ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK Hallgató neve: **Nagy Norbert**

2021 Hallgató törzskönyvi száma: **T/006325/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Nagy Norbert**
Törzskönyvi száma: T/006325/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Épületdetektálás műholdképek alapján
Building detection from aerial imagery

Intézményi konzulens: Dr. Sergyán Szabolcs
Külső konzulens:

Beadási határidő: 2021. december 15.

A záróvizsga tárgyai: Számítógép architektúrák
Szoftvertervezés és –fejlesztés specializáció

A feladat

A szakdolgozat célja egy olyan webes alkalmazás fejlesztése, amely valós időben képes információt szolgáltatni egy adott terület beépítettségéről. Az elkészült alkalmazás képes egy konkrét cím alapján meghatározni annak szűkebb környezetében lévő épületek számát, valamint a beépítettség mértékét. A megvalósításhoz különböző képfeldolgozási módszerekkel támogatott, neurális hálóra alapuló megoldást kell készíteni.
Az alkalmazás működése során a felhasználó egy címet ad át bemenetként, amiből az alkalmazás egy külső dependencia segítségével az adott területről egy műholdképet generál. Az előálló kép fog bemenetként szolgálni az épületdetektáló modulnak. A képfeldolgozó program színinformációk segítségével egészíti ki az alapul szolgáló objektumdetektáló modellt, amely épületekre lett specializálva. A detektált épületek összegzésre kerülnek, majd az eredmény ember által olvasható formában visszakerül a felhasználóhoz.
A külön applikációk REST API-n keresztül kommunikálnak.

A dolgozatnak tartalmaznia kell:

- a feladat leírását,
- a szakirodalom alapján megismert hasonló módszerek, rendszerek ismertetését és értékelését,
- a választott módszer bemutatását,
- a megvalósítandó feladat tervét,
- a felhasználói leírást,
- a tesztadatokat, feltüntetve a tanító minták forrását és a teszteredményeket,
- az elért eredmények értékelését,
- a továbbfejlesztési lehetőségeket,
- a dokumentációt, a programot, a szükséges input adatokat, valamint a rendszert bemutató prezentációt CD mellékleten.

Ph.

ÓBUDAI EGYETEM Neumann János Informatikai Kar

.....
Dr. Vámossy Zoltán
intézetigazgató

A szakdolgozat elévülésének határideje: **2023. december 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2021. 12. 12.

.....

hallgató aláírása

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

.....Nagy Norbert ..... [redacted] ..... Nappali

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

..... Épületdetektálás műholdképek alapján

Szakdolgozat / Diplomamunka[2] címe angolul:

.....Building detection from aerial imagery

Intézményi konzulens: Külső konzulens:

.....Dr. Sergyán Szabolcs .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	**2021. 03. 11.**	**Szakdolgozat feladatlap és konzultációs napló kitöltésének módja, Machine learning modell választás objektum detekcióhoz**	Sergy.
2.	**2021. 03. 31.**	**Megvalósításhoz használt eszközök dokumentálásának áttekintése**	Sergy.
3.	**2021. 04. 27.**	**Dolgozat állapotának ellenőrzése**	Sergy.
4.	**2021. 05. 03.**	**Dolgozat véglegesítése**	Sergy

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 20 21. 05. 11.

Sergy.
.....
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

.....Nagy Norbert ..... [redacted] ..... Nappali

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[5] címe magyarul:

..... Épületdetektálás műholdképek alapján

Szakdolgozat / Diplomamunka[6] címe angolul:

.....Building detection from aerial imagery

Intézményi konzulens: Külső konzulens:

.....Dr. Sergyán Szabolcs .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	**2021.11.09.**	**Előző féléves munka kiegészítési lehetőségei**	
2.	**2021.11.23.**	**Implementációs lépések dokumentálásának módja**	
3.	**2021.11.30.**	**Értékelő fejezet áttekintése és bővítése**	
4.	**2021.12.07.**	**A dolgozat végső ellenőrzése**	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[7] tantárgy követelményét teljesítette, beszámolóra / védésre[8] bocsátható.

Budapest, 20 21. 12. 13.

.....
intézményi konzulens

[5] Megfelelő aláhúzandó!
[6] Megfelelő aláhúzandó!
[7] Megfelelő aláhúzandó!
[8] Megfelelő aláhúzandó!

Tartalom

Téma leírása

Modern korunk egyik legnagyobb kihívást jelentő problémája, az emberi populáció rohamtempójú növekedése. Ennek következtében állandó probléma marad az emberek lakhatásának megoldása. Ez a folyamat rengeteg kérdést vet fel maga után és további problémákat szül. Távolabbról nézve, ilyen például a várostervezés és bővítés, az ingatlanpiac helyzete, egyének szempontjából pedig az ingatlan keresés, bérlés és vásárlás. Ezesetben az egyének esetén felmerülő problémákra fókuszálnék. Az ingatlankeresés egy hosszas és körülményes folyamat, különösen azért, mert az igények mellet az árak is folyamatosan emelkednek. Számtalan tulajdonság szerint válogathatunk az elérhető ingatlanok közül. A tulajdonságok közötti prioritás személyenként különbözhet, de általánosan elmondható, hogy a legfontosabbak közé tartozik az ár, méret, felszereltség és elhelyezkedés. Különösen kiemelném a legutóbbit, az elhelyezkedést. Az ingatlan közvetlen és közeli környezete hatással van az emberek hangulatára, komfortérzetére, ezért nagy valószínűséggel választanak olyan lakóhelyet, ami nem túl zsúfolt, parkokkal övezett és meglehetősen csendes.

A keresés megkönnyítéséhez rendelkezésre áll számos ingatlankereső portál, ahol tetszés szerint böngészhetnek a felhasználók a hirdetések között. Személyre szabott szűrési feltételeket is alkalmazhatnak, amivel a fentebb említett szempontokra is tudják szűkíteni kereséseiket. Innentől kezdve a számukra szimpatikus ingatlan kiválasztása jóval leegyszerűsödik, azonban egy adott ingatlan környezetéről még mindig nem áll rendelkezésükre elegendő információ, ugyanis csak a címet, eseténként néhány csatolt képet tudnak felhasználni arra, hogy felmérjék egy ingatlan környezetét.

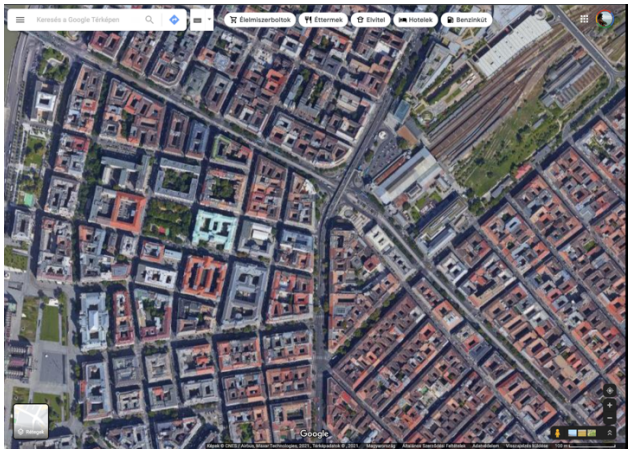
Ez az extra adat azért rendkívül hasznos, mert nagyvárosaink túlnépesedésének és az urbanizációs változások következtében nehezen található olyan terület, amely ne lenne szorosan beépítve lakóházakkal vagy irodaépületekkel, így a kiválasztott ingatlan környezetének beépítettsége hamar döntő szemponttá válhat. Több tíz vagy akár több száz megtekintett hirdetés esetén ez a manuális adatkiegészítés rendkívül zavaróvá és időigényessé válik, nem beszélve arról, ha még személyesen is meg szeretnénk győződni a területi adatokról. Jelenleg sajnos egyik hirdető portál sem biztosít olyan szolgáltatást, amely ilyen jellegű információval bővítené ki kereséseinket.

Megoldás

A megoldás célja egy olyan kiegészítő szolgáltatás létrehozása, amely teljesen eliminálja a lakóhelyek környezetének manuális felmérését, ezzel is kényelmesebb, gyorsabb és gördülékenyebb felhasználói élményt nyújtva. Ez a funkcionalitás egy olyan automatizációval lenne kivitelezve, amely az ingatlan címét felhasználva gyűjtene annak környezetéről adatokat, melyek a beépítettségről adnának információt.

Manuális keresés esetén egy lehetséges megközelítés az, amikor a felhasználó a rendelkezésére álló cím segítségével valamilyen ingyenesen elérhető, online térkép használatával digitálisan felfedezi annak környezetét. Erre alkalmas szolgáltatás például a

Google Maps vagy az Apple Maps. Ezek az eszközök többféle nézettel rendelkeznek, melyek mind más-más információval szolgálnak. A lakókörnyezetek felderítése szempontjából leghasznosabbnak a műholdkép nézet bizonyul (ld.: 1. ábra). Ez az opció csakúgy, mint a többi, fokozatos nagyítással rendelkezik, ami lehetővé teszi a terület részletes vizsgálatát.



1. ábra Google Maps műholdkép nézet

A manuális keresésnél példának használt módszert alapul véve, az automatizáció is műholdképek kivizsgálására építene. Ezért egy olyan alkalmazás elkészítésére van szükség, amely képes ilyen képeken épületfelismerésre. A felismert épületek számából és az általuk lefedett területből már előállítható egy metrika, amely az adott terület beépítettségét reprezentálja. Ezzel a szoftverrel szemben fontos elvárások is vannak. Kifejezetten nagy hangsúlyt kell fektetni a kellően pontos, és közel valós idejű működésre. Ha nem lenne képes elég rövid idő alatt eredményt adni, azzal nagyban rontaná a felhasználói élményt, ugyanis pont a hosszadalmas folyamatok kiküszöbölésére szolgálna.

Az alkalmazás esetében a következő használati esetek merülnek fel:

A különböző ingatlan hirdető portálok számára rendelkezésre állna egy fejlesztői csomag, amely segítségével integrálható lenne a szolgáltatás saját rendszereikbe. Ez a megvalósítás minden hirdetést kibővítene az érintett extra területi adatokkal. Amint egy felhasználó rákattintana egy hirdetésre, elindulna egy párhuzamos kérés, amely a hirdetésben szereplő cím környező területét mérné fel és válaszul visszaadná a beépítettségről szóló adatokat. Azért lenne kulcsfontosságú a kérés párhuzamosítása, mert így már az adott oldal betöltése közben is futhatna a vizsgálat, ezáltal mire a felhasználó elé kerülnének az ingatlan adatai, kis késéssel a terület beépítettségéről is informálódhatna a megtekintő, ezzel is javítva felhasználói élményen. Üzleti szempontból tekintve, ez a megoldás előfizetésen alapulna, mert az automatikus kérés indítások nagy forgalmat generálhatnak.

A másik megvalósítás egy általánosabb felhasználási módot céloz meg, ami nincs lekorlátozva a hirdető weboldalakra, ezért inkább a magánszemélyeknek kedvez. Ez egy böngészőbe beépülő kiegészítő formájában valósulna meg, amelynek bemenetül bármilyen cím tetszőlegesen megadható lenne. Alap működésében túlságosan nem különbözne az integrált megoldástól. Ez a változat egy kellemesebb böngészési élményhez járulna hozzá, ugyanis nem lenne szükség új ablak vagy oldal megnyitására, illetve nem lenne elvárt

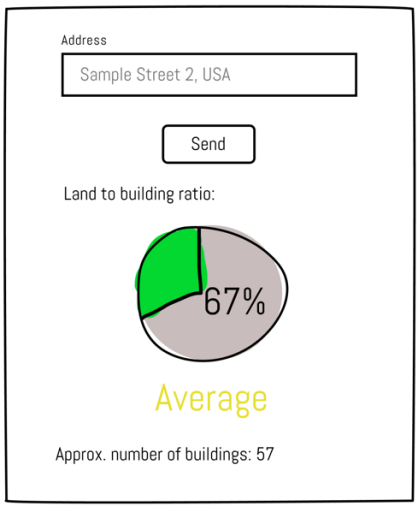
semmiféle szakértelem a funkció beüzemeléséhez, elegendő lenne egy telepítést végigcsinálni. A beépülő modul egyszerűen lenyitható lenne a böngésző fejlécéből, amellyel pillanatokon belül értékes adatokhoz juthatnának a felhasználók. Jelenleg, 2021 év végén, a legelterjedtebb webböngésző a Google által fejlesztett Chrome. Erre a népszerűségre támaszkodva, a kiegészítő egy Google Chrome Extension formájában lenne megvalósítva. Ahhoz, hogy még elérhetőbb legyen az alkalmazás, ezért ez a változat ingyenes lenne a Google webes áruházában.

1. Bevezetés

1.1. Elképzelt megoldás ismertetése

Mivel a két vizsgált eset közül, a webböngészős megoldás majdnem teljes egészében tartalmazza az integrált verziót, ezért előbbi megvalósítását fejtem ki és viszem végbe dolgozatom keretein belül.

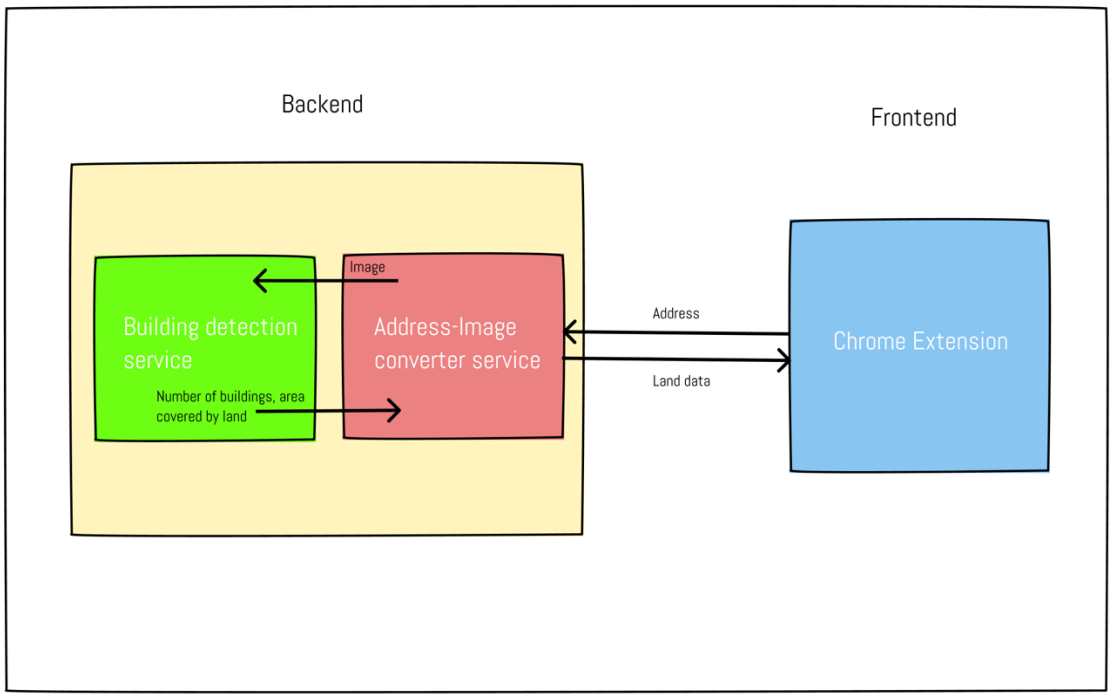
A felhasználói felület egy szöveges bemeneti mezőből, egy beküldésre szolgáló gombból és az eredményt vizualizáló részből áll. Az alkalmazás bemenetének megadása triviális, így azt nem részletezném, azonban az eredmény megjelenítése fontos a felhasználói élmény szempontjából. Egy olyan dizájn használatára van szükség, ami egyszerűbben feldolgozhatóvá és kézzelfoghatóbbá teszi a kiértékelés eredményét. A műholdkép feldolgozás eredménye, az adott cím környezetében található épületek közelítőleges számát, a vizsgált térképterület és az azon fellelhető épületek területének arányát, illetve az arányhoz rendelt osztályozásból áll. Az osztályozás az következő szintekből állna: rossz, zsúfolt, átlagos, jó, kényelmes. Ezzel az ember által olvasható értékeléssel máris érthetőbb az eredmény. Érdemes lenne az arányszámot is vizualizálni valamilyen módon, így ennek megvalósítására célszerű lenne egy diagramon ábrázolni ezt az adatpontot. Az elképzelt kinézet a 2. ábrán látható.



2. ábra Wireframe ábra az alkalmazás felhasználói felületéről

A bemenet feldolgozása egy többlépcsős folyamaton keresztül zajlana. A felhasználói felületből továbbított címet egy előfeldolgozó alkalmazás kapná meg. Ennek a rétegnek a felelőssége, a cím átkonvertálása egy olyan műholdképpé, amely az érintett területet ábrázolja. Ezen funkció megvalósításához valamelyik online térkép felhasználása lenne szükséges. A létrehozott kép egy szinttel lejjebb kerülne, egy olyan alkalmazáshoz, amely a tényleges detektálást hajtaná végre. A detekció eredményéből számított értékek válaszul visszakerülnének az egy szinttel feljebb lévő alkalmazáshoz, amely utolsó lépésként továbbítaná a választ a felhasználói felület felé.

A szolgáltatásban észrevehető rétegződés nem véletlen. Egy monolitikus alkalmazással is megvalósítható lenne a részletezett funkcionalitás, azonban ez az architektúrális döntés komoly hátrányokkal járna. Ha esetleg az alkalmazás valamely részében módosítást kéne végrehajtani, akkor az, az egész szoftvert érintené, ezért ezesetben nem lenne lehetséges a szoftver szeparált részeinek külön verziózása. Ez hosszútávon egyre több problémához vezethet, mivel a kód növekedésével, annak karbantartása is arányosan nehezebbé válik. Keresztfüggőségek alakulhatnak ki, amik még nehezebbé tennék az alkalmazás működtetését. Ezek a problémák könnyen elkerülhetők, ha már a tervezés során kellő figyelmet és időt áldozunk arra, hogy egy dinamikusan skálázható architektúrát hozzunk létre. Első lépésként meg kell határozni, hogy az elképzelt megoldás milyen komponensekre választható szét. A rendszer jól elhatárolható részeinek meghatározása után, a továbbiakban már ezen modulok egymástól függetlenül fejleszthetővé válnak. Ezek a kisebb alkalmazások, mind saját felelősségi körrel rendelkeznek, ezért, ha bármelyikben valamiféle változtatásra lenne szükség, akkor az teljesen izoláltan végrehajtható. Ezt a tervezési mintát mikroszerviz alapú architektúrának nevezik. Ezt a tervezési módszert alkalmazva hoztam létre a 3-as ábrán látható rendszertervet. Az architektúra felépítését a dolgozatom struktúrája is tükrözi. A rendszer dekompozíciójából született komponensek elhatárolt módon kerülnek kidolgozásra, megvalósításra és kiértékelésre.



3. ábra Vázlatos rendszerterv

2. Irodalomkutatás

2.1. Felhasználói felület

Egy felhasználói felület elkészítésének több lehetséges alternatívája lehet. Mivel ez az a rész, amivel a végfelhasználók interakcióba lépnek, ezért fontos, hogy minél gyorsabb, interaktívabb, érthetőbb és korszerűbb legyen. Felhasználási módtól függően, lehet szó asztali vagy webes alkalmazásokról, illetve ezek hibrid megoldásairól. Pár évvel ezelőttig nagy népszerűségnek örvendtek az asztali alkalmazások, viszont manapság egyre jobban mozdulnak a trendek a webes alkalmazások irányába. Asztali alkalmazások esetén elterjedt keretrendszerek mint például .NET-es WinForms és WPF, vagy a Java-hoz készített Swing egyre elavultabbnak számítanak manapság. Sok cég már olyan platformfüggetlen keretrendszereket használ fel, amelyekkel webes technológiákkal (HTML, JavaScript, CSS stb.) fejleszthetők asztali alkalmazások (Ilyen például az Electron). Ebből adódóan a megírt szoftverhez nem szükséges külön webre specializált variánst létrehozni, ami miatt rengeteg fejlesztési erőforrás és idő megspórolható. Mivel az általam elképzelt szolgáltatás az online böngészés megkönnyítése céljából készülne, ezért célszerű lenne webes technológiákat használni a megvalósításához.

A modern webböngészők által nyújtott lehetőségeknek köszönhetően, akár már böngészőbe beágyazott kiegészítőket is készíthetünk. Ezek a kiegészítők a böngészőkben található áruházakban letölthetők és telepíthetők, és a böngésző kiegészítéseiként viselkednek. Épp ezért nincs szükség új oldal vagy ablak megnyitására használatukhoz, ugyanúgy elérhetők az adott böngésző fejlécéből, mint bármely egyéb beépített funkció. A jelenleg legnépszerűbb webböngésző, a Google Chrome kiegészítőit, a Chrome Extension-öket például véve, könnyen megérthető ezek koncepciója, működése és létjogosultsága.

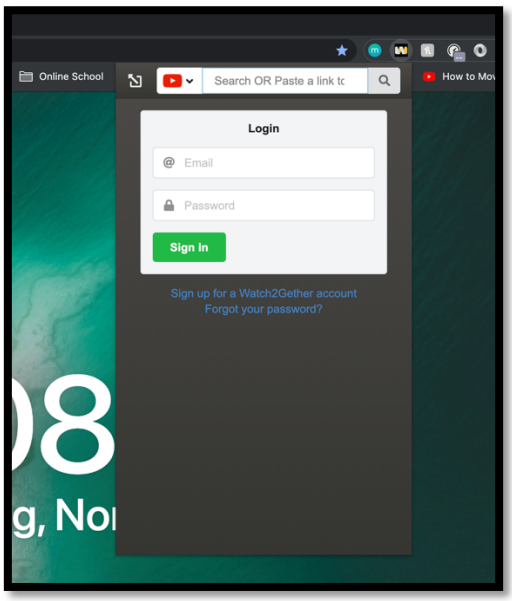
Chrome Extension

A Chrome kiegészítők webes technológiákon alapuló (HTML, CSS, Javascript) szoftverek, melyek segítségével a felhasználók személyre szabhatják és kiegészíthetik böngészőjük működését. Azért hívják őket kiegészítőknek, mert kiterjesztik a böngésző alap funkcionalitását a Google által biztosított eszközök segítségével. A rendelkezésre álló kiegészítők egy egységes áruházban találhatók, ahonnan akár ingyenesen is letölthetjük és telepíthetjük őket. A kiegészítőknek két fajtája létezik, a passzív és az aktív kiegészítők. A passzív kiegészítők családjába tartozik minden olyan kiegészítő, amelyekkel nem képes direkt módon interaktálni a felhasználó. Ezek valamilyen automatikusan alkalmazott változtatást hoznak a böngészőbe, pl. reklám blokkolók, helyesírás ellenőrzők. Aktív kiegészítőnek minősül minden egyéb, amellyel interaktálva valamilyen extra működést érhetünk el. Ilyen például a 4. ábrán látható kiemelt példa, amely segítségével csoportos videónéző konferenciát indíthatunk.

Pozitív tulajdonsága a weboldalakkal szemben, hogy a tartalmak többszöri letöltésére nincs szükség, ugyanis ez telepítéskor történik meg egyszer. Viszont ebből az is adódik, hogy a weboldalaknál tapasztalt dinamikus változások követése nem lehetséges. Míg a weboldalak a klienstől függetlenül frissíthetők, a kiegészítőket manuálisan kell frissen tartanunk, vagy a

böngészőre bízni a telepített funkciók aktualizálását. Ennek köszönhetően a felhasználók nem tapasztalnak észrevehető töltési időket, ami jóval kényelmesebbé teszi a felhasználói élményt.

Lehetőségünk van kiegészítőket fejleszteni is, amihez nincs szükség másra, csupán egy tetszőleges kódszerkesztőre és a Google Chrome fejlesztői környezetére, amit a böngészőn keresztül érünk el. A fejlesztéssel kapcsolatos részletes információk megtalálhatók a Google releváns dokumentációs oldalán.



4. ábra Chrome kiegészítő példa

2.2. Cím-kép konvertálás

A felhasználói felületről érkező cím feldolgozásához egy olyan szolgáltatásra van szükség, amely kellő konfigurálhatósággal, gyorsasággal és pontossággal képes műholdképeket generálni a megadott koordináták vagy címek alapján. Ennek kiválasztásakor az következő szempontokat kell figyelembe venni:

A szolgáltatás rendelkezzen olyan interfésszel, amely képes a geokódolásra, illetve az ehhez tartozó földrajzi helyről műholdkép generálására. A geokódolás egy olyan folyamat, ahol egy hely ember által olvasható leírása, egy cím, földrajzi koordinátákká, legtöbb esetben szélességi és hosszúsági fokokká konvertálódik.

A szolgálatatás rendelkezzen személyre szabható tulajdonságokkal, amelyekkel a visszaadott képek minősége, dimenziói, tartalma és stílusa befolyásolható.

A szolgálatás által használt képek jól reprezentálják a való életbeli területek állapotát, vagyis fontos, hogy a képek minél sűrűbben frissüljenek annak érdekében, hogy a szolgáltatás reális adatokat adjon vissza. Az adatok frissessége elég szubjektív a használati esetek változatossága miatt, azonban az általam vázolt kontextusban, a maximum egy éven belül frissülő képek még elfogadhatónak számítanak. Ez a tulajdonság a téma

szempontjából rendkívül fontos, mert egy építkezés átlagos esetben nem tart tovább 1-2 évnél.

A fenti szempontok miatt a MapBox és a Google Maps szolgáltatásokra szűkül a keresés. A többi és a legtöbb online térképre is igaz, hogy csak utak és forgalmi adatokat szolgáltatnak. Mindkét szolgáltatás rendelkezik egy ingyenes lekérdezési kerettel, ami mindkét esetben bőven elég nagy (havi több tízezer) ahhoz, hogy egy koncepció validálásához fel lehessen használni. Mivel mindkét opció rendelkezik az szükséges tulajdonságokkal, ezért a döntő szempont a képek frissessége. A Google Maps széles körű felhasználása és elterjedtsége miatt, a populáltabb területek sűrűbb frissítési rátával rendelkeznek. A MapBox esetében a frissítési idő akár 3 évre is elhúzódhat, mivel a cég az ügyfeleik felhasználási módja szerint frissíti térképadatait.

Ennek következtében a cím-kép konverzióhoz a Google Maps Static API - magyarul alkalmazásprogramozási felületének - használatát választom. A képek generálásáért felelős interfész HTTP kéréseken keresztül érhető el, ahogy az alábbi példán is látható:

https://maps.googleapis.com/maps/api/staticmap?center=Brooklyn+Bridge,New+York,NY&zoom=13&size=600x300&maptype=roadmap&key=API_KEY

5. ábra Google Maps Static API példa

2.3. Bemenet feldolgozása és továbbítása

A felhasználói felület és a képfeldolgozó réteg közötti kommunikációhoz egy olyan protokoll megválasztására van szükség, amely konzisztens, kellően gyors és hibatűrő. A konzisztencia kifejezetten fontos, mivel nem engedhető meg, hogy komponensek között hibás vagy hiányos üzenetek közlekedjenek. A hibatűrőség pedig ezesetben annyit jelent, hogy a kommunikáció közben előforduló hibákat a kommunikációs csatorna képes közvetíteni, ezáltal lehetőséget biztosítva fejlesztőnek azok lekezelésére. Manapság alkalmazások közötti kommunikációra gyakran használt protokollok a HTTP, az erre alapuló REST és SOAP. Igaz, a SOAP már nem annyira elterjedt, de rengeteg régi rendszer használja a mai napig. A HTTP protokoll onnan nyerte el rendkívüli népszerűséget, hogy a manapság legnépszerűbb

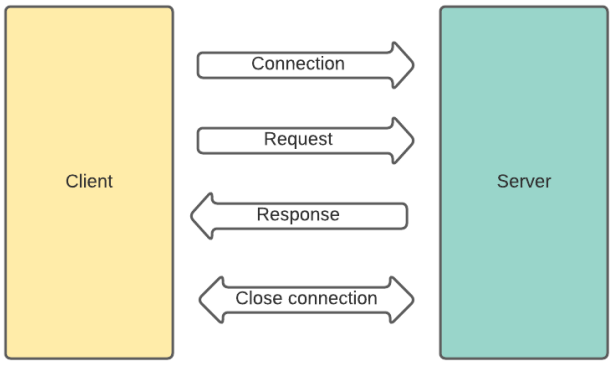
kommunikációs modell, a REST, HTTP kapcsolatra, annak metódusaira és hibakezelésére alapszik. Alábbiakban az említett protokollok közötti különbségeket részletezem.

2.3.1. Kommunikációs protokoll kiválasztása

HTTP protokoll

Napjainkban a webes kommunikáció egyik legnépszerűbb alternatívája a HTTP, Hypertext Transfer Protocoll. A HTTP protokoll kétirányú kommunikációt tesz lehetővé kliens és szerver között. Ez egy TCP-n futó „stateless”, azaz állapotmentes protokoll. A TCP tulajdonságai miatt az üzenetek sikerességéről vagy hibáiról visszajelzést kapunk. Ez lehetővé teszi a hibák helyénvaló kezelését. A kapcsolat állapotmentessége azt jelenti, hogy a külön kérések nem függenek egymás állapotától, tartalmától és sikerességétől.

Üzenet küldése esetén TCP kapcsolat alakul ki a kliens és szerver között. A kapcsolat kiépítése után a szerver feldolgozza a kérést, majd a visszaküld egy választ. Az üzenetcsere megtörténése után a kapcsolat mindkét oldalról megszűnik. Hiba esetén előre definiált hiba kódokat és üzeneteket szolgáltat a szerver a kliens számára. A HTTP protokoll működésének vizuális ábrázolása a 6. ábrán látható.



6. ábra HTTP kapcsolat

Létrehozhatunk olyan kommunikációt is, amely egy vagy több előre létrehozott TCP kapcsolatot használ, esetleg ezeket időközönként rotálva. Az ilyen típusú, nyitott kapcsolatokat használó kommunikációt HTTP Keep-Alive-nak hívjuk. Ez a felhasználás leginkább erősen valós idejű, nagy forgalmú alkalmazások esetében hasznos, mert így a HTTP kapcsolatok kiépítéséhez szükséges időt ki lehet kerülni.

A HTTP több metódussal is rendelkezik, ezek közül egy webes interfész kialakításánál a legfontosabbak a GET, POST, PUT, PATCH és DELETE. Nevükből kikövetkeztethető felhasználásuk is. A GET adatok lekérésére, a POST adatbevitelre, a PUT egy bizonyos erősforrás teljes frissítésére, a PATCH egy bizonyos erőforrás részleges frissítésére, a DELETE pedig erőforrások törlésére szolgál.

A HTTP kérések két fő részre bonthatók. Egyik a header, vagyis fejléc szekció. Ez a rész a tartalom típusáról, hosszáról, a küldőről és egyéb tulajdonságokról tárol, a kommunikáció

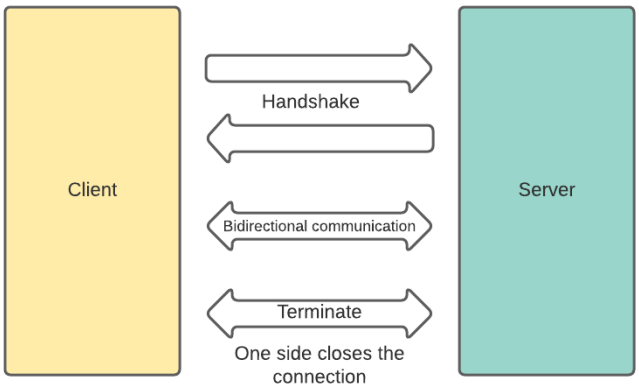
szempontjából létfontosságú adatokat. Lehetőségünk van tetszőleges fejléc elemek definiálására is. Ezekben például saját alkalmazásunkhoz szükséges paramétereket továbbítását tehetjük elvárhatóvá. A kérés paramétereit az URL-en kívül a body, vagyis az üzenet törzsében tudjuk elhelyezni. Az üzenet típusát a 'Content-Type' fejlécben tudjuk definiálni. Ez a paraméter határozza meg, hogy az üzenet törzse milyen formátumban értelmezendő.

WebSocket

A WebSocket egy szintén kétirányú kommunikációra szolgáló protokoll, amely a HTTP egy full-duplex kiterjesztése, viszont nagy különbség, hogy a HTTP-hez képest „stateful", vagyis függ az előző kérésektől oly módon, hogy addig marad nyitva a kapcsolat, amíg azt a kliens vagy a szerver nem bontja fel. A full-duplex kapcsolat annyit rejt, hogy a szimpla HTTP kapcsolattal ellentétben itt egyidőben, mindkét irányban lehetséges az üzenetek küldése.

A HTTP esetében a kliens képes kérést indítani, a szerver pedig erre válaszolni. A WebSocket kiegészíti ezt a működést oly módon, hogy a szerver is képes kéréseket küldeni a kliens irányába. Emellett a szerver bármikor képes üzenetet küldeni a kliensnek, ezért ezzel a működéssel könnyen lehet értesíteni a kliens oldalt, a szerver oldali változásokról. Ez azt jelenti, hogy a kérések egy előre létrehozott csatornán keresztül fognak cserélődni. Belátható, hogy egy előre kiépített kapcsolaton keresztül gyorsabb információáramlás érhető el, ugyanis megspóroljuk a kérésenkénti új kapcsolatok nyitását.

A kapcsolat létrehozása előtt történik egy „handshake" (kézfogás), ahol a kliens és a szerver megegyezik egy új kapcsolat megnyitásáról, és felépíti azt. A két oldal között addig él a kommunikációs csatorna, ameddig azt valamelyik fél meg nem szakítja. Ez majdnem identikussá teszi a HTTP Keep-alive és a WebSocket működést, azonban a HTTP esetében nem lehetséges a szerver oldaláról kéréseket indítani a kliens felé (ld.: 7.ábra).



7. ábra WebSocket kapcsolat

A WebSocket alapú kommunikáció mára széles körben elterjedt a chat és más egyéb valós időben frissülő szolgáltatások népszerűsége miatt, ahol a szabvány tulajdonságai jól kihasználhatók.

Konklúzió

Választásom a HTTP protokollra esett, mivel az én felhasználási esetemben nincs szükség intenzív kétoldalú kommunikációra, így nem tudnám kihasználni a WebSocket nyújtotta lehetőségeket. A HTTP kommunikációnak több iparban elfogadott standardja létezik. Ezek közül jelenleg a kettő legismertebb a SOAP (Simple Object Access Protocol) és a REST (Representational State Transfer).

SOAP

A SOAP – Simple Object Access Protocol, azaz Szimpla Objektum Elérési Protokoll egy Microsoft által fejlesztett standardizált modell, amely a már akkor elavulttá vált webes kommunikációs technikák leváltására szolgált. Lehetőséget nyújt távoli applikációk összekötésére és azok metódusainak meghívására.

A szereplők közti üzenetek továbbítására és fogadására XML formátumot használ (ld.: 8. ábra). Nagyon szigorú és merev felépítésű, ami az XML-lel társítva rendkívül komplex és átláthatatlan üzeneteket eredményezhet, azonban a legtöbb nyelvhez már léteznek könyvtárak, amelyek az üzenetek készítését és feldolgozását megkönnyítik. A szemmel nehezen feldolgozhatóságának következtében, beépített hibakezeléssel is rendelkezik, amely egyszerűbbé teszi a rosszul definiált XML javítását.

Egy SOAP üzenet, a HTTP-hez hasonlóan több részből épül fel, melyeket az alábbiakban részletezek:

Envelope (boríték) – Egy SOAP kérésben kötelező megjelennie, ugyanis az üzenet eleje és vége definiálható vele.

Header (fejléc) – A HTTP-hez hasonlóan az itteni fejléc is az üzenettel kapcsolatos metaadatok tárolására szolgál. Ez egy opcionális elem.

Body (törzs) – Egy kötelező elem, amely az üzenet tartalmát tárolja.

Fault (hiba) – Egy olyan opcionális elem, amely az esetekben előforduló hibainformációkat tárolja.

A XML azon tulajdonságából kiindulva, hogy elég kis méretű és terhelésű, ezért maga a SOAP is betudható ebbe a kategóriába. Alkalmazások közötti kommunikációra kifejezetten alkalmas, mert azok metódusait direkt módon elérhetjük. Ebből adódóan a távoli komponensek között jól definiált interfészeket hozhatunk létre. Nyílt webes végpontok elkészítéséhez azonban elég kényelmetlen, mivel a kérések elkészítéséhez ismerni kell az elérni kívánt alkalmazás struktúráját, illetve meg kell küzdeni a viszonylag bonyolult XML formátummal, ezért erre az esetre célszerűbb lenne egy másfajta kommunikációs formát bevonni.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" xmlns:cwmp="urn:dslforum-org:cwmp-1-0">
  <SOAP-ENV:Header>
    <cwmp:ID SOAP-ENV:mustUnderstand="1">112</cwmp:ID>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <cwmp:SetParameterValues>
      <ParameterList SOAP-ENC:arrayType="cwmp:ParameterValueStruct[1]">
        <ParameterValueStruct>
          <Name>Device.WiFi.AccessPoint.10001.Enable</Name>
          <Value xsi:type="xsd:boolean">1</Value>
        </ParameterValueStruct>
      </ParameterList>
      <ParameterKey>bulk_set_1</ParameterKey>
    </cwmp:SetParameterValues>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

8. ábra SOAP üzenet XML formátumban

REST

A REST (Representative State Transfer) egy olyan HTTP üzenetváltásra épülő architektúrális stílus, amely a dinamikusan növekvő és változó internetes kommunikáció valamilyen szintű egységesítésére szolgál. Ennek érdekében megkötéseket definiál, amik szerint a webes kommunikációnak viselkednie kéne. Ha egy alkalmazás teljesíti ezeket a feltételeket akkor azt RESTful-nak nevezzük.

Az első és egyik legfontosabb megkötés az, hogy a felhasználók felé definiált interfész struktúrája egységes legyen. Emellett ehhez az interfész struktúrához a felhasználóknak kötelezően ragaszkodniuk kell, illetve minden erőforrás kizárólag egy logikai URI végpont által kell, hogy elérhető legyen.

A kliens és szerver közötti kapcsolatnak teljesen transzparensnek lennie. Ez azt jelenti, hogy a kliens és szerver alkalmazásoknak képesek kell lenniük az egymástól való független változásra. A kliens birtokában csak az erőforrások elérési útjai lehetnek és nem függhet a szerver oldali fejlődésektől, kizárólag a biztosított interfészektől.

Minden kliens és szerver között kialakított kapcsolatnak állapotmentesnek kell lennie. A szerver nem tárolhat el előző kérési adatokat annak érdekében, hogy az újakat felülbírálja. Minden külön kérést újnak kell tekinteni, teljes izoláltsággal. Azonban, ha mégis állapotfüggő funkcionalitás elérése a cél, akkor az autentikáció után minden egyes kérésben továbbítani kell a szerver felé az autentikációs adatokat.

Az áramoltatott adatoknak cache-elhetőnek kell lennie szerver vagy kliens oldalon. Ez azért létfontosságú, mert bizonyos adatok átmeneti tárolásával rengeteg erőforrás spórolható meg, ezzel is javítva a teljesítményen, felhasználói élményen és a rendszer skálázhatóságán.

Legutolsó sorban a kliens nem tudhat a szerver oldali alkalmazás rétegződéséről, ezért ez az információ biztosított végpontokon sem jelentkezhet.

A REST a SOAP-nál egy jóval flexibilisebb architektúrát nyújt a kommunikációhoz. Direkt módon nem rendelkezünk az erőforrások felett, azoknak csak a felénk biztosított interfészein keresztül tudunk adatot cserélni. Ennek következtében nem alakul ki függés a konkrét megvalósítás és a kliens között. XML helyett URL-en keresztül vagy a HTTP kérés törzsében

fogalmazhatók meg a kérés paraméterei. A REST a fentebb említett négy fontosabb HTTP metódust képes használni: GET, POST, PUT, DELETE.

A SOAP-al ellentétben nem kötelező egységes formátumot használni a válaszban. A definiált végpontok adhatnak kimeneti adatot XML, JSON, CSV vagy egyéb formátumokban. Jelenleg a legelterjedtebb a JSON, könnyű áttekinthetősége és egyszerű strukturáltsága miatt, illetve ez áll legközelebb az adatok valós reprezentálásához. A JSON struktúra a 9. ábrán látható.

9. ábra JSON formátum

SOAP vs. REST

Olyan alkalmazások esetében, ahol a kommunikáció kizárólag belső komponensek között zajlik és csak különálló szolgáltatások összeköttetésére szolgál, ott érdemesebb lehet a SOAP-ot, és a vele járó előnyönöket választani. A SOAP standardizáltsága és beépített hibakezelése nagy előnyt jelenthet nagyobb, elosztott rendszereknél. Általánosabb használatnál azonban, sokkal célravezetőbb a REST alkalmazása. Rugalmassága, bővíthetősége és átláthatósága miatt egyszerűbb integrálhatóságot tesz lehetővé, illetve a REST interfészek a felhasználók számára is elérhetővé tehetők anélkül, hogy részletesen ismerniük kelljen az adott szoftver felépítését és architektúrális hátterét.

2.3.2. Keretrendszer kiválasztása

A front -és backend közötti kommunikációs protokoll kiválasztása után, következő lépésben a fejlesztéshez felhasználható eszközöket, ideértve az implementálásra használt nyelvet és opcionális esetben keretrendszert kell megválasztani. A rendszer backendjét képező alkalmazások megvalósításához célszerű olyan eszközöket kiválasztani, amelyek támogatják webes szoftverek elkészítését és elég megbízhatók ahhoz, hogy hosszú távon skálázható és terheléstűrő megoldásokat készítsünk velük.

A fejlesztéshez használt nyelv kiválasztását az alkalmazott programozási paradigma is meghatározza. Jelenleg még mindig a legelterjedtebb paradigma az Objektum Orientált Programozás (OOP), mert általánosságban a legtöbb probléma megoldásánál jól alkalmazható.

Előfordulnak olyan speciális esetek, amikor nem az OOP használata bizonyul a legjobb megoldásnak. Például adatfolyam feldolgozáson alapuló alkalmazásoknál érdemesebb lehet a Funkcionális Programozás paradigma alkalmazása. Az én esetemben az érintett probléma könnyen modellezhető az OOP elvek segítségével, ezért az backend megírásához mindenképp objektum orientált nyelveket használnék.

Jelenleg webalkalmazások elkészítéséhez a legnépszerűbb nyelvek közé tartozik a Java és a C#. Mindkét nyelvhez létezik olyan keretrendszer, amely webalkalmazások létrehozásához biztosít kiegészítéseket. Java esetében a leggyakrabban használt alternatíva a Spring Boot, míg C#-nál az ASP.NET Core, amely a nyelvvel együtt a .NET családjába tartozik.

ASP.NET Core

Az ASP.NET Core a .NET platform egy kiterjesztése, ami webalkalmazások készítéséhez használt eszközöket és könyvtárakat tartalmaz, illetve webes kérések kiszolgálásának lehetővé tételéhez egy alap keretrendszert biztosít. Fontos megjegyezni, hogy az ASP.NET nem összekeverendő az ASP.NET Core-al, ugyanis az előbbi kizárólag Windows platformon elérhető, utóbbi pedig cross-platform.

Az MVC (Model-View-Controller) architektúrális tervezési mintát alapértelmezetten támogatja. Ez abban is megnyilvánul, hogy erre a mintára előre létrehozott projekt sablont is tartalmaz a keretrendszer, aminek használatával jó pár fejlesztési órát megspórolhatunk. Dinamikus weboldal készítésére is lehetőséget nyújt a keretrendszer Razor segítségével. A dinamikus jelző azt jelenti, hogy a Razor szintakszis használatával C# kód integrálható a webes nézetekbe. Ez valamennyire hasonlít a Java EE (Enterprise Edition) -ben megtalálható JSP (JavaServer Pages) technológiára, amely Java szintaktika beágyazását teszi lehetővé HTML kódba. A Razor az adatkötést is támogatja, ezért az üzleti logikában történő adat változások közvetlen kivezethetők a felhasználói felületekre.

Objektumorientált programok fejlesztésénél alapelvnek számító dependencia szegregáció is könnyen megvalósítható az ASP.NET-be integrált IoC (Inversion of Control – Függőségek megfordítása) konténer segítségével. A kódban használt függőségeket, így egy helyen definiálhatjuk, amiket már képes a konténer kezelni futási időben, ezzel is elkerülve nagy mennyiségű sablon kód megírását.

Bemeneti adatok validálásra, a későbbiekben részletezett Spring-hez hasonló megoldás alkalmazható. Alapértelmezetten beépített és saját attribútumok segítségével érhetünk el adatellenőrzést. A validációs logikát elkülöníthetjük, majd erre, a hozzá definiált attribútum segítségével hivatkozhatunk rá. Ez azért egy rendkívül hasznos funkció, mert sokkal átláthatóbb és kevésbé komplex kódot eredményez. Emellett a kód duplikációt is csökkenti, mert ezek az attribútumok újra felhasználhatók, ezért a szükséges logikát elég csak egyszer, egy helyen definiálni.

Alapértelmezetten különféle autentikációs metódusokat is támogat a keretrendszer. Használhatunk külső forrásokat, mint például Facebook, Google, Microsoft, vagy akár token alapú megoldásokat, mint például a JWT (JSON Web Token).

Spring és Spring Boot

A Spring keretrendszer egy átfogó fejlesztési és konfigurációs modell modern Java alkalmazások fejlesztéséhez. Célja, hogy a fejlesztőknek minél kevesebb időt kelljen tölteniük, nem az üzleti logikához köthető sablon kód írásával, ezzel felgyorsítva a nagyvállalati szoftverek lefutásának idejét.

Egy szoftver fejlesztésének és hosszútávú karbantarthatóságának szempontjából nagyon fontos, hogy a belső komponensek direkt módon ne függjenek egymástól. Ezeket a függőségeket különböző absztrakciós rétegek megvalósításával célszerű feloldani, azonban ezek implementálása is rengeteg sablon kóddal jár, amelynek megírása lassítja a fejlesztés menetét. Erre megoldásul, a Spring ökoszisztéma egyik legértékesebb funkcióját alkalmazhatjuk, a beépített függőségkezelést. Az ASP.NET-hez hasonlóan itt is előre definiálhatjuk a program során használt függőségeket, komponenseket, azonban ezesetben ezt sokkal rugalmasabban tehetjük meg.

Régebbi Spring verziókban kézzel kellett XML konfigurációk segítségével az úgynevezett „Bean”-eket definiálni, amik a Spring IoC konténerének elemeiként viselkednek majd. Modernebb verziókban ez a funkcionalitás már sokkal rugalmasabb formában érhető el, ugyanis nem egy helyen kell definiálni az összes függőséget. A komponenseket külön fájlokban helyezhetjük el, amiket a megfelelő annotációkkal ellátva ugyanúgy képes felismerni a Spring, mint a régi típusú konfigurációkat. Ilyen annotációk például a @Service vagy a @Component. Ez a szintaktika teszi lehetővé a függőségek injektálását is. Ahhoz, hogy a Spring minden függőséget tartalmazzon, induláskor felfedezi az összes ilyen annotációval ellátott fájlt, amik alapján összeállít egy függőségi fát, majd futás során ezt a struktúrát használja fel a megfelelő dependenciák injektálására.

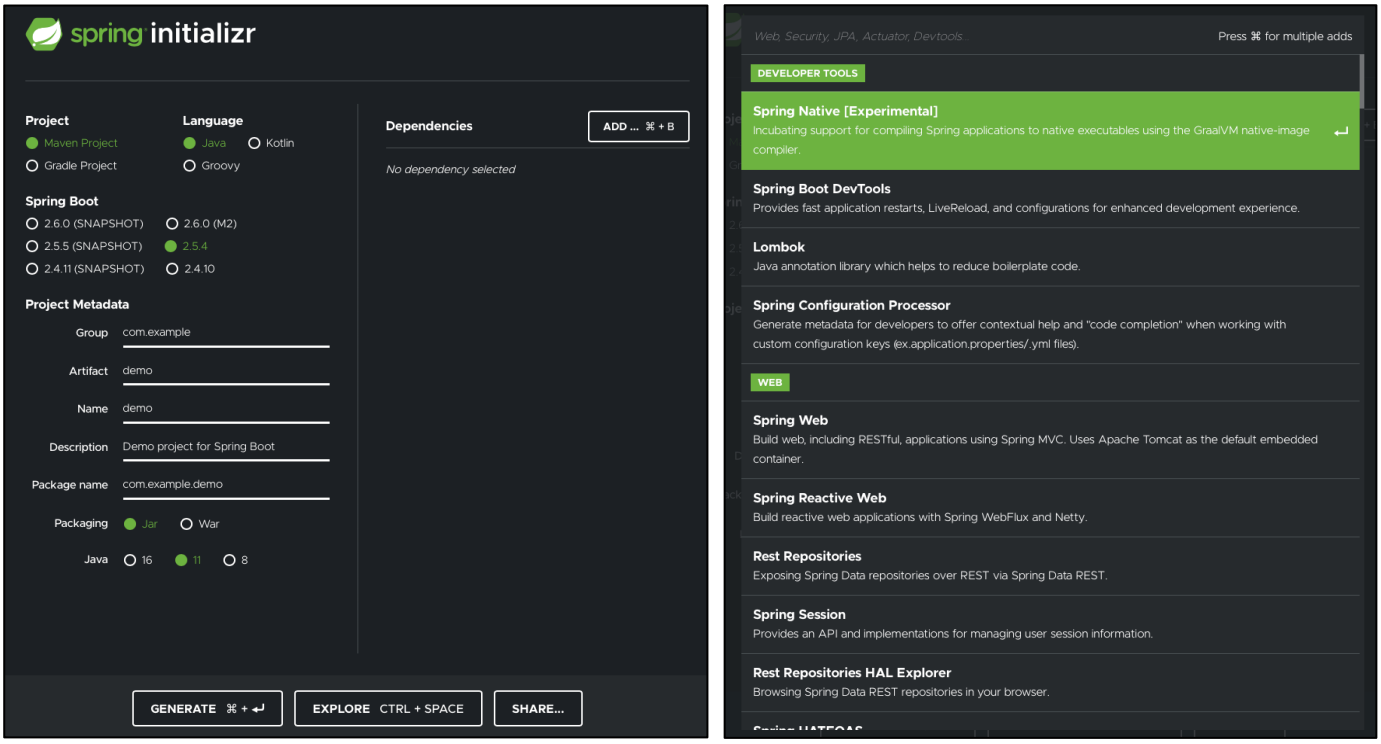
Egy másik fontos objektum orientált alapelv a rétegeket átvágó aggályok (cross-cutting concerns) kezelése. Gyakran ezek nem olyan tisztán szétválaszthatók, mint a rendszer többi része, emiatt kód duplikációhoz és szoros függőségek kialakulásához vezethetnek. A rétegeken átívelő komponenseket a hagyományos elvek szerint nem igazán lehet elhelyezni a programstruktúrában, ezért implementálásukra, mindig egy kevésbé kellemetlen módot kell választani. Ilyen lehet például a logolás, monitorozás, bizonyos biztonsági funkciók, illetve hibák és események kezelése. Ennek feloldására az AOP (Aspektus Orientált Programozás) alkalmazása egy jó megoldást nyújt. Célja, hogy az aggályokat enkapszulálja, oly módon, hogy azok az eredeti kód módosítása nélkül használhatók legyenek. Ezt úgy teszi, hogy ezeket a kódrészeket „aspektusokba” helyezi, amelyeket belépési pontok definiálásával használhatunk. A Springben, ezek a belépési pontok annotációk segítségével implementálhatók. Hogyha például van néhány metódusunk, amelyeknek a futási idejét logolni szeretnénk, akkor egyszerűen ezekre a metódusokra elhelyezünk egy, az adott aspektushoz tartozó belépési pontot, amely az aspektusban definiált implementáció alapján végrehajtja a kívánt funkcionalitást.

Autentikálásra és autorizációra egy külön erre a célra kialakított, jól személyre szabható keretrendszer áll rendelkezésünkre, a Spring Security. Alapértelmezett több elterjedt metódust is támogat, mint például az OAuth és az SAML (Security Assertion Markup Language).

Ha felhasználói bemenetek validálásáról van szó, a Spring erre is alapértelmezetten nyújt egy elég kézenfekvő megoldást. Annotációk alkalmazásával alapértelmezett vagy akár kusztomizált validátorokat is implementálhatunk. Ez szintén azért rendkívül hasznos, mert így az ellenőrzéshez használt logika jól elkülöníthető, ezáltal újra felhasználhatóvá válik.

A Spring mindezek mellett teszteléshez, adatbáziseléréshez és webes kommunikációhoz is biztosít megoldásokat, illetve rengeteg kiterjesztéssel rendelkezik, amelyekkel még több funkcionalitást adható a keretrendszerhez. Érzékelhető, hogy a Spring erősen az annotációk használatát helyezi előnybe. Megfelelő felhasználásukkal rendkívül letisztult kód elkészítése válik lehetővé, mert minden egyéb, a kódhoz nem feltétlen tartozó függőség jól leválasztható és szükség esetén behivatkozható.

A Spring Boot egy olyan eszközként fogható fel, amely lehetővé teszi Spring keretrendszerre alapuló Java alkalmazások fejlesztését, minimális komplikációval, kellő gyorsasággal és kényelemmel. Egy előre összegyűjtött dependencia halmazból választhatunk alkalmazásunk létrehozásakor, mindezt pár kattintással. Ezután konfigurációra nincs is szükség, kivéve, ha azt az adott szolgáltatás megköveteli. Az projekt alapjának összeállítását a https://start.spring.io oldalon tehetjük meg. Ez az úgynevezett Spring Initializer. Ezen a weboldalon egy átlátható felhasználói felületen válogathatjuk össze a projekthez használni kívánt technológiákat és függőségeket, amiből egy kész projekt sablon generálhatunk (ld.: 10. ábra).



10. ábra Spring Initializer és dependencia választás

Konklúzió

Mindkét keretrendszer remek tulajdonságokkal bír és nagyon jól támogatják webes alkalmazások elkészítését, azonban a Springet jelen helyzetben nagyobb közösség támogatja, emiatt nagyobb mennyiségű szakirodalom található hozzá az interneten. Az ASP.NET Core platformfüggetlensége még nem teljesen kiforrott, ugyanis sajnos Windows-on kívül a legtöbb operációs rendszeren problémás a fejlesztői környezet felállítása és az adott projektek működésre bírása, illetve a keretrendszer erősen függ a Microsoft által kiépített ökoszisztémától. Ezt az is alátámasztja, hogy a legjobb .NET kódszerkesztő, a Visual Studio még nem teljesen támogatott más operációs rendszereken, illetve vannak olyan fejlesztői eszközök, amelyek csak Windows-os környezetben érhetők el, mint például a LocalDb, amit lokális adatbázisok létrehozása használhatunk. Ez Windows-on remekül működik és megkönnyíti a fejlesztést, azonban más platformokon egyéb megoldások használatára kényszerülünk. Emellett vannak olyan kiegészítések, amik papíron támogatnak más operációs rendszereket is, azonban a gyakorlatban ez nem minden esetben igaz. A Spring azonban nem platformfüggő, ezért bármilyen rendszeren könnyen használható.

A Spring egyszerűsége és függetlensége miatt hosszú távon jobb fejlesztői eszköznek bizonyul, ezért az adatfeldolgozásért felelős backend alkalmazást Java-ban, a Spring keretrendszer segítségével fogom implementálni.

2.4. Képfeldolgozó alkalmazás

A bemeneti adatok konvertálása és feldolgozása után létrejött műholdképek kiértékeléséhez egy olyan alkalmazásra van szükség, amely kellő gyorsasággal és hatékonysággal tudja kezelni az objektumdetektáláshoz használt modellt. Ebbe beletartozik a modell beolvasása, memóriában tartása és kiértékelése. Ehhez azonban jól kell megválasztani a használt eszközöket, annak érdekében, hogy megbízható eredményeket kapjunk.

Az épületdetektáláshoz használt módszer kiválasztásánál figyelembe kell venni a használati eset által kitűzött elvárásokat. Egy olyan neurális háló felhasználására van szükség, amely képeken végrehajtott műveletekre, azon belül objektum felismerésre van specializálva. Opcionálisan az algoritmus legyen képes a detektált objektumokra maszkot készíteni, ezzel pontosítva annak határait, eredményül egy pontosabb detekciót adva. Mivel a szolgálatatás közel valós idejű működést ígérne, ezért rendkívül fontos, hogy választott algoritmus másodpercek alatt képes legyen az objektumdetektálás lefuttatására, mindezt konzisztens módon.

A légifelvétel alapú képfeldolgozás számos területen alkalmazott és hasznosítható módszer. Segítséget nyújt térképkészítésnél, az urbanizáció okozta változások megfigyelésénél, úthálózatok felderítésénél, katasztrófa sújtotta területek kárának felmérésénél és még sok más hasonló területen. Egy légifelvétel rengeteg információval szolgál, melynek feldolgozása, egyéb adatpontokkal való kibővítése csak növeli a felhasználhatóságát, illetve hasznosságát.

A számítástechnika gyors fejlődése és a számítástudományban tett hatalmas előrelepéseknek köszönhetően, egyre több olyan eszközt alkalmazhatnak a kutatók, amelyekhez eddig nem volt hozzáférésük. Az érintett terület még a mai napig kihívásokat okoz. Mivel a légifelvételek korántsem általánosak, ezért az kiértékelések eredménye a feldolgozott kép tulajdonságai függvényében változhat, emiatt kifejezetten nagy kihívást okoz olyan módszereket találni, amelyek minden esetben hasonlóan jó eredményt adnak. Számításba kell venni a képek felbontását, minőségét, a fényviszonyokat, a színeket, a fénykép készítésének a merőlegestől számított szögét és még számos olyan tulajdonságot, amit nem lehet kifejezetten biztosnak ítélni.

Rengeteg cég, kutató és lelkes hobbiprogramozó foglalkozik a témával, így többféle megoldás is született már az épületek képi alapú detektálására. A továbbiakban az alapfogalmak részletezése után, néhány megoldás bemutatása és értékelése következik.

Neurális hálók [1]

A gépi tanulást az elmúlt évszázadban drasztikus fellendülés érte, ami az élet minden területére kihatással volt. A modern társadalom számos területén képes befolyásolni az emberek gondolkodásmódját, emellett a közösségi hálókon jelen lévő tartalmak szűrésére, illetve a fogyasztókra szabott célzott hirdetésekre is kapott felhasználást. Egyre meghatározóbb szerepet tölt be mai társadalmunkban az élet legtöbb területén. Manapság már hétköznapinak nevezhető dolgok hátterében is tanuló rendszerek állnak, mint például a beszéd szöveggé alakítása vagy képeken való szövegfelismerés.

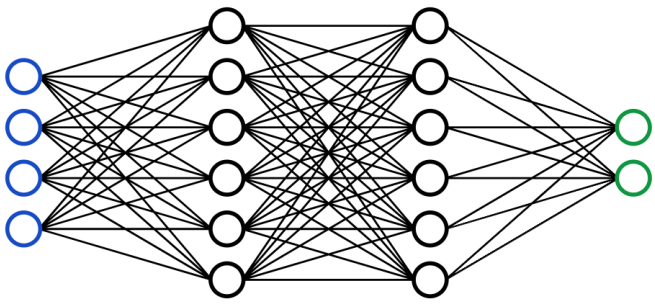
Az évek során bebizonyosodott, hogy nagyon nagy adathalmazok esetében, a komplex problémák exponenciálisan bonyolódnak, ezért az adatok közötti összefüggéseket, korrelációkat már túl időigényes lenne a hagyományos számítási modellekkel meghatározni. A mesterséges neurális hálózatok alkalmazásával ezek a problémák nagy pontossággal közelíthetők, sokkal kisebb erőforrás és- időigény mellett. A kezdetekben, amikor még nem álltak a kutatók rendelkezésére kellően jó erőforrások, a hagyományos gépi tanulási technikák csak bizonyos korlátok között voltak képesek az adatok nyers formában történő feldolgozására. Egy neurális hálózat megalkotásához kemény mérnöki munkára és kellően nagy szakértelemre volt szükség, azonban az elmúlt évek alatt végzett kutatások eredményeként, rengeteg olyan eszközt hoztak létre, amellyel ez a folyamat meglehetősen leegyszerűsödött. Ilyen iramú fejlődéshez hozzájárult az algoritmusok és a szoftveres, illetve hardveres infrastruktúrák fejlődése.

A mélytanuló algoritmusokat a neurális hálókból fejlesztették ki, melyek mostanra kezdenek elterjedté válni a számítástechnika területén. A gépi tanulással kapcsolatos kutatásokat már az 1940-es években elkezdték, azonban akkor még közel sem állt rendelkezésre megfelelő hardver ahhoz, hogy jelentőséggel bíró eredményeket tudjanak felmutatni a kutatók. A mélytanuló modellek általában hierarchikus struktúrát alkalmaznak a hálózatok rétegeinek összeköttetésére. Alsóbb rétegek kimenete, lineáris vagy nemlineáris számítások során átalakított értéke szolgálhat magasabb rétegek bemenetéül. Ezáltal a számítógépek a hasznos

jellemzőket automatikusan, közvetlenül a nyers adatokon keresztül tanulják meg, ezzel eliminálva a manuális hangolást. A fejlettebb algoritmusok már képesek alacsony szintű jellemzők, magasabb szintűvé alakításra. Ennek köszönhetően a mélytanulási modellek jóval hatékonyabbak lettek a jellemzők reprezentálásában, mint az egyszerűbb őseik.

A neurális hálók (Neural Networks - NN), más néven mesterséges neurális hálózatok (Artificial Neural Networks - ANN) vagy szimulált neurális hálózatok (Simulated Neural Networks - SNN), a gépi tanulási algoritmusok egy fajtái. Ezek az algoritmusok az emberi agyban jelen lévő neuronok, egymást közötti kommunikációs folyamatát modellezik. A neuronok, az agyunk alap építőelemei olyan idegsejtek, amelyek elektromos impulzusok továbbítására képesek, ezzel információt áramoltatva testünk minden része között. A neuronok felelősek az emberi testet érő külső impulzusok feldolgozásáért, izmaink mozgatásáért, illetve minden egyéb köztes lépésért, ami információ továbbítást igényel.

A mesterséges neurális hálózatok mesterséges neuronokból épülnek fel, amelyek összességéből rétegződést hozhatunk létre. A mesterséges neuronok, az agy működéséhez hasonlóan, információt továbbítanak a többi neuron felé. Neurális hálózatok esetében ez az információ szám értékek formájában kerül reprezentálásra. Minden neuron kapcsolatban áll, legalább egy másik neuronnal, a köztük lévő kapcsolatot éleknek hívjuk. Az egyes élek rendelkeznek hozzájuk rendelt súly értékekkel, illetve aktiválási tartománnyal. Ha bármelyik neuron értéke átlépi az aktiválási tartomány értékét, aktiválódik és adatot továbbít a hálózat következő rétege felé. Ezt hívjuk a neuronok tüzelésének. A lenti ábrán látható egy neuronok összeköttetéséből álló, két rétegű neurális háló. A két szélső réteg, a bemenet és a kimenet, nem számítanak bele a hálózat rétegeinek számosságába.



11. ábra Egy neurális háló felépítése

A neurális hálózatok tanítás útján fejleszthetők és pontosíthatók. Általában klasszifikációra, detektálásra és összefüggések megtalálására alkalmazzák őket. Ezekre a felhasználásokra, megfelelő minőségű tanítási adat, és kellő idő rendelkezésre állásával rendkívül hatékony megoldásul tudnak használni a neurális hálózatok. Jóval pontosabb és gyorsabb kiértékelés érhető el velük a hagyományos programozási megközelítésekhez viszonyítva. Megfelelő idejű és minőségű tanítás után rendkívül hasznos eszköznek bizonyulnak, ugyanis segítségükkel nagy sebességgel értékelhető ki nagy mennyiségű és/vagy komplex adat. Segítségükkel bonyolultabb kép vagy hangfelismerési feladatok, órák helyett akár percek alatt megoldhatóvá válnak, óriási számításigény nélkül. Egy széles körben ismert és használt példa a neurális hálózatokra, a Google kereső motorja, a Google Search.

Működésük

Az egyes neuron-ok lineáris regressziós modellekként írhatók le, amelyek a bemeneti adatokból, súlyokból, tetszőleges konstansból/tartományból és egy kimenetből épülnek fel. Minden egyes bemenethez súlyok tartoznak, amelyek az adott bemenetekkel összeszorzódnak, majd ezek a súlyozott értékek összeadódnak. Az így keletkezett eredmény egy aktivációs függvény bemenetéül szolgál, amely meghatározza az adott neuron végső állapotát. Az aktivációs függvény kimenete határozza meg, hogy a neuron tűzel, azaz továbbít-e adatot a következő réteg felé vagy sem. Ha egy neuron aktiválódik, akkor az értéke továbbítódik a következő kapcsolt neuronok felé. Egyes rétegek kimenetei, a következő rétegek bemeneteiként szolgálnak. Ezeket a hálózatokat előre csatolt (feedforward) neurális hálózatoknak nevezik. A hálózatok úgy 'tanulnak', hogy a tanulási iterációkon belül módosulnak a rétegeket összekötő élek súlyai, oly módon, hogy végeredményül a modell pontosabb eredményeket adjon. Ennek elérésére specifikus módszerek léteznek, amelyeket lentebb részletezek.

Gyakorlati esetekben, mint például képfelismerés, felügyelt tanítást alkalmazunk, előre felkészített adathalmazokkal. Az algoritmus tanítása közben számon kell tartanunk annak pontosságát, amit egy jóság függvénnyel tehetünk meg. A veszteség vagy költség az a szám, amely a modell predikcióinak minőségét reprezentálja. Ha a modell tökéletes lenne, akkor a veszteség értéke nulla lenne. Ez a jellemző a modell adott iteráció elvégezte utána állapotának pontosságát adja meg, ezzel visszajelzést adva a tanítás állapotáról. A veszteség kiszámítására használt függvény megvalósítására, általában az átlagos négyzethiba módszerét alkalmazzák (MSE – Mean Squared Error), melynek a képlete az alábbi ábrán látható.

$$MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \widehat{y}\right)}_{\text{The square of the difference between actual and predicted}}^{2}$$

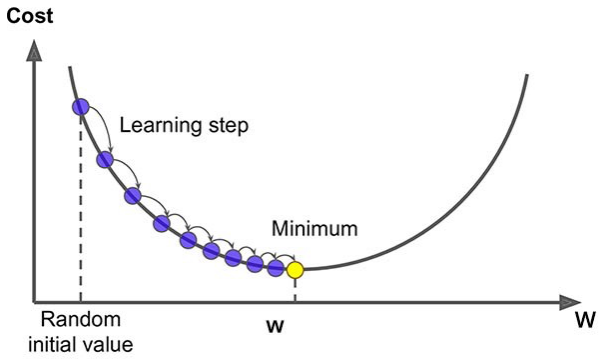
12. ábra Átlag négyzethiba képlete

A tanítás célja, a veszteség érték folyamatos csökkentése, amíg lehetséges. Az alacsonyabb érték végeredményül nagyobb pontosságot jelent. Ezt a jóság függvény minimum értékének megtalálásával érhetjük el. Egy függvény minimumának meghatározására több módszer is létezik. Ezek közül az egyik, neurális hálózatok esetében leggyakrabban használt algoritmus, a gradiens ereszkedés (gradient descent).

Az algoritmus egy randomizált pontból indul, majd a saját helyzetén található és a körülötte lévő értékeket vizsgálja, majd abba az irány lép tovább, amerre kisebb értéket talál. Ezen a lépéssorozaton addig iterál, amíg nem talál az adott pozícióban található értéknél jobbat, vagyis kisebbet. A lépések méretét is megadhatjuk, viszont ügyelnünk kell ennek a lépésköznek a helyes meghatározására. Ha túl apró lépésekben haladunk, akkor könnyen beleütközhetünk lokális minimumokba, amikből nem tud kilépni az algoritmus, vagy ha túl nagy lépésekben haladunk, akkor könnyen lehet, hogy túllépünk a minimumon és ezután már a maximum felé lépkedünk. Olyan lépés méret meghatározására van szükség, amely mindkét eset bekövetkezésének valószínűségét képes minimalizálni. A lépések megállapítását a felhasznált jóság függvény is befolyásolja. Például, ha egy olyan függvény használunk, amelynek több

lokális minimuma is lehet, érdemes nagyobb lépésközt választani, ellenkező esetben pedig kisebbet.

Ezért következtetésül levonható, hogy felépítéséből adódóan, ez a módszer hajlamos könnyen beragadni lokális minimumokba, ami a tanítás folyamatát is befolyásolhatja. Ha az algoritmus beragad, akkor megeshet, hogy a veszteséget nem tudjuk tovább csökkenteni, annak ellenére, hogy a használt súlyokon lehetne még optimalizálni. Ez a tulajdonság a módszer nagy hátrányaként jelentkezik, azonban léteznek olyan változatai is, amelyek figyelembe veszik a beragadás lehetőségét, ezáltal nagyobb eséllyel találják meg a globális minimumot, vagy legalább képesek közelebb kerülni hozzá. A könnyebb érthetőség érdekében a lenti ábrán, a fentebb részletezett működés vizualizációja látható.



13. ábra Gradient descent

A legtöbb neurális háló előre csatolt, ami azt jelenti, hogy csak egy irányba zajlik rajtuk az adatáramlás, bemenettől a kimenetig. Azonban létezik olyan módszer is, amelyen a kimenettől visszavezetve a bemenetig folyik az adattovábbítás. Ezt a technikát visszaterjesztéses tanításnak hívják. Használatával a neuronok veszteségei és ezáltal a hozzájuk tartozó súlyok értékei is pontosabban meghatározhatók. A hagyományos, előre csatolt hálózatok működését kiegészítve, az adott iterációból kiszámított veszteség függvény eredménye visszacsatolódik a következő iteráció bemenetére, ami azt jelenti, hogy minden neuron visszajelzést kap arról, hogy a hozzájuk csatolt súlyok épp mennyire pontosak vagy pontatlanok. Ezzel a módszerrel hamarabb megvalósíthatóbb az elérni kívánt optimum.

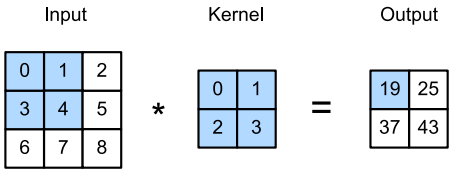
Konvolúciós neurális hálózatok [2]

A neurális hálózatokat két típusba lehet sorolni, melyek különböző célokra alkalmasak. Egyikük a fentiekben is részletezett hagyományos neurális hálók. A másik típus a konvolúciós neurális hálók (Convolutional Neural Networks - CNN). Az utóbbiak, legtöbb esetben képeken gépi látásra, objektumfelismerésre és minták beazonosítására használatosak. Ezek a hálózatok tipikusan a képfeldolgozási módszereknél használt mátrix műveleteket alkalmaznak, melyek segítenek jellemzők azonosításában, klasszifikációban, minták felismerésében digitalizált képeken. A konvolúciós hálóknak létezik egy ismétlődést felhasználó változata (Recurrent Neural Networks - RNN). Ezek az algoritmusok legfőképpen időben változó adathalmazokon alkalmazottak. Gyakori felhasználási esetek a tőzsde és sales predikciók.

A konvolúciós neurális hálózatok, olyan speciális mélytanulási algoritmusok, amelyek képek feldolgozására és kiértékelésére képesek. Ezek a hálózatok kifejezetten arra a célra lettek kifejlesztve, hogy automatikusan tanuljanak jellemzőket és azok térbeli hierarchiáját. Mindezt a visszaterjesztéses tanítás alkalmazásával, konvolúciós rétegek, pooling rétegek és teljesen összekapcsolt rétegek felhasználásával. A konvolúciós rétegek létfontosságú szerepet töltenek be a hálózatok kialakításakor, mivel ezek a rétegek végzik a jellemzők kiemelését, illetve egyéb képfeldolgozási műveletek végrehajtását.

A digitális képek pixeleinek értékei mátrixokban reprezentálhatok, ami a nagyobb méretű képek esetében több millió pixelt is jelenthet. A nyers bemenetek feldolgozása emiatt iszonyúan nagy számításigényűvé válhat, ezért egy olyan módszerre van szükség, amely segítségével kisebb méretekbe leképezhető a bemeneti kép, oly módon, hogy annak jellemzői megmaradnak. Erre a feladatra a konvolúciós rétegek használhatók fel, amelyeknek végig kell vizsgálniuk a képek részeit a jellemzők meghatározása érdekében, ezt a feladatot pedig az úgynevezett kernel funkciókkal teszik., amelyek kisebb mátrixokba képezik le a képek jellemzőit.

A kernel egy NxN-es mátrixként fogható fel, amely segítségével tetszőleges műveleteket hajthatunk végre képmátrixok adott részein, melyek kimeneteiből egy jellemző mátrix állítható elő eredményül (ld.: 13. ábra). Ez azért bizonyul nagyon hatékonynak, mert így pixelek egy csoportját együtt tudjuk kezelni, ahelyett, hogy mindegyiket egyesével dolgoznánk fel.



13. ábra Konvolúciós művelet

Ez a tulajdonság a jellemzők kinyerésénél is jelentős előnyt hoz. Ahogy a bizonyos rétegek egymásba továbbítják a kiértékelt műveletek eredményeit, csoportosított jellemzőit, ezáltal a hálózat elemei fokozatosan egyre komplexebbé vállnak a hierarchiában haladva. Hagyományos esetben, a képek manuális előfeldolgozásánál a bemenetek skálázása, filterezése és egyéb műveletek végrehajtása kézzel történik, ezeket a lépéseket azonban a konvolúciós hálók képesek megtanulni. Ezzel a neurális hálózatokba bemenetként vezetett képeken kisebb mennyiségű előkészítő lépés végrehajtására van szükség.

Következő lépésként a konvolúció eredményeként kapott jellemzőtérképek aktivációs függvényeken kerülnek átvezetésre, amelyek meghatározzák, hogy milyen adatok fognak továbbítódni a következő rétegek felé. Az aktivációs függvények feladata a súlyozott neuron értékek továbbítása egy következő réteg felé. Erre a funkcionalitásra alacsony komplexitású és számításigényű megoldásokat szoktak alkalmazni. A legegyszerűbb aktivációs függvények lineárisak, ezeknél semmilyen transzformáció nem történik a neuronok között. Az ilyen aktivációs függvényekre épülő hálózatok nagyon egyszerű taníthatósággal rendelkeznek, azonban komplexebb jellemzők megtanulására nem képesek. Emiatt a nem lineáris aktivációs függvények használata preferált, mivel ezek lehetővé teszik, hogy a neuronok összetetteb adatstruktúrák megtanulására is képesek legyenek. Két gyakran használt nem lineáris függvény a sigmoid és a tangens hiperbolikusz. A sigmoid azon az elven működik, hogy az egynél nagyobb vagy hozzá közeli értékekhez egyet, a nullánál kisebb vagy ahhoz közel lévő

értékekhez pedig nullát rendel. Egészen a kilencvenes évekig ez számított a neurális hálók esetében az alapértelmezetten használt aktivációs függvénynek. A tangens hiperbolikusz egy hasonló elven működő függvény, annyi különbséggel, hogy mínusz egy és egy közötti értékekkel tér vissza. A tangenst a kilencvenes évek végén és a kétezres évek elején kezdték el alkalmazni a sigmoid helyett, mivel felhasználása pontosabb predikciókat eredményezett.

A két említett módszer hátránya, hogy túlszaturált értékeket adnak eredményül. Ez azt jelenti, hogy a nagy értékek a felső határhoz, egyhez, míg a kis értékek az alsó küszöbhöz, mínusz egyhez vagy nullához tartanak. Ebből adódóan csak a középértékeikhez közelítő változásokra érzékenyek, ami ahhoz vezet, hogy olyan neuronok is tüzelésre hajlamosak, amelyek nem tartoznak hasznos információt. Ez pedig azt eredményezi, hogy a mélyebb neurális hálók, egy ponton túl nem taníthatók tovább. Ennek a problémának, a szakirodalomban használt neve az „elhalványuló gradiens” (vanishing gradient), ami onnan kapta a nevét, hogy a nem lineáris aktivációs függvényeket használó hálózatokba visszavezetett hiba, gyorsan olyan apróvá válik, hogy a gradiens ereszkedésnél használt gradiens majdnem eltűnik, ezért ezen a ponton túl nem tanítható tovább a haló, mert nem találunk olyan pontot a térben, a konfigurált lépésközzel, amely jobb jósággal rendelkezne.

Ahhoz, hogy megfelelő hatékonysággal tudjuk tanítani hálózatainkat, olyan nem lineáris aktivációs függvény kiválasztására van szükség, amely úgy néz ki és úgy viselkedik, mint egy lineáris függvény. Ha ilyen függvényt választunk, akkor a hálózat jól taníthatóvá válik és komplex jellemzők hatékony tanulására is alkalmassá válik. Manapság az iparban legelterjedtebb, az említett tulajdonságokkal rendelkező függvény, a rektifikált lineáris aktivációs függvény (ReL – Rectified Linear). Azokat a neuronokat, amelyek ezt a megoldást alkalmazzák, rektifikált lineáris egységeknek (ReLU – Rectifical Linear Unit) nevezzük. A használt algoritmus rendkívül egyszerűen leírható. Ha a bemeneti érték kisebb, mint nulla, akkor a visszatérési érték nulla, különben a bemeneti érték, önmaga. Ez megvalósítható egy ’if’ vezérlési szerkezet használatával a 14. ábrán látható módon.

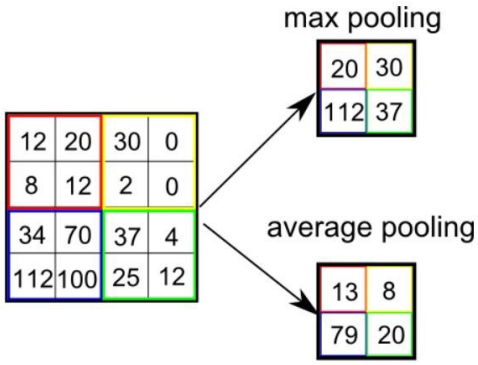
```
ReLU(x) {
    if (x < 0) {
        return 0;
    } else {
        return x;
    }
}
```

14. ábra A ReLU egy lehetséges implementációja

Azonban használhatunk egy jóval egyszerűbb megoldást is, amely a **f(x) = max(0, x)** képlettel írható le. Általánosságban elmondható a függvényről, hogy javítja a predikciók minőségét, sőt az egyik legfontosabb tényezőként jelentkezik egy neurális hálózat tanítását tekintve [3]. Konvolúciós neurális hálózatok esetében pedig kifejezetten ajánlott őket használni. Mindezek mellett a ReLU-nak is vannak hátrányai, működéséből adódóan. Ha például folyamatosan olyan értékek kerülnek a függvény bemenetére, amelyek negatívak, akkor az aktivációs függvény mindig nullával tér vissza. Ez azt eredményezi, hogy az érintett neuron soha nem fog aktiválódni. A nem aktiválódott neuronok súlyai nem kerülnek további módosításra, ami végeredményben rossz irányba befolyásolja a hálózat betanítását. Ennek kiküszöbölése érdekében bevezethetünk olyan megoldásokat, amelyek kisebb negatív számokat

engedélyeznek, vagy transzformációk segítségével pozitív értékeket negatívokká alakítanak, ami miatt a gradiens hamarabb nullához húz, ezzel gyorsítva a tanítást.

A konvolúciós műveletekből kisebb méretű mátrixokat kapunk, azonban ezek méretét tovább csökkenthetjük a pooling rétegek segítségével, amelyek nem-lineáris lemintavételezést látnak el, ami a bemenet kompresszióját jelenti. Ezek a rétegek a konvolúciós mátrixokból, szintén a mátrix elemeinek csoportos kiértékeléséből állít elő egy újabb értékhalmazt. Ez végrehajtható átlagon vagy maximumon alapuló kiértékeléssel, ahogy ezt a 15. ábra is szemlélteti. Ezeket a lépéseket viszont csak addig érdemes ismételni, amíg elég jellemző információnk marad az adott képről. Kellően sok iteráció túl nagy adatvesztést eredményezhet.



15. ábra Lehetséges pooling módszerek

A kép jellemzőinek kinyerése után történik meg ezeknek klasszifikációja. Ezt a feladatot a teljesen összekapcsolt rétegek látják el. Ezt a lépést általában egy hagyományos értelemben vett neurális háló hajtja végre, visszaterjesztéses tanulás alkalmazásával. Ennek a hálónak bemenetül, a konvolúciós és pooling rétegeken végigvezetett kép kilapított leképezését adjuk meg, amelyen már képes alkalmazni a hagyományos módszereket. Az eredménymátrix kilapítása, annak oszlopvektorrá alakítását jelenti.

Épület detektálás légifelvételek alapján, állandó színek és árnyékinformáció alapján. [2]

A képek sokoldalúságából adódóan ez a módszer több, jól elválasztható részből épül fel. Kiindulópontnak a tetők felismerése szolgál. A kutatásban szereplő környéken a leggyakoribb háztető szín a vörös/narancssárga, ezért az első szegmens erre alapoz.

A képeket RGB formátumban kezelve és csak bizonyos színcsatornákra fókuszálva, egyszerűen elkülöníthetővé válnak a kép fontos részei. Erre a szerzők egy korábbi tanulmányból felhasznált módszert alkalmaznak, ami az R(vörös) és G(zöld) színcsatornák segítségével emeli ki a piros területeket. Ugyanezt a módszert alkalmazva az árnyékok elkülönítése is automatikusan elvégezhető a B(kék) és G(zöld) színcsatornák használatával.

Abban az esetben, ha az épületek nem vörös tetővel rendelkeznek, a kép többi tulajdonságából következtethetünk az épületek elhelyezkedésére, méretére és orientációjára. Korábbi kutatások alatt manuálisan határozták meg a megvilágítás irányát, pontosabban szögét. Itt az épülethez tartozó és árnyék terület középpontjának koordinátái segítségével megállapítható a képen fellelhető objektumok megvilágításának szöge. A szög pontosításához felhasználhatók a már

korábban meghatározott pontok paraméterei. Ha feltételezzük, hogy az épület a megvilágítás vektorral ellentétes irányban található, akkor becslést adhatunk az épület középpontjának helyére. A feltételezett középpont köré helyezünk egy 30 x 30 méretű négyzetet. Az így kapott terület az épület közelítőleges méretét adja meg.

A legtöbb épület szögletes formákkal rendelkezik. Ebben a tanulmányban egy újfajta illesztési metódust mutatnak be a kutatók. A módszer alapjaiban az élek elhelyezkedésére épít, ezen éleket pedig a *Canny edge detection* módszerrel határozza meg. A problémára létrehozott algoritmus, a megtalált élekből konstruál egy téglalapot, ami az épületet reprezentálja. Ez a módszer bármilyen színű tetőn alkalmazható, így nem áll függésben az adott területen jellemző tetőszínnel.

Értékelés:

Összességében egy jól hasznosítható módszer. A kutatásban állított valós és fals pozitív találatok aránya kiemelkedően jó. Néhány esetben a képen található zaj és egyéb befolyásoló tényezők miatt fák csoportjait is épületnek jelölte

Legfőbb előnyei: A megvilágítás irányát, az épületet és annak árnyékát alapjául vevő algoritmus, ami képes megbecsülni egy épület elhelyezkedését, illetve a szerzők által bevezetett új ’bedobozoló’ algoritmus, amely az élek elhelyezkedése alapján működik.

Épületek alaprajzának kinyerése multispektrális, űrből készült felvételekből, strukturálisan optimalizált U-Net konvolúciós neurális hálózat segítségével. [3]

Ez a kutatás a 2018-as Michael hurrikán sújtotta terület kárainak felmérésére készült. A megoldás már meglévő eredményekre és technikákra épít. A fenti kutatás eredményei is felhasználásra kerültek, így az ott részletezett algoritmusok, leírások erre a részre is igazak. Korábban, a neurális hálózatok nagy teljesítményigénye és a teszteléshez szükséges nagy méretű adathalmazok hiánya miatt, a kutatóknak nem volt lehetőségük egy ehhez hasonló megoldás létrehozására. Azonban a számítástudomány folyamatos fejlődésének köszönhetően ez is egy opcióvá nőtte ki magát. Az évente megrendezésre kerülő Spacenet Challanges néven futó kihívásoknak köszönhetően, rendkívül nagy mennyiségű és nagyon magas minőségű teszt adat áll rendelkezésünkre. A szabadon felhasználható adatokat ebben a specifikus esetben is hasznosítani tudták.

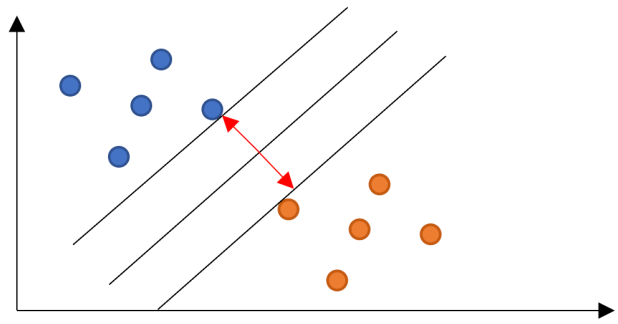
Az előző kutatáshoz hasonló megoldás szerepel itt is, viszont már csak azért is érdemes megemlíteni, mert jól szemlélteti, hogy mennyi felderítetlen lehetőség rejlik még ebben a szakterületben.

R-CNN [4]

A gépi látás terjedésével egyre nagyobb figyelem alá került az objektumdetektálás témaköre, azonban 2014-ig nem állt rendelkezésre egy szimpla, jól skálázható megoldás. Az addigi próbálkozások nagy komplexitású -és számításigényűek voltak. Ennek az volt az oka, hogy a detektálás szempontjából fontos régiók meghatározása nagyon költségesnek bizonyult. A képeken külön orientációval, mérettel és elhelyezkedéssel találhatók objektumok, emiatt nagyon nagy mennyiségű, különböző méretű régió meghatározására és kiértékelésére van szükség.

Ezt a problémát hivatott orvosolni az R-CNN (Regions with CNN). Ez a módszer a hasznos régiók meghatározására nyújt egy gyors megoldást. A rengeteg régió helyett ~2000 darabot választ ki és ezeken végzi el a jellemzők kiválasztását egy konvolúciós neurális háló segítségével. A régiók kiválasztása 'Szelektív Keresés' [5] módszerével történik, ahol először sok kisebb régió kerül meghatározásra, majd a hasonlók egy köztes lépésben egy mohó algoritmus segítségével összevonásra kerülnek, végsősorban pedig a kiválasztott régiók négyzetbe torzítva kerülnek továbbításra a jellemzők kiértékeléséhez. A jellemzők detektálására SVM (Support Vector Machine) -ek kerülnek felhasználásra.

Az SVM egy olyan klasszifikációs módszer, amely a régiók közti legjobb határvonal megtalálására ad megoldást (ld.: 16. ábra). Az a határvonal a legjobb, amelyik a legnagyobb margóval (a határvonal és a régiók közti távolság) rendelkezik.



16. ábra SVM működése

A detektálás után az SVM-ek segítségével, határoló dobozok kerülnek az objektumok köré.

Mindezek ellenére, az R-CNN-nek az alábbi hátrányai fellelhetők:

- Valós időben nem használható, mert a képek kiértékelése így is közel 1 percig tart.
- Rengeteg időbe telik a hálózat tanítása, mivel képenként még mindig közel 2000 régió átvizsgálására van szükség, illetve a köztes állapotok lemezre kerülnek kiírásra, melynek olvasási és írási sebessége jóval lassabb, mint a memóriáé.
- A Szelektív keresés egy statikus algoritmus, ezért semmilyen módon nem fejlődik a tanítás során, emiatt nagy valószínűséggel adhat rossz régió javaslatokat.

Fast R-CNN [6]

Az R-CNN már nagy előrelépésnek volt betudható, azonban valós idejű használathoz még mindig túl lassúnak és számításigényesnek bizonyult, nem beszélve a tanításhoz felhasznált időről és annak erőforrásigényességéről. Akkoriban erősnek számító grafikai kártyákkal is több napba és több száz gigabájtnyi felhasznált tárhelybe került egy tanítási folyamat. A Fast R-CNN célja, hogy pontosabb, gyorsabb és kevésbé erőforrásigényes megoldást nyújtson. Az előző generációs módszerhez hasonlóan itt is fontos szerepe van a számunkra érdekes régiók meghatározásának, azonban ez egy sokkal hatékonyabb módon történik.

A konvolúciós neurális hálózat bemenete az eredeti kép, amiből kimenetként egy jellemző térképek kapunk. Ezt a kimenetet a következő réteg szintén Szelektív Keresés alkalmazásával tovább alakítja, majd egy régió javasló (RoI – Region of Interest) réteg javaslatokat tesz azokra a területekre, ahol nagy valószínűséggel találhatók objektumok. A fontos területek megtalálása után egy Softmax aktivációs réteg segítségével történik az objektumok osztályának meghatározása. A Softmax függvény célja az, hogy a neurális háló kimenetét normalizálja, vagyis nulla és egy közé szorítsa. Ezek a normalizált értékek lesznek a különböző osztályokhoz tartozó valószínűségek.Végső körben pedig határolódobozok kerülnek az objektumok köré.

Ez az átgondolt folyamat azért jelent hatalmas teljesítménybeli ugrást, mert a konvolúciós neurális hálónak nem kell mind a ~2000 régiójavaslaton végig iterálnia, hanem elég egyszer, az eredeti bemeneti képen. Ebből adódóan a tanítási idő nagyjából 10x-es, a kiértékelés 20x-os gyorsulást érhet el.

A rendkívüli gyorsulás arra is fényt derített, hogy a további gyorsulás elérését megakadályozó szűk keresztmetszetet a régiók felmérése rejti.

Faster R-CNN [7]

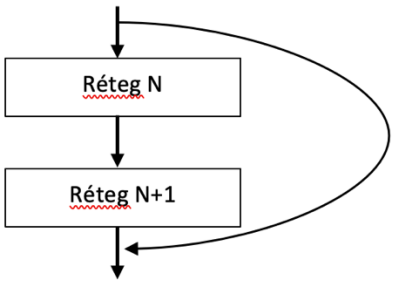
A Fast R-CNN már jelentős javulást mutatott előző generációjához képest, azonban a régiók hatékony ajánlása még megoldatlan problémának bizonyult. A problémát maga a régió kereső módszer, a Szelektív Keresés okozta. Ez az algoritmus már túl időigényes a közel valós idejű működés eléréséhez.

Így, hogy már világossá vált, mi a Fast R-CNN gyenge láncszeme, a következő a cél egy olyan régió ajánló réteg létrehozása volt, amely képes minimális veszteséggel működni. Kísérletezések azt az eredmény mutatták, hogy erre a célra tanított neurális hálókkal jóval gyorsabban, szintén pontos eredményeket lehet elérni. Ezeket a hálókat RPN (Region Proposal Network), azaz régió javasló neurális hálózatoknak nevezik.

Ezzel az új megközelítéssel akár már 100-200x -os gyorsulás érhető el, lehetővé téve a közel valós idejű objektum detektálást. A cikkben szereplő esetben már 5 fps (frames per second – képkocka/másodperc) gyorsaságú videófelvételen is tudtak objektumokat detektálni.

ResNet [8]

A ResNet-ek, másnéven Residual Neural Networks, magyarul Maradékos Neurális Hálózatok, olyan neurális hálózatok, amelyek a mély rétegek által okozott degradálódást hivatottak orvosolni. Nagy mélységű előre csatolt hálózatok esetén, egy idő után észrevehető egy feltűnő pontosság béli romlás, amelyet a rétegek közötti transzformációk okoznak. Erre a mellékhatásra megoldásul, olyan rétegek bevezetése szolgál, amelyek aktiválódásakor nem a közvetlen következő rétegre lépünk tovább, hanem egy olyanra, amely kihagy néhány köztes réteget. Ez jóval pontosabb és mélyebb hálók létrehozását teszi lehetővé. A rétegek között léptetés a 17. ábrán látható.



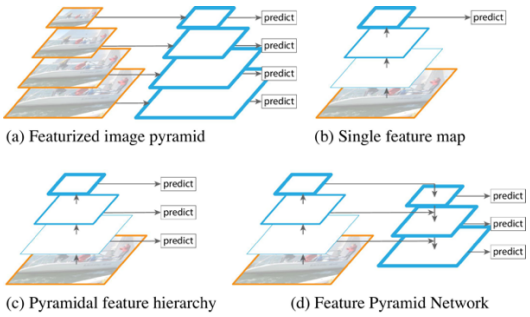
17. ábra ResNet építő blokk

Mask R-CNN [9]

A Mask R-CNN a Faster R-CNN egy olyan kiterjesztése, mely felismert objektumokhoz maszkolást is társít. Működésében két fázisra bontható. Az elsőben javaslatokat generál azokról a területekről, ahol objektumok lehetnek, majd a második fázisban ezen javaslatok alapján végzi a predikciót, osztályozást, keretezést és maszkolást. Mindkét fázis a 'gerinc (backbone)' struktúrához kapcsolódik.

A backbone egy FPN (Feature Pyramid Network – Jellemző Piramus Hálózat, ld.: 18.ábra) és egy ResNet-ből áll. A ResnNet, másnéven Residual Neural Network, azaz Maradékos Neurális Hálózat, egy olyan fajta neurális hálózat, amelyben a rétegek között engedélyezett a kihagyásos tovább ugrás. Ez csupán annyit tesz, hogy ha egy olyan réteg aktiválódik, amely egy ugrást hajt végre, akkor ez az ugrás kihagy néhány köztes réteget, ezzel orvosolva a „halványuló gradiens problémát".

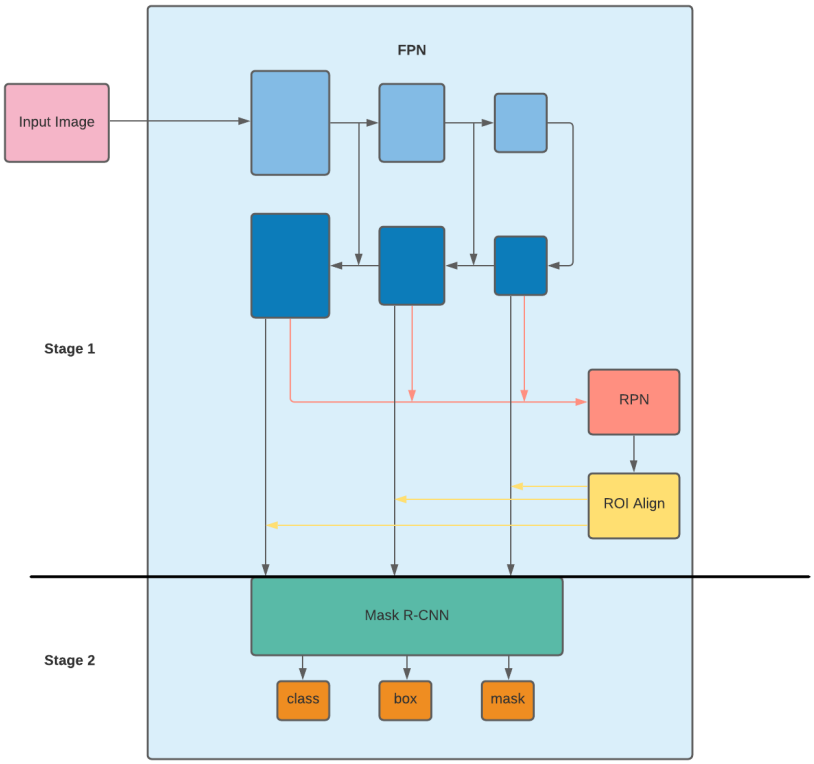
Az FPN bizonyos jellemzők kinyerésére alkalmas módszer. Szerkezete lentről-felfelé, fentről-lefelé és oldalsó irányú kapcsolatokból épül fel. Az alap koncepciója a bemeneti kép különböző méretű kiértékelésére, és ezek összehangolására épül.



18. ábra Feature Pyramid Network

Az első fázisban egy RPN (Region Proposal Network – Régió Javasló Neurális Háló) neurális háló végig szkenneli a fentről-lefelé tartó irányt és javaslatokat készít azokról a területekről, ahol objektumok lehetnek. Ahhoz, hogy az eredeti képen jelezni tudjuk az objektumok feltételezett helyét, úgynevezett horgonyokra van szükség. Ezek a horgonyok igazából dobozok, előre meghatározott mérettel és elhelyezkedéssel a képen. Ezután az RPN határoló dobozokat rendel a horgonyokhoz, ezzel nagyjából meghatározva az objektumok feltételezett helyét. Azért nagyon hatékony ez a módszer, mert a jellemzők elhelyezkedése nem vész el a fel -és leméretezés során, mivel a dobozok is mérethez relatívan változnak.

A második fázisban a javasolt területek alapján egy másik neurális háló szintén végig szkenneli a feltételezett objektumokat, osztályozza, bekeretezi és maszkolja őket. Ezt azonban a meglévő horgonyok nélkül teszi, ROIAlign (Region of Interest Align) nevű módszer segítségével, ami a hasznos területek pontosítását végzi. A Mask R-CNN architektúrális felépítése a 19. ábrán látható.



19. ábra Mask R-CNN architektúrájának illusztrációja

2.4.1. Technológiák kiválasztása

Python

A Python az interpretált nyelvek családjába tartozik. Ez annyit jelent, hogy a programkód, nem futás előtt kerül lefordításra gépi kóddá, hanem minden egyes sor futásidőben értékelődik ki. Ennek a tulajdonságának előnyei és hátrányai is vannak. Az előre fordított nyelveknél jobb teljesítmény érhető el, illetve a fejlesztő szabadabb kezet kap a hardveres erőforrásokhoz (memória menedzsment, CPU használat). Az előre fordítottság előnye egyben hátrány is, mert a fordítás extra időt igényel, azonban interpretált nyelveknél ez a folyamat nincs jelen. Előre fordított nyelveknél a szintakszis helyességének validálása egyszerűbb, mert már a fordítás során hibát okoznak a rossz szintakszissal írt kódrészek. Az alacsonyabb teljesítmény ellenére nagy előnye az interpretált nyelveknek, hogy a megírt programon újra fordítás nélkül lehet módosítani.

Az interpretált nyelvek teljesítménybeli lemaradását egyre kisebbre sikerült csökkenteni az évek során. Alap esetben a sorok azonnal kiértékelésre kerülnének, de a 'Just in Time Compilation' módszer segítségével kiértékelés előtt még gépi kóddá fordulnak, ezzel ötvözve két típus működését. Dinamikusan típusos, azaz kódírás közben nem kell a változókhoz típust rendelni, azok futásidőben értékelődnek ki.

A nyelv fejlesztését Guido van Rossum kezdte el az 1980-as években. Fő szempontjai közé tartozik az olvashatóság és strukturáltság. Több programozási paradigmát is támogat, többek közt: funkcionális és objektum-orientált programozás. Még a mai napig folyamatos fejlesztés alatt van, és jelenleg a 3-as verziónál tart a projekt.

Nagyon gyakran alkalmazott az adattudósok és gépi tanulással foglalkozók körében. Nem is véletlen, hogy ezt a nyelvet választottam az épületdetektáló modul megalkotásánál. Nagy előnye, hogy rengeteg gépi tanulással, illetve adatelemzéssel kapcsolatos segédkönyvtár és keretrendszer létezik hozzá, az egyre csak növekvő közösségének köszönhetően.

TensorFlow

Az egyik legelterjedtebb nyílt forráskódú gépi tanulási keretrendszer, amelyet a Google Brain csapat fejlesztett ki, annak érdekében, hogy felgyorsítsák a területen végzett kutatásokat. Legtöbb termékükben egyébként is alkalmaznak gépi tanulást, pl.: Google Search Engine javaslatai, Google Translate javaslatai stb., így elég nagy hangsúly került a projektre. A TensorFlow segítségével egyszerűbb a neurális hálok létrehozása és tanítása. Előre tanított modellkészlettel is rendelkezik, amely jó kiindulópontként szolgálhat egy hozzá nem értőnek.

Nagy előnye, hogy a fejlesztőnek nem kell a használni kívánt algoritmusok implementálásával foglalkoznia, mert ezeket a háttérben megoldja a TensorFlow. Így egyszerűbb megalkotni a fő logikát, ami miatt a fejlesztés menete jelentős mértékben egyszerűsödik és gyorsul. Emellett fontos, hogy egy Python könyvtár, így ez még egy ok amiért ezt a nyelvet alkalmazom.

Dedikált chipeket is fejlesztettek hozzá, amiket TPU (Tensor Processing Unit) -nak hívnak. Ezek az egységek kifejezetten a keretrendszer által végzett műveletekre lettek specializálva. Évek elteltével be is bizonyosodott, hogy ezek a speciális chip-ek jobb teljesítményt nyújtanak, mint hagyományos társaik.

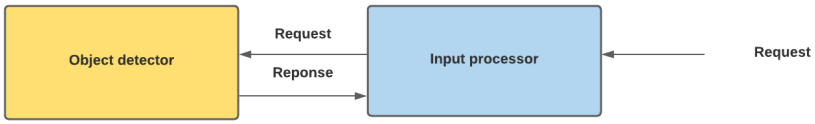
Google Colaboratory Pro (Colab)

Egy ingyenes, felhő alapú Jupyter Notebook fejlesztői környezet, amely lehetővé teszi Python kód írását és futtatását böngészőből. A Jupyter Notebook egy nyílt forráskódú webalkalmazás, amellyel létrehozhatók olyan dokumentumok, amik egyszerre tartalmazhatnak futtatható programkódot, egyenleteket, vizualizációkat és szöveget.

Nincs szükség hozzá előzetes konfigurációra, és még a felhasználni kívánt erőforrásainkat is kiválaszthatjuk. Különösen fontos kiemelni, hogy CPU-t, GPU-t és TPU-t is használhatunk számítás intenzív feladatokra. Az ingyenes verzió viszonylag szigorú erőforrás megszorításokkal használható, viszont rendelkezésre áll egy Pro verzió is, amivel még több erőforrást használhatunk, így hosszabb ideig tartó/nagyobb számításigényű műveleteket is végrehajthatunk.

2.5. Kommunikáció a backend komponensek között

A bemenetet feldolgozó és az objektumdetektáló alkalmazások között, hasonlóan a frontend-backend kommunikációhoz, lehetne REST alapú adatcserét alkalmazni. Ha ezt módszert alkalmaznám, akkor szigorúan kérés-válasz típusú kommunikáció alakulna ki, ahogy az a 20-as ábrán látható.



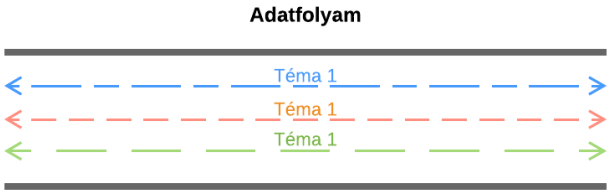
20. ábra Kérés-válasz alapú kommunikáció

Azonban, ha egy másik alkalmazásnak szüksége lenne akár a bemeneti adatra, vagy az eredményül kapott képre, akkor ezt külön-külön az adott alkalmazástól kellene elkérnie. Ez mindenképp extra implementációt von maga után mindkét alkalmazás esetében. Sokkal praktikusabb lenne egy olyan adatfolyamot használni, amibe dinamikusan más felek is becsatlakozhatnak, ezzel elkerülve a redundáns, és később komplikációkat okozható kódbázis kialakítását. Erre egy kiváló megoldás lehet az Apache Kafka.

Apache Kafka

Az Apache Kafka, egy eredetileg a LinkedIn mérnökei által fejlesztett, elosztott, alacsony késleltetésű, magas átvitellel rendelkező adatfolyam (streaming) alapú üzenettovábbítás megvalósítását lehetővé tévő platform. Rengeteg cég által használt újonnan használt eszköz. Felhasználási esetei közé tartozik például az adatgyűjtés és feldolgozás, nagyméretű elosztott rendszerek összeköttetése, analitika és még sok más.

A hagyományos értelemben vett relációs adatbázisokkal szemben, a Kafka nem entitásokhoz vagy objektumokhoz tartozó státuszok elvén, hanem bizonyos események bekövetkezésére alapulva működik. Ezért a Kafka-t esemény vezéreltnek nevezzük. A rendszer által használt adatstruktúra egy olyan adatfolyammal írható le, amelyben az üzenetek egymástól szeparálva, kategorizálva közlekednek. Az egyes kategóriákat 'Témáknak' (Topic) nevezik, és konkrétan ezek tartalmazzák az üzeneteket (ld.: 21. ábra).



21. ábra Adatfolyam szemléltetése

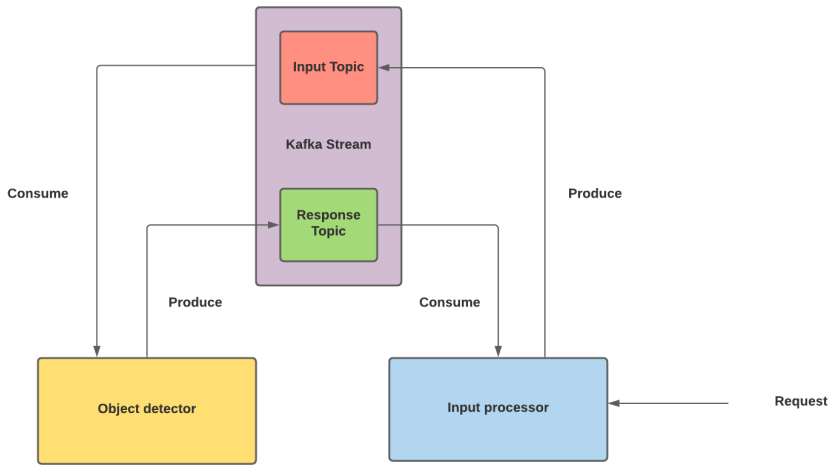
Ebből adódóan, nincs szükség arra, hogy az adatfolyamban a számunkra fontos információk után keressünk, mivel képesek vagyunk ezeket jól elkülönítetten kezelni. Minden esemény a beérkezése sorrendjében kerül feldolgozásra, ami azt jelenti, hogy az egyes témák a FIFO (First In First Out) elvet követik. A FIFO azt jelenti, hogy a folyamba elsőnek bekerülő adat fog elsőnek kikerülni. Ahhoz, hogy a létrehozott témákba adatok felé adatot továbbítsunk, vagy épp adatok olvassunk belőlük, akkor erre specializált eszközökre van szükség. Ezek a termelők (Producer), amelyek segítségével adat vihető fel a témákba, illetve a fogyasztók (Consumer), melyek a témákban lévő információ kiolvasására képesek. Ami miatt rendkívül jó adattovábbító eszköznek bizonyul a Kafka, az az, hogy egyszerre több termelő és fogyasztó is rákapcsolódhat a témákra, melyek ezáltal párhuzamosan írhatóvá és olvashatóvá válnak. A termelő és fogyasztó szerepekre nem vonatkozik szabály, ezért egy alkalmazás felveheti egyszerre mindkét szerepkört.

Nagy rendszerek fejlesztésénél gyakran fordul elő olyan eset, amikor a rendelkezésre álló adatokat több belső szolgáltatás vagy adattárház felé kell továbbítani. Például, ha arra van szükség, hogy egy műveletsorozat végeredményét több adatbázisba is perzisztáljuk, akkor ezt a lépést mindegyik irányba külön végre kéne hajtani, az adott szolgáltatással létrehozott kapcsolaton keresztül. Ennek az a nagy hátránya, hogy hosszútávon egyre több felelősség kerül a forrás alkalmazásra, mert minden egyes adattároló felé külön kell továbbítania az adatokat. Ez a probléma azonban feloldható, ha az információt szolgáltató alkalmazás, egy adatfolyamra helyezi fel a szükséges adatokat, amiket a különböző adatbázisok, fogyasztók módjára képesek feldolgozni. Ez a megközelítés azért optimális, mert így az adatok forrásaként szolgáló alkalmazásnak már nem tartozik a felelősségi körébe, hogy a külső szolgáltatások, hogyan kapják meg és dolgozzák fel az üzeneteket. Innentől kezdve ez a Kafka-ra, illetve az érintett

szolgáltatásokra helyeződik át. A Kafka Connect keretrendszer ezt a funkcionalitást valósítja meg, úgynevezett 'Csatlakozók' (Connector) segítségével. A Kafka Connect egy külön futtatható szolgáltatás, amelyet egy REST API-n keresztül lehet elérni és konfigurálni. A bizonyos adatforrások eléréséhez, specifikus csatlakozók szükségesek, melyek konfigurációkként foghatók fel. A Kafka Connect ezeket a konfigurációkat felhasználva képes az adatfolyam és a külső szolgáltatások között kapcsolatot létesíteni. A legelterjedtebb adattároló szolgáltatásokhoz már több kész csatlakozó implementáció is létezik, azonban, ha ezek nem megfelelőek számunkra, akkor egy saját megvalósítás elkészítésére is van lehetőségünk. Ha mindezek ellenére úgy döntünk, hogy nem kívánjuk alkalmazni a Kafka Connect-et, akkor egy ezzel megegyező funkcionalitást is implementálhatunk, a Kafka-ban alapértelmezetten elérhető termelők és fogyasztók felhasználásával.

A témákban áramló adatok perzisztálására is nyújt megoldást a Kafka, skálázható és replikálható formában. Az adatok átmeneti tárolása azért nagyon hasznos funkció, mert így, ha bizonyos üzenetek már kikerültek a folyamból, azonban a kiolvasó alkalmazás valami okból összeomlik, akkor a kiolvasott adatok nem vesznek el. Miután az érintett alkalmazás újra elérhetővé válik, a hiba bekövetkezésénél elveszett adatok ismét kiolvashatók a folyamból, ezzel megőrizve a rendszer konzisztenciáját. Ilyen esetekben az adatok tárolására érdemes egy lejárati időt is megadni. Ezzel elkerülhetjük a tárolt adatok manuális eltávolítását.

Az én felhasználási esetemben is ki lehetne használni a Kafka nyújtotta lehetőségeket. Az alkalmazás részeinek elválasztásakor ügyelni kell a felelősségek azonos szétválasztására is. Ha az szolgáltatások Kafka adatfolyamokon keresztül kommunikálnának, akkor ez a felelősség mindkét oldalról áthelyezhető a kommunikációs csatornára. Ha pedig más alkalmazások vagy adatbázisok felé is szükségessé válna az üzenetek továbbítása, akkor ez könnyen megoldható lenne a meglévő kódbázis módosítása nélkül. Kizárólag az új szereplőknek kéne feliratkozniuk a folyamra, vagy épp a Kafka Connect-hez kéne hozzáadni egy új csatlakozó konfigurációt. Ezek miatt a Kafka egy rendkívül erős, konzisztens és jól skálázható megoldásnak bizonyul. Az általam felvázolt backend komponensek közötti kommunikáció, a 22. ábrán látható módon valósulhatna meg.



22. ábra Kafka felhasználása alkalmazások közötti kommunikációra

Ebben a megoldásban bemenetet feldolgozó alkalmazás ír az ’Input Topic’-ba és feliratkozik a ’Response Topic’-ra. A képfeldolgozó alkalmazás pedig ugyanezt az elvet követi, csak értelemszerűen fordítva. Jól látszik, hogy ez a módszer teljesen nyitott a bővítésre, így, ha szükség lenne egy új alkalmazásra, mely becsatlakozik az adatfolyamba akkor csak fel kell iratkoznia a megfelelő témákra. Illetve, ha egy adatbázis bevezetésére lenne szükség, akkor a Kafka Connect segítségével ez is könnyen megoldható, ahogy ezt fentebb is részleteztem.

3. Megvalósítási terv

Az irodalomkutatás kiértékelésének eredményeként, az alábbi megvalósítási terv született meg:

3.1. Frontend

A felhasználói felületet egy Google Chrome Extension keretein belül valósítom meg, ahol a felhasználó böngészés közben bármilyen cím megadására képes. Válaszként információt kap az adott területen elhelyezkedő épületek számáról, a terület beépítettségi arányáról, és ezt az arányt leíró értékelésről. Az értékek vizualizálására egy diagramot is megjelenítenék, ezzel segítve az adatok érthetőbbé tételét.

Használt technológiák:

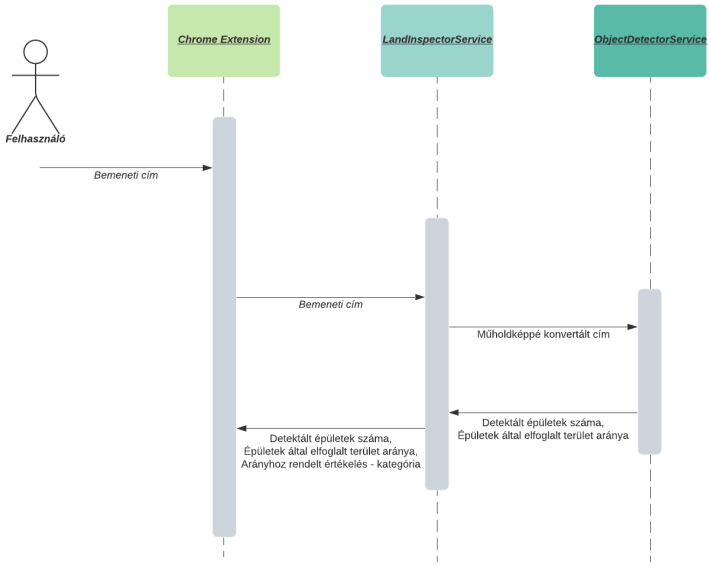
HTML, CSS, Javascript

3.2. Backend

A backend két alkalmazásból áll össze, melyből csak az egyikkel kommunikál a frontend. Ez az alkalmazás végzi el a bemeneti cím műholdképpé konvertálását, majd ezt a képet továbbítja az objektumdetektáló szolgáltatás felé.

LandInspectorService

Egy Spring Boot keretrendszerre épülő Java alkalmazás, amely egy REST API-t nyújt a frontend felé. A bemeneti címek képpé konvertálását a Google Maps Static API-val segítségével hatja végre, majd ezt a képet továbbítja az objektumdetektáló alkalmazás felé, egy Kafka adatfolyam témán keresztül. Az műholdkép egy bájt tömb formájában kerülne fel az adatfolyamra az egyszerűbb feldolgozhatóság érdekében. A válasz is egy adatfolyamon keresztül érkezik vissza, amire feliratkozik ez a modul. A leírt folyamat a 23. ábrán látható módon hajtódik végre.



23. ábra Folyamatábra a rendszer működéséről

A felhasználói felület számára szolgáltatott interfész, egy HTTP GET metódussal elérhető végpont, amelynek URL paraméteren keresztül kerül átadásra a felhasználó által megadott cím. Mivel ez egy nyilvánosan elérhető végpont lenne, ezért szükséges valamilyen autentikációs megoldás alkalmazása. Felhasználókezelést még nem tartalmazna a szolgáltatás, ezért a frontend és a backend között egy egyszerűbb, token alapú megoldást választok. Ezzel már biztosítható az, hogy csak az autorizált felek érjék el az alkalmazást. Erre a JWT (JSON Web Token) technológiát alkalmazom. A token a HTTP kérés fejlécében kerül továbbításra.

REST végpont: **GET**, LandInspectorService/land-building-ratio/{address}

Lehetséges hibák: 1001 – Invalid address (HTTP-400 Bad Request)

5000 – Internal server error (HTTP-500 Internal Error)

cURL példa:

```
$ curl http://[host_address]/LandInspectorService/land-building-ratio/{address}] \
 -H 'JWT: [token]' \
 -H 'Cache-Control: no-cache'
```

Válasz:

```
{
    "approximate_number_of_buildings": [numeric],
    "land_building_ratio": [numeric],
    "grade": [string]
}
```

ObjectDetectorService

Az objektumdetektálást végző alkalmazás, Python nyelven lesz megvalósítva. Ennek az alkalmazásnak felelősségei közé tartozik, a detektálásra használt modell memóriába való betöltése, annak hatékony kiértékelése, illetve a beérkező adatok feldolgozása és a kiértékelés eredményeinek továbbítása. Az információáramlás Kafka témákon keresztül történik, ezért ehhez a megfelelő szerepkörökre fel kell készíteni az alkalmazást. Szükség van egy Kafka fogyasztó és egy termelő implementálására. A fogyasztó az „Input Topic" nevű témára iratkozik fel, ahonnan egy bájt tömb formájában kapja meg a kép adatokat. A termelő pedig a „Response Topic" témára továbbítja a detektálás eredményeit JSON formátumban (ld.: 24. ábra).

```
{
    "approx_number_of_buildings": [numeric],
    "ratio": [numeric],
}
```

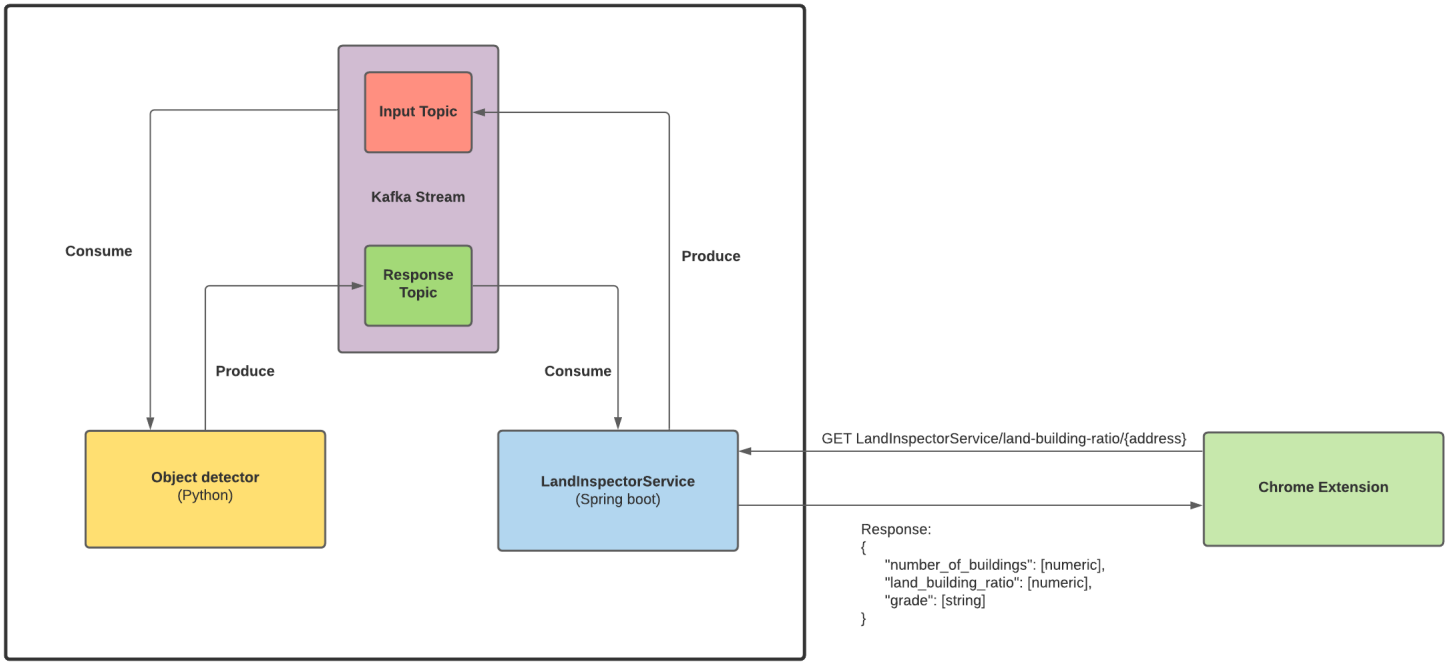
24. ábra A 'Response Topic' üzenet formátuma

Az objektumdetektáló modell egy kép pixelérétkeinek, tömbként való reprezentálását használja bemenetként, majd ennek transzformált változatát használja a predikciók elkészítésére. Emiatt az adatfolyamból érkező bájt tömb formátumú adatok nyers formában, direkt módon nem vezethetők a modell bemenetére. Emiatt a bemenet előzetes feldolgozására van szükség, amelynek kimenete már lehetővé teszi annak felhasználását. Ez a transzformáció két lépésből áll össze. A bájt tömböt először képpé kell alakítani, majd a létrejött képből, egy annak tulajdonságait tároló pixel mátrixot kell létrehozni.

Az objektumdetektálásra a Mask R-CNN algoritmust használom fel. Azért ezt a neurális hálót választottam, mert abból a tulajdonságából kiindulva, hogy a Faster R-CNN-re épül, annak minden pozitív tulajdonságát magával hozza. Ez azt jelenti, hogy a modell képes lesz nagy pontossággal, rövid időn belül a detekció elvégzésére, illetve az azonosított területek köré nem csak egy keretet rendel, hanem az ezekben található objektumokra maszkot is készít. A maszkok segítségével nagyobb pontossággal megállapítható az adott épületek által elfoglalt terület.

A detektáló modell használatának előfeltétele, annak tanítása. A hálózat tanításához szükség van kellő mennyiségű és megfelelő minőségű képre. Ez igaz a tanításhoz, illetve a validáláshoz használt képhalmazokra is. A tanítási folyamat rendkívül erőforrás és időigényes, ezért ehhez a folyamathoz megfelelő erősségű hardver is szükséges. Mivel a dolgozat elkészítése közben nem áll rendelkezésemre kellő erősségű hardver, ezért a tanítási folyamat teljes egészében Google Colab felhős környezetben lesz végrehajtva.

3.3. Architektúrális áttekintés



25. ábra Rendszerterv

4. Megvalósítás

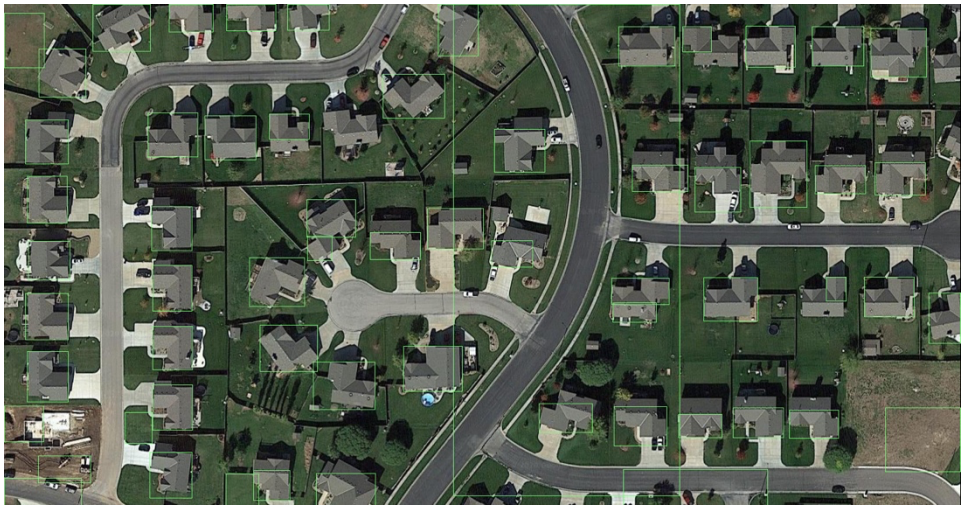
4.1. Épületekre specializált objektum detektáló neurális háló

Mielőtt elkezdtem volna foglalkozni a neurális háló implementálásával, előbb egy másik megközelítéssel próbálkoztam. Az ötlet, a műholdképek szín szerinti szegmentációjára alapult. Kiindulópontja az volt, hogy az épületek sok esetben hasonló színárnyalatú tetővel rendelkeznek, ezért ebből arra következtettem, hogy a detektálás alap képfeldolgozási eszközök felhasználásával is megfelelően elvégezhető.

A prototípust C# nyelven írva, az EmguCV képfeldolgozási műveleteket tartalmazó külső dependencia felhasználásával készítettem el. Az EmguCV, az OpenCV .NET-es interfésze. Első lépésben a jellegzetes tetőszínek (narancssárga, barna, szürke, kékes árnyalatok), illetve a vegetáció szűrését valósítottam meg, majd ennek eredményéből fekete-fehér maszkot állítottam elő, ahol fehér színnel jelöltem az épületnek feltételezett területeket a képen. Azt tapasztaltam, hogy a képek részletessége miatt, nagyon sok olyan apró részlet jut át a maszkolási fázison, aminek nincs jelentős információtartalma a detektálás szempontjából. Emiatt szükség volt valamilyen előzetes transzformációt végeznem a képeken, amellyel kellően el tudtam simítani az éles, apró részleteket. Ezt a funkcionalitást egy tetszőleges homályosító szűrő alkalmazásával lehet elérni, épp ezért én egy Gauss-szűrőt használtam, 5x5-ös kernel mérettel. Ennél nagyobb mértékű homályosítás már túl nagy információ vesztéssel járt. A végső maszkon

fellelhető fehér területek köré, utolsó lépésben határoló dobozokat rajzoltam ki, majd ezt az eredeti képen megjelenítettem.

A detekció pontosítását Canny-éldetektálással próbáltam meg pontosítani, azonban ez a próbálkozás hamar kudarcba fulladt, ugyanis a képeken nem csak az épületek rendelkeztek erős élekkel, hanem a környező utak, esetekben növényzet is, amiket homályosítás útján sem tudtam figyelmen kívül hagyni. Emiatt nagyon sok hamis adattal egészült ki a szegmentáció, ezért ezt a lépést végül elvetettem. A 26-os ábrán láthatók a próbálkozás eredményei, amiken jól látszik, hogy általánosságban ez nem egy hatékony módszer. A képek rengeteg olyan változó tényezővel rendelkeznek, amelyek feldolgozásához ilyen alapszintű képfeldolgozási módszerekkel nem lehet kellően felkészülni. Volt néhány példa, ahol egész jól teljesített a megoldás, azonban ez csak a kifejezetten jó minőségű képeknél volt igaz. A legtöbb esetben a vegetáció nagy része is épületnek lett minősítve a változó színárnyalatok miatt, illetve magas számban voltak kihagyva épületek a detekcióból, olyan képeken, ahol a tetők színe nagyban egyezett a környező terület színeivel. Zajos képek esetében pedig, nagyon hiányos eredmények születtek. Ezek miatt nagyon magas lett fals-pozitív és fals-negatív detekciók aránya.



26. ábra Színalapú épületdetektáló prototípus eredményei

A modell megfelelő kiválasztásánál fontos szerepet töltött be a kellő pontosság és gyorsaság, ezért először fentebb említett Faster R-CNN-nel próbálkoztam. Azért ezt a modellt választottam kiindulásnak, mert a többi módszerrel ellentétben ez jóval gyorsabban tanítható és kiértékelhető, a struktúrájának és régió javasló neurális háló (RPN) használatának köszönhetően.

A modell kiválasztása után a következő lépés, a megfelelő tanító adathalmaz elkészítése volt. A tanuló adatokba beletartoznak a tanítás során használt validációs, illetve teszt adatok is, ezért ezek megválasztásánál fontos törekedni az adatok közötti heterogenitásra. Ez alatt csak annyi értendő, hogy ebben a konkrét esetben, olyan képeket célszerű választani, amelyek minőségben és tartalomban is kellően különböznek egymástól. Ha erre a feltételre ügyelünk, akkor jó eséllyel, nem fogunk olyan hálózatot tanítani, ami csak egy bizonyos fajta képhalmazra ad értékelhető eredményt.

Egy használható adathalmaz elkészítéséhez olyan képek szükségesek, amelyek mellé tartozik képenként egy-egy leíró fájl, ami az adott képen fellelhető objektumok elhelyezkedésének koordinátáit tartalmazza. Ezeknek a leíró fájloknak a létrehozását, címkézésnek hívják, ugyanis ez a folyamat felfogható úgy, mintha címkéket aggatnánk a kép bizonyos részeire. Elsőkörben manuális címkézéssel próbálkoztam, és ehhez a labelImg [9] nevű címkéző eszközt használtam. A manuális címkézéshez felhasznált képek beszerzése sem triviális probléma, ugyanis bizonyos nagyítású műholdképek szükségesek, de szerencsére az interneten több ingyenesen hozzáférhető (kutatási célokra) előre aggregált képcsomagokból válogathatunk. Ilyenek például a SpaceNet [10] vagy DOTA [11] adathalmazai.

A DOTA által szolgáltatott adatokat használtam, mert a SpaceNet csak néhány bizonyos területet fed le. A konkrét adathalmaz, amit használtam, a DOTA-v1.5 volt. Ez összesen 2806 képet tartalmaz ebből tanításra 1/2, validálásra 1/6, tesztelésre pedig 1/3 részben elosztva. A képekhez tartoztak előre meghatározott címkék, azonban azok nem vonatkoztak épületekre, ezért azokat nekem kellett kézzel felvinnem. Mivel a majdnem 3000 kép manuális feldolgozása rengeteg időbe telt volna, ezért csak egy kisebb, nagyjából 1200 képből álló adathalmazon végeztem el a címkézést. Illetve ez a próbálkozás még csak arra a célra szolgált, hogy validáljam

a Faster R-CNN működőképességét és sebességét, hogy a későbbi megoldásomat biztosan tudjam rá alapozni. Ezekkel a képekkel és az előre definiált, tanításhoz szükséges paraméterek használata mellett, a tanított háló nagyjából 23%-os pontossággal adott eredményeket, ami várható volt a meglehetősen alacsony számú tanító adat mellett.

Azonban a kísérlet sikerrel zárult, mert a tanítási folyamat még lokális hardveren is viszonylag gyorsan futott, a predikciós fázis pedig az elvártnál jobban teljesített. Az eredményeket az 1. számú táblázat tartalmazza.

Tanító halmaz (db)	Válidáló halmaz (db)	Tesztelő halmaz (db)	Tanítás ideje (óra)	Átlagos predikció idő (mp)	Átlagos pontosság (%)
587	***213***	***392***	***26***	***5.2***	***22.6***

1. táblázat Kiindulási fázis tanítási eredményei

A Faster R-CNN-nel való próbálkozásból levonható következtetések alapján, a modell alapjaiban megfelel az én felhasználásomnak, viszont mindenképp szükség lenne egy nagyobb méretű képhalmazra, amin a tanítási folyamat történne.

Az én felhasználási esetemben a detektált épületek területe, és a vizsgálat alatt lévő alapterület egymáshoz viszonyított aránya az egyik legfontosabb érték. Ahhoz, hogy a valós arány közelítése minél kisebb hibaszázalékkal történjen, törekedni kell az épületek területének lehető legpontosabb meghatározására. Ebből adódóan a jelenlegi objektum detektáló funkcionalitás önmagában nem elég. A határolódobozok inkább csak vizualizálásra alkalmasak, ugyanis, ha a dobozok területét vennénk a kalkuláció alapjának, akkor durván túlzott értékeket kapnánk. Épp ezért szükség lenne egy utólagos processzálásra, amely a dobozokkal határolt területeken belül maszkolást végezne, ezzel pontosítva a hasznos területek méretét. Ez a folyamat azonban redundáns lépéseket vezetne be, mert a detektálás után még egyszer végig kéne iterálni az összes detektált területen, majd mindegyiken ugyanazt a műveletsorozatot végrehajtani. A probléma jellegéből egyértelműen adódik, hogy jól párhuzamosítható, azonban ezzel egy új komponens, ezáltal még több komplexitás bevezetésére lenne szükség. Illetve a redundáns lépések még mindig jelen lennének, ezért hatékonyabb lenne egy olyan megoldás, amely a dobozok kiszámításakor végzi a maszkolást.

Az R-CNN hálózatok evolúciójában a következő lépcső pont orvosolja ezt a problémát. A Mask R-CNN kifejezetten ennek a hiányosságnak a kiküszöbölésére épít, és mivel alapjaiban megegyezik a Faster R-CNN-nel ezért a rendkívül jó teljesítményét is örökli. Az újonnan kiválasztott modell tanítására már egy jóval nagyobb számosságú, erre a célra készített adathalmazt alkalmaztam. Ezek az adatok ingyenesen elérhetők a CrowdAi oldalán egy ’Mapping Challenge’ nevezetű publikus kihívás alatt [12]. Ez az adathalmaz 280741 db tanító, 60317 db validáló és 60697 db 320x320 felbontású teszt képet tartalmaz. Ekkora mennyiségű kép feldolgozására már nem volt elegendő a saját hardverem, ezért a tanítás további szakaszait Google Colab felhő alapú környezetre migráltam át. A kihívás mellé tartozik egy kiindulási alap, amit viszonyításképpen kiértékeltem. A Mask R-CNN implementációját egy létező

megoldásra alapoztam, melyet a Matterport nevű organizáció publikált [13], illetve az adathalmazt forrásaként szolgáló kihívás is erre alapoz.

Első problémaként a Google Colab fizikai határai jelentkeztek, ugyanis néhány napnál tovább még előfizetéssel sem lehet aktív szerverkapcsolatot fenntartani, emiatt a teljes adathalmaz feldolgozása közel lehetetlenné vált. Ekkor döntöttem úgy, hogy csökkenteni kezdem a képek és a tanító lépések számát, oly módon, hogy a predikciók minősége ne essen át drasztikus romláson. Hosszas kísérletezés után elértem egy határt, amely ~10000 db tanító és ~2000 validáló képpel képes volt teljes a futtatásra, három fázissal, fázisonként 1000 iterációval. Ez az adatmennyiség már elegendőnek bizonyult arra, hogy a tanítás használható eredményeket adjon, betartva a Google Colab limitációit. A futtatáshoz felhasznált tanító konfigurációs paramétereket a 2. táblázat tartalmazza:

Backbone	GPU-k száma	Képek/GPU	Osztályok száma	Iterációk száma/fázis	Iterációk száma/iteráció	Validációs lépések száma	Tanulási ráta
ResNet50	***1***	***2***	***2 (épület + háttér)***	***30/40/50***	***1000***	***50***	***0.001***

2. táblázat Első tanítás paraméterei

A módszer működésének validálása után elkezdtem a saját igényeim szerint alakítani a paramétereket. Első lépésben a ResNet50-et, ResNet101-gyel helyettesítettem, ezzel már pár százalékos javulást elérve. A kiindulási állapotnál a három tanítási fázis az architektúra különböző részein végzett iterációkat. Az első fázisban a hálózat fejeit: a régió javasló, a jellemző piramis és a maszkolást végző hálók tanítása történt 30 iteráción keresztül. A második fázisban a ResNet alsó rétegeitől kezdve felfelé, illetve a fejek tanítása történt 40 iteráción keresztül. A harmadik fázisban pedig a hálózat összes rétegén történt a tanítás 50 iteráción keresztül.

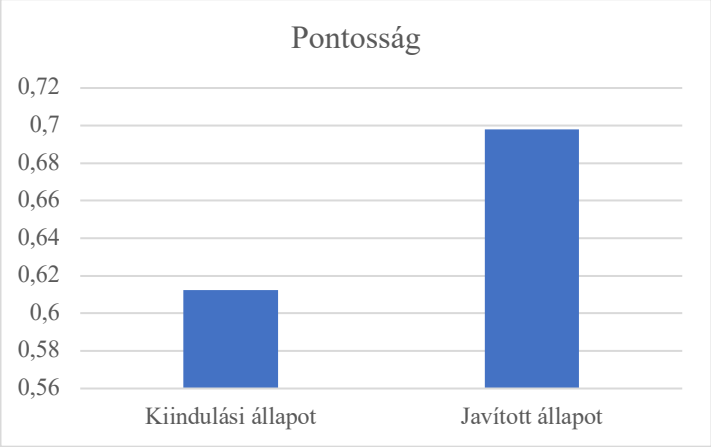
A fázisokat és a tanulási rátát addig módosítottam, amíg a 3. táblázatban látható végállapotot elértem.

Backbone	GPU-k száma	Képek/GPU	Osztályok száma	Iterációk száma/fázis	Iterációk száma/iteráció	Validációs lépések száma	Tanulási ráta
ResNet101	***1***	***2***	***2 (épület + háttér)***	***30/45/60/80***	***2000***	***40***	***0.0005***

3. táblázat Hangolt tanítási paraméterek

A kísérletezés során arra következtetésre jutottam, hogy a tanulási ráta csökkentésével, ellenben az iterációkon belüli iterációk számának arányos növelésével javulást érek el. Illetve

az utolsó fázisban, a teljes hálózat tanításakor jobb eredményeket hozott, ha kisebb tanulási rátát alkalmaztam. Kísérletképpen kipróbáltam mi történik, ha növelem a fázisok számát, ezzel hosszabbá téve a tanulást. Kiderült, hogy ez volt az egyik szűk keresztmetszet, mivel túlságosan alul volt méretezve a tanítás. Ezért három helyett négy fázist vezettem be. Az elsőben a hálózat fejeit, a másodikban a ResNet alsó rétegeitől felfelé, a harmadikban a ResNet felső rétegeitől kiindulva a negyedikben pedig az összes réteget tanítottam. Az eredményként elért javulás, a 27. ábrán látható.



27. ábra A tanítás módosításával elért javulás mértéke

Az újonnan elért pontosság elegendőnek bizonyult a hatékony detektáláshoz, ezért a következő lépés a tanított modell exportálása, majd e-köré egy alkalmazás építése volt.

4.2. ObjectDetectorService

Az objektum detektáló alkalmazás alapjait, kizárólag a Python által biztosított könyvtárak segítségével valósítottam meg, mert nem volt szükség egy masszívabb keretrendszer használatára. A detektálást végző kód a *predictor.py* fájlban található. Itt történik a predikcióhoz használt konfigurációs objektum inicializálása, a modell betöltése és a detektáláshoz használt metódus definíciója. A predikció konfigurációjánál ügyelni kellett arra, hogy a tanításnál használt paraméterekkel is össze legyen hangolva. A tanítás alapjául a ResNet101-et használtam fel, ezért a predikciós konfigurációban is ezt kellett alkalmazni. A képeknek 320x320-as méretet állítottam be, mivel a modellen is ilyen dimenziókkal végeztem el a tanítást. Minden, a predikció során azonosított objektum, rendelkezik egy valószínűség értékkel, ami a detekció magabiztosságát jelöli. Mivel a neurális hálók nem a bináris logikára alapozva működnek, ezért kimenetül csak valószínűségeket adnak. A detektált objektumoknál,

ezért egy értékhatár megadására is szükség van, hogy csak egy bizonyos magabiztossági szint felett fogadjunk el predikciókat. Legelőször, ezt az értéket 0.75-re állítottam be.

Így a predikciós konfiguráció kiindulási állapota a következőképpen nézett ki:

BACKBONE = „resnet101”

IMAGE_MIN_DIM = 320

IMAGE_MAX_DIM = 320

DETECTION_MIN_CONFIDENCE = 0.75

A detekciót végző metódus, egy bájt tömböt vár paraméterül, amit egy konverzió segítségével képpé alakítok, majd ezt a képet átadom a modellnek kiértékelésre. A modell kimenete egy olyan kollekció, amely tartalmazza a detektált objektumokat, az ahhoz tartozó határoló dobozokat és maszkokat, illetve ezekhez rendelt valószínűségeket. A kimenethez szükséges adatok közül az épületek számát közvetlen a modell kimenetéből elő lehet állítani, azonban az arányszám meghatározásához további műveletek elvégzésére van szükség. Ennek meghatározásához, a kollekcióból le kell kérni a maszkokat, amikből azok összegzett mérete már meghatározható. Az arány az alábbi képlettel számítható ki:

$$T_{mask} = \sum_{i=0}^{n} masks[i].area$$

$$Ratio_{building}(\%) = (T_{mask} / (image\_size * image\_size)) * 100$$

Az adatfolyamra kerülő kimenet egy objektum, amely a detektált épületek számát és a vizsgált terület, illetve a maszkok által lefedett területek arányát adja meg. Az alkalmazásban opcionálisan, a kiértékelt képek vizualizálására is lehet engedélyt adni.

Mivel a kérések fogadására és továbbítására Kafka-t használok, ezért ennek megfelelően implementálnom kellett egy fogyasztót és egy termelőt. Ennek megvalósításához a kafka-python [14] külső dependenciát használtam. A könyvtár segítségével egyszerűen definiálhatók ezek a szerepkörök, csupán csak konfigurációkat kell meghatároznunk, amelyek tartalmazza a Kafka adatfolyam elérhetőségét, a használt témák nevét, az üzenetek formátumát és még sok más opcionális beállítást. Az ehhez kapcsolódó kódrész a *kafkaconf.py* nevű fájlban található.

Az épületdetektáló modell futtatása egy objektummal tér vissza, amely tartalmazza a területi adatokat az alábbi formában:

approx_number_of_buildings: [numeric]

ratio: [numeric]

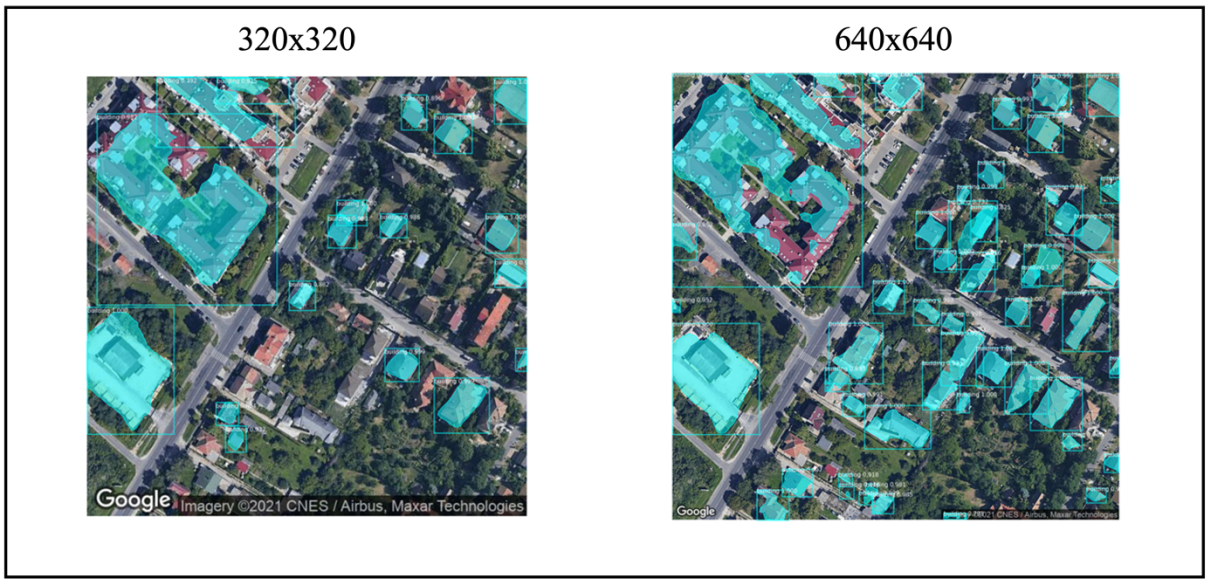
Mivel a predikció eredménye egy Python-ban értelmezett objektum, ezért azt még JSON formátumúvá kell alakítani, annak érdekében, hogy kompatibilis legyen az adatfolyam által elvárt üzenetstruktúrával. A Kafka termelők esetében, lehetőségünk adódik egy bemenetet szerializáló kód definiálására. Ez a funkció felel a termelőnek átadott adatok transzformálásáért, ezért felfogható úgy is mint egy átalakító interfész a programunk által értelmezhető és az adatfolyamon közlekedő adatok között. Az én esetemben ez a konfigurációs paraméter egy rövid funkcionális módon megírt lambda függvény, ahol *x* jelképezi a termelőnek átadott objektumot:

value_serializer=lambda x: json.dumps(x).encode(“utf-8”)

A fenti kód, a Python-ban megtalálható *json* könyvtár segítségével, szerializálja a bemeneti objektumot, UTF-8 karakterkódolással.

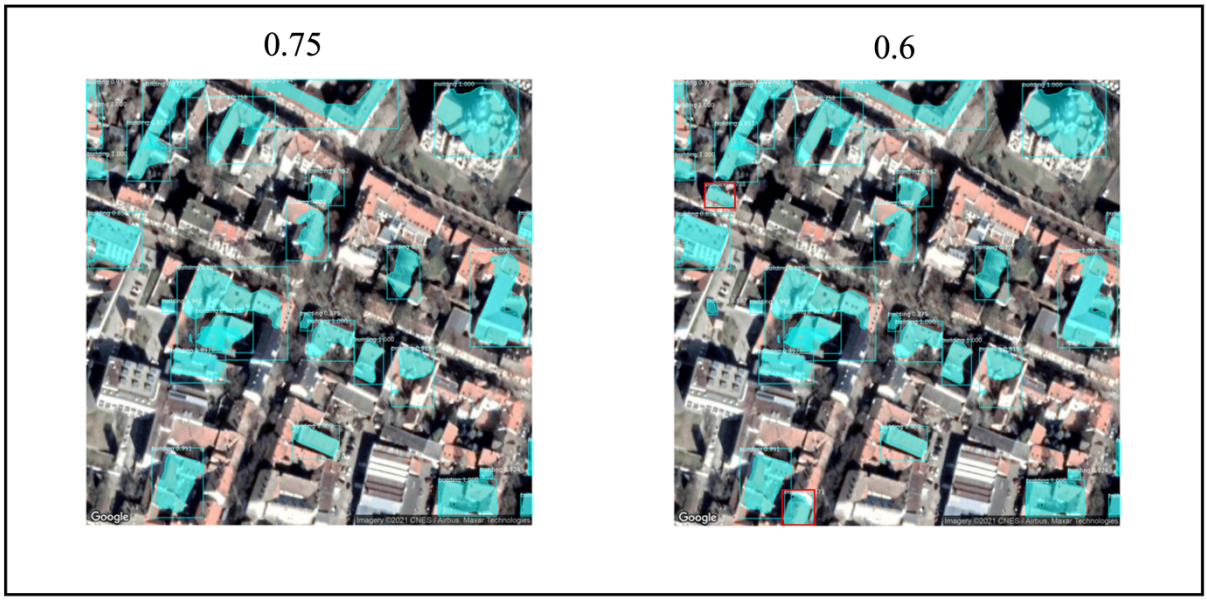
Némi kísérletezés után feltűnt, hogy a modell elég sok épületet hagy észrevétlenül, viszont ez a viselkedés a tanítás és validálás során nem jelentkezett. Ezért elkezdtem keresni a szuboptimális működés okát, így első lépésként megpróbáltam a magabiztossági határ változtatását. A határérték csökkentése, azonban nem segített a problémán. Következő lépésben a bemeneti képek méretének növelésével próbálkoztam. A kiindulásként definiált 320x320-as méret növelésével arányosan kezdett el javulni a detektálás minősége. Kis lépcsőnként haladva

eljutottam a 640x640-es mérethatárig, amin túl már negatív irányba mozdult a predikciók megbízhatósága, illetve nagyobb méretű képeknél a kiértékelés ideje is akkora mértékben megnőtt, hogy az már nem bizonyult elfogadhatónak a szolgáltatás szempontjából. A megoldást az magyarázza, hogy a duplázott méretű képek több információval rendelkeznek, mint a kisebb változatok. Annak ellenére, hogy a hálózat 320x320-as képeken lett tanítva, a kétszer ekkora képekkel hatékonyabban működik, ahogy az a 28. ábrán is látható.



28. ábra A kép felbontásának növelésével elért javulás

Adódtak olyan esetek is, amikor erősen elmosott, zajos, rossz minőségű képeken nem volt képes kisebb épületeket azonosítani a modell. Azonban itt a magabiztosság szint csökkentése, javulást eredményezett. A legtöbb esetben, illetve a jobb minőségű képeknél ez azért nem okozott változást, mert azok esetében nagyrészt csak magas valószínűségű detekciók születtek. A határ 0.75-ről 0.6-ra csökkentésével apró javulást értem el (ld.: 29. ábra), ehhez hasonló esetekben.



29. ábra Az elfogadási határ csökkentésével elért javulás

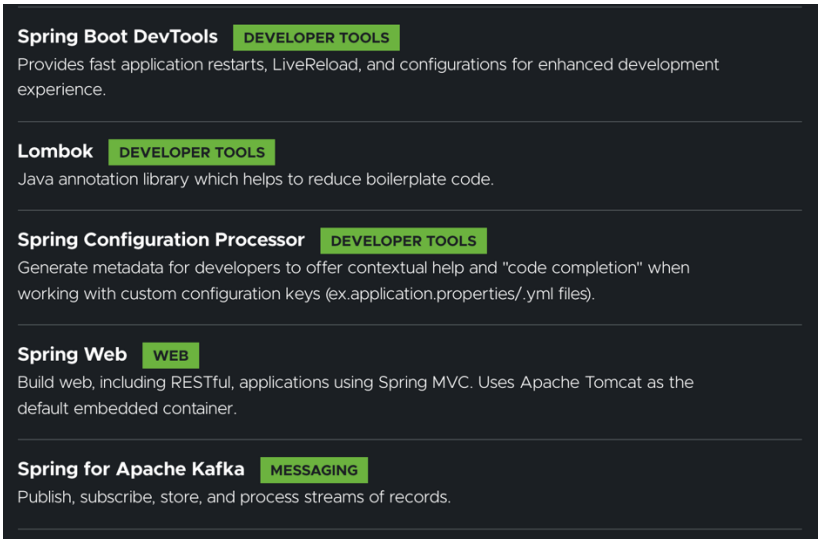
4.3. LandInspectorService

A felhasználói felülettel kommunikáló szolgáltatás megírását Java nyelven, a Spring Boot keretrendszer segítségével valósítottam meg. A projekt létrehozásánál választhattam, hogy milyen ’build tool’-t fogok használni. A ’build tool’-ok olyan szoftverek, amelyek segítségével automatizálttá tehetők a szoftverek életciklusának lépései. Ide tartozik a forráskód lefordítása gépi kóddá, a dependenciák és csomagok kezelése, illetve a folyamatos integrációt elősegítő eszközök támogatása. Ezeknek az eszközöknek a használata természetesen nem kötelező, de erősen ajánlott.

A modern szoftverfejlesztés korára mára ipari standarddá váltak, ezért vállalati környezetben majdnem mindenhol előfordulnak. Használatuk már csak azért is elvárt, mert így rengeteg időt spórolhatunk a fejlesztés menetén, az egyébként manuális feladatok automatizálásával. Cégek vagy akár magánszemélyek esetében ez a megspórolt idő, rengeteg megtakarított erőforrásköltséget is jelent. Manapság a két legelterjedtebb automatizációs eszköz, a Maven és a Gradle. A Gradle egy sokkal újabb eszköz, ezért a fejlesztésénél már figyelembe vették az újabb technológiákat, viszont kora miatt még nem annyira elterjedt, mint a Maven. A két szoftver között jelentős különbség, modularizálhatóságukban nyilvánul meg. Mindkét eszköz futtatási fázisokból épül fel, amik egymás után hajtódnak végre. Ezek a lépések legtöbb esetben dependálnak egymásra. Lehetőségünk van saját fázisok definiálására is, amikben személyre szabott kódot futtathatunk. A Gradle legnagyobb előnye a Maven-nel szemben, hogy jóval személyre szabhatóbb, azonban ebből a tulajdonságából ered a hátulütője, hogy komplexebb, mint a régebbi társa. A Maven egy hamar megérthető, jól átlátható struktúra mentét épül fel. XML alapú konfigurációt tesz lehetővé, a Gradle pedig a mára már jobban elterjedté vált JSON formátumot használja.

Esetemben a Gradle mellett szóló érvek ellen mégis a Maven-t választottam, mert régre nyúló múltja miatt szélesebb körben támogatott. Illetve a Gradle használata, azokban az esetekben hozhat jelentős előnyt, ha a fejlesztési folyamat nem átlagos, azaz rengeteg személyre szabott, egymástól függő részből áll össze. Ilyen szituációkban jól ki lehet használni a Gradle nyújtotta szabadságot, azonban kevésbé komplex felépítésű projekteknél nem biztos, hogy megéri a konfigurációval extra időt eltölteni, ha végeredményben ugyanazt érjük el vele, mintha Maven-t használnánk.

A szoftver-automatizációs eszköz megválasztása után, a következő lépés a projekt létrehozása volt. A projekt csontvázának megalkotását megtehettem volna manuálisan is, azonban ez nagyobb hibázási lehetősséggel és több elpocsékolt idővel járt volna. Ezért inkább a Spring készítői által biztosított Spring Initializert alkalmaztam, amely segítségével egy vizuális felhasználói felületen, egyszerűen összeállítottam és kigeneráltattam a projektem alap struktúráját, minden függőségével együtt (ld.: 30. ábra). Az alkalmazás készítésekor épp aktuális Spring Boot verziót használtam, a 2.5.5-öt. A projekthez az alábbi dependenciákat adtam hozzá:



30. ábra A Spring alkalmazás függőségei

Spring Boot DevTools - Ez a csomag kizárólag a fejlesztés ideje alatt használatos. Megkönnyíti és lerövidíti a fejlesztési iterációk számát. Használatával megoldható, hogy a futó alkalmazás automatikusan újra induljon, bármilyen elmentett kódváltoztatás esetén. Megvalósítást biztosít arra is, hogy alkalmazásunkat távolról futtassuk, illetve debugoljuk.

Lombok – Annotációk használatával lehetővé teszi a sablon kódok írásának elhagyását, mint például konstruktorok, setter-ek, getter-ek. Ilyen annotációk például a @NoArgsConstructor, @Setter, @Getter, @Data, @Value.

Spring Configuration Processor – Ez dependencia teszi lehetővé, hogy alkalmazásunk többféle futtatási profillal rendelkezhessen. Konfigurációs fájlok létrehozásával olyan változókat definiálhatunk vagy vezethetünk ki, amelyek környezetenként más-más értékekkel rendelkeznek, vagy esetleg futás közben változtatni kell őket a forráskód módosítása nélkül.

Spring Web – Az alkalmazás szempontjából legfontosabb építőelem, ugyanis ez nyújt támogatást webes alkalmazások elkészítéséhez, és tartalmaz minden ehhez szükséges programkönyvtárat. Alapértelmezetten tartalmazza a Tomcat alkalmazásszervert is.

Spring for Apache Kafka – Lehetővé teszi a Kafka által definiált szerepkörök konfigurációját és alkalmazását. Egy interfészt nyújt az adatfolyam felé.

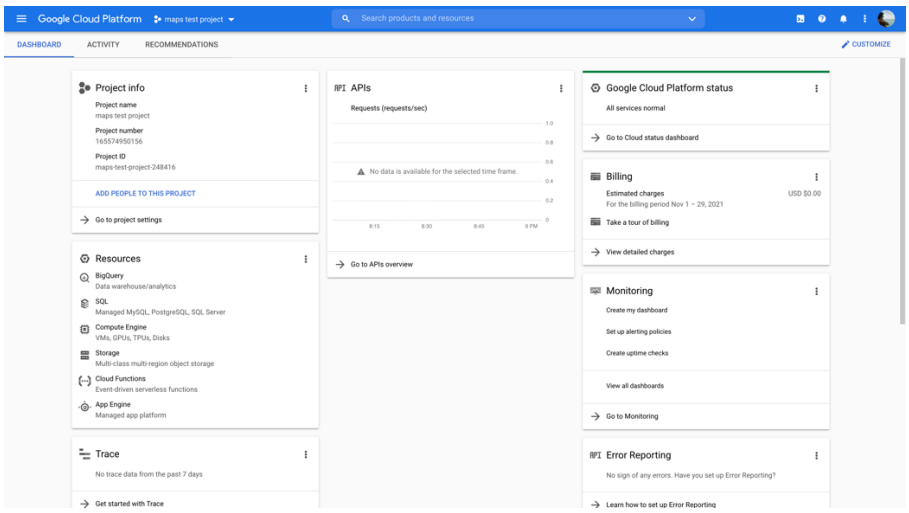
Kezdetben létrehoztam a webes alkalmazásoknál általánosságban használt rétegeket, ami után a következő csomagstruktúra alakult ki:

- web
  - response
- model
  - domain
- dao

A web rétegben definiáltam egy kontrollert, amibe az alkalmazás publikált végpontja kap helyet. A kontroller teszteléséhez először létrehoztam egy példa végpontot, amelyben még nem volt elhelyezve logika, csupán egy előre megírt válasz. Ez az interfész az következő URL-en keresztül érhető el: */land-building-ratio/example/{address}*.

A felelősségi körök szétválasztásának elvét követve, a kontroller szint kizárólag az üzenetek továbbításáért felel, a kliens és a szoftver belső komponenseinek irányába. Egy szinttel lejjebb, a *model* rétegben kapott helyet az üzleti logika részeinek konkrét megvalósítása, illetve az ahhoz tartozó objektumok definiálása.

Következő a lépésben a bemeneti cím konverzióját kellett megvalósítanom. Erre létrehoztam egy külön komponenst, amely egy HTTP kéréssel meghívja a Google Maps Static API végpontját. Azonban ettől az interfésztől nem lehet csak úgy adatot lekérni. A használatához rendelkeznünk kell egy Google Cloud fiókkal, amit előbb a felhasználásunknak megfelelő módon fel kell konfigurálnunk. Ezt a Google Cloud Console felületén tehetjük meg (ld.: 31. ábra).



31. ábra A Google Cloud Console vezérlőfelülete

Első lépésben meg kell adnunk a számlázási adatainkat, mivel minden szolgáltatás bérlés alapú. Viszont a legtöbb szolgáltatásnál léteznek olyan határok, amiken belül ingyenes használhatjuk őket. A Maps Static API esetében is létezik egy ilyen határ, azonban ez viszonylag magas, ezért nem kellett aggódnom annak túllépése miatt. Ha még felmerül ez a lehetőség, akkor természetesen követni tudjuk a használati statisztikáinkat egy erre külön létrehozott felületen. Ezután létre kell hoznunk egy projektet, amihez a felhasználni kívánt szolgáltatásokat tudjuk majd a későbbiekben hozzárendelni. A projekt beállításaiban meg kell adnunk az első lépésben felvett számlázási módot, majd ezután tudjuk elkezdeni használni az engedélyezett szolgáltatásokat. Minden szolgáltatáshoz kapunk egy API kulcsot, amivel autentikálhatjuk a magunkat, a kérések végrehajtásakor.

A számlázási adatok megadásával akadt egy kis problémám, mert semelyik bankkártyámat nem fogadta el a rendszer, mindegyikre ismeretlen hibát adott. Egy kis utánajárás után kiderült, hogy a Google Cloud Európában még nem használható rendesen, mert a számlázás folyamata nem felel meg minden Európai Uniós előírásnak. Ezért megoldásul amerikaira állítottam a számlázási országot és ezután már elfogadásra került a fizetési módom. Miután sikeresen készítettem egy használható API kulcsot a Maps Static végpontjának meghívásához, össze kellett állítanom egy kérést, amely a számomra megfelelő méretű, nagyítású és formátumú műholdképet ad vissza. Ezen paraméterek megadását, URL-en keresztül lehet megtenni. A kérésnek a következő paramétereit használtam fel:

center: A kép középpontjában lévő földrajzi hely definiálására használatos.

zoom: A kép nagyításának mértéke szabályozható vele.

scale: Szorzó, amellyel a kép pixeleinek számát állíthatjuk.

Pl.: scale=2 esetén a pixelek száma megduplázódik.

maptype: A térkép stílusát adja meg. Pl. műhold, forgalom, domborzat

size: A kép mérete definiálható vele.

(maximum 640x640, illetve skálázással 1024x1024)

format: A kép formátuma. (png/png8, gif, jpg, jpg-baseline)

key: Autentikációhoz használt API kulcs

A fenti paraméterek hangolása után elértem egy olyan állapotot, amely megfelelő minőségű képeket eredményezett, a következő értékekkel: maptype=satellite, zoom=18, scale=2, size=640x640, format=jpg. 18-as nagyítással, a megadott címek 500-700 méteres körzetét sikerül a képbe foglalni, ez a terület pedig már elegendőnek bizonyult, ahhoz, hogy hiteles információkkal szolgáljon. A kép formátumát azért JPG-nek választottam, mert az objektumdetektáló modell is ilyen formátumú képeken képes végrehajtani a predikciókat. A skálázást pedig, azért növeltem kétszeresre, hogy a képek több információt tároljanak, ezzel is javítva a predikciók eredményét.

Az API paraméterek beállítása után, a válaszul érkező kép bájt tömb reprezentációjának továbbítását kellett implementálnom. Mivel az adattovábbítás már nem az üzleti logika feladatkörébe tartozik, ezért az erre specifikusan létrehozott *dao* rétegen belül valósítottam meg

a Kafka szerepkörökhöz tartozó műveleteket. Az üzenet termelők és fogyasztók definiálását külön konfigurációs komponensekben hoztam létre, annak érdekében, hogy bárhonnan könnyen lehessen hivatkozni rájuk. A Python alkalmazáshoz hasonlóan ezek a konfigurációk is a Kafka adatfolyam elérését, a témák neveit, az üzenetek formátumát és egyéb opcionális beállításokat tartalmaztak. A konkrét üzenettovábbító és feldolgozó logikát egy *KafkaDao.java* nevű osztályba szerveztem ki, ahol a korábban definiált konfigurációkkal hajtom végre a kívánt műveleteket. Ez azt eredményezi, hogy a *model* rétegben csak erre az objektumra kell hivatkoznom, az adatfolyam eléréséhez.

Az objektumdetektáló alkalmazástól a válasz JSON formátumban érkezik vissza, ezért ezt még át kell alakítanom, illetve ki kell bővítenem. A terület jellemzését egy kategória paraméter meghatározásával bővítem ki, ami tulajdonképpen egy ember által olvasható értékelése az eredménynek. A kategóriák meghatározását az arányszámokra alapozva implementáltam, különböző határértékek definiálásával. Kiindulásnak előre meghatározott értékeket állítottam be a következőképpen:

COMFORTABLE(<=15), GOOD(<=25), AVERAGE(<=40), CROWDED(<=60),BAD(>60)

Későbbi bővítési lehetőségként felmerül a felhasználókezelés, ami már lehetővé tenné ezeknek az értékeknek a személyre szabhatóságát.

A területi adatok meghatározása után, a *model*-ben összeállított objektum vissza kerül a kontroller rétegbe, ahonnan az alábbi formátumban továbbítódik egy válasz a kliens felé:

```
{
   "data": {
      "approximate_number_of_buildings": [numeric],
      "land_building_ratio": [numeric],
      "grade": [string]
   },
   "error": { (Optional)
      "code": [string],
      "message": [string],
   }

}
```

4.4. Kafka adatfolyam [18]

Az adatfolyam alapú kommunikációs csatorna kialakításához szükség van egy Kafka példány/bróker (node/broker) létrehozására. Ha csak egy példány futtatására van szükség vagy a jövőben nem lesz szükség az elosztottság kihasználására, akkor ezt bármiféle egyéb eszköz nélkül megtehetjük. Azonban lássuk be, hogy ez az eset korántsem felel meg a való élet

elvárásainak. Ha már a Kafka-ra kerül választásunk, akkor biztos, hogy nagyobb mértékű adatáramlással számolunk, ahol már egy példány helyett, példányok csoportjait kell kezelnünk, beleértve az elemek közötti szinkronizációt, replikációt, versenyhelyzeteket és még sok más nehézséget. A felmerülő problémák feloldása rendkívül nehéz feladat, ezért, ha ezt saját magunknak kellene megvalósítanunk, valószínűleg egy hosszas, masszívabb fejlesztés születne belőle. Szerencsére az elosztottság eredményezte kihívások kiküszöbölésére, létezik egy nagyon hasznos szoftver, az Apache ZooKeeper.

Az ZooKeper egy olyan centralizált szolgáltatás szerepét tölti be, amely konfigurációk menedzselését végzi, rendkívül rugalmasan skálázható és megbízható módon, elosztott rendszerek esetében. Nyomon tartja a Kafka példányok témáit, partícióit, státuszát és minden fontos tulajdonságát. Röviden egy metaadat tárolóként használt szolgáltatás. A ZooKeeper Atomic Broadcast (ZAB) protokollra alapozva, lehetővé teszi a kliensek számára, hogy egyidejűleg írjanak és olvassanak a rendszerben. A tárolt és közlekedő adatok több példány között vannak szétosztva, így, ha egy példány valami okból kifolyólag meghibásodik, akkor egy másik át tudja venni a helyét és ki tudja szolgálni a beérkező kéréseket. Erre az azonnali váltásra képes a ZooKeeper, ezért ennek köszönhetően lesz magas rendelkezésre állású, illetve konzisztens.

A ZAB protokoll biztosítja a Kafka példányok közötti replikáció helyes végbemenetelét. Koordinálja a replikáció sorrendjének betartatását, és emellett a meghibásodás esetén a következő vezető, illetve a kiesett node-ok kezeléséért is felel. A replikáció sorrendje azt jelenti, például, ha beérkezik egy A tranzakció, majd utána egy B, akkor ezt a beérkezési sorrendet minden replikáló példánynak kötelezően tartania kell, annak érdekében, hogy az üzenetek integritása megmaradjon. Ez a sorrendiség, nem csak az üzenetek beérkezésére, hanem azok feldolgozására is érvényes. Így minden példányon megegyező sorrendben kerülnek feldolgozásra az adatok, ezáltal nem fordulhat elő olyan eset, amikor valamelyik node-on már formában kerül eltárolásra információ.

A brókerek meghibásodása esetén a ZooKeeper által végzett koordináció teszi lehetővé a rendszer további működőképességét. Minden partícióban létezik egy kontroller példány, amely a többi példánnyal kontroller-követő kapcsolatban áll. Ez azt jelenti, hogy a kontrollert követik a csoport többi tagjai, így, ha egy bróker kiesik, akkor a kontroller feladata utasítani a többi brókert, hogy helyettesítsék a kiesett csomópontot. Ilyen esetekben új kontroller megválasztása is lehetséges, de úgy, hogy garantált, hogy csak egy bróker veszi át ezt a szerepet.

Emellett folyamatosan követi a létező témákat, azok partícióit, a replikák helyét, téma konfigurációkat, illetve számontart minden brókert és kontrollert. Azért rendkívül fontos az állandó megfigyelés, mert egyrészt így garantálható a kontroller brókerek megfelelő rotálása, illetve a csoportok közötti konfiguráció is erre alapul.

Ezért, ha egy bővíthető kommunikációs csatornát akarunk kialakítani Kafka-val, akkor mindenképp érdemes megfontolni a ZooKeeper használatát, amíg el nemkészül a Kafka saját megoldása a problémára.

A fentiekben részletezett okok miatt én is a ZooKeeper használata mellett döntöttem. A Kafka-ZooKeeper páros elindítása könnyen megvalósítható az alábbi terminál parancsok lefuttatásával, ha a két szoftver függőségeit már letöltöttük.

bin/zookeeper-server-start.sh config/zookeeper.properties

bin/kafka-server-start.sh config/server.properties

Mindkét sor végén látható egy *.properties* kiterjesztésű fájl, amiben az adott szolgáltatáshoz tartozó konfigurációkat határozhatjuk meg. Ez a fajta megoldás nem túl praktikus, ugyanis a szolgáltatások csak addig futnak, amíg a terminál ablakuk nyitva van. Érezhető, hogy ez éles környezetben nem egy megbízható módszer. Sokkal egyszerűbb és célszerűbb lenne elkülönítve kezelni őket. Ennek megvalósítására Docker-t alkalmaztam. A Docker lehetővé teszi alkalmazások, szolgáltatások konténerizált futtatását, ami azt jelenti, hogy ezek a folyamatok számukra dedikált, a külvilágtól elhatárolt virtuális környezetben futnak. A külvilág felé való elérést nyitott portok definiálásával érhetjük el. Ahhoz, hogy egy alkalmazás futtatható legyen Docker-ben, szükség van egy hozzá tartozó 'image' fájlra, amely egy futtatható állományként fogható fel. A legtöbb gyakran használt 'image' a https://hub.docker.com/ weboldalon elérhető és letölthető.

Egy működő konfiguráció megalkotása után, rögtön átváltottam a Docker használatára és a 32. ábrán látható *docker-compose.yml* fájl definiálásával hoztam létre a Kafka-ZooKeeper konténerizáltan futtatható verzióját. Ez a *docker-compose* nem egy általam kitalált specifikus elnevezés, hanem a Docker által elvárt és használt elnevezés, az ilyen konfigurációs fájlok esetében.

A Docker Compose segítségével könnyen hozhatunk létre multi-konténerizált alkalmazásokat, amiket aztán egy parancs futtatásával elindíthatunk, illetve leállíthatunk. Ahhoz, hogy el tudjuk indítani a fájlban található konténereket, a fájlt tartalmazó mappába kell navigálnunk, majd kiadnunk a *docker-compose up* parancsot. A futó konténereket pedig a *docker-compose down* parancs segítségével tudjuk leállítani. Az ábrán jól látszik a két konténer, illetve azok paraméterei. A két konténer között egy belső hálózaton keresztül teszem lehetővé a kommunikációk, ami az *app-network* névvel azonosítható. Amit még fontos kiemelni, az az, hogy a portok definiálásánál használt szintakszis pontosan mit is jelent. A kettőspont bal oldalán lévő port szám jelöli a virtuális környezetben megfeleltetett portot, a jobb oldali, pedig a külvilág számára elérhetőt. Ezért értelemszerűen a jobb oldali portra hivatkozva leszünk képesek a konténerekben futó alkalmazások elérésére.

```yaml
version: '2'
services:
  zookeeper:
    container_name: zookeeper
    image: wurstmeister/zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - 22181:2181
    networks:
      - app-network

  kafka:
    container_name: kafka
    image: confluentinc/cp-kafka:latest
    #image: wurstmeister/kafka:latest
    ports:
      - "9092:9092"
      - "29092:29092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_HOST_NAME: kafka
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_CREATE_TOPICS: "landDataInput:1:1,landDataOutput:1:1"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    depends_on:
      - zookeeper
    networks:
      - app-network

networks:
  app-network:
    driver: bridge
```

32. ábra Kafka és ZooKeeper docker-compose konfiguráció

Miután sikerült működésre bírnom a Kafka-t, már csak annyi maradt hátra, hogy a két backend alkalmazás konfigurációját hozzáigazítsam a *docker-compose* végleges változatához. Ezután a két alkalmazás már képes volt az egymással való kommunikációra. A fejlesztés utolsó fázisában már csak a felhasználói felület implementálását kell végbe vinni.

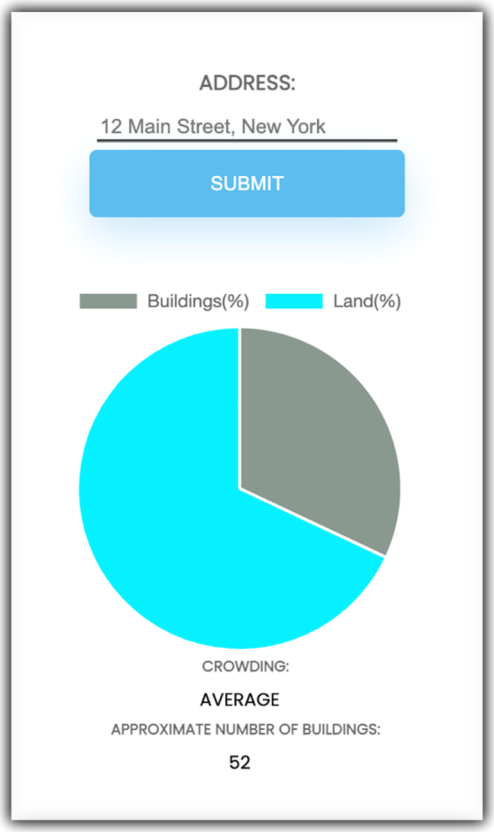
4.5. Chrome Extension felhasználói felület

A felhasználói felületként szolgáló Google Chrome Extension elkészítését kizárólag a webes fejlesztés alapeszközeivel (HTML, CSS, JavaScript) szerettem volna megvalósítani. Ennek az volt az oka, hogy egy olyan alkalmazás megalkotását tűztem ki célul, amely minél kevesebb erőforrást és külső dependenciát vesz igénybe futása közben. Ez a tervezéi döntés hosszú távon egyszerűbb karbantarthatóságot és átláthatóságot eredményez.

A felület tartalmaz egy szöveges bemeneti mezőt, ahova a kiválasztott cím adható meg, alatta pedig egy gombot, amellyel elindítható a területvizsgálati kérés. A kiértékelt eredmények,

a bemenet megadására fenntartott rész alatt jelennek meg. Elsőként a diagram kap helyet, amelyen a területi adatok vizualizációja látható, majd ez alá kerül a kategorizált értékelés, illetve az épületek közelítőleges száma.

Első lépésként a felület tervezett kinézetét valósítottam meg HTML és CSS használatával. A végleges dizájn a 33. ábrán látható módon alakult.



33. ábra A felhasználói felület kinézete

Egyedül a diagram megjelenítéséért felelős komponens esetében alkalmaztam külső dependenciát, mert annak kézzel való implementálása túl problémás lett volna, illetve erre már egyébként is léteznek optimálisan megírt megoldások. Többek között ilyen a Google Charts [18], a Chart.js [19] vagy az amCharts [20]. Mindegyik könyvtár tartalmaz széles körben személyre szabható kördiagramokat, azonban a Chart.js nyerte el a tetszésemet. A Chart.js által biztosított sablonok illeszkednek legjobban az elképzelt dizájn tervembe, illetve ez a könyvtár nyújtotta a legletisztultabb megvalósítást. Emellett tartva magam a célkitűzésemhez, miszerint minél kevesebb külső függőséggel szeretném elkészíteni a felületet, a Chart.js diagramok ennek is megfelelve, egyszerűen implementálhatók HTML és JavaScript kód segítségével. Az eredeti kinézeti tervben a százalékos arány érték is külön megjelenítésre kerül, azonban a végleges megoldásban ez akkor válik láthatóvá, ha a diagram felé navigálunk a kurzorral.

A kinézet elkészítése után bekötöttem a LandInspectorService HTTP végpontjának meghívását a JavaScript kódba, amely a területi adatok szolgáltatásáért felel. Ez a rész a *backend.js* fájlban található. Az eredményeket, a feldolgozás után pedig kivezettem a felhasználói felületre. Abban az esetben, ha a Backend komponensek valamiért elérhetetlenné válnának, akkor ez a felhasználó felé egy hibaüzenettel közlöm: *Could not reach server*, és a

küldés gombot inaktív állapotba teszem. A hibakezelésen kívül, minimális input validáció is implementálásra került. Ha a felhasználó nem ad meg bemeneti értéket vagy, ha a megadott érték kizárólag üres (whitespace) karakterekből áll, mint a *Space* vagy a *Tab*, akkor nem kerül lefuttatásra az detekció. Annak a meggátolására, hogy egy felhasználó képes legyen elárasztani a szervert kérésekkel a beküldés gomb folyamatos lenyomásával, a gombot addig inaktív állapotba helyezem, amíg az elindított kéréstől válasz nem érkezik. Ha valaki nagyon akarná, akkor ezt a védelmet is ki tudná kerülni, azonban ennek valós meggátolására már a mögöttes infrastruktúrában, például egy API Gateway használatával kell felkészülni.

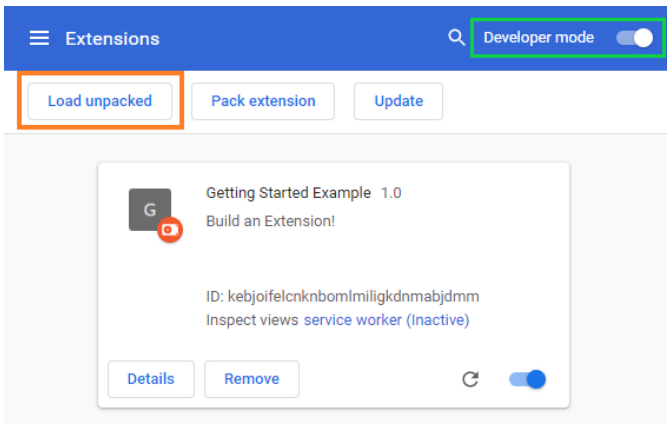
Ahhoz, hogy egy Chrome kiegészítő futtatható legyen a böngészőben, egy *manifest.json* nevű konfigurációs fájl definiálására van szükség, amely az alkalmazás beállításait, köztük a belépési pontját is tartalmazza, ahogy az a 34-es számú ábrán látható.

```
{
  "name": "Building detector",
  "manifest_version": 3,
  "version": "1.0",
  "action": {
    "default_popup": "popup.html"
  }
}
```

34. ábra Chrome Extension manifest.json konfigurációs fájl

Fontos paramétere a *manifest_version*, amely a manifest specifikáció típusát jelöli, azaz a Google Chrome Extension platform aktuális állapotát. A jelenlegi Chrome Extension manifest verzió a V2, viszont mivel a 2020-ban megjelent a V3, ezért a Google már arra ösztönzi a fejlesztőket, hogy az újabb verzió alkalmazásával hozzák létre alkalmazásaikat. A hármas verzió már sokkal szélesebb körű személyes adatvédelmet tesz lehetővé, mint a korábbi társa, illetve rengeteg új technológiát is támogat, mindamellett, hogy még teljesítménybeli javulásokat is tartalmaz. Emiatt a kiegészítőt már a hármas verzió használatával készítettem el.

A megfelelő konfiguráció elvégzése után, lokálisan elérhetővé tettem a kiegészítőt a Google Chrome böngésző Extension-jei között. A telepített kiegészítőket úgy érhetjük el, ha a Chrome keresőjébe *chrome://extension* elérést beírjuk. Majd ezen az oldalon a fejlesztői mód bekapcsolásával, a 'Load Unpacked' gombra kattintás után tudjuk kiválasztani fájlok elérési útvonalát. Ezután már láthatóvá válik a kiegészítő a listában, illetve a böngésző fejlécében (ld.: 35. ábra).



35. ábra Chrome Extension lokális hozzáadása a Google Chrome-hoz

5. Eredmények kiértékelése

A dolgozat céljaként kitűzött szolgáltatás elkészítése sikerrel zárult. Az elvárásként definiált feladatok mindegyike megvalósításra került. Az alkalmazás összes része teljesíti a funkcionális elvárásokat, illetve képes a közvetlen kapcsolatban álló komponensekkel való kommunikációra. A szolgáltatás alapjaként szolgáló épületdetektáló neurális hálózat, illetve a többi komponens kiértékelését a következő alfejezetekben részletezem.

5.1. Épületdetektáló neurális hálózat

Az objektum detektáló vagy klasszifikációt végző neurális hálózatok esetében gyakran használt pontosságot mérő metrika, az mAP (mean average precision) [21], azaz átlagos átlag precízió. Nevéből könnyen arra következtethetünk, hogy a precíziós értékek átlagát rejti, azonban a valóság közel sem ennyire egyszerű. A precízió egy olyan mérőszám, amely megadja, hogy mennyire pontosak a predikcióink, azaz hány százaléka pontos az elvégzett predikcióknak. Ennek a kiszámítására az alábbi képlet alkalmazására van szükség:

$$Precision = \frac{TP}{TP\ + FP}$$

A képletben a TP (True Positives), a helyesen, az FP (False Positives), pedig a helytelenül detektált objektumok számát adja meg. Azonban ezen értékek meghatározása sem triviális. Klasszifikáció esetében ez viszonylag egyszerű, mert ott csak a jól és rosszul kiértékelt képek számait kell meghatároznunk, azonban objektum detektálásnál már a predikciók minőségét is figyelembe kell vennünk.

Az objektumok helyzetének meghatározására használt határoló dobozok lokációjának és méretének validálását is el kell végeznünk valamilyen módon, ahhoz, hogy értékelhető eredményt kapjunk. Ezt úgy tehetjük meg, ha megmérjük a kiértékelt dobozok és az adott objektum valós helyzetét jelölő doboz (ground truth box) közötti átfedést. Ennek a módszernek a hivatalos neve az IoU (Intersection Over Union), amely nevéből adódóan, a két doboz által lefedett területek metszetének és uniójának hányadosát adja meg az alábbi képlet segítségével:

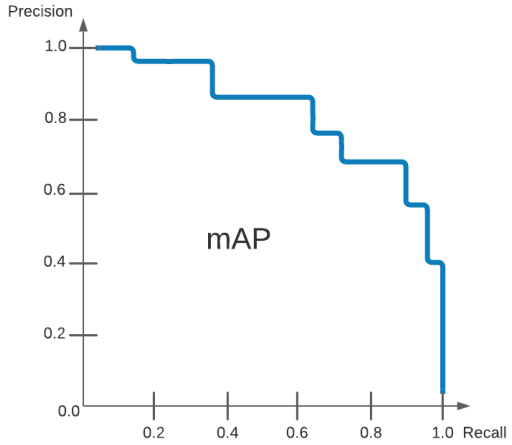
$$IoU = \frac{Area\ of\ intersection}{Area\ of\ union}$$

Az eredményül kapott nulla és egy közötti értéket, egy határértékként szokás használni további mérőszámok meghatározásánál. Legtöbb esetben ez a szám 0.5. Tehát például, ha IoU=0.5, akkor az ennél nagyobb értékek TP-nak, a kisebbek pedig FP-nak számítanak.

Az objektum detektáló feladatoknál egy másik fontos mérőszám a Recall, amely azt határozza meg, hogy az algoritmus mennyire jól képes megtalálni az összes TP-ot. Ez az alábbi képlettel írható le:

$$Recall = \frac{TP}{TP + FN}$$

Itt már bekerül egy új fogalom a képletbe, az FN (False Negatives), amely a detektálás során kihagyott valós objektumok számát foglalja magába. A számított Precision és Recall értékékből felrajzolható egy görbe, amely alatti terület lesz a mAP (ld.: 36. ábra).



36. ábra Precision-Recall görbe, mAP vizualizációja

A fent részletezett metrikákat alkalmazva, az általam használt neurális háló paramétereit és elért eredményeit a 4-es táblázat tartalmazza.

Modell	Tanító halmaz	Detektálás minimum magabiztossága	IoU hatérérték	mAP
Mask R-CNN	***Crowd-AI Mapping challange aerial dataset***	***0.6***	***0.5***	***0.71***

4. táblázat Az épületdetektáló neurális hálózat méréséhez használt paraméterek, illetve az elért pontosság

További manuális tesztelések azt az eredményt mutatták, hogy az algoritmus változatos tartalmú képeken is viszonylag jól működik, azonban a zajt nem minden esetben kezeli jól, illetve a szokatlan formájú épületeket nem minden esetben, vagy csak részben képes beazonosítani.

5.2. Az szolgáltatás end-to-end értékelése

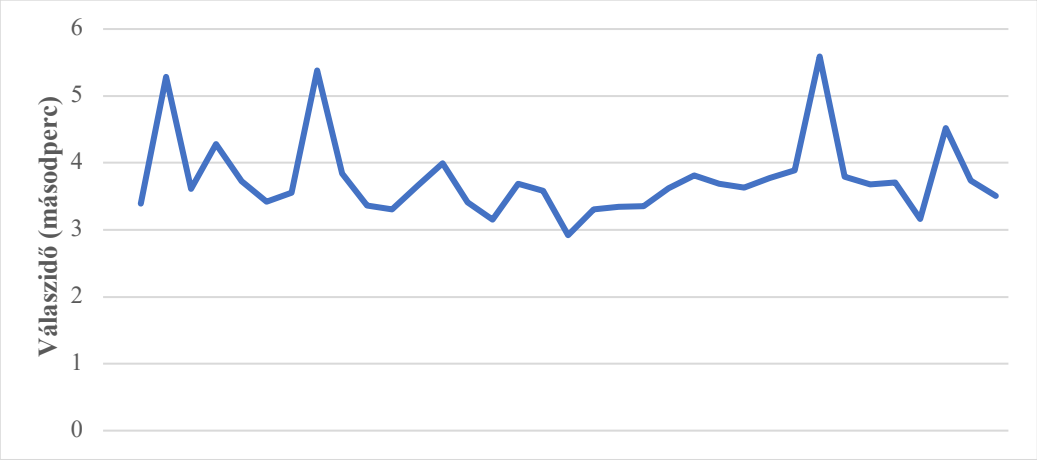
Mivel a szolgáltatás a böngészési élményhez járul hozzá, ezért rendkívül fontos elvárás, hogy közel valós időben legyen képes működni. Ha ezt a feltételt nem tudja teljesíti, lehet bármennyire pontos, nem lesz használható az eredetileg kitalált célra. Emiatt a teszteléskor

figyelt metrika, a felhasználói felülettől induló kérések teljes válaszideje volt. A szolgáltatást, a fejlesztésre használt hardveren teszteltem, aminek konfigurációja az 5. számú táblázatban található.

Processzor	RAM	Videókártya
Intel Core i5 10th Gen, 2GHz Quad-Core	***16 GB***	***Processzorba integrált***

5. táblázat A dolgozat írása közben rendelkezésemre álló hardver konfiguráció

A külső videókártya hiánya nagyban rontja az épületdetektáló alkalmazás futási idejét, ugyanis nagy számú mátrix művelet elvégzésére sokkal lassabban képes egy átlag CPU, mint az erre a célra specializált grafikai kártyák. Ennek ellenére kifejezetten jó eredményeket értem el, ahogy azt a futási időket vizualizáló grafikon is mutatja (ld.: 37. ábra).



37. ábra Válaszidő grafikon

Jól látható, hogy a kiértékelés átlagosan három és fél másodperc körül mozog. A hirtelen kicsúcsosodások, pedig azoknál a kéréseknél fordultak elő, ahol nagy számban detektált épületeket a modell. Ez azzal magyarázható, hogy az épületek számával együtt nő a feldolgozásukhoz szükséges műveletek száma is. Itt különösen nagy súllyal rendelkeznek a képfeldolgozási műveletek, mint például a maszkok meghatározása.

A felhasználási cél szempontjából, azonban még akár a 8-10 másodperc körüli válaszok is elfogadhatónak számítanak.

6. Továbbfejlesztési lehetőségek

Az objektumdetektáló modellnél esetenként felmerült pontatlanságokon lehetne javítani azzal, ha a tanítás egy sokkal változatosabb adathalmazon történne. Azonban az adathalmaz manuális bővítésének elkészítése rengeteg időt emésztene fel, egyrészt a képek beszerzése, másrészt azok felcímkézése miatt. Azonban már léteznek olyan szolgáltatások, amiken keresztül dedikált címkéző csapatokat bérelhetünk fel igényeink szerint. Ezek árazása a feldolgozandó képek mennyisége, illetve a feladat komplexitása függvényében változhat.

Minden kérés esetében külön detekciót végrehajtani, nagy méretben iszonyú számításigényű lenne, ezért érdemes lenne valamilyen cache réteg bevezetését alkalmazni, annak érdekében, hogy a szolgáltatás skálázható maradjon. Ez a réteg még az objektum detektáló komponens meghívása elé lenne helyezve. Egy kulcs-érték párokat tartalmazó adatszerkezet lenne ideális, amely kulcsként a címet tárolná, ahhoz értékként pedig a területi adatokat. A cache bejegyzések kapnának egy TTL (Time To Live) értéket, ami meghatározza, hogy mennyi ideig maradjanak memóriában tárolva. Ezt az értéket egy viszonylag nagyobb időtartamra lehetne hagyni, például hónapokra, mivel nagyon kicsi a valószínűsége, hogy egy terület ennyi idő alatt drasztikus mértékben változzon. Arra is érdemes gondolni, hogy egy bizonyos címet többféle képpen is meg lehet adni, ha például megváltoztatjuk a tagjai sorrendjét. Emiatt célszerű lenne a cache írását és olvasását a címek közti hasonlóságra alapozni. Ahhoz, hogy ne kerüljenek hamis cache elérések a műveletsorozatba, csak a nagyon magas hasonlósági értékkel rendelkező címek lennének megfeleltethetők egymásnak.

Ahhoz, hogy a rendszer felhasználása jól kontrollálható és követhető legyen, szükség lenne felhasználó kezelésre. A szolgáltatás igénybevétele előzetes regisztrációhoz lenne kötve és csak bejelentkezés után lenne elérhető a funkcionalitás. Ez nem csak a monetizáció, hanem a felhasználói élmény személyre szabhatósága szempontjából is rendkívül hasznos lépés lenne. Így például felhasználási céltól függően, lehetővé válna a személyre szabott zsúfoltságot eldöntő értékelési határok beállítása is. Ennek beállításához viszont már szükség lenne egy adminisztrátori felületre, ahol a felhasználók tetszés szerint személyre szabhatnák az állítható paramétereket. Ehhez viszont már célszerű lenne egy külön weboldalt is létrehozni, amin keresztül akár a regisztrációs folyamatot is át lehetne vezetni. A termék jövője szempontjából szükségszerű lenne egy weboldalakba integrálható megoldás is, amit például az ingatlan értékesítő portálok tudnának saját felületükbe beépíteni.

Felhasználók kezelése esetén a tranzakcióik tárolására is szükség lenne. Azonban ez is könnyen megoldható lenne, a Kafka által nyújtott egyszerű modularizálhatóságnak köszönhetően. Az adatfolyamra csak rá kéne kötni egy adattárolót egy Kafka Connector segítségével. Illetve, ha több célba is el kellene juttatni az adatokat, akkor ezt a példát követve egyszerűen bővíthető lenne a rendszer, a már meglévő komponensektől teljesen független módon, azok változtatása nélkül.

Az objektumdetektálás folyamatát fel lehetne gyorsítani azzal, ha dedikált GPU-n futnának a predikciók. Ezzel akár le lehetne szorítani a válaszidőt egy-két másodperc alá, ami már sokkal kényelmesebb élményt nyújtana. Mivel a rendszer architektúrájából adódik, hogy ezek a komponensek egymástól függetlenül futnak, ezért nem is szükséges őket egy hardveren kezelni. Megoldható lenne a szolgáltatás saját hosztolása, azonban a manapság már egyre nagyobb teret nyerő felhő szolgáltatások használatával ez egyszerűbben kivitelezhető és hosszú távon jobban karbantartható lenne.

Irodalomjegyzék

[1] I. C. Education, „Neural Networks,” 2020.

[2] Ian Goodfellow, Yoshua Bengio és Aaron Courville, Deep Learning, The MIT Press, 2016.

[3] Andrew L. Maas, Awni Y. Hannun és Andrew Y. Ng, „Rectifier Nonlinearities Improve Neural Network Acoustic Models,” Computer Science Department, Stanford University, 2013.

[4] C. Unsalan és B. Sirmacek, „Building detection from aerial images using invariant color features and shadow information,” Istanbul, Yeditepe University, 2014.

[5] G. Pasquali, G. C. Iannelli és F. Dell'Acqua, „Building Footprint Extraction from Multispectral, Spaceborne Earth Observation Datasets Using a Structurally Optimized U-Net Convolutional Neural Network,” MDPI, 2019.

[6] Ross Girshick, Jeff Donahue, Trevor Darrell és Jitendra Malik, „Rich feature hierarchies for accurate object detection and semantic segmentation,” 2014.

[7] K.E.A. van de Sande, J.R.R Uijlings, T. Gevers és A.W.M. Smeulders, „Selective Search for Object Recognition,” 2012.

[8] R. Girshick, „Fast R-CNN,” 2015.

[9] Shaoqing Ren, Kaiming He, Ross Girshick és Jian Sun, „Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” 2016.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren és Jian Sun, „Deep Residual Learning for Image Recognition,” 2015.

[11] H. Kaiming, G. Georgia, D. Piotrs és G. Ross, „Mask-RCNN,” 2017.

[12] „labelImg,” [Online]. Available: https://github.com/tzutalin/labelImg.

[13] „SpaceNet,” [Online]. Available: https://spacenet.ai/datasets/.

[14] „DOTA,” [Online]. Available: https://captain-whu.github.io/DOTA/dataset.html. [Hozzáférés dátuma: Június 2020].

[15] „AiCrowd,” [Online]. Available: https://www.aicrowd.com/challenges/mapping-challenge. [Hozzáférés dátuma: December 2020].

[16] Matterport, „matterport/MaskRCNN,” [Online]. Available: https://github.com/matterport/Mask_RCNN. [Hozzáférés dátuma: December 2020].

[17] „kafka-python,” [Online]. Available: https://github.com/dpkp/kafka-python/blob/master/docs/index.rst. [Hozzáférés dátuma: Szeptember 2021].

[18] „Apache Kafka,” [Online]. Available: https://kafka.apache.org/documentation/. [Hozzáférés dátuma: 07 2021].

[19] „Google Charts,” [Online]. Available: https://developers.google.com/chart. [Hozzáférés dátuma: Augusztus 2021].

[20] „Chart.js,” [Online]. Available: chartjs.org. [Hozzáférés dátuma: Augusztus 2021].

[21] „AmCharts,” [Online]. Available: amcharts.com. [Hozzáférés dátuma: Augusztus 2021].

[22] „COCO Detection Evaluation,” [Online]. Available: https://cocodataset.org/#detection-eval. [Hozzáférés dátuma: November 2021].