# Model umjetne inteligencije za detekciju popunjenosti parkirnih mjesta

**Severinski, Karlo**

**Master's thesis / Diplomski rad**

**2022**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **University of Rijeka, Faculty of Engineering / Sveučilište u Rijeci, Tehnički fakultet**

*Permanent link / Trajna poveznica:* https://urn.nsk.hr/urn:nbn:hr:190:799409

*Rights / Prava:* Attribution 4.0 International/Imenovanje 4.0 međunarodna

*Download date / Datum preuzimanja:* **2024-07-19**

*Repository / Repozitorij:*

Repository of the University of Rijeka, Faculty of Engineering

SVEUČILIŠTE U RIJECI
**TEHNIČKI FAKULTET**
Diplomski sveučilišni studij elektrotehnike

Diplomski rad

# Model umjetne inteligencije za detekciju popunjenosti parkirnih mjesta

Rijeka, rujan 2022.

Karlo Severinski
0069079586

SVEUČILIŠTE U RIJECI
**TEHNIČKI FAKULTET**
Diplomski sveučilišni studij elektrotehnike

Diplomski rad

# Model umjetne inteligencije za detekciju popunjenosti parkirnih mjesta

Mentor: prof.dr.sc. Zlatan Car

Rijeka, rujan 2022.

Karlo Severinski
0069079586

**SVEUČILIŠTE U RIJECI**
**TEHNIČKI FAKULTET**
POVJERENSTVO ZA DIPLOMSKE ISPITE

Rijeka, 21. ožujka 2022.

Zavod: **Zavod za automatiku i elektroniku**
Predmet: **Primjena umjetne inteligencije**
Grana: **2.03.06 automatizacija i robotika**

# ZADATAK ZA DIPLOMSKI RAD

Pristupnik: **Karlo Severinski (0069079586)**
Studij: Diplomski sveučilišni studij elektrotehnike
Modul: Automatika

Zadatak: **Model umjetne inteligencije za detekciju popunjenosti parkirnih mjesta/AI based model for parking space occupancy detection**

Opis zadatka:

Predstaviti i opisati suvremene algoritme za detekciju objekata na slikama. Iskoristiti jedan od YOLO modela detekcije objekata temeljenog na umjetnoj inteligenciji i realizirati detekciju automobila. Realizirati algoritam za potrebe detekcije popunjenosti parkirnih mjesta. Komentirati dobivene rezultate.

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Zadatak uručen pristupniku: 21. ožujka 2022.

Mentor:

Prof. dr. sc. Zlatan Car

Predsjednik povjerenstva za diplomski ispit:

Prof. dr. sc. Viktor Sučić

## Izjava o samostalnoj izradi rada

Izjavljujem da sam samostalno izradio ovaj rad uz pomoć mentora prof. dr. sc. Zlatana Cara i asistena dr. sc. Ivana Lorencina

Rijeka, rujan 2022.

Karlo Severinski

# Zahvala

Zahvaljujem se svim onima koji su mi dali podršku za vrijeme studija. Posebno bih se htio zahvaliti mentoru prof. dr. sc. Zlatanu Caru i asistentu dr. sc. Ivanu Lorencinu na iskazanoj pomoći pri izradi ovog diplomskog rada. Isto tako, htio bi zahvaliti tvrtki d3vlab d.o.o. na ustupljenim podacima za izradu diplomskog rada.

# Sadržaj

# Popis slika

*Popis slika*

# Popis tablica

# Poglavlje 1

# Uvod

## 1.1 Cilj rada

Kao što je navedeno u zadatku, cilj ovog diplomskog rada je realizacija modela umjetne inteligencije za detekciju objekata. Ideja je razviti model umjetne inteligencije i algoritam detekcije praznih i popunjenih parkirnih mjesta korištenjem razvijenog modela.

Za realizaciju navedenog modela potrebno je istražiti koji su modeli primjereni za ovu primjenu. Isto tako, potrebno je pripremiti set podataka za treniranje, testiranje i validaciju modela. Potrebno je procijeniti točnost modela na realnim primjerima detekcije.

Glavni cilj rada je razvijanje takvog algoritma detekcije popunjenosti parkirnih mjesta kako bi se on mogao koristiti na nekom od web servisa, to jest kako bi on bio primjenjiv na realnom internetskom servisu i u realnom vremenu. Zahtjevi za takav sustav su prije svega da bude brz i točan, a u isto vrijeme primjenjiv na svim parkirnim kamerama.

## 1.2 Primjeri riješenja problema

Danas se popunjenost parkirnih mjesta generalno izvodi putem sustava koji se sastoje od nekog mikroservisa (kao što su mikrokontroleri koji komuniciraju s nekom centralnom jedinicom) i senzora prisutnosti (induktivni senzori, ultrazvučni senzori...) [1, 2].

Napretkom umjetne inteligencije razvijeni su i sustavi zasnovani na neuronskim mrežama. Takvo modernije rješenje zahtjeva model umjerne inteligencije koji na neki način procjenjuje prisutnost automobila na slici. Uz model umjetne inteligencije potrebno je imati i kamere dovoljno dobre rezolucije kako bi se mogla vršiti dovoljno precizna detekcija [3].

Kako je ranije navedeno i ovaj rad će se baviti tom tematikom.

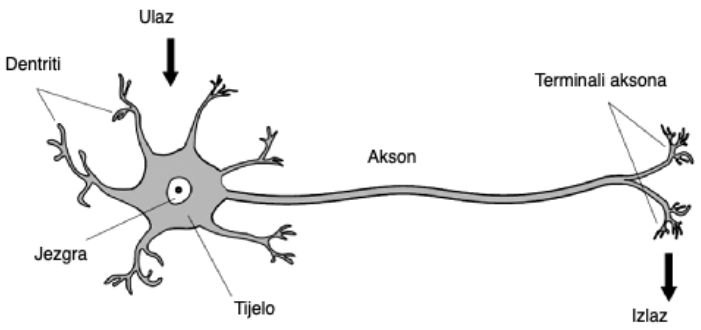
# Poglavlje 2

# Umjetne neuronske mreže

## 2.1 Umjetni neuroni

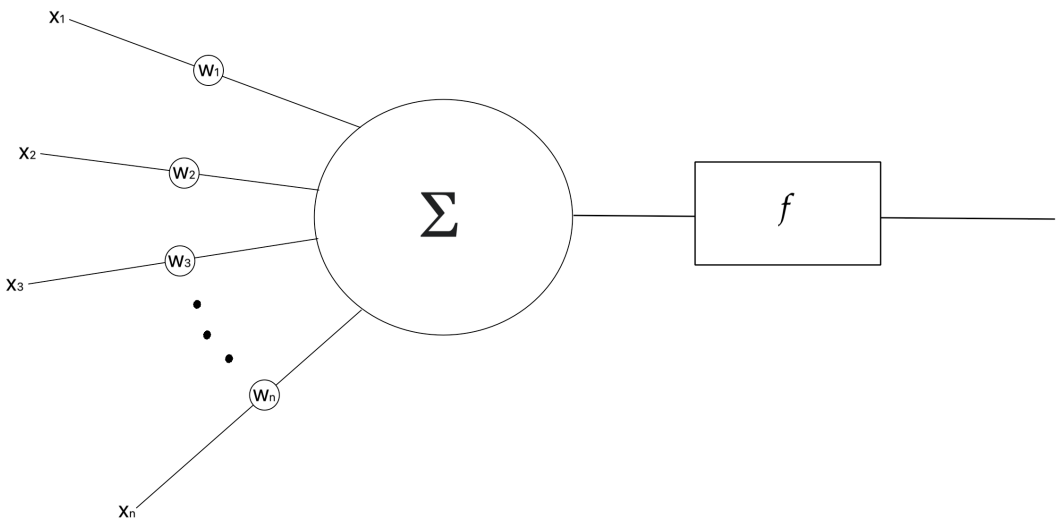
Umjetni neuron glavni je sastavni dio umjetne neuronske mreže. Inspiracija za umjetni neuron nastala je iz biološkog neurona. Umjetni neuron zapravo simulira rad stvarnog biološkog neurona [4].

Biološki neuron prikazan je na Slici 2.1. Može se primijetiti kako se biološki neuron u osnovi sastoji od tijela neurona i aksona. Dendriti primaju signal s drugih neurona, a zatim jezgra neurona "obradi primljeni podatak". Zatim se podatak pomoću aksona šalje na terminale aksona koji prosljeđuju podatak prema drugim neuronima.




Slika 2.1 Biološki neuron [5]

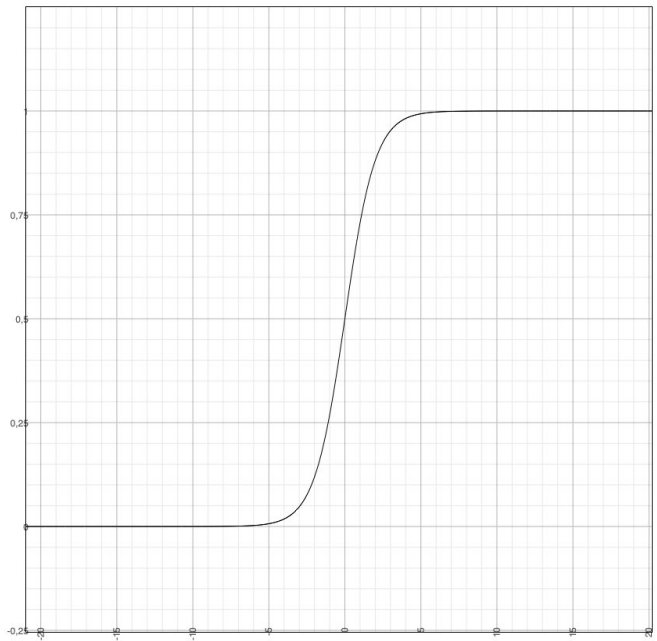
Umjetni neuron radi na sličnom principu. Na slici 2.2. prikazan je matematički model umjetnog neurona.



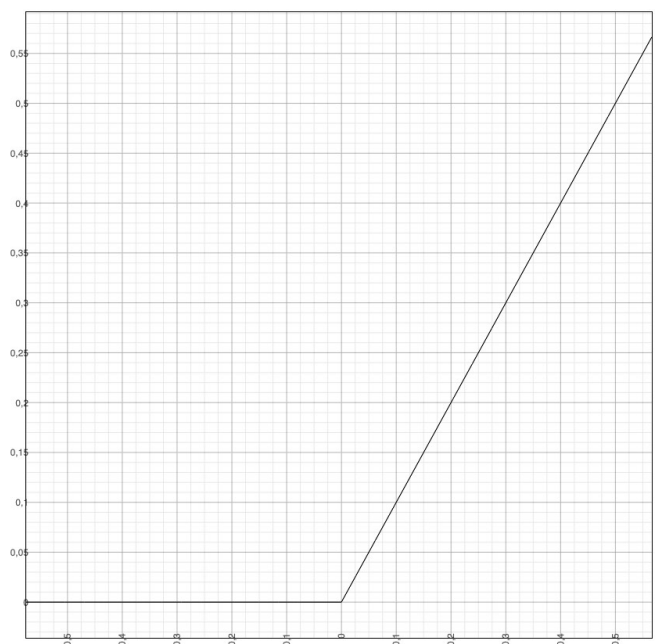

Slika 2.2 Matematički model umjetnog neurona

Ulazni podaci su prikazani kao članovi $x_n$ i oni predstavljaju ulazne signale dobivene s drugih neurona. Zatim se ulazni signali množe s težinskim faktorima $w_n$ koji predstavljaju jakost sinapsi neurona. Svi ti signali nakon množenja dolaze na zbrajalo koje se ponaša kao tijelo neurona, dakle obrađuje podatke. Prijenosna funkcija, to jest aktivacijska funkcija se ponaša kao akson, prosljeđuje obrađeni signal prema izlazu [4]. Može se primijetiti značajna analogija između biološkog i umjetnog neurona.

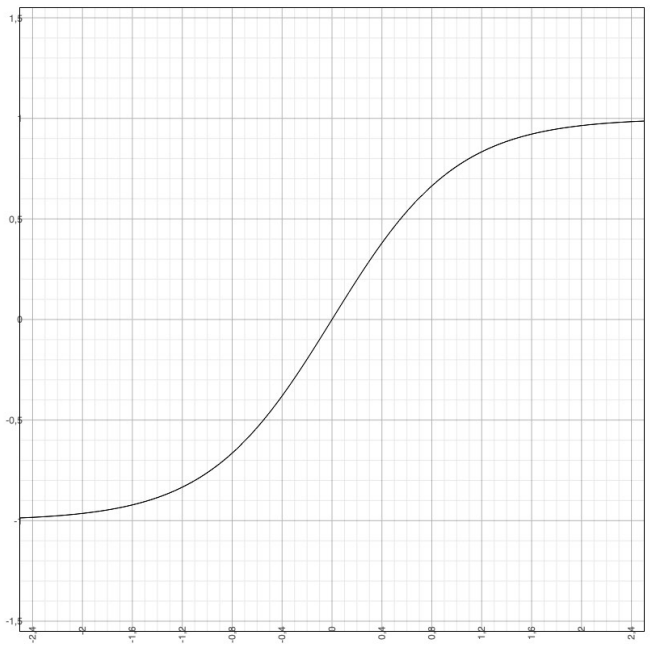
Aktivacijske funkcije neurona mogu biti različitog tipa. Najčešće korištene aktivacijske funkcije su sigmoidalna aktivacijska funkcija prikazana na Slici 2.3, ReLu aktivacijska funkcija prikazana na Slici 2.4, tangens hiperbolna aktivacijska funkcija prikazana na Slici 2.5. i *leaky* ReLu aktivacijska funkcija prikazana na Slici 2.6.



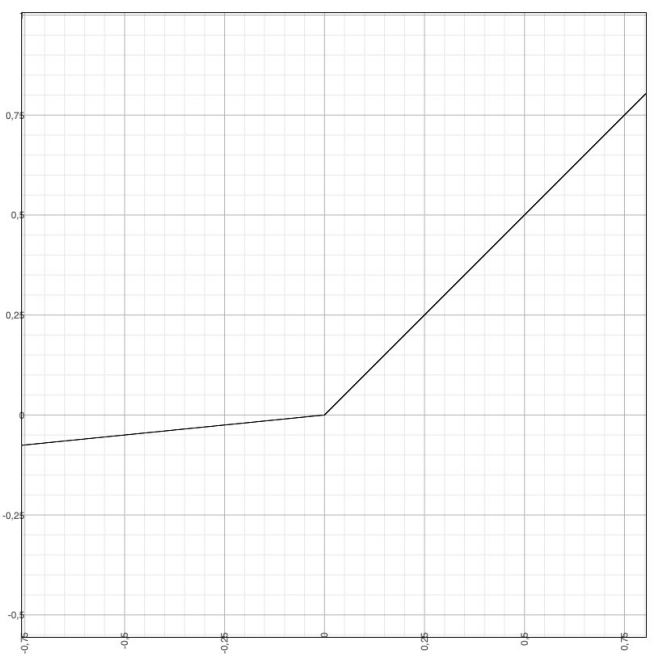

Slika 2.3 Sigmoidalna aktivacijska funkcija




Slika 2.4 ReLu aktivacijska funkcija




Slika 2.5 Tangens hiperbolna aktivacijska funkcija




Slika 2.6 *Leaky* ReLu aktivacijska funkcija

Matematička jednadžba sigmoidalne aktivacijske funkcije prikazana je jednadžbom

$$f(x) = \frac{1}{1 + e^{-x}}, \tag{2.1}$$

te se ona još naziva i logističkom aktivacijskom funkcijom. Sljedeća po redu ReLu funkcija dana je jednadžbom

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}. \tag{2.2}$$

Jednadžba tangensa hiperbolnog dana je jednadžbom

$$f(x) = \tanh = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \tag{2.3}$$

a jednadžba *leaky* ReLu funkcije dana je kao

$$f(x) = \begin{cases} x, & x \geq 0 \\ a \cdot x, & x < 0 \end{cases}. \tag{2.4}$$

Kod *leaky* ReLu aktivacijske funkcije, faktor $a$ se odabire tako da bude što manji, to jest da funkcija prije $x = 0$ ima što manji vertikalni nagib. U slučaju sa Slike 2.6. vrijednost parametra $a$ postavljenja je na vrijednost 0.1.

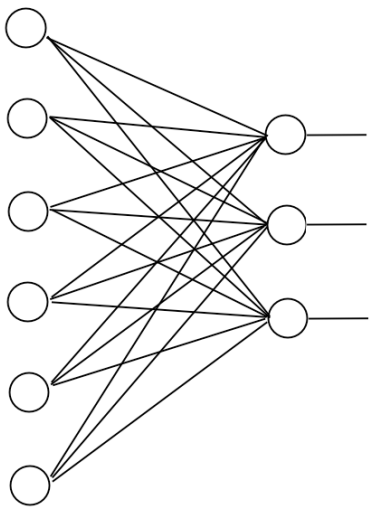
## 2.2 Neuronske mreže

U ovom potpoglavlju razmotrit će se razlike između različitih temeljnih struktura neuronskih mreža i kakvo je njihovo treniranje te njihova točnost. Uglavnom se neuronske mreže dijele na jednoslojne, višeslojne i ljestvičaste neuronske mreže te neuronske mreže s povratnim vezama [4].

Isto tako, u ovom poglavlju razmotrit će se jednoslojne i višeslojne neuronske mreže. Bit će govora i o dubokim neuronskim mrežama (*deep neural networks*).

### 2.2.1 Jednoslojne neuronske mreže

Jednoslojna neuronska mreža predstavlja najjednostavniji oblik neuronske mreže u kojoj postoji samo jedan sloj ulaznih čvorova koji šalju ulaze pomnožene težinskim faktorima sljedećem sloju prijemnih čvorova, ili u nekim slučajevima, jednom primajućem čvoru [6]. Izgled jednoslojne neuronske mreže prikazan je na Slici 2.7.

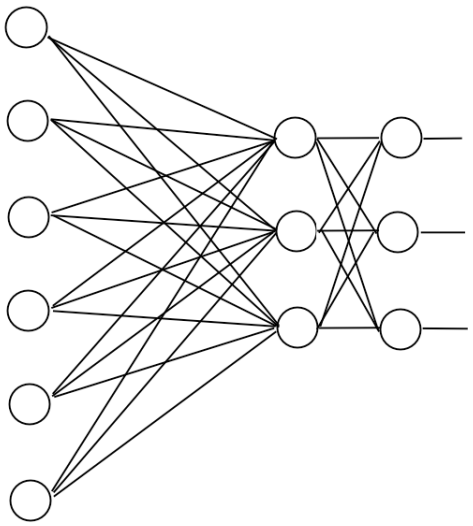


Slika 2.7 Jednoslojna neuronska mreža

Na Slici 2.7. prikazana je jednoslojna neuronska mreža sa 6 ulaznih neurona i 3 izlazna neurona. Bitno je primijetiti da kod ovakvih struktura nema takozvanih skrivenih slojeva neurona. Neuronske mreže s jednim ili više skrivenih slojeva nazivaju se višeslojne neuronske mreže [7].

### 2.2.2 Višeslojne neuronske mreže

Kao što je već navedeno, višeslojne neuronske mreže se sastoje od ulaznog i izlaznog sloja neurona, a između njih nalazi se barem jedan skriveni sloj neurona. Na Slici 2.8. se može vidjeti primjer višeslojne neuronske mreže.



Slika 2.8 Višeslojna neuronska mreža

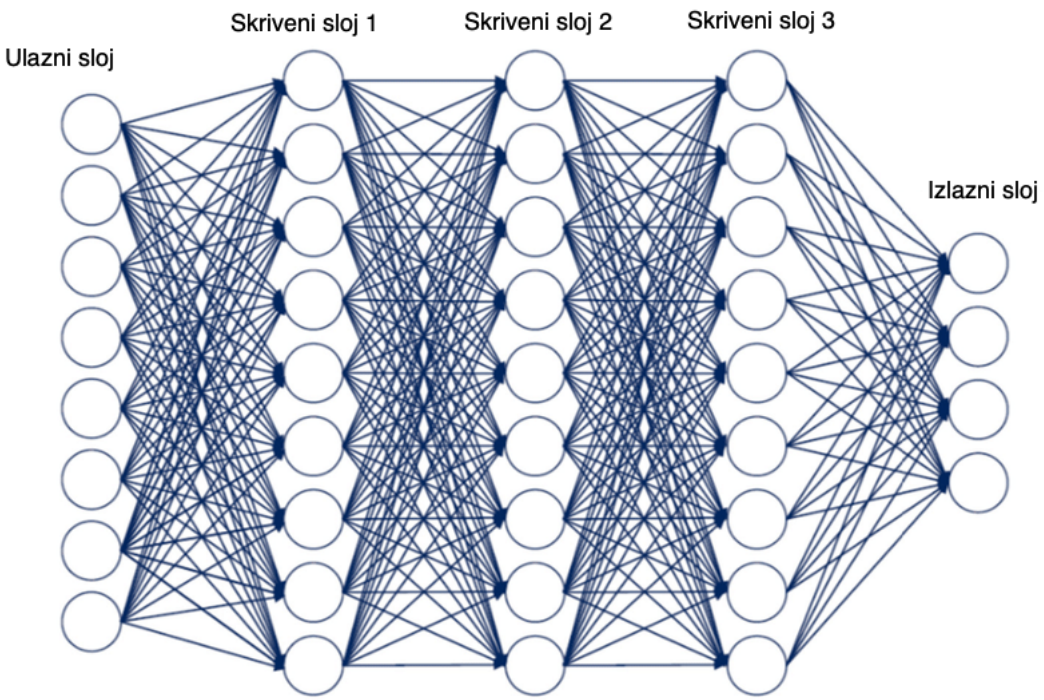
Prikazana višeslojna neuronska mreža sastoji se ulaznog sloja sa 6 neurona, skrivenog sloja od 3 neurona i izlaznog sloja od 3 neurona.

Višeslojne neuronske mreže imaju prednost nad jednoslojnim neuronskim mrežama jer se pomoću njih može trenirati za nelinearne modele i omogućuju treniranje u realnom vremenu. Nedostatak im je složenija struktura i za treniranje ovakvim modela ne postoji recept, već je potrebno mijenjati parametre slojeva na slučajan način kako bi se dobili što bolji izlazni rezultati. Međutim, danas se višeslojne neuronske mreže koriste češće nego jednoslojne neuronske mreže zbog očitih prednosti. Štoviše, danas se za treniranje modela umjetne inteligencije koriste takozvane duboke neuronske mreže (deep neural networks) koje se sastoje od velikog broja skrivenih slojeva. Takve se neuronske mreže koriste i u detekciji objekata, što je na posljetku i tema ovog diplomskog rada. O primjeni dubokih neuronskih mreža u detekciji objekata bit će govora nešto kasnije kroz ovo poglavlje.

### 2.2.3 Duboke neuronske mreže

Duboke neuronske mreže su vrsta višeslojnih neuronskih mreža koje se sastoje od velikog broja skrivenih slojeva. Najčešće su potpuno povezane, što znači da su im svi

slojevi međusobno povezani. Takva izvedba višeslojnih neuronskih mreža dovodi do velike kompleksnosti strukture neuronske mreže. Primjer strukture duboke neuronske mreže dan je na Slici 2.9.




Slika 2.9 Duboka neuronska mreža [8]

Zbog svojih dobrih svojstava duboke neuronske mreže pogodne su i koriste se u poljima kao što su klasifikacija slika, prepoznavanje govora ili obrada prirodnog govora. Zbog učenja s dubokim neuronskim mrežama ova područja su postigla veliku točnost koja čak može parirati čovjeku [9].

Kako je već navedeno, ovi se tipovi neuronskih mreža mogu koristiti za digitalnu obradu govora, to jest sintezu, prepoznavanje i obradu prirodnog govora. Jedan od primjera sinteze govora je *Merlin Speech Synthesis*. Merlin je sustav za sintezu govora temeljen na neuronskoj mreži razvijen u Centru za istraživanje tehnologije govora (CSTR), Sveučilišta u Edinburghu. Merlin se zapravo sastoji od skupa alata za izgradnju modela duboke neuronske mreže za statističku parametarsku sintezu govora [10].

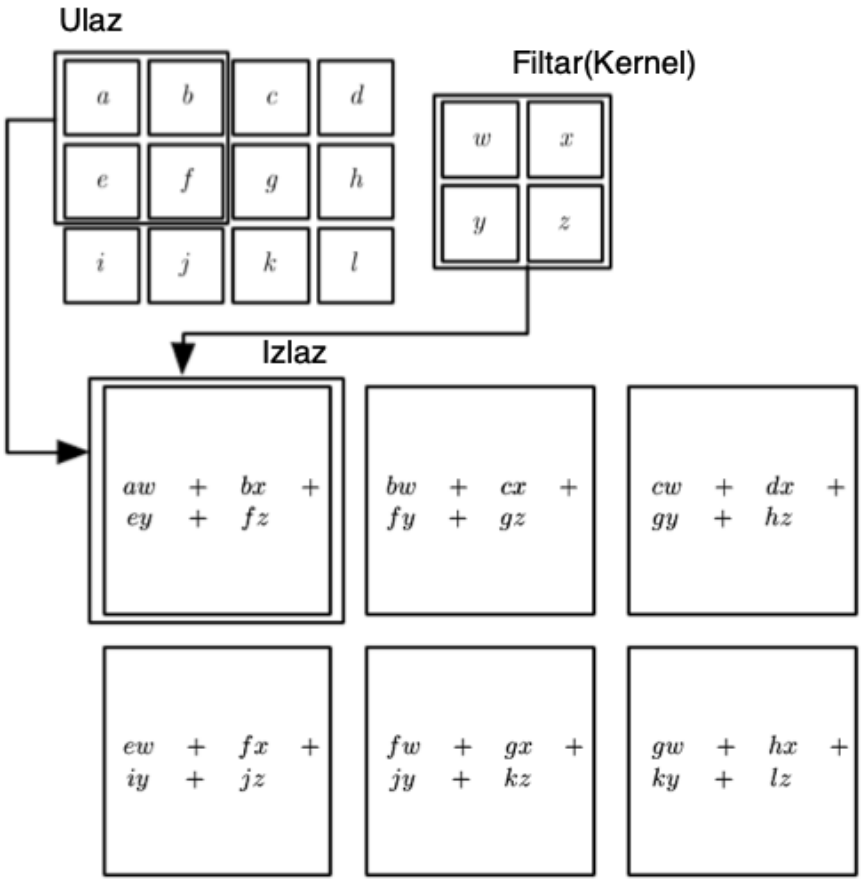
Kako je za ovaj diplomski rad bitna obrada slike i klasifikacija objekata na slici, bitno ih je spomenuti u kontekstu dubokih neuronskih mreža. Za obradu slike najpogodnije su konvolucijske neuronske mreže. Konvolucijske neuronske mreže bit će detaljnije obrađene u sljedećem poglavlju.

## 2.3 Konvolucijske neuronske mreže

Konvolucijske neuronske mreže (CNN) slične su osnovnim strukturama neuronskih mreža, međutim nadograđene su s drugim tipovima slojeva. Sastoje se od neurona koji rade samooptimizaciju kroz učenje. Svaki od neurona prima ulazni podatak te nad njime radi neku operaciju. Kao i kod ostalih umjetnih neuronskim mreža ideja je putem različitih operacija koje neuroni vrše na ulaznim podacima dobiti težinske vrijednosti pomoću kojih će se kasnije vršiti detekcija [11]. Konvolucijske neuronske mreže se uglavnom koriste za procesiranje nestrukturiranih podataka. Nestrukturirani podatak je na primjer piksel ili audio podatak, pa se time može vidjeti zašto je CNN pogodan za obradu slike i govora.

Osnovna arhitektura konvolucijske neuronske mreže sastoji se od ulaznog sloja, konvolucijskog sloja, aktivacijske funkcije i sloja sažimanja. U arhitekturi bitno je svojstvo potpune povezanosti slojeva, gdje je svaki neuron povezan sa svim neuronima iz prethodnog sloja. Ulazni sloj predstavlja sliku. Konvolucijski sloj primjenjuje konvoluciju slike i željenog filtra, time se stvara mapa značajki koja u suštini pokazuje gdje se promatrana značajka nalazi na slici. Aktivacijska funkcija je klasična aktivacijska funkcija koja za neku ulaznu vrijednost daje određenu izlaznu vrijednost, samo se ona sada ne koristi na strukturiranom podatku već na nestrukturiranom podatku. To znači da se aktivacijska funkcija koristi na svakoj pojedinoj mapi značajki. Kako bi se smanjio utjecaj pomaka objekata na prepoznavanje značajki u prostoru koristi se sloj sažimanja. Sloj sažimanja sažima veličinu aktivacijske mape i time smanjuje utjecaj pomaka značajki u prostoru [12].

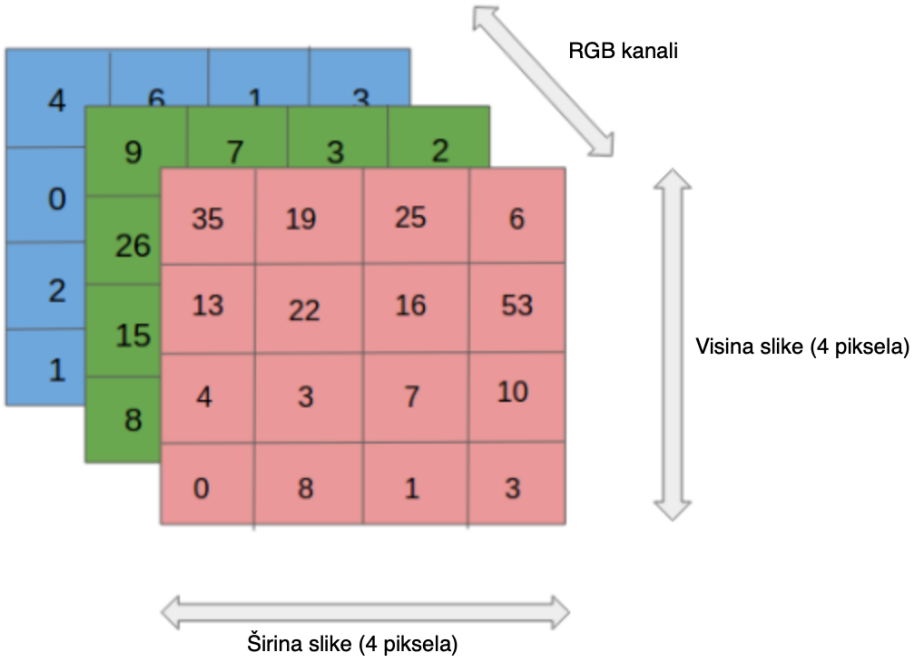
Konvolucijski filtar se najčešće u literaturi naziva kernel. Prikaz rada konvolucijskog filtra dan je na Slici 2.10.




Slika 2.10 Primjer 2D konvolucije [13]

Na jednostavnom primjeru sa Slike 2.8. može se vidjeti djelovanje filtra. U ovom primjeru govorimo isključivo o 2D filtriranju. Ulazni podaci (slika) su dimenzija 3x4, a filtar (*kernel*) je oblika 2x2. Filtriranje se vrši pomoću matematičke operacije konvolucije, gdje kernel prolazi preko slike. Vrijednost piksela slike i kernela se množe te zatim zbrajaju. Time nastaje nova mapa značajki koja se na Slici 2.8. vidi ispod ulaznih podataka i kernela. Nova mapa značajki sada je dimenzija 2x3.

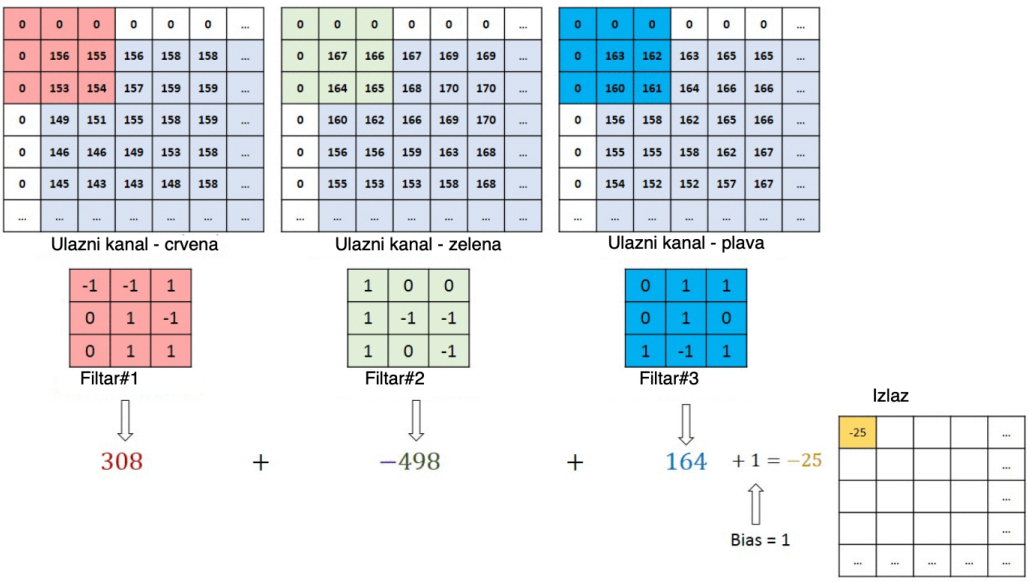
Međutim, ovaj proces nije tako jednostavan kada se govori o slikama formata kao što su JPEG ili PNG. Kod tih vrsta slika konvolucija ne može biti samo dvodimenzionalna već je potrebno u obzir uzeti komponente boje u slici. Time slika više nije samo 2D objekt s visinom i širinom, već ona postaje 3D objekt s visinom, širinom i 3 sloja dubine (Slika 2.11).




Slika 2.11 Slika kao 3D objekt [14]

Zbog toga je potrebno postaviti 3 kanala kernela koji vrše konvoluciju nad slikom u svakom od prikazanih slojeva. Ovo je prikazano na Slici 2.12.

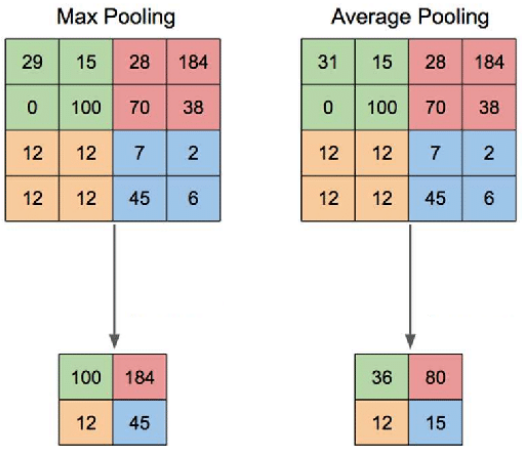
Svako novo filtriranje s novim filtrom stvara novu aktivacijsku mapu. Na primjer, ako bi se neka slika filtrirala 6 puta s nekim određenim filtrom dobilo bi se 6 aktivacijskih mapa. One zapravo samo govore koji se neuroni moraju aktivirati [12]. Pri odabiru filtara najčešće se pazi na to da filtri budu manjih dimenzija od slike koju filtriraju.




Slika 2.12 Konvolucija na RGB slici [14]

Još je potrebno nešto reći o sloju sažimanja (*Pooling layer*). Ideja sloja sažimanja je da smanji dimenzije prethodne aktivacijske mape, a da značajke ostanu iste. Time se smanjuje potreba za većom moći procesiranja podataka i smanjuje se osjetljivost detekcije značajki pomicanjem istih na slici [14, 15]. Razlikuju se 2 vrste *pooling layer*-a, *max-pooling* i *average-pooling*.

*Max-pooling* izračunava maksimalnu vrijednost za svaki *patch* mape značajki, dok *average-pooling* izračunava srednju vrijednost za svaki *patch* mape značajki [15]. Primjer rada *max-pooling* i *average-pooling* slojeva dan je na Slici 2.13.




Slika 2.13 Način rada slojeva sažimanja [16]

Ako definiramo da su polja unutar bazena vrijednosti označena s $A_x$, a rezultat nakon sažimanja s $M_{pool}$, tada matematički mozemo zapisati da za *max-pooling* operaciju vrijedi

$$M_{pool} = \max(A_x). \tag{2.5}$$

Analogno za *average-pooling* vrijedi da je

$$M_{pool} = \frac{A_0 + A_1 + A_2 + ... + A_{n-1}}{n} = \frac{\sum_0^{n-1} A_x}{n}. \tag{2.6}$$

# Poglavlje 3

# Detekcija objekata

## 3.1 Modeli detekcije objekata

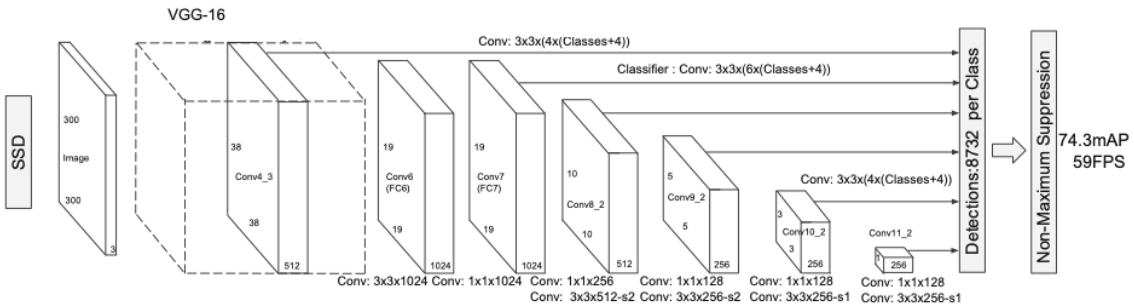
Modeli detekcije objekata temelje se na prethodno objašnjenim konvolucijskim neuronskim mrežama. Uglavnom se dijele na *one-stage* i *two-stage* metode. Poznatiji primjeri *one-stage* modela su SSD, YOLO i RetinaNet, dok su neki od češće korištenih *two-stage* modela *Faster R-CNN*, *Mask R-CNN* i *Cascade R-CNN* [17].

Glavno i najbitnije svojstvo *one-stage* modela detekcije je njegova brzina detekcije. *One-stage* modeli detekcije uglavnom rade brzo i moguće ih je koristiti za detekciju u realnom vremenu. Za razliku od *one-stage* modela, *two-stage* modeli su sporiji i stoga nisu prikladni za korištenje u realnom vremenu. Svoju sporost nadoknađuju time što omogućuju veliku točnost detekcije, to jest dobru lokalizaciju rješenja [17].

Kako je u ovom radu bitna detekcija automobila u realnom vremenu potrebno je žrtvovati veliku točnost detekcije za veliku brzinu detekcije. Zbog toga za rješavanje ovakve vrste problema odabrani su *one-stage* detektori i oni će biti objašnjeni u narednim poglavljima.

## 3.2 SSD

SSD (*single shot detection*) detekcija objekata temelji se na *feed-forward* konvolucijskoj mreži. Ideja je odrediti kolekciju pravokutnika koji označavaju prisutnost objekta na slici. Također, SSD pokazuje kolika je sigurnost detekcije u bodovima od 0 do 1, to jest u postocima od 0% do 100%. SSD model je strukturiran tako da vrši 8732 detekcije po klasi podatka koristeći samo 6 slojeva [18]. Struktura SSD-a dana je na Slici 3.1.




Slika 3.1 Struktura SSD modela [18]

Kako je SSD *one-stage* model, omogućuje veliku brzinu detekcije objekata pa je samim time i validan odabir za rješavanje problema koji se razmatra u ovom radu.

## 3.3 YOLO

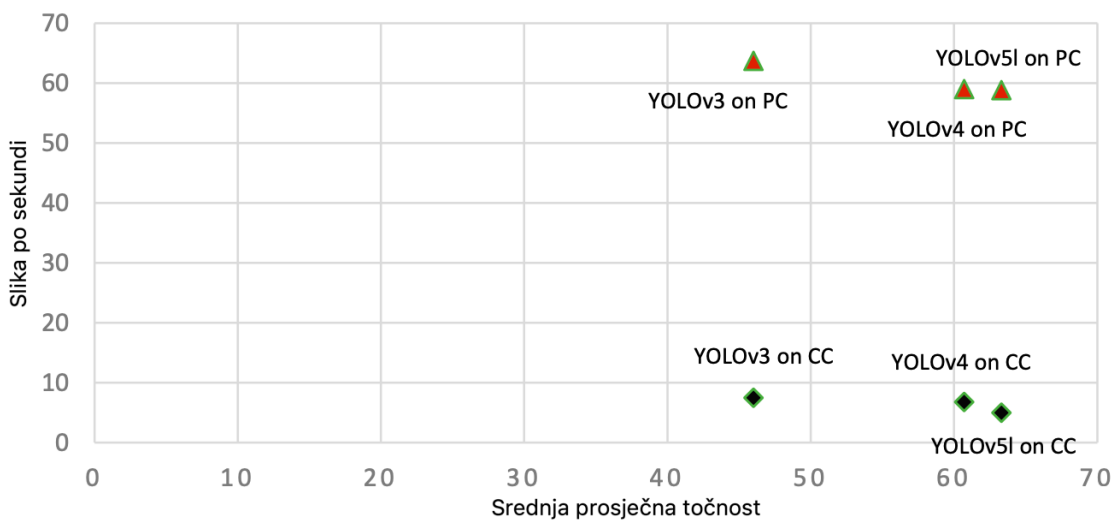
YOLO (*you only look once*) je još jedan od *one-stage* modela detekcije objekata. Kako su YOLO modeli open-source modeli i moguć je uvid u njihovu strukturu, razvijen je veliki broj različitih modela od strane mnogih autora. Isto tako, zbog toga što su open-source dolazi do stalnog napretka u njihovoj brzini i točnosti detekcije. Time se osigurava stalna mogućnost ažuriranja modela koji se koristi unutar algoritma detekcije automobila novim i boljim modelima detekcije. Ovo je veliki plus kod odabira modela koji će se koristiti. Kako je YOLO isto *one-stage* model osigurava veliku brzinu detekcije uz smanjenu, ali i dalje visoku točnost detekcije. Temelji se

na sličnom principu kako je opisan i SSD. U ovom će radu biti razmotreni neki noviji i brži YOLO modeli te njihove arhitekture.

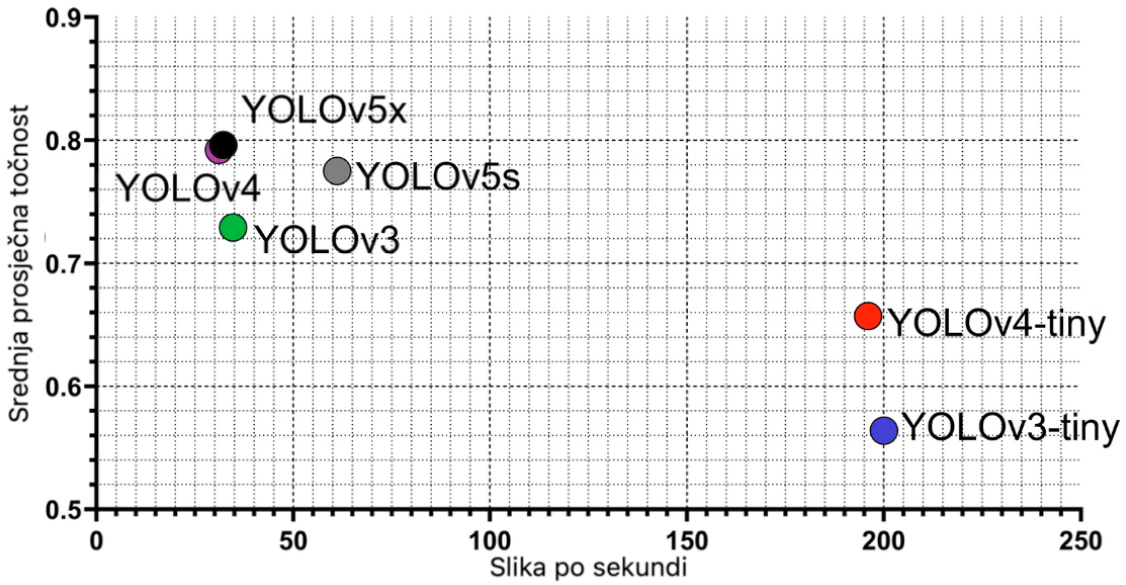
Trenutno postoji 5 verzija YOLO modela. Te verzije su YOLO, YOLOv2, YOLOv3, YOLOv4 i YOLOv5. Noviji od njih su YOLOv3, YOLOv4 i YOLOv5 te zbog njihove slične brzine i preciznosti ulaze u uži izbor za rješavanje problematike ovog rada.

Prilikom prvog pokušaja rješavanja ovog problema prvo je uzet u obzir najstariji, već dobro razvijeni, model YOLOv3. Ono što se je pokazalo kao problem nakon što je model treniran i testiran na manjem broju slika testnog seta podataka, je to da puno puta promašuje detekciju očitih objekata koje bi trebao pronaći s velikom točnošću. Dodatnim treniranjem modela nije se znatno poboljšala točnost detekcije. Isto tako, detekcija je bila relativno spora za potrebe detekcije automobila na parkirnim mjestima u realnom vremenu. Nakon toga, prije daljnjeg treniranja ostalih modela, uspoređene su točnost i brzina detekcije YOLOv4 i YOLOv5 modela na već gotovim rješenjima sličnih problema detekcije.

Za potrebe usporedbe uzeti su rezultati iz 2 rada, kao što je prikazano na Slikama 3.2. i 3.3.

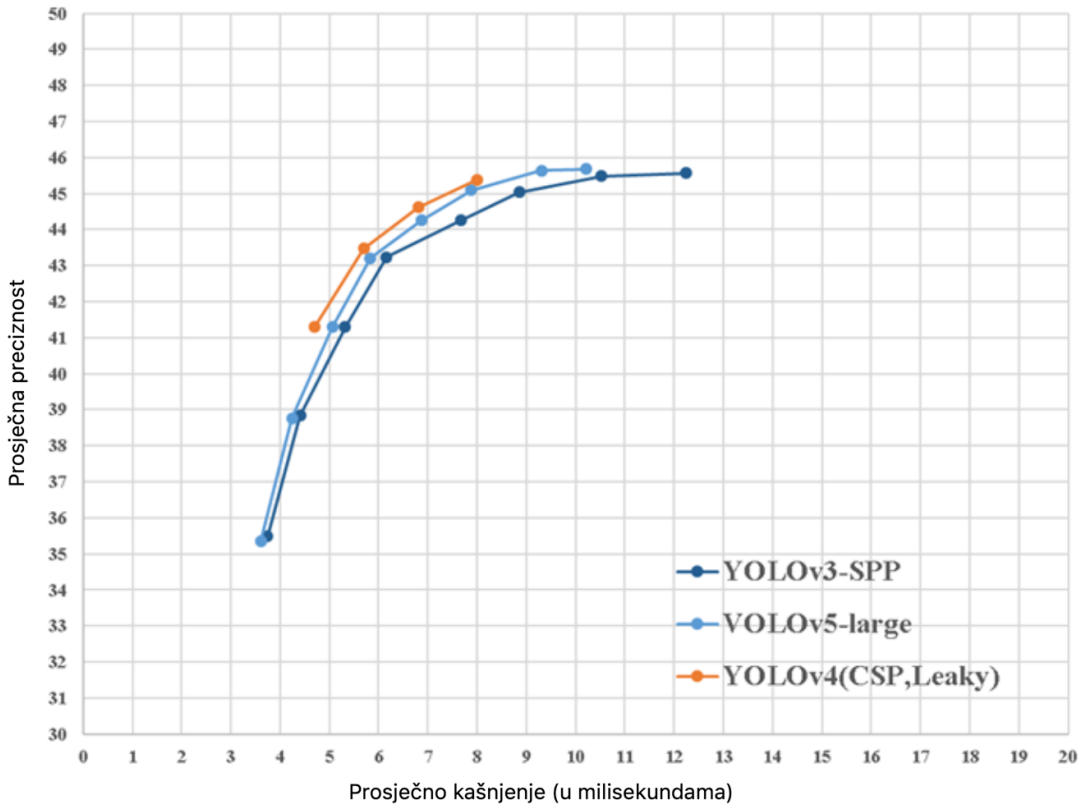



Slika 3.2 Detekcija mjesta sletanja bespilotne letjelice [19]




Slika 3.3 Usporedba detekcije vrsta grožđa [20]

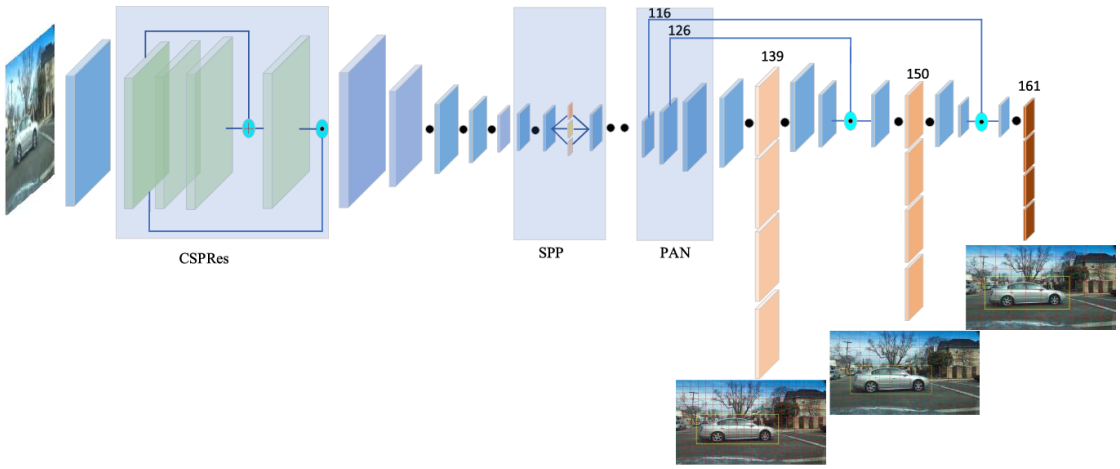
Iz gornjih se podataka vidi da YOLOv5 uglavnom daje bolje rezultate nego što daju ostali modeli. Međutim, uspoređujući rezultate YOLOv4 i YOLOv5 modela može se primijetiti kako se oni ne razlikuju znatno po točnosti i brzini detekcije. Isto tako, bitno je napomenuti kako je YOLOv5 i dalje u razvoju, pa se kod njega javljaju određeni problemi. Primjer takvog problema je prikazan na Slici 3.4.




Slika 3.4 Usporedba YOLOv4 i YOLOv5 modela na istom setu podataka [21]

Ostali modeli prikazani na slikama (YOLOv4-tiny, YOLOv3-tiny itd.) neće se razmatrati u ovom radu, stoga ih se može zanemariti u kontekstu brzine i točnosti detekcije.

Na Slici 3.4. se jasno vidi kako u nekim slučajevima, kada se oba algoritma treniraju s identičnim setom podataka i zatim testiraju, dobe se bolji rezultati s YOLOv4 modelom. Kako YOLOv4 ima sličnu brzinu i kvalitetu detekcije kao noviji YOLOv5, a u njega je uloženo znatno više vremena i duže je razvijan, YOLOv4 je odabran kao najpogodniji od YOLO modela za realizaciju ovog rada. Još je jedino potreno usporediti detekcije YOLOv4 modela sa SSD modelom. Arhitektura korištenog YOLOv4 modela danas je na Slici 3.5.




Slika 3.5 YOLOv4 arhitektura [22]

## 3.4 Usporedba SSD i YOLO modela

Oba modela potrebno je usporediti po najbitnijim značajkama koje bi mogle utjecati na performanse sustava detekcije popunjenosti parkirnih mjesta. Dakle, dva najbitnija su točnost detekcije i brzina. Isto tako, bitno je odabrati onaj model koji daje optimalan omjer obje značajke, a ne koncentrirati se samo na točnost ili samo na brzinu modela. Za početak, dano je objašnjenje rada pojedinog algoritma.

SSD algoritam pokreće konvolucijsku mrežu na ulaznoj slici samo jednom i izračunava mapu značajki. Zatim filtrira mapu značajki kernelom veličine $3 \times 3$ kako bi predvidio granične okvire i vjerojatnost kategorizacije. Isto tako, SSD koristi okvire različitih visina i širina usporedive s *Faster-RCNN* metodom [23].

YOLOv4 algoritam radi na principu regresije. YOLOv4 je regresijski algoritam koja uzima ulaznu sliku i uči vjerojatnost klase s koordinatama *bounding box*-a. To u pravilu znači da su mu ulazni podaci slike i prozori koji označavaju klasu objekta koju treba klasificirati, a izlazni podatak središnja točka detektiranog objekta i sigurnost (vjerojatnost) detekcije između 0 i 1. YOLO svaku sliku dijeli na mrežu od $S \times S$ i svaka mreža predviđa N okvira i vjerojatnost detekcije za pojedini okvir. Vjerojatnost detekcije odražava preciznost okvira i sadrži li okvir zapravo objekt. Dakle, predviđeno je ukupno $S \times S \times N$ kutija. Međutim, većina tih kutija će

imati manju pouzdanost detekcije pa im se najčešće postavlja prag ispod kojeg se detekcija ne smatra validnom. Time se filtriraju sve detekcije koje su pogrešne ili niske pouzdanosti [23]. Iako YOLOv4 algoritam zvuči kompliciraniji i uspoređujući Slike 3.1 i 3.5 izgleda kompliciraniji, zapravo nije. Strukturno je jednostavniji što mu daje veću brzinu detekcije s i dalje velikom točnošću.

Jedna od velikih prednosti SSD modela je točnost detekcije. Dakle, SSD ima veću točnost detekcije od YOLOv4, međutim SSD ima smanjenu točnost u slučajevima kada je objekt na slici manjih dimenzija. Ovdje YOLOv4 radi bolji posao, iako generalno ima manju točnost. Uz veliku točnost SSD model omogućuje i veliku brzinu detekcije objekata, međutim YOLOv4 model ponovno radi bolji posao vezan za brzinu detekcije objekata. YOLOv4 algoritam je zbog svoje velike brzine pogodan za korištenje s autonomnim vozilima (autonomni automobili, bespilotne letjelice itd.) i zbog mogućnosti detekcije manjih objekata s medicinskim sustavima detekcije različitih oblika tumora [24].

Za kraj, potrebno je donesti zaključak koji je algoritam optimalan za korištenje u detekciji automobila na parkiralištu u realnom vremenu. Parkiralište se uglavnom može smatrati sporim sustavom gdje nema puno promjena u nekom vremenskom intervalu, ali isto tako brze promjene su moguće i ako dođe do njih potrebno ih je detektirati. Za brze promjene pogodniji je YOLO model jer je nešto brži od SSD modela. Isto tako, kako je ideja ovog sustava da bude komercijalan i primjenjiv u realnom vremenu, može doći do potreba dodavanja novih značajki sustavu kao što su detekcija brzine automobila, nepropisnog parkiranja, prolaska ljudi i drugih objekata po parkiralištu i drugih. Zbog toga potrebno je omogućiti da sustav potencijalno detektira manje objekte gdje YOLOv4 ima prednost jer, kako je navedeno, SSD ima problem s detekcijom manjih objekata.

Nakon svih iznesenih podataka YOLOv4 model je odabran kao bolja opcija za realizaciju ovakvog sustava.

# Poglavlje 4

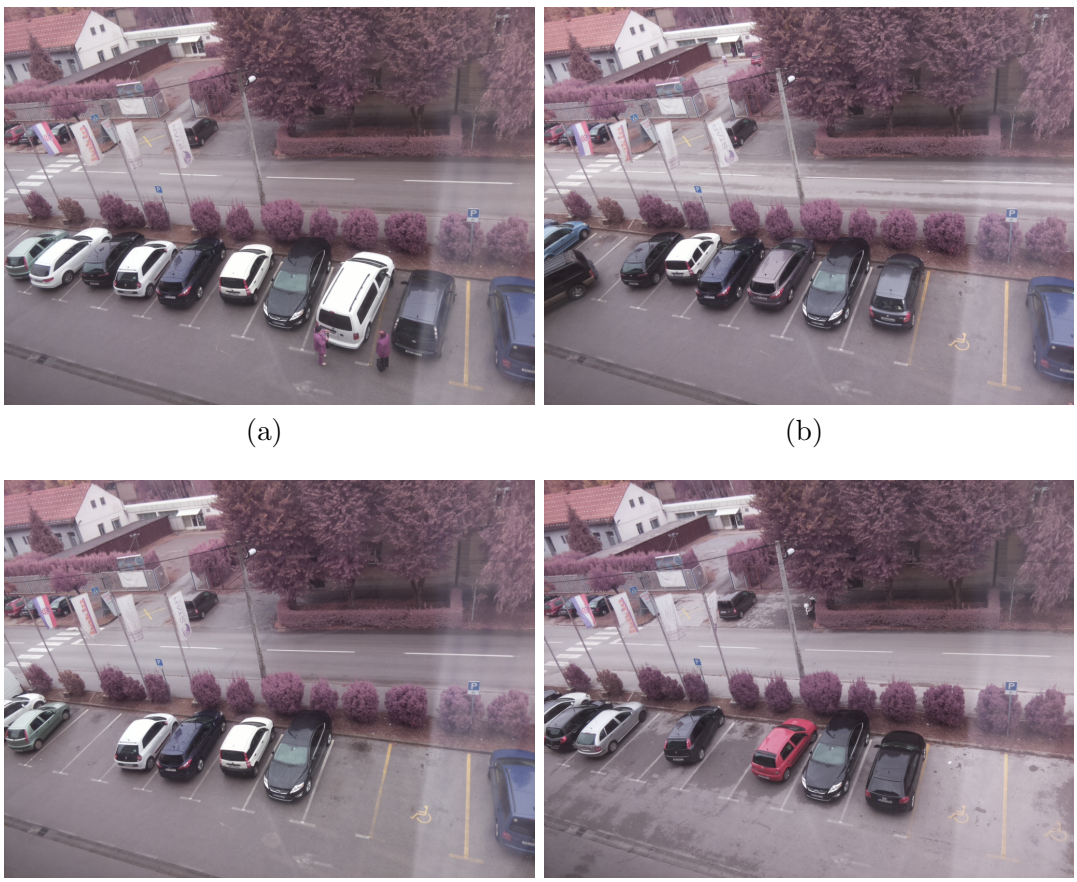
# Izrada YOLOv4 modela

## 4.1 Prikupljanje podataka

Podaci za izradu modela detekcije objekata prikupljani su iz više izvora tijekom dužeg vremenskog perioda. Većinu podataka ustupila je tvrtka d3vlab d.o.o.

Podaci su prikupljani s različitih dostupnih parkirnih i nadzornih kamera čija pozicija i gledanje najviše nalikuje na parkirnu kameru. Dobiveni dataset je zatim prebran tako da se odaberu slike s najviše različitih automobila i parkirnih pozicija kako ne bi došlo do *overfitting*-a modela. Dataset je također ograničen na relativno malen broj podataka radi male dostupnosti navedenih kamera, to jest radi male dostupnosti kvalitetnih kamera s kvalitetnim kutem gledanja.

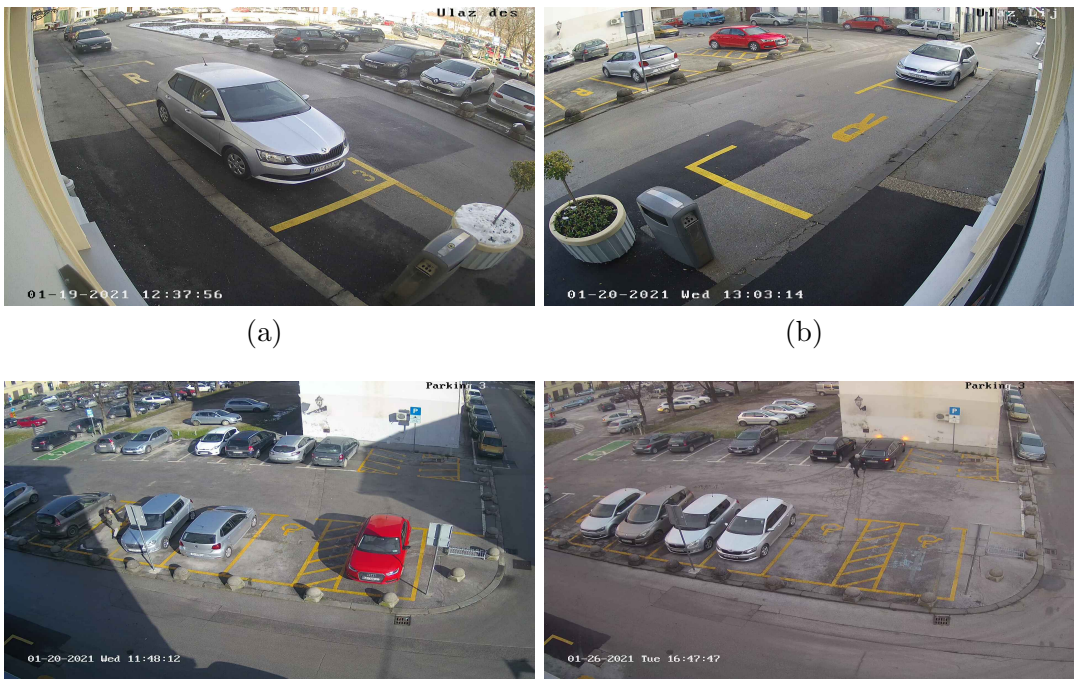
Prilikom samostalnog prikupljanja podataka od strane tvrtke, prije realizacije ovog rada, korištene su dvije dostupne pozicije kamere. Set podataka s IP kamere postavljene u ured tvrtke, gdje su podaci prikupljani u dva navrata i set podataka s IP kamera Karlovačke gradske vijećnice, koja pokazuje na velik i prometan parking ispred gradske uprave. Potpuni set podataka IP kamere s pogledom iz ureda sastoji se od oko 200 slika, a set podataka s kamera ispred gradske uprave sastoji se od oko 900 podataka. Primjer slika iz seta podataka IP kamere s pogledom iz ureda dan je na Slici 4.1, dok je primjer IP kamere gradske vijećnice dan na Slici 4.2.



(a) (b)

(c) (d)

Slika 4.1 Primjer dataseta - IP kamera ured




(a) (b)

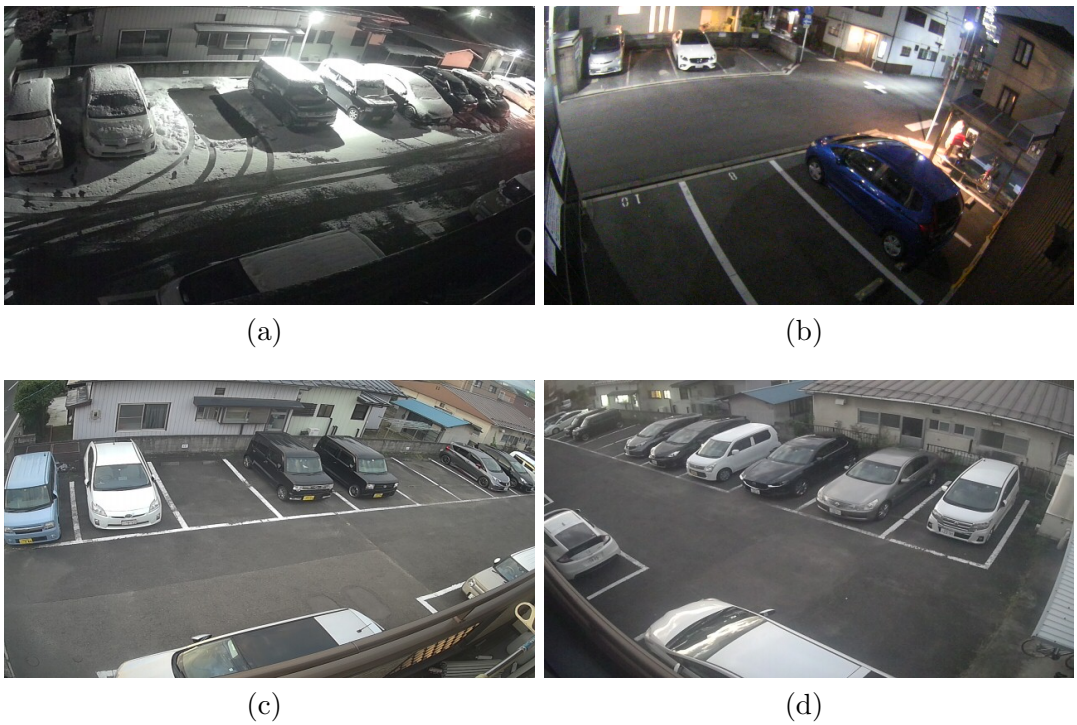
(c) (d)

Slika 4.2 Primjer dataseta - IP kamere gradska viječnica

Glavni problem kod većine podataka uzetih s parkinga ispred gradske uprave je taj što su slike preuzete s IP kamere u nekim manjim vremenskim intervalima, pa su zato slične. Samim time se smanjuje set podataka za treniranje jer je generalno pravilo da slike budu što više različite. Isto tako, bitno je primijetiti da se kod istog parkinga koristi više kuteva kamere, točnije 3. Svi kutevi su pogodni za korištenje i time se ipak set podataka djelomično povećava.

Zbog navedenih problema napravljen je još jedan dataseta. Taj se dataset sastoji od otvorenih internetskih IP kamera koje pokazuju na parkirališta, ali većina nisu parkirne kamere već nadzorne. Kako je jedno od tih parkirališta u naseljenoj ulici, broj sličnih podataka je velik pa se time nešto smanjuje mogućnost slaganja završnog trening dataseta od tih podataka. Kod te kamere je dobro što ima više kuteva gledišta pa se time proširuje dataset. Druga kamera je nadzorna kamera koja se nalazi na općem gradskom parkingu i tu dataset može biti velik, ali je kamera pod malo više nezgodnim kutem pa ne sadrži puno mogućnosti anotacije podataka. Uzet je i manji

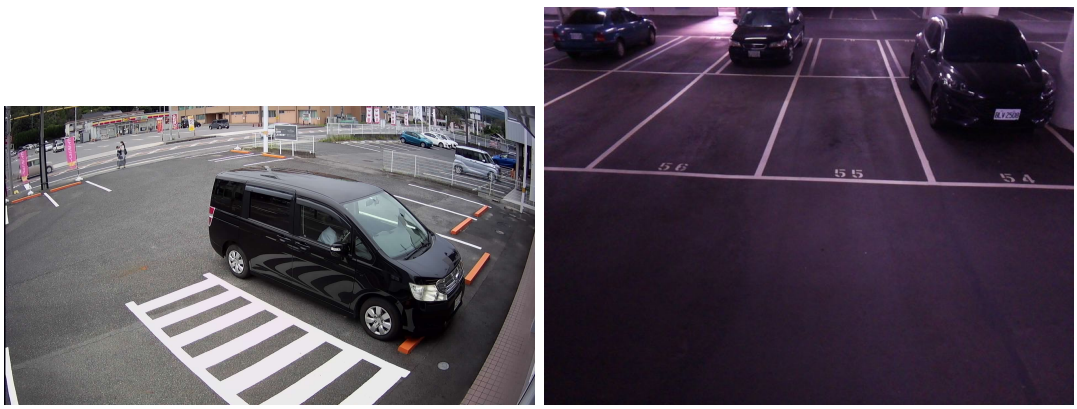
broj podataka s jedne garažne kamere koja je manje prometna i s još jednog gradskog parkirališta koje se nalazi između zgrada. Primjeri podataka dobivenih s internetskih kamera dan je na Slikama 4.3 i 4.4. Generalno se cijeli dataset može podijeliti na slike privatnog parkirališta i javnih parkirališta.



(a) (b)

(c) (d)

Slika 4.3 Primjer dataseta s IP kamere s interneta - Privatno parkiralište



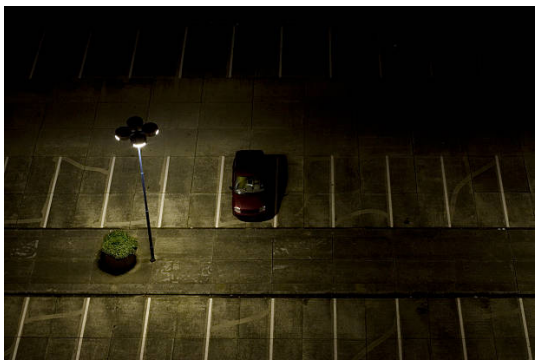

(a) (b)

Slika 4.4 Primjer dataseta s IP kamere s interneta - Javna parkirališta

Za kraj, bilo je bitno odrediti veličinu seta podataka za trening modela detekcije objekata. Podaci su ručno prebrani tako da se izbjegne većina sličnih ili istih slika pri tome gledajući da dataset bude što veći. Također, u dataset su ukomponirane slike uzete po noći kako bi se povećala efektivnost modela tijekom noćnih sati. Iz svih prikupljenih podataka prebrano je 425 podataka koji su najviše različiti. Ti podaci su podijeljeni u trening set od 340 slika i testni set od 85 slika. Također, kako bi se set validirao dodan je set podataka za validaciju. Taj set se sastoji od slika s interneta koje najviše sliče parkirnim kamerama i od par slika na kojima je teže detektirati automobile kako bi se vidjela točnost modela na ne savršenim podacima. Primjeri slika iz validacijskog seta dani su na Slici 4.5.

(a) https://www.dreamstime.com/royalty-free-stock-photo-car-parking-place-view-above-image21981905

(b) https://artikel.rumah123.com/park-and-ride-akan-dibangun-dekat-stasiun-bekasi-dan-stasiun-cikarang-53715



(c) https://www.nomadicnews.com/is-walmart-camping-safe-2022/


(d) https://www.nextech.sk/a/Smartcity–Inteligentne-riesenie-odstrani-problem-s-parkovanim-v-meste

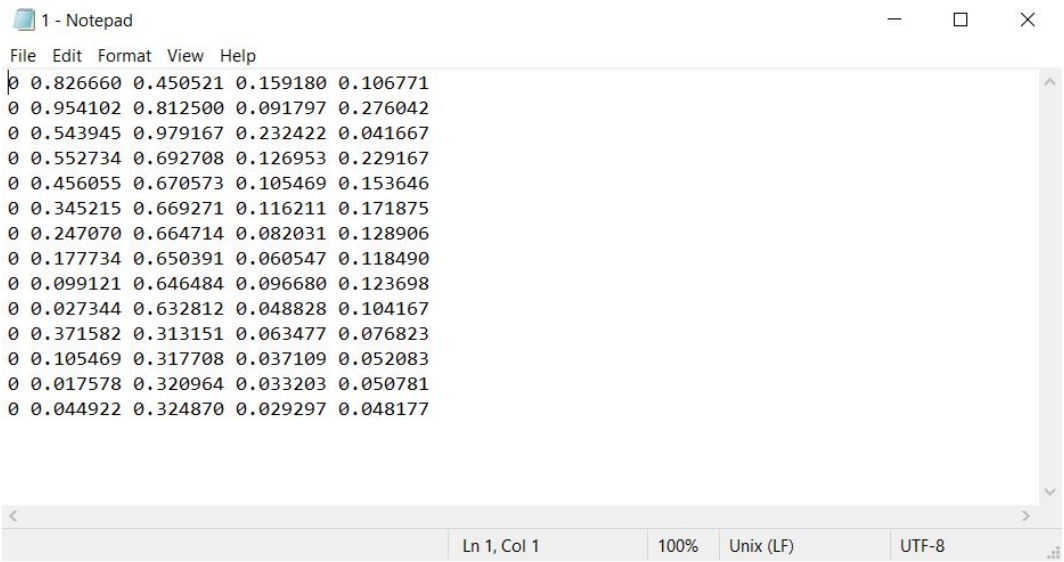
Slika 4.5 Primjer validacijskog dataseta

## 4.2 Anotacija podataka

Za anotaciju podataka korišten je software LabelImg. LabelImg software je za grafičku anotaciju podataka. Napisan je u python programskom jeziku i sprema anotacije kao XML podatke. LabelImg također dopušta spremanje anotacija u tip podatka koji je pogodan za treniranje YOLO modela [25]. Primjer anotacije podataka korištenjem LabelImg software dan je na Slici 4.6.

Slika 4.6 Primjer grafičke anotacije - LabelImg

YOLO anotacija se sprema putem formata danog na Slici 4.7. Prvi broj u nizu označava klasu objekta. U ovom slučaju postoji samo jedna klasa, automobil. Drugi broj u nizu označava X koordinatu središta *bounding box*-a, a sljedeći broj označava Y koordinatu označenog *bounding box*-a. Ostala dva broja, redom, označavaju širinu i visinu *bounding box*-a koji označava objekt. Slika 4.7 sadrži tekstualne podatke anotacija sa Slike 4.6. Za svaku od slika u datasetu potrebno je napraviti anotacije. Kako je prikazano na Slici 4.7, svaka klasa i koordinata *bounding box*-a nalazi se u novom retku u datoteci "1.txt". Nazivi datoteka moraju odgovarati jedna drugoj, npr. ako se slika zove *slika1.jpg* tada se tekstualna datoteka s anotacijama mora zvati *slika1.txt*. Nakon što su svi podaci anotirani na prikazani način i nakon što imena "txt" i "jpeg" datoteke odgovaraju jedan drugome, tada su podaci spremni za treniranje modela.




Slika 4.7 Primjer izgleda tekstualne anotacije za treniranje YOLOv4 modela

## 4.3 Treniranje modela

Za treniranje YOLOv4 modela korištena je *google colab* bilježnica. Treniranje modela pomoću *google colab* bilježnice omogućuje korištenje google GPU-a što znatno smanjuje vrijeme treniranja modela. Kod za treniranje YOLOv4 modela dan je u prilogu A.1. i baziran je na kodu danom u članku "TRAIN A CUSTOM YOLOv4 OBJECT DETECTOR (Using Google Colab)" danom od strane autora Techzizou [26]. Bitno je naglasiti da je cijeli kod napisan u programskom jeziku Python.

Kod će u ovoj sekciji biti detaljno objašnjen.

Kao što je objašnjeno u članku [26], za početak je potrebno kreirati praznu datoteku pod nazivom "yolov4". U datoteku "yolov4" potrebno je napraviti novu datoteku pod nazivom "training" u koju će se spremati trenirani model, to jest njegovi težinski faktori (*weights*). Iza toga, potrebno je napraviti *mount Google Drive*-a kako bi mogao biti korišten u *Google Colab* skripti. Za kraj se stvara simbolički link na korišteni *Google Drive* i skripta se pozicionira unutar datoteke "yolov4".

Sljedeći je korak kloniranje repozitorija "darknet" s githuba. To je github repozitorij koji sadži sve potrebne skripte za treniranje YOLOv4 modela. Nakon završetka

ova dva koraka potrebno je složiti set podataka tako da se svi podaci (.txt i .jpeg) postave u jednu datoteku pod nazivom "obj". Zatim je tu datoteku potrebno sažeti u datoteku pod nazivom "obj.zip". Istu ".zip" datoteku potrebno je prenjeti na *Google Drive* u folder "yolov4".

Sljedeći korak je priprema "yolov4-custom.cfg" konfiguracijske datoteke kako bi ona odgovarala podacima korištenim za treniranje. Unutar konfiguracijske datoteke potrebno je promijeniti vrijednosti varijabli "batch", subdivisions, width i height kako je dano u tablici 4.1. Uz to potrebno je i izračunat vrijednost varijable "max_batches" i "steps". Vrijednost varijable "max_batches" računa se pomoću jednadžbe (4.1) ispod

$$max\_batches = classes \cdot 2000. \tag{4.1}$$

, dok se za vrijednost "steps" uzima vrijednost 80% i 90% od dobivene vrijednosti "max_batches" kao što je prikazano jednadžbom (4.2) ispod

$$steps = max\_batches \cdot (80\%/90\%). \tag{4.2}$$

Nadalje, potrebno je promijeniti vrijednosti svih varijabli "filters" u ispod svakog od [convolutional] naslova unutar konfiguracijske datoteke koji dolazi prije [yolo] naslova. To je potrebno napraviti tako da vrijednost "filters" odgovara jednadžbi (4.3) prikazanoj ispod.

$$filters = (classes + 5) \cdot 3. \tag{4.3}$$

Za kraj je potrebno vrijednost linije classes ispod svakog [yolo] naslova postaviti u broj klasa koji se nalaze u setu podataka za treniranje. Ovdje je taj broj 1. Svi izračunati podaci promijenjeni u ".cfg" datoteci nalaze se u već navedenoj Tablici 4.1.

Tablica 4.1 Prilagodna vrijednosti za trening - yolov4-custom.cfg

| varijabla | promijenjena vrijednost |
|---|---|
| batch | 64 |
| subdivisions | 16 |
| width | 416 |
| height | 416 |
| max_batches | 2000 |
| steps | 1600, 1800 |
| filters | 18 |
| classest | 1 |

Nakon modifikacije konfiguracijske datoteke potrebno je napraviti datoteku "obj.data" koja mora sadržavati broj klasa i puteve do svih datoteka potrebnih za treniranje modela. U ovom slučaju datoteka mora sadržavati vrijednosti kao što je dano u Tablici 4.2.

Tablica 4.2 Podaci u datotecu obj.data

| ime varijable | vrijednost |
|---|---|
| classes | 1 |
| train | data/train.txt |
| valid | data/test.txt |
| names | data/obj.names |
| backup | /mydrive/yolov4/training |

Datoteka iz tablice "obj.names" sadrži samo vrijednost imena klasa na temelju kojih će se vršiti treniranje. U ovom slučaju sadrži samo naziv "car" kako se model trenira samo za jedan objekt.

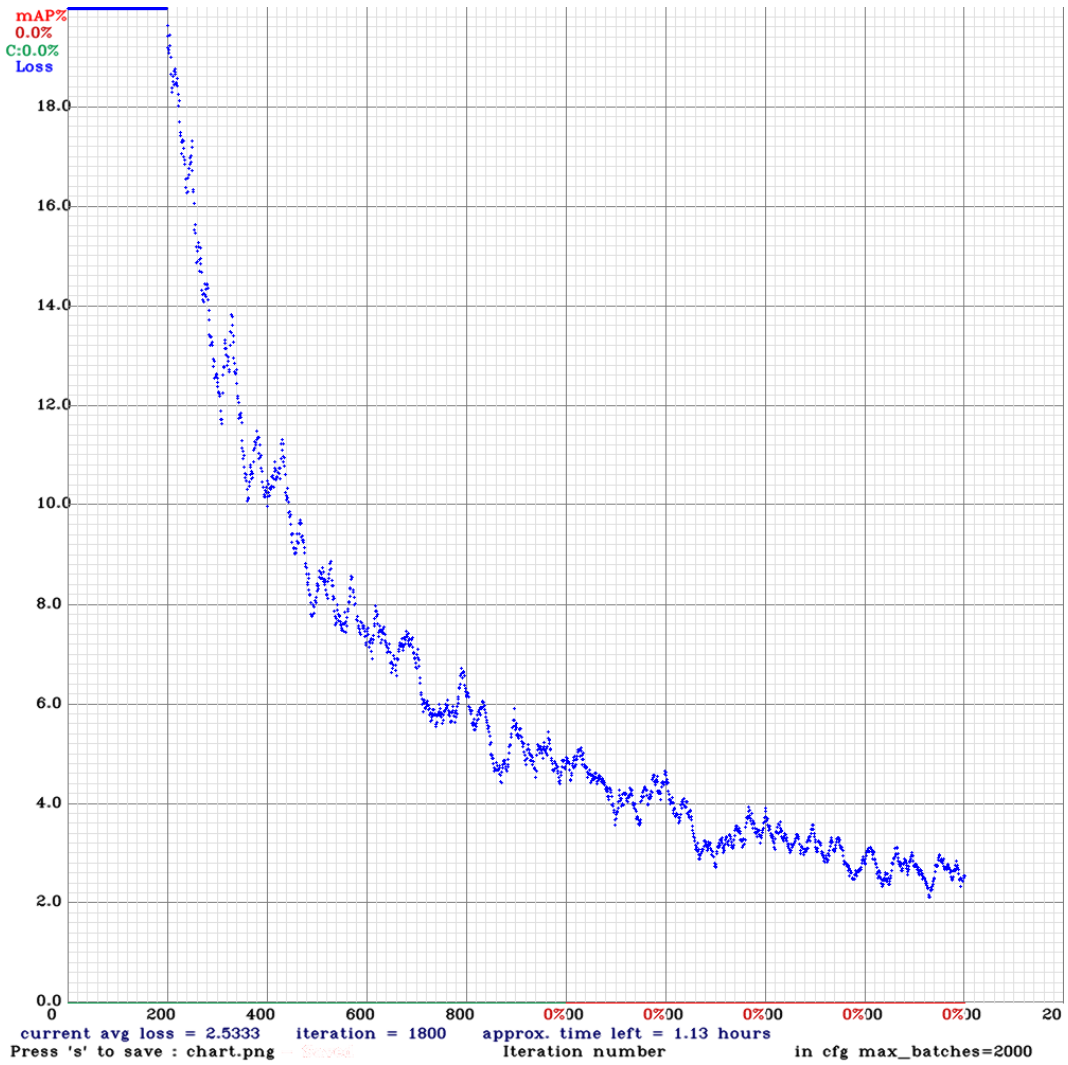
Za podjelu podataka prije samog treniranja modela potrebno je dodatno modificirati python skriptu "process.py" koja se može pronaći na githubu od autora Techzizou gdje je link na github repozitorij dan u članku [26]. Ta skripta služi za

raspodjelu podataka u test i train datoteke. Također, ista skripta će generirati tekstualne datoteke "test.txt" i "train.txt" koje su dane u tablici 4.2. Skripta standardno podijeli podatke dane u "obj.zip" na 90% trening podataka i 10% testing podataka. U slučaju ovog rada, skripta je modificirana tako da sve podatke prenesene na *Google Drive* prebaci u trening datoteku. Testiranje i validacija bit će izvršene naknadno nakon treniranja modela. Modificirana skripta dana je kao prilog A.2.

Prije treniranja potrebno je omogućiti korištenje OpenCV-a i GPU-a unutar darkneta, kao što je dano u prilogu A.1 i napraviti build na darknetu pomoću naredbe "!make". Potrebno je generirati pojedine datoteke i očistiti ostale kako je dano u prilogu A.1. Zatim se pokreće "process.py" skripta koja raspodjeljuje podatke, preuzimaju se predtrenirani težinski faktori za yolov4 model i za počinje se trening naredbom "!./darknet detector train data/obj.data cfg/yolov4-custom.cfg yolov4.conv.137 -dont_show -map". Nakon toga model se ostavlja na treniranju koliko je potrebno za optimalnu detekciju. Svakih 1000 iteracija spremaju se težinski faktori za model kao *backup* i nakon prestanka treniranja dobije se datoteka "yolov4-custom_last.weights" koja sadrži zadnje iteracije treniranja, to jest to je najbolja verzija težinskih faktora i ona se koristi u daljnjoj detekciji i razvoju modela.

Model u ovom radu treniran je na 1800 iteracija. Krivulja gubitaka prilikom treniranja ovog modela dana je na Slici 4.8.

Uglavnom se čeka da krivulja gubitaka padne na što nižu vrijednost. Prilikom treniranja ovog modela odlučeno je stati s treniranjem na 1800 iteracija i prvi isprobati model prije daljeg treniranja. Uspostavilo se da model na 1800 iteracija daje dovoljno dobre rezultate na nekoliko testnih slika, pa je iz toga zaključeno da se može ići na daljnje korake testiranja i validacije s testnim i validacijskim setom podataka.




Slika 4.8 Loss function prilikom treniranja

## 4.4 Testiranje i validacija modela

Za testiranje i validaciju treniranog YOLOv4 modela korišteni su setovi slika za testiranje i validaciju. Set slika za testiranje dan je ranije na Slici 4.3, a set za validaciju dan je na Slici 4.4. Za testiranje i validaciju korišten je kod dan u prilogu A.3. Navedeni kod modificirana je verzija koda koji je dan u članku "YOLO object detection using Opencv with Python" [27]. Isto kod u prilogu A.3 koristi se i za detekciju popunjenosti parkirnih mjesta, ali o tome će biti govora nešto kasnije.

Za početak se provjera kvalitete detekcije modela vršila na testnom setu podataka. Dobiveni primjeri detekcije dani su na Slici 4.9. Temeljem dobivenih podataka može se zaključiti da trenirani model ima izrazito zadovoljavajuću detekciju objekata. Većina automobila je detektirana sa sigurnošću detekcije od 90% ili više. Udaljene automobile se može zanemariti zato što i sama kvaliteta kamere može utjecati na kvalitetu detekcije. Isto tako, nije bitno kolika je točnost detekcije automobila koji se nalaze van parkirališta koje promatra kamera, jer je kasnije za detekciju bitno što algoritam detektira bliže, unutar parkirališta. Zaključak je s toga da se prijeđe nas sljedeći korak validacije modela na setu slika koje algoritam prilikom učenja još nije vidio.

Dakle, druga provjera valjanosti modela je na validacijskom setu podataka. Primjeri detekcije na validacijskom setu podataka dani su na Slici 4.10. Jasno se vidi da i na validacijskom setu podataka dolazi do zadovoljavajuće detekcije automobila. Jasno je vidljivo kako detekcija dobro funkcionira i po noći ako je parkiralište dovoljno dobro osvijetljeno kao što je vidljivo na Slici 4.10 a). Isto tako, detekcija je bila valjana i na slici koja predstavlja *CGI* model parkirališta, kao što se vidi na Slici 4.10 b). Na svim se slikama generalno primjećuje kako je detekcija ponovno iznad 90% što govori da će model biti u stanju zadovoljavajuće detektirati automobile na različitim nepoznatim parkirnim kamerama.

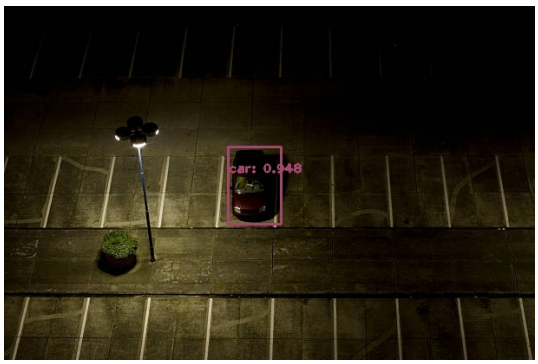
Prema dobivenim rezultatima iz oba seta podataka može se zaključiti da je dobiveni model pogodan za realizaciju algoritma detekcije popunjenosti parkirnih mjesta.


(a) (b)

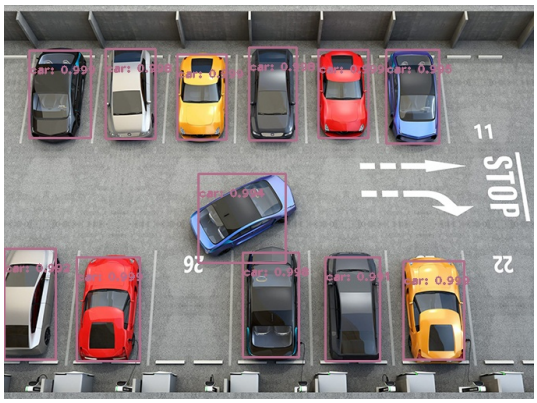
(c) (d)

(e) (f)

Slika 4.9 Detekcija YOLOv4 modela na testnom setu podataka

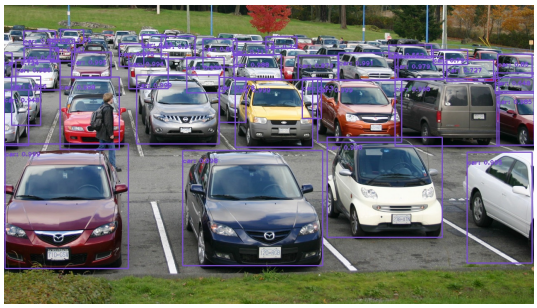



(a) https://www.nomadicnews.com/is-walmart-camping-safe-2022/




(b) https://www.nextech.sk/a/Smartcity–Inteligentne-riesenie-odstrani-problem-s-parkovanim-v-meste

(c) https://www.novilist.hr/rijeka-regija/rijeka/rijeka-plus-otkrio-koliko-ce-naplacivati-parking-u-ulici-franje-rackog-stanari-su-to-trazili/



(d) https://camosun.ca/community/facility-rentals/facilities-and-rates/parking-areas

Slika 4.10 Detekcija YOLOv4 modela na validacijskom setu podataka

# Poglavlje 5

# Algoritam za detekciju popunjenosti parkirnih mjesta

## 5.1 Ideja

Ideja za razvoj ovog algoritma javila se kao potreba unaprjeđenja postojećeg algoritma razvijenog od strane tvrtke devlab d.o.o. Sličan algoritam razbijen je ranije korištenjem programskog jezika Python i Python biblioteke *tensorflow* 1.0.

Sustav se sastojao od 3 skripte, gdje je svaka imala svoj zadatak. Prva skripta je dohvaćala slike s kamere, druga je rezala te slike na manje slike tako da je svaka izrezana slika prikazivala jedno parkirno mjesto i treća skripta je vršila detekciju temeljem *vacant* i *occupied* faktora treniranog modela koji je mogao prepoznati samo ta 2 slučaja. Uglavnom je takva ideja bila dobra, ali zbog nekih grešaka u detekciji, loše optimiziranosti zbog previše paralelnih skripti i limitiranosti dodavanja *feature*-a sustavu javila se potreba za novom idejom realizacije takvog sustava.

Zahtjevi za novi sustav su bili smanjenje broja skripti koje sustav pokreće u pozadini, kvalitetnija detekcija popunjenosti parkirnih mjesta i mogućnost dodavanja novih podsustava (*features*) koji mogu vršiti ostale zadatke, kao što su detekcija nepravilnog parkiranja, detekcija popunjenosti invalidskih mjesta, detekcija popunjenosti i tipa automobila na rezerviranim mjestima itd.

Nakon postavljenih zahtjeva za sustav odlučeno je pokušati razviti sustav de-

tekcije popunjenosti parkirnih mjesta pomoću modela detekcije objekata YOLOv4. Ranije kroz rad je naveden proces razmišljanja, to jest kako je došlo do korištenja baš ovog modela. Glavna ideja je bila detektirati objekt, u ovom slučaju automobil i odrediti njegovu središnju točku detekcije, što YOLOv4 model vrača kao izlazni podatak nakon detekcije. Zatim je ideja bila prepoznati nalazi li se ta točka unutar označenog prostora na slici, to jest unutar nekog označenog parkirnog mjesta.

Za treniranje YOLOv4 modela bio je potreban set podataka koji je prikupljen kako je ranije objašnjeno u radu. Uglavnom se pristup rješavanju problema može svesti na 6. točaka kao što slijedi:

1. Prikupiti dovoljno podataka za treniranje modela
2. Anotacija podataka
3. Treniranje modela
4. Testiranje i validacija modela
5. Izrada algoritma detekcije
6. Testiranje funkcionalnosti algoritma detekcije

Koraci pod stavkama od 1 do 4 objašnjeni su u ranijim poglavljima. Stavke 5 i 6 biti će objašnjene u sklopu ovog poglavlja.

Za kraj, ono što je vrlo bitno kod ovakvog pristupa rješavanja problema je to da ovakav sustav omogućuje ono što prethodni nije. Ovaj sustav omogućuje detekciju i procjenu popunjenosti parkirnih mjesta unutar jedne skripte (što je idealno za internetski servis/aplikaciju) i to što relativno jednostavno omogućuje proširenje sustava s različitim drugim funkcionalnostima, kao u ranije navedenim primjerima - detekcija nepravilnog parkiranja, detekcija popunjenosti invalidskih mjesta i ostalo.

## 5.2 Realizacija

Sustav, to jest skripta, za detektiranje popunjenosti parkirnih mjesta dana je u prilogu A.3. Isti kod se koristi i za prikaz prozora detektiranog objekta kao što je

prikazano ranije na slikama 4.9. i 4.10. Za početak bit će predstavljeni osnovni dijelovi na temelju kojih skripta radi, a zatim će dodatno biti pojašnjena ideja i način funkcije koji stoji iza programskog koda.

Skripta je napisana u programskom jeziku Python. Za realizaciju koda korištene su navedene biblioteke i navedeni su kako su pozvani u kodu:

1. OpenCV - import cv2
2. Numpy - import numpy as np
3. matplotlib - import matplotlib.pyplot as plt
4. Shapely - from shapely.geometry import Point, from shapely geometry.polygon import Polygon

OpenCV je Python *open source computer vision* biblioteka koja omogućuje implementaciju YOLOv4 modela. OpenCV u kodu omogućuje učitavanje YOLOv4 modela i njegovo korištenje za detekciju objekata. Numpy i matplotlib su standardne matematičke Python biblioteke koje se koriste za različite matematičke kalkulacije i manipulacije te grafički prikaz rezultata računanja. Shapely je biblioteka koja služi za kompleksnu manipulaciju geometrijskim objektima. Shapely pojednostavljuje strukturiranje i konstrikciju različitih geometrijskih objekata i omogućuje njihovu interakciju. Ideja iza korištenja Shapely biblioteke je kontrola da li se središnja točka detektiranog objekta nalazi unutar označenog prozora koji predstavlja dimenzija parkirnog mjesta.

Uz navedene biblioteke, kako bi se smanjila količina koda unutar glavne skripte, dodana je i skripta pod nazivom "Crtanje.py" i ona je pozvana u kodu kao:

- Crtanje.py - import Crtanje

Skripta "Crtanje.py" u suštini radi posao vizualizacije dobivenih rješenja. Time se može vršiti kontrola kvalitete rješenja i dodatno uštimavati kod za različite *test case*-ove.

Sljedeća stavka je objašnjenje programskog koda. Na početku učitane su navede biblioteke s jednakom sintaksom kao što je dano ranije. Također učitana je skripta

za prikaz rezultata "Crtanje.py". Iza toga, unutar "main.py" skripte, u varijablu net učitava se YOLOv4 trenirani model pomoću funkcije *cv2.dnn.readNet(model_weights, model_configuratio_file)*. Potrebno je definirati klase, to jest koji se objekti detektiraju, imena layera koji se nalaze u varijabli *net* i izlazni layeri unutar iste varijable.

Nakon učitanog modela potrebno je učitati i sliku na kojoj se vrši detekcija. Za to se koristi naredba *cv2.imread(ime_slike.tip_slike)*. Učitanoj se slici smanjuju dimenzije na 90% originalne radi bolje preciznosti detekcije (iz iskustva za vrijeme rada na sustavu) i zatim se iz te slike uzimaju parametri visine, širine i kanala(height, width, channels). Kanali predstavljaju boje u slici (RGB).

Sljedeći dio koda služi za detekciju objekata. Dio koji vrši detekciju objekata nalazi se ispod komentara "Detecting objects" kako je dano u prilogu A.3. Izlazne vrijednosti nalaze se unutar varijable *outs*. Varijabla *outs* ponaša se kao polje koje sadrži polja (*array of arrays*) i ta polja zapravo označavaju gdje se na slici nalazi detektirani objekt, to jest njegovu koordinatu.

Kada je završena detekcija definiraju se pomoćne varijable u Python polja. Ta polja spremaju bitne informacije za crtanje okvira oko detektiranih objekata na slici. Također, vezano za detekciju popunjenosti jako su bitne varijable *CoordinateX* i *CoordinateY*. Te će se varijable koristiti za spremanje centralne X i Y koordinate točke detekcije sa slike. To je bitno jer će upravo te varijable biti korištene za provjeru je li se točka nalazi unutar zadanog okvira koji označava parkirno mjesto.

Zatim u kodu slijede dvije *for* petlje koje iteriraju po varijabli *outs* i zatim po polju koje se nalazi u varijabli *outs*. Unutar varijable *scores* spremaju se vrijednosti vjerojatnosti postojanja objekta, to jest vrijednost od 0 do 1 u koliko je algoritam siguran da je objekt detektiran. Provjerava se je li detektirana određena klasa, u ovom slučaju klasa 0 jer samo ona i postoji te se zatim gleda je li vjerojatnost za pojedini detektirani objekt veća od 50%. Ako jest on se uzima u obzir za crtanje i za procjenu popunjenosti parkirnih mjesta. U varijable *center_x* i *center_y* spremaju se koordinate centra detektiranog objekta i te se varijable nadodaju na ranije definirana polja *CoordinateX* i *CoordinateY*. Time se stvaraju 2 polja koja sadrže sve koordinate svih objekata nakon što završi izvršavanje *for* petlji. Također, radi crtanja okvira oko detektiranog objekta, na varijable *w, h, x, y* i *boxes, confidences, class_ids* dodaju se vrijednost redom, širine prozora, visine prozora, centralne koordinate prozora x

i y te polje koordinata za dimenzije prozora, vjerojatnosti detekcija i klase koje se pojavljuju u detekciji.

Izlaskom iz *for* petlje dodatno se filtrira detekcija tako da se uklanja šum. Naime, YOLOv4 će detektirati više točaka oko navedenog objekta, pa se naredbom *cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)* umanjuje broj tih točaka. Kada se ukloni šum crtaju se okviri oko objekata na ranije učitanoj slici. To se radi pomoću dvije *for* petlje od kojih se jedna vrti onaj broj puta koliko ima okvira za crtanje, a zatim se vrti po svim okvirima unutar polja varijable *boxes* i crta okvire te im kasnije dodaje tekst. To rade naredbe *cv2.rectangle()* i *cv2.putText()*.

Nakon crtanja okvira na originalnu sliku poziva se već ranije spomenuta skripta "Crtanje.py". Unutar nje nalazi se funkcija "crtanje_točaka" koja prima širinu slike, visinu slike i centralne koordinate X i Y svih detektiranih točaka. Skripta "Crtanje.py" dana je u prilogu A.4.

Funkcija "crtanje_točaka" unutar navedene skripte odrađuje posao cijele vizualizacije detekcije popunjenosti parkirnih mjesta. Na početku učitava jedan od polja JavaScript (.json file u web aplikaciji dobiven s *request*-om) objekata koji su u kodu spremljeni pod varijablu *ParkinSpaceCoordinates*. Postoji ih veći broj u kodu radi određenog broja online kamera koje su korištene za testiranje algoritma, uglavnom učitava se samo jedan od njih. Iza toga se definira prazna lista *ParkingNames* i lista *PopunjenostMjesta* koja je popunjena s vrijednostima false za onoliko vrijednosti kolika je dužina polja *ParkinSpaceCoordinates*. Na sličan se način definira i prazna lista *POLI* koja služi za spremanje pojedinih poligona koji će biti nacrtani pomoću koordinata iz *ParkinSpaceCoordinates* varijable. Prije daljnjeg procesa detekcije definira se *matplotlib* prozor u kojem će biti prikazani rezultati.

Sljedeći korak u skripti je definiranje *for* petlje koja se izvršava onoliko puta koliko ima parkirnih mjesta na nekoj slici, kako je definirano varijablom *ParkinSpaceCoordinates*. Iz iste liste (polja) uzimaju se pojedinačne točke i sprema ih se u varijable *Dot1a, Dot2a, Dot3a* i *Dot4a*. Kako je ranije slika skalirani radi točnosti i točke se moraju skalirati na 90% originalne vrijednosti. Sve se točke nakon skaliranja spremaju u varijable *Dot1, Dot2, Dot3* i *Dot4*. Uz tako spremljene točke moguće je definirati objekt poligona pomoću *Shapely* biblioteke. Poligon se definira kako je prikazano u prilogu A.4. pomoću pozivanja funkcije *Polygon()* koji prima *array* točaka.

On se sprema u listu *POLI*. Poligoni se zatim crtaju pomoću *matplotlib* biblioteke i crtaju se vrhovi tih poligona.

Sljedeća *for* petlja unutar koda ponovno se izvršava onoliko puta koliko ima označenih parkirnih mjesta. Ideja ove petlje je crtanje točaka koje su pronađene samo unutar poligona. Na početku se na kraj liste *ParkingNames* dodaju imena svih parkirnih mjesta iz *ParkinSpaceCoordinates* liste. Iza toga se pokreće još jedna *for* petlja koja provjerava nalazi li se točka detekcije unutar poligona. Ukratko, definira se *Shapely* objekt *Point()* i zatim se provjerava sadrži li neki od poligona tu točku. Ako da, ažurira se stanje varijable *PopunjenostMjesta* i na to mjesto koje sadrži točku postavlja se istinita vrijednost.

Na kraju se radi preglednosti dodaju dvije točke na rubove *figure*-a koji će prikazati rezultate. Time se osigurava dobra rezolucija prikaza i bolji prikaz nacrtanih parkirnih mjesta. Funkcija vrača *main* skripti *PopunjenostMjesta* i *ParkingNames* liste.

Zadnji dio glavnog dijela koda sastoji se od formatiranja ispisa popunjenosti parkirnih mjesta. Prva *for* petlja prebrojava broj popunjenih parkirnih mjesta i određuje broj mjesta koji je slobodan. Sljedeći dio koda se koristi kako bi se na terminalu formatirano prikazala popunjenost parkirališta. Složeno je od još jedne *for* petlje koja prolazi po varijabli *Mjesta* i ako je vrijednost varijable na $i$-tom mjestu 'True' mjesto je popunjeno, ako ne onda je prazno. Prikaz izlaznih vrijednosti terminala bit će prikazan kasnije.

Na samom kraju glavnog dijela programa prikazuju se detektirana vozila pomoću YOLOv4 algoritma, kako je ranije definirano u kodu. Isto tako moguće je istu sliku spremiti u lokalnu memoriju ako se makne komentar sa zadnje linije koda.

Kako je objašnjenje u tekstualnom obliku relativno dugačko i manje razumljivo kod će još biti objašnjen temeljem liste, to jest kojim redoslijedom se što događa. Ukratko se to može zapisati ovako:

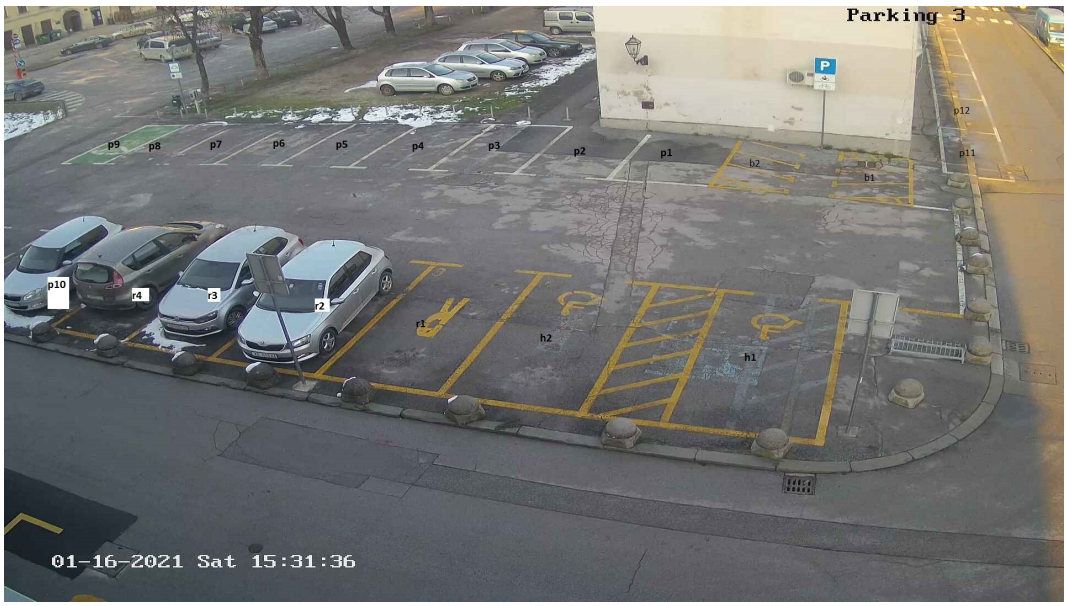
1. Učitavaju se potrebne biblioteke i skripte
2. Učitava se YOLOv4 model
3. Iz YOLOv4 modela uzimaju se potrebne vrijednosti

4. Učitava se slika, skalira se i iz nje se uzimaju potrebne vrijednosti
5. Vrši se detekcija objekata
6. Definiraju se pomoćne varijable za korištenje u algoritmu
7. Uzimaju se sve točke detektiranih automobila i zasebno se spremaju u listu
8. Uklanja se šum iz podataka
9. Crtaju se prozori oko detektiranih automobila (YOLOv4 prikaz detekcije)
10. Skripta *Crtanje.py* - određuje popunjenost mjesta i grafički prikazuje rezultate
11. Formatirani ispis popunjenosti parkirališta

## 5.3 Rezultati

Za ovaj skup podataka gradske kamere definiran je veći broj parkirni mjesta nego za ostale kamere koje su bile dostupne pa je njihove rezultate odlučeno prikazati detaljnije. Definirana su imena parkirnih mjesta kako je prikazano na Slikama 5.1, 5.2 i 5.3. Oznaka mjesta sa "R" označava rezervirano parkirno mjesto, oznaka mjesta sa "P" označava obično parkirno mjesto, a oznaka "H" označava parkirno mjesto za invalide.

Oznake su bitne za kasnije razlikovanje koje je parkirno mjesto popunjeno. To se ispisuje u konzoli i bit će prikazano u ovom poglavlju. Isto tako, bitno je napomenuti kako se ovdje ne radi o pravim parkirnim kamerama pa je dogovoreno da se za detekciju promatraju samo ona mjesta koja se jasno vide na kameri. Ona koja se nalaze daleko i ne osiguravaju praktičnu razinu detekcije se ne uzimaju u obzir prilikom detekcije popunjenosti.



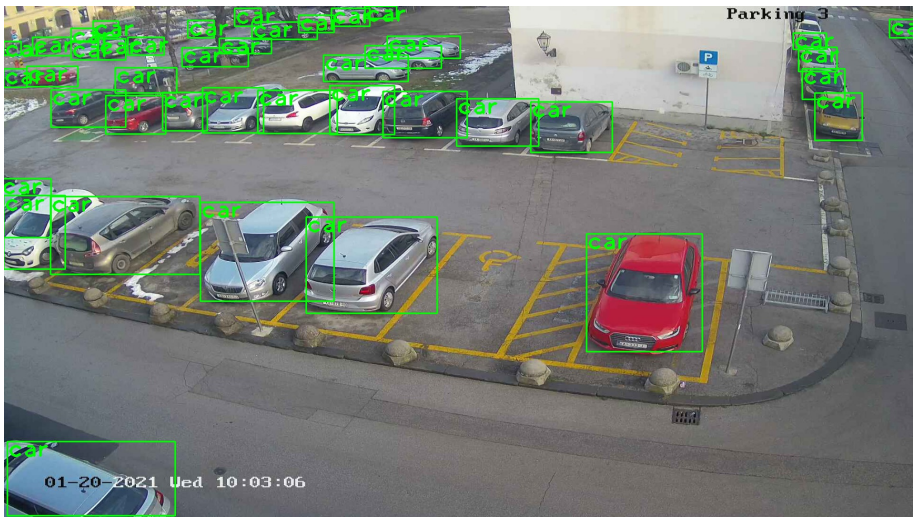

Slika 5.1 Označeni parking - frontalno


Slika 5.2 Označeni parking - ulaz lijevo

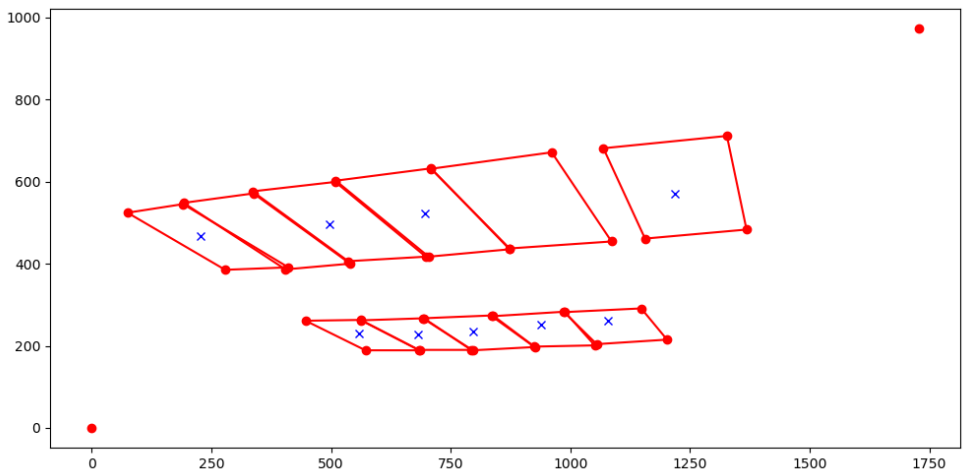

Slika 5.3 Označeni parking - ulaz desno

Za kameru sa slike 5.1 dobiveni su rezultati detekcije popunjenosti kao što su prikazani na Slici 5.4. Jasno se vidi da su svi automobili detektirani, čak i oni koji se nalaze izvan parkirališta. Isto tako, vidljivo je da je zadovoljena točnost detekcije, svi automobili su detektirani unutar svojih parkirnih mjesta. Prikaz konzole za ovaj slučaj detekcije dan je na Slici 5.5.

Vezano za grafički prikaz detekcije, bitno je dodati kako je kod crtanja grafika invertirana, to se događa zbog *matplotlib*-a.




(a)



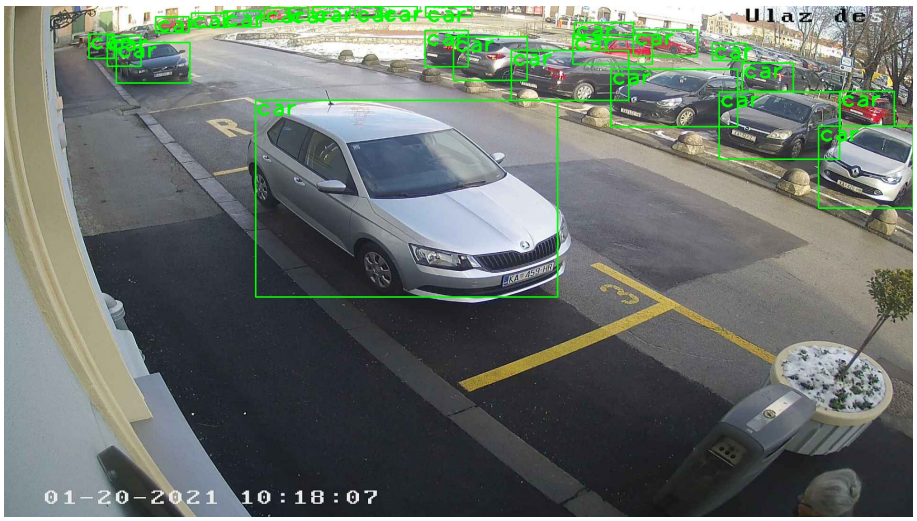

(b)

Slika 5.4 Detekcija - kamera frontalno

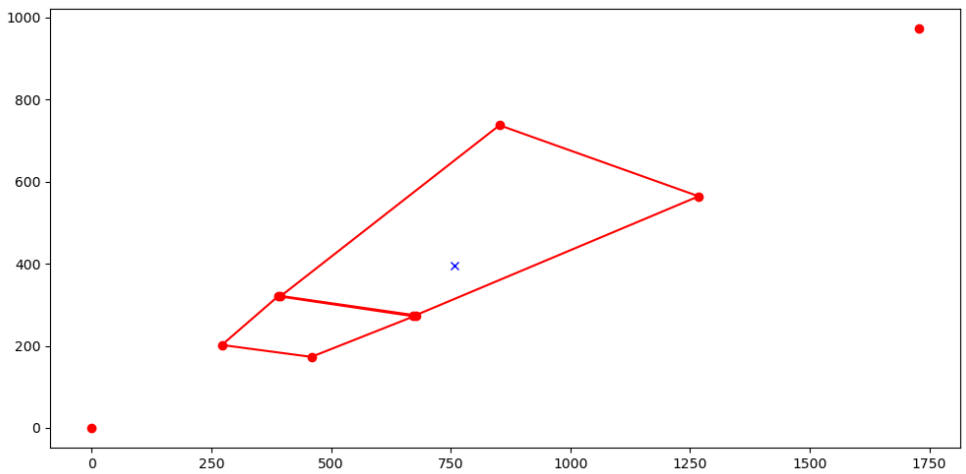
```
-------
['h1', 'h2', 'r1', 'r2', 'r3', 'r4', 'p1', 'p2', 'p3', 'p4', 'p5']
[True, False, True, True, False, True, True, True, True, True, True]
------------------------------------
h1---- Popunjeno
h2---- Prazno
r1---- Popunjeno
r2---- Popunjeno
r3---- Prazno
r4---- Popunjeno
p1---- Popunjeno
p2---- Popunjeno
p3---- Popunjeno
p4---- Popunjeno
p5---- Popunjeno
Mjesta slobodno: 2
```

Slika 5.5 Terminal - frontalno

Rezultati grafičkih prikaza ostalih kamera s gradske uprave dani su ispod na Slikama 5.6 i 5.7, a izgleda terminala dani su na Slikama 5.8. i 5.9.

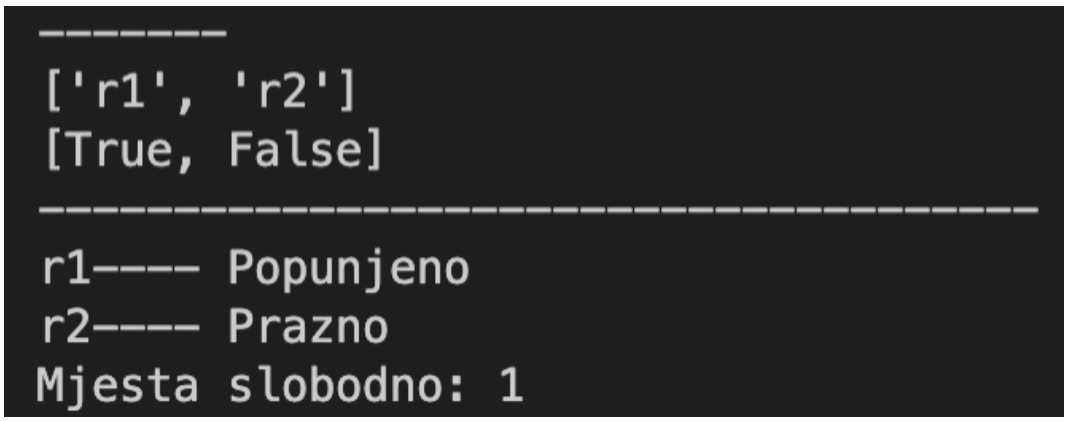



(a)



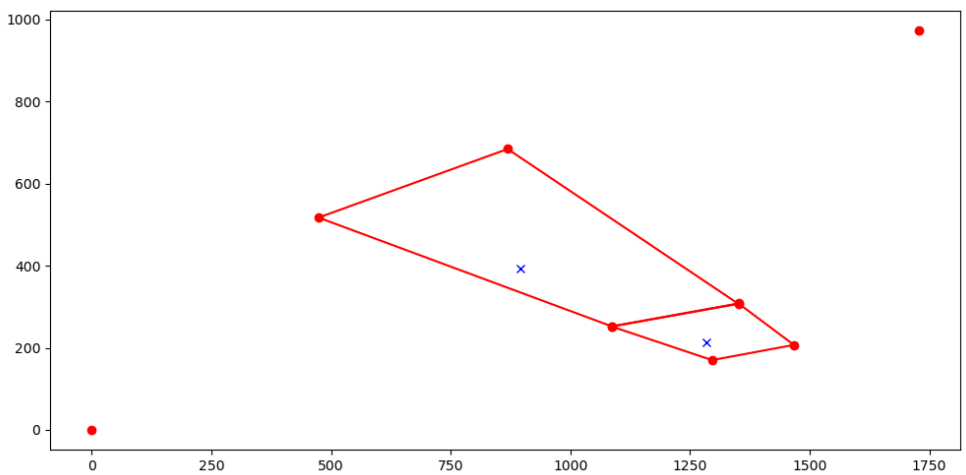

(b)

Slika 5.6 Detekcija - kamera desno



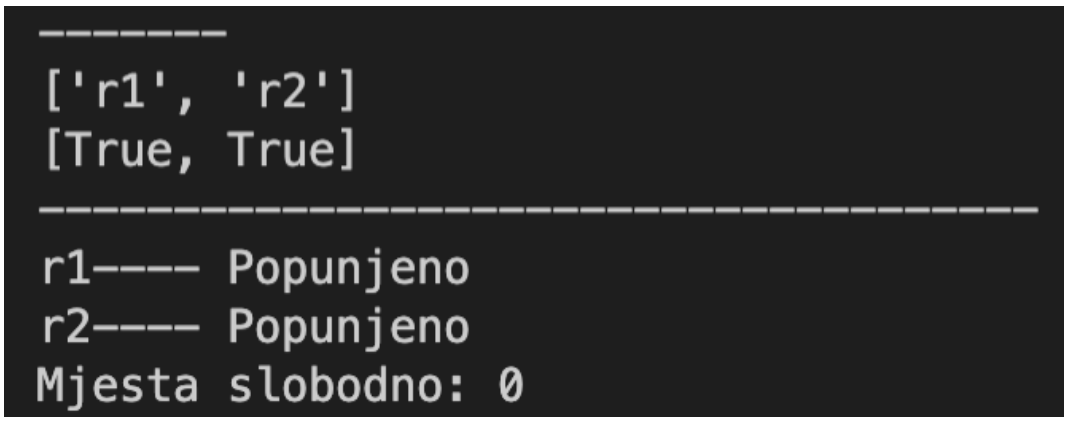

Slika 5.7 Terminal - desno


(a)



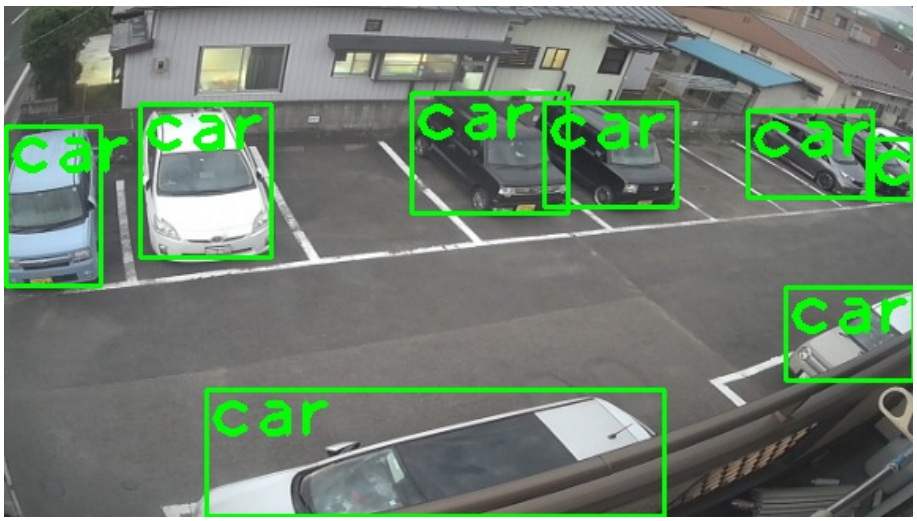

(b)

Slika 5.8 Detekcija - kamera lijevo

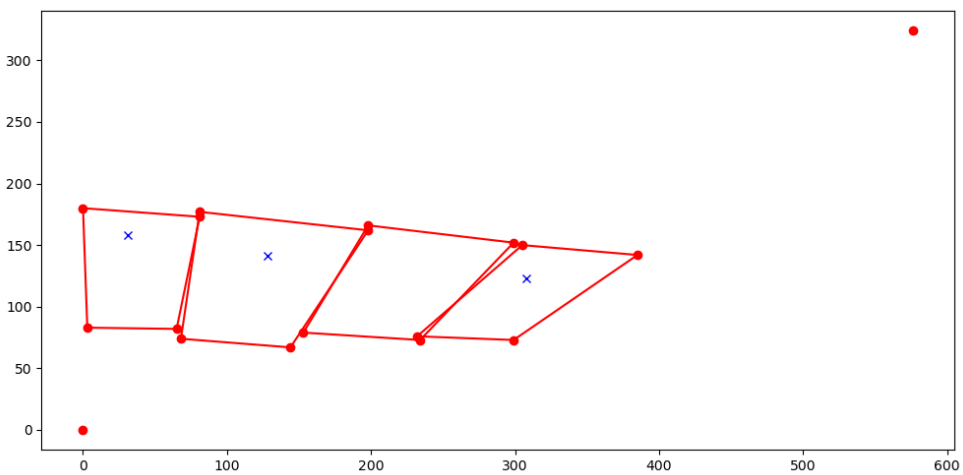



Slika 5.9 Terminal - lijevo

Neki primjeri ostalih rezultata dani su na Slikama 5.10, 5.11, 5.12, 5.13, 5.14, 5.15 i 5.16.

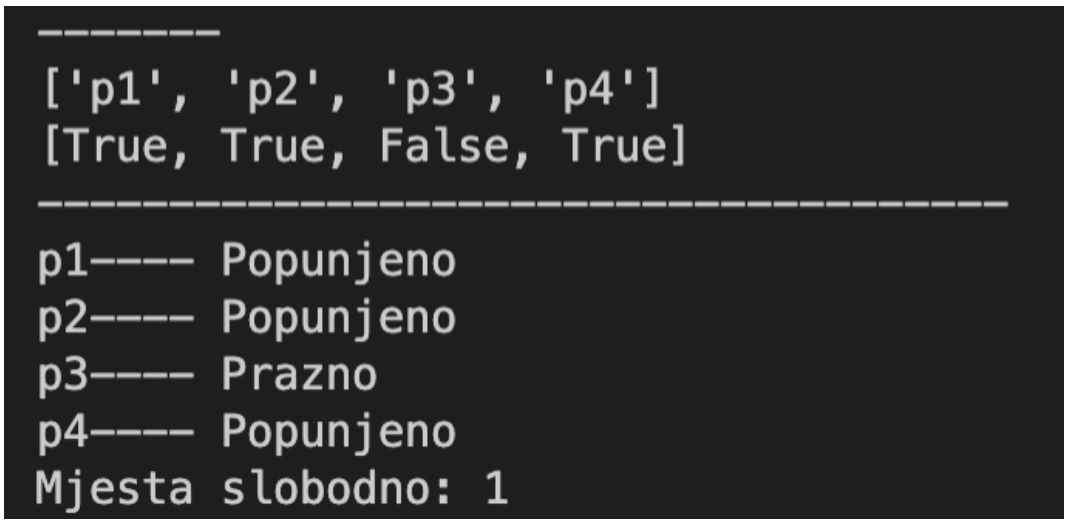



(a)

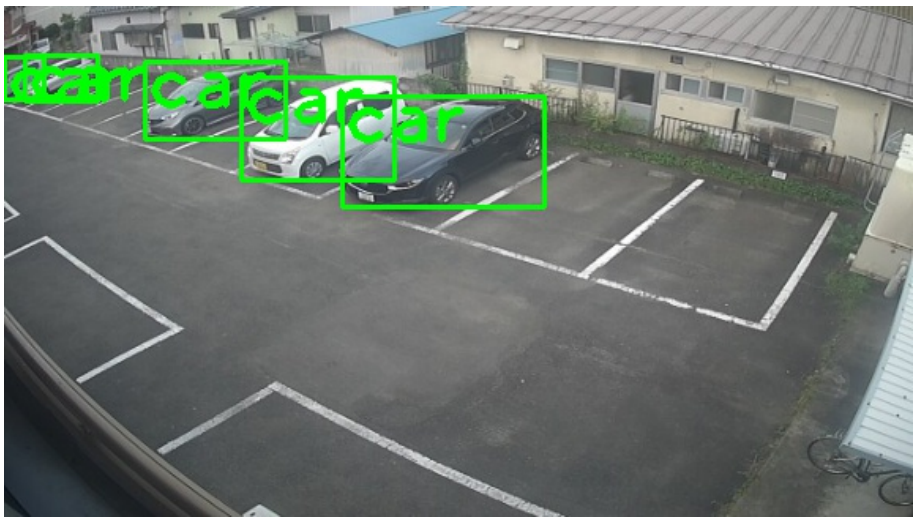



(b)

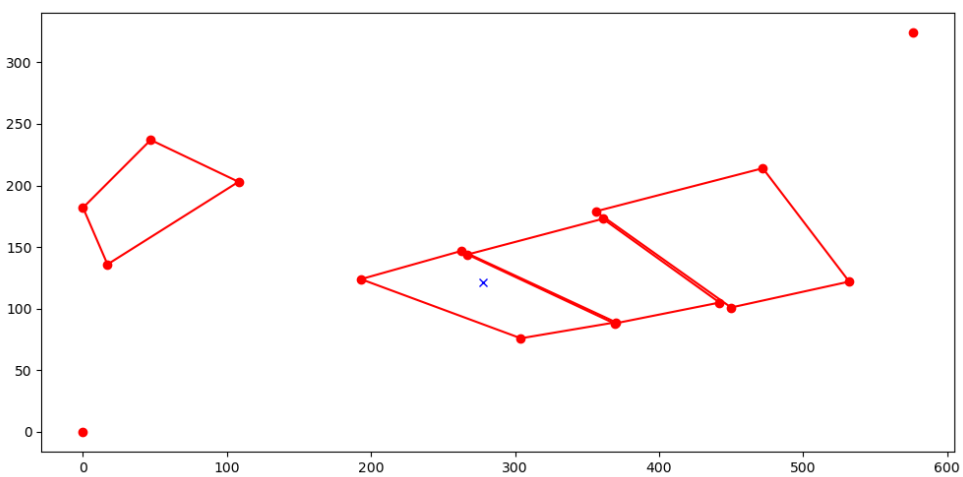
Slika 5.10 Detekcija - kamera Japan s kanala 1




Slika 5.11 Terminal - kamera japan s kanala 1

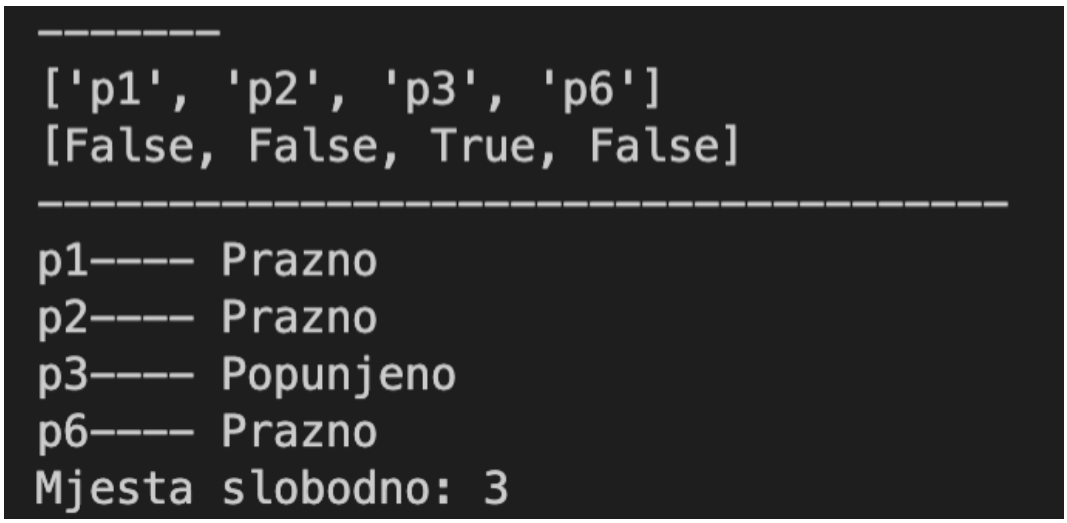



(a)



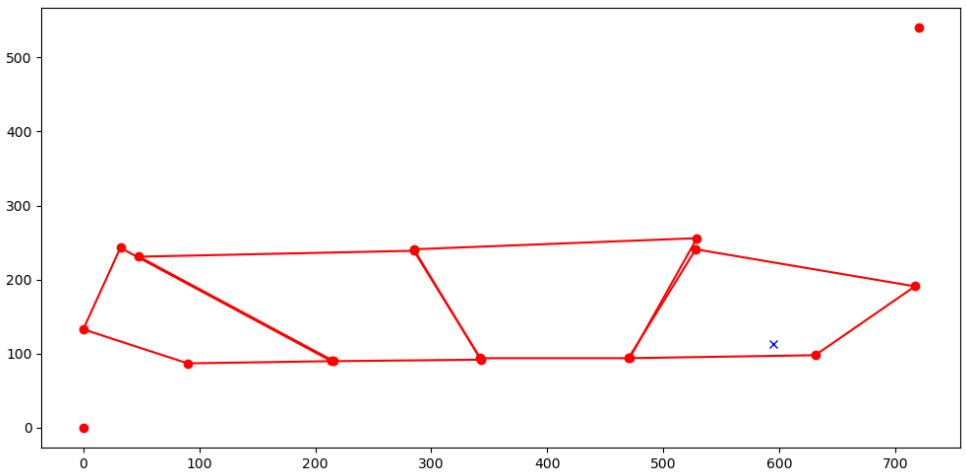

(b)

Slika 5.12 Detekcija - kamera Japan s kanala 2



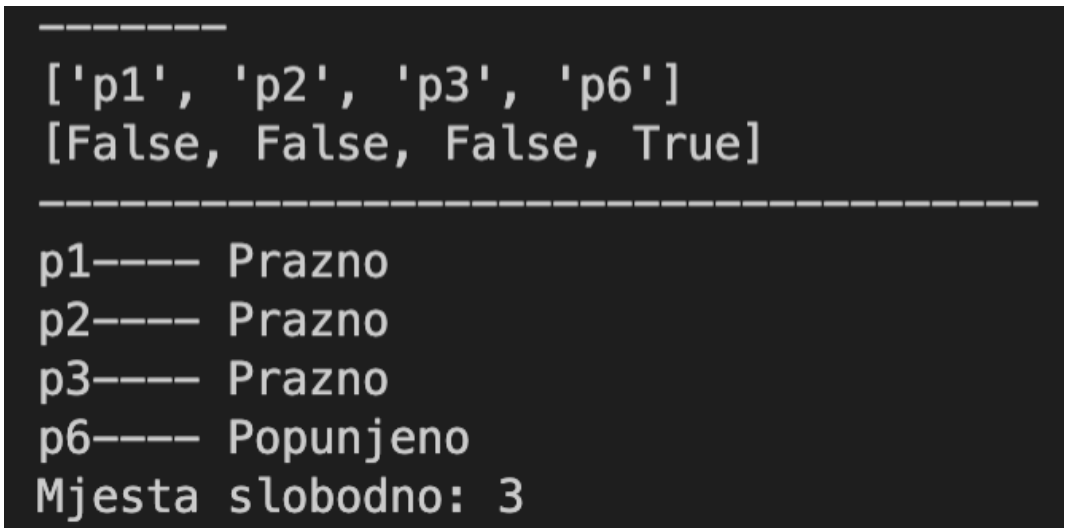

Slika 5.13 Terminal - kamera japan s kanala 1


(a)




(b)

Slika 5.14 Detekcija - kamera Taiwan




Slika 5.15 Terminal - kamera Taiwan

# Poglavlje 6

# Zaključak

U ovom radu opisan je postupak izrade algoritma za detekciju popunjenosti parkirnih mjesta korištenjem modela umjetne inteligencije za detekciju objekata. Na početku dan je uvod u umjetnu inteligenciju i pregled neuronskih mreža. Zatim su detaljnije objašnjene konvolucijske neuronske mreže (CNN) koje se koriste za detekciju objekata. Iza toga dani su i uspoređeni neki modeli detekcije objekata. Nakon usporedbe odabran je najpogodniji, YOLOv4. Iz usporedbe modela detekcije objekata može se zaključiti da moderni modeli ne odstupaju znatno jedan od drugog i da nijanse odlučuju o odabiru modela. U ovom slučaju prevladala je dobra karakteristika brzine YOLOv4 modela.

Nakon odabira modela prikazan je proces treniranja YOLOv4 modela. Na početku je bilo potrebno prikupiti podatke. Set podataka prikupljen je pomoću privatnih i internetski dostupnih nadzornih kamera koje pokazuju na parkirališta. Jedan je od zaključaka da je prikupljanje ovakvog seta podataka na većem broju parkirališta prilično zahtjevno radi male dostupnosti javnih kamera na istim. Posebno je teško pronaći kamere sa zadovoljavajućim kutem gledišta. Taj je set podataka zatim bilo potrebno i anotirati, to jest označiti gdje se nalaze automobili. Prilikom označavanja odlučeno je da se označava samo vidljivi dio automobila, a ne njegove cijele dimenzije. Ovaj se pristup kasnije pokazao boljim jer se time centralna točka objekta prilikom detekcije pomaknula prema gledištu kamere i osigurala točniju procjenu popunjenosti parkirnih mjesta. Nakon anotacija modificirana je konfiguracijska datoteka i model je treniran. Takav model bilo je potrebno testirati. Model je testiran na testnom

setu i validacijskom setu. Testni set se sastojao od slika sličnih onima pomoću kojih je bio treniran dok se validacijski set sastojao od potpuno drugih slika koje model još prije nije vidio. Iz testiranja može se zaključiti da je trenirani model bio pogodan za daljnje korištenje s algoritmom za detekciju popunjenosti parkirnih mjesta.

Zadnji dio rada bavi se izradom algoritma detekcije parkirnih mjesta temeljem treniranog modela. Ideja je detaljno objašnjena u radu. Nakon što je dana ideja objašnjena je i realizacija algoritma. Za izradu algoritma korišten je programski jezik Python. Zaključak je da Python ima velik broj dostupnih *computer vision* biblioteka pa je zato odličan odabir za izradu ovakvog algoritma. Algoritam je testiran na setu podataka onih na kojima je model treniran. Funkcija algoritma i rješenja te vizualizacija rješenja dana je u radu. Temeljem dobivenih rezultata može se zaključiti da je algoritam zadovoljavajuć i da više nego dobro obavlja svoj zadatak.

# Bibliografija


[1] Daubaras A., Zilys M.: "Vehicle detection based on magneto-resistive magnetic field sensor", Elektronika Ir Elektrotechnika, Vol. 118, No. 2, pp. 27-32, 2012.

[2] Bao X., Zhan Y., Xu C.: "A novel dual microwave Doppler radar based vehicle detection sensor for parking lot occupancy detection", IEICE Electronics Express, Vol. 14, No. 1, pp. 20161087-20161087, 2017.

[3] Amato G., Carrara F., Falchi F., i dr.: "Car parking occupancy detection using smart camera networks and Deep Learning", IEEE Symposium on Computers and Communication (ISCC), pp. 1212-1217, 2016.

[4] Car Z.: "Umjetne neuronske mreže", materijali s predavanja, 2022.

[5] Neves, A. C., Gonzalez I., Leander J., i dr.:"A New Approach to Damage Detection in Bridges Using Machine Learning", International Conference on Experimental Vibration Analysis for Civil Engineering Structures, pp. 73-84, 2018.

[6] Techopedia:,"Single-Layer Neural Network", s interneta, https://www.techopedia.com/definition/33267/single-layer-neural-network, 24.7.2022.

[7] Simplilearn:, "An Overview on Multilayer Perceptron (MLP)", s interneta, https://www.simplilearn.com/tutorials/deep-learning-tutorial/multilayer-perceptron, 25.7.2022.

[8] Merenda M., Porcaro C., Iero D.:"Edge machine learning for ai-enabled iot devices: A review.", Sensors, Vol. 20, No. 9, 2020.

[9] Montavon G., Samek W., Müller K. R.: "Methods for interpreting and understanding deep neural networks", Digital Signal Processing, Vol. 73, pp. 1-15, 2018.

[10] ZackHodari:, "Merlin: The Neural Network (NN) based Speech Synthesis System", s interneta, https://github.com/CSTR-Edinburgh/merlin, 26.7.2022.

[11] O'Shea K., Nash R.: "An introduction to convolutional neural networks", arXiv preprint arXiv:1511.08458, 2015.

[12] Car Z.: "Konvolucijske neuronske mreže", materijali s predavanja, 2022.

[13] Goodfellow I., Bengio Y., Courville A.: "Deep learning", MIT press, 2016.

[14] Saha S: "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way", s interneta, https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, 26.7.2022.

[15] Brownlee J.: "A Gentle Introduction to Pooling Layers for Convolutional Neural Networks", s interneta, https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/,26.7.2022.

[16] Yani M., Irawan S., Setianingsih, C.: "Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail", Journal of Physics: Conference Series, Vol. 1201, No. 1, 2019.

[17] Papers With Code: "Object Detection", s interneta, https://paperswithcode.com/task/object-detection, 2.8.2022.

[18] Liu, W., Anguelov, D., Erhan, D. i dr.:"Ssd: Single shot multibox detector", European conference on computer vision, pp. 21-37, 2016.

[19] Upesh N., Eslamiat H.: "Comparing YOLOv3, YOLOv4 and YOLOv5 for autonomous landing spot detection in faulty UAVs", Sensors, Vol. 22, No. 2, 2022.

[20] Sozzi M., Cantalamessa S., Cogato A. i dr.: "Automatic bunch detection in white grape varieties using YOLOv3, YOLOv4, and YOLOv5 deep learning algorithms", Agronomy, Vol. 12, No. 2, 2022.

[21] Joseph Nelson, Jacob Solawetz: "Responding to the Controversy about YOLOv5", s interneta, https://blog.roboflow.com/yolov4-versus-yolov5, 3.8.2022.

[22] LA Tran: "YOLOv4–5D: An Enhancement of YOLOv4 for Autonomous Driving", s interneta, https://towardsdatascience.com/yolov4-5d-an-enhancement-of-yolov4-for-autonomous-driving-2827a566be4a, 2.8.2022.

[23] Technostacks: "YOLO Vs. SSD: Choice of a Precise Object Detection Method", s interneta, https://technostacks.com/blog/yolo-vs-ssd/, 7.8.2022.

[24] GeeksforGeeks: "Difference between YOLO and SSD", s interneta, https://www.geeksforgeeks.org/difference-between-yolo-and-ssd/ , 7.8.2022.

[25] heartexlabs.: "Labelimg.", s interneta, https://github.com/heartexlabs/labelImg, 16.8.2022.

[26] Techzizou.: "TRAIN A CUSTOM YOLOv4 OBJECT DETECTOR (Using Google Colab)", s interneta, https://medium.com/analytics-vidhya/train-a-custom-yolov4-object-detector-using-google-colab-61a659d4868, 16.8.2022.

[27] Canu S..: "YOLO object detection using Opencv with Python", s interneta, https://pysource.com/2019/06/27/yolo-object-detection-using-opencv-with-python, 20.8.2022.

# Sažetak

U ovom se radu opisuje postupak izrade algoritma za detekciju popunjenosti parkirnih mjesta. Zadatak je bio napraviti algoritam tako da koristi neuronske mreže, to jest detekciju objekata. Objašnjeni su neki od algoritama detekcije objekata koji su pogodni za ovu primjenu i pokazano je kako bi oni mogli utjecati na točnost i brzinu rada sustava. Odabran je YOLOv4 model i objašnjena je njegova struktura, prednost i mane u usporedbi s drugim modelima detekcije objekata. Pokazan je postupak prikupljanja podataka za treniranje modela, postupak treniranja modela i testiranja te validacije modela. Sustav detekcije popunjenosti parkirnih mjesta razvijen je pomoću programskog jezika Python. Implementirana je i vizualizacija detekcije popunjenosti. Rad algoritma je testiran i potvrđen na više parkirnih kamera.

***Ključne riječi*** **— detekcija objekata, konvolucijske neuronske mreže, neuronske mreže, parking, popunjenost parkirnih mjesta, Python, Umjetna inteligencija, YOLOv4**

# Abstract

This paper describes the procedure for creating an algorithm for detecting the occupancy of parking spaces. The task was to create an algorithm that uses neural networks, that is, object detection. Some of the object detection algorithms suitable for this application are explained and it is shown how they could affect the accuracy and speed of the system. The YOLOv4 model is selected and its structure, advantages and disadvantages compared to other object detection models are explained. The procedure for collecting data for model training, the procedure for model training and testing, and model validation is presented. The parking space occupancy detection system was developed using the Python programming language. Occupancy detection visualization has also been implemented. The working algorithm was tested and confirmed on several parking cameras.

***Keywords*** **— object detection, convolutional neural networks, neural networks, parking, occupancy of parking spaces, Python, Artificial Intelligence, YO-**

**LOv4**

# Dodatak A

# Programski kodovi i skripte

## A.1 Kod za treniranje YOLOv4 modela

```
%cd ..
from google.colab import drive
drive.mount('/content/gdrive')
!ln -s /content/gdrive/My Drive/ /mydrive
%cd /mydrive/yolov4
!git clone https://github.com/AlexeyAB/darknet
%cd darknet/
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile
!sed -i 's/LIBSO=0/LIBSO=1/' Makefile
!make
!unzip /mydrive/yolov4/obj.zip -d data/
!cp /mydrive/yolov4/yolov4-custom.cfg cfg
!cp /mydrive/yolov4/obj.names data
!cp /mydrive/yolov4/obj.data data
!cp /mydrive/yolov4/process.py .
```

```
!python process.py
!ls data/
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet
_yolo_v3_optimal/yolov4.conv.137
!./darknet detector train data/obj.data cfg/yolov4-custom.cfg
yolov4.conv.137 -dont_show -map
```

```
import tensorflow as tf
tf.test.gpu_device_name()
%cd cfg
!sed -i 's/batch=64/batch=1/' yolov4-custom.cfg
!sed -i 's/subdivisions=16/subdivisions=1/' yolov4-custom.cfg
%cd ..
```

```
def imShow(path):
import cv2
import matplotlib.pyplot as plt
%matplotlib inline
image = cv2.imread(path)
height, width = image.shape[:2]
resized_image = cv2.resize(image,(3*width, 3*height), interpolation = cv2.INTER_CUBIC)
fig = plt.gcf()
fig.set_size_inches(18, 10)
plt.axis('off')
plt.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
plt.show()
```

```
!./darknet detector test data/obj.data cfg/yolov4-custom.cfg /mydrive/yolov4/training/yolov4-
custom_last.weights/mydrive/yolov4/im2.jpg - thresh 0.3
imShow('predictions.jpg')
```

## A.2 Modificirana skripta "process.py"

```
import glob, os

# Current directory
current_dir = os.path.dirname(os.path.abspath(__file__))

print(current_dir)

current_dir = 'data/obj'

# Percentage of images to be used for the test set
percentage_test = 0;

# Create and/or truncate train.txt and test.txt
file_train = open('data/train.txt', 'w')
file_test = open('data/test.txt', 'w')

# Populate train.txt and test.txt
counter = 1

for pathAndFilename in glob.iglob(os.path.join(current_dir, "*.jpeg")):
  title, ext = os.path.splitext(os.path.basename(pathAndFilename))
  file_train.write("data/obj" + "/" + title + '.jpeg' + "\n")
  counter = counter + 1
```

## A.3 Programski kod za detekciju i prepoznavanje popunjenosti parkirnih mjesta

```
import cv2
import numpy as np
import Crtanje #def crtanje_točaka(width, height,CentarX,CentarY):




# Load Yolo
net = cv2.dnn.readNet("yolov4-custom_last.weights", "yolov4-custom.cfg")

classes = ["car"]

layer_names = net.getLayerNames()
output_layers = [layer_names[i - 1] for i in net.getUnconnected
OutLayers()]
colors = np.random.uniform(0, 255, size=(len(classes), 3))

# Loading image
img = cv2.imread("slika.jpeg")
img = cv2.resize(img, None, fx=0.9, fy=0.9) #bolje slika detektira
height, width, channels = img.shape



# Detecting objects
blob = cv2.dnn.blobFromImage(img, 0.00392, (416, 416), (0, 0, 0)
, True, crop=False)
net.setInput(blob)
outs = net.forward(output_layers)
print(outs)
```

```
# Showing informations on the screen
class_ids = []
confidences = []
boxes = []
CoordinateX = []
CoordinateY = []
for out in outs:
        for detection in out:
                scores = detection[5:]
                class_id = np.argmax(scores)

        if(class_id == 0):

                confidence = scores[class_id]

        if confidence > 0.5:

                # Object detected
                center_x = int(detection[0] * width)
                center_y = int(detection[1] * height)

                #pomaknut za bolju tocnost na desno/lijevo
                #CoordinateX.append(center_x + int(15))
                #CoordinateY.append(center_y + int(15))

                #CoordinateX.append(center_x)
                #CoordinateY.append(center_y)

                #desni/lijevi parking
                CoordinateX.append(center_x)
                CoordinateY.append(center_y + int(30))

                #Japan1
```

```
                #CoordinateX.append(center_x)
                #CoordinateY.append(center_y + int(100))

                #ovaj dio vezan za centar i to treba
                # regulirat ovisno o parkingu,
                #kasnije parametrima di se kamera nalazi.

                #komentari iznad odnose se na korekciju detekcija

                w = int(detection[2] * width)
                h = int(detection[3] * height)

                # Rectangle coordinates
                x = int(center_x - w / 2)
                y = int(center_y - h / 2)
                boxes.append([x, y, w, h])
                confidences.append(float(confidence))
                class_ids.append(class_id)
                #print(class_id)



# remove noise
indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)

#ovo je iskljucivo crtanje pravokutnika
font = cv2.FONT_HERSHEY_PLAIN
for i in range(len(boxes)):
        if i in indexes:
                x, y, w, h = boxes[i]
                label = str(classes[class_ids[i]])
                color = (0,255,0)
                cv2.rectangle(img, (x, y), (x + w, y + h), color, 2)
                cv2.putText(img, label, (x, y + 30), font, 3, color, 3)
```

```
#print(CoordinateX)
#print(CoordinateY)

#print('-------------------------------------------')

Mjesta, ImeMjesta = Crtanje.crtanje_točaka(width,
height,CoordinateX,CoordinateY)

print(ImeMjesta)
print(Mjesta)

#Ovo prebrojava broj punih i praznih mjesta,
#"BrojMjesta" je kolko ih ima a "Prazno" kolko ih je prazno.
Popunjeno = 0
for i in range(len(Mjesta)):
        if( Mjesta[i] == True ):
        Popunjeno = Popunjeno + 1

BrojMjesta = len(Mjesta)
Prazno = BrojMjesta - Popunjeno
print("-----------------------------------------")

#Ovo printa mjesta o redu i govori jesu
#prazna il su puna, to jest formatirano ispisuje.
for i in range(len(ImeMjesta)):
        if(Mjesta[i] == True):
        print(str(ImeMjesta[i]) + '---- Popunjeno')
        else:
        print(str(ImeMjesta[i]) + '---- Prazno')

print('Mjesta slobodno: ' + str(Prazno))

cv2.imshow("Image", img)
cv2.waitKey(0)
```

```
cv2.destroyAllWindows()
#cv2.imwrite('C:/Users/Karlo Severinski/Desktop/Praksa proba
 /Test_output_15_9_2021/test_1.jpeg', img)
        
```

## A.4 Skripta "Crtanje.py"

```
from shapely.geometry import Point
from shapely.geometry.polygon import Polygon
import matplotlib.pyplot as plt


def crtanje_točaka(width, height,CentarX,CentarY):

#parking veliki gradska uprava

ParkingSpaceCoordinate = [

{ "name": "h1", "T1": [1188,757],"T2":[1285,513],
          "T3":[1520,537], "T4": [1475,791] },
{ "name": "h2", "T1": [788,702],"T2":[971,486],
          "T3":[1207,505], "T4": [1068,746] },
{ "name": "r1", "T1": [567,669],"T2":[783,464],
         "T3":[971,484], "T4": [788,703] },
{ "name": "r2", "T1": [374,640],"T2":[596,452],
          "T3":[776,464], "T4": [566,666] },
{ "name": "r3", "T1": [214,609],"T2":[450,429],
          "T3":[601,445], "T4": [376,635] },
{ "name": "r4", "T1": [84,583],"T2":[311,428],
          "T3":[457,435], "T4": [213,606] },
{ "name": "p1", "T1": [1097,314],"T2":[1173,227],
          "T3":[1336,239], "T4": [1276,324] },
{ "name": "p2", "T1": [929,303],"T2":[1027,220],
```

```
          "T3":[1168,224], "T4": [1098,315] },
{ "name": "p3", "T1": [769,297],"T2":[885,210],
         "T3":[1029,219], "T4": [932,305] },
{ "name": "p4", "T1": [626,292],"T2":[759,212],
          "T3":[882,212], "T4": [772,297] },
{ "name": "p5", "T1": [497,290],"T2":[636,210],
          "T3":[762,210], "T4": [625,293] },

]


#ulaz desno uprava
"""
ParkingSpaceCoordinate = [

{ "name": "r1", "T1": [438,356],"T2":[747,303],
          "T3":[1409,627], "T4": [947,819] },
{ "name": "r2", "T1": [303,225],"T2":[510,193],
          "T3":[754,305], "T4": [435,358] },

]
"""

#ulaz lijevo uprava
"""
ParkingSpaceCoordinate = [

{ "name": "r1", "T1": [527,575],"T2":[1208,281],
          "T3":[1502,342], "T4": [966,761] },
{ "name": "r2", "T1": [1207,280],"T2":[1441,189],
          "T3":[1631,231], "T4": [1502,343] },

]
"""
```

```
# Japan- chiba_asahi
"""
ParkingSpaceCoordinate = [

{ "name": "p1", "T1": [1474,951],"T2":[741,599],
          "T3":[1083,472], "T4": [1691,685] },
{ "name": "p2", "T1": [1658,672],"T2":[1007,450],
          "T3":[1189,353], "T4": [1760,528] },
{ "name": "p3", "T1": [1618,470],"T2":[1134,336],
          "T3":[1256,280], "T4": [1706,412] },

]
"""
# Japan2 - defeway_japan_chn_0
"""
ParkingSpaceCoordinate = [

{ "name": "p1", "T1": [0,200],"T2":[4,93],
          "T3":[73,92], "T4": [90,193] },
{ "name": "p2", "T1": [91,197],"T2":[76,83],
          "T3":[161,75], "T4": [220,181] },
{ "name": "p3", "T1": [221,185],"T2":[170,88],
          "T3":[260,82], "T4": [333,169] },
{ "name": "p4", "T1": [339,167],"T2":[258,85],
          "T3":[333,82], "T4": [428,158] },

]
"""
#Japan3 - defeway_japan_chn_1
"""
ParkingSpaceCoordinate = [

{ "name": "p1", "T1": [396,199],"T2":[501,113],
          "T3":[592,136], "T4": [525,238] },
{ "name": "p2", "T1": [297,161],"T2":[411,98],
```

```
            "T3":[492,117], "T4": [402,193] },
{ "name": "p3", "T1": [215,138],"T2":[338,85],
            "T3":[412,99], "T4": [293,164] },
{ "name": "p6", "T1": [53,264],"T2":[121,226],
           "T3":[19,152], "T4": [0,203] },

]
"""

#Japan4 - Osaka
"""
ParkingSpaceCoordinate = [

{ "name": "p1", "T1": [111,716],"T2":[87,439],
            "T3":[314,380], "T4": [501,716] },
{ "name": "p2", "T1": [505,718],"T2":[314,386],
            "T3":[518,340], "T4": [919,715] },
{ "name": "p3", "T1": [920,719],"T2":[523,348],
            "T3":[685,313], "T4": [1047,585] },
{ "name": "p4", "T1": [80,213],"T2":[124,114],
            "T3":[213,102], "T4": [202,196] },
{ "name": "p5", "T1": [201,198],"T2":[210,97],
            "T3":[294,95], "T4": [323,183] },
{ "name": "p6", "T1": [325,180],"T2":[295,97],
            "T3":[382,93], "T4": [444,169] },
{ "name": "p7", "T1": [449,180],"T2":[382,96],
            "T3":[465,92], "T4": [561,169] },

]
"""

#Taiwan
"""
ParkingSpaceCoordinate = [
```

```
{ "name": "p1", "T1": [36,271],"T2":[238,100],
          "T3":[100,97], "T4": [0,148] },
{ "name": "p2", "T1": [317,266],"T2":[382,103],
          "T3":[241,100], "T4": [54,257] },
{ "name": "p3", "T1": [588,285],"T2":[524,105],
          "T3":[380,105], "T4": [317,268] },
{ "name": "p6", "T1": [797,213],"T2":[702,109],
          "T3":[523,105], "T4": [587,268] },

]
"""

#Amerika posta
"""
ParkingSpaceCoordinate = [

{ "name": "p1", "T1": [0,358],"T2":[37,323],
          "T3":[77,325], "T4": [27,377] },
{ "name": "p2", "T1": [28,374],"T2":[76,327],
          "T3":[120,329], "T4": [77,379] },
{ "name": "p3", "T1": [76,380],"T2":[122,329],
          "T3":[165,332], "T4": [127,385] },
{ "name": "p4", "T1": [128,383],"T2":[167,327],
         "T3":[212,331], "T4": [179,389] },
{ "name": "p5", "T1": [178,390],"T2":[217,325],
          "T3":[260,329], "T4": [230,393] },
{ "name": "p6", "T1": [231,392],"T2":[260,333],
          "T3":[308,334], "T4": [288,397] },
{ "name": "p7", "T1": [290,391],"T2":[310,333],
         "T3":[356,335], "T4": [345,393] },
{ "name": "p8", "T1": [345,395],"T2":[358,334],
          "T3":[407,333], "T4": [403,401] },
{ "name": "p9", "T1": [404,401],"T2":[407,336],
          "T3":[456,331], "T4": [462,405] },
{ "name": "p10", "T1": [461,402],"T2":[455,334],
```

```
            "T3":[508,333], "T4": [520,402] },
{ "name": "p11", "T1": [520,404],"T2":[506,335],
            "T3":[554,334], "T4": [576,401] },
{ "name": "p12", "T1": [578,398],"T2":[556,338],
            "T3":[606,335], "T4": [634,400] },
{ "name": "p13", "T1": [637,408],"T2":[608,341],
            "T3":[657,337], "T4": [692,405] },

]
"""

#prazna lista
ParkingNames = []

#lista sa mjestima, gje se svako referira na popunjenost mjesta
PopunjenostMjesta = [False]*len(ParkingSpaceCoordinate)

#print(PopunjenostMjesta) ---- dobro dode za provjere

#ovo su je samo broj poligona na temelju mjesta
POLI = [None]*len(ParkingSpaceCoordinate)

#i = 0
#print("----------")

#ovo je radi crtanja, pa je prozor
#u kome se crta slika 20x10 u incima
plt.rcParams["figure.figsize"] = [20, 10]
plt.rcParams["figure.autolayout"] = True


for i in range(len(ParkingSpaceCoordinate)):

#Vadenje tocaka iz liste
List = ParkingSpaceCoordinate[i]
```

```
Dot1a = List["T1"]
Dot2a = List["T2"]
Dot3a = List["T3"]
Dot4a = List["T4"]

#skaliranje tocaka, tako da bi se dobila tocna
#slika gdje se preklapaju poligoni i tocke, bitno bi bilo skalirat
#svaku sliku na neku određenu veličinu koja ce bit default.
#Netreba mijenjat ako nam je slika sa kamere dovoljno velika!
Dot1 = [int(i * 0.9) for i in Dot1a]
Dot2 = [int(i * 0.9) for i in Dot2a]
Dot3 = [int(i * 0.9) for i in Dot3a]
Dot4 = [int(i * 0.9) for i in Dot4a]

#definiranje poligona
POLI[i] = Polygon([Dot1, Dot2, Dot3, Dot4])
#print("POLIGON:   ")
#print(POLI[i])

#priprema poligona za crtanje
x, y = POLI[i].exterior.xy
plt.plot(x,y, c="red")

#cranje tocaka oko poligona, radi potrebe vizualizacije
plt.plot(Dot1[0],Dot1[1], '-ro')
plt.plot(Dot2[0],Dot2[1],'-ro')
plt.plot(Dot3[0],Dot3[1], '-ro')
plt.plot(Dot4[0],Dot4[1], '-ro')



#petja koja crta sve tocke, crta plave "x" znakove
#for i in range(len(CentarX)):
#plt.plot(CentarX[i],CentarY[i], '-bx')

#crta samo točke "x" koje su pronađene unutar
```

```
#parkinga, ova funckija iznad crta sve
for i in range(len(ParkingSpaceCoordinate)):

ParkingNames.append(ParkingSpaceCoordinate[i]["name"])

for j in range(len(CentarX)):

point = Point(CentarX[j],CentarY[j])

if(POLI[i].contains(point)):
PopunjenostMjesta[i] = True
plt.plot(CentarX[j],CentarY[j], '-bx')
break


#crta velicinu slike radi raspona, potreba vizualizacije
plt.plot(width, height, '-ro')
plt.plot(0,0,'-ro')
plt.show()
print("--------")

#vraca koja su mjesta popunjena i imena parkinga
#kako su definirana u strukturama na početku funkcija
return PopunjenostMjesta, ParkingNames
```