Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Miha Pleško

Razpoznavanje objektov kot spletna storitev

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Matej Kristan

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00091 / 2013
Datum: 10.4.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MIHA PLEŠKO**

Naslov: **RAZPOZNAVANJE OBJEKTOV KOT SPLETNA STORITEV**
OBJECT RECOGNITION AS A WEB SERVICE

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

V nalogi obravnavajte problem razpoznavanja objektov v kontekstu spletne storitve. Kot praktičen primer kompleksnega sistema, ki nudi tako storitev, opišite sistem Vicos Eye. Osredotočite se tako na metode računalniškega vida, ki jih storitev implementira, kakor tudi na vidik učinkovitosti izvedbe teh metod. Kljub učinkoviti implementaciji, je pomanjkljivost sistema Vicos Eye slaba sposobnost razpoznavanja objektov posnetih s poljubnega zornega kota. V nalogi predlagajte možno rešitev s postopkom predhodne geometrijske normalizacije. Analizo primernosti izboljšave izvedite na standarni slikovni zbirki s praktičnim sistemom razpoznavanja objektov.

Mentor:
doc. dr. Matej Kristan

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Miha Pleško, z vpisno številko **63100330**, sem avtor diplomskega dela z naslovom:

Razpoznavanje objektov kot spletna storitev

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Mateja Kristana,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 10. marca 2014 Podpis avtorja:

Iskreno se zahvaljujem mentorju doc. dr. Mateju Kristanu ne le za nasvete na visokem algoritemskem nivoju, temveč tudi za pomoč na nivoju konkretne implementacije. Zahvaljujem se diplomiranem inženirju računalništva in informatike Domnu Taberniku za pomoč pri eksperimentalnem vrednotenju algoritma na samem sistemu. Hvala tudi staršem in Alji za podporo v času nastajanja diplome.

Kazalo

Seznam uporabljenih kratic in simbolov

1. **ViCoS** - The Visual Cognitive Systems Laboratory
2. **HoC** - Histogram of Compositions
3. **lHoP** - Learnt Hierarchy of Parts
4. **SVM** - Support Vector Machine
5. **HDFS** - Hadoop Distributed File System
6. **YARN** - Hadoop Yet-Another-Resource-Negotiator
7. **PCA** - Principal Component Analysis
8. **SVD** - Singular Value Decomposition
9. **DoG** - Difference of Gaussian

Povzetek

V diplomski nalogi obravnavamo problem razpoznavanja objektov kot spletne storitve. Osredotočimo se na konkreten sistem ViCoS Eye, ki so ga razvili v Laboratoriju za umetne vizualne spoznavne sisteme na Fakulteti za računalništvo in informatiko.

Na začetku predstavimo najpomembnejše algoritme računalniškega vida, na katerih temelji sistem ViCoS Eye. Predstavimo metodo podpornih vektorjev, naučeno hierarhijo delov in opisnik HoC ter kombinacijo slednjih dveh. Sledi obravnava sistema s tehnološkega vidika. Predstavimo arhitekturi Apache Hadoop, na kateri se izvajajo algoritmi učenja modelov, in Storm, na kateri se izvajajo algoritmi predikcije. Nato prikažemo, kako so vsi ti algoritmi učinkovito vpeti na pleča arhitekture. Opišemo način, kako je algoritem učenja učinkovito razdeljen na množico zaporednih korakov, ki so kompatibilni s paradigmo MapReduce, in način, kako je algoritem predikcije učinkovito razdeljen na množico zaporednih korakov, ki so skladne s predpostavkami arhitekture Storm.

V nadaljevanju predstavimo, implementiramo in eksperimentalno ovrednotimo naš predlog za izboljšavo sistema. Opazimo, da je deskriptor HoC občutljiv na rotacije objektov, zato predstavimo algoritem afine normalizacije vhodnih slik na podlagi odvodov. Uspešnost izboljšave smo testirali na standardni bazi Caltech-101 v dveh različnih kontekstih. Algoritem do neke mere izboljša rezultate, kadar so slike poljubno rotirane. Vendar pa bistveno poslabša rezultate, kadar objekti niso slikani v poljubnih orientacijah.

Ključne besede: računalniški vid v oblaku, sistem za kategorizacijo, afina normalizacija, ViCoS Eye

Abstract

The main focus of this work is on object categorization as a web service. A concrete system is studied, ViCoS Eye, which was developed by Visual Cognitive Systems Laboratory at Faculty of Computer and information Sciences.

First we introduce the core computer vision algorithms that ViCoS Eye is based on. Support Vector Machine, Learnt-hierarhy-of-parts, Histogram of compositions descriptor and a combination of the last two are presented. Then we discuss open-source framework Apache Hadoop, that is used to efficiently run machine learning part of system. We also discuss another open-source framework, Storm, that is used for running categorization part of system. In addition, the efficient integration of algorithms onto the two frameworks is described. We split learning algorithm into smaller steps to follow the MapReduce paradigm. Similarly, prediction algorithm is split into smaller steps that are perfectly compatible with Storm framework.

Finally we introduce, explain, implement and experimentally evaluate our own improvement of the system. Since we noticed that HoC descriptor is very sensitive when it comes to rotation of the input picture, we introduce an affine derivatives-based normalisation algorithm. Tests on standard Caltech-101 database were performed in two different contexts. The algorithm improves the results to some degree, when the input images are rotated. But a significant drop of classification accuracy was detected when the input objects were not deformed.

Keywords: cloud computer vision, categorisation system, affine normalisation, ViCoS Eye

Poglavje 1

Uvod

Predstavljajmo si, da smo v gozdu in naletimo na zanimivo gobo. Takoj privlečemo svoj zmogljivi mobilni telefon in jo slikamo. Sliko želimo poslati prijateljem, vendar bi poleg nje radi napisali, katera goba je to. Nekoč je bila edina možnost, da najdemo odgovor ta, da smo vzeli ustrezen priročnik in v njem poiskali sliko naše gobe ter prebrali njeno ime. Druge možnosti ni bilo. Danes, ko živimo v informacijski dobi, se nam listanje po slikovnih priročnikih zdi odveč. Pojavlja se potreba po spletni storitvi, na katero bi uporabnik poslal fotografijo poljubnega predmeta (npr. gobe) in bi v odgovor dobil besedno poimenovanje (npr. "Jesenski goban, znan tudi pod imenom jurček").

1.1 Sorodna dela

Na prvi pogled se nam zazdi, kot da takšni sistemi že obstajajo. Poznamo npr. Google Image Search [1, 2], Google Googles [3], TinEye [4], Macroglossa [5, 6]. Vendar se ti sistemi ukvarjajo z iskanjem podobnih slik in ne s kategorizacijo. V našem primeru bi nam bili sposobni vrniti množico fotografij z gobi podobnimi objekti, da smo fotografirali jurčka, pa nam ne bi znali povedati.

Značilno za obstoječe sisteme je, da (i) so namenjeni iskanju vhodni fotografiji podobnih slik in ne klasifikaciji, (ii) delujejo dobro le na slikah, ki so jih že kdaj videli in (iii) se ne zavedajo semantične vsebine fotografij [7, 8]. V tem se

sistem ViCoS Eye že v svoji zasnovi razlikuje od njih: (i) namenjen je detekciji in klasifikaciji objektov na fotografiji, (ii) klasificirati naj bi znal tudi povsem nove fotografije in (iii) zavedal naj bi se semantične vsebine fotografij [8, 9].

Navedeni sorodni sistemi so vsi zaprte komercialne narave in do informacij o njihovem konkretnem pristopu, npr. s kakšnem deskriptorjem opisujejo slike, nimamo dostopa. Po drugi strani za sistem ViCoS Eye obstaja dovolj strokovnih člankov, iz katerih je razvidno, da le-ta uporablja deskriptor HoC (ang., Histogram of Compositions) [10, 11]. HoC se dobro obnese pri opisovanju slik na različnih skalah, saj uporablja fiksno število regij, dvanajst, na katere razdeli sliko. Za primerjavo, sorodni deskripotr HoG (ang., Histogram of Oriented Gradients) [12] sliko razdeli na fiksno velike regije kvadratne oblike. Njihovo število zavisi od dimenzij slike, s čimer nastane problem različnosti deskriptorja iste slike, če jo skaliramo. Tako deskriptorju HoC, kot deskriptorju HoG, je skupna občutljivost na rotacijo. Če primerjamo vrednosti HoC-a nad izbrano sliko in nad za 90 stopinj rotirano to isto sliko, opazimo, da se bistveno razlikujeta, sistem odpove. Zato je smiselno sliko predhodno geometrijsko normalizirati.

1.2 Cilji in prispevki

Ob ustvarjanju diplomske naloge smo si zadali dva cilja. Najprej si želimo predstaviti obstoječi sistem ViCoS Eye, tako z vidika računalniškega vida, kot z vidika strežniške arhitekture. Sistem v sebi skriva čudovito neizčrpno kompleksnost, ki je posledica kompleksnosti narave področja, s katerim se ukvarja - računalniškega vida.

Drugi cilj, ki smo si ga zadali, je poskus vnosa svoje ideje v sistem z namenom izboljšanja natančnosti klasifikacije. Predlagamo uporabo metode za lokalno afino normalizacijo izbrane regije. S tem želimo vse slike normalizirati tako, da zmanjšamo vpliv geometrijske deformacije, ki nastane ob zajemu slike, in s tem izboljšati delovanje deskriptorja HoC. Cilj je študija primernosti te metode za normalizacijo slik in njen vpliv na uspešnost klasifikacijskega sistema ViCoS Eye.

1.3 Sestava diplomskega dela

V Poglavju 1 predstavimo tematiko diplomskega dela, njene cilje in prispevke, ter sestavo diplomskega dela. V Poglavju 2 predstavimo najpomembnejše algoritme računalniškega vida, na katerih temelji sistem ViCoS Eye. Opišemo metodo podpornih vektorjev, naučeno hierarhijo delov in opisnik HoC ter kombinacijo slednjih dveh. V Poglavju 3 sledi obravnava sistema s tehnološkega vidika: predstavimo arhitekturi Apache Hadoop, na kateri se izvajajo algoritmi učenja modelov, in Storm, na kateri se izvajajo algoritmi predikcije. Nato si odgovorimo na vse bolj očitno vprašanje - kako so vsi ti algoritmi učinkovito vpeti na pleča arhitekture.

V nadaljevanju v Poglavju 4 predstavimo, v Poglavju 5 implementiramo in v Poglavju 6 eksperimentalno ovrednotimo naš predlog za izboljšavo sistema, afino normalizacijo vhodnih slik na podlagi njihovih odvodov.

Poglavje 2

Predstavitev ključnih algoritmov računalniškega vida

Aplikacije s področja računalniškega vida zahtevajo uporabo nekaterih osnovnih algoritmov. Ena od najuspešnješih metod za razpoznavnaje objektov v računalniškem vidu je metoda podpornih vektorjev. Prav tako si klasifikacije slik brez algoritma naučene hierarhije delov skoraj ne moremo predstavljati. Za učinkovito razpoznavanje moramo slike najprej na čim bolj diskriminativen način pretvoriti v vektorje. Za to obstajajo posebni deskriptorji, npr. deskriptor naučene hierarhije delov. Osnove omenjenih algoritmov si bomo ogledali v nadaljevanju.

Podpoglavje 2.1 nas seznani z metodo podpornih vektorjev, sledi mu Podpoglavje 2.2, kjer predstavimo algoritem naučene hierarhije delov. V Podpoglavju 2.3 opišemo, kako je definiran deskriptor HoC, in v Podpoglavju 2.4 kako lahko kombiniramo HoC z naučeno hierarhijo delov.

2.1 Metoda podpornih vektorjev

Metoda podpornih vektorjev (ang., Support Vector Machine, znana pod kratico SVM) [13,14] je matematični postopek učenja klasifikacije primerkov (ang., samples) v pozitivni oz. negativni razred. Za podano učno množico primerkov z

znanimi pripadnostmi razredov z metodo podpornih vektorjev izračunamo model, s katerim nato klasificiramo poljuben primerek v pozitiven oz. negativen razred. Izračun modela izvedemo v dveh korakih:

1. preslikava primerkov v vektorski prostor $\mathbb{R}^n$
2. izračun ločilne hiperravnine

Preslikava primerkov v vektorski prostor

Preslikava primerkov v vektorski prostor pomeni, da moramo vsak primerek predstaviti z vektorjem iz $\mathbb{R}^n$. Metoda podpornih vektorjev namreč deluje nad množico vektorjev, ki pripadajo dvem različnim razredom (tj. pozitivnim, ki ga označimo s +1 in negativnim, ki ga označimo z -1), in so linearno ločljivi. Komponente vektorja posameznega primerka morajo na čimbolj diskriminativen način opisovati primerek; od tega je odvisna kakovost strojnega učenja.

2.1.1 Izračun ločilne hiperravnine

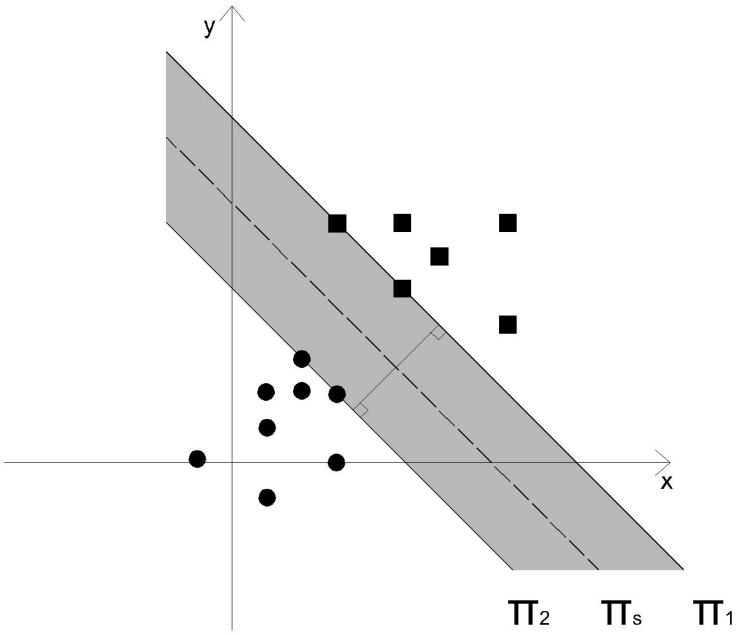
Izračun ločilne hiperravnine se izvede v vektorskem prostoru $\mathbb{R}^n$. Izračunati moramo enačbo hiperravnine, ki najbolje ločuje pozitivne vektorje od negativnih. To storimo tako, da poiščemo takšen pas, katerega širina je maksimalna možna za dano množico vektorjev, ki prostor razdeli na dva dela in so v prvem delu le vektorji pozitivnega razreda, v drugem delu le vektorji negativnega razreda, v samem pasu pa ni nobenega vektorja (glej Sliko 2.1).

Ločilno hiperravnino enolično določata njen normalni vektor $\vec{n}$ in skalarni parameter b (ang., bias):

$$\begin{array}{lccccc} \Pi_S: & Ax & + & By & + \mathrm{b} & = 0 \\ & \begin{bmatrix} A \\ B \end{bmatrix} & \cdot & \begin{bmatrix} x \\ y \end{bmatrix} & + \mathrm{b} & = 0 \\ & \vec{n} & \cdot & \vec{x} & + \mathrm{b} & = 0 \end{array}$$

Robova pasu v svojih enačbah vsebujeta ista dva parametra $\vec{n}$ in b:

$$\Pi_1 : \vec{n}\vec{x}_1 + b = +1 \tag{2.1}$$



Slika 2.1: Grafični prikaz ločilne hiperravnine Π_S (črtkano) za vektorski prostor $\mathbb{R}^2$. Krogci predstavljajo vektorje, ki pripadajo negativnemu razredu, kvadratki pa pozitivnemu. Pas med hiperravninama Π_1 in Π_2 je najširši možen za to učno množico. Vektorje, ki ležijo natanko na robu pasu, imenujemo **podporni vektorji**. Ostali vektorji nimajo vpliva na položaj ločilne hiperravnine Π_S, ki poteka natanko po sredini med robovoma Π_1 in Π_2, zato jih pri izračunu zanemarimo.

$$\Pi_2 : \vec{n}\vec{x}_2 + b = -1 \tag{2.2}$$

Vsi kvadratki ležijo nad hiperravnino Π_1, vsi krogci pa pod hiperravnino Π_2:

$$\begin{aligned} \text{vsi kvadratki } (y_i = +1): &\quad \vec{n}\vec{x}_i + b \geq +1 \\ \text{vsi krogci } (y_i = -1): &\quad \vec{n}\vec{x}_i + b \leq -1 \end{aligned}$$

$$y_i(\vec{n}\vec{x}_i + b) \geq 1 \tag{2.3}$$

Opazimo, da za vsak vektor $\vec{x}_i$ velja enačba (2.3), ki je unija pogojev za vse kvadratke in vse krogce.

Če enačbi ravnin Π_1 in Π_2 odštejemo med seboj in normiramo z dolžino normale:

$$\Pi_1 - \Pi_2 : \vec{n}(\vec{x}_1 - \vec{x}_2) = 2$$
$$\frac{\Pi_1 - \Pi_2}{|\vec{n}|} : \frac{\vec{n}}{|\vec{n}|}(\vec{x}_1 - \vec{x}_2) = \frac{2}{|\vec{n}|}$$

opazimo, da skalarni produkt na levi strani enačbe ravno ustreza pravokotni projekciji povezave med podpornim vektorjem na eni strani pasu in podpornim vektorjem na nasprotni strani pasu na normirano normalo. Vrednost skalarnega produkta torej natanko ustreza širini pasu.

$$d(\Pi_1, \Pi_2) = \frac{2}{|\vec{n}|}$$

Naloga metode podpornih vektorjev je poiskati enačbo tiste hiperravnine, ki bo imela maksimalno širino pasu in pri kateri bo veljal pogoj (2.3). Rešiti moramo enačbo:

$$\min(|\vec{n}|) \text{ pri pogoju } y_i(\vec{n}\vec{x}_i + b) \geq 1$$

Pogoji, v katerih nastopajo vektorji, ki niso podporni vektorji (torej ne ležijo na robu pasu), ne vplivajo na rešitev enačbe, zato jih preprosto izpustimo. V pogojih, v katerih nastopajo podporni vektorji, pa lahko neenakosti zamenjamo z enakostmi, saj ti vektorji ležijo natanko na robu pasu. Torej rešujemo ekvivalentno enačbo:

$$\min(|\vec{n}|) \text{ pri pogoju } y_i(\vec{n}\vec{x}_i + b) = 1;\ \vec{x}_i \text{ podporni vektor}$$

Iskanje vezanih ekstremov lahko izvedemo s pomočjo Lagrangeovih multiplikatorjev, vendar je računanje zapleteno, saj moramo rešiti nepolinomski sistem šestih enačb s šestimi neznankami. Rešitev lažje poiščemo s prevedbo problema na dualni problem, kot je opisano v naslednjem primeru.

2.1.2 Primer izračuna parametrov ločilne hiperravnine

Ponazorimo delovanje metode podpornih vektorjev na preprostem linearno ločljivem primeru v prostoru $\mathbb{R}^2$ [15]. Recimo, da imamo podano učno množico s primerki

negativnega razreda, opisanimi z naslednjimi vektorji:

$$\{\begin{bmatrix}-1\\0\end{bmatrix}, \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}\}$$

ter s primerki pozitivnega razreda, opisanimi z:

$$\{\begin{bmatrix}3\\2\end{bmatrix}, \begin{bmatrix}2\\3\end{bmatrix}, \begin{bmatrix}3\\3\end{bmatrix}\}$$

Iz Slike 2.2 je očitno, da so podporni vektorji trije:

$$\{\vec{x}_1 = \begin{bmatrix}1\\0\end{bmatrix}, \vec{x}_2 = \begin{bmatrix}3\\2\end{bmatrix}, \vec{x}_3 = \begin{bmatrix}2\\3\end{bmatrix}\}$$

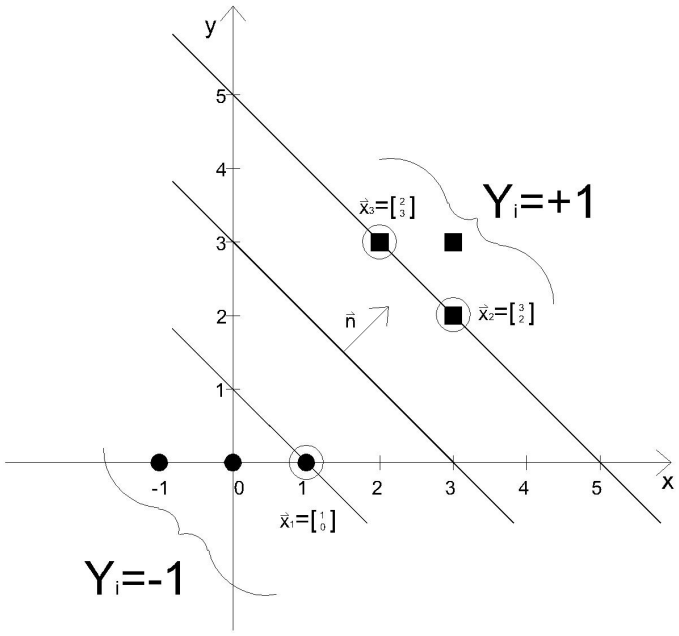
Pozoren bralec bo že zgolj na podlagi Slike 2.2 uganil, da je iskana ločilna hiperravnina kar premica z enačbo $y = -x + 3$ oz. implicitno zapisano $x + y - 3 = 0$. Naš cilj je priti do istega rezultata po metodi podpornih vektorjev.

Spomnimo se, da je cilj metode podpornih vektorjev poiskati enačbo hiperravnine, ki najbolje ločuje podana dva razreda učne množice. V našem demonstracijskem primeru, ki je postavljen v dvodimenzijski prostor, je ločilna hiperravnina preprosto premica. Njena enačba bo:

$$Ax + By + b = 0$$

pri čemer koeficienta A in B ustrezata prvi in drugi komponenti normale $\vec{n}$ na hiperravno, koeficient b (ang., bias) pa predstavlja odmik hiperravnine od izhodišča koordinatnega sistema.

V nadaljevanju bomo pri izračunih namesto 2D vektorjev uporabljali njihovo preslikavo v homogene 3D koordiante. Preslikava je trivialna, 2D vektorju le dodamo tretjo dimenzijo, ki ima vrednost 1. Tako se na primer 2D vektor $(1, 0)$ preslika v 3D vektor $(1, 0, 1)$. Dodatna dimenzija je potrebna, ker se pri izračunu vanjo shrani vrednost parametra b (ang., bias). Ime homogene različice vektorja nadpišemo s tildo. Na primer 2D vektor $\vec{x}_1$ dobi v svoji homogeni inačici oznako $\widetilde{x}_1$.



Slika 2.2: Preprosta, vendar ne trivialna učna množica, preslikana v vektorski prostor $\mathbb{R}^2$. Okrogle točke pripadajo negativnemu razredu z oznako -1, kvadratne točke pripadajo pozitivnemu razredu z oznako 1.

$$\widetilde{n} = \begin{bmatrix} A \\ B \\ b \end{bmatrix} = \sum_i \alpha_i \widetilde{x_i} \tag{2.4}$$

$$y_k = \widetilde{n} \cdot \widetilde{x_k} = \sum_i \alpha_i \widetilde{x_i} \widetilde{x_k} \tag{2.5}$$

Oglejmo si zgornji enačbi. Enačba (2.4) je zapis znanega dejstva, da je normala na hiperravnino vedno linearna kombinacija podpornih vektorjev, le da je tokrat zapisana v homogenem koordinatnem sistemu. Njena zadnja komponenta bo po izračunu vsebovala pravilno vrednost parametra b (ang., bias). Enačba (2.5) predstavlja eno izmed oblik predpisa klasifikacijskega razreda danega vektorja $\vec{x}_k$. Za vsakega izmed treh podpornih vektorjev vstavimo znane vrednosti v

enačbo (2.5):

$$\begin{aligned}
-1 &= \alpha_1 \begin{bmatrix}1\\0\\1\end{bmatrix}\begin{bmatrix}1\\0\\1\end{bmatrix} + \alpha_2 \begin{bmatrix}3\\2\\1\end{bmatrix}\begin{bmatrix}1\\0\\1\end{bmatrix} + \alpha_3 \begin{bmatrix}2\\3\\1\end{bmatrix}\begin{bmatrix}1\\0\\1\end{bmatrix} \\
1 &= \alpha_1 \begin{bmatrix}1\\0\\1\end{bmatrix}\begin{bmatrix}3\\2\\1\end{bmatrix} + \alpha_2 \begin{bmatrix}3\\2\\1\end{bmatrix}\begin{bmatrix}3\\2\\1\end{bmatrix} + \alpha_3 \begin{bmatrix}2\\3\\1\end{bmatrix}\begin{bmatrix}3\\2\\1\end{bmatrix} \\
1 &= \alpha_1 \begin{bmatrix}1\\0\\1\end{bmatrix}\begin{bmatrix}2\\3\\1\end{bmatrix} + \alpha_2 \begin{bmatrix}3\\2\\1\end{bmatrix}\begin{bmatrix}2\\3\\1\end{bmatrix} + \alpha_3 \begin{bmatrix}2\\3\\1\end{bmatrix}\begin{bmatrix}2\\3\\1\end{bmatrix}
\end{aligned}$$

Izračunamo skalarne produkte in dobimo sistem treh linearnih enačb s tremi neznankami:

$$\begin{aligned}
2\alpha_1 + 4\alpha_2 + 3\alpha_3 &= -1 \\
4\alpha_1 + 14\alpha_2 + 13\alpha_3 &= 1 \\
3\alpha_1 + 13\alpha_2 + 14\alpha_3 &= 1
\end{aligned}$$

Rešimo sistem (npr. z Gauss-Jordanovo eliminacijo) in dobimo vrednosti manjkajočih parametrov:

$$\alpha_1 = -\frac{17}{8},\ \alpha_2 = \frac{11}{8},\ \alpha_3 = -\frac{6}{8}$$

Sedaj poznamo vse parametre, ki jih potrebujemo za izračun normale na hiperravnino in parametra b (ang., bias) po enačbi (2.4):

$$\vec{n} = \begin{bmatrix}A\\B\\b\end{bmatrix} = \sum_i \alpha_i \widetilde{x}_i =$$

$$= \alpha_1 \widetilde{x}_1 + \alpha_2 \widetilde{x}_2 + \alpha_3 \widetilde{x}_3 =$$

$$= -\frac{17}{8}\begin{bmatrix}1\\0\\1\end{bmatrix} + \frac{11}{8}\begin{bmatrix}3\\2\\1\end{bmatrix} + \frac{6}{8}\begin{bmatrix}2\\3\\1\end{bmatrix} =$$

$$= \frac{1}{2}\begin{bmatrix}1\\1\\-3\end{bmatrix}$$

Iz rezultata razberemo, da je normalni vektor enak $\vec{n} = \frac{1}{2}\begin{bmatrix}1\\1\end{bmatrix}$, parameter b (ang., bias) pa $b = -\frac{3}{2}$. Hiperravnina, ki najbolje loči pozitiven in negativen razred, ima tedaj enačbo:

$$Ax + By + b = 0$$
$$\frac{1}{2}x + \frac{1}{2}y - \frac{3}{2} = 0$$
$$x + y - 3 = 0$$

Po metodi podpornih vektorjev smo torej izračunali enačbo ločilne hiperravnine, ki je enolično določena s parametroma $\vec{n} = \frac{1}{2}\begin{bmatrix}1\\1\end{bmatrix}$ in $b = -\frac{3}{2}$. Sedaj lahko za poljuben primerek napovemo, v kateri razred spada. To storimo tako, da novi primerek najprej preslikamo v vektor $\vec{x}$ vektorskega prostora $\mathbb{R}^2$, nato pa preverimo, če pade pod ločilno hiperravnino ali nad njo:

$$y_i = sign(\vec{n}\vec{x} + b)$$

Zgornjo funkcijo imenujemo odločitvena funkcija (ang., decision function).

2.1.3 Linearna ločljivost učne množice

Metoda podpornih vektorjev v svoji osnovni obliki predpostavlja linearno ločljivost primerkov preslikanih v vektorski prostor $\mathbb{R}^n$. Na prvi pogled se to zdi ostra omejitev, vendar obstaja preprosta rešitev: če vektoji niso linearno ločljivi v osnovnem prostoru $\mathbb{R}^n$, jih preslikamo v prostor dovolj visoke dimenzije (ang., feature space) [16], kjer postanejo linearno ločljivi.

2.1.4 SVM za več kot dva razreda

Metoda podpornih vektorjev v svoji osnovni obliki deluje nad natanko dvema razredoma: pozitivnim, ki ga označimo s +1, in negativnim, ki ga označimo z -1. V praksi pogosto naletimo na problem, ko se želimo naučiti klasificirati tudi primerke, ki lahko pripadajo več kot dvema razredoma.

Problem lahko rešimo tako, da ga razdelimo na več binarnih problemov, kakršne zna rešiti osnovni SVM. To lahko storimo na več načinov [17, 18]:

1. **en-proti-vsem** (ang., one-vs-all) je pristop, ko prvemu razredu priredimo oznako +1, vsem ostalim pa -1. Tako dobimo binarni problem, ki se ga naučimo z osnovnim SVM-jem. Postopek ponovimo za vsak razred in si zapomnimo enačbe ločilnih hiperravnin. Nov primerek nato postavljamo v vlogo različnih razredov in ga klasificiramo v tistega, v katerem je dobil največ točk. Učili smo se K-krat, kjer je K število razredov.

2. **vsak-proti-vsakemu** (ang., all-vs-all ali one-vs-one) je pristop, ko iz učne množice jemljemo le po dva in dva razreda. Tako dobimo več manjših binarnih problemov, ki se jih naučimo z osnovnim SVM-jem. Postopek ponovimo za vsak par in si zapomnimo ločilne hiperravnine. Nov primerek nato testiramo proti vsakemu razredu in ga klasificiramo v tistega, v katerega je padel največkrat. Postopku pravimo glasovanje (ang., voting). Učili smo se $\binom{K}{2}$-krat, kjer je K število razredov.

Kateri pristop je boljši, je težko reči, ker so rezultati zelo podobni [19, 20]. Tudi kar se tiče hitrosti izvajanja, sta oba pristopa primerljiva, čeprav se zdi pristop vsak-proti-vsakemu veliko počasnejši, ker se namesto K-krat učimo kar $\binom{K}{2}$-krat. Vendar se učenje pri pristopu vsak-proti-vsakemu izvaja na mnogo manjši podmnožici primerkov, kot pri pristopu en-proti-vsem.

2.2 Naučena hierarhija delov

Naučena hierarhija delov (ang., learnt-hiearchy-of-parts, znano pod kratico lHoP) [10, 11, 21] je algoritem računalniškega vida za detekcijo in kategorizacijo objektov na

sliki. Algoritem je sestavljen iz dveh delov:

1. učenje večnivojskega besedišča

2. prevedba dane slike v naučeno besedišče

V prvem koraku na podlagi analize množice slik izgradimo hierarhičen model delcev, "besedišče", pri katerem se delci nižjega nivoja povezujejo v vedno bolj kompleksne delce višjega nivoja, kot je prikazano na Sliki 2.3. Učenje besedišča je podoben proces, kot ga doživi slehernik v prvih letih svojega življenja, ko se uči govoriti: najprej spozna razne šume, ki jih sčasoma poveže v foneme, foneme v zloge, sledijo kratke besede, dolge besede, nato stavki. Enako se algoritem pomika vedno na višji nivo.

V drugem koraku uporabimo abecedo, ki smo se jo naučili v prvem koraku, da z njo opišemo vsebino poljubne slike. Učenje abecede načeloma izvedemo le enkrat, potem pa opišemo poljubno mnogo slik. Dobrodošla je lastnost algoritma, da se uči tudi med prevedbo dane slike v naučeno besedišče. Sčasoma tako izboljšujemo stopnjo učenosti algoritma lHoP.

2.2.1 Učenje večnivojskega besedišča

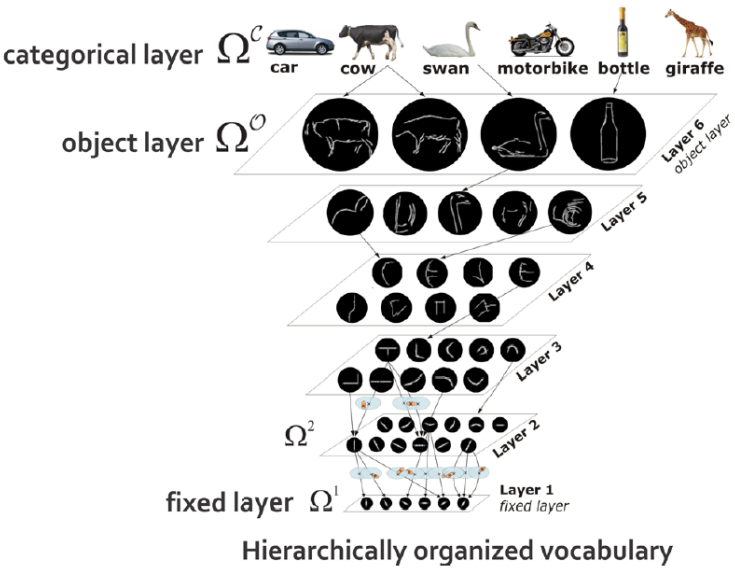
Učenje večnivojskega besedišča pomeni izgradnjo drevesa, kot ga vidimo na Sliki 2.3. Drevo matematično opišemo s parametrom $\mathcal{L}_n$ in množico vseh delcev $\mathcal{P}_i^n$:

$\mathcal{P}_i^n$ pomeni i-ti delec n-tega nivoja, ki ga opišemo z njegovo oznako $\mathcal{P}_i^n$, orientacijo α_i, masnim centrom (x_i, y_i), in seznamom oznak in parametrov njemu podrejenih delcev $\{\mathcal{P}_j^{n-1}, \alpha_j, (x_j, y_j), (\sigma_{xj}, \sigma_{yj})\}_j$:

$$\{\mathcal{P}_i^n, \alpha_i, (x_i, y_i), \{\mathcal{P}_j^{n-1}, \alpha_j, (x_j, y_j), (\sigma_{xj}, \sigma_{yj})\}_j\}_i \tag{2.6}$$

Seznam podrejenih delcev poleg svoje oznake $\mathcal{P}_j^{n-1}$, relativne orientacije α_j in relativne pozicije (x_j, y_j) (relativno glede na $\mathcal{P}_n^i$) vsebuje še dovoljeno odstopanje pozicije poddelca v smeri x (σ_{xj}) in y (σ_{yj}).

$\mathcal{L}_n$ je oznaka za n-ti nivo (ang., level) v hierarhiji. Čeprav je graf na Sliki 2.3 že enolično opisan z množico vseh delcev $\mathcal{P}_i^n$, algoritem naučene hierarhije delov



Slika 2.3: Hierarhija delcev, kakršno izgradi algoritem lHoP. Spodnji nivo je fiksen, imenujemo ga banka filtrov. Običajno ga sestavlja šest ali osem Gaborovih filtrov [22] (v tem primeru šest). Gaborove filtre si lahko predstavljamo kot črtice pod različnimi koti, v našem primeru jih sukamo za 60°. Vir: [23].

(lHoP) izgradi tudi množico seznamov povezav *Links* za posamezni nivo $\mathcal{L}_n$. *Links* za posamezni delec n-tega nivoja pove, s katerimi delci v naslednjem višjem nivoju je povezan. Tako je $Links(\mathcal{P}_i^n)$ seznam vseh delcev $\mathcal{P}_k^{n+1}$ iz nivoja $\mathcal{L}_{n+1}$, katerih poddelec je $\mathcal{P}_i^n$.

Za izgradnjo grafa na Sliki 2.3 je potrebna velika množica slik, nad katero izvedemo učenje. Podroben opis algoritma in njegovo psevdokodo si radovedni bralec lahko pogleda v [21]. V nadaljevanju bomo predpostavili, da imamo abecedo (predstavljeno z omenjenim grafom) že izgrajeno, torej da poznamo tako množico vseh delcev $\mathcal{P}_i^n$, kot tudi množico vseh povezav *Links*.

2.2.2 Prevedba dane slike v naučeno besedišče

Z oznako π_{ik}^n označimo k-to detektirano realizacijo delca $\mathcal{P}_i^n$ v dani sliki. Opišemo jo z oznako delca, kateremu pripada ($\mathcal{P}_i^n$), kotom, pod katerim smo jo detektirali

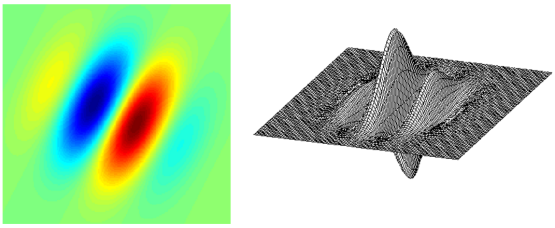
(α_k) in pozicijo v sliki, kjer je bila detektirana (x_k, y_k):

$$\pi_{ik}^n = \{\mathcal{P}_i^n, \alpha_k, (x_k, y_k)\}$$

Z oznako Λ_n pa označimo množico seznamov povezav za posamezni nivo. Λ_n za posamezno realizacijo n-tega nivoja pove, na podlagi katerih vseh lokacij slike smo ga prepoznali. Tako je $\Lambda_n(\pi_{ik}^n)$ seznam vseh pikslov (x_q, y_q), ki ustrezajo lokacijam tistih realizacij nižjih nivojev $\pi_{iq}^{\leq n-1}$, iz katerih je zgrajena nova realizacija π_{ik}^n.

Opisati dano sliko z abecedo delcev pomeni definirati njeno množico realizacij $\{\pi_{ik}^n\}_{n,i,k}$ vseh delcev na vseh nivojih in množico povezav Λ_n za vse nivoje.

Začeti moramo od spodaj, z Gaborovimi filtri za detekcijo robov. Prvi korak je, da s posameznim Gaborovim filtrom velikosti 11x11 pikslov prefiltriramo sliko in si zapomnimo koordinate pikslov, kjer je bil odziv močan. Koordinate močnih pikslov posameznega filtra shranimo v seznam $\Lambda_1(\pi_{ik}^1)$. Množica Λ_1 je edina, ki jo določimo neposredno na podlagi fizičnih rezultatov filtriranja slike, zato jo imenujemo *nivo slike* (ang., image layer). Vsako nadaljnjo plast Λ_n izračunamo rekurzivno iz prejšnje plasti Λ_{n-1} kot je opisano v Algoritmu 1.



Slika 2.4: Primer Gaborovega filtra za detekcijo robov. Tvori ga sinusoida pomnožena z 2D Gaussovim filtrom. Močan odziv ima na robovih. Vir: [24, 25].

Poudariti je potrebno, da je Algoritem 1, ki je originalno predstavljen v [21], zavoljo jasnosti in razumljivosti tukaj poenostavljen, zato se ne ujema povsem z izvorno verzijo.

Da prevedemo dano sliko v naučeno besedišče, moramo Algoritem 1 zaporedno pognati za vsak nivo grafa, dokler ne pridemo do vrhnjega nivoja. Rezultat poganjanja algoritma je množica vseh detektiranih realizacij, najpomembnejša informacija pa se nahaja na najvišjem nivoju, ki ga imenujemo tudi kategorični

Algorithm 1 Iskanje realizacij delcev za stopnjo višje plasti

INPUT: $\{\pi_{ik}^{n-1}, \Lambda_{n-1}\}_k$
OUTPUT: $\{\pi_{ik}^{n}, \Lambda_{n}\}_k$
for all π_{ik}^{n-1} **do**
  rotiraj okolico za $-\alpha_k$
  // poišči še ostale realizacije poddelcev mojega naddelca
  for all $\mathcal{P}_i^n$ iz $Links(\mathcal{P}_i^{n-1})$ **do**
    if najdem še ostale poddelce (π_{ik}^{n-1}) od $\mathcal{P}_i^n$ v okolici **then**
      dodaj π_{ik}^n v rezultat
      $\Lambda_n(\pi_{ik}^n)$ = unija vseh $\Lambda_{n-1}(poddelec)$ za vsak poddelec
    end if
  end for
  izloči prekrivanja (ang., local inhibition)
end for
return $\{\pi_{ik}^{n}, \Lambda_{n}\}_k$

nivo (ang., categorical layer). Na tem nivoju nam vsaka realizacija π_{ik}^n predstavlja posamezno kategorijo, npr. avto, kravo, laboda, ... (glej Sliko 2.3). V zapisu kategorije imamo shranjeno tudi lokacijo, kje v sliki se le-ta nahaja (x_k, y_k), pod kakšnim kotom je prikazana (α_i) in kateri pomembni piksli jo sestavljajo $(\Lambda_n(\pi_{ik}^n))$. Torej imamo lokaliziran semantičen opis slike.

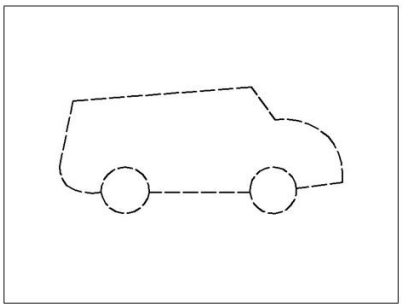
2.3 Opis slike s histogramom kompozicij

Histogram kompozicij (ang., Histogram of Compositions, znano pod kratico HoC) je deskriptor slike, ki s pomočjo histogramov opisuje prikazane objekte. [10, 11].

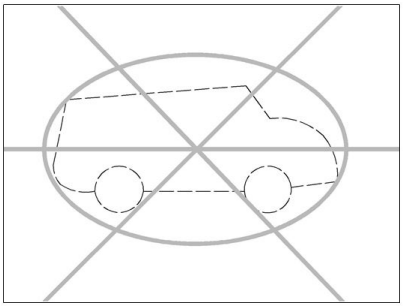
Izračun deskriptorja HoC temelji na vmesnem rezultatu algoritma naučene hierarhije delov (lHoP). Kot vhod namreč pričakuje seznam detektiranih realizacij delcev drugega nivoja (glej Poglavje 2.2):

$$\{\pi_{ik}^2\}_k$$

Grafično je ta seznam prikazan na Sliki 2.7.



Slika 2.5: Detektirane realizacije delcev drugega nivoja $\{\pi^2_{ik}\}_k$ so preproste strukture.

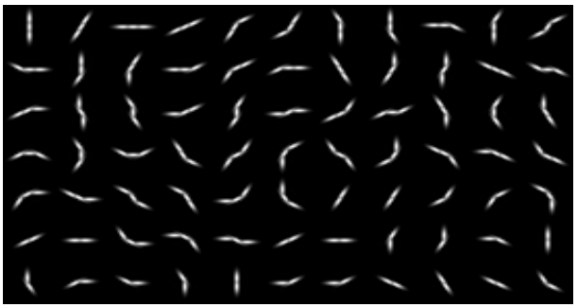


Slika 2.6: Delitev vhodne slike na 12 con. Za vsako cono izračunamo svoj histogram.

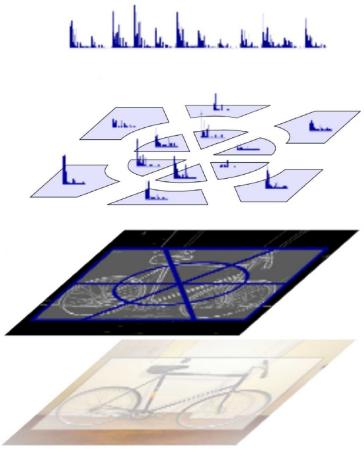
Deskriptor HoC vhodno množico realizacij delcev razdeli na 12 con, kot kaže Slika 2.6. Za vsako cono preštejemo, kolikokrat se je delec $\mathcal{P}^2_i$ realiziral v njej in ustvarimo histogram; dobimo torej 12 histogramov, za vsako cono po enega. Dolžina vsakega izmed 12-ih histogramov je enaka številu vozlišč 2. nivoja lHoP grafa, kar je natanko enako številu delcev drugega nivoja:

$$N = |\{\mathcal{P}^2_i\}_i|$$

Deskriptor slike, ki se imenuje histogram kompozicij (HoC), dobimo tako, da v vektor dolžine $12N$ zložimo histograme vseh dvanajstih con.



Slika 2.7: Primer množice vseh delcev drugega nivoja $\mathcal{P}_i^2$. Za ta primer velja $N = |\{\mathcal{P}_i^2\}_i| = 77$, torej bo HoC deskriptor dolžine $12 \cdot 77 = 924$. Vir: [11].



Slika 2.8: Grafični prikaz izračuna deskriptorja HoC. Vir: [10].

2.3.1 Uporaba deskriptorja HoC

Če imamo na voljo množico slik objektov s pripadajočimi kategorijami (učna množica), se sama od sebe ponuja možnost uporabe metode podpornih vektorjev.

Deskriptor HoC uporabimo v prvem koraku "preslikava primerkov v vektorki prostor $\mathbb{R}^n$"(glej Poglavje 2.1). Terminu *primerek* tedaj ustreza posamezna slika, njen deskriptor HoC pa predstavlja preslikavo v vektorski prostor $\mathbb{R}^{924}$. Od tu naprej je preprosto: SVM izvede drugi korak "izračun ločilne hiperravnine"in se s tem nauči klasifikacije. Tedaj zna klasificirati poljubno sliko v enega izmed naučenih razredov.

2.4 Preverjanje hipotez algoritma lHoP z deskriptorjem HoC

Algoritem naučene hierarhije delov (lHoP) nam izračuna množico realizacij delcev na vseh nivojih $\{\pi_{ik}^n\}_{n,i,k}$. Zanimiv je predvsem najvišji nivo (kategorični nivo), kjer posamezni delec ustreza kar kategoriji (npr. avto, krava, labod). Označimo ta nivo z N. Tedaj je množica vseh kategorij kar:

$$\{\pi_{ik}^N\}_{i,k} \tag{2.7}$$

V praksi se pokaže, da algoritem naučene hierarhije delov ne deluje najbolje. Sicer pravilno zazna veliko večino kategorij, ki so na sliki, vendar pogosto konstruira tudi kategorije, ki jih v resnici ni na sliki (ang., false positives) [10]. Množica opisana z enačbo (2.7) torej poleg pravilnih elementov vsebuje tudi napačne. Razlog za pojavitev *halucinacij* se skriva v razgibanih teksturah, v katerih lHoP nekako najde elemente za tvorbo kategorij, ki jih naše oko ne zazna. Tega se z nastavitvijo preprostih pragov ne da filtrirati, ker so odzivi tekstur premočni.

K problemu lahko pristopimo s preverjanjem hipotez o kategorijah, ki nam jih predlaga algoritem lHoP, preden jih razglasimo za resnične [10]. Če se hipoteza o neki kategoriji izkaže za nepravilno, jo izbrišemo iz množice rešitev.

Hipotezo za dano kategorijo preverimo tako, da na območju realizacij njenega delca π_{ik}^N izračunamo še deskriptor HoC in ga klasificiramo s SVM-jem. Če se klasifikacija, ki jo predlaga algoritem lHoP ujema s tisto, ki jo predlaga SVM, hipotezo potrdimo, sicer jo ovržemo. Pojavita se dve težavi:

1. iskanje območja v sliki, kjer se nahaja kategorija
2. SVM je potrebno naučiti klasificiranja

Iskanje območja v sliki, kjer se nahaja kategorija, je trivialno, saj so vsi potrebni podatki (tj. slikovne lokacije realizacij poddelcev in kazalci nanje) shranjeni v rezultatu algoritma lHoP. Izračunamo ga na podlagi njihovih maksimumov in minimumov pozicij in dobimo:

$$\{\pi_{ik}^N, (x, y, w, h)\}_{i,k} \tag{2.8}$$

kjer parametri x, y, w in h določajo lego in velikost območja (ang., bounding-box). Nad tem območjem izvedemo delitev slike na 12 con, izračun histogramov in končno izračun deskriptorja HoC.

SVM je potrebno naučiti klasificiranja na labelirani učni množici **preden** ga lahko uporabimo za preverjanje hipotez. Pri tem moramo zagotoviti, da je učenje potekalo nad množico vseh kategorij, ki se nahajajo na najvišjem nivoju lHoP drevesa, sicer se bo klasifikacija razlikovala. SVM mora biti najmanj toliko razgledan, kot lHoP, lahko je pa tudi bolj.

Rezultati kažejo, da se ob uporabi preverjanja hipotez, rezultati klasifikacije izboljšajo povprečno za 5% [10].

Poglavje 3

Sistem ViCoS Eye

ViCoS Eye je sistem, ki so ga razvili v Laboratoriju za umetne vizualne spoznavne sisteme na Fakulteti za računalništvo in informatiko, in je namenjen razpoznavanju objektov. Avtorji sistema so se pri načrtovanju in implementaciji držali treh pomembnih zahtev [7]:

1. izpostavitev funkcij sistema navzven v obliki spletne storitve
2. zmožnost hitre obravnave več sto klicev spletne storitve na sekundo
3. skalabilnost s stališča strojne opreme (porazdeljeni sistemi)

3.1 Arhitektura sistema

3.1.1 Spletna storitev

Predpostavka (1) je relativno preprosta za implementacijo. Postavljen je spletni strežnik in na njem izpostavljena spletna storitev, ki na vhodu sprejme našo sliko z metapodatki, vrne pa tekstovni rezultat klasifikacije.

3.1.2 Zaledje

Arhitektura zalednega sistema ViCoS Eye je dvonivojska:

1. podsistem za porazdeljeno strojno učenje

2. podsistem za klasifikacijo

Nivoja se med seboj bistveno razlikujeta. Podsistem za učenje je namenjen strojnemu učenju na učnih primerih, podsistem za klasifikacijo pa je namenjen uporabi naučenega znanja v namen klasifikacije. Učenje je računsko izjemno zahtevna operacija, ki je ne moremo izvajati realno časovno, medtem ko je sama klasifikacija bistveno hitrejša in jo lahko izvajamo realno časovno.

3.2 Porazdeljeno razpoznavanje – tehnološki vidik

Strojno učenje je izjemno zahteven proces, tako z vidika procesiranja, kot dostopanja do diska. Podsistem za porazdeljeno strojno učenje je zato zasnovan za porazdeljeni, paralelni način delovanja. Uporablja paradigmo MapReduce v implementaciji Apache Hadoop, ki si ju bomo ogledali v sledečih dveh podpoglavjih.

3.2.1 MapReduce

MapReduce je ideja, kako avtomatizirano (torej brez programerjevega neposrednega programiranja niti in zaklepov) sprocesirati veliko količino podatkov paralelno na množici procesorjev (gruča vozlišč). Paradigma MapReduce si iz funkcijskega programiranja izposodi dva koncepta [26]. Funkcija *Map* procesira vhodne podatke in jih pretvori v množico vmesnih parov (key, value). Funkcija *Reduce* pravilno združi vmesne pare z enakim ključem. S tako definiranima funkcijama lahko MapReduce avtomatizira vzporedno izvajanje programa, saj sta tako funkcija *Map* kot *Reduce* neodvisni od stanja sistema in lahko nad neprekrivajočimi se podatki tečeta v poljubno mnogo instancah. Vhodni podatki so lahko v poljubnem formatu (ang., application specific).

Programer svoj algoritem določi v funkcijah *Map* in *Reduce*. Predstavljati si mora, kot da funkcija *Map* za vhod dobi množico vseh vhodnih podatkov in

jih mora pretvoriti v množico parov $(key_1, value_1)$, imenujmo jih 'vmesni pari'. Podobno si mora predstavjati, kot da funkcija *Reduce* za vhod dobi posamezen ključ key_1 ter vse vrednosti s tem ključem iz množice vmesnih parov $values_1$ in jih mora združiti v en sam par $(key_2, value_2)$.

Zaželimo si, na primer, prešteti, kolikokrat se pojavi posamezna ocena 1, 2, ..., 10 pri vseh študentih na svetu. Recimo, da imamo seznam vseh ocen, ki jih je kdajkoli dobil katerikoli študent na svetu, shranjen v tekstovni datoteki. Funkciji *Map* in *Reduce* bi tedaj bili:

```
function Map(Document input) {
     // input: velka tekstovna datoteka s seznamom ocen
     // output: množica parov (key1=ocena, value1=pojavitve)
     for each ocena in input
          Emit(ocena, 1);
}
function Reduce(int key1,  int[] values1) {
     // input: key1=ocena, values1=vse pojavitve za ta ključ
     // Opomba: values1 je v tem preprostem primeru polje samih enic
     // output: agregiran par (key2=ocena, value2=vsota pojavitev)
     key2 = key1;
     int count = 0;
     for n in values1
          count += n;
     Emit(key2, count);
}
```

Čar paradigme MapReduce je v tem, da programer implementira omenjeni dve funkciji tako, kot da se bo *Map* klicala le enkrat za cel dokument, *Reduce* pa le enkrat za vsak različen ključ key_1. O vzporednosti mu ni bilo potrebno razmišljati, čeprav nam je nevede ravnokar definiral, kako paralelizirati njegov algoritem.

Na prvi pogled se nam zdi, da je primer z računanjem pojavitev posamezne ocene 1,2, ..., 10 na celem svetu malce pretiran, prevelik. Zdi se nam, da je podatkov enostavno preveč. V odgovor omenimo, da je Google uporabil paradigmo

MapReduce za popolno prenovo indeksiranja svetovnega spleta [27, 28].

MapReduce je zgolj paradigma, za katero pa obstaja mnogo različnih implementacij. V sistemu ViCoS Eye je uporabljena implementacija Apache Hadoop.

3.2.2 Apache Hadoop

Apache Hadoop [29, 30] je odprtokodna javanska implementacija paradigme MapReduce, prvič implementirana leta 2005, ki sta jo avtorja zasnovala po zgledu Googlove implementacije [31] omenjene paradigme. Implementacija vključuje štiri pomembne module:

1. porazdeljeni datotečni sistem (ang., Hadoop Distributed File System)
2. programski modul MapReduce (ang., Hadoop MapReduce)
3. upravljalec virov (ang., Hadoop Yet-Another-Resource-Negotiator)
4. skupni del (ang., Hadoop Common)

Porazdeljeni datotečni sistem

Porazdeljeni datotečni sistem (ang., Hadoop Distributed File System, znan tudi pod kratico HDFS) je napreden datotečni sistem specializiran za shranjevanje ekstremno številnih in/ali enormnih datotek (npr. učne množice ali datoteke velike več terabajtov) na gruči majhnih poceni vozlišč. Od ostalih porazdeljenih datotečnih sistemov se HDFS loči po tem, da zmore visoko stopnjo prilagajanja na odpovedi posameznih vozlišč [32]. To je dobrodošla lastnost predvsem ob dejstvu, da HDFS pogosto združuje stotine ali celo tisoče vozlišč, pri čemer se verjetnost, da bo vsaj eno vozlišče odpovedalo, povečuje z vsakim novim vozliščem.

Datotečni sistem HDFS se tudi zaveda lokacije, na kateri je shranjena posamezna datoteka, in zna to informacijo uporabiti pri procesiranju podatkov (ang., Rack Awareness). Tako procesorjem dostavi tiste podatke, ki so jim fizično najbližje, s čimer zmanjša t. im. interni promet (ang., backbone traffic).

HDFS deluje hierarhično, eno vozlišče uporabi za glavno (ang., Name Node), ki vodi evidenco nad vsemi ostalimi podrejenimi vozlišči (ang., Data Nodes) [32]. Uporabniku ponudi enovit pogled na prostor za shranjevanje, čeprav vsako datoteko interno razdeli na več blokov, ki so lahko shranjeni na fizično različnih vozliščih. Podatek o dejanski lokaciji posameznih blokov se nahaja v glavnem vozlišču, ki tudi izdaja navodila podrejenim vozliščem, kam in kako naj shranijo posamezni blok. Dobrodošlo je dejstvo, da HDFS shrani posamezne bloke v večih kopijah, s čimer zagotovi visoko stopnjo varnosti podatkov pred izgubo. Uporabnik za komunikacijo z glavnim vozliščem uporabi prilagojen Uporabniški protokol (Client Protocol), ki bazira na protokolu TCP/IP. Glavno vozlišče nato komunicira s svojimi podrejenimi vozlišči v posebnem HDFS protokolu (Data Node Protocol), ki prav tako bazira na protokolu TCP/IP.

Programski modul MapReduce

Programski modul MapReduce sestoji iz enega glavnega vozlišča (ang., JobTracker) in iz mnogih podrejenih vozlišč (ang., TaskTrackers). Uporabnik zahtevo za izvedbo MapReduce posla (ang MapReduce Job) pošlje JobTrackerju, ki porazdeli delo med najprimernejše TaskTrackerje. Na izbiro najprimernejšega TaskTrackerja izmed vseh dejavnikov najbolj vpliva fizična razdalja med shranjenimi podatki, ki jih je treba obdelati, in procesorji, ki opravijo računanje. Informacijo o lokaciji podatkov hrani HDFS.

Če se v nekem trenutku sredi procesiranja zgodi, da nek TaskTracker odpove, ga nadomesti naslednji najbližji TaskTracker, če pa se zgodi, da odpove JobTracker, je običajno potrebno izračunati vse od začetka. Izjema so novejše verzije Apache Hadoopa, ki podpirajo shranjevanje vmesnih kontrolnih točk, na podlagi katerih se lahko procesiranje nadaljuje tudi po okrevanju iz morebitne odpovedi JobTrackerja. Apache Hadoop zato velja za sistem z visoko stopnjo odpornosti na odpoved posameznega dela strojne opreme.

Vozlišče tipa TaskTracker ima tipično štiri okolja, v katerih lahko izvaja funkciji Map in Reduce. Ta okolja so med seboj ločena, da se v primeru odpovedi enega ne izničijo kalkulacije vseh štirih, ampak le od tistega, v katerem se je

napaka zgodila.

Stanje vozlišč se stalno preverja s t. im. srčnim utripom (ang., heartbeat), ki ga morajo vsa TaskTracker vozlišča pošiljati JobTrackerju. Ob odpovedi vozlišča JobTracker zazna izpad srčnega utripa, zato se ne odloči za dodelitev procesiranja temu vozlišču. Po okrevanju poškodovano vozlišče zopet pošilja JobTrackerju srčni utrip in JobTracker ga vzame v zakup pri razdeljevanju dela.

Upravljalec virov

Upravljalec virov (ang., Hadoop Yet-Another-Resource-Negotiator, znan tudi pod kratico YARN) je visokonivojska programska knjižnica, ki jo programski modul MapReduce uporablja za upravljanje porazdeljenih virov, npr. procesorjev in pomnilnikov. YARN poskrbi za optimalno razvrščanje zahtevkov za vire ter nadzoruje njihovo izvajanje. [33]

Skupni del

Skupni del (ang., Hadoop Common) je nabor javanskih funkcij, ki jih uporabljajo vsi ostali Hadoop moduli. Vključuje na primer podporo za avtorizacijo uporabnikov in kriptiranje podatkov.

Za ilustracijo reda velikosti podatkov, ki se shranjujejo in preračunavajo na konkretnih sistemih Apache Hadoop, omenimo podjetje Facebook. Leta 2010 je imel na datotečnem sistemu tipa HDFS shranjenih 21PB (peta bajtov, 10^{15}), leta 2012 pa že 100PB. Zaposleni v podjetnju trdijo, da se količina podatkov *dnevno* poveča približno za dva in pol petabajta [34]. Za tolikšno količino podatkov uporabljajo sicer mnogo vozlišč (približno 2000), vendar posamezno vozlišče samo zase ni nič posebnega (8 jedrni procesor, 32 GB delovnega pomnilnika, 12TB velik disk). V tem se kaže pozitivna stran sistema Apache Hadoop, ki zna delovati z manj zanesljivo strojno opremo.

3.3 Klasifikacija – tehnološki vidik

Podsistem za klasifikacijo uporablja tehnologijo Storm, ki omogoča paralelno realnočasovno procesiranje podatkov.

3.3.1 Storm

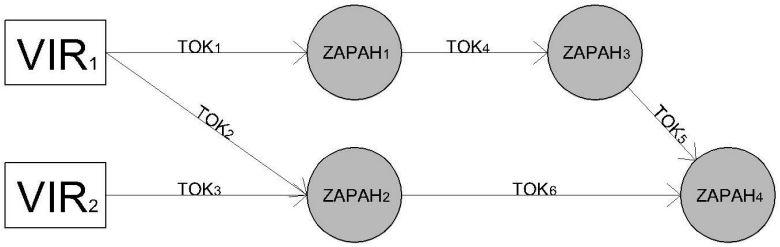
Storm [35] je priljubljena odprtokodna rešitev (ang., framework), napisana v Javi, ki na intuitiven način omogoča uporabniku, da določi topologijo sistema, kakršna njegovi problemski domeni najbolj ustreza. Na prvi pogled je arhitektura Storm podobna arhitekturi Apache Hadoop, vendar so razlike velike. Opazimo na primer, da se posel na Apache Hadoopu (MapReduce Job) v nekem trenutku izvede do konca in zaključi, medtem ko je arhitektura Storm predvidena za kontinuirano procesiranje.

Arhitektura Storm temelji na štirih abstrakcijah, katerih konkretne implementacije mora določiti uporabnik, za vse ostalo poskrbi arhitektura Storm v ozadju:

1. vir (ang., spout)
2. tok (ang., steram)
3. zapah (ang., bolt)
4. topologija (ang., topology)

Vir

Vir (ang., spout) je Javanski razred, ki iz poljubnega sistema (npr. vrsta, ang. queue) v poljubnem formatu pridobiva podatke, katere nato preoblikuje v terke



Slika 3.1: Shematični prikaz preproste topologije (celoten graf) z dvema viroma (pravokotnika) in štirimi zapahi (krogi). Tokovi (puščice) predstavljajo povezave med vozlišči grafa topologije. Viri nenehno proizvajajo tok terk in ga pošiljajo do zapahov, kot jim predpisuje topologija. Posamezni zapah absorbira tok terk in izvede kalkulacije, nato lahko oblikuje poljuben nov tok terk, ki ga pošlje do naslednjih zapahov. Razlika med virom in zapahom je v tem, da vir na vhodu sprejema poljubno vrsto podatkov (npr. sporočila iz JMS vrste ali datoteke iz datotečnega sistema), zapah pa vedno sprejme le tok terk. Poleg tega vir izvaja le preproste kalkulacije, ki so potrebne za proizvajanje toka terk, medtem ko zapahi izvajajo tudi procesorsko zahtevne operacije.

in jih pošilja zapahom kot tok.

```
class MySpout extends Spout {
      @Override
      public void nextTuple() {
            sporočilo = preberiSporočiloIzVrste();
            terka = preoblikujVTerko(sporočilo);
            Emit(terka);
      }
}
```

Tok

Tok (ang., stream) je vedno sestavljen izključno iz terk. Terka lahko vsebuje poljuben objekt, za katerega smo definirali način serializacije. Posamezna vrsta terk je enolično določena z imenom, na podlagi katerega konzumer toka ve, kakšno vsebino terka nosi.

Zapah

Zapah (ang., bolt) je Javanski razred, ki procesira tok terk z določenim imenom in lahko proizvaja nove terke.

```
class MyBolt extends Bolt {
      @Override
      public void execute(Tuple terka) {
            masa = izračunajMasoTerke(terka);
            nova_terka = preoblikujVTerko(masa);
            Emit(nova_terka);
      }
}
```

Topologija

Topologija (ang., topology) je usmerjeni graf, ki določa smer tokov. Definiramo ga s postopnim dodajanjem novih vozlišč (skupaj s pripadajočimi povezavami) v graf. Hkrati določimo, koliko instanc tega vozlišča naj se kreira. Bolj obremenjenim vozliščem določimo več instanc, da je procesiranje hitrejše.

```
Topology topologija = new Topology();
//  priklopimo vir, ki bo proizvajal terke tipa "terke_tip1"
//  dodelimo mu eno instanco, ker ima lahko delo
topologija.setSpout("terka_tip1", new MySpout(), 1);
//  priklopimo zapah, ki bo sprejemal terke tipa "terka_tip1"
//  in proizvajal terke tipa "terka_tip2"
```

```
//  dodelimo mu 10 instanc, ker ima zahtevno delo
topologija.setBolt("terka_tip2", new Bolt(), 10)
        .shuffleGrouping("terka_tip1");
//  priklopimo zapah, ki bo sprejemal terke tipa "terka_tip2"
//  dodelimo mu 2 instanci, ker ima srednje težko delo
topologija.setBolt(null, new MyBolt(), 2)
        .allGrouping("terka_tip2");
```

Storm omogoča več različnih načinov priklopa različnih instanc zapaha na tok. ShuffleGrouping pomeni, da naj se tok porazdeli enakomerno med vse instance zapaha tako, da posamezno terko dobi natanko ena instanca. AllGrouping pomeni, da naj se tok pomnoži tako, da kopijo posamezne terke dobi vsaka instanca zapaha. FiledGrouping pa pomeni, da naj vse terke z istim imenom vedno gredo do iste instance; s tem omogočimo npr. štetje.

Potem, ko smo enkrat implementirali abstrakcije virov, tokov in zapahov ter jih povezali v topologijo, lahko Storm poženemo. In deluje.

3.4 Porazdeljena implementacija ključnih algoritmov sistema ViCoS Eye

3.4.1 Podsistem za strojno učenje

Strojno učenje je računsko zahteven problem, ki pa ga je možno paralelizirati. V ta namen je uporabljena paradigma MapReduce v implementaciji Apache Hadoop (glej Poglavje 3.2).

MapReduce

Glede na to, da se VicosEye želi naučiti klasifikacije po algoritmu preverjanja hipotez z deskriptorjem HoC (glej Poglavje 2.4), je potrebno posamezno sliko iz učne množice pretvoriti v pravilno obliko. Delo, ki ga je potrebno opraviti

pri pretvarjanju, se deli na tri zaporedne korake, ki jih moramo po paradigmi MapReduce predstaviti s tremi pari funkcij Map in Reduce [7, 8]:

1. lHoP korak
2. HoC korak
3. SVM korak

lHoP korak zahteva pretvorbo učne množice slik v lHoP graf. To pomeni, da moramo izračunati množico vseh realizacij delcev $\{\pi_{ik}^n\}_{n,i,k}$ (glej Poglavje 2.2) za posamezno skalo posamezne slike. Problem paraleliziramo kar po slikah, saj je računanje lHoP strukture ene slike popolnoma neodvisno od ostalih slik.

Kot vhod za funkcijo Map_{lHoP} si torej predstavljamo eno sliko s pripadajočim razredom. Ključ je prazen, vrednost pa nosi sliko s pripadajočim razredom:

$$(key = nothing, value = (\mathcal{I}_q, c_q))$$

V skladu z zahtevami paradigme MapReduce vhodni par pretvorimo v nov par, ki pa ima zopet ključ prazen. Za vrednost ima množico vseh detektiranih realizacij lHoP delcev, še vedno s pripadajočim razredom :

$$(key = nothing, value = (\{\pi_{ik}^n\}_{n,i,k}, c_q))$$

Opomba: Ker slike učne množice vedno pripadajo natanko eni kategoriji in ker delec najvišjega sloja lHoP grafa predstavlja ravno eno kategorijo (kategorična plast, glej Poglavje 2.2), sledi, da je možna le ena detektirana realizacija delca na najvišjem nivoju N in ta pripada razredu c_q. Označimo ga lahko s π_{qk}^N. Zato lahko kar cel lHoP graf označimo, kot da pripada razredu c_q. Če bi slika učne množice pripadala več kot enemu razredu, tega ne bi mogli storiti. Iz tega sledi, da mora učna množica nujno vsebovati le slike s po enim razredom naenkrat.

Funkcije Reduce v tem primeru ne potrebujemo, lahko si jo predstavljamo kot identiteto.

HoC korak zahteva izračun območja slike (ang., bounding-box), na katerem je bila detektirana posamezna realizacija delca najvišjega nivoja lHoP grafa, π_{ik}^{N}, ter izračun deskriptorja HoC nad tem območjem (glej Poglavje 2.4).

Kot vhod v funkcijo Map_{HoC} dobimo izhod funkcije Map_{lHoP}, torej množico parov, ki imajo ključe prazne, za vrednost pa imajo seznam struktur lHoP grafa s pripadajočim razredom. Izračunamo območje (ang., bouning-box) in pripadajoči deskriptor HoC. Vrnemo par s praznim ključem in vrednostjo z deskriptorjem HoC ($\mathcal{H}_q$) ter pripadajočim razredom:

$$(key = nothing, value = (\mathcal{H}_q, c_q))$$

Funkcije Reduce tudi v tem primeru ne potrebujemo in si jo lahko predstavljamo kot identiteto.

SVM korak, ki ga izvedemo s tehniko eden-proti-vsem (glej Poglavje 2.1, podpoglavje SVM za več kot dva razreda), zahteva, da primerke enega razreda ločimo od vseh ostalih in izračunamo ločilno hiperravnino. To storimo za primerke vsakega razreda in dobimo SVM model, ki vsebuje M enačb ločilnih hiperravnin, če je M število različnih razredov.

Opazimo, da koraka SVM ne moremo paralelizirati po slikah, saj hkrati potrebuje vektorske predstavitve vseh slik. Lahko pa ga paraleliziramo po razredih. Predstavljajmo si torej, da imamo na vhodu vektorske predstavitve vseh slik učne množice in da jih moramo naučiti modela samo za en razred. Naivni vhod za funkcijo Map_{SVM} je torej par s ključem, ki predstavlja razred (c_q) in vrednostjo, ki predstavlja seznam vseh deskriptorjev HoC za razred c_q:

$$(key = c_q, value = \{\mathcal{H}_{qk}\}_k)$$

Kakovost učenja je odvisna tudi od nekaterih parametrov za SVM (npr. funkcija za transformacijo v višji vektorki prostor, glej Poglavje 2.1), ki morajo biti za različne razrede različno nastavljeni. Ker vnaprej ne vemo, katera funkcija je najboljša, izvedemo trik: funkciji Map_{SVM} na vhod namesto po en par za vsak razred raje vrinemo po T parov za vsak razred. Vsakemu izmed inačic para za

isti razred podtaknemo drugačne SVM parametre. Funkcija Map_{SVM} tako na vhodu dobi:

$$(key = c_q, value = (\{\mathcal{H}_{qk}\}_k, svm\_parametri))$$

Funkcija Map_{SVM} se nauči modela za posamezni razred c_q s parametri podanimi v *svm_parametri*. Vrne nam par s ključem, ki predstavlja razred c_q, in vrednostjo, ki predstavlja naučeni model (m_{qk}) pri teh SVM parametrih, ter oceno kakovosti modela (p_{qk}):

$$(key = c_q, value = (m_{qk}, c_{qk}))$$

Metoda $Reduce_{SVM}$ na vhodu sprejme po rezredih grupirane modele, ki se jih je naučil SVM, skupaj s pripadajočimi ocenami kakovosti modelov:

$$(key = c_q, value = \{m_{qk}, c_{qk}\}_k)$$

in za vsak takšen par vrne par z modelom, ki je ocenjen z najvišjo oceno:

$$(key = c_q, value = m_{qK}, c_{qK}) \;\; ; c_{qK} = \max c_{qk}$$

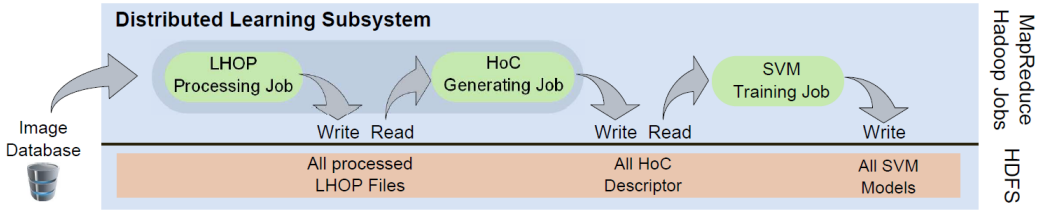
Množica vseh teh parov je končni rezultat strojnega učenja in jo shranimo na disk.

Apache Hadoop

Paradigmo MapReduce implementiramo v programskem jeziku Java, pri čemer uporabimo arhitekturo Apache Hadoop (glej Poglavje 3.2). Za shranjevanje vmesnih rezultatov uporabimo porazdeljen datotečni sistem (HDFS), za procesiranje pa implementiramo tri zaporedne JobTracker-je: prvega za lHoP, drugega za HoC in tretjega za SVM [8].

3.4.2 Podsistem za klasifikacijo

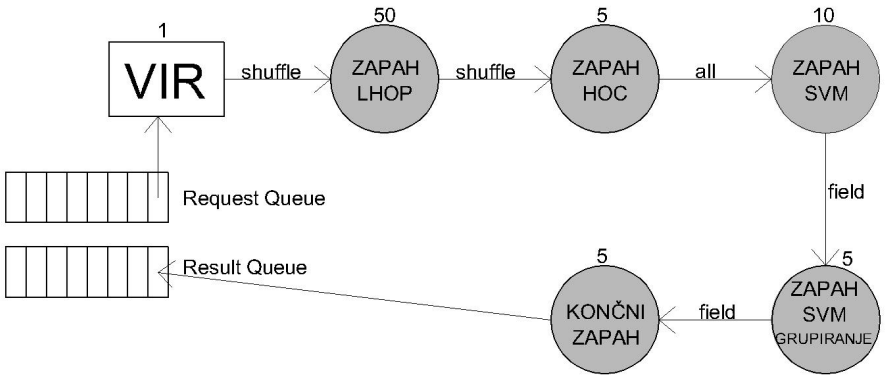
Sistem VicoEye za klasifikacijo uporablja arhitekturo Storm, katere osnove smo si ogledali v Poglavju 3.3. Glede na to, da je VicosEye naučen klasifikacije po algoritmu preverjanja hipotez z deskriptorjem HoC (glej Poglavje 2.4), je potrebno posamezni primerek (vhodno sliko) pretvoriti v pravilno obliko. Delo, ki ga je



Slika 3.2: Prikaz Hadoop implementacije, ki je uporabljena za sistem VicosEye. Vir: [8].

potrebno opraviti pri pretvarjanju, se deli na tri zaporedne korake [36], ki jih v arhitekturi Storm predstavimo z enim virom in petimi zapahi (glej Sliko 3.3):

1. lHoP korak
2. HoC korak
3. SVM korak



Slika 3.3: Prikaz Storm topologije, kot je uporabljena v sistemu VicosEye.

Korak lHoP zahteva pretvorbo vhodne slike v lHoP graf. To pomeni, da moramo izračunati množico vseh realizacij delcev $\{\pi_{ik}^n\}_{n,i,k}$ (glej Poglavje 2.2) za posamezno skalo slike.

Korak lHoP v arhitekturi Storm predstavimo z virom (ang., spout), ki vhodne slike jemlje iz vhodne vrste (ang., queue), in jih pretvarja v tok terk (ang., stream), katere nato pošlje na zapah (ang., bolt) z lHoP implementacijo. Ker je računanje realizacij lHoP delcev procesorsko intenzivna operacija, je zapahu lHoP dodeljenih kar 50 instanc, medtem ko je viru dodeljena le ena sama. Pomembno je, da je vrsta povezave med virom in zapahom lHoP ravno shuffleGrouping, da se tok zahtevkov enakomerno porazdeli med vseh 50 instanc zapaha lHoP, s čimer omogočimo pretočnost sistema.

Zapah lHoP kot rezultat kalkulacij vrne po eno terko vseh realizacij delcev za vsako skalo slike in jo pošlje naprej. Pomembno je, da za vsako skalo kreiramo svojo terko, ker s tem omogočimo več paralelizma v naslednjem koraku.

Korak HoC zahteva izračun območja slike (ang., bounding-box), na katerem je bila detektirana posamezna realizacija delca najvišjega nivoja lHoP grafa, π_{ik}^N, ter izračun deskriptorja HoC nad tem območjem (glej Poglavje 2.4).

Korak HoC v arhitekturi Storm predstavimo z zapahom HoC, ki na vhodu sprejema terke od zapaha lHoP (s povezavo tipa ShuffleGrouping) in iz njih izračuna območja ter pripadajoče deskriptorje HoC. Rezultat kalkulacij zapaha HoC je po ena terka za posamezno realizacijo delca najvišjega nivoja lHoP grafa. Računanje območij in pripadajočih deskriptorjev HoC ni zahtevno opravilo, zato je zapahu HoC dodeljenih skromnih 5 instanc.

Korak SVM zahteva testiranje deskriptorja HoC proti posameznemu razredu in nato klasifikacijo na podlagi točk vseh teh testov (glej Poglavje 2.1, podpoglavje SVM za več kot dva razreda). S pomočjo te klasifikacije bodisi potrdimo bodisi ovržemo hipotezo.

Korak SVM v arhitekturi Storm predstavimo z desetimi instancami zapaha SVM; vsaka instanca testira proti podmnožici razredov, ki je disjunktna glede na ostale instance. Zapah SVM je v topologijo povezan s povezavo tipa allGrouping,

ker mora vsaka instanca dobiti vsako terko. Takoj za zapahom SVM priklopimo še poseben zapah za sestavljanje rezultatov vseh instanc za isto sliko v končno klasifikacijo. Ker je za končno klasifikacijo potrebno sočasno imeti rezultate testiranj proti *vsem* razredom, je ta zapah priključen s povezavo tipa fieldGrouping. Rezultat zapaha je terka, ki vsebuje končno klasifikacijo detektirane realizacije, kot jo predlaga SVM.

Zadnji korak je združevanje hipotez o klasifikaciji, ki nam jih je generiral lHoP, z ovrednotenjem, ki nam ga je prispeval SVM na podlagi deskriptorja HoC. Vse hipoteze za dano sliko, s katerimi se strinjata oba, vrnemo uporabniku kot končni rezultat klasifikacije.

V arhitekturi Storm ta zadnji korak predstavimo z dodatnim zapahom za združevanje. Na vhodu sprejema terke s končnimi ocenami SVM za posamezne lHoP realizacije in realizacije sprejme, če se SVM in lHoP ujemata, oz. zavrže, če se razlikujeta. Ko dobi podatke za vse detektirane realizacije v neki sliki, oblikuje odgovor za uporabnika. V odgovoru navede razrede potrjenih detektiranih realizacij. Ker je za oblikovanje končnega odgovora potrebno sočasno imeti rezultate testiranj proti *vseh* detektiranih realizacij, je ta zapah priključen s povezavo tipa fieldGrouping.

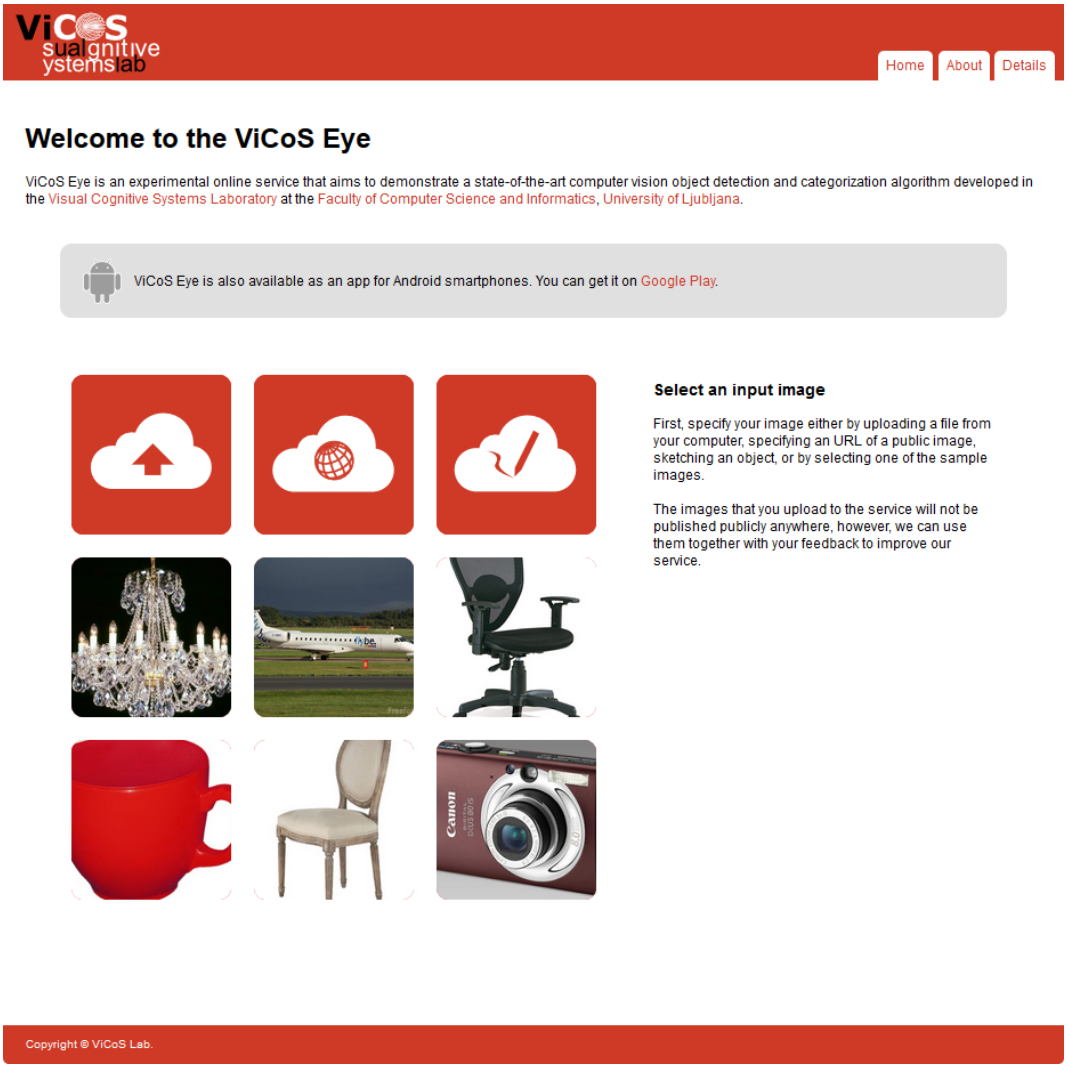
3.4.3 Spletna storitev ViCoS Eye

Spletna storitev ViCoS Eye je implementirana v programskem jeziku Python in uporablja arhitekturo Tornado Web Server [37], ki je specializirana za upravljanje z množico spletnih zahtevkov.

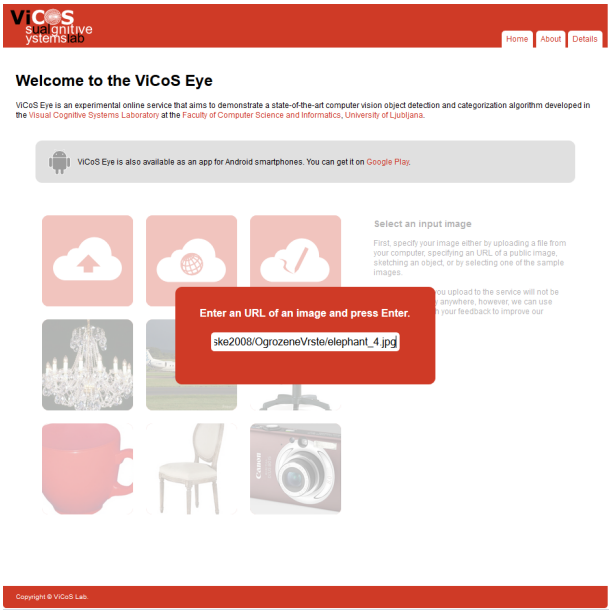
Naloga spletne storitve je glede na kompleksnost celotnega sistema silno preprosta: od uporabnika prejme sliko in jo skupaj z metapodatki pošlje podsistemu za klasifikacijo (glej Poglavje 3.4.2). Nato enostavno posreduje odgovor uporabniku, ki je sprožil zahtevek [8].

Odjemalec spletne storitve je lahko spletni brskalnik (http://eye.vicos.si), razvit je tudi javanski klient za mobilne telefone z operacijskim sistemom android. Oglejmo si nekaj posnetkov zaslona, ki demonstrirajo uporabo spletne storitve

ViCoS Eye preko spletnega brskalnika (glej Slike 3.4–3.7).

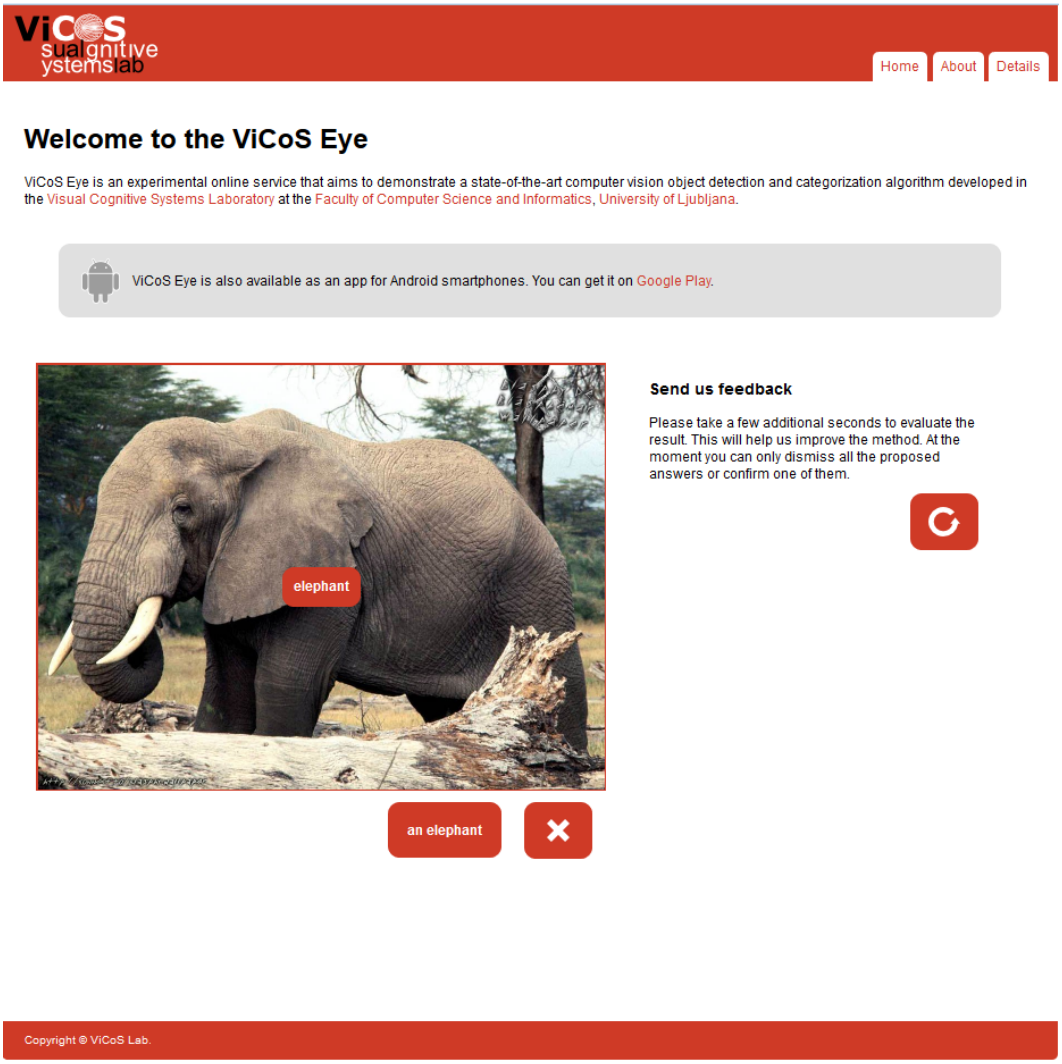


Slika 3.4: Domača stran. Odločimo se lahko za nalaganje slike z diska, spleta ali za skiciranje lastne slike.



Slika 3.5: Nalaganje slike (primerka, ki ga želimo klasificirati) s spleta. Vnesemo URL naslov slike.

Slika 3.6: Čakamo, da podsistem za klasifikacijo opravi svoje delo.



Slika 3.7: Rezultat. Objekt na sliki je bil pravilno klasificiran v kategorijo 'slon' (ang., 'elephant').

Poglavje 4

Predlog izboljšave

V prvem delu diplomskega dela smo si ogledali trenutno arhitekturno in algoritemsko definicijo sistema ViCoS Eye. V drugem delu bomo predstavili naš predlog za izboljšavo klasifikacijske točnosti sistema, afino geometrijsko normalizacijo.

4.1 Pomankljivosti obstoječega sistema

Obstoječi sistem ViCoS Eye na vhodu pričakuje 'lepo' sliko brez afinih in perspektivnih geometrijskih deformacij, kar je omejujoče za uporabnika. Neredko se zgodi, da uporabnik ob zajemu slike npr. mobilni telefon obrne v vodoraven položaj in zajeto sliko pošlje sistemu, brez da bi jo pravilno obrnil. Ker je poslal deformirano sliko (deformacija je v tem primeru rotacija za 90°), je sistem ne bo prepoznal ali jo bo celo klasificiral v napačno kategorijo (glej Sliko 4.1).

V praktičnem delu diplomskega dela smo naslovili omenjeno pomankljivost obstoječega sistema s pristopom, ki je opisan v naslednjem poglavju.

4.2 Geometrijska normalizacija slike

Obstoječi sistem ViCoS Eye izvede algoritem klasifikacije neposredno na vhodni sliki, pri čemer se zanaša, da bo uporabnik sam poskrbel, da bo slika pravilno obrnjena, da ne bo zastrižena, da ne bo prevelika ali premajhna. V praksi je od

Slika 4.1: Rezultat klasifikacije 'lepe' slike objekta (zgoraj) in z rotacijo za 90° deformirane slike (spodaj). Čeprav uporabimo isto sliko v obeh primerih, je sistem v drugem primeru že zaradi preproste rotacije ne prepozna. Predpostavlja se namreč pravilno obrnjen vhod.

uporabnika nemogoče pričakovati tako discipliniran vhod, zato smo se v diplomskem delu posvetili avtomatski normalizaciji vhodne slike. To pomeni, da želimo odpraviti geometrijsko deformacijo, še preden sliko posredujemo obstoječemu klasifikacijskemu algoritmu.

Geometrijske deformacije v splošnem delimo v dve skupini:

1. perspektivne deformacije
2. afine deformacije (translacija, rotacija, skaliranje)

Perspektivne deformacije nastanejo zaradi različnih leč fotoaparatov, različne razdalje med predmetom in fotoaparatom ipd. Skupna lastnost perspektivnih deformacij je, da ne ohranjajo vzporednosti in pravokotnosti, ohranjajo pa črte. V pričujočem diplomskem delu se z normalizacijo te vrste geometrijskih deformacij nismo ukvarjali.

Afine deformacije nastanejo tako, da hkrati vse predmete na sliki premikamo, obračamo in povečujemo. Skupna lastnost afinih transformacij je, da ohranjajo vzporednost, pravokotnost in črte. Afine deformacije smo poskusili odpraviti (normalizirati) s pomočjo algoritma PCA (ang., Principal Component Analysis) [38].

4.2.1 Afina normalizacija s postopkom PCA

Normalizacijo smo zasnovali na podlagi članka [39]. Postavili smo hipotezo, da se afina deformacija odraža v porazdelitvi gradientov v sliki.

Hipoteza 1 *Slika jakosti odvodov je deformirana z enako afino geometrijsko transformacijo kot regija slike, ki vsebuje objekt.*

Slika jakosti odvodov ima običajno obliko **elipse**, normalizirana slika jakosti odvodov pa obliko **enotske krožnice**. Pojem *deformacija* slike jakosti odvodov ustreza tisti afini geometrijski transformaciji, s katero je transformirana enotska krožnica, da je zavzela obliko elipse.

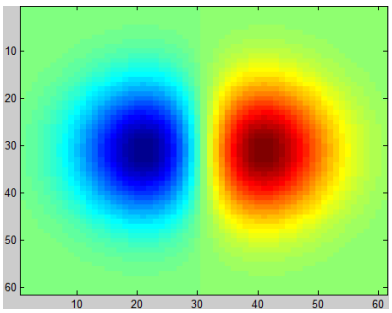
Naš algoritem zato v prvem koraku odvaja sliko in izračuna sliko jakosti odvodov. Da oceni, kakšna elipsa obkroža dobljene točke, izračuna njihovo kovariančno matriko $\mathcal{E}$. Z razcepom SVD nato izračuna, s kakšno translacijo (matrika T), rotacijo (matrika R) in skaliranjem (matrika S) je deformirana. Da izničimo deformacijo slike, celotno sliko v zadnjem koraku transformiramo z inverzno transformacijo in s tem dobimo normalizirano sliko.

Odvajanje slike

Izračunali smo odvode slike v horizontalni (matrika dX) in vertikalni smeri (matrika dY). Odvajanje slike pomeni preprosto izračun odzivov ustreznega filtra v vsakem pikslu slike. Intuitiven in najbolj preprost filter za odvajanje v horizontalni smeri je:

$$F_x = [-1, 0, 1] \tag{4.1}$$

v vertikalni smeri pa ravno transponirana verzija le-tega. Vendar se izkaže, da



Slika 4.2: Vizualizacija mehkega filtra za odvajanje slike v horizontalni smeri. Modro območje predstavlja 'vgreznjeni del' (negativne vrednosti), rdeče območje pa 'izbočeni del' filtra (pozitivne vrednosti). Opazimo lahko podobnost z osnovnim filtrom za odvajanje (enačba 4.1), le da je ta zmehčan v vse smeri in zato manj občutljiv na šum.

je takšen filter preveč občutljiv na šume [40], zato smo se za odvajanje odločili uporabiti odvod Gaussovega filtra. Če po spremenljivki x odvajamo 2D Gaussov filter (predstavljen z enačbo (4.2)), namreč dobimo konvolucijsko jedro, ki zagotavlja odporno 'mehko' odvajanje v horizontalni smeri (glej Sliko 4.2). Dualno

dobimo masko za odvajanje v vertikalni smeri, če 2D Gaussov filter odvajamo po spremenljivki y:

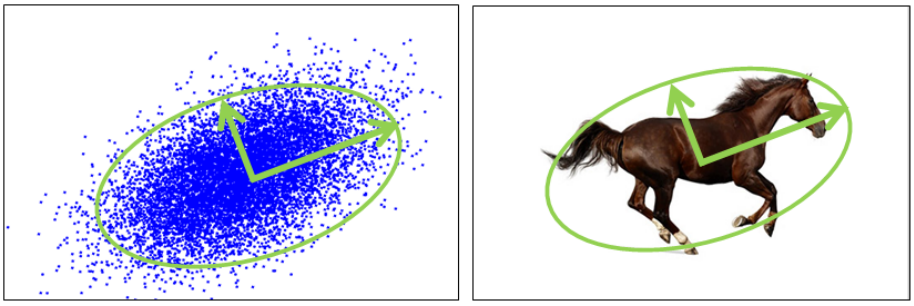
$$f(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{4.2}$$

$$\frac{df(x,y)}{dx} = xe^{-\frac{x^2+y^2}{2\sigma^2}} \tag{4.3}$$

$$\frac{df(x,y)}{dy} = ye^{-\frac{x^2+y^2}{2\sigma^2}} \tag{4.4}$$

Graf jakosti odvodov slike - elipsa

Ko odvajamo sliko v horizontalni in vertikalni smeri, kot rezultat dobimo dve matriki enake velikosti kot originalna slika. Prva vsebuje jakosti odzivov v horizontalni smeri, druga pa v vertikalni smeri. Če raztegnemo matriki v stolpična vektorja dX in dY in narišemo graf jakosti odvodov dY(dX), dobimo oblak točk, ki jih lahko obrišemo z elipso (Slika 4.3).



Slika 4.3: Graf odvisnosti odzivov dY od odzivov dX (levo). Opazimo, da oblika elipse sovpada z obliko objekta na vhodni sliki (desno). Cilj našega algoritma normalizacije je poiskati transformacijo, ki obrne glavno smer elipse (daljša puščica) da kaže v neko dogovorjeno konstantno smer (npr. vodoravno levo) in jo skalira tako, da postane krožnica.

Normaliziran graf jakosti odvodov slike - krožnica

Algoritem PCA zahteva, da izračunamo kovariančno matriko $\mathcal{E}$ vektorjev dX in dY. Dobimo matriko velikosti 2x2 elementa:

$$\mathcal{E} = \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix}$$

To matriko po navodilih algoritma PCA s pomočjo algoritma SVD (ang., Singular Value Decomposition) [41] razdelimo na tri matrike U, Z in V, tako da velja:

$$\mathcal{E} = UZV^T$$

pri čemer je U matrika, ki ima v stolpcih po vrsti zapisane lastne vektorje, Z pa diagonalna matrika, ki ima na diagonali po velikosti zapisane lastne vrednosti. Matrika U je po definiciji ne le ortonormirana, temveč tudi natanko ustreza rotacijski matriki R, s katero je rotirana elipsa, ki opisuje sliko jakosti odvodov slike. Matrika Z je po definiciji kvadrat skalirne matrike, s katero je skalirana množica vseh elementov. Če matriki R in S zapišemo v homogenih koordinatah, velja:

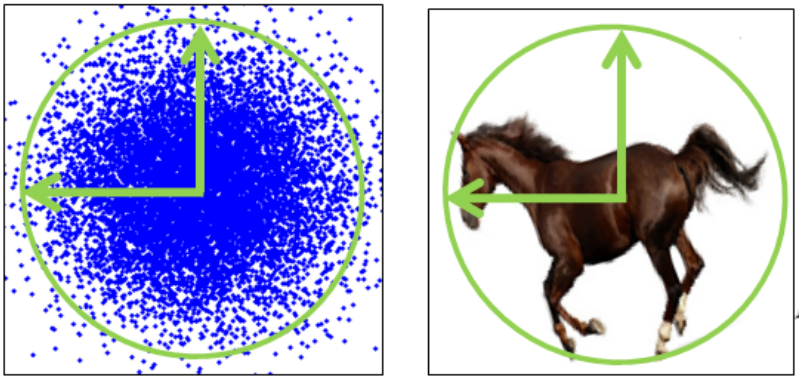
$$R = \begin{bmatrix} & & 0 \\ & \text{U} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.5}$$

$$S = \begin{bmatrix} & & 0 \\ & \sqrt{Z} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

Pozorni moramo biti na to, da se skaliranje in rotacija izvedeta okrog centralnega piksla slike $\vec{x}_c$:

$$\vec{x}_c = \begin{bmatrix} x_c \\ y_c \end{bmatrix} = \begin{bmatrix} E(x) \\ E(y) \end{bmatrix} = \begin{bmatrix} \sum_i x_i dX_i \\ \sum_i y_i dY_i \end{bmatrix} \tag{4.7}$$

Pri čemer $E(x)$ oz. $E(y)$ pomeni matematično upanje koordinate x oz. y, dX_i oz. dY_i pa i-ti element vektorja dX oz. dY, normaliziranega, da ima vsoto 1.



Slika 4.4: Normaliziran graf jakosti odvodov slike (levo) in normalizirana slika (desno). Opazimo, da se je elipsa normalizirala v krožnico. Z rotacijo smo dosegli, da je nekoč daljša puščica elipse sedaj obrnjena v levo, s skaliranjem pa, da se je elipsa 'skrčila' v krožnico in sta puščici sedaj enako dolgi.

Zapišemo lahko translacijsko matriko, s katero je slika premaknjena iz izhodišča, v homogenih koordinatah:

$$T = \begin{bmatrix} 1 & 0 & x_c \\ 0 & 1 & y_c \\ 0 & 0 & 1 \end{bmatrix} \tag{4.8}$$

Algoritem PCA nam je tako izračunal vse tri matrike afinih transformacij, s katerimi je deformiran graf jakosti odvodov slike (elipsa): T, R in S. Združimo vse tri matrike v končno transformacijsko matriko M:

$$M = TRS = TU\sqrt{Z} \tag{4.9}$$

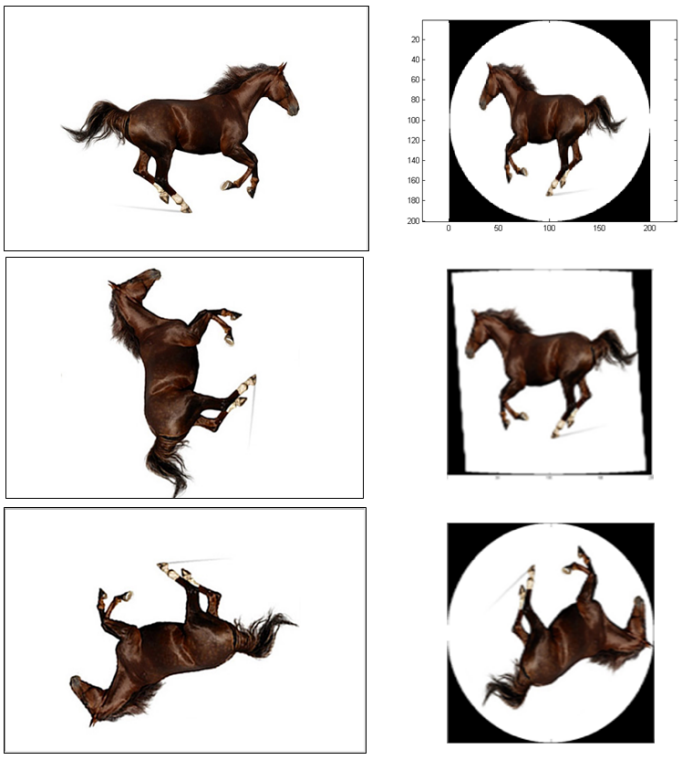
Če želimo izničiti deformiranost grafa jakosti odvodov slike, torej normalizirati elipso v enotsko krožnico, ga moramo pomnožiti ravno z inverzno transformacijo M^{-1}:

$$M^{-1} = (TU\sqrt{Z})^{-1} = \sqrt{Z}^{-1}U^{-1}T^{-1} \tag{4.10}$$

Enačbo (4.10) interpretiramo takole: elipso premaknemo v izhodišče (T^{-1}) ter jo tam rotiramo (U^{-1}) in skaliramo ($\sqrt{Z}^{-1}$).

Hipoteza, ki smo jo predpostavili na začetku, pravi, da je regija slike, ki vsebuje objekt, deformirana z enako transformacijo, kot je deformirana njena

slika jakosti odvodov. To pomeni, da če poznamo inverzno transformacijo, ki pretvori elipso v enotsko krožnico, lahko uporabimo isto inverzno transformacijo tudi nad piksli slike. Rezultat je afino geometrijsko normalizirana slika (glej Sliko 4.4).



Slika 4.5: Normalizirane slike konja v diru pri treh različnih rotacijah. Originalne slike (levi stolpec) so transformirane z matriko M^{-1}, ki izniči deformacije. Dobimo normalizirane slike (desni stolpec). Opazimo, da je normalizirana slika v tretji vrstici za zrcaljenje napačna. Če bi izvedli zrcaljenje po horizontali in vertikali, bi dobili približno enak rezultat kot v zgornjih dveh vrsticah.

Problem zrcaljenja

Na Sliki 4.5 so prikazani primeri normaliziranih slik. Opazimo, da je slika v tretji vrstici nepravilno zrcaljena. Do tega pride, ker je telo konja po dolžini skoraj simetrično (brez bistvene razlike med sliko odvodov glave in repa).

Problem smo naslovili na preprost in intuitiven način: normalizirano sliko smo navidezno razpolovili po vertikali in horizontali, s čimer smo dobili štiri enako velike dele slike, četrtine. Nato smo v vsaki četrtini poiskali, koliko struktur detektira detektor DoG (ang., Difference of Gaussians) [42] (glej Sliko 4.6).

DoG je pogosto uporabljen detektor, ki na podlagi zaporednih glajenj slike detektira signifikantna območja slike. Njegova prednost pred ostalimi detektorji je invariantnost na skalo; detektirati zna tako velike, kot majhne signifikantne dele slike. Nas je zanimalo število detektiranih signifikantnih območij v pozameznem izmed štirih delov slike.

Z zrcaljenjem čez horizontalo, vertikalo ali diagonalo smo poskrbeli, da se največ signifikantnih območij pojavi v zgornjem levem delu slike. S tem smo poskusili odpraviti težavo z nepravilnim zrcaljenjem zaradi simetrične oblike objekta, ki se pojavlja pri našem algoritmu normalizacije.

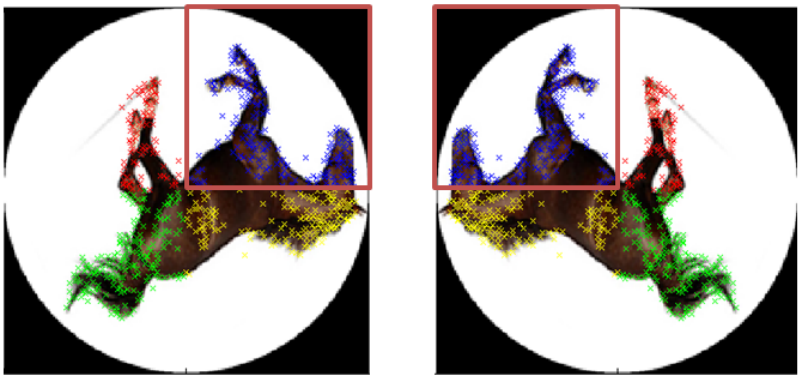
Končni algoritem je sestavljen iz dveh delov: v prvem koraku sliko normaliziramo s pomočjo algoritma PCA, v drugem pa z zrcaljenji poskrbimo, da se najbolj signifikanten del slike nahaja v zgornji levi četrtini slike.

4.2.2 Implementacija

Zavoljo hitrosti smo se odločili implementirati algoritem normalizacije v programskem jeziku C++ [43] s pomočjo knjižnic OpenCV in VlFeat.

Za programski jezik C++ smo se odločili, ker je zanj na voljo množica uporabnih knjižnic tudi za delo s področja procesiranja slik, konkretno OpenCV [44] in VlFeat [45], in ker je prilagojen za delo na višjem nivoju abstrakcije kot izvorni C.

OpenCV [44] je knjižnica za delo s slikami, ki je implementirana v mnogih jezikih (C, C++, Python, Java,...). Njena poglavita prednost je dobro premišljena predstavitev slike z objektom, ki zna sam skrbeti za rezervacijo in sproščanje pomnilnika. Z uporabo takšnega objekta se izognemo puščanju pomnilnika (ang., memory leak), ki bil sicer pereč problem zaradi velike količine podatkov pri pro-



Slika 4.6: Signifikantna območja slike, kot jih je detektiral detektor DoG, različno obarvani za vsako izmed štirih območij slike. Z zrcaljenji poskrbimo, da se največ detekcij pojavi v zgornji levi četrtini slike. Na prvotni normalizirani sliki (leva slika) opazimo, da je največje število detekcij v zgornjem desnem območju, zato izvedemo zrcaljenje preko vertikale in dobimo rezultat, ki ima največje število detekcij zgoraj levo (desna slika).

cesiranju slik. Poleg tega knjižnica OpenCV implementira osnovne operacije nad slikami in matrikami, npr. seštevanje in množenje, ki nam olajšajo delo, in tudi kompleksnejše operacije, npr. glajenje, spreminjanje velikosti, branje, pisanje v standardne formate, brez katerih si procesiranja slik ne moremo predstavljati.

VlFeat [45] je napredna knjižnica za specifične naloge pri procesiranju slik. Ponuja implementacijo kompleksnih in pogosto uporabljenih algoritmov, kot so na primer, DoG, Sift, K-means. Namenjena je za delo z Matlabom ali programskim jezikom C++. Pri implementaciji afine normalizacije slike nam je predvsem prav prišla njena implementacija detektorja DoG in njena sposobnost računanja afine deformacije posameznih regij slike.

Programirali smo torej v programskem jeziku C++, predstavitev slike in osnovne operacije izvajali s pomočjo knjižnice OpenCV, za iskanje afine deformacije slike in detektiranje z detektorjem DoG pa smo se zanesli na knjižnico VlFeat.

4.2.3 ViCoS Eye in geometrijska normalizacija

V sistem ViCoS Eye smo geometrijsko normalizacijo vključili tik pred algoritem klasifikacije. Vhod v naš algoritem je bil tisti del slike, ki ga je uporabnik spletne storitve označil kot območje objekta (ang., bounding box). Izhod našega algoritma in hkrati vhod v obstoječi sistem klasifikacije pa je normalizirana slika fiksne velikosti 200x200 pikslov skupaj z binarno masko, ki jo bomo predstavili v nadaljevanju.

Pozorni smo morali biti na dejstvo, da se originalna slika pri normalizaciji lahko tudi rotira. Težava je v tem, da se pri rotiranju pojavijo nedefinirana črna območja, ki vnesejo nenaravne (umetne) robove v sliko. Ti robovi bi zavedli algoritem klasifikacije, saj bi Gaborovi filtri v njegovi bližini vrnili visoke odzive, čeprav bi le-ti morali biti ničelni. Deskriptor takšne slike bi bil povsem napačen.

Problem smo naslovili tako, da smo poleg normalizirane slike algoritmu poslali še binarno masko, s pomočjo katere je le-ta iz izračunov izključil črne robove slike pri procesiranju slike. Masko smo izračunali tako, da smo z isto inverzno transformacijo kot sliko, transformirali še belo sliko enakih dimenzij.

Poglavje 5

Eksperimentalno vrednotenje izboljšav

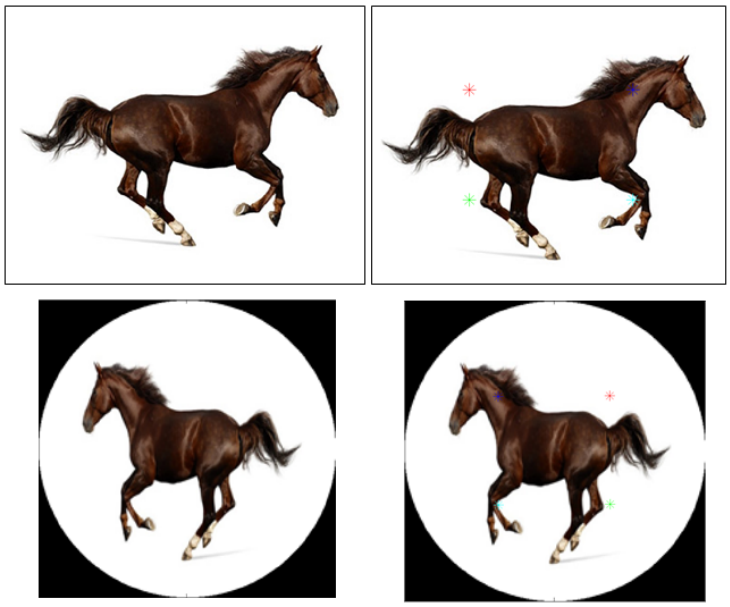
V prejšnjem poglavju smo si ogledali, na kakšen način smo se odločili izboljšati klasifikacijsko moč sistema ViCoS Eye. Predstavili in implementirali smo afino normalizacijo slik. Zanimalo nas je, kako uspešna je naša normalizacija. Najprej smo izvedli test stabilnosti same normalizacije, nato smo preverili še, kakšen je njen vpliv na kakovost klasifikacije sistema ViCoS Eye.

5.1 Test stabilnosti normalizacije

Namen afine normalizacije je, da vhodno sliko pretvori v enako referenčno sliko, četudi je vhodna slika afino deformirana (npr. rotirana, neenakomerno skalirana vzdolž x in y ter premaknjena). Kako učinkovit je naš algoritem pri tem, smo preverili s sledečim eksperimentom.

Vzeli smo vhodno sliko, si na njej izbrali štiri kontrolne točke in vhodno sliko pretvorili v referenčno obliko. Nato smo z enako transformacijo transformirali kontrolne točke. Tako smo dobili originalno sliko s štirimi kontrolnimi točkami, in referenčno sliko s prav tako štirimi kontrolnimi točkami semantično na enakih mestih (glej Sliko 5.1).

Originalno sliko smo tedaj deformirali z rotacijo, enako njene kontrolne točke.



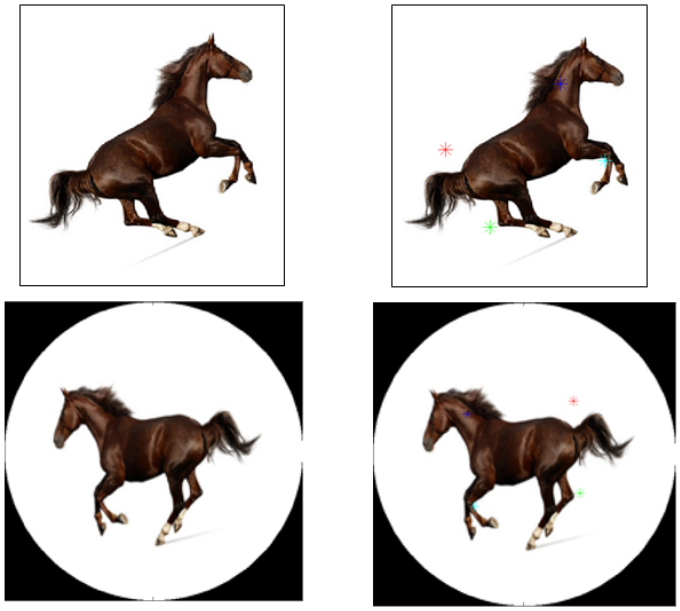
Slika 5.1: Originalni sliki (zgoraj levo) določimo štiri kontrolne točke (zgoraj desno). Nato sliko normaliziramo v referenčno obliko (spodaj levo) in z enako transformacijo transformiramo kontrolne točke (spodaj desno). Opazimo, da so kontrolne točke glede na konja na enakih mestih tako na originalni sliki, kot na referenčni: modra je na vratu, rdeča nad repom, zelena pri zadnjem kolenu in ciano pri sprednjem kolenu.

Nato smo rotirano sliko normalizirali z našim algoritmom in zopet enako transformirali kontrolne točke. Dobili smo torej normalizirano sliko in kontrolne točke na njej (glej Sliko 5.2).

Zanimalo nas je, kolikšne so razdalje med kontrolnimi točkami referenčne slike (Slika 5.1 spodaj desno) in normalizirane rotirane slike (Slika 5.2 spodaj desno) pri različnih kotih. Naša mera različnosti med referenčno in normalizirano rotirano sliko je torej:

$$d_\alpha = \frac{1}{4}\sum_{k=1}^{4}\sqrt{(x_{ref_k} - x_{rot_{k,\alpha}})^2 + (y_{ref} - y_{rot_{k,\alpha}})^2} \tag{5.1}$$

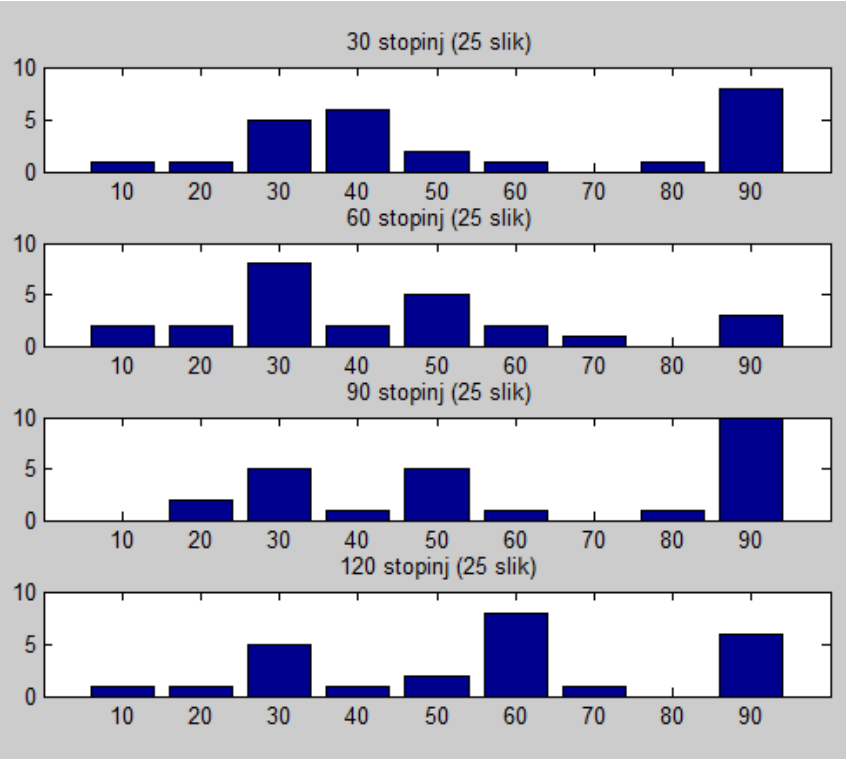
pri čemer je α kot rotacije deformirane slike, d_α mera različnosti dveh normalizacij, x_{ref_k} in y_{ref_k} x oz. y-koordinata k-te kontrolne točke na referenčni sliki ter



Slika 5.2: Z rotacijo za 30 stopinj deformirana slika (zgoraj levo) in pripadajoče kontrolne točke (zgoraj desno). V spodnji vrstici vidimo normalizirano obliko rotirane slike (spodaj levo) in nanjo vrisane kontrolne točke (spodaj desno).

$x_{rot_{k,\alpha}}$ in $y_{rot_{k,\alpha}}$ x oz. y-koordinata k-te kontrolne točke na normalizirani rotirani sliki. Nižja kot je povprečna različnost d_α, bolj so si podobne normalizirane slike, bolje deluje algoritem normalizacije.

Pripravili smo test na 25 slikah, ki smo jih deformirali za 30, 60, 90 in 120 stopinj ter primerjali z njihovimi referenčnimi kontrolnimi točkami. Odločili smo se uporabiti slike na belem ozadju, da smo se izognili problemu umetnih robov, ki bi sicer nastali pri rotaciji. Pri slikah z belim ozadjem lahko preprosto zapolnimo nedefinirana mesta z belo barvo. Dobili smo histogram porazdelitve napak za posamezni kot, kot prikazuje Slika 5.3. Histogrami prikazujejo porazdelitev izmerjenih različnosti d_α, ki smo jih razdelili na devet razredov širine 10 slikovnih pikslov. Izjema je zadnji razred, v katerega razvrstimo vse izmerjene d_α, ki so večje od 80 piklsov, in je zato neomejene širine.

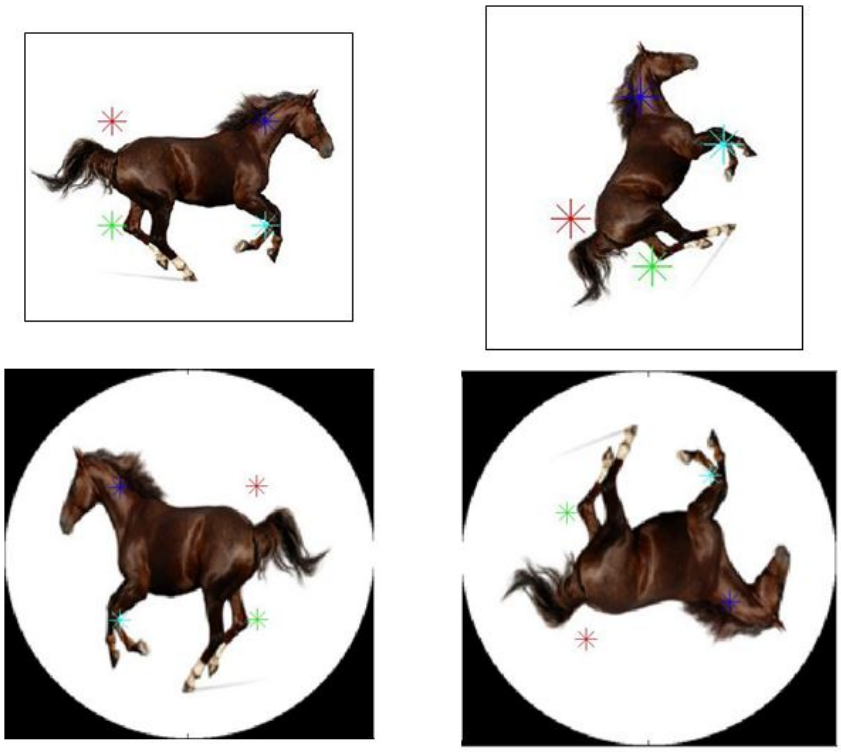


Slika 5.3: Histogram z rezultati testa stabilnosti normalizacije. Oznaka n na vodoravni osi histograma pomeni vrednost d_α med $n-10$ in n piksli. Oznaka 40 tako pomeni, da je d_α med 30 in 40 piksli. Zadnji stolpec z oznako 90 pomeni, da je bila različnost 80 pikslov ali več.

5.1.1 Diskusija rezultatov

Opazimo, da je najvišji stolpec zadnji, torej da je bila različnost večja kot 80 pikslov. Takšen rezultat pripisujemo razliki v zrcaljenju (glej Sliko 5.4). Če imamo dva objekta različno zrcaljena, kontrolne točke namreč ravno preskočijo diagonalo, vertikalo ali horizontalo. Tako se razdalja med *istoležnimi* kontrolnimi točkami močno poveča, čeprav rezultat normalizacije morda niti ni tako slab (normalizirani sliki sta si od daleč podobni). Iz tega sklepamo, da naš algoritem normalizacije kljub naslavljanja tega problema z detektorjem DoG, še vedno ni dovolj učinkovit glede pravilnega zrcaljenja.

Druge najvišje stolpce opazimo pri vrednostih od 30 do 60 pikslov. Glede na



Slika 5.4: Primer visoke vrednosti parametra d_α zaradi nepravilnega zrcaljenja. Izmerjena je povprečna različnost $d_{60} = 103.52px$, zato jo razvrstimo v zadnji stolpec histograma.

to, da je normalizirana slika široka vedno natanko 200x200 pikslov, je 30 pikslov (oz. 11% diagonale slike) dokaj dobra normalizacija.

Test učinkovitosti normalizacije je pokazal, da lahko pričakujemo težave iz domene normalizacije zrcaljenja. Če bi imeli algoritem s popolno normalizacijo zrcaljenja, bi smeli pričakovati odstopanja kontrolnih točk okrog 40 pikslov.

5.2 Test vpliva normalizacije na klasifikacijo

Testiranje vpliva normalizacije na klasifikacijsko točnost sistema ViCoS Eye smo izvedli na testnem strežniku s pomočjo vmesnika, napisanega v skriptnem jeziku Python [46]. Vmesnik omogoča zagon posameznih modulov zaledja testnega

sistema ViCoS Eye in komunikacijo s porazdeljenim datotečnim sistemom.

Z namenom čim lažjega testiranja je testni strežnik v celoti postavljen na arhitekturo Apache Hadoop (glej Poglavje 3.2.2), tako podsistem za učenje, kot podsistem za klasifikacijo.

Odločili smo se za testiranje na svetovno znani anotirani bazi slik Caltech-101 [47,48], za katero že imamo uradne podatke o klasifikacijski točnosti sistema ViCoS Eye. Baza slik Caltech-101 vsebuje 101 kategorijo, za vsako kategorijo pa ima 40 do 800 slik. Vključili smo tudi dodatno kategorijo, ozadje (ang., google-background). Testiranje smo se odločili izvesti nad vsemi 101 kategorijami, vendar le s po 15 učnimi in 15 testnimi slikami za vsako. Odločitev, katerih 15 slik vzamemo za učne in katerih 15 za testne primerke, smo izvedli naključno za pet različnih delitev (ang., splits).

Ker iz strokovnih člankov [11] avtorjev sistema poznamo klasifikacijsko točnost sistema brez naših nadgradenj, smo testiranje brez normalizacije pognali le enkrat, kot kontrolni preizkus. Dobili smo 54.84% klasifikacijsko točnost (razmerje med pravilno klasificiranimi primerki in številom vseh primerkov), kar je v skladu z navedbami članka, kjer so v povprečju dobili dobrih 51%. Zaradi zamudnosti poganjanja testov, smo se odločili izpustiti ponovitvene teste osnovnega sistema in predpostaviti klasifikacijsko točnost 51%.

Izvedli smo tri različne teste klasifikacijske točnosti sistema:

1. **obstoječi sistem** smo najprej testirali brez normalizacije (kontrolni preizkus). Tako učenje, kot predikcijo smo računali neposredno nad originalnimi vhodnimi slikami.

2. **klasični test normalizacije** smo izvedli tako, da smo sistemu tako za učenje, kot za predikcijo na vhod dali normalizirane različice vhodnih slik.

3. **predikcija z deformacijo** pomeni test, ki smo ga izvedli na sledeči način: sistem smo naučili klasifikacije na normaliziranih slikah. Vse slike za predikcijo pa smo najprej afino deformirali - rotirali za 90 stopinj. Tako deformirane smo šele poslali čez naš algoritem normalizacije in od tu v predikcijo.

Tabela 5.1: Uspešnost klasifikacije. Uspešnost osnovnega sistema (*obstoječi sistem*), uspešnost potem, ko smo tako za učenje, kot za predikcijo namesto originalnih slik sistemu na vhod poslali normalizirane slike (*klasični test normalizacije*) in ko smo testirali predikcijo z deformiranimi slikami na znanju iz nedeformiranih (*predikcija z deformacijo*). *predpostavljeno na podlagi uradnih rezultatov

Delitev	obstoječi sistem	klasični test normalizacije	predikcija z deformacijo
1	54.84%	23.99%	13.66%
2	*51.00%	24.64%	15.49%
3	*51.00%	25.75%	14.31%
4	*51.00%	26.99%	14.38%
5	*51.00%	29.35%	14.12%
povpr.	**51.77%**	**26.14%**	**14.39%**

Kot je razvidno iz Tabele 5.1, smo z našo normalizacijo dobili za 25.63 odstotnih točk slabše rezultate, kot smo jih dobili brez normalizacije. Rezultati so torej bistveno slabši.

5.2.1 Diskusija rezultatov

Da bi bolje razumeli, od kod pridejo takšni rezultati, smo za vsako delitev (ang., split) pogledali, katere kategorije so bile bolj in katere manj uspešne pri klasifikaciji (glej Sliki 5.5 in 5.6). S pomočjo omenjenih histogramov smo našli najboljšo in najslabšo kategoriji glede na klasifikacijsko točnost.

Slika 5.5 prikazuje porazdelitev klasifikacijske točnosti pri kontrolnem testu obstoječega sistema brez normalizacije, Slika 5.6 pa pri klasičnem testu normalizacije in pri testu predikcije z normalizacijo. Opazimo zanimiv pojav, da se pri večini histogramov pojavi maksimum pri kategoriji *minaret* (indeks 66). Opazimo tudi, da je klasifikacijska točnost pri vseh kategorijah slabša od kontrolne, tako pri klasičnem testu, kot pri testu predikcije z deformacijo.

Na Sliki 5.7 je prikazan primer kategorije *minaret*, ki je bila klasificirana najbolj natančno. Razlog za tako dobro klasifikacijo je izredna podobnost med nor-

maliziranimi slikami različnih primerkov iste kategorije (srednji stolpec - vse so rotirane podobno). Opazimo, da sta si drugi in tretji stolpec podobna, čeprav sta nastala iz različnih začetnih slik. Zato je klasifikacija za to kategorijo zelo uspešna tudi v testu predikcije z deformacijo.

Na Sliki 5.8 je prikazan primer kategorije *ant*, ki je bila zelo slabo klasificirana. Opazimo, da sta si drugi in tretji stolpec podobna, čeprav sta nastala iz različnih začetnih slik, kar priča o učinkoviti normalizaciji. Razlog za slabo klasifikacijo je v neskladjih med normaliziranimi različicami različnih primerkov iste kategorije (srednji stolpec - vsaka je rotirana drugače).

Na Sliki 5.9 vidimo primer slike letala na enakomernem ozadju in primer na šumnem ozadju. Opazimo, da je v prvem primeru algoritem normalizacije zaznal pravilno elipso, ki lepo obkroža letalo. V drugem primeru opazimo, da se oblika in orientacija elipse ne ujemata z objektom, ker drevesa iz ozadja preveč vplivajo na izračun njene oblike. Poleg šumnega ozadja se nam postavlja tudi vprašanje, ali sta si letali v sliki gradientov sploh dovolj podobni, da bi ju lahko naš algoritem normaliziral podobno.

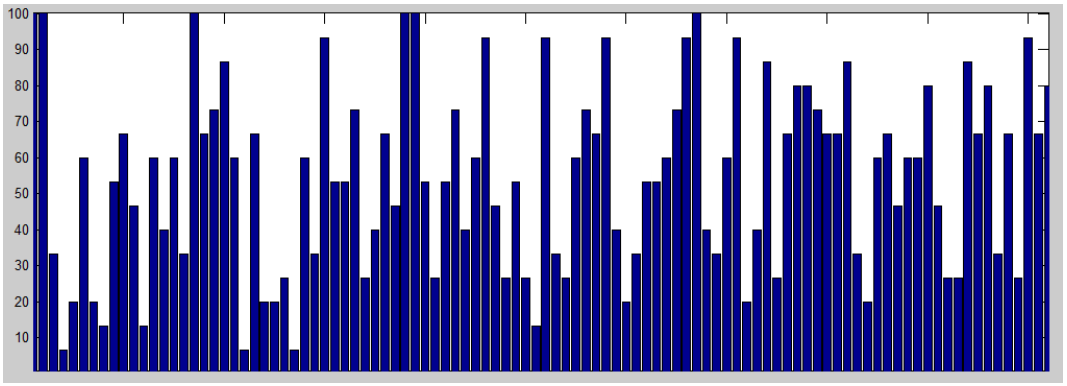
Eksperimentalno raziskovanje je torej pokazalo, da afina normalizacija slike ne prispeva nujno pozitivno h klasifikacijski točnosti sistema za razpoznavanje. Toda zakaj?

Najbolj verjetna razlaga se zdi, da je algoritem normalizacije, ki svoje temelje gradi na sliki odvodov slike, občutljiv na motnje, ki jih prinaša ozadje slike. Zaradi močnih odzivov odvodov v ozadju slike se prepozna napačna oblika objekta (elipsa), iz česar se seveda ne more izračunati pravilna afina normalizacija. Če predpostavimo to razlago, je krivec naš algoritem normalizacije, ki slabo deluje na slikah z izrazitim ozadjem. V tem primeru bi do boljših rezultatov prišli tako, da bi poskušali empirično nastaviti parametre (npr. velikost filtra za odvajanje slike ali velikost območja, kjer se sploh upošteva odvode ipd.), dokler ne bi našli pravega razmerja med ne-upoštevanjem ozadja in posledičnim zanemarjanjem ospredja.

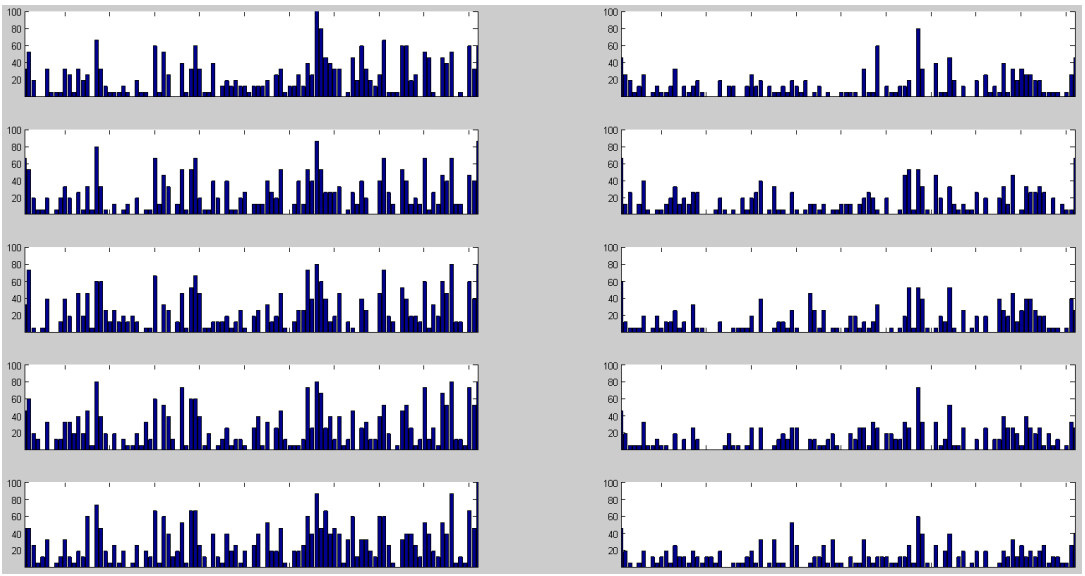
Druga, težje premostljiva razlaga slabših rezultatov pa je ta, da si slike iste kategorije z vidika gradientov niso dovolj podobne. To pomeni, da četudi bi imeli

algoritem, ki vedno zna poiskati pravilno obliko objekta brez napak zaradi motenj iz ozadja slike in sliko vedno normalizira pravilno, ne bi dobili podobnih normaliziranih slik za različne primerke iste kategorije. Predstavljajmo si npr. konja, ki je enkrat v diru z dvignjeno glavo in repom ter pokrčenimi sprednjimi nogami in tega istega konja, ko mirno stoji na mestu, s sklonjeno glavo in spuščenim repom ter iztegnjenimi sprednjimi nogami. Čeprav gre za objekt iste kategorije, v omenjenih dveh položajih zavzema bistveno različne oblike.

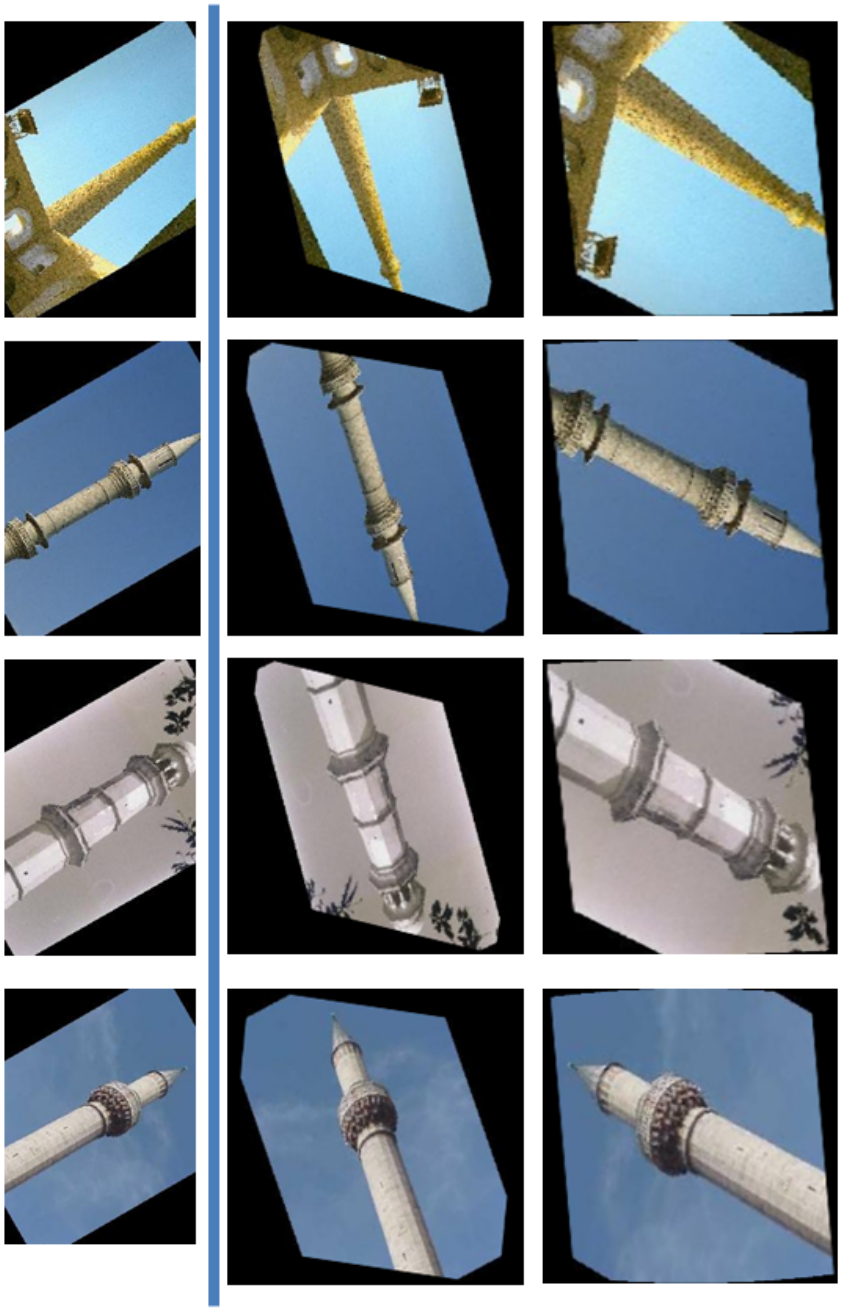
Tretja razlaga pa je sama baza Caltech-101. Vsebuje namreč nenavadno rotirane slike, ki vsebujejo umetne črnine z močnimi robovi, kot lahko opazimo npr. na Sliki 5.7 (celoten levi stolpec) ali na Sliki 5.8 (2. slika prvega stolpca). Ti robovi bistveno vplivajo na sliko odvodov slike. Klasifikacijska točnost bi se zelo verjetno izboljšala, če bi uporabili bolj naravno bazo, brez umetnih robov.



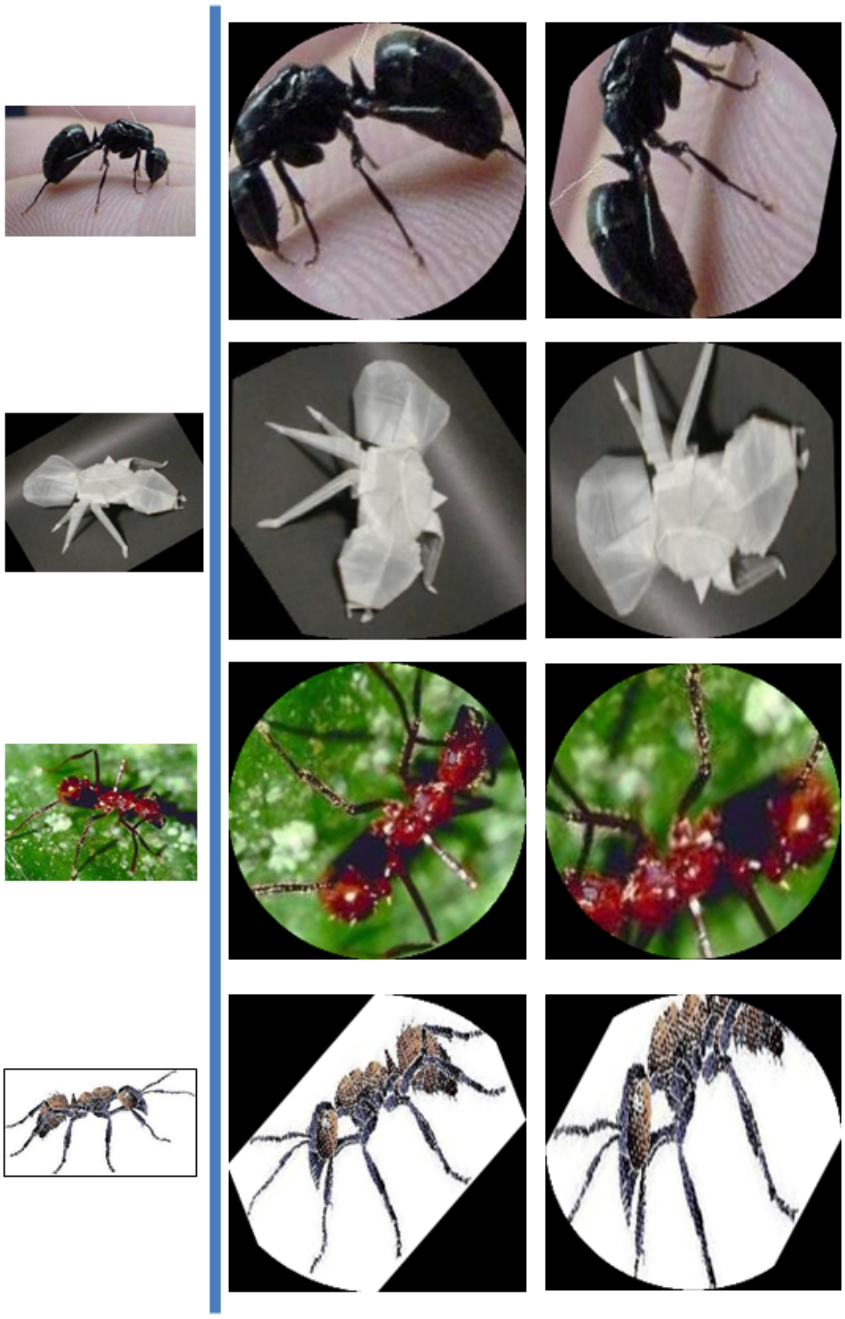
Slika 5.5: Odstotki pravilnih klasifikacij po kategorijah pri kontrolnem testu obstoječega sistema (brez normalizacije). Višina stolpcev pomeni odstotek, ki ga je dana kategorija dosegla, vodoravni položaj stolpca pa pove, za katero kategorijo gre. Kategorije so navedene na x-osi od 1 do 102 po abecednem vrstnem redu: 1=accordion, 2=airplanes, 3=anchor, 4=ant, 5=background-google, 6=barrel, 7=bass, 8=beaver, 9=binocular, 10=bonsai, 11=brain, 12=brontosaurus, 13=buddha, 14=butterfly, 15=camera, 16=cannon, 17=car-side, 18=ceiling-fan, 19=cellphone, 20=chair, 21=chandelier, 22=cougar-body, 23=cougar-face, 24=crab, 25=crayfish, 26=crocodile, 27=crocodile-head, 28=cup, 29=dalmatian, 30=dollar-bill, 31=dolphin, 32=dragonfly, 33=electric-guitar, 34=elephant, 35=emu, 36=euphonium, 37=ewer, 38=faces, 39=faces-easy, 40=ferry, 41=flamingo, 42=flamingo-head, 43=garfield, 44=gerenuk, 45=gramophone, 46=grand-piano, 47=hawksbill, 48=headphone, 49=hedgehog, 50=helicopter, 51=ibis, 52=inline-skate, 53=joshua-tree, 54=kangaroo, 55=ketch, 56=lamp, 57=laptop, 58=leopards, 59=llama, 60=lobster, 61=lotus, 62=mandolin, 63=mayfly, 64=menorah, 65=metronome, 66=minaret, 67=motorbikes, 68=nautilus, 69=octopus, 70=okapi, 71=pagoda, 72=panda, 73=pigeon, 74=pizza, 75=platypus, 76=pyramid, 77=revolver, 78=rhino, 79=rooster, 80=saxophone, 81=schooner, 82=scissors, 83=scorpion, 84=sea-horse, 85=snoopy, 86=soccer-ball, 87=stapler, 88=starfish, 89=stegosaurus, 90=stop-sign, 91=strawberry, 92=sunflower, 93=tick, 94=trilobite, 95=umbrella, 96=watch, 97=water-lilly, 98=wheelchair, 99=wild-cat, 100=windsor-chair, 101=wrench, 102=yin-yang



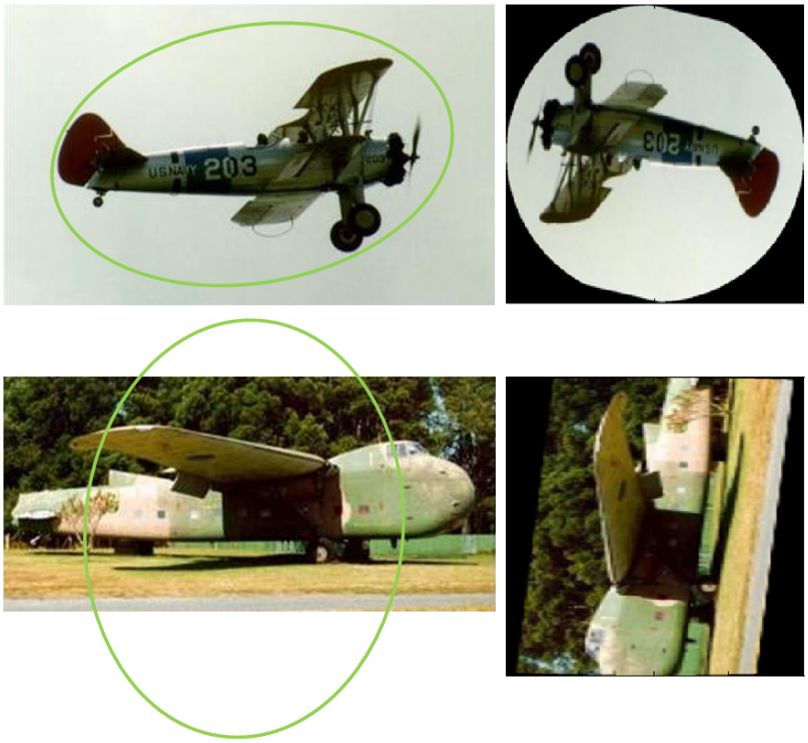
Slika 5.6: Odstotki pravilnih klasifikacij po kategorijah pri klasičnem testu normalizacije (levi stolpec) in pri testu predikcije z deformacijo (desni stolpec). Vsaka vrstica je pridobljena iz različne delitve (ang., split).



Slika 5.7: Primeri iz kategorije *minaret* (indeks 66), ki je bila zelo dobro klasificirana. Originalne slike, kakršne so v bazi Caltech-101 (levi stolpec), skupaj s svojimi normaliziranimi oblikami (srednji stolpec). Desni stolpec prikazuje rezultate normalizacij originalnih slik, ki smo jih pred tem deformirali (tj. rotirali za 90 stopinj v nasprotni smeri urinega kazalca).



Slika 5.8: Primeri iz kategorije *ant* (indeks 4), ki je bila zalo slabo klasificirana. Originalne slike, kakršne so v bazi Caltech-101 (levi stolpec), skupaj s svojimi normaliziranimi oblikami (srednji stolpec). Desni stolpec prikazuje rezultate normalizacij originalnih slik, ki smo jih pred tem deformirali (tj. rotirali za 90 stopinj v nasprotni smeri urinega kazalca).



Slika 5.9: Slab primer zaradi šuma ozadja. V zgornji vrstici vidimo primer normalizacije letala na nešumnem ozadju, v spodnji vrstici pa primer normalizacije letala na izrazito šumnem ozadju.

Poglavje 6

Sklepi

V pričujočem diplomskem delu smo si najprej ogledali najpomembnejše algoritme računalniškega vida, na katerih sloni klasifikacijski sistem ViCoS Eye. Ogledali smo si algoritem strojnega učenja, metodo podpornih vektorjev. Sledile so osnove algoritma detekcije objektov v sliki, naučena hierarhija delov. Potem smo se seznanili še z deskriptorjem histogram kompozicij, ki v navezi z algoritmom naučene hierarhije delov in metode podpornih vektorjev omogoča klasifikacijsko sposobnost celotnega sistema.

Obdelali in opisali smo arhitekturo, potrebno za uspešno delovanje tako kompleksnega in zahtevnega sistema računalniškega vida. Seznanili smo se z dvema zelo različnima arhitekturama, ki ju potrebujemo za delovanje sistema ViCoS Eye, Apache Hadoop in Storm. Posebno pozornost smo namenili vprašanju učinkovite vpetosti algoritmov na arhitekturo.

Opazili smo, da je deskriptor HoC občutljiv na afine deformacije, zato smo predstavili predlog izboljšave, afino geometrijsko normalizacijo slike, ki deluje na podlagi odvodov slike. Implementirali smo jo v programskem jeziku C++ in eksperimentalno ovrednotili njegovo uspešnost normalizacije ter njegov vpliv na klasifikacijsko točnost sistema.

Eksperiment uspešnosti normalizacije smo izvedli tako, da smo merili razdalje med istoležnimi kontrolnimi točkami referenčne slike in normalizirane deformirane slike. Rezultati so pokazali, da je algoritem natančen do zrcaljenja.

Eksperiment vpliva na klasifikacijo smo izvedli na testni verziji sistema. Uporabili smo bazo Caltech-101. Rezultati so pokazali, da je naš algoritem občutljiv na šume iz ozadja, zato je klasifikacijska točnost bistveno padla, iz prejšnjih 52% na 26%.

6.1 Smernice za nadaljnji razvoj

Ob implementaciji afine normalizacije slike smo dobili idejo, da bi bilo koristno nad vsako sliko najprej izvesti segmentacijo, s čimer bi pri izračunu afine deformacije slike lahko učinkovito izločili šume iz ozadja. Ker je omenjena nadgradnja izven okvirov pričujočega diplomskega dela, jo samo omenjamo kot možno smernico za izboljšavo.

Opazili smo, da sistem ViCoS Eye nikjer v procesu učenja klasifikacije ne upošteva intenzitet barvnih kanalov slike. Predvidevamo, da bi se klasifikacijska točnost izboljšala, če bi deskriptor HoC obogatili še z barvno informacijo. Obstajajo namreč objekti, ki jih zgolj na podlagi oblike ni mogoče dovolj diskriminativno razlikovati, npr. pomaranča in rdeča granivka, na podlagi barve pa to ne bi bilo težko. Posebej zanimiv se nam je zdel algoritem "imena barv" [49], vendar je tudi to izven okvirov pričujočega diplomskega dela.

Literatura

[1] Google. (2014, Feb.) Google images. [Online]. Available: http://images.google.com

[2] Y. Wan, X. Liu, J. Bing, and Y. Chen, "Online image classifier learning for Google image search improvement," in *International Conference on Information and Automation*, 2011.

[3] Google. (2014, Feb.) Goggles overview and requirements. [Online]. Available: http://www.google.com/mobile/goggles

[4] I. Inc. (2014, Feb.) Tineye reverse image search. [Online]. Available: http://www.tineye.com

[5] M. 2010. (2014, Feb.) Macroglossa visual search engine. [Online]. Available: http://www.macroglossa.com

[6] L. Judic. (2014, Feb.) Macroglossa's visual search engine fails to meet basic expectations. [Online]. Available: http://searchenginewatch.com/article/2050950/Macroglossas-Visual-Search-Engine-fails-to-meet-basic-expectations

[7] D. Tabernik, L. Čehovin, M. Kristan, M. Boben, and A. Leonardis, "Vicos eye - spletna storitev za kategorizacijo vizualnih objektov," 2013.

[8] ——, "Vicos eye - a webservice for visual object categorization," 2013.

[9] ——. (2013, Aug.) Vicos eye. [Online]. Available: http://www.vicos.si/images/1/1e/Vicos-eye-cvww13-presentation.pdf

[10] D. Tabernik, M. Kristan, M. Boben, and A. Leonardis, "Hypothesis verification with histogram of compositions improves object detection of hierarchical models," 2013.

[11] ——, "Learning statistically relevant edge structure improves low-level visual descriptors," in *International Conference on Pattern Recognition*, 2012.

[12] N. Dalal and B. Triggs, "Histograms of Oriented Gradients for Human Detection," in *Computer Vision and Pattern Recognition*, vol. 1, 2005, pp. 886–893.

[13] Wikipedia. (2013, Dec.) Support vector machine. [Online]. Available: http://en.wikipedia.org/wiki/Support_vector_machine

[14] ——. (2013, Dec.) Metoda podpornih vektorjev. [Online]. Available: http://sl.wikipedia.org/wiki/Metoda_podpornih_vektorjev

[15] D. Ventura. (2013, Dec.) Svm example. [Online]. Available: http://axon.cs.byu.edu/Dan/678/miscellaneous/SVM.example.pdf

[16] M. A. Hearst. (2013, Dec.) Support vector machines. [Online]. Available: http://pages.cs.wisc.edu/ jerryzhu/cs540/handouts/hearst98-SVMtutorial.pdf

[17] E. J. Bredensteiner and K. P. Bennett, "Multicategory Classification by Support Vector Machines," 1999.

[18] H. H. Zhang. (2013, Dec.) Lecture 16: Multiclass support vector machines. [Online]. Available: http://math.arizona.edu/ hzhang/math574m/2013Lect16_msvm.pdf

[19] J. Milgram, M. Cheriet, and R. Sabourin, ""One Against One"or "One Against All": Which One is Better for Handwriting Recognition with SVMs?"

[20] R. Rifkin. (2013, Dec.) Multiclass classification. [Online]. Available: http://www.mit.edu/ 9.520/spring08/Classes/multiclass.pdf

[21] S. Fidler and A. Leonardis, "Towards scalable representations of visual categories: Learning a hierarchy of parts." in *IEEE Computer Vision and Pattern Recognition*, 2007. [Online]. Available: http://vicos.fri.uni-lj.si/data/alesl/cvpr07fidler.pdf

[22] B. R. Oltombi-Diba and J. Miyamichit, "8-5 Edge-Based Segmentation of Textured Images Using Optimally Selected Gabor Filters."

[23] M. Kristan, M. Boben, D. Tabernik, and A. Leonardis, "Adding discriminative power to hierarchical compositional models for object class detection," Jun 2013.

[24] Wikipedia. (2014, Jan.) Gabor filter. [Online]. Available: http://en.wikipedia.org/wiki/Gabor_filter

[25] U. o. P. The Computer Vision Lab at GET. (2014, Jan.) Active stereo vision. [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/TRAPP1/filter.html

[26] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *International Symposium on High-Performance Computer Architecture*, 2007, pp. 13–24.

[27] Wikipedia. (2013, Aug.) Mapreduce. [Online]. Available: http://en.wikipedia.org/wiki/MapReduce

[28] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of The ACM*, vol. 51, pp. 107–113, 2008.

[29] M. A. Bhandarkar, "MapReduce programming with apache Hadoop," in *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, 2010, pp. 1–1.

[30] Wikipedia. (2013, Aug.) Apache hadoop. [Online]. Available: http://en.wikipedia.org/wiki/Apache_Hadoop

[31] J. Berthold, M. Dieterle, and R. Loogen, *Implementing Parallel Google Map-Reduce in Eden*, 2009.

[32] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Symposium on Mass Storage Systems*, 2010.

[33] i. Cloudera. (2013, Dec.) Mr2 and yarn briefly explained. [Online]. Available: http://blog.cloudera.com/blog/2012/10/mr2-and-yarn-briefly-explained/

[34] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *International Conference on Management of Data*, 2010, pp. 1013–1020.

[35] S. Community. (2014, Jan.) Storm, distributed and fault-tolerant realtime computation. [Online]. Available: http://storm-project.net/

[36] D. Tabernik, L. Čehovin, M. Kristan, M. Boben, and A. Leonardis, "A web-service for object detection using hierarchical models," 2013.

[37] T. Community. (2013, Dec.) Tornado web server. [Online]. Available: http://www.tornadoweb.org/en/stable/

[38] Wikipedia. (2014, Jan.) Principal component analysis. [Online]. Available: http://en.wikipedia.org/wiki/Principal_component_analysis

[39] K. Mikolajczyk and C. Schmid, "Scale & Affine Invariant Interest Point Detectors," *International Journal of Computer Vision*, vol. 60, pp. 63–86, 2004.

[40] M. Zucker. (2014, Jan.) filter-slides. [Online]. Available: http://www.swarthmore.edu/NatSci/mzucker1/e27/filter-slides.pdf

[41] Wikipedia. (2014, Jan.) Singular value decomposition. [Online]. Available: http://en.wikipedia.org/wiki/Singular_value_decomposition

[42] W. E. Polakowski, D. A. Cournoyer, S. K. Rogers, M. P. Desimio, D. W. Ruck, J. W. Hoffmeister, and R. A. Raines, "Computer-Aided Breast Cancer Detection and Diagnosis of Masses Using Difference of Gaussians and Derivative-Based Feature Saliency," *IEEE Transactions on Medical Imaging*, vol. 16, pp. 811–819, 1997.

[43] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[44] Itseez. (2014, Jan.) Opencv. [Online]. Available: http://opencv.org/

[45] T. authors of VLFeat. (2014, Jan.) Vlfeat. [Online]. Available: http://www.vlfeat.org

[46] P. S. Foundation. (2014, Jan.) Python programming language. [Online]. Available: http://www.python.org/

[47] M. A. Fei-Fei Li and M. A. Ranzato. (2014, Jan.) Caltech101. [Online]. Available: http://www.vision.caltech.edu/Image_Datasets/Caltech101/

[48] L. Fei-Fei, R. Fergus, and P. Perona, "Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories," in *Computer Vision and Pattern Recognition*, 2004.

[49] J. van de Weijer, C. Schmid, J. J. Verbeek, and D. Larlus, "Learning Color Names for Real-World Applications," *IEEE Transactions on Image Processing*, vol. 18, pp. 1512–1523, 2009.

[50] V. Lab. (2013, Aug.) Vicos eye. [Online]. Available: http://www.vicos.si/Research/VicosEye

[51] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer, "Apache hadoop goes realtime at Facebook," pp. 1071–1080, 2011.

[52] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, 1995.

[53] F. Shahbaz Khan, R. Anwer, J. van de Weijer, A. Bagdanov, M. Vanrell, and A. Lopez, "Color attributes for object detection," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, June 2012, pp. 3306–3313.

[54] J. van de Weijer team. (2014, Jan.) Joost van de weijer. [Online]. Available: http://lear.inrialpes.fr/people/vandeweijer/color_names.html

[55] P. Kay, "Basic color terms: their universality and evolution," 1969.

[56] N. Petkov and D. o. C. S. I. S. M.B. Wieling, University of Groningen. (2014, Jan.) Gabor filter for image processing and computer vision. [Online]. Available: http://matlabserver.cs.rug.nl/cgi-bin/matweb.exe

[57] Wikipedia. (2014, Jan.) C++. [Online]. Available: http://sl.wikipedia.org/wiki/C%2B%2B

[58] Itseez. (2014, Jan.) Opencv. [Online]. Available: http://opencv.org/documentation.html