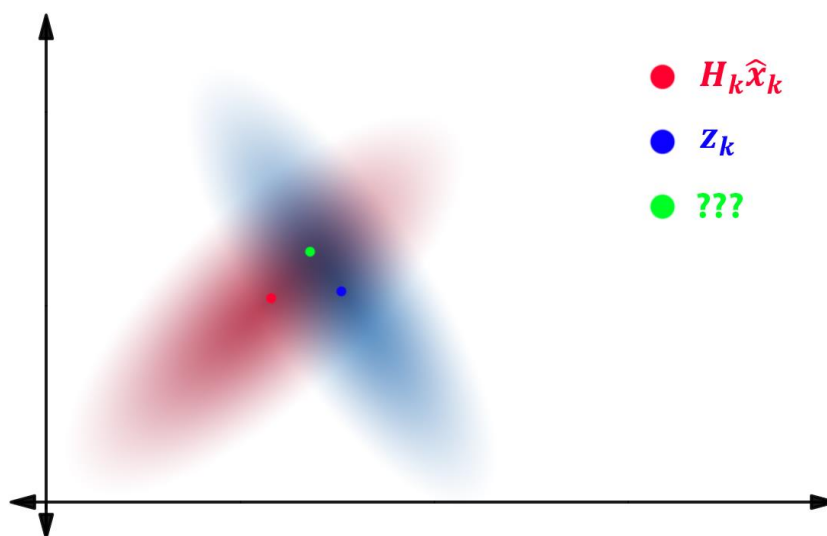
$$\mu_{becsült} = H_k \hat{x}_k \quad (18)$$

$$\Sigma_{becsült} = H_k P_k H_k^T \quad (19)$$

A méréseink bizonytalanságát az  $R_k$  kovarianciamátrix fogja megadni, míg az eloszlásunk várható értékének a mérési adatainkat vesszük, amik a  $z_k$  vektorban lesznek.

Itt érdemes megjegyezni, hogy a Kálmán-szűrő minden esetben normális eloszlású vektorváltozókat vár egy hamarosan ismertetésre kerülő tulajdonságuk miatt. A való életben ez a feltétel sokszor nem teljesül. Ilyen esetekben a szakirodalom azt javasolja, hogy a  $Q$  és az  $R$  kovarianciamátrixokat használjuk egyfajta finomhangoló paraméterként.

Vagyis van két Gauss-eloszlásunk (normális eloszlás). Az egyik az algoritmus előző iterációs lépésében kiszámolt, a szűrő által elvárt – a  $H$  transzformáció utáni -  $\hat{x}_k$  állapotvektorral, mint várható értékkel és a  $P_k$  kovarianciamátrixszal, míg a másik az előbb említett mérési adatok a maguk bizonytalanságával. Arra vagyunk kíváncsiak, hogy mi lesz a rendszerünk várható értéke úgy, hogy az a legnagyobb valószínűséggel szerepeljen mindkét eloszlásban.



27. ábra - Valószínűségek szemléltetése 2D-ben. A zöld pontot keressük.



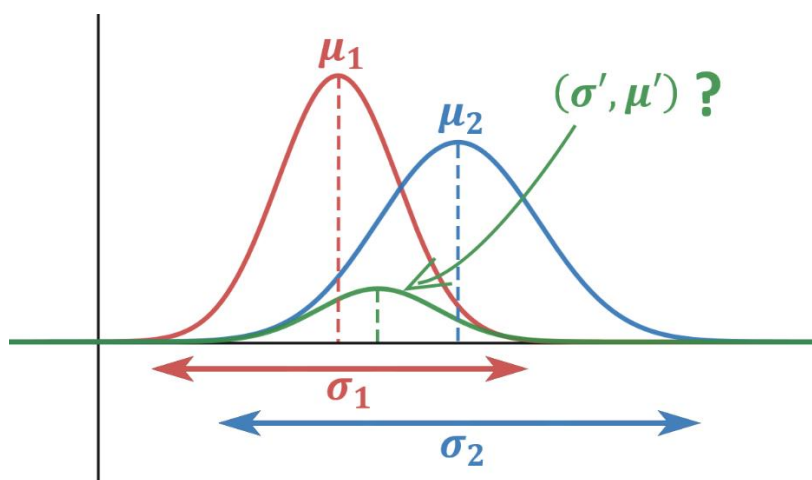
Ahhoz, hogy ezt megkapjuk nincs más dolgunk, mint a két Gauss-eloszlást – pontosabban a sűrűség függvényüket – össze kell szoroznunk. De hogyan is tudjuk ezt megtenni?

Nézzük a legegyszerűbb 1 dimenziós esetet. A normális eloszlás sűrűségfüggvénye egy  $\sigma^2$  szórásnégyzettel és egy  $\mu$  várható értékkel a következőképpen van definiálva:

$$\mathcal{N}(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (20)$$

Ha összeszorozunk két Gauss-görbét, akkor azok eredménye is egy Gauss-görbe lesz:

$$\mathcal{N}(x, \mu_1, \sigma_1) \cdot \mathcal{N}(x, \mu_2, \sigma_2) = \mathcal{N}(x, \mu', \sigma') \quad (21)$$



28. ábra - Két Gauss-eloszlás szorzata

Ezen szorzás elvégzésével a következő  $\mu'$  és  $\sigma'$  értékeket kapjuk eredményül [23]:

$$\mu' = \frac{\mu_1\sigma_2^2 + \mu_2\sigma_1^2}{\sigma_1^2 + \sigma_2^2} \quad (22)$$

$$\sigma'^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2} \quad (23)$$

Ezeket átalakítva kapjuk:

$$\begin{aligned}\mu' &= \frac{\mu_1\sigma_2^2 + \mu_2\sigma_1^2}{\sigma_1^2 + \sigma_2^2} = \frac{\mu_1\sigma_2^2 + \mu_2\sigma_1^2 + \mu_1\sigma_1^2 - \mu_1\sigma_1^2}{\sigma_1^2 + \sigma_2^2} = \frac{\mu_1(\sigma_1^2 + \sigma_2^2) + \sigma_1^2(\mu_2 - \mu_1)}{\sigma_1^2 + \sigma_2^2} \\ &= \mu_1 + \frac{\sigma_1^2(\mu_2 - \mu_1)}{\sigma_1^2 + \sigma_2^2}\end{aligned}\quad (24)$$

$$\sigma'^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2} = \frac{\sigma_1^2\sigma_2^2 + \sigma_1^4 - \sigma_1^4}{\sigma_1^2 + \sigma_2^2} = \frac{\sigma_1^2(\sigma_1^2 + \sigma_2^2) - \sigma_1^4}{\sigma_1^2 + \sigma_2^2} = \sigma_1^2 - \frac{\sigma_1^4}{\sigma_1^2 + \sigma_2^2}\quad (25)$$

Legyen  $k$  a következő:

$$k = \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}\quad (26)$$

És helyettesítsük is be a (25) és (24) egyenletekbe:

$$\mu' = \mu_1 + k(\mu_2 - \mu_1)\quad (27)$$

$$\sigma'^2 = \sigma_1^2 - k\sigma_1^2\quad (28)$$

Írjuk át ezeket mátrixos alakba, ahol  $\Sigma'$  jelentse a kovarianciamátrixát egy adott Gauss-eloszlásnak, míg a  $\vec{\mu}'$  vektorváltozó legyen az eloszlás várható értéke:

$$K = \Sigma_1(\Sigma_1 + \Sigma_2)^{-1}\quad (29)$$

$$\begin{aligned}\vec{\mu}' &= \vec{\mu}_1 + K(\vec{\mu}_2 - \vec{\mu}_1) \\ \Sigma' &= \Sigma_1 - K\Sigma_1\end{aligned}\quad (30)$$

Ezt a  $K$  mátrixot nevezzük Kálmán-nyereségnek.

Az eredeti két normális eloszlásunkat (mérések és az előzetes becslés) helyettesítsük be a (21)-es egyenletünkbe, azaz legyen:

- $(\mu_1, \Sigma_1) = (H_k \hat{x}_k, H_k P_k H_k^T)$
- $(\mu_2, \Sigma_2) = (z_k, R_k)$

Ezeket beírva a (30)-as egyenletekbe:

$$\begin{aligned} H_k \hat{x}'_k &= H_k \hat{x}_k + K(z_k - H_k \hat{x}_k) \\ H_k P'_k H_k^T &= H_k P_k H_k^T - K H_k P_k H_k^T \end{aligned} \quad (31)$$

A Kálmán-nyereség pedig a következő lesz:

$$K = H_k P_k H_k^T (H_k P_k H_k^T + R_k)^{-1} \quad (32)$$

Ezeket tovább egyszerűsítve meg is kapjuk az algoritmus második iterációs lépésének végső egyenleteit:

$$K = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \quad (33)$$

$$\begin{aligned} \hat{x}_k^+ &= \hat{x}_k^- + K(z_k - H_k \hat{x}_k^-) \\ P_k^+ &= P_k^- - K H_k P_k^- = (I - K H_k) P_k^- \end{aligned} \quad (34)$$

## Paraméterek beállítása

### Állapotvektor ( $x$ – State Vector)

A járművek állapotát leíró állapotvektorra ún. *nullad-* (konstans), *első-* (lineáris) és *másodfokú* (polinomiális) megközelítést alkalmazhatunk. Nulladfokú esetben az állapotvektor csak a pozíciókat tartalmazza, azaz  $x = [x \ y]^T$  lesz. Elsőfokú esetben az *egyenestvonalú egyenletes mozgás* egyenletét ( $x_t = x_{t-1} + v\Delta t$ ) próbáljuk modellezni, azaz az állapotvektor  $x = [x \ y \ \dot{x} \ \dot{y}]^T$  lesz. Másodfokú esetben pedig az *egyenestvonalú egyenletesen változó mozgás* egyenletét ( $x_t = x_{t-1} + v\Delta t + \frac{1}{2}a\Delta t^2$ ) próbáljuk a modellünkkel szimulálni, vagyis  $x = [x \ y \ \dot{x} \ \dot{y} \ \ddot{x} \ \ddot{y}]^T$  lesz az állapotvektorunk, ahol az egy- és kétpontos jelölések rendre a sebesség és a gyorsulás értékei lesznek az  $x$  és  $y$  tengelyek mentén.

(A nullad-, első- és másodfokú elnevezés abból adódik, hogy hányadfokú Taylor-polinommal tudjuk a mozgásegyenletünket közelíteni.)

A csak pozíciókkal való varázslás rendkívül rossz eredményt adna. Ezenkívül elmondható az is, hogy egy gépjármű mozgása minden, de nem lineáris. Viszont azt is tudjuk, hogy az „alap” Kálmán-szűrő lineáris modelleket vár, illetve esetünkben rendelkezésre áll a teljes adatsor, amiből a sebességértékek rendszeres frissítésével a gyorsulást „belecsempészhetjük” a modellünkbe. Vagyis állapotvektornak a már az előbbi levezetésben is használt, elsőrendű  $x$  vektort használom, azaz:

$$x = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}$$

ahol  $x$  és  $y$  a járművek pozícióját, míg  $\dot{x}$  és  $\dot{y}$  a járművek sebességét írja le egy adott időpillanatban külön-külön az  $x$  és az  $y$  tengelyek mentén.

### Állapot-átmenet mátrix ( $F$ – State Transition Matrix)

A következő lépés az  $F$  állapot-átmenet mátrixnak a meghatározása. Ennek a mátrixnak a már előbb említett  $x_t = x_{t-1} + v\Delta t$  egyenletet kell megvalósítania. Írjuk fel az állapotvektor minden egyes elemére külön-külön az egyenleteket:

$$\begin{aligned} x_k &= x_{k-1} + \dot{x}_{k-1} \cdot \Delta t \\ y_k &= y_{k-1} + \dot{y}_{k-1} \cdot \Delta t \\ \dot{x}_k &= \dot{x}_{k-1} \\ \dot{y}_k &= \dot{y}_{k-1} \end{aligned} \tag{35}$$

Mivel esetünkben minden egyes lépés egy-egy képkockát jelent a videofolyamban, így a  $\Delta t$ -t egységnyiinek vehetjük, és a (35)-ös egyenleteket tovább egyszerűsíthetjük:

$$\begin{aligned} x_k &= x_{k-1} + \dot{x}_{k-1} \\ y_k &= y_{k-1} + \dot{y}_{k-1} \\ \dot{x}_k &= \dot{x}_{k-1} \\ \dot{y}_k &= \dot{y}_{k-1} \end{aligned} \tag{36}$$

Ezeket mátrixos alakba átírva kapjuk:

$$\mathbf{x}_k = \begin{bmatrix} x_k \\ y_k \\ \dot{x}_k \\ \dot{y}_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \dot{x}_{k-1} \\ \dot{y}_{k-1} \end{bmatrix} = F \cdot \mathbf{x}_{k-1}$$

Azaz az  $F$  mátrix értéke:

$$F = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### Mérési mátrix ( $H$ – Measurement Matrix)

A mérési mátrix adja meg, hogy hogyan tudjuk az állapotvektorunkat leképezni a mérési vektorunkba a  $\mathbf{z} = H \cdot \mathbf{x}$  egyenletnek megfelelően. A YOLO detektortól csak a jármű jelenlegi pozíciójára vonatkozó információ ( $x$  és  $y$  koordináta) érkezik, vagyis a  $\mathbf{z}$  vektor kételemű lenne, de mivel nem valós időben próbáljuk követni a járműveket, hanem rendelkezésünkre áll a teljes adatsor egy adott jármű pozíciójára vonatkozóan, így abból tudunk számolni egy nem pontos, de a célnak megfelelő sebesség értéket. (Emlékezzük, hogy a Kálmán-szűrőnek az egyik erőssége pont az, hogy nagyon jól tudja kezelni a mérés- és számítás során keletkező zajokat.)

Azaz a  $\mathbf{z}$  vektorunk is 4 elemű lesz, akár csak az  $\mathbf{x}$ , így a  $H$  mátrixnak is  $4 \times 4$  eleműnek kell lennie. A  $H$  mátrixnak esetünkben semmi mást nem kell tennie, mint az  $\mathbf{x}$  elemeit rendre meg kell feleltetnie a  $\mathbf{z}$  elemeinek, vagyis:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### Mérési zaj ( $R$ – Measurement Noise Covariance Matrix)

A mérési vektorunk ( $\mathbf{z}$ ) alapján az  $R$  mátrixnak definíció alapján a következőnek kell lennie:

$$R = \begin{bmatrix} \Sigma_{pos}^2 & \Sigma_{pos}\Sigma_{vel} \\ \Sigma_{pos}\Sigma_{vel} & \Sigma_{vel}^2 \end{bmatrix}$$

A  $\Sigma$ -k a pozícióra és a sebességre vonatkozó szórásnégyzeteket tartalmazó részmátrixokat jelentik. Azaz például:

$$\Sigma_{pos}^2 = \begin{bmatrix} \sigma_x^2 & \sigma_x \sigma_y \\ \sigma_x \sigma_y & \sigma_y^2 \end{bmatrix}$$

Mivel  $x$  és  $y$  független valószínűségi változók, így  $\sigma_x \sigma_y = 0$ . A  $\sigma_x^2$  és  $\sigma_y^2$  szórásnégyzetek meghatározása viszont egy kicsit problémásabb feladat. A szórásnégyzet értéke definíció szerint:

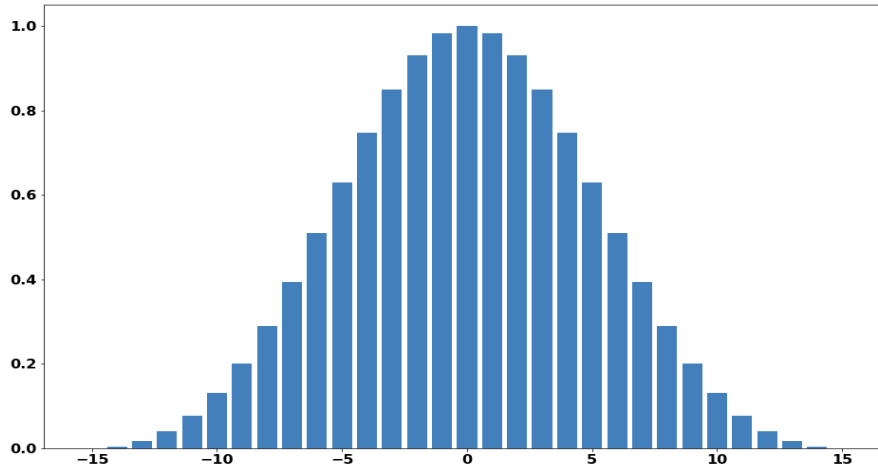
$$\sigma_x^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2 \quad (37)$$

azaz az adatsor értékeinek a várható értéktől vett távolságának a négyzetösszege, osztva az adatsor elemeinek a számával. A várható értéket pontosan nem tudjuk meghatározni, hiszen, ha tudnánk a jármű pontos pozícióját egy adott pillanatban, akkor nem lenne szükségünk erre a felhajtásra. Viszont egy elég pontos becslést tudunk adni az értékére, ha mozgóátlagot számolunk pl. egy Blackman-ablakkal<sup>14</sup>. Erre a feladatra igazából bármilyen simító ablakot használhatnánk, de a Numpy könyvtárban fellelhető megvalósítások közül ennek tetszett legjobban a neve, úgyhogy erre esett a választásom. De nézzük meg, hogy milyen elemeket is tartalmaz egy 31 elemű Blackman-ablak. Ennek a simító ablaknak az elemeit definíció szerint a következő függvény generálja:

$$w(n) = 0.42 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right) + 0.08 \cos\left(\frac{4\pi n}{N-1}\right) \quad n = 0, 1, \dots, N \quad (38)$$

---

<sup>14</sup> **Blackman-ablak:** A simító ablakokat eredetileg a jelfeldolgozásban a spektrális szóródás jelenségének csillapítására használják. Ezek egyike – pl. a Hanning-, Hamming-, Kaiser-Bessel-, stb. ablakok mellett – a Blackman-ablak is.



29. ábra - Egy 31 elemű Blackman-ablak értékei.

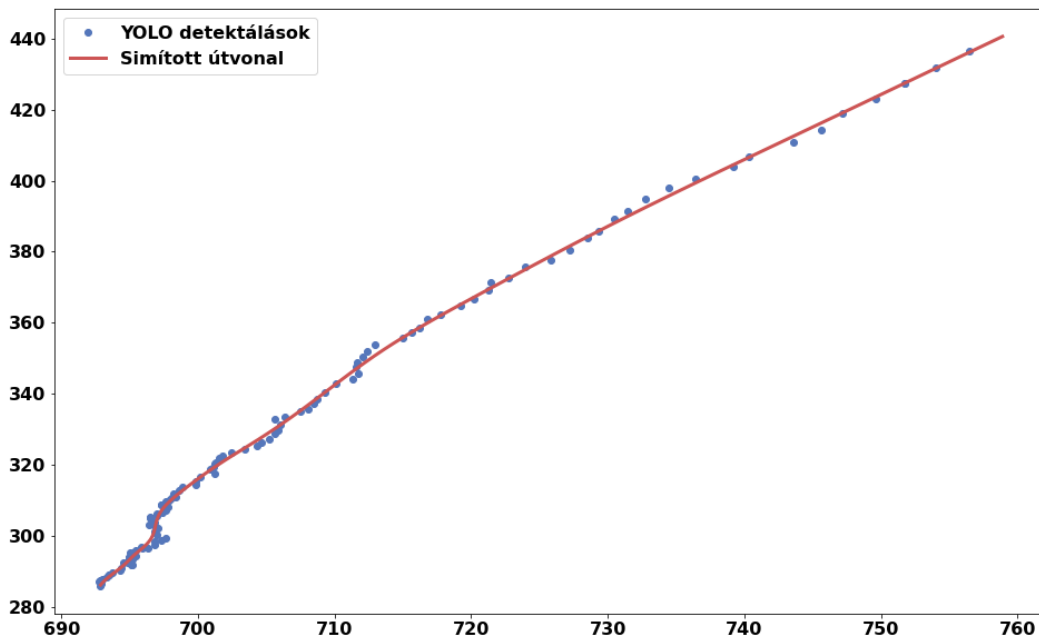
Ahogy az a fenti ábrán látható, lényegében egy 31 elemű tömböt kell elképzelni, amelyek a középső, 16. elemének az értéke 1, majd mind a tőle jobbra és ballra található elemek egyre kisebbek, ahogy haladunk a tömb szélei felé.

A módszerünk abból áll, hogy egy gépjármű első 31 detektálására súlyozott átlagot számolunk a Blackman-ablak elemeit súlyként felhasználva, majd ezt az ablakot egyesével eltologatva elvégezzük a műveletet az adatsor többi elemére is. Képlettel leírva:

$$x_i = \frac{1}{\sum_{k=1}^{31} b_k} \sum_{j=1}^{31} b_j x_{i-16+j} \quad i = 16, \dots, n - 15 \quad (39)$$

ahol  $b_i$  a Blackman-ablak  $i$ -edik eleme,  $x_i$  az  $i$ -edik detektálás értéke,  $n$  pedig a detektálások száma. Ezt a műveletsort természetesen mind az  $x$ , mind az  $y$  koordinátára elvégezzük egy adott képkockán. Így kapunk egy simított adatsort, ami elég közel lehet a gépjármű valódi pozíciójához, és viszonyítási pontként használhatunk a továbbiakban.

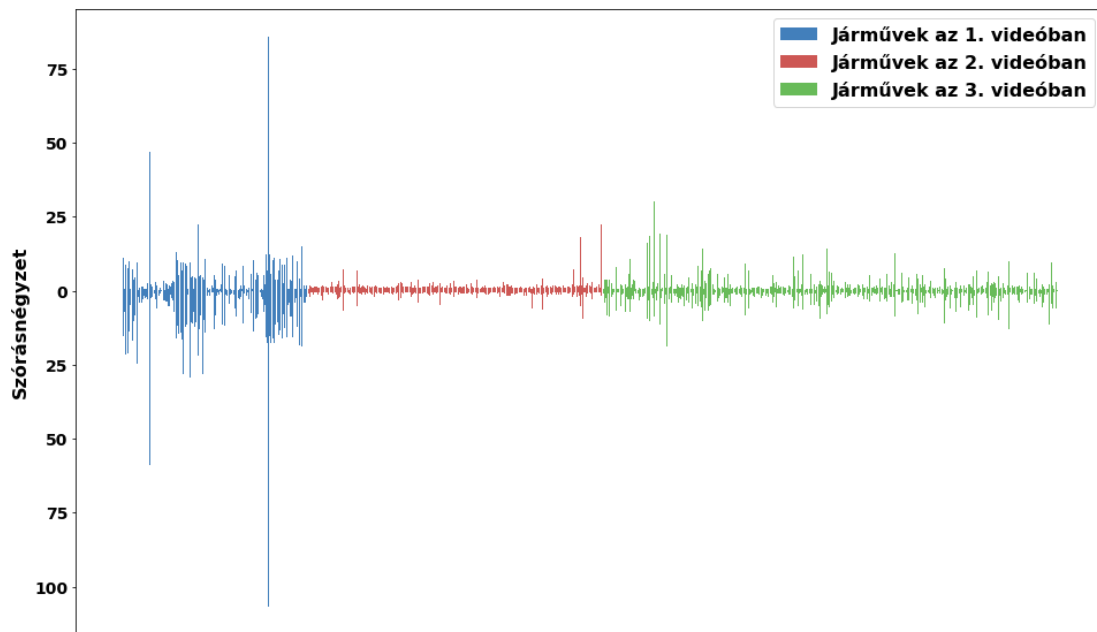
Felvetődik a kérdés, hogy miért nem használjuk az így kapott adatokat, és miért kell a Kálmán-szűrővel bűvészkedni? Különböző ablakméreteket próbálgatva arra jutottam, hogy ez az a legkisebb ablakméret, amivel még viszonylag zajmentes eredményt kapunk, viszont így az útvonalnak mind az elejéből, mind a végéből elvesztünk 15-15 képkockát. Ez egy 30 FPS-es képfrissítés esetén fél másodperc, ami idő alatt egy 90 km/h-val (25 m/s) haladó jármű 12,5 métert tesz meg. Ha egy ekkora szakaszt levágunk a detektált útvonalnak mind az elejéből, mind a végéből, az sok esetben (főleg a kamerához közel) megnehezíti a járművek virtuális kapukkal történő számlálását.



30. ábra - YOLO detektálások simítása Blackman-ablakkal.

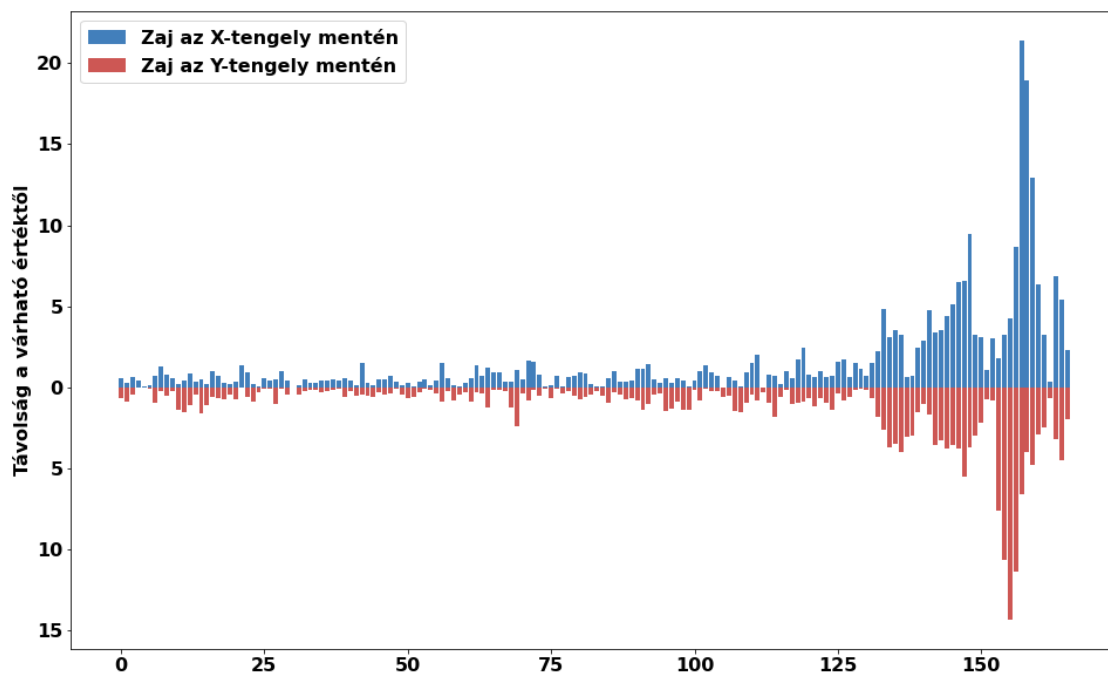
Ezek után 3 különböző videófájlban detektált összes gépjárműre kiszámoltam a szórásnégyzeteket. Ahogy azt a 31. ábra is jól mutatja, a detektor pontossága nem csak a különböző videók között mutat jelentős eltéréseket, de egy adott videón belül is elég nagy szórást tapasztalunk a különböző gépjárművek között. Ennek az egyik fő oka eltérő kamerapozíciókból adódik. Amikor a kamera viszonylag magasan helyezkedik el (pl. 2. videó az ábrán), akkor relatív kis szórást tapasztalunk. Fordított esetben (pl. 1. videó) pedig a számolt szórásnégyzetek kilőnek a sztratoszférába. Egy videón belüli kiugróan magas értékek pedig jellemzően akkor keletkeznek, amikor nagy járművek (pl. kamionok, buszok) a kamera látóterét elhagyják.





31. ábra - Három különböző videófájlban az összes detektált gépjárműre számolt szórásnégyzetek. A középvonal feletti értékek az x-tengelyre, míg az az alatti értékek az y-tengelyre számolt értékeket jelentik.

Az eddigi adatok által okozott kilátástalanság érzését tovább fokozza, ha vetünk egy pillantást egy adott gépjármű esetén keletkező zajra:



32. ábra - Zaj egy adott gépjármű esetén.

Ahogy az a fenti ábrán is jól látható, a zaj mértéke nem csak különböző gépjárművek esetén tud rendkívül eltérő eredményt adni, hanem egy adott gépjármű különböző detektálásai

esetén is elég nagy szórást tud mutatni. Habár a detektálások nagy része viszonylag pontosan történik, viszont bizonyos zavaró tényezők (részlegesen kitakart jármű, képernyőből kilógó jármű, stb.) hatására a detektor meg tud örülni.

Legvégül – több próbálkozás után – arra a döntésre jutottam, hogy az  $R$  mátrixban szereplő  $\sigma_x^2$  és  $\sigma_y^2$  értékének pozíció esetén 50-et, míg a sebesség esetén 2-t vettem fixen, vagyis az  $R$  mátrix értéke következő lesz:

$$R = \begin{bmatrix} 50 & 0 & 0 & 0 \\ 0 & 50 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Ez nyilván nem a legoptimálisabb megoldás, viszont ahogy azt mindjárt látni fogjuk, a már ugyancsak korábban említett RTS-simító algoritmus egy kevésbé pontos Kálmán-szűrővel is elég jó eredményt tud adni.

## Folyamat kovariancia mátrix ( $Q$ – Process Covariance Matrix)

Az  $Q$  mátrix a modellünkben jelenlévő bizonytalanságot hivatott reprezentálni. A szakirodalom ehhez hasonló esetekre kétféle modellt javasol:

- *Random velocity process noise model:*

$$Q = \begin{bmatrix} \Delta t^2 & \Delta t \\ \Delta t & 1 \end{bmatrix} \cdot q$$

- *Random acceleration process noise model:*

$$Q = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 \end{bmatrix} \cdot q$$

Ahogy azt már korábban is említettem, esetemben a  $\Delta t$  egységnyinek vehető, mivel a videófolyamban az egyes képkockák között ugyanannyi idő telik el. A  $q$  pedig tetszőleges konstans értéket jelöl, amely értékének a meghatározása a hivatalos álláspont szerint is a próbálkozás.

Kipróbáltam mindkét modellt több  $q$  értékkel is, de azonos  $q$  értékek mellett esetemben nagy különbséget nem véltem felfedezni a két modell között. (Feltételezhetőleg az egységnyinek vett  $\Delta t$  miatt.) A választásom legvégül a *Random acceleration process noise*

modellre esett, mivel a legtöbb általam látott példában ezt alkalmazták. Így legvégül a  $Q$  mátrixom a következő lett:

$$Q = \begin{bmatrix} 1/4 & 0 & 1/2 & 0 \\ 0 & 1/4 & 0 & 1/2 \\ 1/2 & 0 & 1 & 0 \\ 0 & 1/2 & 0 & 1 \end{bmatrix} \cdot 4 \cdot 10^{-2}$$

Az  $x$  és  $y$  tengelyek nyilván függetlenek egymástól, ezért kerültek 0-k az adott helyekre.

### **Megjegyzés:**

*Itt érdemes megjegyezni, hogy mind az  $R$ , mind a  $Q$  kovariancia mátrix beállításához léteznek adaptív módszerek is (pl: [24]). Ezekkel sokkal jobb eredmény érhető el, mint a „hagyományos”, konstans kovariancia mátrixok alkalmazása esetén. Viszont az RTS-algoritmus még így is minden tekintetben kielégítő eredményt tud produkálni számunkra, így eltekintettem az alkalmazásuktól.*

## **Állapot kovariancia mátrix ( $P$ - State Covariance Matrix)**

A  $P$  mátrix reprezentálja, hogy az algoritmus mennyire bízik az első iterációs lépésben generált becsült (*priori*) állapotvektorban. Az állapotvektorral együtt ez az egyetlen mátrix, aminek folyamatosan változik az értéke az algoritmus működése során, így ennek – akár csak az állapotvektornak – csak egy kezdeti  $P_0$  értéket tudunk megadni, ami a kezdeti  $x_0$  állapotvektorra vonatkozik. A szakirodalom ide valamilyen diagonális mátrixot javasol. Esetemben ez a következő lett:

$$P_0 = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot 10^{-1}$$

## Rauch–Tung–Striebel simító algoritmus

A Rauch-Tung-Striebel (RTS) simító egy kétlépéses, fix intervallumú simító algoritmus. Az első lépés egy az egyben megegyezik a Kálmán-szűrővel. Ebben a lépésben az adatokon „előre haladva” a Kálmán-szűrő minden iterációs lépésében elmentjük mind a mebecsült  $\hat{x}_k^-$ , mind pedig a pontosított  $\hat{x}_k^+$  állapotvektorokat és a  $P_k^-$  és  $P_k^+$  kovariancia-mátrixokat is.

A második lépésben az előbbi, elmentett változókat „finomítjuk”. Ebben a lépésben, az előzőleg megkapott adatsoron visszafelé haladva az alábbi lépéssorozatot hajtjuk végre minden iterációs lépésben:

$$\hat{x}_{k|n} = \hat{x}_{k|k} + C_k(\hat{x}_{k+1|n} - \hat{x}_{k+1|k}) \quad (40)$$

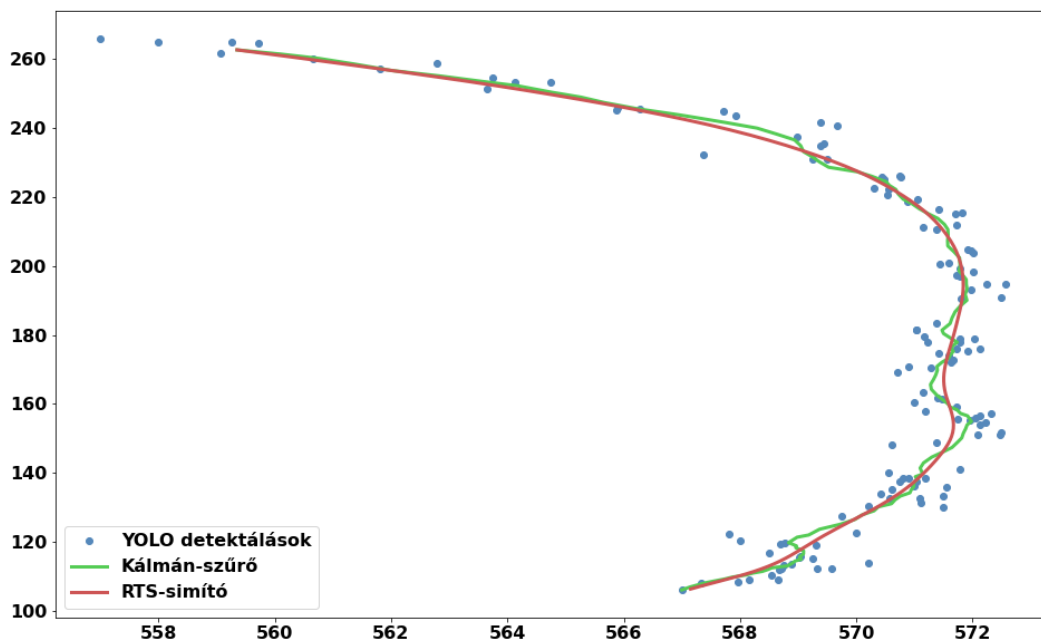
$$P_{k|n} = P_{k|k} + C_k(P_{k+1|n} - P_{k+1|k})C_k^T \quad (41)$$

ahol:

$$C_k = P_{k|k}F_{k+1}^T P_{k+1|k}^{-1} \quad (42)$$

Az előbbi (40)-es és (41)-es egyenletekben  $\hat{x}_{k|k}$  jelenti az első lépésben a Kálmán-szűrő által generált frissített (*posterior*) állapotvektort a  $k$ -edik lépésben, míg a  $\hat{x}_{k+1|k}$  a  $(k + 1)$ -edik lépésbeli becsült (*prior*) állapotvektort jelöli. Ugyanez a jelölés vonatkozik  $P_{k|k}$  és  $P_{k+1|k}$  kovarianciamátrixokra is.

Végezetül az alábbi ábra jól szemlélteti, hogy egy még egy rendkívül zajos környezetben is ez a kétlépéses simító algoritmus rendkívül jó eredményeket tud produkálni. Ennek persze feltétele – a korábban bemutatott módon – egy megfelelően felparaméterezett Kálmán-szűrőnek az elkészítése.



33. ábra - Egy adott jármű YOLO detektálásai, ezek Kálmán-szűrővel feldolgozott, majd pedig az RTS-simítóval tovább finomított útvonala.

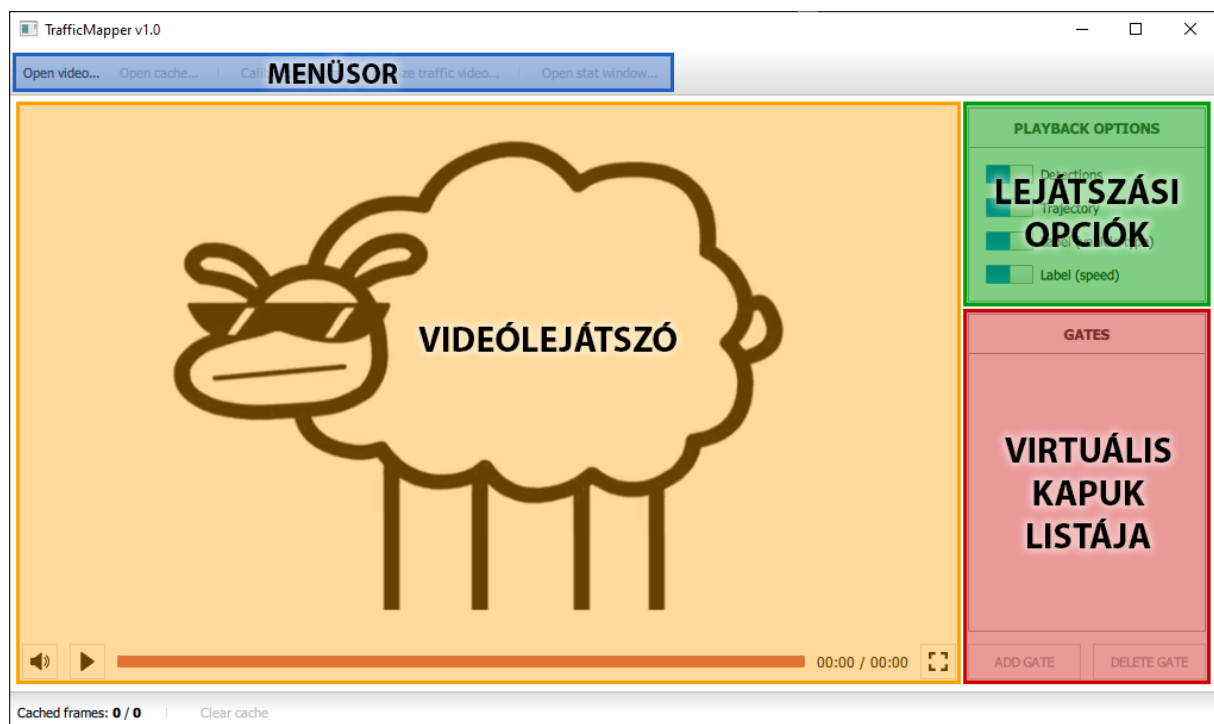
# Felhasználói dokumentáció

## Áttekintés

A program fixen rögzített, közutak forgalmát rögzítő kamerák felvételéből képes a felvételen látható járművek követésére és statisztikák létrehozására. A program működésének lépései a következők:

- Videó fájl betöltése
- Cache fájl betöltése (opcionális)
- Kamera kalibráció beállítása (opcionális)
- Virtuális forgalomszámláló kapuk megadása
- Videófolyam analízálása
- Statisztikák megjelenítése

## A kezdőképernyő felépítése



34. ábra – Kezdőképernyő felépítése

A kezdőképernyő 4 részből áll:

- Menüsor
- Videólejátszó
- Lejátszási opciók
- Virtuális kapuk listája

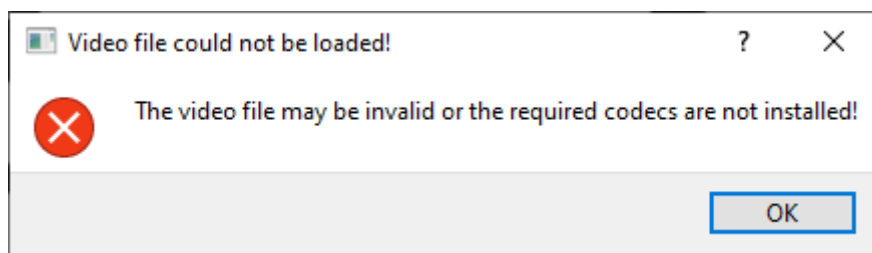
## Program használatának lépései

### Videófájl betöltése

A menüsoron az „Open video...” gomra kattintva megnyílik az operációs rendszer sztenderd fájlkiválasztó párbeszédablaka (operációs rendszerenként eltérő lehet), amelyben egy helyileg tárolt videófájlt kiválasztva azt be tudjuk tölteni.

Ahhoz, hogy egy videófájlt meg tudjunk nyitni, a rendszeren telepítve kell lennie az adott videófájl lejátszásához szükséges codec-nek telepítve kell lennie. A K-Lite Codec Pack gyakorlatilag minden ismert videóformátum dekódolásához szükséges codek-et tartalmaz, ennek a használata javasolt.

Amennyiben valamilyen okból kifolyólag egy fájlt nem tudunk megnyitni (szükséges codec nincs telepítve, vagy a fájl nem videófájl, esetleg sérült), a következő hibaüzenetet kapjuk:



35. ábra - Videófájl sikertelen megnyitása esetén látható hibaüzenet.

A videófájl sikeres betöltése után a lejátszás elindítható, illetve a lejátszási pozíció a *progress bar*-ra kattintva megváltoztatható.

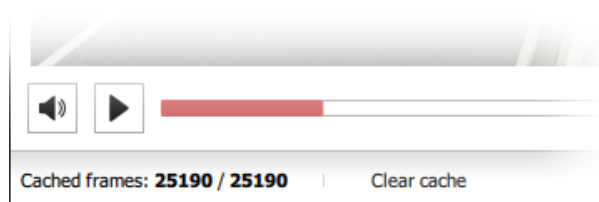
## Cache betöltése

Ez a lépés opcionális. Mivel a YOLO detektor futtatása rendkívül számításigényes (emiat hosszadalmas) folyamat, így a programhoz mellékelt videófájlokhoz tartoznak *cache* fájlok is, amelyek lényegében egyszerű szöveges fájlok, és a YOLO detektor NMS<sup>15</sup>-sel szűrt kimenetét tartalmazzák. Például:

```
0 4
2 0.9972610473 0.5372621417 0.3980102539 0.0173080284 0.0402680449
2 0.9799189567 0.4462951123 0.4011403918 0.0165650248 0.0384823679
2 0.8224077224 0.5559462308 0.3750407993 0.0146834189 0.0340093746
2 0.5864080190 0.5343507528 0.3826115429 0.0115941921 0.0187944602
1 4
...
```

Az első sor a képkocka indexét, majd a képkockán található detektálások számát tartalmazza. Ezután a detektálások számának megfelelő számú sorban maguk a detektálások találhatók. Az első szám a jármű kategóriája, a második a detektor által adott valószínűségi index, az utolsó négy számjegy pedig a *bounding box* pozíciója (x, y, width, height) a kép szélességének és magasságának arányában. És ez így ismétlődik.

A cache fájl betöltéséhez a menüsoron az „Open cache...” gomra kattintva, a videófájl megnyitásához hasonlóan ki kell választani a videó nevével megegyező .tmc (*TrafficMapper Cache*) kiterjesztésű fájlt. Ha sikeresen betöltöttük a cache fájlt, akkor a bal alsó sarokban megjelenik a cache-ben található képkockák száma:



36. ábra - Gyorsítótár tartalma

Az első szám a cache tartalma, a második az összes képkocka számát jelenti. A mellettük található „Clear cache” gombra kattintva pedig – egy megerősítést követően – törölhetjük a gyorsítótár tartalmát.

---

<sup>15</sup> Non-Maximum Suppression

Amennyiben nem töltünk be cache fájlt, és futtatjuk a detektor, akkor a futás közben generált detektálások ugyanúgy bekerülnek a gyorsítótárba, így egy esetleges második, pl. más paraméterekkel történő futtatásnál a tárban található adatokat fog dolgozni, és a detektor nem fog ezekre a képkockákra újból lefutni.

## Forgalomszámláló kapuk megadása

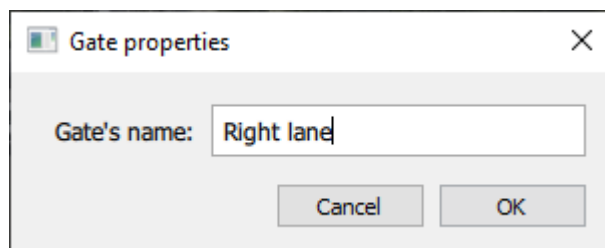
Ahhoz, hogy a forgalmat számlálni tudjuk, meg kell határoznunk a virtuális forgalomszámláló kapuk helyzetét. Ezt a bal alsó sarokban található **ADD GATE** gombra kattintva tehetjük meg. A gombra kattintva annak színe pirosra változik, ezzel jelezve, hogy kijelölhetjük a kapu helyzetét a képen. Ehhez az egérrel a képen a bal egérgombot lenyomva (*click-and-drag*) tartva a képen „megrajzolhatjuk” a kaput.



37. ábra - Főképernyő egy elhelyezett kapuval

A gomb felengedése után egy párbeszédpanel jelenik meg, amiben a későbbi azonosítás céljából nevet adhatunk a kapunak:





38. ábra - Kapu nevének megadására szolgáló párbeszédablak

Utólag a listában kijelölt kapu a **DELETE GATE** gombra kattintva törölhető.

## Kamera kalibrálása

Ez a lépés is opcionális. Amennyiben kihagyjuk, úgy a sebességek kiszámítására nem kerül sor. A kalibráció elvégzését a menüsoron található **Calibrate camera...** gombra kattintás után felugró párbeszédablakban lehet megtenni.

A kalibrációhoz ki kell jelölnünk egy, az útburkolat síkjával párhuzamos téglalapot, és meg kell adnunk ezen a téglalap oldalainak a hosszát. Ehhez a bal alsó sarokban található **SET** gombra kattintás után kezdjük el a sarokpontokat a képen történő kattintásokkal meghatározni.



Ha ezzel megvagyunk, meg kell adni a téglalap oldalainak a méretét az ablak alján található két beviteli mező segítségével. Ezek után az **Apply** gombra kattintva a program kiszámítja a homográfiát, és ellenőrzésképpen az kijelölt téglalappal párhuzamos, sárga pontokkal jelölt síkot rajzol az útburkolatra, mely pontok között mind a függőleges, mind a vízszintes irányban a távolság pontosan 1 méter.



39. ábra - Kiszámított homográfia megjelenítése

Végezetül a **SAVE** gombra kattintva elmenthetjük a beállítást, míg a **CANCEL** gombra kattintva törölhetjük a kiszámított leképezési mátrixot.

## Videó elemzése

A videó elemzéséhez a főmenü **Analyze traffic video...** menüpontjára kattintva tudjuk megtenni. A gombra kattintva felugrik egy párbeszédablak, amelyben megadhatjuk a folyamat különféle paramétereit.

A paramétereket a következő 3 csoportba sorolhatók:

- **YOLO Object Detector**

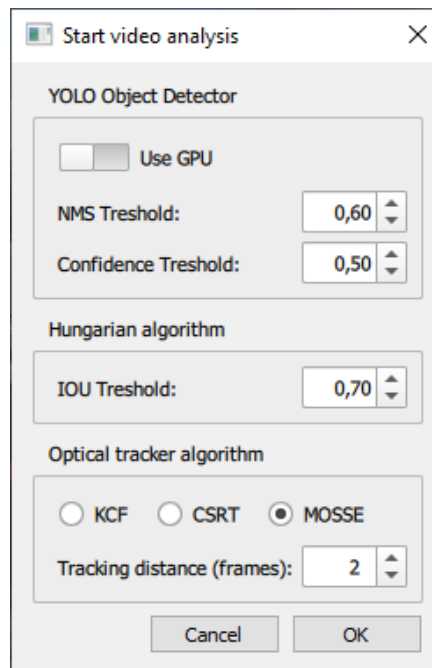
- *Use GPU*: Ezzel engedélyezhetjük a grafikus gyorsító használatát a YOLO számára.
- *NMS Threshold*: A Non-Maximum Suppression küszöbértéke.
- *Confidence Threshold*: Valószínűségi küszöbérték az összes detektálás számára.  
Az ennél kisebb valószínűségi pontszámmal rendelkező detektálásokat elvetjük.

- **Hungarian Algorithm**

- *IOU Threshold*: IOU küszöbérték a járművek követéséhez. Az egymást követő képkockákon csak az ennél nagyobb IOU értékű boundig-boxokat tekintjük potenciális pároknak.

- **Optical Tracker Algorithm**

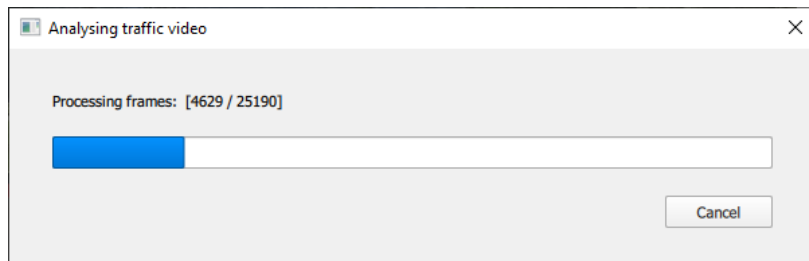
- *KCF / CCSRT / MOSSE*: Követő algoritmus a kimaradt detektálások áthidalására.
- *Tracking Distance*: Optikai elvű követés hossza képkockák számában megadva.  
Ha az itt megadott érték után sem érkezik újabb detektálás a YOLO-tól, akkor a jármű követését befejezettnek tekintjük.



40. ábra - Az algoritmus paramétereinek a megadása

A paraméterek beállítása után az OK gombra kattintva elkezdődik a videó feldolgozása. Felugrik egy párbeszédablak, ami folyamatosan jelzi a feldolgozottság állapotát. A feldolgozás folyamata bármikor megszakítható a **Cancel** gombra kattintva. Az addig

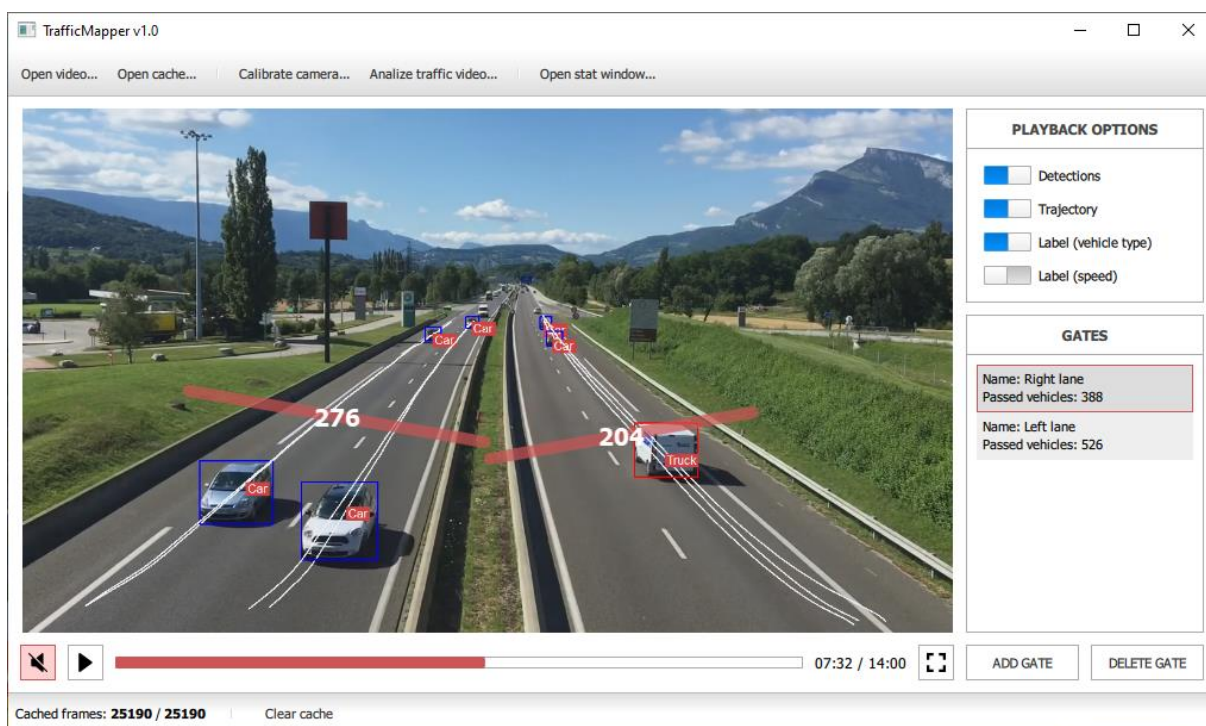
feldolgozott detektálások bekerülnek a gyorsítótárba, így az algoritmus ismételt (esetleg más paraméterekkel történő) elindítása esetén a már feldolgozott képkockák újbóli feldolgozására nincs szükség.



41. ábra - Videó feldolgozottságának állapotát jelző párbeszédablak.

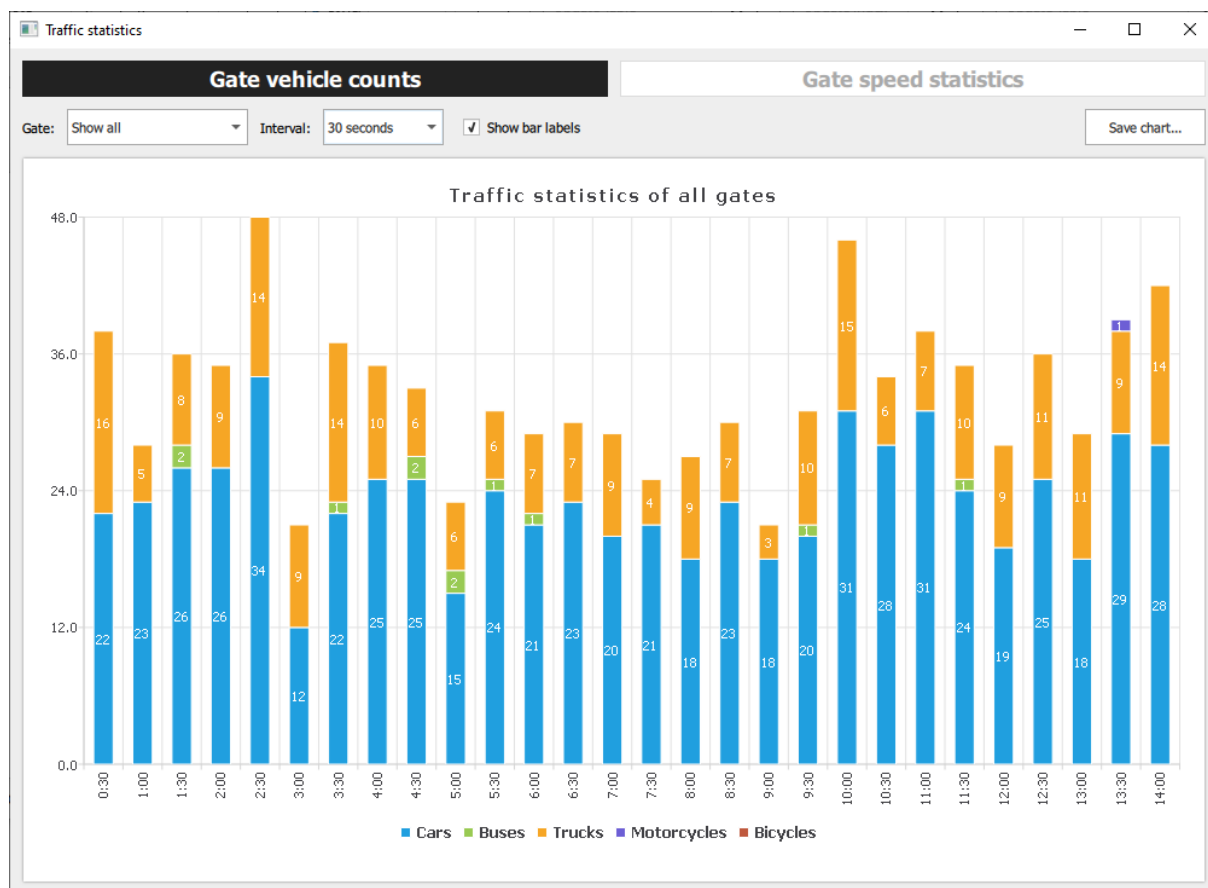
## Statisztikák megjelenítése

Miután a videó feldolgozása megtörtént, a forgalomszámláló kapuk listájában a kapuk neve alatt megjelenik az adott kapun áthaladt összes járműnek a száma. A videó lejátszást elindítva pedig már a járművek körül kirajzolásra kerülnek a detektálások (bounding-boxok), a járművek útvonala, illetve egy címke a jármű típusával és sebességével. Ezek megjelenítését a **PLAYBACK OPTIONS** panelban található kapcsolókkal lehet ki- és bekapcsolni.



42. ábra - Videó lejátszása a már detektált járművek adataival.

További részletesebb statisztikai adatok érhetők el a főmenü **Open Stat Window...** gombjára kattintva felugró ablakban. Itt oszlopgrafikonokban időintervallumokra és járműtípusokra lebontva megtekinthető akár adott kapukon, akár az összes kapun összesítve áthaladó járművek száma és átlagsebessége.



43. ábra - Részletes járműstatisztikák ablaka.



# Fejlesztői dokumentáció

## Feladat

A program fixen rögzített kamerák által, közúti forgalomban résztvevő járművekről készített felvételekből képes a forgalomra vonatkozó információkat kinyerni. Ehhez egy *YOLO Object Detector*tal először felismeri minden egyes képkockán a járművek helyzetét és típusát, majd képkockáról-képkockára haladva ezekből a detektálásokból meghatározza, hogy melyek tartoznak ugyanazon járműhöz.

Miután megkaptuk a jármű útvonalát, a program ellenőrzi, hogy ezen járművek áthaladtak-e a felhasználó által meghatározott virtuális forgalomszámláló kapukon. Ezen kívül – a kamera előzetes kalibrálása esetén – a program képes a járművek pillanatnyi sebességről is egy becslést adni.

Végezetül, ezen adatokat felhasználva a forgalomról egy összefoglaló statisztikát készít a program. Ezen adatok megjeleníthetők akár az eredeti videót lejátszva, azon egy *overlay* réteggént, akár egy külön ablakban grafikonok formájában.

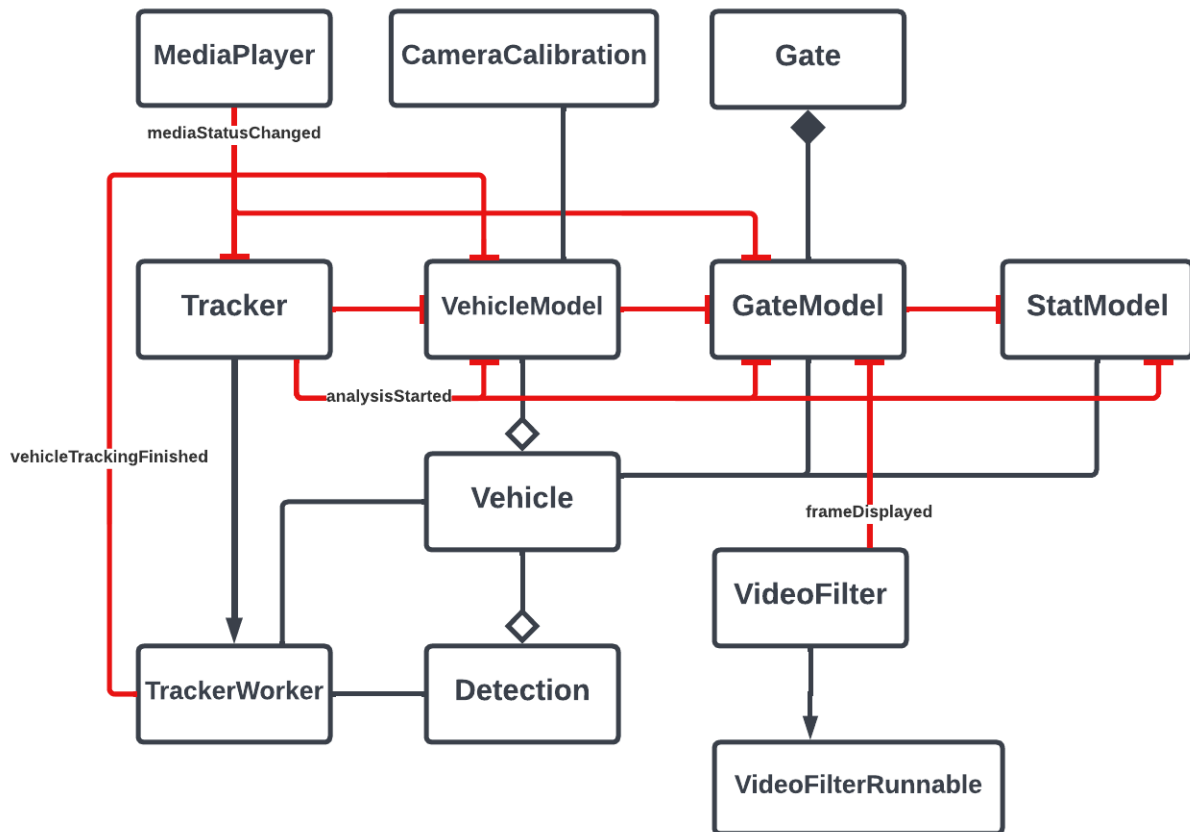
## A program felépítése

A program C++ nyelven, a **Qt Framework** használatával készült. A Qt-ről a hagyományos, a C++-ban jól ismert objektumorientált paradigmák használatán felül bevezet egy ún. *signal-slot* mechanizmust. ([Bővebben...](#)) A módszer lényege, hogy adott objektumok *signal*-jait összekapcsolhatjuk más objektumok *slot*-jával, és ezeken a kapcsolatokon keresztül „jelzéseket” küldhetnek egymásnak az objektumok. Ilyen típusú kapcsolatok ábrázolására az UML nem alkalmas, így ezek jelölésére az osztálydiagramon a következő piros színű jelölést alkalmaztam:



44. ábra - A Qt SIGNAL-SLOT kapcsolatainak ábrázolása az osztálydiagrammon.

Az előbbi jelöléseket felhasználva a program osztálydiagramja a következő:



45. ábra - Osztálydiagram

## Osztályok leírása

Az UML-ben a változók és tagfüggvények láthatóságának jelölésére a név előtti + (*public*), # (*protected*) vagy – (*private*) jelölést alkalmazzuk. Ahogy az fentebb említésre került, a Qt-ban található *signal*-ok és *slot*-ok osztálydiagrammokban történő jelölésére a szabvány UML láthatósági jelölések a következőkkel lesznek kiegészítve:

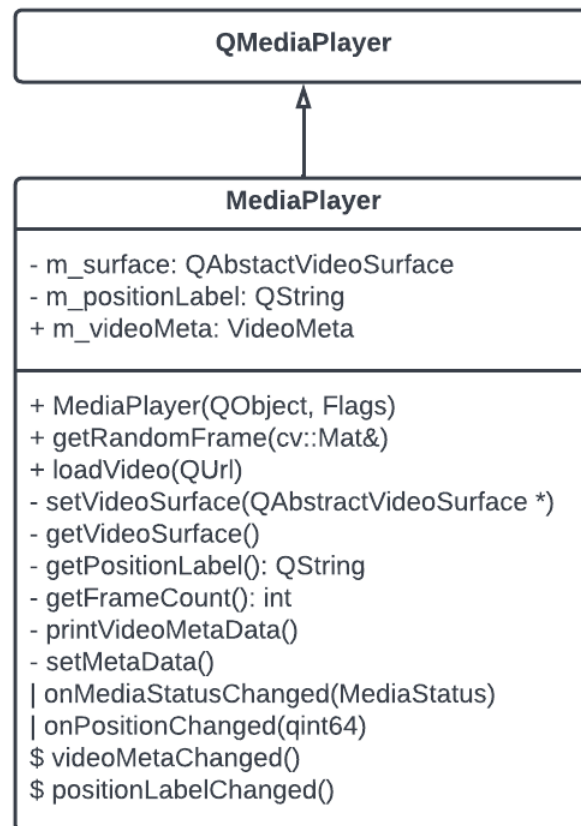
- + *public*
- × *public slots*
- *private*
- | *private slots*
- \$ *signals*

Az osztályok a következő csoportokba vannak sorolva:

- **Media**
  - MediaPlayer
  - VideoFilter
  - VideoFilterRunnable
- **Modules**
  - CameraCalibration
  - GateModel
  - StatModel
  - Tracker
  - VehicleModel
- **Tools**
  - Helpers
  - HungarianAlgorithm
  - KalmanSmoother
- **Types**
  - Detection
  - Gate
  - Vehicle
  - VideoMeta
- **Workers**
  - TrackerWorker



## MediaPlayer



Ez az osztály felelős a főablakban megjelenő videólejátszó megvalósításáért. Az osztály a Qt-ban található **QMediaPlayer**ből származik.

A video betöltése a `loadVideo()` függvénnyel történik melynek meghívása a QML-ből történik. Az `onMediaStatusChanged()` és az `onPositionChanged()` private slotok össze vannak kapcsolva (konstruktorban) az ősoosztály `mediaStatusChanged()` és `positionChanged()` signaljaihoz, így sikeres betöltés után a `onMediaStatusChanged()` elmenti a videófájl metaadatait (fájlnév, felbontás, hossz, stb.) az `m_videoMeta` változóban. A videó lejátszása során az `onPositionChanged()` slot pedig frissíti az `m_positionLabel` változóban tárolt, az aktuális lejátszási pozíciót mutató címkét.

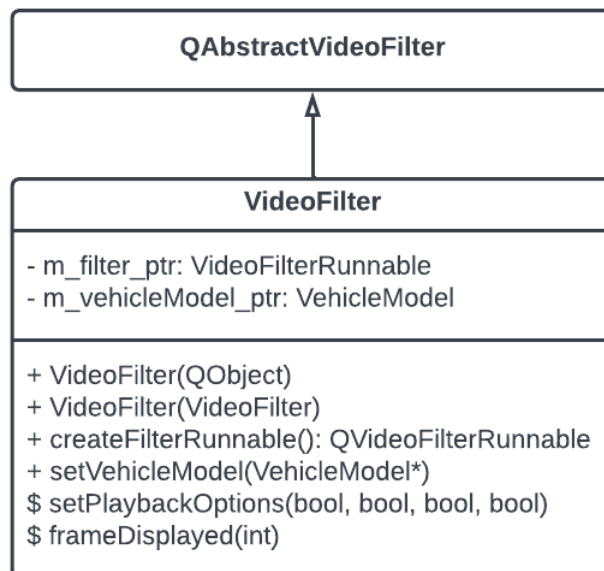
Az osztály 3 *property*-t valósít meg a QML felé:

```
1. Q_PROPERTY(QAbstractVideoSurface * videoSurface READ getVideoSurface WRITE
   setVideoSurface)
2.
3. Q_PROPERTY(QString positionLabel READ getPositionLabel NOTIFY positionLabelChanged)
4. Q_PROPERTY(int frameCount READ getFrameCount NOTIFY videoMetaChanged)
```

A legelső `videoSurface` *property* a videó megjelenítéséhez szükséges egy `VideoOutput` QML objektumon belül. A QML ennek az értékét a `getVideoSurface` és `setVideoSurface` függvényekkel olvassa és írja.

A másik két *property* (`positionLabel` és `frameCount`) csak olvasható, és olvasásuk a `getPositionLabel` és `getFrameCount` nevű privát függvényekkel történik. Amikor ezen értékek változnak, a QML a `positionChanged` és `videoMetaChanged` *signal*on keresztül értesül a változásokról, és frissíti a UI-ban az értékeket.

## VideoFilter



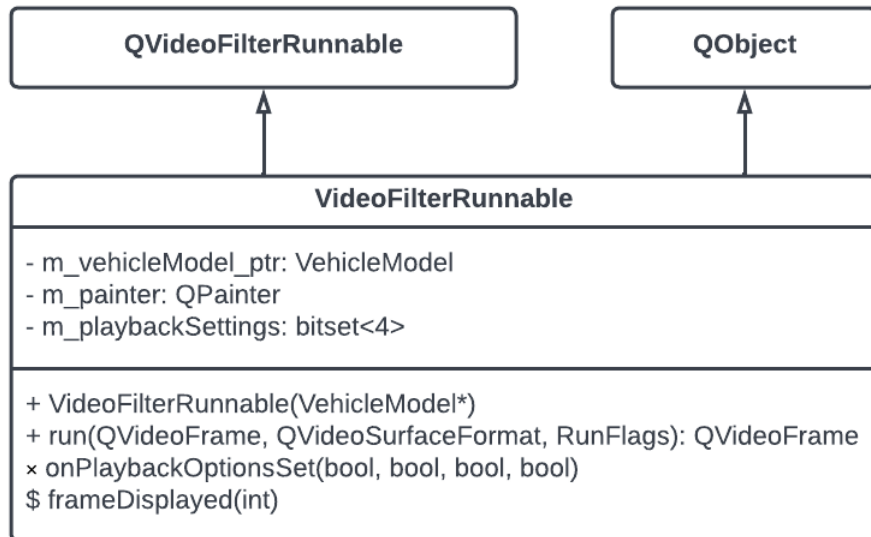
Ez az osztály a `QAbstractVideoFilter` osztályból származik, és segítségével a lejátszott videófolyam utófeldolgozása valósítható meg, vagyis ennek segítségével tudjuk a visszajátszott videóra a járművek helyzetét, útvonalát stb. „rárajzolni”.

A `createFilterRunnable()` függvény egy, a szülőosztályban található virtuális függvény megvalósítása, amely létrehoz egy `QVideoFilterRunnable` típusú szűrő objektumot, a pointerét elmenti az `m_filter_ptr` változóban, majd visszatér az objektum pointerével. Ez a szűrő objektum fogja a tényleges rajzolást elvégezni (lásd lentebb).

Az osztálynak ezenkívül van még két *signal*-ja (`setPlaybackOptions()` és `frameDisplayed()`), amelyeket az előbb említett `createFilterRunnable()` függvény kapcsol össze a létre szűrő megfelelő *slot*-jaival.

A `setVehicleModel()` függvény pedig egy `VehicleModel` objektum pointerét menti el az `m_vehicleModel_ptr` változóban, ami a szűrő létrehozásakor a konstruktorának van átadva.

## VideoFilterRunnable



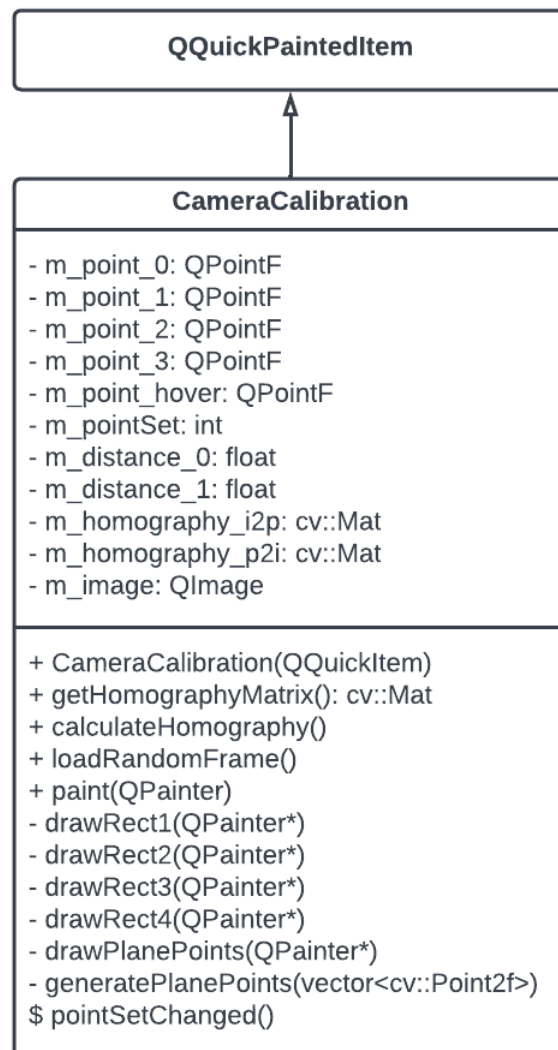
A `VideoFilterRunnable` az előbbi `VideoFilter` által létrehozott szűrőt megvalósító osztály. Egyrészt a `QVideoFilterRunnable` osztályból származik, amelynek funkciói a képkockák utófeldolgozását teszik lehetővé. Ezen túl a `QObject`-ból való származtatásra pedig a *Qt Framework* már említett *signal-slot* mechanizmusának megvalósítása miatt van szükség.

Az `onPlaybackOptionsSet` publikus *slot*-on keresztül lehet ki- és bekapcsolni a megjeleníteni kívánt tartalmakat, melyek elmentésre kerülnek az `m_playbackSettings` változóban.

A konstruktornak átadott `VehicleModel` pointer elmentésre kerül az `m_vehicleModel_ptr` változóban, majd ezt felhasználva a minden egyes képkockára lefutó `run()` függvény a paraméterként megkapott képkockára rárajzolja a megjeleníteni kívánt – az `m_playbackSettings` változó által engedélyezett – jármű információt.

A `run()` függvény minden egyes lefutásakor elküldi a `frameDisplayed()` *signal*-t az aktuális képkocka indexével, amelyet `VideoFilter`-en keresztül a `GateModel` objektum kap meg.

## CameraCalibration



Ez az osztály felelős a kamera kalibrációjához szükséges homográfia kiszámításáért. A **QQuickPaintedItem** osztályból származik, amire azért van szükség, mert a kalibráció beállítása során szeretnénk egy random képkockára rajzolni.

Az ablak betöltődésekor **loadRandomFarne()** függvény a videóból véletlenszerűen kiválaszt egy képkockát, és elmenti azt az **m\_image** változóban. Az ősoosztályból származó, és itt felülbíralt **paint()** függvény legelső dolga ennek a képnek a kirajzolása.

Az osztály a következő *property*-ket valósítja meg a QML felé:

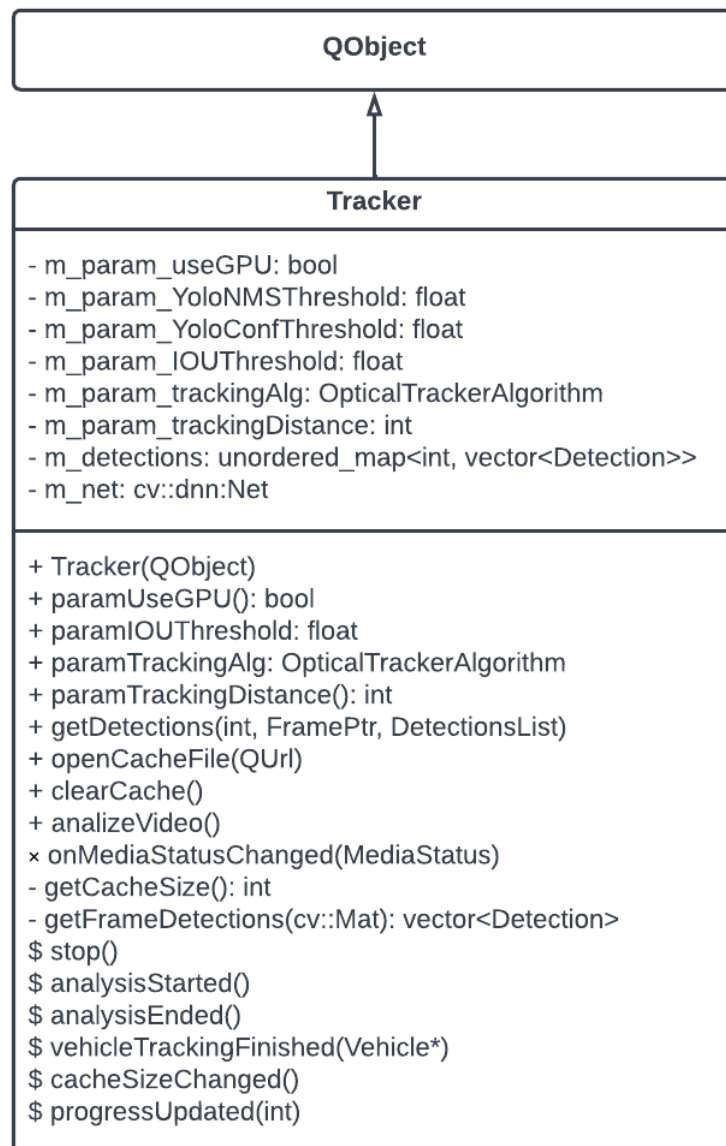
```
1. Q_PROPERTY(QPoint point_0 MEMBER m_point_0)
2. Q_PROPERTY(QPoint point_1 MEMBER m_point_1)
3. Q_PROPERTY(QPoint point_2 MEMBER m_point_2)
4. Q_PROPERTY(QPoint point_3 MEMBER m_point_3)
5. Q_PROPERTY(QPoint point_hover MEMBER m_point_hover)
6. Q_PROPERTY(int pointSet MEMBER m_pointSet NOTIFY pointSetChanged)
7. Q_PROPERTY(float distance_0 MEMBER m_distance_0)
8. Q_PROPERTY(float distance_1 MEMBER m_distance_1)
```

Mindegyik *property* a **MEMBER** kulcsszóval hozzá van kapcsolva az egyik privát változóhoz, az értékük beállítása a QML felől közvetlenül a változóba írással történik.

A kalibrációs téglalap értékeinek beállítása után az ablakban az **Apply** gombra kattintva meghívódik a `calculateHomography()` függvény, ami kiszámolja a homogén mátrixot, majd meghívja az őosztály `update()` függvényét, aminek hatására „újra festi” az ablak tartalmát, azaz lefut a `paint()` függvény. Az `update()` amúgy a kalibrációs téglalap beállítása alatt is rendszeresen meghívódik.

Az osztályban található privát függvények mindegyike *inline* függvény, és a `paint()` függvény részfeladatait valósítják meg.

## Tracker



Ez az osztály felelős az járművek felismeréséért és azok követéséért. A QObject ősosztályból származik, amire a Qt *signal-slot* mechanizmusának használata miatt van szükség.

Az osztály a következő *property*-ket valósítja meg a QML felé:

```
1. Q_PROPERTY(int cacheSize READ getCacheSize NOTIFY cacheSizeChanged)
2.
3. Q_PROPERTY(bool useGPU MEMBER m_param_useGPU)
4. Q_PROPERTY(float YoloNMSThreshold MEMBER m_param_YoloNMSThreshold)
5. Q_PROPERTY(float YoloConfThreshold MEMBER m_param_YoloConfThreshold)
6. Q_PROPERTY(float IOUThreshold MEMBER m_param_IOUThreshold)
7. Q_PROPERTY(TrackerAlg trackerAlgorithm MEMBER m_param_trackingAlg)
8. Q_PROPERTY(int trackingDistance MEMBER m_param_trackingDistance)
```

A `cacheSize` *property* csak olvasható, és olvasása a `getCacheSize()` függvénnyel történik. Az érték megváltozása esetén a `cacheSizeChanged()` *signal* küld jelzést a QML-nek az érték UI-ben történő frissítésére. A többi *property* MEMBER-ként van összekapcsolva az osztály megfelelő privát változóival. Az értékük beállítása a QML-ből közvetlenül történik. Ezek a *property*-k tárolják az algoritmus különféle paramétereit.

Az `openCacheFile()` függvény a paraméterként átadott cache fájlból betölti annak tartalmát az `m_detections` változóba, míg a `clearCache()` függvény pedig törli ennek a változónak a tartalmát.

A betöltött videó elemzése az `analyzeVideo()` függvény meghívásával történik. A függvény létrehoz egy külön szálon futó objektumot a **TrackerWorker** osztályból, összekapcsolja a megfelelő *signal*-okat és *slot*-okat, majd elindítja ennek a szálnak a futását.

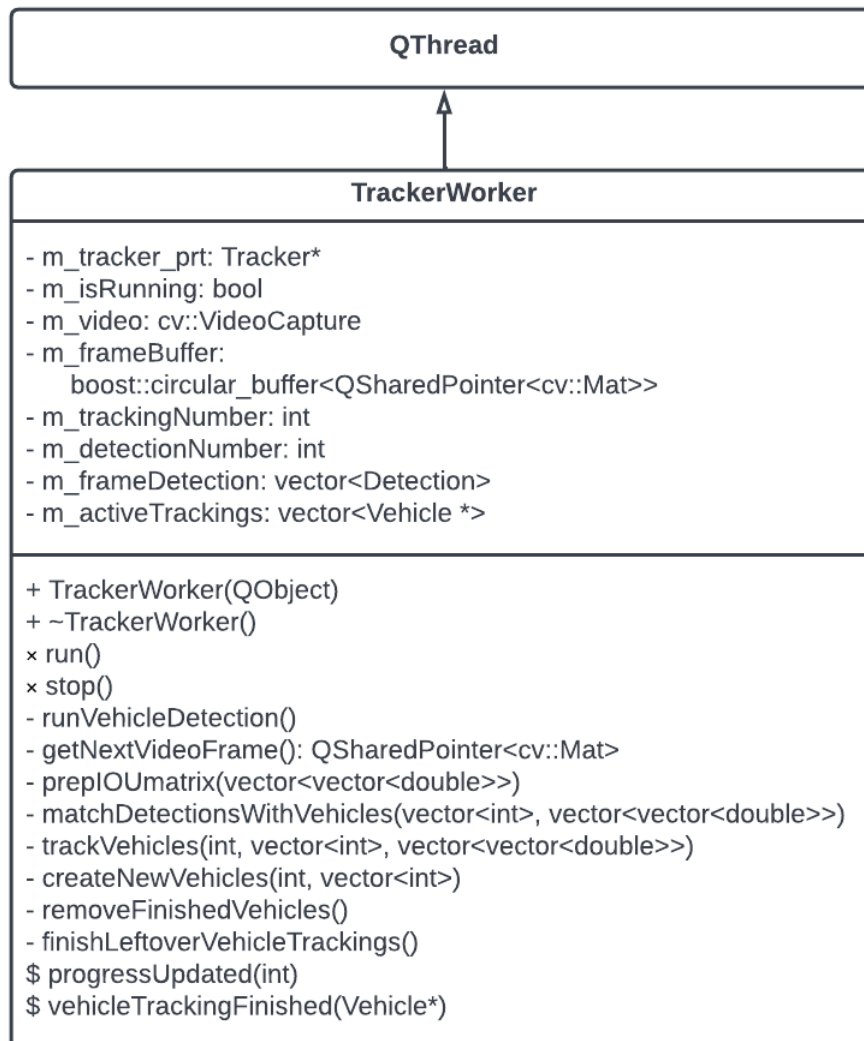
Szignálok:

- `stop()`: A felhasználtól érkező, az elemzés megszakítására vonatkozó kérelmet továbbítja a végrehajtó szál felé.
- `analysisStarted()`: Az `analyzeVideo()` függvényből van útjára indítva, és a többi modul felé van továbbítva. Hatására a többi feldolgozó modul alaphelyzetbe állítja magát.
- `analysisEnded()`: A feldolgozó szál megszűnésének eseményét továbbítja a **VehicleModel** modul felé, aminek hatására az elkezd a járművek utófeldolgozását.
- `vehicleTrackingFinished()`: A feldolgozó szál ezen a szignálon keresztül továbbítja a jármű objektumokat a **VehicleModel** modul felé a feldolgozás során.
- `cacheSizeChanged()`: A már említett **cacheSize** QML *property* frissítését jelző szignál.
- `progressUpdated()`: A feldolgozottság pillanatnyi állapotát továbbítja a UI felé.

Slot-ok:

- `onMediaStatusChanged()`: A **MediaPlayer**-ből érkező `mediaStatusChanged()` szignál van rákötve. Amennyiben videó betöltésére kerül sor, akkor kiüríti a detektálásokat tartalmazó *cache*-t (`m_detections` változó).

## TrackerWorker



A járművek tényleges detektálását végző osztály. Mivel az algoritmus számításigényes, így a főszálon futtatva a GUI-t megakaszthatja (vagyis a program nem válaszolna a felhasználói interakciókra), így egy külön szálon kerül futtatásra. Ennek a külön szálon futtatásnak a feladatát valósítja meg a QThread ősosztály.

Két publikus *slot*-ja van:

- **run()**: Ez futtatja magát algoritmust, amelynek végrehajtása az ősosztály **start()** metódusának a **Tracker** objektumból történő meghívására indul.
- **stop()**: Megállítja az algoritmus futását. Az algoritmus leállításával pedig felszabadítja a szálát.

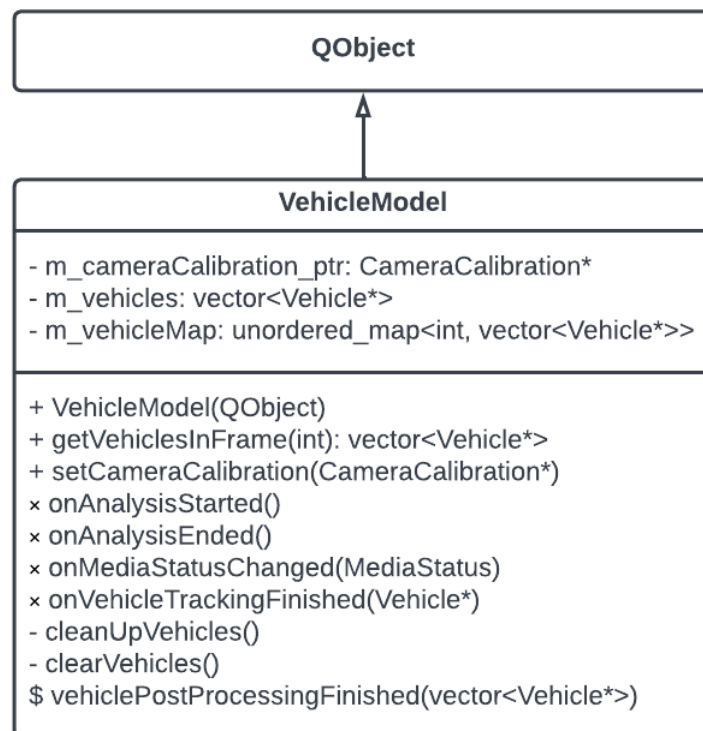


Szignálok:

- `progressUpdated()`: A **Tracker** objektum azonos nevű szignálján keresztül frissíti a GUI-t.
- `vehicleTrackingFinished()`: A **Tracker** objektum azonos nevű szignálján keresztül elküldi a detektált járművet reprezentáló objektumot a **VehicleModel** modul felé.

Az *inline* privát függvények mindegyike pedig az algoritmus különböző részfeladatait valósítják meg.

## VehicleModel



Ez az osztály felelős a járművek utófeldolgozásáért, és egyben ez az objektum tárolja a hivatkozásokat a járműobjektumokra. A Qt *signal-slot* mechanizmusának használata érdekében az osztály a **QObject** osztályból származik.

#### Publikus függvények:

- `getVehiclesInFrame(int)`: A visszaadja egy adott képkockán látható járművek listáját.
- `setCameraCalibrationInFrame(CameraCalibration*)`: Elmenti a paraméterként kapott **CameraCalibration** típusú pointert az `m_cameraCalibration_ptr` változóba.

#### Publikus *slot*-ok:

- `onAnalysisStarted()`: A videófeldolgozás kezdetén törli az esetlegesen tárolt járműobjektumokat.
- `onAnalysisEnded()`: Az összes jármű felismerése után az utófeldolgozást végző függvény.
- `onMediaStatusChanged(MediaStatus)`: Videó betöltése esetén törli az összes tárolt járműobjektumot.
- `onVehicleTrackingFinished(Vehicle*)`: Ezen a *slot*-on keresztül érkeznek a felismert járműobjektumok a **Tracker** futása alatt. A kapott objektum pointereket az `m_vehicles` változóban tárolja.

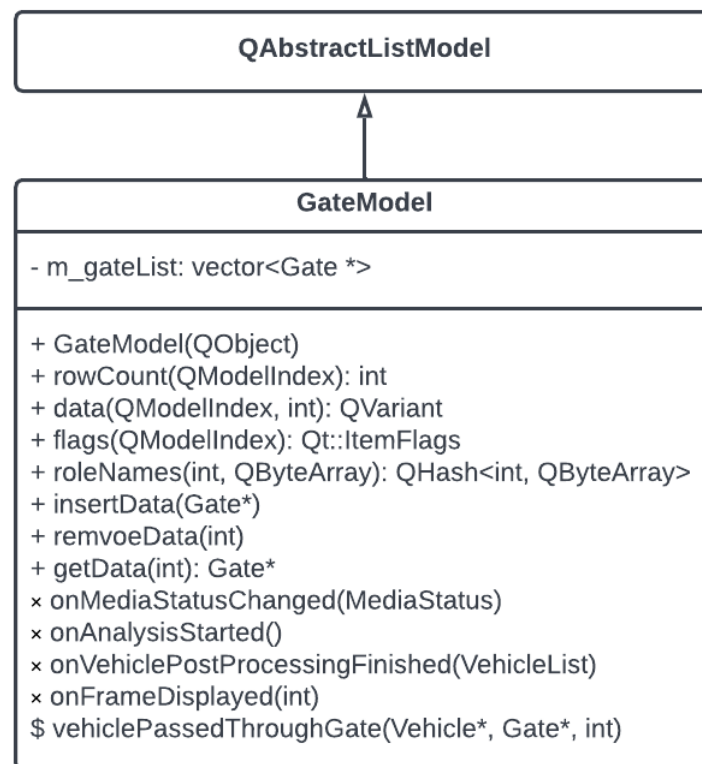
#### Privát függvények:

- `cleanUpVehicles()`: Törli a tárolt járműveket.
- `clearVehicles()`: Egy utófeldolgozási lépés. Törli azon járműveket amelyeknek az útvonala túl rövid.

#### Szignálok:

- `vehiclePostProcessingFinished(std::vector<Vehicle *>)`: Az utófeldolgozási lépés végén továbbítja a járműobjektumok pointereinek listáját a **GateModel** modulnak.

## GateModel



A virtuális forgalomszámláló kapukat kezelő osztály. A Qt-beli **QAbstractListModel** osztályból származik, ami egy listát valósít meg.

Publikus függvények:

- **GateModel(QObject\*)**: Konstruktor.
- **rowCount(QModelIndex)**: A szülőosztály virtuális függvénye. Visszaadja a listában található elemek (ez esetben kapuk) számát.
- **data(QModelIndex, int)**: A szülőosztály virtuális függvénye. Egy konkrét elemet ad vissza a listából.
- **flags(QModelIndex)**: A szülőosztály virtuális függvénye. Egy adott listaelem „állapotát” adja vissza. Esetünkben nem használt. (Minden elemre az `Qt::ItemIsEditable` állapottal tér vissza.)
- **roleNames(int, QByteArray)**: A szülőosztály virtuális függvénye. Az elemeknek több tulajdonsága (role) is lehet, ezek neveivel tér vissza.
- **insertData(Gate\*)**: Beilleszt egy kaput a listába.

- `removeData(int)`: Eltávolít egy kaput a listából.
- `getData(int)`: Egy adott kapuval tér vissza. A `data(...)` függvénynek egy egyszerűsített változata.

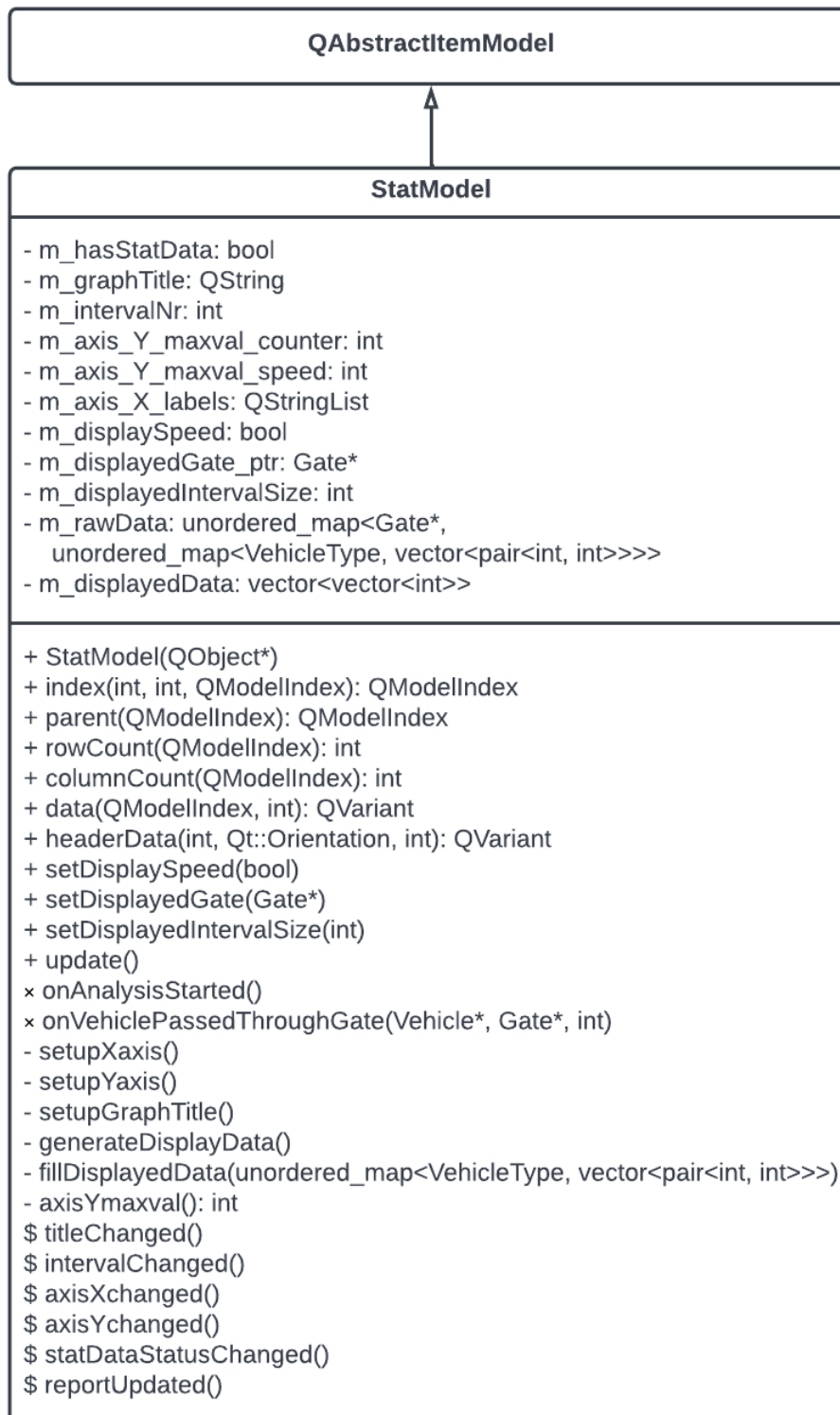
Publikus *slot*-ok:

- `onMediaStatusChanged(MediaStatus)`: Egy videó betöltésekor törli az összes kaput.
- `onAnalysisStarted()`: A feldolgozás kezdetekor inicializálja a kapukat. (Törli a korábbi regisztrált járműáthaladásokat, és nullára állítja a számlálókat.)
- `onVehiclePostProcessingFinished(VehicleList)`: Utófeldolgozási lépés. A `VehicleModel` objektumtól kapott járműlista minden elemén ellenőrzi, hogy azok áthaladtak-e valamelyik kapun.
- `onFrameDisplayed(int)`: Videó lejátszása közben folyamatosan frissíti a megjelenített számláló értékét.

Szignálok:

- `vehiclePassedThroughGate(Vehicle *, Gate *, int)`: Az utófeldolgozás során, amennyiben egy jármű áthaladását érzékelte egy adott kapun, akkor a jármű- és a kapu azonosítóját, valamint az időpontot (képkocka sorszáma) ezen a szignálon keresztül továbbítja a **StatModel** objektumnak.

## StatModel



Az utófeldolgozás legutolsó lépését, a forgalmi statisztikákat megvalósító osztály. A Qt-beli **QAbstractListModel** osztályból származik, ami egy táblázatot valósít meg. A modell a QML-

ben hozzá van kapcsolva több oszlopgrafikont megvalósító objektumhoz, amik a táblázat tartalmát vizuálisan megjelenítik.

Az osztály a következő *property*-ket valósítja meg a QML felé:

```
1. Q_PROPERTY(QString graphTitle MEMBER m_graphTitle NOTIFY titleChanged)
2. Q_PROPERTY(int intervalNr MEMBER m_intervalNr NOTIFY intervalChanged)
3. Q_PROPERTY(QStringList axis_X_labels MEMBER m_axisX_labels NOTIFY axisXchanged)
4. Q_PROPERTY(int axis_Y_maxval READ axisYmaxval NOTIFY axisYchanged)
5. Q_PROPERTY(bool hasStatData MEMBER m_hasStatData NOTIFY statDataStatusChanged)
```

Ezen *property*-k mindegyike **MEMBER**-ként közvetlenül egy privát adattaghoz van kapcsolva.

Az osztály szignáljai ezen értékek frissüléséről értesítik a QML-t.

Publikus függvények:

- `StatModel(QObject *)`: Konstruktork.
- `index(int, int, QModelIndex)`: A szülőosztály virtuális függvénye. A paraméterként kapott sor- és oszlopindexeket reprezentáló **QModelIndex** objektummal kell visszatérnie.
- `parent(QModelIndex)`: A szülőosztály virtuális függvénye. Esetünkben nem használt.
- `rowCount(QModelIndex)`: A szülőosztály virtuális függvénye. A táblázatban szereplő sorok számát adja vissza.
- `columnCount(QModelIndex)`: A szülőosztály virtuális függvénye. A táblázatban szereplő oszlopok számát adja vissza.
- `data(QModelIndex, int)`: A szülőosztály virtuális függvénye. A táblázat egy adott cellájának lekérdezésére való.
- `headerData(int, Qt::Orientation, int)`: A szülőosztály virtuális függvénye. A táblázat fejlécének a neveit (oszlopok neveit) adja meg.
- `setDisplaySpeed(bool)`: A paraméterként kapott *boolean* paraméter alapján beállítja, hogy a táblázat a sebességeket vagy a járművek számát reprezentálja.
- `setDisplayGate(Gate *)`: A paraméterként kapott Gate pointer alapján beállítja, hogy a táblázat mely kapura vonatkozó adatokat reprezentálja. Ha `nullptr` a paraméter, akkor az összesített adatokat fogja mutatni.
- `setDisplayIntervalSize(int)`: Az időintervallumok méretét lehet vele beállítani.

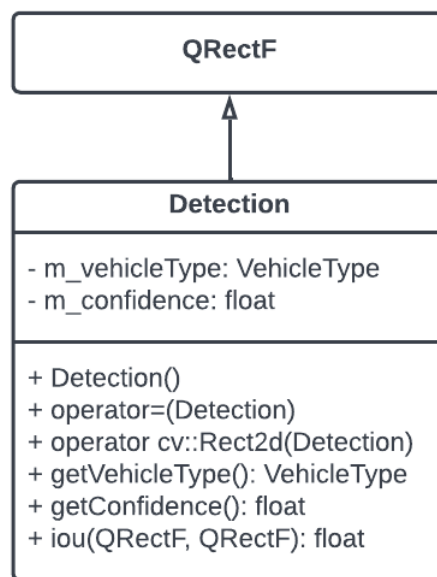
- `update()`: Frissíti a táblázatot, azaz legenerálja, hogy a fentebbi (`setXXX`) függvények által beállított értékek alapján milyen adatokat jelenítsen meg a grafikonban.

Publikus *slot*-ok:

- `onAnalysisStarted()`: A videófeldolgozás kezdetén törli a tárolt adatokat.
- `onVehiclePassedThroughGate(Vehicle *, Gate *, int)`: Amikor a **GateModel** egy jármű adott kapun történő áthaladását detektálja, akkor egy szignállal elküldi a **StatModel** objektumnak, ami ezen a *slot*-on keresztül kapja meg, majd elmenti az eseményt az `m_rawData` változóba.

A privát függvények *inline* függvények, és egy-egy részfeladatot valósítanak meg.

## Detection

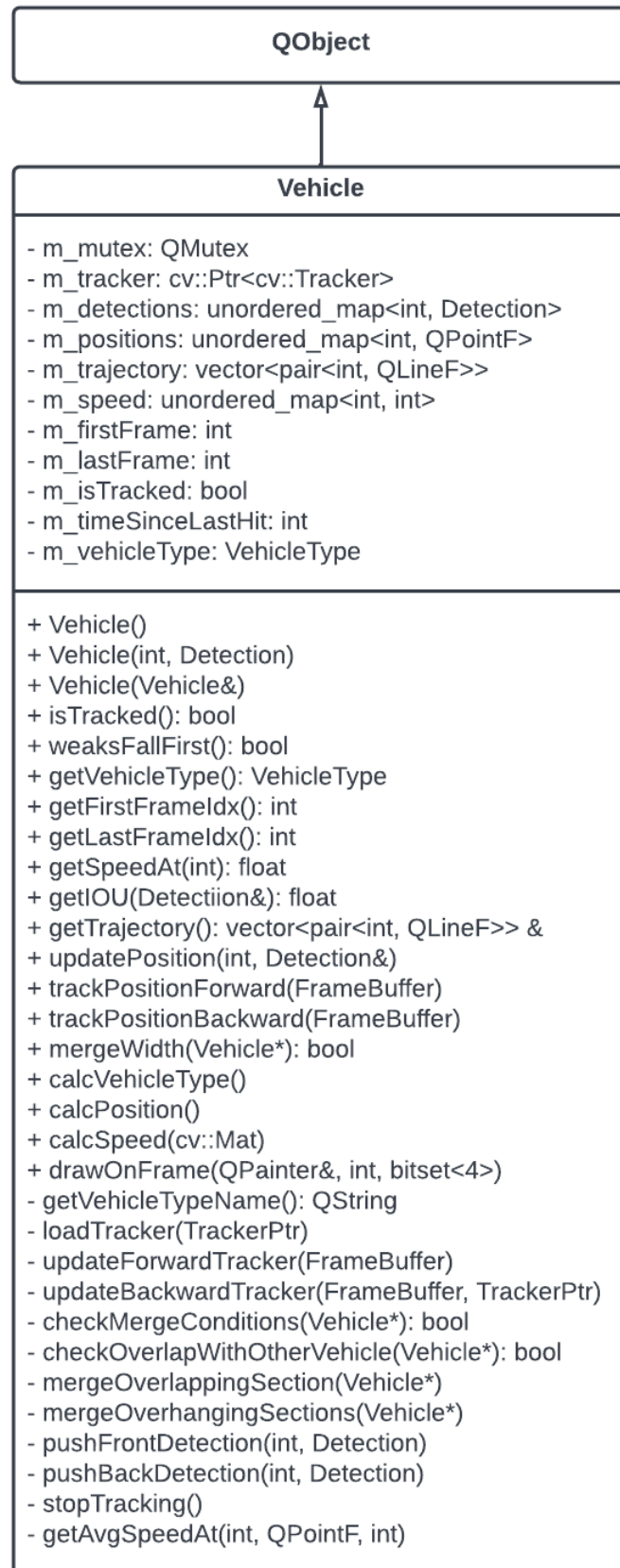


Egy adott YOLO detektálás reprezentálására szolgál. Mivel alapjában véve *bounding box*-okról van, így a Qt-beli **QRectF** őssosztályból van származtatva.

A *bounding box* pozíciójára (**x**, **y**) és méretére (**width**, **height**) vonatkozó adatokat maga **QRectF** őssosztály tárolja. Ezen kívül szükségünk van még jármű típusának (`m_vehicleType`) és a YOLO által adott valószínűségi értéknek (`m_confidence`) a tárolására, amiket rendre a `getVehicleType()` és `getConfidence()` függvényekkel kérhetjük le.

Az `iou()` statikus fv. pedig a két paraméterül kapott detektálás IOU értékét határozza meg.

## Vehicle





Egy adott járművet reprezentáló osztály. A `QObject` őssztályból származik, amire az osztályon belül definiált *enum class* `QML`-be való injektálása miatt van szükség.

Publikus függvények:

- `Vehicle()`: Konstruktor.
- `Vehicle(int, Detection)`: Konstruktor.
- `Vehicle(Vehicle)`: Konstruktor.
- `isTracked()`: Megadja, hogy a jármű a feldolgozás folyamán egy követett jármű-e.
- `weaksFallFirst()`: Megadja, hogy a jármű útvonala rövidebb-e (képkockák számában), mint egy beállított érték.
- `getVehicleType()`: Megadja a jármű típusát.
- `getFirstFrameIdx()`: Megadja a legelső képkockának az indexét, amin a jármű detektálva lett.
- `getLastFrameIdx()`: Megadja a legutolsó képkockának az indexét, amin a jármű detektálva lett.
- `getSpeedAt(int)`: Megadja a jármű sebességét egy adott képkockán.
- `getIOU(Detection)`: IOU értéket számol a paraméterként kapott detektálás, és a jármű utolsó pozíciója között.
- `getTrajectory()`: Visszaadja a jármű útvonalát egy vektorban.
- `updatePosition(int, Detection)`: Egy adott időpillanatra (képkocka index) megad egy detektálást, és ezzel együtt beállítja az `m_lastFrame` értékét erre az indexre.
- `trackPositionForward(FrameBuffer)`: Optikai trackerrel megpróbálja követni a jármű pozícióját az utolsó detektálástól.
- `trackPositionBackwards(FrameBuffer)`: Optikai trackerrel megpróbálja visszafelé követni a jármű útvonalát a legelső detektálástól.
- `mergeWith(Vehicle *)`: A paraméterként kapott másik járműobjektum útvonalát megpróbálja hozzáfűzni a saját útvonalához, amennyiben a másik objektum megfelelő.
- `calcVehicleType()`: A különálló detektálásokhoz tartozó járműtípus adatokból meghatározza a jármű végleges (tényleges) típusát.

- `calcPosition()`: A detektálások sorozatát – illetve azok középpontjait – egy RTS-simító algoritmuson átengedve kiszámolja a jármű tényleges pozícióit minden egyes képkockára.
- `calcSpeed(cv::Mat)`: A paraméterül kapott homogén mátrix segítségével kiszámolja a jármű sebességét minden egyes képkockára.
- `drawOnFrame(QPainter, int, bitset<4>)`: A jármű detektálásának (*bounding box*ának), útvonalának és címkéinek a videó képkockákra rajzolásáért végzi el.

A privát függvények az egyes részfeladatokat megvalósító *inline* függvények. Kivéve a `getVehicleTypeName()` függvényt, amely a jármű típusát adja meg szöveges formában.




## Tesztek

Mivel a feladat jellegéből adódóan a unit tesztelés kis híján lehetetlen, így egyfajta *black box* tesztelés mellett döntöttem.



### 1. Aktív/inaktív gombok/menüpontok

Teszteset	Elvárt viselkedés	
Programindítás után, videó nincs betöltve	A videólejátszó gombjai inaktívak, a <b>GATES</b> panel <b>ADD GATE</b> és <b>DELETE GATE</b> gombjai inaktívak, a menüsor gombjai közül az <b>Open video...</b> gomb aktív, a többi inaktív, a <b>Clear cache</b> gomb inaktív.	✓
Videó betöltése	A videólejátszó gombjai aktívak, a <b>GATES</b> panel <b>ADD GATE</b> gombja aktív, a <b>DELETE GATE</b> gomb inaktív, a menüsor gombjai az <b>Open stat window...</b> gomb kivételével aktívak, a <b>Clear cache</b> gomb inaktív.	✓
Cache betöltése	A <b>Clear cache</b> gomb inaktívról aktívra vált.	✓
Cache törlése	A <b>Clear cache</b> gomb aktívról inaktívra vált.	✓
Kapu hozzáadása	Az <b>ADD GATE</b> gombra kattintva a gomb „kiválasztott” állapotra vált (piros szöveg és keret).	✓
Kapu hozzáadva	A <b>DELETE GATE</b> gomb aktiválódik.	✓
Utolsó kapu törlése	A <b>DELETE GATE</b> gomb inaktiválódik.	✓

## 2. Videó betöltése

Teszteset	Elvárt viselkedés	
<b>Valid videó betöltése</b>	Az <b>Open video...</b> gombra kattintva, a felugró párbeszédablakból egy érvényes videót kiválasztva, a videó betöltődik, az első képkocka megjelenik a vagány birka helyén, a <i>progress bar</i> mellett megjelenik a videó hossza, és az ablak láblécében megjelenik a képkockák száma.	
<b>Videó betöltése már tárolt adatok után</b>	Az új videó betöltődik, a tárolt adatok (cache, kapuk) törlődnek, az <b>Open stat window...</b> gomb inaktíválódik.	
<b>Invalid videó betöltése</b>	Az <b>Open video...</b> gombra kattintva, a felugró párbeszédablakból egy érvénytelen fájlt kiválasztva hibaüzenetet kapunk.	

## 3. Cache betöltése / törlése

Teszteset	Elvárt viselkedés	
<b>Cache betöltése</b>	Az <b>Open cache...</b> gombra kattintva, a felugró párbeszédablakból egy <i>cache</i> fájlt kiválasztva, a <i>cache</i> betöltődik, és a <i>cache</i> -ben található képkockák száma megjelenik az ablak láblécében.	 *
<b>Cache törlése</b>	A <b>Clear cache</b> gombra kattintva a láblécben a <i>cache</i> tartalma 0-ra vált.	

**\*Megjegyzés:** Némelyik videó esetén a cache- és a videó képkockáinak a száma eltérhet egymástól. Ez annak a következménye, hogy a cache exportálásakor végig iteráltam a videó összes képkockáján, míg a képkockák számának lekérdezésére az *OpenCV*-t használja a program. És az *OpenCV* (a forráskódja szerint) csak kiszámolja az eredményt a videó hosszából és FPS értékéből. Ez kerekítési hibákhoz vezethet, főleg a nem egész FPS értékek (pl. 29.97) esetén. Ettől függetlenül a program helyesen működik, mivel a videó feldolgozása során is képkockákon iterál végig, amíg el nem fogynak.





#### 4. Kapuk hozzáadása / eltávolítása

Teszteset	Elvárt viselkedés	
Kapu hozzáadása nem aktivált állapotban	Amikor az <b>ADD GATE</b> gomb nem aktív (nem piros), a kapuk megrajzolása nem lehetséges.	✓
Kapu hozzáadása aktivált állapotban	Amikor az <b>ADD GATE</b> gomb aktív (piros), a kaput egy <i>click-and-drag</i> módszerrel meg lehet rajzolni a videó fölött.	✓
Kapu nevének megadása	A kapu megrajzolása során az egérgomb felengedése után egy felugró párbeszédablakban a kapu neve megadható.	✓
Kapu elfogadása	A kapu nevének megadása után az <b>OK</b> gombra kattintva, a kapu hozzáadódik a listához. A számlálónak mind a listában, mind a kapu vizualizációján 0-t kell mutatnia.	✓
Kapu elutasítása	A kapu nevének megadása után a <b>Cancel</b> gombra kattintva a kapu törlődik.	✓
Kapu törlése	A listából egy kaput kiválasztva, majd a <b>DELETE GATE</b> gombra kattintva törlődik a vizualizációja és a bejegyzés a listából is.	✓

#### 5. Kamera kalibráció

Teszteset	Elvárt viselkedés	
Kalibrációs ablak megnyitása	A főmenü <b>Camera calibration...</b> menüpontjára kattintva megnyílik a kalibrációs ablak egy véletlen képkockával a videóból.	✓
Kalibrációs téglalap megadása nem aktivált állapotban	Az ablakban nem kezdődik el a téglalap kijelölése, vagy a kijelölt téglalap nem változik.	✓
Kalibrációs téglalap megadása aktivált állapotban	Az ablakban elkezdődik a téglalap kijelölése, azaz minden egyes kattintásra egy csúcsa rögzül a téglalapnak. A negyedik csúcs rögzítése után további kattintásra változás nem történik.	✓
Homográfia számítása	A téglalap megadása után az <b>Apply</b> gombra kattintva a téglalappal párhuzamos síkon sárga pontok jelennek meg egy négyzetrács mentén.	✓

## 6. Videó lejátszása

Teszteset	Elvárt viselkedés	
Videó lejátszása elindítása	A <b>Play</b> gombra kattintás után a felvétel lejátszása elindul, a pozíciót mutató címke elkezd felfelé számolni és a piros jelzőcsík elkezd megtölteni a <i>progress bar</i> -t. Ha az elemzés után indítjuk a lejátszást, akkor a videón megjelennek a <b>Playback Options</b> panelen található engedélyezett kapcsolóknak megfelelő nyomkövetési adatok.	
Visszajátszási opciók változtatása (elemzés után)	Lejátszás közben a <b>Playback Options</b> panel kapcsolóinak állítására a	
Némítás	A némítás gombra kattintva lejátszás közben a hang elhallgat.	
Lejátszási pozíció változtatása	A <i>progress bar</i> bármely pontjára kattintva a lejátszási pozíció az adott helyre ugrik.	

## További fejlesztési lehetőségek

A program jelenlegi állapotában (v1.0.0) még rengeteg fejlesztési lehetőséget rejt magában. Itt van néhány ötlet, amik több-kevesebb időráfordítással megoldhatók:

- A programhoz mellékelt YOLO modell csak az ugyancsak a programhoz mellékelt videókra lett betanítva. Ennek következtében a modell elég rendesen az *overfitting* nevű betegség jeleit mutatja, de a célom nem is egy általánosan használható program elkészítése volt, hanem csak egyfajta „*proof-of-concept*” létrehozása. Vagyis a YOLO modell lecserélhető egy nagy adathalmazon tanított, általánosan használható modellre. Vagy különféle helyzetekre (pl. drónfelvétel, nappali/éjszakai felvétel, stb.) különböző modellek alkalmazása is elképzelhető.
- Ha drónfelvételeket is támogatni szeretnénk, akkor szembe kell néznünk azzal a problémával, hogy drónfelvételek nem stabil felvételek. Vagyis mind a kapuk, mind a járművek pozícióját kompenzálni kell a kamera mozgásának mértékével.
- A program megvalósítható szerver-kliens architektúrában is, ahol a járművek felismerése és követése a szerveren történik. Ekkor akár egy web applikációval is elérhető a szerver, ami egyrészt egy nagyon egyszerű multiplatform megvalósítást eredményezne, másrészt egy erős szerver akár több kliens által feltöltött feladatot is fel tud dolgozni, megfelelő ütemezés mellett.
- A Kálmán szűrő pontosabb működése érdekében érdemes lehet implementálni a fentebb említett,  $Q$  és  $R$  mátrixok beállításához való adaptív módszert. [24]

## Zárószó és köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani az ELTE Informatikai kar összes oktatójának, akik végig rugdostak a tanulmányaim során. Ezentúl szeretném megköszönni Csetverikov Dmitrij tanár úrnak, aki első körben elvállalta a szakdolgozatom mentorálását, illetve külön köszönet illeti Hajder Levente tanár urat, aki nemcsak átvette a mentorálást, de az általa oktatott tantárgyakon (*GPGPU, Számítógépes látás, stb.*) szerzett ismeretek nemcsak hogy nagyban hozzájárultak a szakdolgozatom elkészítéséhez, de fel is keltették az érdeklődésemet a számítógépes látás és az HPC (High Performance Computing) világa iránt. Még egyszer köszönöm mindenkinek!

# Irodalomjegyzék

- [1] S. Mallick, „Learn OpenCV,” 2017. 02. 17. 02 2017. [Online]. Available: <https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/>.
- [2] T. W. A. I. K. Y. S. Malik M. Khan, „Semantic Scholar,” 2014-10 10 2014. [Online]. Available: <https://pdfs.semanticscholar.org/a700/d1aa990b001654d9911a7365a71bf7df351f.pdf>.
- [3] T. S. T. S. Erik Bochinski, „Technische Universität Berlin,” [Online]. Available: <http://elvera.nue.tu-berlin.de/files/1547Bochinski2018.pdf>.
- [4] J. D. T. D. J. M. Ross Girshick, „Rich feature hierarchies for accurate object detection and semantic segmentation,” [Online]. Available: [https://www.cv-foundation.org/openaccess/content\\_cvpr\\_2014/papers/Girshick\\_Rich\\_Feature\\_Hierarchies\\_2014\\_CVPR\\_paper.pdf](https://www.cv-foundation.org/openaccess/content_cvpr_2014/papers/Girshick_Rich_Feature_Hierarchies_2014_CVPR_paper.pdf).
- [5] R. Girshick, „Fast R-CNN,” [Online]. Available: [https://www.cv-foundation.org/openaccess/content\\_iccv\\_2015/papers/Girshick\\_Fast\\_R-CNN\\_ICCV\\_2015\\_paper.pdf](https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Girshick_Fast_R-CNN_ICCV_2015_paper.pdf).
- [6] K. H. R. G. J. S. Shaoqing Ren, „Faster R-CNN: Towards Real-Time ObjectDetection with Region Proposal Networks,” [Online]. Available: <https://arxiv.org/pdf/1506.01497.pdf>.
- [7] K. C. J. S. H. F. W. O. D. L. Jiangmiao Pang, „Libra R-CNN: Towards Balanced Learning for Object Detection,” [Online]. Available: <https://arxiv.org/pdf/1904.02701v1.pdf>.
- [8] D. A. D. E. C. S. S. R. C.-Y. F. A. C. B. Wei Liu, „SSD: Single Shot MultiBox Detector,” [Online]. Available: <https://arxiv.org/pdf/1512.02325.pdf>.
- [9] S. D. R. G. A. F. Joseph Redmon, „You Only Look Once:Unified, Real-Time Object Detection,” [Online]. Available: <https://arxiv.org/pdf/1506.02640.pdf>.
- [1 A. F. Joseph Redmon, „YOLO9000: Better, Faster, Stronger,” [Online]. Available: 0] <https://arxiv.org/pdf/1612.08242.pdf>.
- [1 A. F. Joseph Redmon, „YOLOv3: An Incremental Improvement,” [Online]. Available: 1] <https://arxiv.org/pdf/1804.02767.pdf>.
- [1 C.-Y. W. H.-Y. M. L. Alexey Bochkovskiy, „YOLOv4: Optimal Speed and Accuracy of Object 2] Detection,” [Online]. Available: <https://arxiv.org/pdf/2004.10934.pdf>.
- [1 L. W. X. B. Z. L. S. Z. L. Shifeng Zhang, „Single-Shot Refinement Neural Network for Object 3] Detection,” [Online]. Available: <https://arxiv.org/pdf/1711.06897.pdf>.
- [1 P. G. R. G. K. H. P. D. Tsung-Yi Lin, „Focal Loss for Dense Object Detection,” [Online].



- 4] Available: <https://arxiv.org/pdf/1708.02002.pdf>.
- [1 H.-Y. M. L. I.-H. Y. Y.-H. W. P.-Y. C. J.-W. H. Chien-Yao Wang, „CSPNet: A New Backbone  
5] that can Enhance Learning Capability of CNN,” [Online]. Available:  
<https://arxiv.org/pdf/1911.11929.pdf>.
- [1 X. Z. S. R. J. S. Kaiming He, „Spatial Pyramid Pooling in Deep Convolutional Networks for  
6] Visual Recognition,” 2015-04-23 04 2015. [Online]. Available:  
<https://arxiv.org/pdf/1406.4729.pdf>.
- [1 L. Q. H. Q. J. S. J. J. Shu Liu, „Path Aggregation Network for Instance Segmentation,”  
7] 2018-09-18 09 2018. [Online]. Available: <https://arxiv.org/pdf/1803.01534.pdf>.
- [1 Z. L. L. v. d. M. K. Q. W. Gao Huang, „Densely Connected Convolutional Networks,” 2018-  
8] 01-28 01 2018. [Online]. Available: <https://arxiv.org/pdf/1608.06993.pdf>.
- [1 P. D. R. G. K. H. B. H. S. B. Tsung-Yi Lin, „Feature Pyramid Networks for Object Detection,”  
9] 2017-04-19 04 2017. [Online]. Available: <https://arxiv.org/pdf/1612.03144.pdf>.
- [2 J. W. Zhanchao Huang, „DC-SPP-YOLO: Dense Connection and Spatial Pyramid Pooling  
0] Based YOLO for Object Detection,” [Online].
- [2 J. P. J.-Y. L. S. K. Sanghyun Woo, „CBAM: Convolutional Block Attention Module,” 2018-  
1] 07-18 07 2018. [Online].
- [2 H. W. Kuhn, „The Hungarian Method for the assignment problem,” *Naval Research*  
2] *Logistics Quarterly*, p. 83–97, 1955.
- [2 „Product of Two Gaussian PDFs,” Stanford University, [Online]. Available:  
3] [https://ccrma.stanford.edu/~jos/sasp/Product\\_Two\\_Gaussian\\_PDFs.html](https://ccrma.stanford.edu/~jos/sasp/Product_Two_Gaussian_PDFs.html).
- [2 N. Z. Z. H. Shahrokh Akhlaghi, „Adaptive Adjustment of Noise Covariance in Kalman Filter  
4] for Dynamic State Estimation,” 2018. [Online]. Available:  
<https://arxiv.org/ftp/arxiv/papers/1702/1702.00884.pdf>.
- [2 M. M. S. B. L. B. R. G. J. H. P. P. D. R. C. L. Z. P. D. Tsung-Yi Lin, „Microsoft COCO: Common  
5] Objects in Context,” 2015-02-21 02 2015. [Online]. Available:  
<https://arxiv.org/pdf/1405.0312.pdf>.
- [2 F. D, „Batch normalization in Neural Networks,” Medium, 2017-10-20 10 2017. [Online].  
6] Available: <https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c>.
- [2 J. Brownlee, „A Gentle Introduction to Pooling Layers for Convolutional Neural  
7] Networks,” 2019-04-22 04 2019. [Online]. Available:  
<https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/>.