OpenCV

Držaić, Jelena

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Jelena Držaić

OPENCV

Diplomski rad

Voditelj rada:
doc. dr. sc. Zvonimir Bujanović

Zagreb, Veljača, 2018.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Računalni vid naziv je za tehnološku i znanstvenu disciplinu koja se bavi načinima na koje računala mogu razumijeti sliku ili video. Računalni vid srodan je mnogim drugim znanstvenim disciplinama, od automatike, robotike i obrade slika pa sve do umjetne inteligencije. Mogli bismo reći i da je današnji računalni vid i dio umjetne inteligencije.

U ovom radu donosimo pregled biblioteke OpenCV, jedne od najpoznatijih biblioteka s primjenom u području računalnog vida. Uz to, u praktičnom dijelu rada fokusirat ćemo se na mogućnosti integracije ove biblioteke s platformom Android.

U prvom poglavlju rada ukratko opisujemo osnove računalnog vida, njegove primjene i povijest razvoja te donosimo osnove biblioteke OpenCV i motivaciju za njeno korištenje. U nastavku rada detaljnije dajemo pregled funkcionalnosti biblioteke OpenCV - opisujemo strukturu bibloteke, osnovne strukture podataka te detaljnije opisujemo neke od algoritama i njihovu teorijsku podlogu.

Treće poglavlje fokusirano je na integraciju OpenCV-a s operacijskim sustavom Android. Ukratko opisujemo Android i navodimo načine za korištenje OpenCV-a u sklopu Android aplikacije. Posebno ističemo platformu NVIDIA Tegra koja se koristi u uređajima s operacijskim sustavom Android, a za nju postoje vrlo značajne optimizacije dijelova biblioteke OpenCV.

U zadnjem dijelu rada bavimo se praktičnim dijelom rada - izradom Android aplikacije za rješavanje Sudokua. Opisana je aplikacija i njene funkcionalnosti te su detaljno pojašnjene komponente aplikacije vezane uz računalni vid, a koje su gotovo u potpunosti izrađene korištenjem funkcionalnosti biblioteke OpenCV.

Poglavlje 1

Računalni vid. Biblioteka OpenCV

1.1 Uvod

Kroz ovo poglavlje ukratko ćemo opisati osnove računalnog vida kao grane, motivaciju iza razvitka te neke od bitnijih primjena. Nakon toga dajemo prikaz OpenCV-a, biblioteke otvorenog koda koja je jedna od najčešće korištenih biblioteka u području računalnog vida.

1.2 Osnove računalnog vida

Cilj algoritama računalnog vida je automatizirati zadaće jednake onima ljudskog vidnog sustava. To uključuje ekstrakciju, analizu i razumijevanje informacija iz jedne ili više slika (npr. uzastopni frameovi danog videa).

Navodimo nekoliko primjera problema računalnog vida.

- Odlučivanje nalazi li se na slici lice osobe;
- Prepoznavanje rukom pisanih znamenaka;
- Prepoznavanje teksta na slici;
- Klasifikacija slika cvijeća na pripadne vrste;
- Praćenje objekata objektivom kamere;
- Promjena veličine slike;
- Primjena efekata na slike (npr. *sepia* efekt, prilagodba prozirnosti ili kontrasta).

Povijesni pregled

Razvoj računalnog vida započeo je u 60-im godinama 20. stoljeća, paralelno prvim istraživanjima na području umjetne inteligencije [8]. Neki od prvih problema kojima su se znanstvenici bavili bio je problem prepoznavanja 3D objekata na slici, kako bi se postiglo potpuno razumijevanje scene. U 70-im su godinama i začetci brojnih standardnih algoritama računalnog vida, kao što su algoritmi za detekciju linija i rubova i praćenje pokreta objekata (eng. *Optical flow*).

U 90-im godinama prvi se puta javlja i primjena statističkih metoda u računalnom vidu (npr. algoritam *Eigenfaces*). Također, sve je veći presjek računalne grafike i računalnog vida, kroz spajanje slika (eng. *Image stitching*), kreiranje 3D modela iz skupa slika (eng. *Image-based rendering*), ...

U posljednjih dvadesetak godina računalni se vid razvija sve brže, prvenstveno iz dva razloga:

- **Bolji hardver**
  Računala postaju sve jeftinija i dostupnija za upotrebu. Na primjer, u svrhu treniranja modela za prepoznavanje mačaka u *YouTube* videima korišten je distribuirani sustav sastavljen od 16000 računala [7]. S druge strane, hardver je bolje dizajniran za algoritme računalnog vida. GPU (*Graphics Processing Unit*) su se pokazale kao vrlo učinkovit pristup kod korištenja neuronskih mreža (u svrhu problema iz spektra računalnog vida, ali i općenito).

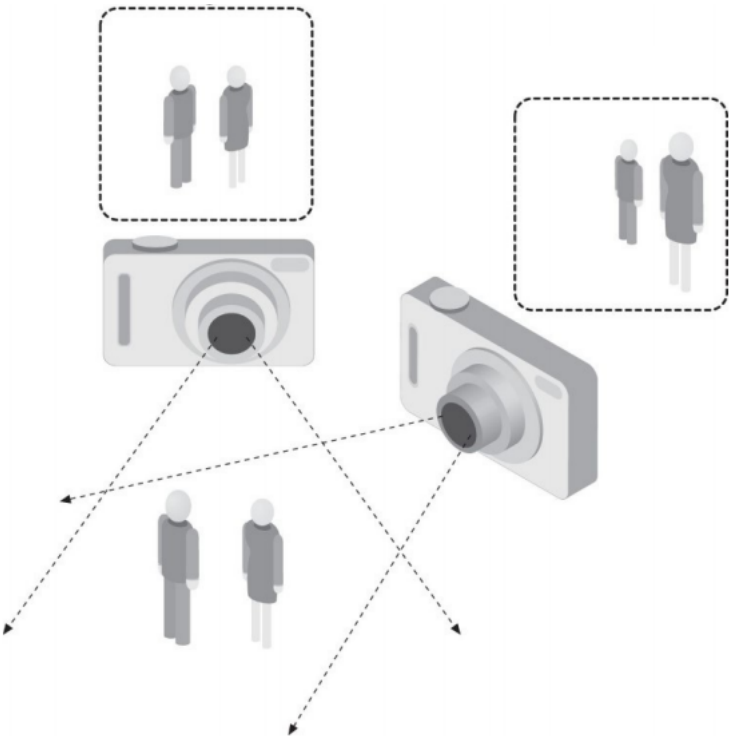
- **Dostupne veće količine podataka**
  U primjeru spomenutom ranije, na 16000 računala analizirano je 10 milijuna video zapisa. Količina dostupnih podataka uzrokuje bolju kvalitetu razvijenih modela.

Danas, veliki je dio primjene računalnog vida i umjetne inteligencije općenito fokusiran na razvoj različitih vrsta neuronskih mreža, koje su, u svojoj elementarnom obliku, i bile jedan od prvih koncepata u području umjetne inteligencije.

Ljudski i računalni vid

Budući da se mi, ljudi, u svakodnevnom životu u vrlo velikoj mjeri oslanjamo na osjetilo vida, često nam se čini kako su problemi iz područja računalnog vida trivijalni. Na primjer, koliko teško može biti prepoznati ljudsko lice na fotografiji? Potrebno je uzeti u obzir da je ljudski vidni sustav vrlo kompliciran - od oka do ljudskog mozga dolazi mnogo različitih signala koji nam omogućuju da razumijemo ono što vidimo. Nadalje, vrlo je važna i činjenica da kod "obrade slike" koristimo iskustvo stečeno kroz godine svakodnevne izloženosti sličnim situacijama. S druge strane, računala ulaznu sliku vide samo kao matricu brojeva, inicijalno bez ikakvih dodatnih informacija. Kod prepoznavanja objekata,

ne postoji 1-1 veza između objekta i njegove slike. Kao što je to vidljivo na slici 1.1, ovisno o npr. kutu fotografiranja, razlika između dobivenih slika vrlo je zamjetna. Dodatno, u realnim situacijama vrlo često slike sadrže i dodatne šumove (eng. *noise*) u podatcima, kao što su refleksije, osvjetljenje i slično.



Slika 1.1: Isti objekti fotografirani iz različitih kuteva bit će reprezentirani na različite načine

U nastavku navodimo neke od najvažnijih primjena računalnog vida.

Primjene

- **Medicinski imaging (eng. *Medical computer vision*)**
  Područje koje se bavi ekstrakcijom informacija iz slika u svrhu medicinske dijagnostike. Neki od primjera su mikroskopske i rendgenske snimke, ultrazvuk te angiografija. Računalni vid najčešće je korišten kako bi se detektirali tumori, problemi s krvotokom.

- **Strojni vid (eng. *Machine vision*)**
  Primjena računalnog vida u industriji, kako bi se došlo do informacija korisnih u

raznim proizvodnim procesima. Jedan od primjera je kontrola kvalitete (eng. *quality control*) - proizvodi se provjeravaju automatski te se izdvajaju oni koji ne zadovoljavaju zadane standarde.

- **Autonomna vozila (eng. *Autonomous vehicles*)**
  Spektar sustava - od potpuno autonomnih automobila pa do sustava za slijetanje zrakoplova ili detekciju opasnosti u automobilu.

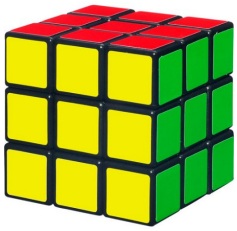
Prije nego započnemo s opisivanjem biblioteke OpenCV, kojom ćemo se baviti i u većem dijelu rada, dajemo osnove računalnog prikaza slike, koje će nam kasnije biti korisne.

1.3 Računalni prikaz slike

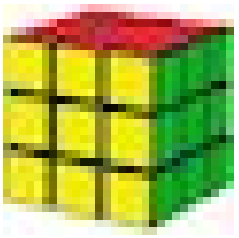
Sliku možemo predstaviti funkcijom f dviju varijabli tj. koordinata (x, y). Cilj je sliku prikazati u memoriji računala, što znači da je domena funkcije $D_f \subset \mathbb{N}^2$.
Za $D_f = \{(x, y) : x \in \{1, 2, ..., m\}, y \in \{1, 2, ..., n\}\}$ vrijednosti funkcije reprezentiramo matricom $M \in \mathbb{R}^{m \times n}$. Elementi te matrice nazivaju se *pikselima*.

Broj piksela na slici nazivamo *rezolucija* ili *razlučivost*. Veća razlučivost slike jamči bolji prikaz detalja, no u isto vrijeme povećava vrijeme potrebno za obradu slike. Zbog toga je u računalnom vidu često potrebno pronaći balans između visoke rezolucije slike i prihvatljivog vremena obrade.

Način prikaza piksela u memoriji može biti različit, ovisno o primjeni; u nastavku su opisani standardni načini prikaza.



(a) Rezolucija 450x450



(b) Rezolucija 50x50

Slika 1.2: Slike različitih rezolucija

Slika u boji

Čuvanje slike u boji zahtjevnije je od korištenja crno-bijele ili binarne slike, no u brojnim problemima računalnog vida može biti korisno za njihovo rješavanje. Postoji nekoliko standarda za čuvanje slika u boji u memoriji.

RGB (Red-Green-Blue)

Vrijednost piksela se čuva kao utjecaj triju boja - crvene, zelene i plave. Svaka boja tipično se zadaje s 8 bitova, što nam ukupno daje $16,777,216$[1] boja. Ako želimo pretvoriti sliku iz RGB u akromatsku, možemo koristiti formulu

$$s_{(i,j)} = 0.299 \cdot r_{(i,j)} + 0.587 \cdot g_{(i,j)} + 0.114 \cdot b_{(i,j)} \tag{1.1}$$

gdje je $r_{(i,j)}$ iznos crvene, $g_{(i,j)}$ zelene, a $b_{(i,j)}$ plave boje [9].

CMY (Cyan-Magenta-Yellow)

Kao i *RGB*, ovaj standard temelji se na tri boje - cijan (C), magenta (M) i žuta (Y). Zapravo, vrijednost piksela dobiva se oduzimanjem od vrijednosti bijele boje (255), što je pogodno kod ispisa na papir pa se ovaj standard često koristi kod pisača. U nastavku je formula za pretvorbu iz *RGB* modela.

$$\begin{aligned} C &= 255 - R, \\ M &= 255 - G, \\ Y &= 255 - B \end{aligned} \tag{1.2}$$

HLS (Hue-Luminance-Saturation)

Često korišten model u sklopu računalnog vida, budući da omogućuje razdvajanje *akromatske* (crne, bijele ili sive) svjetlosti od *kromatske*. Sjajnost je dana komponentom *Luminance* (L), nijansa boje komponentom *Hue* (H), a zasićenost sa *Saturation* (S). Pretvorba iz *RGB* vrši se na sljedeći način. Neka su R, G i B vrijednosti u *RGB* modelu, skalirane na interval $[0, 1]$.

$$L = \frac{\max(R, G, B) + \min(R, G, B)}{2} \tag{1.3}$$

$$S = \begin{cases} \frac{\max(R,G,B) - \min(R,G,B)}{\max(R,G,B) + \min(R,G,B)}, & L < 0.5 \\ \frac{\max(R,G,B) - \min(R,G,B)}{2 - (\max(R,G,B) + \min(R,G,B))}, & L \geq 0.5 \end{cases} \tag{1.4}$$

[1] 256 * 256 * 256 = 16,777,216

$$H = \begin{cases} \frac{60\cdot(G-B)}{S}, & R = \max(R,G,B) \\ \frac{120+60\cdot(B-R)}{S}, & G = \max(R,G,B) \\ \frac{240\cdot(R-G)}{S}, & B = \max(R,G,B) \end{cases} \tag{1.5}$$

Ako u formuli (1.5) vrijednost H ispadne negativna, dodaje joj se 360.

Crno-bijela slika

U ovom slučaju svaki piksel sadrži samo informaciju o sjajnosti zadane točke na slici. Broj nijansi određen je brojem bitova koje koristimo za prikaz piksela u memoriji računala. Najčešće je korišteno 8 bitova, što nam daje 256 različitih nijansi.

Crno-bijela slika često se koristi u računalnom vidu jer je jednostavnija za rad te zauzima manje memorije od prikaza u boji, što smanjuje vrijeme njene obrade.

Binarna slika

Ponekad, u problemu računalnog vida, želimo samo odrediti je li neki piksel dio traženog objekta ili svojstva, neovisno o njegovoj boji. Npr. takva se potreba javlja kod prepoznavanja rubova na slici. U tu svrhu možemo koristiti binarnu sliku. Binarnu sliku konstruiramo na sljedeći način: neka je b granična vrijednost, iz skupa vrijednosti piksela početne slike (bilo kojeg formata). Tada za piksele binarne slike B postavljamo

$$B(i,j) = \begin{cases} 1, & X(i,j) < b \\ 0, & X(i,j) \geq b \end{cases} \tag{1.6}$$

gdje je $X(i,j)$ vrijednost piksela (i,j) početne slike. Metodu korištenu u formuli (1.6) nazivamo *thresholding*[2].

Sada dajemo osnovne informacije o biblioteci OpenCV, koju ćemo kasnije detaljnije opisati.

1.4 Opis biblioteke OpenCV

Osnovne informacije

OpenCV je biblioteka otvorenog koda s funkcionalnostima iz područja računalnog vida. Biblioteka je implementirana u programskim jezicima C i C++, a moguće ju je koristiti pod operacijskim sustavima *Linux, Windows* i *Mac OS X*. Osim ranije navedenih jezika, postoje sučelja prema većini popularnih jezika kao što su *Java*, *Python* i *MATLAB*. Biblioteku je

[2] threshold - engleski naziv za *prag*

moguće koristiti i za izradu mobilnih, *Android* i *iOS* aplikacija. Razvoju aplikacija za *Android* ćemo se vratiti u nastavku rada.

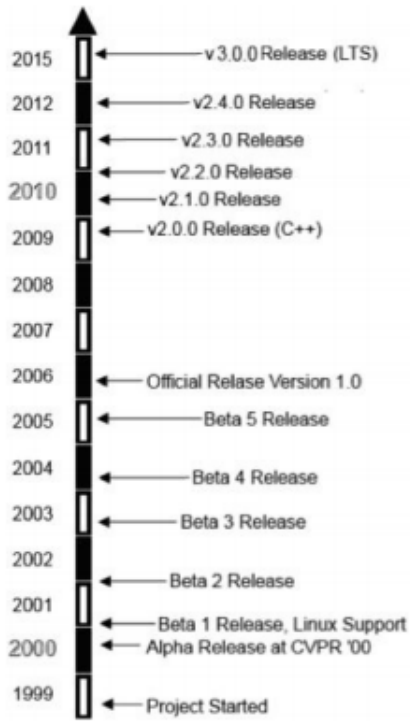
OpenCV sadrži više od 500 funkcija iz spektra računalnog vida [7], kao i modul *Machine learning*, fokusiran na prepoznavanje uzoraka (eng. *pattern recognition*) i klasteriranje.

Povijesni pregled

OpenCV je objavio Gary Bradski, tada dio istraživačkog tima u tvrtci *Intel*, 1999. godine, s ciljem ubrzanja razvitka računalnog vida i umjetne inteligencije. Fokus je bio na tome da se ubrzaju skupe operacije - operacije koje zahtijevaju mnogo CPU resursa. Kasnije, biblioteka je dobila još više na važnosti razvojem višejezgrenih procesora i dolaskom novih primjena računalnog vida.

Biblioteka je postigla veliku popularnost, te je trenutno preuzeta približno 11 milijuna puta, a taj broj raste za 160 tisuća svaki mjesec [7]. Također, budući da je OpenCV biblioteka otvorenog koda, mnogo programera doprinosi njenom razvoju.

Na slici 1.3 dana je vremenska crta razvoja OpenCV-a po verzijama.



Slika 1.3: Vremenska crta razvoja OpenCV-a

OpenCV licenca je strukturirana tako da je moguće razvijati komercijalne aplikacije koristeći cijelu biblioteku ili neki njen dio.

Primjene

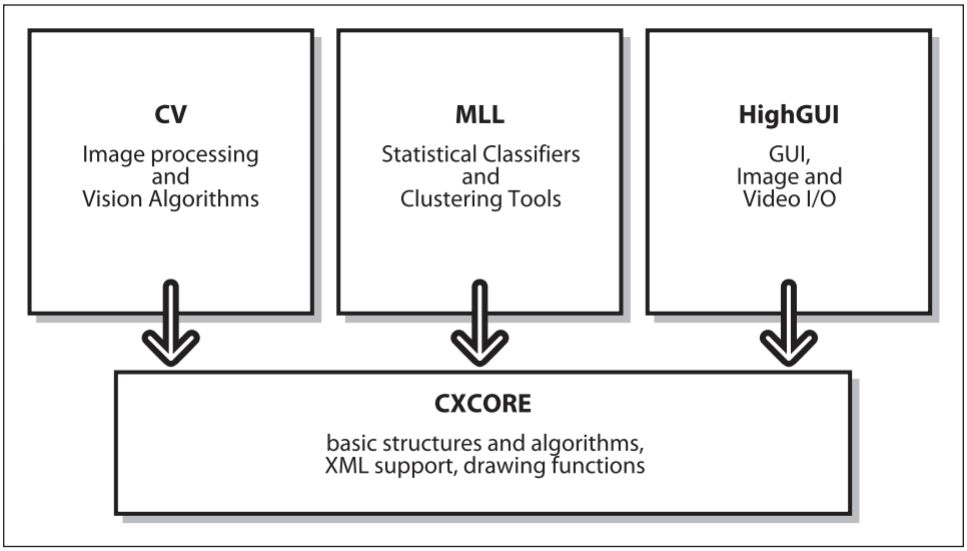
OpenCV u nekim od svojih proizvoda koriste mnoge od najvećih IT tvrtki - *Microsoft*, *Google*, *Intel*, *IBM*, *Honda*, *Sony* te istraživački centri - *Stanford*, *MIT*, *Cambridge* itd. Konkretan primjer primjene je spajanje slika u *Streetviewu* (tzv. *image stitching*), kontrola robota (navigacija u prostoru, interakcija s objektima), provjera deklaracija proizvoda u industriji, praćenje objekata sigurnosnim kamerama.

Poglavlje 2

Sadržaj biblioteke OpenCV

2.1 Uvod

Biblioteka OpenCV podijeljena je u pet osnovnih komponenti [7]. Sve osim *CvAux* su prikazane na slici 2.1 Sada ukratko opisujemo sadržaj i namjenu svake od njih.



Slika 2.1: Osnovna struktura OpenCV-a

- **CXCORE**
  Sadrži osnovne strukture podataka i algoritme, podršku za jezik XML i crtanje.
  To uključuje operacije na kolekcijama, dinamičke strukture podataka (red, skup,

graf, stablo), pohranjivanje podataka (eng. *Data persistance*), obradu grešaka i sistemske funkcije.

- **CV**
  Najopsežniji dio biblioteke. Sadrži algoritme za obradu slike. Također, ovdje se nalaze algoritmi računalnog vida više razine apstrakcije. Opširniji pregled tih algoritama dat ćemo u nastavku ovog poglavlja.

- **MLL**
  Machine Learning Library (MLL) je skup klasa i funkcija koje možemo koristiti za klasifikaciju, regresiju i klasteriranje podataka.

- **HighGUI**
  Iako je OpenCV u većini slučajeva korištena u aplikacijama bez eksplicitnog korisničkog sučelja ili aplikacijama čiji je UI razvijen u nekom od razvojnih okvira (npr. WinForms, Qt, JavaFX), ponekad je korisno imati mehanizam pomoću kojeg je lako brzo vizualizirati rezultate primjenjenih algoritama. HighGUI koristimo upravo u tu svrhu. On pruža jednostavno sučelje za kreiranje prozora, crtanje, procesiranje jednostavnih korisničkih akcija i manipulaciju slikama.

- **CvAux**
  Ova komponenta sadrži algoritme koji više nisu u aktivnom dijelu biblioteke (npr. *Hidden Markov Model* prepoznavanje lica) te ekspetimentalne algoritme (npr. prepoznavanje pozadinskog i prednjeg segmenta slike). Ovaj dio biblioteke prilično je loše dokumentiran te ga je iz tog razloga teže primjenjivati.

U nastavku ovog poglavlja opisujemo osnovne strukture podataka unutar biblioteke OpenCV te opisujemo neke od najčešće korištenih algoritama, s naglaskom na algoritme korištene za razvoj aplikacije iz poglavlja 4 ovog rada.

2.2 Strukture podataka

OpenCV sadrži mnogo različitih struktura podataka, od onih jednostavnih (primitivnih) pa do kompliciranijih struktura za rad s velikim matricama (npr. slikama). Tim redoslijedom ih u nastavku navodimo.

Osnovni (primitivni) tipovi podataka

Ova kategorija sadrži prvenstveno tipove podaka konstruirane direktno iz C++ primitiva (*int*, *float* i sl.), a služe za reprezentaciju jednostavnih polja i matrica, veličina (*size*) i točaka.

Reprezentacija točke (*The point classes*)

Ove klase bazirane su na predlošcima (eng. *templates*), što omogućuje korištenje različitih tipova podataka kod njihovih kreiranja (npr. integer, floating-point). Postoje dva predloška koja služe za prikaz točaka - jedan za 2D, drugi za 3D slučaj. Navedeni tipovi mogu se konvertirati iz starijih, C struktura `CvPoint` i `CvPoint2D32f`.

U tablici 2.1 prikazani su neki od uobičajnih načina korištenja ovih klasa.

Operacija	**Primjer korištenja**

Konstruktor	`cv::Point2f p;`
	`cv::Point3f p2(p1);`
	`cv::Point2i p(2,3);`

Cast u vektor	`(cv::Vec3f) p;`
Pristup komponentama	`p.x; p.y; p.z;`
Skalarni produkt	`float x = p1.dot(p2);`
Vektorski produkt	`cv::Point2f p = p1.cross(p2);`

Tablica 2.1: Primjeri korištenja

Reprezentacija veličine (*The size classes*)

Klase kojima reprezentiramo veličinu u pravilu su slične ranije opisanim `cv::Point_` klasama te su međusobno konvertibilne. Ključne su razlike u nazivima članskih varijabli.

Primjeri korištenja dani su u tablici 2.2.

Operacija	**Primjer korištenja**

Konstruktor	`cv::Size sz;`
	`cv::Size2f sz2(sz1);`
	`cv::Size2i sz(2,3);`

Pristup komponentama	`sz.width; sz.height;`
Površina zadanog pravokutnika	`sz.area();`

Tablica 2.2: Primjeri korištenja

cv::Rect

Klasa koja predstavlja pravokutnik čije su stranice paralelne koordinatnim osima. Kao članske varijable sadrži koordinate gornjeg lijevog kuta pravokutnika (varijabla tipa `cv::Point`) i njegovu veličinu (varijabla tipa `cv::Size`).

Također, ova klasa sadrži preopterećenja mnogih korisnih operatora.

Operacija	**Primjer korištenja**

Konstruktor	`cv::Rect r;`
	`cv::Rect r2(r1);`
	`cv::Rect r(x, y, s, d);`
	`cv::Rect r(p, sz);`

Pristup komponentama	`r.x; r.y; r.width; r.height;`
Površina	`r.area();`
Pripadnost točke pravokutniku	`r.contains(p);`

Tablica 2.3: Primjeri korištenja klase `cv::Rect`

Operacija	**Primjer korištenja**
Presjek `r1` i `r2`	`cv::Rect r3 = r1 & r2;`
Translacija `r` za iznos `x`	`cv::Rect rx = r + x;`
Usporedba `r1` i `r2`	`bool eq = (r1 == r2);`

Tablica 2.4: Primjeri preopterećenih operatora

cv::RotatedRect

`cv::RotatedRect` klasa jedna je od rijetkih OpenCV tipova koje se ne temelje na korištenju predložaka. Kao članske varijable sadrži točku koju zovemo center, veličinu pravokutnika i kut koji određuje rotaciju pravokutnika oko *centra*.

Operacija	**Primjer korištenja**

Konstruktor	`cv::RotatedRect rr();`
	`cv::RotatedRect rr2(rr1);`
	`cv::RotatedRect(p1, p2);`
	`cv::RotatedRect rr(p, sz, theta);`

Pristup komponentama	`rr.center; rr.size; rr.angle;`
Pristup vrhovima	`rr.points(pts[4]);`

Tablica 2.5: Primjeri korištenja klase `cv::RotatedRect`

Matrice statičkih veličina

Ovaj tip podataka koristimo kada nam je veličina matrice poznata za vrijeme kompilacije programa. Kao i većina tipova, temelji se na predlošku - `cv::Matx<>`. Ne preporučuje se korištenje ovog tipa za pohranjivanje velikih matrica. Umjesto toga, bolje je koristiti `cv::Mat`, koji ćemo opisati kasnije.

Operacija	Primjer korištenja

Konstruktor	`cv::Matx33f m;`
	`cv::Matx34f m;`
	`cv::Matx22 m2(m1);`
	`cv::Matx21f m(x0,x1);`

Pristup komponentama	`m(i, j); m(i);`
Izdvajanje 2x2 podmatrice	`m.get_minor<2, 2>(i, j);`
Izdvajanje retka/stupca	`mr = m.row(i); mc = m.col(j);`

Tablica 2.6: Operacije nad `cv::Matx`

Primjetimo, *alias* imena klase kreiran je po pravilu `cv::Matx{1,2,...}{1,2,...}{f,d}` gdje prva dva broja označavaju dimenziju matrice, dok posljednji broj određuje tip podataka (d - double ili f - float).

Polja statičkih veličina

Tip `cv::Vec` samo je predložak specijaliziran iz `cv::Matx` supstituiranjem broja stupaca s 1. Signature metoda analogne su onima definiranim u `cv::Matx` pa ih ne navodimo ponovno.

Još jedan tip podataka, korišten rjeđe, a spada u osnovne tipove podataka, je `cv::Complex{f,d}`, kojim reprezentiramo kompleksne brojeve.

Pomoćni tipovi podataka

Pomoćni tipovi podataka osiguravaju nam kontrolu izvođenja algoritama (npr. kriteriji zaustavljanja), operacije nad spremnicima. Također, tipovi iz ovog skupa implementiraju *garbage collection* sustav te zbog toga nije potrebno brinuti o alokaciji i dealokaciji OpenCV objekata.

`cv::TermCriteria`

Označava kriterij zaustavljanja algoritama. Moguće je koristiti različite kriterije:
`cv::TermCriteria::MAX_ITER` - ograda na broj iteracija i `cv::TermCriteria::EPS` - ograda na *epsilon*, koje proslijeđujemo kao argument *type* pripadnom konstruktoru
`TermCriteria(int type, int maxCount, double epsilon);`

Garbage collection

`cv::Ptr<>a` je OpenCV implementacija tzv. pametnog pokazivača. Korištenje je analogno onom kod standardnih C++ pametnih pokazivača - npr.

```
cv::Ptr<Matx22f> p = makePtr<cv::Matx22f>();
cv::Ptr<Matx22f p(new cv::Matx22f);
```

Također, postoje dodatne funkcije za brojanje referenci - funkcije `addref()` i `release()` za umanjivanje i uvećavanje broja referenci danog pokazivača. Dodatno, `delete_obj()` poziva se automatski u slučaju da brojač referenci padne na 0. Potrebno ju je preopteretiti u izvedenoj klasi budući da inicijalna implementacija ne radi ništa. Na primjer,

```
template<> inline void cv::Ptr<IplImage>::delete_obj() {
    cvReleaseImage(&obj);
}
```

Korisno je još spomenuti da OpenCV procesira pogreške kroz mehanizam iznimki - bazna klasa `cv::Exception` naslijeđena je iz standardne `std::exception`.

Velika polja (*Large array types*)

U ovom odlomku promatramo jednu od centralnih struktura podataka u sklopu biblioteke, `cv::Mat`. Velika većina funkcija su članske funkcije te klase, objekti te klase su im ulaz ili izlaz. `cv::Mat` koristimo za prikaz *dense* (gustih) polja proizvoljne dimenzionalnosti. Spremljeni podatci ne moraju nužno biti osnovnih tipova - svaki element može biti jedan broj ili više njih (tzv. *multichannel array*).

Kreiranje instanci `cv::Mat`

U nastavku su dane signature osnovnih konstruktora `cv::Mat`.

Konstruktor
`cv::Mat;`
`cv::Mat(int rows, int cols, int type);`
`cv::Mat(int rows, int cols, int type, const Scalar& s);`
`cv::Mat(int rows, int cols, int type, void* data, size_t step);`
`cv::Mat(cv::Size sz, int type);`

Tablica 2.7: Konstruktori `cv::Mat`

Kao što smo spomenuli, elementi objekata ovog tipa mogu biti različitih tipova. Ti su tipovi definirani posebnim konstantama, formata `CV_{8U,16S,16U,32S,32F,64F}C{1,2,3}`.

Tako na primjer, `CV_64FC3` implicira da je instanca `cv::Mat` polje 64-bitnih *floatova*, koje ima 3 *kanala*.

Dohvat elemenata

Osnovni način dohvata podataka pojedinačnih elemenata iz matrica je korištenjem predloška funkcije `at<>()`. Ona funkcionira na način da različiti tipovi matrica vrše specijalizaciju tog predloška za dani slučaj.

```
// One channel instance.
cv::Mat m = cv::Mat::eye(5, 5, 32FC1);
printf("Element (2,2) is %f\n", m.at<float>(2,2));
// Multichannel instance.
cv::Mat m = cv::Mat::eye(5, 5, cv::DataType<cv::Complexf>::type);
printf("Element (2,2) is %f + i%f\n", m.at<cv::Complexf>(2,2).re,
    m.at<cv::Complexf>(2,2).im);
```

Aritmetika na `cv::Mat`

Klasa `cv::Mat` preopterećuje mnoge operatore korisne za računanje s instancama. U nastavku je dano nekoliko primjera.

```
m1 + m2; m1 - m2;
m1 * m2; s * m;
m1 / m2;
m1 & m2; m1 | m2; ~m;
```

Rijetke matrice

U slučaju da su naši podatci rijetki (većina elemenata matrice je jednaka 0), koristimo `cv::SparseMat` klasu. Operacije na toj klasi vrlo su slične onima na `cv::Mat`.

2.3 Algoritmi

U nastavku ćemo opisati neke od algoritama iz biblioteke OpenCV. OpenCV ima mnogo funkcionalnosti, a u ovom radu opisujemo primarno one korištene za potrebe izrade aplikacije opisane u poglavlju 4. Uvodno dajemo pregled jednostavnijih operacija koje su korištene u kompleksnijim algoritmima.

Osnovne operacije na matricama i slikama

Bitovne operacije

Bitovne operacije djeluju na pojedinačne elemente matrice. Implementirane su standardne operacije `AND`, `OR`, `XOR` i `NOT`. U nastavku je dana deklaracija funkcije `AND`. Deklaracije ostalih funkcija su potpuno analogne.

```
void cv::bitwise_and(
   cv::InputArray src1,
   cv::InputArray src2,
   cv::OutputArray dst, // Resulting array
   cv::InputArray mask = cv::noArray(), // Optional, do only where nonzero
);
```

Ove funkcije mogu biti korisne kod npr. invertiranja binarnih slika (crna boja postaje bijela, a bijela crna) što je u nekim slučajevima potrebno za pretprocesiranje slika za korištenje s skupom podataka *MNIST*[1].

Pretvorba između različitih prostora boja

Često je potrebno konvertirati npr. sliku u boji u njen crno-bijeli prikaz. *OpenCV* sadrži funkciju za tu svrhu.

```
void cv::cvtColor(
   cv::InputArray src,
   cv::OutputArray dst, // Result array
   int code, // color mapping code
   int dstCn = 0 // channels in output
);
```

Ovdje treći argument, `code`, može poprimiti predefinirane vrijednosti. Navodimo nekoliko češće korištenih.

Konstanta	Značenje
`cv::COLOR_BGR2RGB`	Konverzija iz *BGR* u *RGB*
`cv::COLOR_RGB2RGBA`	Dodavanje *alpha*[2] kanala prikazu slike
`cv::COLOR_RGB2GRAY`	Pretvaranje *RGB* u akromatsku[3] sliku
`cv::COLOR_RGB2HLS`	Pretvaranje iz *RGB* u *HLS* prikaz slike

Tablica 2.8: Vrijednosti argumenta *code*

[1] Standardni skup rukom pisanih znamenaka, dostupan na `http://yann.lecun.com/exdb/mnist/`

Dekompozicija singularnih vrijednosti

Budući da *SVD*[4] ima primjenu u području računalnog vida - vrlo dobri rezultati mogu se postići za npr. problem prepoznavanja rukom pisanih znamenaka, korisna je *OpenCV* funkcija za izračunavanje te dekompozicije.

```
void SVD::compute(
   cv::InputArray A, // Linear system, array to be decomposed
   cv::OutputArray W, // Output array 'W', singular values
   cv::OutputArray U, // Output array 'U', left singular vectors
   cv::OutputArray Vt, // Output array 'Vt', right singular vectors
   int flags = 0 // what to construct, and if A can be scratch
);
```

Matematički, teorem 2.3.1. daje pojašnjenje *SVD-a*.

Teorem 2.3.1. *Neka je* $A \in \mathbb{R}^{m\times n}$ *proizvoljna matrica. Tada postoje matrice* $\hat{U}, \Sigma, V$ *takve da*

$$A = \widehat{U}\begin{bmatrix}\Sigma\\0\end{bmatrix}V^* = U\Sigma V^*,$$

gdje je $\Sigma = diag(\sigma_1, ..., \sigma_n)$, *uz* $\sigma_1 \geq ... \geq \sigma_n \geq 0$, $\widehat{U} = \begin{bmatrix}U, & U_0\end{bmatrix}$ *je unitarna*[5] *matrica reda m, a V je unitarna matrica reda n.*

Preslikavanje preko osi

U *OpenCV-u* postoji funkcija za preslikavanje slike preko x i y osi.

```
void cv::flip(
   cv::InputArray src,
   cv::OutputArray dst, // Result array, size and type of 'src'
   int flipCode = 0 // >0: y-flip, 0: x-flip, <0: both
);
```

Standardna devijacija piksela

```
void cv::meanStdDev(
   cv::InputArray src,
   cv::OutputArray mean,
   cv::OutputArray stddev,
```

[4] Singular value decomposition ili Dekompozicija singularnih vrijednosti

[5] Matrica U je unitarna ako vrijedi $U^*U = UU^* = \mathrm{I}$

```
   cv::InputArray mask = cv::noArray(), // Optional, do only where nonzero
);
```

`cv::meanStdDev()` služi za izračuvanje prosječne vrijednosti elemenata ulaznog polja te njihovu standardnu devijaciju. Ako se ulazno polje sastoji od više *kanala*, rezultati su izračunati pojedinačno za svaki od njih.

Funkcionalnosti crtanja

Ukratko ćemo opisati načine vizualizacije slika i geometrijskih oblika kao što su četverokuti, krugovi, linije.

Linija

```
void line(
   cv::Mat& img, // Image to be drawn on
   cv::Point pt1, // First endpoint of line
   cv::Point pt2 // Second endpoint of line
   const cv::Scalar& color, // Color, BGR form
   int lineType = 8, // Connectedness, 4 or 8
   int shift = 0 // Bits of radius to treat as fraction
);
```

Gornja funkcija služi za crtanje linije zadanom bojom `color` od točke `pt1` do `pt2`.

Pravokutnik

U nastavku je dana signatura funkcije za crtanje pravokutnika. Kao i sve ostale funkcije za crtanje, prima početnu sliku na koju dodajemo pravokutnik.

```
void rectangle(
   cv::Mat& img, // Image to be drawn on
   cv::Point pt1, // First corner of rectangle
   cv::Point pt2 // Opposite corner of rectangle
   const cv::Scalar& color, // Color, BGR form
   int lineType = 8, // Connectedness, 4 or 8
   int shift = 0 // Bits of radius to treat as fraction
);
```

Krug i elipsa

```
void circle(
   cv::Mat& img, // Image to be drawn on
   cv::Point center, // Location of circle center
   int radius, // Radius of circle
   const cv::Scalar& color, // Color, RGB form
   int thickness = 1, // Thickness of line
   int lineType = 8, // Connectedness, 4 or 8
   int shift = 0
);
```

```
bool ellipse(
   cv::Mat& img, // Image to be drawn on
   cv::Point center, // Location of ellipse center
   cv::Size axes, // Length of major and minor axes
   double angle, // Tilt angle of major axis
   double startAngle, // Start angle for arc drawing
   double endAngle, // End angle for arc drawing
   const cv::Scalar& color, // Color, BGR form
   int thickness = 1, // Thickness of line
   int lineType = 8, // Connectedness, 4 or 8
   int shift = 0 // Bits of radius to treat as fraction
);
```

Funkcija `cv::ellipse()` vrlo je slična `cv::circle()`. Razlika je u argumentima `axes` te `angle`, `startAngle` i `endAngle`. `axes` je tipa `cv::Size`, kojemu su članske varijable `width` i `height` jednake duljinama velike i male osi elipse. Argument `angle` označava nagib elipse, mjereno od velike osi, u smjeru kazaljke na satu. Ova funkcija omogućava crtanje luka koji je dio elipse, što zadajemo argumentima `startAngle` i `endAngle`. Za crtanje cijele elipse, potrebno ih je postaviti na 0 i 360.

Sada prelazimo na kompleksnije operacije za obradu slike.

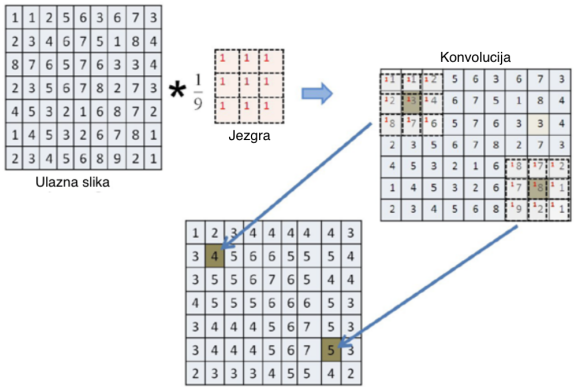
Vrste i korištenje filtara

Funkcije koje ćemo u nastavku opisati temelje se na konceptu *filtara slike*. Općenito, *filter* je algoritam koji kao ulaz prima sliku I i izračunava sliku I', na način da za piksel na koordinati (x, y) vrijednost funkcije ovisi o vrijednosti piksela u nekoj okolini (x, y) na slici I. Matricu koja određuje veličinu i sadržaj te okoline nazivamo *maskom* ili *jezgrom*. Mnoge

maske u primjeni su linearne, tj. možemo ih prikazati izrazom

$$I^{'}(x,y) = \sum_{i,j \in jezgra} k_{i,j} I(x+i, y+j) \tag{2.1}$$

gdje su $k_{i,j}$ elementi jezgre.



Slika 2.2: Djelovanje filtera na sliku

Na slici 2.2 vizualizirana je ideja korištenja filtera. Mogli bismo reći da "pomičemo" filter po ulaznoj slici - taj proces nazivamo *konvolucijom*.

Thresholding

Ponekad je potrebno na slici odbaciti piksele čija je vrijednost u određenom intervalu, a zadržati ostale. Operaciju kojom to postižemo nazivamo *thresholding*. Mogli bismo je poistovjetiti s filtrom čija je veličina maske 1×1.

```
double cv::threshold(
   cv::InputArray src, // Input image
   cv::OutputArray dst, // Result image
   double thresh, // Threshold value
   double maxValue, // Max value for upward operations
   int thresholdType // Threshold type to use
);
```

U tablici 2.9. su pojašnjena značenja argumenata funkcije `cv::threshold()`. Sa `src` je označena polazna slika, dok `dst` označava sliku dobivenu *thresholdingom*.

Osim konkretne vrijednosti *thresh* $\in \mathbb{R}$, kao argument `thresh` moguće je proslijediti i konstantu `CV::THRESH_OTSU`. U tom slučaju, pomoću *Otsuovog* algoritma određuje se optimalan prag (threshold).

Tip *thresholda*	**Značenje**
`CV::THRESH_BINARY`	$dst(i,j) = (src(i,j) > thresh)\ ?\ maxValue : 0$
`CV::THRESH_BINARY_INV`	$dst(i,j) = (src(i,j) > thresh)\ ?\ 0 : maxValue$
`CV::THRESH_TRUNC`	$dst(i,j) = (src(i,j) > thresh)\ ?\ thresh : src(i,j)$
`CV::THRESH_TOZERO`	$dst(i,j) = (src(i,j) > thresh)\ ?\ src(i,j) : 0$
`CV::THRESH_TOZERO_INV`	$dst(i,j) = (src(i,j) > thresh)\ ?\ 0 : src(i,j)$

Tablica 2.9: *Vrste thresholdinga*

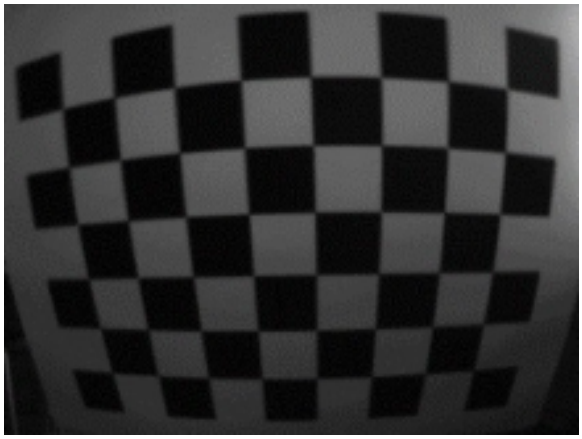
Otsuov algoritam funkcionira na način da minimizira varijance skupova piksela s vrijednošću iznad i ispod praga, tj.

$$\sigma^2 = w_1(t)\sigma_1^2 + w_2(t)\sigma_2^2 \rightarrow min \tag{2.2}$$

Ovdje su $w_1(t)$ i $w_2(t)$ relativne težine skupova, određene na temelju broja piksela koji pripadaju svakom od njih.

Čak i u slučaju korištenja nekog algoritma za određivanje optimalnog praga, još uvijek korištenjem funkcije `cv::threshold()` dobivamo sliku kod koje je jedna vrijednost praga primjenjena na sve piksele.

Ponekad je korisno za svaki od piksela lokalno odrediti optimalni prag. Metodu kojom se to postiže nazivamo *prilagodljiv (adaptivni) thresholding*. Na slikama 2.3 i 2.4 dan je motivacijski primjer. Vidimo da korištenjem konstantnog praga na cijeloj slici gubimo svjetliji dio slike, dok je korištenjem adaptivne verzije dobiven puno bolji rezultat.



Slika 2.3: Originalna slika

(a) Slika dobivena binarnim *thresholdingom*

(b) Slika dobivena adaptivnim binarnim *thresholdingom*

Slika 2.4: Slike dobivene *thresholdingom*

```
void cv::adaptiveThreshold(
   cv::InputArray src, // Input image
   cv::OutputArray dst, // Result image
   double maxValue, // Max value for upward operations
   int adaptiveMethod, // mean or Gaussian
   int thresholdType // Threshold type to use
   int blockSize, // Block size
   double C // Constant
);
```

Omogućeno je korištenje dvije metode za izračunavanje pragova, definiranih konstantama `CV::ADAPTIVE_THRESH_MEAN_C` i `CV::ADAPTIVE_THRESH_GAUSSIAN_C`. U slučaju prve konstante, vrijednost rezultantnog piksela $I'(i, j)$ je prosječna vrijednost piksela u pravokutnoj okolini $I(i, j)$. Za `CV::ADAPTIVE_THRESH_GAUSSIAN_C` vrijednost piksela je težinska suma okolnih piksela, gdje su težine vrijednosti funkcije

$$G(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.3}$$

Ovdje su x i y udaljenosti piksela od središta jezgre.

Zaglađivanje slika

Zaglađivanje (*smoothing*, *blurring*) vrlo je često korištena operacija u obradi slike. Najčešća je svrha zaglađivanja smanjenje šumova (*noise*) na slici. Također, korisna je i kod smanjenja rezolucije slike. U biblioteci OpenCV implementirano je nekoliko različitih vrsta zaglađivanja.

Jednostavno zaglađivanje (*Simple blur*)

Svaki piksel u rezultantnoj slici je prosječna vrijednost okolnih piksela. Veličina jezgre je zadana argumentom `ksize`. Ovaj filter je specijalni slučaj tzv. *box filtera*. *Box filter* je svaki filter kojemu su vrijednosti svih elemenata jezgre jednake. Kod *jednostavnog zaglađivanja* vrijednosti jezgre su normalizirane.

Medijan filter

Svaki piksel je kod ovog filtra zamijenjen s *medijanom* pravokutne okoline tog piksela. Za razliku od jednostavnog zaglađivanja, kod kojeg se koristi prosječna vrijednost, korištenje medijana eliminira utjecaj velikih odstupanja (eng. *outliers*).

Gaussov filter

Gaussov filter je najčešće korišten u primjeni. Sastoji se od konvolucije svakog elementa matrice (slike) s normaliziranom *Gaussovom* maskom, definiranom izrazom (2.3). Matricom (2.4) dan je primjer *Gaussove* maske prije normalizacije, dimenzije 5×5.

$$\begin{bmatrix} 2 & 7 & 12 & 7 & 2 \\ 7 & 31 & 52 & 31 & 7 \\ 12 & 52 & 127 & 52 & 12 \\ 7 & 31 & 52 & 31 & 7 \\ 2 & 7 & 12 & 7 & 2 \end{bmatrix} \tag{2.4}$$

Definicija *OpenCV* funkcije za *Gaussov* filter je sljedeća:

```
void cv::GaussianBlur(
   cv::InputArray src, // Input image
   cv::OutputArray dst, // Result image
   cv::Size ksize, // Kernel size
   double sigmaX, // Gaussian half-width in x-direction
   double sigmaY = 0.0, // Gaussian half-width in y-direction
   int borderType = cv::BORDER_DEFAULT
);
```

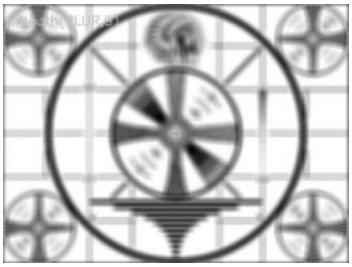
Implementacija je optimizirana za maske veličine 3×3, 5×5 i 7×7.

Bilateralni filter

Ovaj filter također koristi Gaussove maske, na sljedeći način - prva je komponenta identična Gaussovom filtru, dok je druga maska također Gaussova, no vrijednosti argumenata x i y iz formule (2.3) nisu udaljenost od centralnog piksela, već razlika u intenzitetu u odnosu

na taj piksel. Bilateralni filter pomaže u očuvanju rubova na slici.

Slika 2.5: Originalna slika



(a) Slika dobivena Jednostavnim zaglađivanjem

(b) Slika dobivena Medijan filtrom

(c) Slika dobivena Gaussovim filtrom

(d) Slika dobivena Bilateralnim filtrom

Morfološka obrada slike

Termin *matematička morfologija* označava sredstvo za ekstrakciju komponenti slike korisnih za reprezentaciju oblika regija kao što su rubovi i konveksna područja. Na morfologiji se bazira i analiza pokreta na slici. Morfološke operacije najčešće opisuju nelinearne operacije na slici, što se postiže korištenjem nelinearnih maski - *strukturnih elemenata*. Osnovne operacije morfološke obrade slike su *dilatacija* i *erozija*.

Dilatacija

Definicija 2.3.2. *Neka je A binarna slika te B strukturni element. Tada je dilatacija slike A elementom B definirana izrazom*

$$A \oplus B = \bigcup_{b \in B} (A + b) \tag{2.5}$$

gdje

$$A + b = \{a + b \mid a \in A\} \tag{2.6}$$

Dakle, *dilatacija* je operacija kojom "podebljavamo" objekte na slici. Način i količina povećanja objekta kontrolirani su pripadnim *strukturnim elementom*. Dilatacija je asocijativna i komutativna operacija. Često je korištena kako bismo povezali bliske komponente na slici (npr. istanjeni rubovi) [9].

Erozija

Definicija 2.3.3. *Neka je A binarna slika te B strukturni element. Tada je erozija slike A elementom B definirana izrazom*

$$A \ominus B = \bigcap_{b \in B} (A - b) \tag{2.7}$$

gdje

$$A - b = \{a - b \mid a \in A\} \tag{2.8}$$

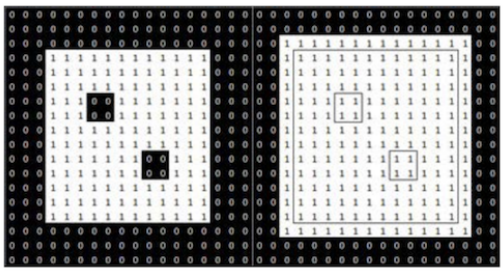
Drugim riječima, *erozijom* smanjujemo objekte na slici. Kao i kod dilatacije, količinu smanjenja kontroliramo odabirom strukturnog elementa. Često koristimo eroziju kako bismo eliminirali šum sa slike.

U *OpenCV-u*, ove su dvije operacije implementirane kroz funkcije `cv::erode()` i `cv::dilate()`.

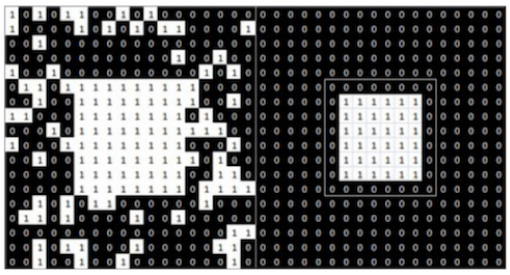
```
void cv::erode(
  cv::InputArray src, // Input image
```

```
  cv::OutputArray dst, // Result image
  cv::InputArray element, // Structuring, a cv::Mat()
  cv::Point anchor = cv::Point(-1,-1), // Location of anchor point
  int iterations = 1, // Number of times to apply
  int borderType = cv::BORDER_CONSTANT // Border extrapolation
  const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()
);
```

```
.void cv::dilate(
  cv::InputArray src, // Input image
  cv::OutputArray dst, // Result image
  cv::InputArray element, // Structuring, a cv::Mat()
  cv::Point anchor = cv::Point(-1,-1), // Location of anchor point
  int iterations = 1, // Number of times to apply
  int borderType = cv::BORDER_CONSTANT // Border extrapolation
  const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()
);
```



(a) Binarna slika prije i poslije primjene dilatacije

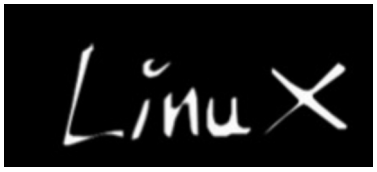


(b) Binarna slika pije i poslije primjene erozije

Slika 2.8: Djelovanje erozije i dilatacije na sliku korištenjem strukturnog elementa veličine 3×3

(a) Originalna slika

(b) Slika dobivena dilatacijom



(c) Slika dobivena erozijom

Kao što je spomenuto ranije, ostale morfološke operacije dobivamo kompozicijom *erozije* i *dilatacije*. U nastavku definiramo morfološko otvaranje i zatvaranje.

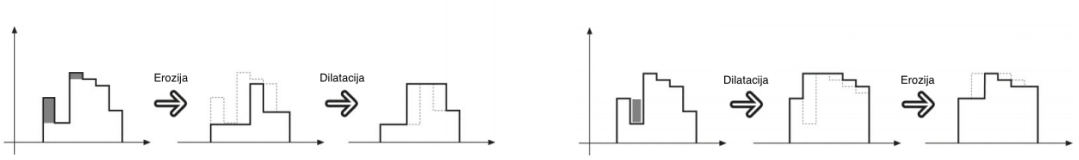
Definicija 2.3.4. *Morfološko otvaranje slike A strukturnim elementom B, u oznaci* $A \circ B$*, definiramo kao*

$$A \circ B = (A \ominus B) \oplus B \tag{2.9}$$

Definicija 2.3.5. *Morfološko zatvaranje slike A strukturnim elementom B, u oznaci* $A \bullet B$*, definiramo kao*

$$A \bullet B = (A \oplus B) \ominus B \tag{2.10}$$

Intuitivno, otvaranje u prvom koraku eliminira elemente slike manje od strukturnog elementa, što izglađuje rubove objekta, te nakon toga, koristeći *dilataciju* poveća površinu objekta kako bi otprilike odgovarala stvarnom objektu. S druge strane, *zatvaranje* popunjava praznine manje od strukturnog elementa te nakon toga erozijom uklanja manje skupine piksela.



(a) Djelovanje *otvaranja* na sliku (b) Djelovanje *zatvaranja* na sliku

Transformacije slike

U nastavku opisujemo neke od općenitih transformacija slike, kao što su promjena veličine i rotacija.

Promjena veličine

Promjena veličine slike u biblioteci *OpenCV* podržana je kroz funkciju `cv::resize()`. Jedan od parametara koje funkcija prima odnosi se na metodu *interpolacije*. U nastavku su dane pojašnjene vrijednosti.

- `CV::INTER_NEAREST`
  Najjednostavnija metoda interpolacije. Vrijednost danog piksela jednaka je vrijednosti najbližeg susjednog piksela.
- `CV::INTER_LINEAR`
  Vrijednost piksela je težinska suma vrijednosti iz 2×2 okoline tog piksela.

- `CV::INTER_AREA`
  U ovom slučaju vrijednost piksela jednaka je srednjoj vrijednosti okolnih piksela koji su prekriveni novim pikselom nakon promjene veličine slike.
- `CV::INTER_CUBIC`
  Interpolacija korištenjem kubičnih *splineova* na okolnih 4×4 piksela.
- `CV::INTER_LANCZOS4`
  Metoda je slična onoj definiranoj s `CV::INTER_CUBIC`, uz korištenje informacije s 8×8 mreže oko danog piksela.

Neuniformne promjene veličine

Funkcije koje služe za rotaciju, promjenu veličine i izobličenje slika nazivamo *geometrijskim transformacijama*. Transformacije koje koriste 2×3 matricu kao matricu transformacije nazivamo *afinim transformacijama*, dok one koje koriste matricu 3×3 zovemo *transformacije perspektive* ili *homografija* [7].

Homografija je fleksibilnija od afine transformacije u toliko što je njome moguće pravokutnik transformirati u proizvoljan četverokut, dok afinom transformacijom možemo transformirati pravokutnike u paralelograme, odnosno paralelnost nasuprotnih stranica ostaje očuvana.

Afine transformacije

U *OpenCV-u* postoje dvije implementacije ove operacije. U prvom slučaju, dan je dio slike kojeg je potrebno transformirati. U drugoj implementaciji, funkcija prima listu točaka.

Prva implementacija dana je kroz funkciju `cv::warpAffine()`. Ako je A početna slika, a s M zadana ranije spomenuta 2×3 matrica transformacije, tada za rezultantnu sliku B vrijedi

$$B(x, y) = A(M_{00}x + M_{01}y + M_{02}, M_{10}x + M_{11}y + M_{12}) \tag{2.11}$$

Općenito, indeksi matrice A u izrazu (2.11) nisu nužno cijeli brojevi - tada je potrebno nekom od metoda interpolacije pronaći odgovarajuću vrijednost piksela.

Sada nam je još potreban način izračunavanja matrice transformacije M. U tu svrhu, postoji funkcija `cv::getAffineTransform()`.

```
cv::Mat cv::getAffineTransform( // Return 2-by-3 matrix
  const cv::Point2f* src, // Coordinates of three of vertices
  const cv::Point2f* dst // Target coords, three vertices
);
```

Sada još spominjemo implementaciju afine transformacije s listom točaka kao ulazom. U tu svrhu postoji funkcija `cv::transform()`.

Homografija

Ponovno, kao i kod *afine transformacije*, imamo dvije različite implementacije - jednu za *dense* slučaj (ulazni podatak je slika) te drugu za *sparse* (ulazni podatak je lista točaka - vrhova).

Za *dense* slučaj koristimo funkciju `cv::warpPerspective()`. Ovdje elemente rezultantne matrice B računamo iz matrice A pomoću matrice transformacije M po sljedećoj formuli:

$$B(x, y) = \left(\frac{M_{00}x + M_{01}y + M_{02}}{M_{20}x + M_{21}y + M_{22}}, \frac{M_{10}x + M_{11}y + M_{12}}{M_{20}x + M_{21}y + M_{22}} \right) \tag{2.12}$$

Za izračunavanje matrice transformacije M, koristimo `cv::getPerspectiveTransform()`.

Detekcija rubova i kutova

Rubovi i kutovi neki su od osnovnih značajki slike te se algoritmi za njihovu detekciju u primjeni koriste vrlo često. U nastavku opisujemo najpoznatije algoritme i njihove implementacije u sklopu *OpenCV-a*.

Cannyjev detektor rubova

Cannyjev detektor rubova jedan je od najpopularnijih algoritama za tu svrhu. Algoritam se sastoji od 4 osnovna koraka:

1. **Zaglađivanje slike**
   Korištenjem *Gaussovog* filtra eliminiramo dio šuma sa slike.

2. **Računanje gradijenata slike**
   Sljedeće, računamo gradijente slike te ih dijelimo na vertikalne, horizontalne i dijagonalne.

3. ***Non-maximal suppression***
   Sada koristimo izračunate gradijente da bismo provjerili je li neki piksel lokalni maksimum u smjeru gradijenta. Ako nije, tada na njemu primjenjujemo *suppression* na tom pikselu - taj piksel nije dio ruba.

4. **Odabir rubova korištenjem *thresholdinga***
   Nakon koraka 3. dobili smo skup točaka koje čine rubove. Sada odabiremo najizraženije rubove, koji postaju krajnji izlaz algoritma.

```
void cv::Canny(
   cv::InputArray image, // Input single channel image
   cv::OutputArray edges, // Output edge image
   double threshold1, // "lower" threshold
   double threshold2, // "upper" threshold
   int apertureSize = 3, // Sobel aperture
   bool L2gradient = false // true=L2-norm (more accurate)
);
```

Bitno je primjetiti da `cv::Canny()` kao parametre prima dva praga, `threshold1` i `threshold2`. Oni se koriste kako bismo klasificirali točke u tri klase - ugušene (*suppressed*) točke, točke slabih rubova i točke jakih rubova. Sve točke s vrijednošću manjom od `threshold1` su klasificirane kao ugušene, one između `threshold1` i `threshold2` smatramo točkama slabih rubova, dok ostale smatramo točkama jakih rubova.

Cannyjev detektor kod konstruiranja rezultata ignorira sve ugušene točke. Točke jakih rubova tvore rubove; za one koje su dio slabih rubova, provjeravamo jesu li povezane s nekim točkama jakih rubova. Ako je odgovor potvrdan, tu točku također smatramo dijelom ruba.

Sobelov operator

Sobelov operator još je jedna tehnika za detekciju rubova. Slično kao kod *Cannyja*, koristi gradijent piksela, no izračunavamo ga na specifičan način - računamo približnu vrijednost gradijenta *konvolucijom* dvije 3×3 maske, svake u jednom smjeru.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.13}$$

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.14}$$

Matrica (2.13) prikazuje filter u x smjeru, (2.14) u y smjeru. Korištenjem tako izračunatih gradijenata G_x i G_y, sada računamo apsolutnu vrijednost gradijenta:

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{2.15}$$

Za aproksimaciju se često koristi i formula

$$|G| = |G_x| + |G_y| \tag{2.16}$$

Koraci *Sobelovog* operatora su:

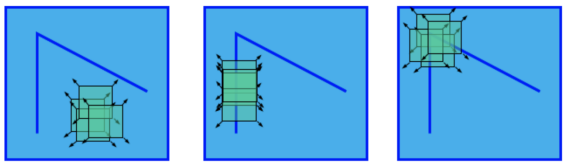
1. Pretvorimo sliku u akromatsku;
2. Izračunamo apsolutnu vrijednost gradijenta u smjeru x i y osi;
3. Izračunamo rezultantni gradijent, koristeći formulu (2.15) ili (2.16) - vrijednosti gradijenta veće od zadanog praga određuju rubove.

Harrisov detektor kuteva

Intuitivno, kutevi su točke u kojima se sjeku dva ruba. Kutevi su često zanimljivi u slici te se koriste u mnogim primjenama kao što je stabilizacija slike ili videa.

Harrisov detektor koristi tzv. *sliding window* (pomični prozor), kako bi izračunao varijacije u vrijednosti piksela. Budući da tipično očekujemo velike varijacije u kutevima, tražimo koordinate u slici gdje *sliding window* pokazuje takav uzorak. Drugim riječima, maksimiziramo izraz

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2 \tag{2.17}$$



Slika 2.11: Ideja *sliding window* koncepta

Na slici 2.11 prikazana je ideja *sliding window-a*. Zelenom bojom prikazani su prozori s različitim pomacima u i v. Ovdje je u horizontalni, a v vertikalni pomak prozora.
E označava razliku u vrijednosti između polaznog prozora (prozora za koji vrijedi $u = 0$, $v = 0$) i pomaknutog prozora. S $w(x,y)$ je označena vrijednost prozora za zadani (x,y) - w je zapravo maska (npr. Gaussova, s vrijednostima kao u izrazu (2.3)). Dodatno, vrijednosti izvan prozora bit će postavljene na 0, tj. biti ignorirane.

Iz razvoja funkcije $E(x,y)$ u Taylorov red dobivamo

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} \tag{2.18}$$

gdje

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_y I_x & I_y I_y \end{bmatrix} \tag{2.19}$$

Ovdje su I_x i I_y parcijalne derivacije od I.

Znamo, za λ_1, λ_2 svojstvene vrijednosti matrice M, vrijedi[6]

$$detM = \lambda_1 \cdot \lambda_2, \quad trM = \lambda_1 + \lambda_2 \tag{2.20}$$

Harrisov detektor sada koristi izraz

$$R = detM - k \cdot (trM)^2 \tag{2.21}$$

za određivanje kuteva. Ovdje je k eksperimentalno određena konstanta.

Za funkciju R vrijedi:

- R poprima velike vrijednosti za kuteve;
- R poprima negativne vrijednosti velikih apsolutnih vrijednosti za rubove;
- $|R|$ je mala za dijelove slike koji nisu kutevi ni rubovi.

Na temelju odabrane vrijednosti praga na vrijednost funkcije R određujemo kuteve na slici.

Houghove transformacije

Ponekad, uz rubove i kuteve, želimo prepoznati i oblike kao što su krugovi, elipse ili linije. Na primjer, recimo da želimo prepoznati loptu na slici - možemo svoju pretragu bez previše rizika svesti samo na traženje elipsi ili krugova. Upravo su *Houghove* transformacije tehnika koja se često koristi u takvim situacijama. U teoriji, ovu je tehniku moguće koristiti za detekciju bilo kojeg oblika za kojeg možemo izvesti parametiziranu jednadžbu.

Houghove transformacije za pravce

Neka je I neka slika. Odabiremo par točaka (x_1, y_1) i (x_2, y_2) sa I i rješavamo sustav jednadžbi po (a, m).

$$y_1 = m \cdot x_1 + a, \quad y_2 = m \cdot x_2 + a \tag{2.22}$$

Postupak ponovimo za sve parove (x_1, y_1), (x_2, y_2) na slici i za svaki par (a, m) pamtimo broj parova točaka za koje su rješenje sustava te vrijednosti. Zapravo, algoritam funkcionira kao "glasovanje" za vrijednosti a i m. Nakon što smo dobili pripadne vrijednosti za svaki par točaka sa slike, na temelju zadanog *thresholda* odlučujemo koji će od pravaca $y = mx + a$ biti smatrani linijom na slici. Složenost osnovne verzije algoritma je $O(m^2n^2)$, gdje je $m \times n$ dimenzija slike I.

U *OpenCV-u* postoje dvije implementacije *Houghove* transformacije za pravce.

[6] S trM označavamo trag matrice M

```
void cv::HoughLines(
   cv::InputArray image, // Input single channel image
   cv::OutputArray lines, // N-by-1 two-channel array
   double rho, // rho resolution (pixels)
   double theta, // theta resolution (radians)
   int threshold, // Unnormalized accumulator threshold
   double srn = 0, // rho refinement (for MHT)
   double stn = 0 // theta refinement (for MHT)
);
```

```
void cv::HoughLinesP(
   cv::InputArray image, // Input single channel image
   cv::OutputArray lines, // N-by-1 4-channel array
   double rho, // rho resolution (pixels)
   double theta, // theta resolution (radians)
   int threshold, // Unnormalized accumulator threshold
   double minLineLength = 0, // required line length
   double maxLineGap = 0 // required line separation
);
```

U odnosu na `cv::HoughLines`, `cv::HoughLinesP` se razlikuje u parametrima `minLineLength` i `maxLineGap`. `minLineLength` određuje minimalnu duljine linije koja može biti uzeta u obzir, a `maxLineGap` najveću dopuštenu duljinu između dijelova linije. Razlika je također u tome što `cv::HoughLinesP` funkcija implementira tzv. probabilističke *Houghove* transformacije - umjesto svih piksela slike, algoritam koristi samo određeni broj piksela za odlučivanje, što u praksi rezultira ubrzanjem algoritma.
Primjetimo da *OpenCV* implementacija koristi parametrizaciju pravca u polarnim koordinatima.

U primjeni, često prije korištenja *Houghove* transformacije, detektiramo rubove na slici nekim od algoritama te dobivenu sliku prosijeđujemo rutini koja implementira računanje *Houghovih* transformacija.

Houghove transformacije za kružnice

Primjena *Houghovih* transformacija za kružnice provodi se na isti način, uz zamjenu jednadžbe pravca parametriziranom jednadžbom kružnice.

Konture

Često problemi računalnog vida iziskuju od nas da podijelimo sliku na semantičke cjeline kako bismo imali bolji fokus na ciljni objekt. Na primjer, u poglavlju 4 implementirat ćemo rješavač Sudokua i bit će nam bitno iz slike na kojoj je mreža Sudokua izdvojiti samu mrežu.

Jedan način da to postignemo je korištenjem *Houghovih* transformacija - npr. detektiramo linije te nakon toga nekim od ranije spomenutih algoritama prepoznamo kuteve.

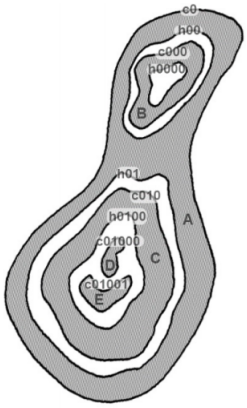
Konture su ništa drugo no granice povezanih komponenti u slici. Često se pronalaze korištenjem detektiranih rubova. Razlika između rubova i kontura je u tome da su konture zatvorene krivulje, dok na rubove ne postoji taj zahtjev.

U *OpenCV-u* prepoznavanje kontura realizirano je kroz funkciju `cv::findContours()`. Implementacija je inspirirana člankom [12].

```
void cv::findContours(
   cv::InputOutputArray image, // Input "binary" 8-bit single channel
   cv::OutputArrayOfArrays contours, // Vector of vectors or points
   int mode, // Contour retrieval mode
   int method, // Approximation method
   cv::Point offset = cv::Point() // (optional) Offset every point
);
```

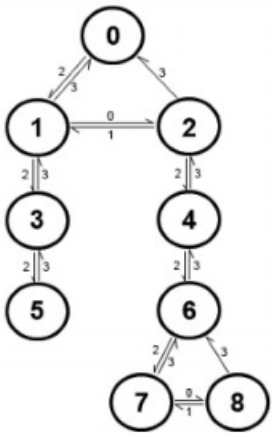
Prvi argument funkcije je ulazna slika. Funkcija zahtjeva da se ona sastoji od samo jednog kanala te da su joj elementi 8-bitni. Važno je spomenuti da `cv::findContours()` mijenja ulaznu sliku. Argument `contours` nakon poziva funkcije sadržavat će detektirane konture - `contours[i]` reprezentira pojedinu konturu, dok s `contours[i][j]` možemo dohvatiti j-ti vrh i-te konture. Parametrom `mode` određujemo način na koji želimo dohvatiti konture. Proslijeđena vrijednost je jedna od predefiniranih konstanti.

- `CV::RETR_EXTERNAL`
  Dohvaćene su samo vanjske konture. Na slici 2.12 to bi obuhvaćalo samo konturu c_0.
- `CV::RETR_LIST`
  Dohvaćene su sve konture u listi.
- `CV::RETR_CCOMP`
  Dohvaćene su sve konture organizirane u 2 nivoa hijerarhije. Prvi nivo sadrži sve vanjske granice komponenti, a drugi nivo sadrži granice tzv. *rupa*. Na slici 2.11., to su konture označene početnim slovom h.



Slika 2.12: Primjer *kontura*

- `CV::RETR_TREE`
  Dohvaćene su sve konture i njihova potpuna hijerarihija. Rezultat za sliku 2.12 prikazan je na slici 2.13.



Slika 2.13: Primjer *kontura*

Jednom kad smo dohvatili konture na slici, možemo ih vizualizirati koristeći funkciju `cv::drawContours()`.

Strojno učenje u *OpenCV-u*

Na kraju ovog poglavlja, ukratko opisujemo MLL (*Machine learning modul*) u sklopu biblioteke *OpenCV*.

Ideja strojnog učenja je pretvoriti podatke u neku smislenu i korisnu informaciju. Podatci općenito mogu biti bilo kojeg oblika, a MLL je, kao i cijela biblioteka, fokusirana na slike.

MLL obuhvaća raznolike algoritme strojnog učenja. Neki od najčešće korištenih su:

- Algoritam K-najbližih susjeda;
- SVM (*Support Vector Machines*);
- Algoritam K-sredina;
- *Bayesov* klasifikator;
- Stabla odlučivanja;
- *Boosting*;
- Kaskadni klasifikator;
- Koncept *Vreća riječi* (Bag of Words).

Biblioteka *OpenCV*, uz ovdje opisane, sadrži još mnogo korisnih funkcija, koje su vrlo kvalitetno dokumentirane što omogućava njihovo jednostavno korištenje.

Poglavlje 3

Integracija OpenCV-a s platformom Android

3.1 Platforma Android

Uvod

Android je operacijski sustav za mobilne uređaje, razvijan od strane tvrtke *Google*. Temeljen je na izmjenjenoj verziji *Linux* jezgre te softveru otvorenog koda. Prvotna primjena je bila korištenje u pametnim telefonima i tabletima, no razvijene su i inačice *Android TV* (integracija s televizijom), *Android Auto* (integracija s automobilima) te *Android Wear* (integracija sa satovima).

Android je najprodavaniji OS prilagođen pametnim telefonima od 2013., a 2017. je prešao 2 milijarde aktivinih korisnika mjesečno. Pripadajuća trgovina aplikacijama, *Google Play*, sadrži više od 3,500,000 aplikacija. Aktualna verzija *Androida* trenutno je 8.1 *Oreo*.

Slika 3.1: Android logo

Razvoj aplikacija

Aplikacije za operacijski sustav *Android* razvijaju se korištenjem tzv. SDK-a (*Android Software Development Kit*). Najčešće je za razvoj korišten programski jezik *Java*. Također, moguće je kombinirati C i C++, te u novije vrijeme *Go* i *Kotlin*.

SDK sadrži mnogo razvojnih alata, kao što su *debuggeri*, emulatori. Od 2014., za Android razvoj postoji i specijaliziran IDE[1], Android Studio.

3.2 OpenCV za Android

Motivacija

Algoritme računalnog vida u početku je bilo moguće koristiti na vrlo malom broju najekskluzivnijih računala. Kroz vrijeme i veću pristupačnost računalnih resursa širem krugu ljudi, to se područje vrlo brzo proširilo. Danas mnogi pametni telefoni i tableti dolaze s dvojezgrenim i četverojezgrenim procesorima, integriranim GPU-ovima, pa se i oni sve više koriste u svrhu računalnog vida. Razlog je tome i što su zahtjevi računalnog vida takvi da je u puno situacija puno praktičnije koristiti mobilne uređaje nego klasična računala. Neki od primjera su prepoznavanje lica (npr. *Instagram* i *Snapchat* koriste dodavanje različitih filtera kod fotografiranja lica), prepoznavanje teksta sa slike i slično.

OpenCV4Android službeni je naziv za Android port[2] za OpenCV. Podrška za Android prvi je puta predstavljena u OpenCV verziji 2.2. Inicijalna *beta* verzija sastojala se od programskog sučelja (API) u Javi te nativne (C/C++) podrške za upotrebu kamere telefona.

OpenCV programsko sučelje u Javi

Najčešće jednostavniji način korištenja je putem programskog sučelja u Javi. Na taj je način moguće koristiti gotovo sve originalne OpenCV funkcije. Svaka funkcija je "omotana" (eng. *wrapped*) u pripadno sučelje u Javi. I dalje, pozivana implementacija je ona nativna.

Mana ovog pristupa je činjenica da gubimo na preformansama aplikacije zbog dodatnog sloja u pozivu nativnih OpenCV funkcija. Naime, poziv nativne funkcije realiziran je preko *Java Native Interface* (JNI)[3]. Zbog toga, dodatne su operacije dodane u dvije različite situacije - kod početka svakog poziva nativnoj funkciji te nakon svakog poziva (tijekom izlaska iz funkcije). Ipak, za veliki dio aplikacija gubitak performansi je zanemariv. Na slici 3.2b prikazana je pojednostavljena struktura poziva Java API-ja.

[1] Integrated Development Environment

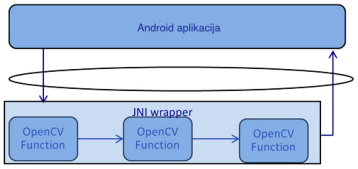
[2] Prilagodba za neki sustav, arhitekturu ili programski jezik

[3] JNI je mehanizam koji omogućava komunikaciju Java koda izvršavanog u JVM-u komunikaciju s nativnim aplikacijama

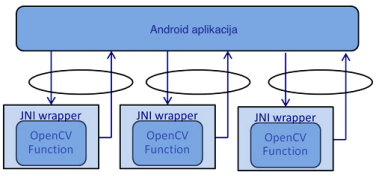
Nativne (C/C++) aplikacije

U slučaju da odlučimo implementirati svoje operacije u jeziku C ili C++, potrebno je koristiti Android NDK (*Native Development Kit*).

Uzmimo za primjer obradu jedne slike. Ako bismo koristili OpenCV programsko sučelje u Javi, svaka od funkcija iz biblioteke OpenCV koju pozivamo dodaje dodatne operacije. S druge strane, korištenjem nativnih funkcija, implementacija iziskuje samo jedan poziv JNI - potrebno je integrirati obradu slike u Android aplikaciju. Na slici 3.2a prikazana je struktura poziva.



(a) Struktura poziva - nativne OpenCV funkcije



(b) Struktura poziva - OpenCV Java API

Slika 3.2: Usporedba Java API i nativnih poziva

3.3 OpenCV Manager

OpenCV Manager je aplikacija koju je potrebno instalirati na Android uređaj kod pokretanja aplikacije koja koristi OpenCV funkcionalnosti. Moguće ju je dohvatiti s *Google Play* trgovine. Svrha te aplikacije je održavanje OpenCV biblioteka instaliranih na uređaj, ažuriranje njihovih verzija i odabir za korištenje one verzije koja je optimizirana za taj uređaj.

Najbolja je praksa dinamički povezati OpenCV biblioteku s aplikacijom. Drugim riječima, sama biblioteka ne bi trebala biti dio aplikacijske arhive, već dohvaćena kod izvršavanja aplikacije. Jedna od najbitnijih prednosti je da dinamičkim povezivanjem nismo ograničeni na jednu verziju aplikacije.

3.4 Sustav Tegra

Tegra je serija čipova (SoC - *System on a chip*) razvijena od strane tvrtke *NVIDIA*. Primjena im je u mobilnim uređajima kao što su pametni telefoni, tableti ili osobni asistenti. *Tegra* integrira ARM[4] arhitekturu procesora, GPU, *northbridge*, *southbridge* i kontrolor memorije.

[4] ARM je linija mikroprocesora koje razvija tvrtka ARM Holdings

Neki od korisnika *Tegre* su *Microsoft*, *Samsung*, *Audi* i *Tesla Motors*. Do danas postoji 7 generacija Tegra čipova:

- Tegra 2;
- Tegra 3;
- Tegra 4;
- Tegra K1;
- Tegra X1;
- Tegra X2;
- Xavier.

NVIDIA je bila jedan od bitnih čimbenika kod dovođenja OpenCV biblioteke na Android platformu, mnogim doprinosima *OpenCV4Android* portu otvorenog koda. NVIDIA je i kreator ranije spomenutog *OpenCV Manager* paketa.

3.5 Optimizacija za sustave Tegra

OpenCV for Tegra je verzija OpenCV-a za Android koji je optimiziran za Tegra platforme. Optimizacije rezultiraju ubrzanjem od nekoliko puta u odnosu na standardni *OpenCV4Android* paket. *OpenCV for Tegra* dio je TADP (Tegra Android Development Pack), paketa koji se sastoji od mnogih korisnih alata za razvoj Android aplikacija, kao što su *Apache Ant*, Tegra Profiler, Android SDK, Android NDK.

OpenCV for Tegra moguće je koristiti i na Android uređajima koji ne koriste Tegra sustav - umjesto verzija funkcija optimiziranih za tu arhitekturu koristit će se osnovna implementacija.

U sklopu OpenCV-a postoji i modul za rad na GPU-u, koji koristi implementaciju pomoću platforme *NVIDIA CUDA*. U sklopu tog modula implementirane su pomoćne funkcije, jednostavni i složeniji algoritmi. Za korištenje tog modula, nije potrebno imati znanje razvoja u *CUDA*-i, potrebno je samo odobriti korištenje grafičkih kartica prilikom kompilacije razvijenih aplikacija.

Poglavlje 4

Aplikacija za rješavanje Sudokua

4.1 Uvod

U ovom poglavlju dan je pregled aplikacije za rješavanje Sudokua razvijene u sklopu rada. Aplikacija pruža funkcionalnost fotografiranja proizvoljne tablice za igru Sudoku (npr. iz dnevnih novina) te njenog automatskog rješavanja. Razvijena je za platformu *Android* i za veći se dio operacija koriste funkcionalnosti ranije opisane biblioteke OpenCV.

U sljedećem odlomku opisujemo osnove igre te nakon toga dajemo detaljniji pregled aplikacije.

4.2 Osnove igre Sudoku

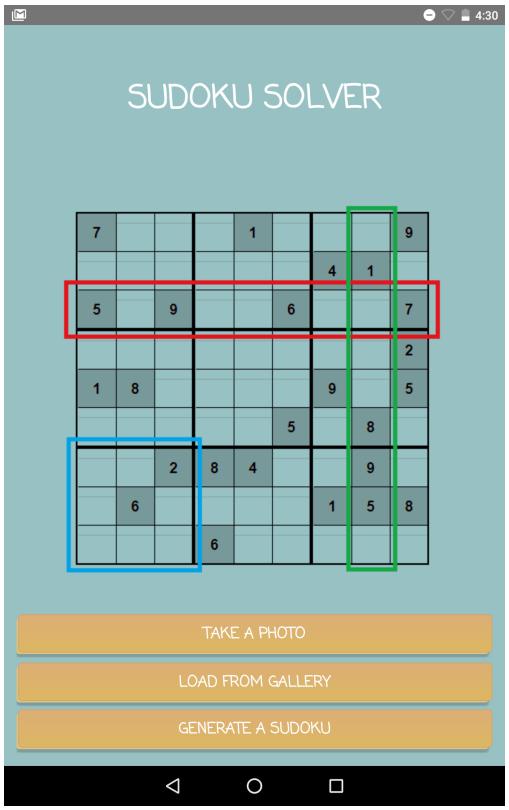
Sudoku je jedna od najpopularnijih logičkih igara. Cilj igre je popuniti 9×9 mrežu brojevima između 1 i 9, na način da se u svakom retku, stupcu i 3×3 mreži dobivenoj dijeljenjem početne mreže svaki broj nalazi točno jednom. Na početku igre, dio je mreže već popunjen brojevima, na način da je nedostajuće brojeve moguće upisati na jedinstveni način. Na slici 4.1a vidimo jednu moguću početnu konfiguraciju igre, prikazanu prilikom pokretanja aplikacije izrađene u sklopu ovog rada.

Postoje brojne strategije za rješavanje Sudokua, od intuitivnih ideja pa sve do kompliciranijih algoritama. Dokazano je da rješavanje Sudokua, kao i ostalih logičkih igara, poboljšava koncentraciju i pamćenje.

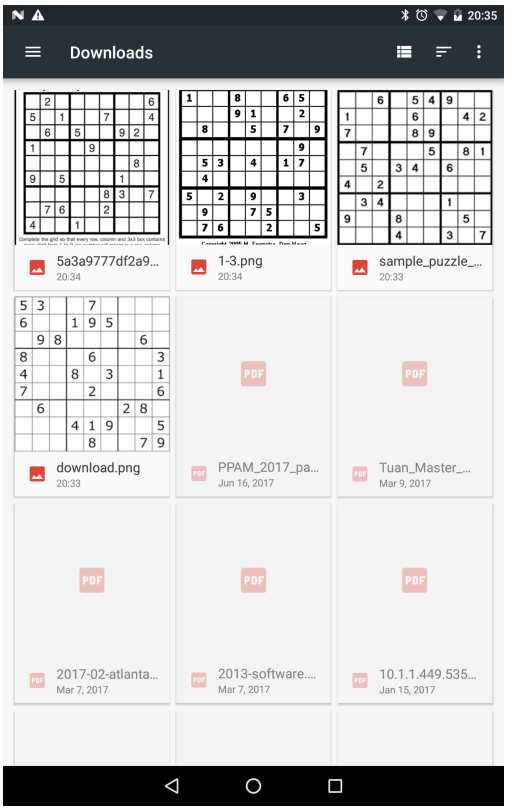
4.3 Općenito o aplikaciji

Aplikacija je zamišljena kao rješavač igre Sudoku. Primarna funkcionalnost sastoji se od fotografiranja *Sudoku*a sa slike, prepoznavanja već upisanih brojeva i njegovog rješavanja.

Uz to, aplikacija sadrži dodatne načine rada koje ukratko opisujemo u nastavku. Pojednostavljeni dijagram toka aplikacije je prikazan slikom 4.5.



(a) Početni ekran aplikacije



(b) Ekran za odabir dokumenta iz lokalnog spremnika

Slika 4.1: Primjeri ekrana iz aplikacije za rješavanje Sudokua

Odabir Sudokua

Na slici 4.1a prikazan je početni ekran aplikacije na kojem se nalazi nekoliko opcija za kreiranje Sudokua. Ukratko u nastavku objašnjavamo prikazane opcije.

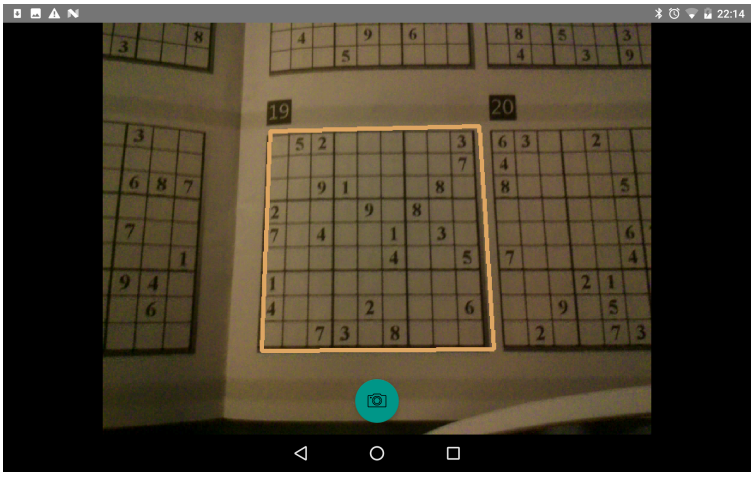
- Opcija **"Take a photo"**
  Korisnika se upućuje na ekran s kojeg može fotografirati Sudoku igru.
- Opcija **"Load from Gallery"**
  Pritiskom na ovu opciju, otvara se aplikacija za odabir slike iz lokalne memorije pametnog telefona ili tableta. Na slici 4.1b je prikazan primjer opisane situacije.

Primijetimo da je moguće odabrati samo slikovne dokumente. Tu je funkcionalnost vrlo lagano postići ugrađenim Android mehanizmima. Nakon što korisnik odabere željenu sliku, obrada je dalje analogna onoj nakon fotografiranja Sudokua.

- Opcija **"Generate a Sudoku"**
  Opcija generiranja Sudokua slučajnim odabirom. Realizirana je na način da inicijalno generiramo riješeni Sudoku te iz njega mičemo određeni broj vrijednosti K (K je zadan kao konstanta u aplikaciji), pazeći pritom da generirani Sudoku ima jedinstveno rješenje.

Fotografiranje igre Sudoku u aplikaciji

Nakon odabira gore spomenute opcije **Take a photo**, korisniku je prikazan ekran sa slike 4.2. U bilo kojem trenutku, uokvireni dio slike predstavlja trenutno detektiranu Sudoku mrežu. Nakon pritiska na tipku za fotografiranje otvara nam se ekran s prikazom fotografije koja će biti korištena za rekonstrukciju sadržaja Sudoku igre.

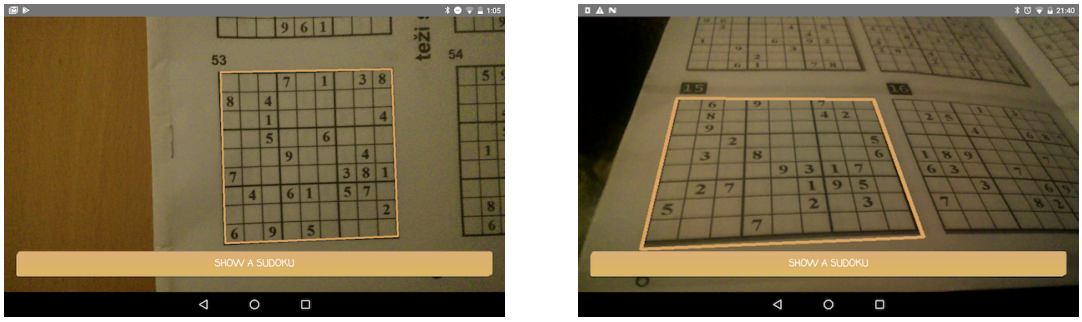


Slika 4.2: Ekran za fotografiranje Sudokua

Ekran za rješavanje igre Sudoku

Nakon bilo koje od opcija - fotografiranje, odabir slike iz galerije ili generiranja slučajne Sudoku igre, korisnik dolazi na ekran za rješavanje igre. Ekran je prikazan na slici 4.4.

Na slici 4.4a prikazan je *screenshot* aplikacije po dolasku na taj ekran. Svi brojevi su upisani crnom bojom, što znači da su inicijalno bili dio Sudokua. Na dnu ekrana vidimo dvije tipke:



(a) Prikaz detektirane Sudoku mreže (b) Prikaz detektirane Sudoku mreže

Slika 4.3: Primjeri ekrana iz aplikacije za rješavanje Sudokua

- **"Solve a Sudoku"** za poziv algoritma za rješavanje Sudokua;
- **"Show Hint"** za prikaz savjeta - pojavljuje se točna znamenka na polju koje je trenutno označeno ili na slučajno odabranom polju, ako nijedno nije označeno.

Nakon što je Sudoku riješen, aplikacija ispisuje prigodnu poruku. Također, korisnik je obaviješten ako Sudoku nije validan (npr. neke od znamenaka nisu ispravno prepoznate na slici).

Primijetimo da su na slici 4.4b neke od tipki zatamnjene, što označava da su blokirane. Naime, aplikacija pomaže korisniku tako što ovisno o okolini odabranog polja prikaže samo brojeve koje je moguće upisati u polje, a da se ne krše pravila igre.

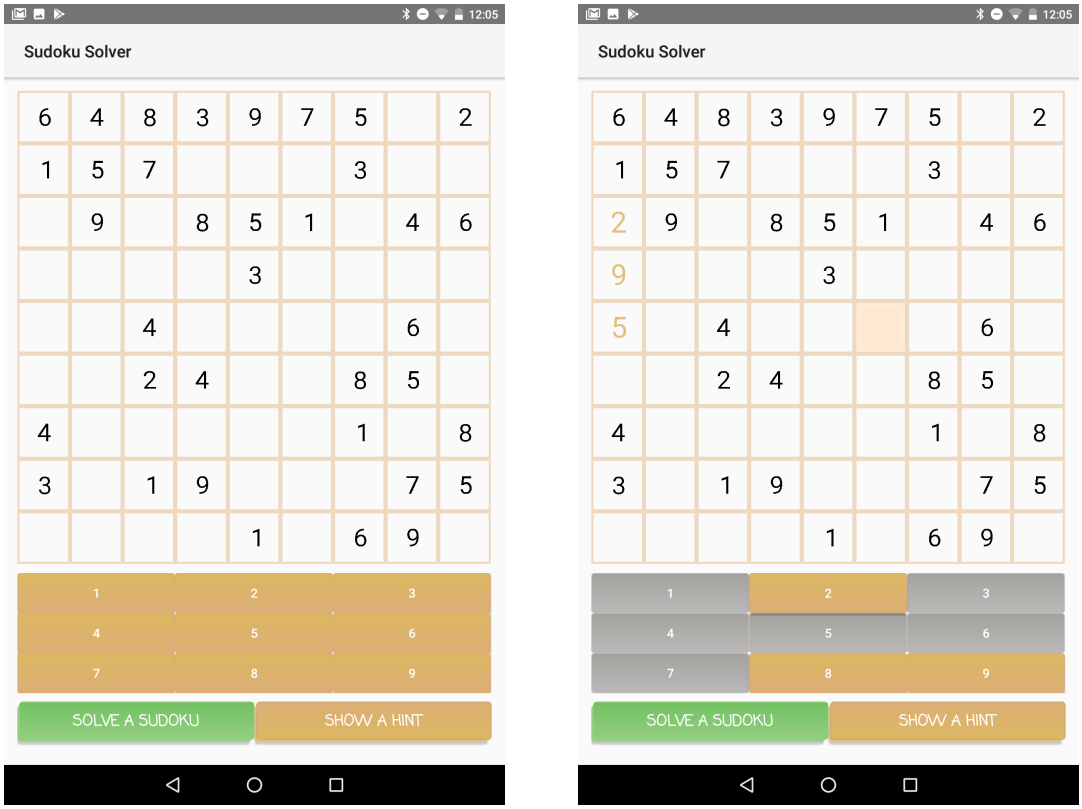
4.4 Komponente aplikacije

U nastavku ćemo se baviti opisom dijelova aplikacije koji su vezane uz primjenu algoritama računalnog vida.

Prepoznavanje rubova sudoku mreže

Prepoznavanje vanjskog ruba Sudoku mreže provodimo na način opisan u niže danom pseudokodu. Ovaj korak provodimo kod fotografiranja Sudokua i nakon odabira slike iz galerije, a izlaz mu je okvir vidljiv na slikama 4.2. i 4.3. Za vrijeme fotografiranja, svakih *N frameova* kamere, ažuriramo okvir Sudoku mreže na osnovu trenutnog *framea*. *N* je zadan kao konstanta u aplikaciji, a na osnovi performansi i FPS[1]. U nastavku je dan dio izvornog koda koji se izvršava za svaki *frame* kamere.

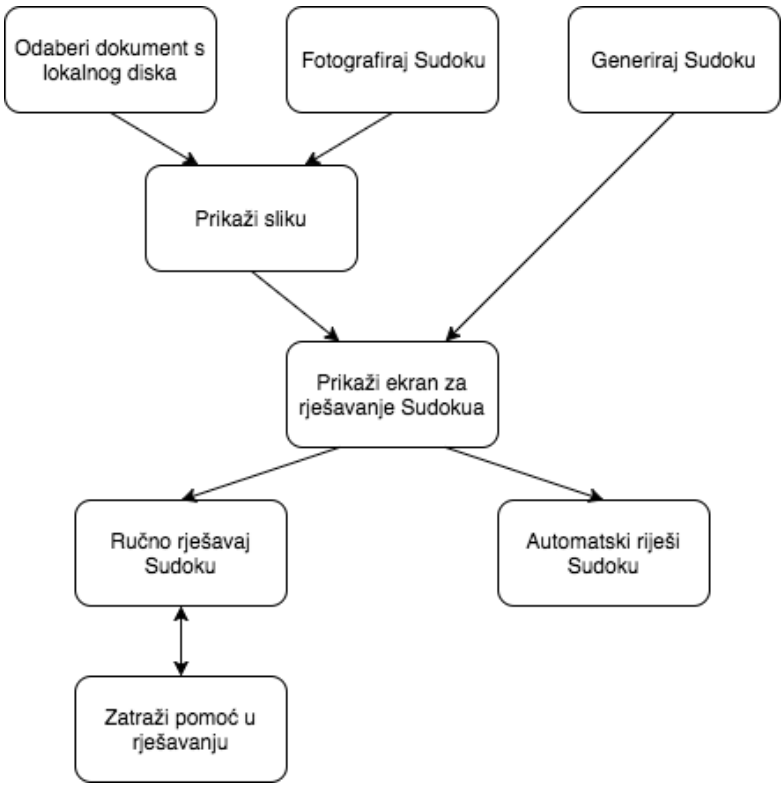
[1]FPS - Frames per Second (broj slika koje kamera producira u sekundi)



(a) Prikaz ekrana za rješavanje Sudokua (b) Prikaz ekrana za rješavanje Sudokua

Slika 4.4: Primjeri ekrana iz aplikacije za rješavanje Sudokua

```
public Mat onCameraFrame(
   CameraBridgeViewBase.CvCameraViewFrame inputFrame) {
    // Dohvatimo originalnu sliku kamere - ovo je slika u boji.
   Mat originalImage = inputFrame.rgba();
    // Kopiramo sadrzaj slike.
   contouredImage = originalImage.clone();
   ++frameCounter;
    // Ako ovo nije N-ti frame, samo vratimo trenutno pronadene granice
        sudoku mreze.
   if (frameCounter % N != 0 && approxContour != null) {
      ImageProcessingUtil.drawApproxContour(contouredImage, approxContour);
      return contouredImage;
   }
    // Pretvorimo sliku u akromatsku i detektiramo konture
```



Slika 4.5: Dijagram toka aplikacije

```
 grayImage = new Mat();
 Imgproc.cvtColor(originalImage, originalImage, Imgproc.COLOR_BGR2RGB);
 Imgproc.cvtColor(originalImage, grayImage, Imgproc.COLOR_RGB2GRAY);
 Mat edgesImage = ImageProcessingUtil.detectEdges(grayImage);
 Mat connectedEdgesImage =
     ImageProcessingUtil.connectComponents(edgesImage);
 // Dohvatimo najvecu konturu
 List<MatOfPoint> contours =
     ImageProcessingUtil.findContours(connectedEdgesImage, false, false);
 if (contours.size() == 0) {
   return inputFrame.rgba();
 }
 // Prva kontura u listi je ona najveca
MatOfPoint outerContour = contours.get(0);
MatOfPoint2f approxContourNew = ImageProcessingUtil.approximateContour(
   outerContour, true);
 if (approxContourNew.total() == 4) {
```

```
      approxContour = approxContourNew;
    }
    if (approxContour == null) {
      return inputFrame.rgba();
    }
   ImageProcessingUtil.drawApproxContour(contouredImage, approxContour);
   return contouredImage;
}
```

Pojasnimo sad ukratko gornji program. Funkcija `onCameraFrame()` dio je sučelja `CameraBridgeViewBase.CvCameraViewListener2` i poziva se na svaki *frame* kamere. Spomenimo još da je `ImageProcessingUtil` pomoćna klasa koja sadrži pomoćne metode, implementirana u sklopu ove aplikacije. Svaki poziv njenim metodama ukratko ćemo u nastavku pojasniti.

Prvo, preskačemo obradu slike ako ona nije N-ta po redu. Na taj način smanjujemo količinu resursa potrebnih za obradu slike te u isto vrijeme poboljšavamo korisničko iskustvo, budući da je, ako koristimo svaki poziv funkcije, teško pratiti promjene rubova na ekranu. Funkcija `ImageProcessingUtil.drawApproxContour()` prima sliku `contouredImage` te na nju crta konturu određenu točkama u `approxContour` koristeći OpenCV funkcije.

Pretvaramo sliku iz njenog *RBG* prikaza u akromatski, kako bismo pojednostavili daljnju obradu. U nastavku detektiramo rubove, koristeći *Cannyjev* algoritam opisan u poglavlju 3. Nakon toga, kako bismo izbjegli situaciju u kojoj su neki rubovi 'istanjeni', izvršavamo funkciju `ImageProcessingUtil.connectConponents()`. Ova funkcija implementira operaciju *morfološkog zatvaranja*, kompoziciju *erozije* i *dilatacije* koju smo također pobliže objasnili u sklopu poglavlja 3.

Znamo da na rubove nemamo garanciju zatvorenosti - u ovoj situaciji želimo pronaći granice Sudoku mreže, koje moraju biti zatvorena krivulja. Zbog toga, koristimo funkciju za pronalazak kontura na slici. Ova funkcija koristi poziv OpenCV funkciji za detekciju kontura te ih nakon toga sortira ovisno o njihovoj površini te vraća onu najveću. Bitno je napomenuti da u ovom koraku aplikacije, pretpostavljamo da će najveći četverokut na slici biti upravo Sudoku mreža. To, naravno, ne mora u svakom slučaju biti točno, no u primjeni ova heuristika daje vrlo dobre rezultate.

Sada smo detektirali željenu konturu na slici. Ipak, još uvijek nam ništa ne garantira da je ta kontura upravo četverokut, što bismo preferirali s obzirom da je u pravilu sudoku pravokutnog oblika. Zbog toga, aproksimiramo konturu najbližim četverokutom, ako je to moguće postići s dovoljno malim odstupanjem od duljine početne konture. Na kraju, ponovo crtamo dobivenu konturu i vraćamo dobivenu sliku.

Izdvajanje Sudoku mreže

Nakon što je korisnik odlučio završiti s fotografiranjem, odveden je na ekran prikazan na slici 4.3. U ovom trenutku i dalje prikazanu imamo cijelu izlaznu sliku kamere, sa označenim dijelom koji želimo u nastavku koristiti. Bitno je napomenuti da, iako na ovom ekranu vidimo sliku u boji, sva obrada je rađena na akromatskoj slici.

Kako bismo iz dobivenog četverokuta dobili kvadrat, što će nam olakšati daljnju obradu Sudokua, koristimo OpenCV funkcije, kao što je prikazano u nastavku. Ovdje je varijabla `srcPoint` lista koja sadrži koordinate kuteva četverokuta prije tranformacije perspektive, a `destPoints` željene kuteve kvadrata nakon primjene funkcije `warpPerspective()`.

```
Mat transform = Imgproc.getPerspectiveTransform(srcPoints, destPoints);
Imgproc.warpPerspective(srcImage, cutImage, transform, cutImage.size());
```

Izdvajanje pojedinačnih ćelija

Za izdvajanje pojedinačnih ćelija iz mreže Sudokua koristimo vrlo jednostavnu heuristiku - dijelimo mrežu na 81 jednakih dijelova (9 u horizontalnom smjeru, 9 u vertikalnom). Ova strategija ima mana, budući da u nekim slučajevima ćelije nisu jednakih veličina, što umanjuje preciznost prepoznavanja znamenaka, no zbog ranije primjenjenih transformacija perspektive, u većini slučajeva to ne predstavlja problem. Neki od alternativnih pristupa bili bi:

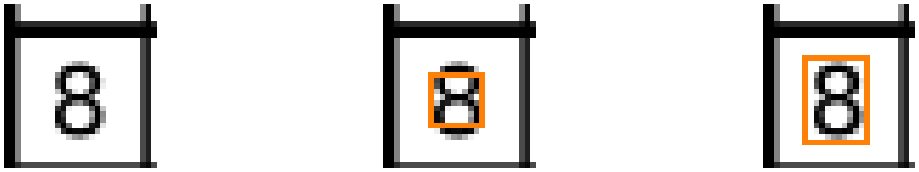
- **Korištenje algoritma *Houghovih linija***
  Primjenjujemo algoritam na Sudoku mrežu i pohranjujemo sve linije koje algoritam vrati. Idealno, pronađenih je 9 horizontalnih i 9 vertikalnih linija. Nažalost, problem ovog pristupa je da je vrlo teško, u općenitom slučaju, to postići - zbog različitih uvjeta vezanih uz osvjetljenje i kvalitetu kamere. Zbog toga, neke od linija nedostaju ili su istanjene, dok u nekim slučajevima postoje dodatne linije za koje ne bismo htjeli da budu detektirane (npr. ponekad je poveznica između ruba ćelije i broja prepoznata kao linija).

- **Detekcija kuteva**
  Nekim algoritmom za detekciju kuteva prepoznamo sve kuteve na slici Sudoku mreže. Ovdje pretpostavljamo da će biti vraćeni točno kutevi svih ćelija. Ovaj pristup ima slične probleme kao i korištenje *Houghovih linija* - neki od kuteva nisu prepoznati, a neki od dijelova slike koji nisu kutevi od interesa su prepoznati.

Prvi pristup, iako naivan, pokazao se kroz testiranje kao najpouzdaniji pa se on nalazi u finalnoj verziji aplikacije.

Nakon što smo izdvojili svaku od ćelija, postoji mogućnost da znamenke u ćelijama nisu u potpunosti centrirane, kao na primjeru slike 4.6a. Zbog toga dodajemo još jedan korak obradi slike.

Za svaku od ćelija, inicijaliziramo kvadrat stranice jednake polovini duljine stranice ćelije i pozicioniramo ga u sredinu ćelije. Proširujemo taj kvadrat (kasnije pravokutnik) sve dok ne dođemo do pravokutnika čiji rub ne pokriva niti jedan crni piksel (naravno, ovo je pojednostavljena verzija onog što se stvarno događa - odabiremo prag na vrijednost piksela, iznad kojeg piksel smatramo bijelim. Ideja je prikazana na slikama 4.6b i 4.6c. U slučaju da je početni pravokutnik jednak onom završnom, pretpostavljamo da je pripadna ćelija *Sudokua* prazna.



(a) Pojedinačna ćelija (b) Početni pravokutnik (c) Krajnji pravokutnik

Slika 4.6: Detekcija znamenke unutar ćelije

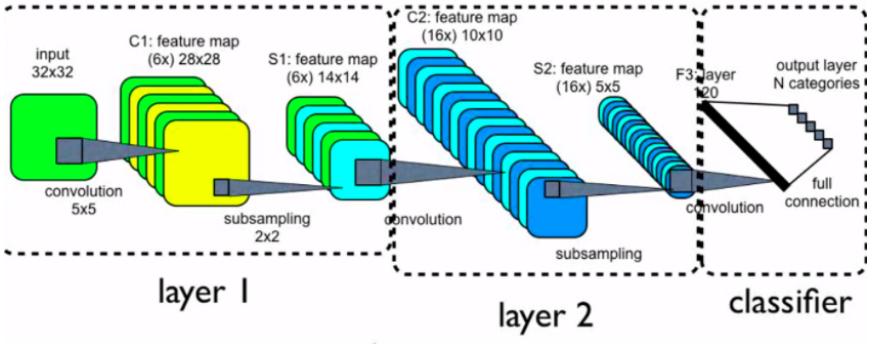
Prepoznavanje znamenaka

Prepoznavanje znamenaka u aplikaciji realizirano je korištenjem konvolucijskih neuronskih mreža. Konvolucijske neuronske mreže algoritmi su koji trenutno postižu najbolje rezultate za problem prepoznavanja rukom pisanih znamenaka (> 99% točnosti).

Korištena mreža sastoji se od 2 konvolucijska sloja, između kojih se nalaze tzv. *pooling* slojevi. Kao posljednji sloj dodan je *dense* sloj. Razvijena je korištenjem TensorFlowa, vrlo popularnog razvojnog okvira za razvoj algoritama strojnog učenja. Specifično, opisani model je predstavljen u sklopu [1]. Struktura mreže prikazana je na slici 4.7. Mreža je trenirana na standardnom setu podataka *MNIST* i na standardiziranom testnom skupu podataka točnost joj je 99.2%.

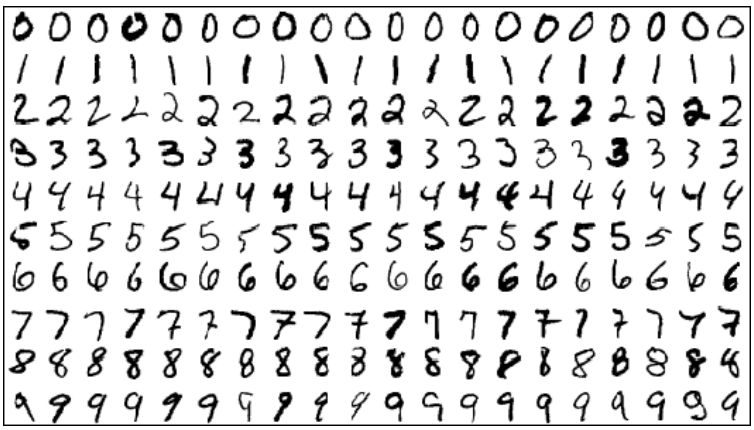
Točnost klasifikatora nešto je niža (oko 97%) kada ga primjenjujemo na naš problem prepoznavanja znamenaka sa slike Sudoku igre. Iako na prvi pogled možda i nije jasno koji je tome razlog, gubitak u točnosti vrlo je lako objasniti.

Naime, iako su znamenke koje su dio skupa podataka *MINST* pisane vrlo raznolikim rukopisima (vidi sliku 4.8), sve su slike normalizirane - crni pikseli na potpuno bijeloj podlozi, znamenka centrirana u kvadrat unutar slike, što uvelike pomaže radu klasifikatora. U našem slučaju to inicijalno nije slučaj. Zbog toga uvodimo dodatni korak, kako bismo poboljšali točnost - skaliramo sliku znamenke (sa slike 4.6c) na veličinu 20×20, centriramo



Slika 4.7: Konvolucijska neuronska mreža

je unutar tog kvadrata, dodajemo bijeli rub toj slici do veličine 28×28. Opisani proces dovodi do poboljšanja točnosti s 50% na 97%. Naravno, točnost dodatno ovisi o kvaliteti kamere, rasvjeti i drugim čimbenicima.



Slika 4.8: Dio *MNIST* skupa podataka

Rješavanje Sudokua

Rješavanje Sudokua implementirano je korištenjem jednostavnog *backtracking*[2] algoritma. U nastavku je prikazana rekurzivna implementacija tog algoritma. Funkcija `legal()` korištena u implementaciji služi za provjeru mogućnosti upisivanja odabrane vrijednosti u ćeliju bez kršenja pravila Sudokua.

[2]backtracking - dubinsko unatražno pretraživanje

```
public static boolean solve(int i, int j, int[][] cells) {
   if (i == 9) {
       i = 0;
      // pronadeno rjesenje
      if (++j == 9) {
         return true;
      }
   }
    // preskoci popunjene celije
   if (cells[i][j] != 0) {
      return solve(i + 1, j, cells);
   }
   for (int val = 1; val <= 9; ++val) {
      if (legal(i,j,val,cells)) {
         cells[i][j] = val;
         if (solve(i+1,j,cells)) {
            return true;
         }
      }
   }
    // resetiraj na povratku
   cells[i][j] = 0;
   return false;
}
```

Slika 4.9: Izvorni kod algoritma za rješavanje Sudokua

U velikoj većini slučajeva, performanse predstavljenog algoritma dovoljno su dobre za upotrebu u aplikaciji. Situacija u kojoj dolazi do usporavanja algoritma pojavljuje se kod proslijeđivanja početne konfiguracije Sudokua koja nije u skladu s pravilima (npr. kod fotografiranja neke od znamenaka nisu korektno prepoznate). U tom slučaju algoritam mora obraditi veliki broj kombinacija pa nakon isteka određenog vremena rada algoritma, korisniku ispisujemo poruku o nemogućnosti rješavanja Sudokua.

Performanse rješavača mogle bi biti poboljšane korištenjem sofisticiranijeg *backtracking* algoritma ili npr. implementacijom genetskog algoritma.

Implementacijski detalji

Većina aplikacije implementirana je korištenjem OpenCV Java API-ja opisanog u poglavlju 4. Dio zahtjevnijih operacija u smislu resursa implementirani su u C++ i integrirani *JNI*

pozivom.

Specifično za Tegra sustav, tj. činjenicu da sadrži integrirani NVIDIA GPU, moguće je, u svrhu poboljšanja performansi, koristiti modul CUDA. Signature OpenCV funkcija iz ovog modula analogne su onim standardnim za CPU, uz neznatne prilagodbe u nazivlju (npr. umjesto strukture `Mat` koristimo `GpuMat`). Ovaj je modul također vrlo dobro dokumentiran i jednostavan za korištenje.

Spomenimo još ukratko način na koji je neuronska mreža korištena za prepoznavanje znamenaka integrirana u Android aplikaciju za rješavanje Sudokua. Mreža je kreirana koristeći TensorFlow programsko sučelje u *Pythonu* te su nakon treniranja njena struktura i dobivene težine spremljene u *Protocol Buffers* formatu [4]. Sama integracija s izvornim kodom u Javi implementirana je korištenjem TensorFlow programskog sučelja za evaluaciju modela (eng. *inference*) u Javi [5].

4.5 Rezultati prepoznavanja znamenaka

Na kraju dajemo još kratku statistiku točnosti prepoznavanja znamenaka. Korišteno je 20 različitih fotografiranih Sudokua i dobivena je točnost iz tablice 4.1.

Točnost - bez praznih ćelija ($\frac{\text{\# Točno prepoznatih znamenaka}}{\text{\# Svih ćelija koja imaju upisane znamenke}}$)	97.68%
Točnost - uključujući prazne ćelije ($\frac{\text{\# Točno klasifikiranih ćelija}}{\text{\# Svih ćelija}}$)	99.32%

Tablica 4.1: Točnost prepoznavanja znamenaka

Osim općenite točnosti prepoznavanja znamenaka, zanimljivo je proučiti i preciznost[3] i recall[4] za pojedinačne znamenke.

Vidimo da je za većinu znamenaka preciznost i recall vrlo blizu 100%. Nižu preciznost primjećujemo kod znamenaka 7 i 3, dok je niži recall za znamenke 9 i 5. Promatranjem ponašanja klasifikatora, vidimo da je najčešći problem pogrešna klasifikacija broja 9 kao broj 7 te broja 5 kao broj 3.

Ove je pogreške djelomično moguće objasniti vizualnom sličnošću parova tih znamenaka te eventualnom informacijom koju gubimo kroz obradu slike (promjena veličine, *thresholding*). U nastavku rada na aplikaciji, poboljšanje navedenih problema moglo bi se postići prilagođavanjem *thresholda* potrebnog za donošenje odluke da je na slici određena znamenka, specifično za problematične znamenke. Ipak, uzimajući u obzir da primjeri znamenaka nisu iz nekog standardnog skupa podataka, već su i ovi rezultati vrlo obećavajući.

[3] Preciznost je definirana kao $\frac{\text{\# primjera točno prepoznate znamenke X}}{\text{ukupan broj primjera klasifiranih kao znamenka X}}$

[4] Recall je definiran kao $\frac{\text{\# primjera točno prepoznate znamenke X}}{\text{ukupan broj primjera znamenke X}}$

Znamenka	**Preciznost**	**Recall**
1	100.00%	98.18%
2	97.92%	100.00%
3	**89.66%**	100.00%
4	100%	98.11%
5	96.87%	**88.10**%
6	100%	98.33%
7	**94.92%**	100.00%
8	100.00%	100%
9	100.00%	**95.00**%

Tablica 4.2: Točnost prepoznavanja znamenaka

Bibliografija

[1] *Deep MNIST for Experts*, https://www.tensorflow.org/versions/r0.12/tutorials/mnist/pros/, posjećeno 10.11.2017.

[2] *NVIDIA TADP dokumentacija*, http://docs.nvidia.com/gameworks/index.html#technologies/mobile/getting_started.htm, posjećeno 12.9.2017.

[3] *OpenCV dokumentacija*, https://docs.opencv.org/, posjećeno 10.12.2017.

[4] *Protocol Buffers*, https://developers.google.com/protocol-buffers/, posjećeno 22.1.2017.

[5] *TensorFlow Inference Interface*, https://developers.google.com/protocol-buffers/, posjećeno 22.1.2017.

[6] Daniel Lélis Baggio, *Mastering OpenCV with practical computer vision projects*, Packt Publishing Ltd, 2012.

[7] Gary Bradski i Adrian Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*, O'Reilly Media, Inc., 2008.

[8] Adrian Kaehler i Gary Bradski, *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*, 1. izd., O'Reilly Media, Inc., 2016.

[9] Salil Kapur i Nisarg Thakkar, *Mastering OpenCV Android Application Programming*, Packt Publishing Ltd, 2015.

[10] Amgad Muhammad, *OpenCV Android Programming By Example*, Packt Publishing Ltd, 2015.

[11] Utkarsh Sinha, *SuDoKu Grabber in OpenCV*, http://aishack.in/tutorials/sudoku-grabber-opencv-plot/, posjećeno 10.12.2017.

[12] Satoshi Suzuki i ost., *Topological structural analysis of digitized binary images by border following*, Computer vision, graphics, and image processing **30** (1985), br. 1, 32–46.

Sažetak

OpenCV je vrlo popularna biblioteka otvorenog koda koja sadrži mnoštvo struktura podataka i algoritama koji se mogu koristiti kao komponente za razvijanje aplikacija povezanih s računalnim vidom. U ovom radu bavimo se OpenCV-om iz perspektive mobilnih uređaja s operacijskim sustavom Android. Također, fokusiramo se na platformu NVIDIA Tegra koja integrira ARM CPU i NVIDIA akceleratore na jednom čipu. Biblioteka je predstavljena kroz primjer aplikacije za rješavanje Sudokua.

Aplikacija koristi funkcionalnosti biblioteke OpenCV za dohvat ulaza kamere mobilnog uređaja i prepoznavanje mreže Sudokua. Prepoznata mreža je nakon toga normalizirana i izdvojene su individualne ćelije. Svaka od ćelija je tada proslijeđena kao ulaz konvolucijskoj neuronskoj mreži (CNN) implementiranoj u *TensorFlowu* koja klasificira piksele ćelija u klasu pripadne znamenke. Konačno, generirana je digitalna verzija skenirane Sudoku igre, koju korisnik moje rješavati ručno ili zatražiti automatsko rješavanje implementiranim algoritmom. Ako se Sudoku rješava ručno, postoji mogućnost dobivanja savjeta u vidu točne znamenke na nekom od polja.

Summary

OpenCV is a widely used open source library providing various data structures and algorithms which can be used as building blocks for computer vision applications. This thesis studies OpenCV from the perspective of portable, Android-based devices. It focuses on the mobile NVIDIA Tegra platform which features an ARM CPU coupled with an NVIDIA GPU accelerator on a single chip. The library is presented in the context of an example Sudoku-solver application.

Sudoku-solver combines primitives from OpenCV to dynamically scan the input of the portable device camera and detect a Sudoku grid. The detected grid is normalized and individual cells extracted. The cells are then passed to a convolutional neural network (CNN), built on top of TensorFlow, to classify the pixels of each cell to the correct digit. Finally, a digital version of the scanned Sudoku is generated, which the user can either solve manually, or request an immediate solution from Sudoku-solver. If solving the puzzle manually, the application can also be queried for hints.

Životopis

Jelena Držaić rođena je 27. kolovoza 1993. godine u Varaždinu. Živi u Žarovnici, gdje pohađa osnovnu školu. Nakon osnovne škole upisuje smjer Opće gimnazije u sklopu Srednje škole Ivanec te se nakon završetka 2012. godine seli u Zagreb gdje upisuje sveučilišni preddiplomski studij matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta. 2015. godine završava preddiplomski studij i upisuje Diplomski studij Računarstvo i matematika na istom fakultetu. Na prvoj godini diplomskog studija bila je demonstrator iz kolegija *Vektorski prostori*.

Tijekom srednje škole sudjeluje na državnim natjecanjima iz matematike i informatike. Nakon završenog preddiplomskog studija, odrađuje praksu u tvrtci *Orion* gdje se bavi sigurnošću web aplikacija. Uz diplomski studij, radi u tvrtci *Implementacija Snova* na pozicijama iOS i Backend developera, a kasnije i kao voditelj tima za kvalitetu softvera. Odrađuje i dvije prakse u tvrtci *Google*, gdje radi na razvoju klasifikatora u sklopu *YouTube ContentID* sustava te poboljšanju performansi *Android Google Search* aplikacije.