ŽILINSKÁ UNIVERZITA V ŽILINE

FAKULTA RIADENIA A INFORMATIKY

DIPLOMOVÁ PRÁCA

Študijný program:

Informačné systémy – Spracovanie dát

BC. FRANTIŠEK KAJÁNEK

Paralelná implementácia extraktora príznakov vhodného pre detekciu objektov pomocou Adaboostu

Vedúci práce: Ing. Peter Tarábek, PhD.
Registračné číslo: 282/2016
Ministerské číslo práce: 28360420172282
Žilina, 2017

ŽILINSKÁ UNIVERZITA V ŽILINE

FAKULTA RIADENIA A INFORMATIKY

DIPLOMOVÁ PRÁCA

Študijný program:

Informačné systémy – Spracovanie dát

BC. FRANTIŠEK KAJÁNEK

Paralelná implementácia extraktora príznakov vhodného pre detekciu objektov pomocou Adaboostu

Žilinská univerzita v Žiline
Fakulta riadenia a informatiky
Katedra matematických metód a operačnej analýzy
Žilina, 2017

Original zadania – scan

ČESTNÉ VYHLÁSENIE

Čestne prehlasujem, že som prácu vypracoval samostatne s využitím dostupnej literatúry a vlastných vedomostí. Všetky zdroje použité v diplomovej práci som uviedol v súlade s predpismi.

Súhlasím so zverejnením práce a jej výsledkov.

V Žiline, dňa 14.4.2017 František Kajánek

POĎAKOVANIE

Chcel by som poďakovať vedúcemu diplomovej práce Ing. Petrovi Tarábkovi, PhD. za odbornú pomoc, pripomienky a usmerňovanie pri tvorbe práce.

ABSTRAKT

KAJÁNEK, František: *Paralelná implementácia extraktora príznakov vhodného pre detekciu objektov pomocou AdaBoostu* [diplomová práca] - Žilinská univerzita v Žiline. Fakulta riadenia a informatiky; Katedra matematický metód a operačnej analýzy – Vedúci: Ing. Peter Tarábek, PhD. - Stupeň odbornej kvalifikácie: Inžinier v odbore Informačné systémy – Analýza dát. FRI ŽU v Žiline, 2017. 61s

Cieľom diplomovej práce je implementácia vlastného deskriptora, použiteľného na detekciu objektov v spojení s AdaBoostom. Súčasťou práce je oboznámenie s dôležitosťou rýchlych výpočtov a s deskriptormi a klasifikátormi. Taktiež sa rozoberajú deskriptory, ktoré už boli použité s AdaBoostom. Taktiež je spomenutý kaskádový prístup ku klasifikácií a Viola-Jones kaskáda. Následne sa práca venuje výberu implementácie AdaBoostu a vlastným pokusom s detekciou obrazu. Samostatná kapitola sa venuje hĺbkovej analýze implementácie Viola-Jones kaskády a jej používania. Práca pokračuje rozborom deskriptorov, výberu jedného (SHOG), ktorý sa implementuje a vyhodnotí. Na koniec sa zhrnú možnosti zlepšenia detekcie objektov.

Kľúčové slová: deskriptor, klasifikátor, AdaBoost, HOG, SHOG, OpenCV, Viola-Jones, kaskáda

ABSTRACT

KAJANEK, Frantisek: *Parallel Implementation of Feature Descriptor for Object Detection Using Adaboost* [Master thesis] - The University of Žilina. Faculty of Management Science and Informatics; Department of Mathematical Methods and Operations Research. Tutor - Ing. Peter Tarábek, PhD. Academic qualification level: Master in field Information Systems – Data Analysis. FRI ŽU in Žilina, 2017. 61p

The goal of the thesis is to implement our own descriptor, capable of detecting objects in connection with AdaBoost. The thesis starts out by explaining the importance of high performance algorithms. Then it deals with descriptors, classifiers and AdaBoost. Next, we have a portfolio of descriptors that have been used in connection with AdaBoost in the past. The thesis also mentions cascade classification and introduces the reader to the Viola-Jones cascade and the OpenCV library. We then analyse existing AdaBoost implementations and also explain our first trials with object detection. As a follow-up on that, the thesis thoroughly analysis the OpenCV Viola-Jones cascade implementation, and how it is used. After that, we compare existing descriptors according to our criteria and then choose one which we implement and test. The thesis concludes with summarizing possible ways of improving object detection.

Keywords: descriptor, classifier, AdaBoost, HOG, SHOG, OpenCV, Viola-Jones, cascade

OBSAH

ÚVOD

Počítačové videnie je obor, ktorý sa zaoberá získavaním informácií z digitálneho obrazu a videa. Jeho snahou je analyzovať, navrhovať a implementovať činnosti, ktoré dokáže vykonávať ľudský zrakový systém. Pochopenie obrazu pre tento obor znamená transformáciu digitálneho obrazu na popisy sveta, s ktorými dokáže pracovať iný proces počítača. Táto oblasť využíva rôzne techniky z geometrie, fyziky, štatistiky a teórie strojového učenia. Široké spektrum typov dát, ktoré využíva počítačové videnie, medzi ktoré patria napríklad videá, multi-dimenzionálne dáta zo skenerov alebo aj len jednoduché fotografie spôsobuj, že práca s obrazom je veľmi náročná po teoretickej aj praktickej stránke kvôli obsiahlosti potrebných algoritmov.

Napriek tomu v dnešnej dobe je potrebné počítačové videnie čoraz viac kvôli rozvoju informačných technológií a množstvu dát, ktoré treba spracovať. Aj toto je jeden z podnetov pre túto prácu. Cieľom práce je nájsť efektívny a paralelizovateľný deskriptor obrazu použiteľný v kaskádovom klasifikátore s použitím AdaBoostu. Pre tento účel je potrebné vykonať hlbokú analýzu súčasného stavu, aby bolo možné popísať a zdokumentovať už dostupné možnosti s cieľom zhodnotenia ich prínosu a poznatkov.

Veľkým problémom hľadania takéhoto deskriptora je široký záber celej tématiky. I keď samotná implementácia nemusí byť priveľmi komplikovaná, aj len jednoduché otestovanie v reálnom svete je veľmi náročné, kvôli všetkým prostriedkom na to potrebným. Táto práca sa taktiež snaží oboznámiť čitateľa so všetkými krokmi potrebnými na umožnenie takéhoto testovania.

V teoretickej časti sa čitateľ oboznámi so všetkými potrebnými prvkami potrebnými na pochopenie logiky umožňujúcej detekciu v obraze. Najskôr je rozobraná dôležitosť rýchlosti, a následne vysvetlené kľúčové pojmy ako deskriptor alebo klasifikátor, v spojení s algoritmom strojového učenia AdaBoost. Taktiež sa práca venuje rozboru niekoľkých deskriptorov a poznatkov, ktoré priniesli pre vývoj počítačového videnia a zrýchlenie detekcie v reálnom svete. Nakoniec je v krátkosti popísaná knižnica OpenCV, ktorá umožňuje uľahčený prístup do komplexného oboru počítačového videnia.

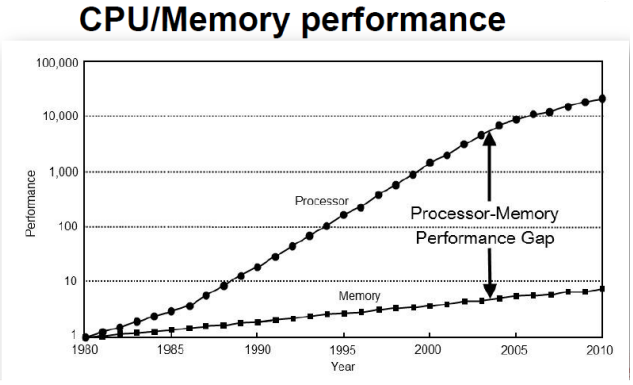
V praktickej časti sa práca venuje rozboru dostupných implementácií AdaBoostu. Taktiež sa čitateľ oboznámi s myšlienkovým procesom za výberom implementácie na rôzne účely. Následne sa dôkladne analyzuje implementácia Viola-Jones kaskády v OpenCV knižnici, jej silné a slabé stránky a možnosti na rozšírenie. Potom sa v práci rozoberá myšlienkový proces za výberom deskriptora, ktorý je implementovaný. Nasleduje popis pri implementácií SHOG deskriptora, jeho vyhodnotenie a rozbor možností, ktoré sa oplatí vyskúmať v budúcnosti, či už v spojení s SHOG deskriptorom, alebo inými algoritmami v tejto oblasti počítačového videnia.

1. TEORETICKÁ ČASŤ

1.1 Rýchlosť výpočtov

V počítačovom videní bolo vždy potrebné rýchle spracovanie dát. Napriek neúprosnému pokroku výpočtových technológií a zrýchleniu na všetkých stranách, počítačové videnie vždy bolo oborom v ktorom výkonu nie je nikdy dosť. Vo väčšine dnešných aplikácií nie je výkon na prvom mieste, kvôli dostupným zdrojom daného zariadenia. Keďže častokrát bežia detekčné algoritmy aj na slabších mobilných zariadeniach, a pretože počítačové videnie má veľa každodenných využití, je potrebné optimalizovať aj kvalitu aj rýchlosť algoritmov.

Za posledné roky narástol výkon všetkých komponentov bežných počítačov. Dva hlavné prvky, ktoré ovplyvňujú rýchlosť potrebných algoritmov sú odjakživa výpočtová rýchlosť a pamäť. Za posledné dve desaťročia sa začala zvyšovať priepasť medzi nárastom výkonu procesorov a výkonu pamäte (obrázok 1). Tento fakt spôsobuje, že napriek dostatku výpočtovej rýchlosti, musíme čoraz viac písať algoritmy, ktorých dizajn umožňuje úpravu interakcie s pamäťou, aby tieto algoritmy boli vôbec použiteľné vo výpočtoch v reálnom čase.



Obrázok 1 – Rozdiel medzi nárastom výkonu CPU a pamäte

Zdroj:Computer Architecture, a quantitative approach; Hennessy,Patterson,Arpaci-Dusseau

Zrýchlenie jednotlivých komponentov počítača prinieslo aj zvýšenie komplexity výpočtových modelov používaných v dnešných výpočtových jednotkách. S príchodom viacvláknových a viacjadrových procesorov nastáva trend paralelizácie a škálovateľnosti algoritmov, ktoré môžu naraz bežať na niekoľkých procesoroch alebo aj zariadeniach. Hlavné dva komponenty používané na rýchle výpočty sú CPU a GPU.

Obe tieto výpočtové jednotky majú svoje pre aj proti. CPU je viacúčelová výpočtová jednotka, ktorá nám umožňuje vykonávať bežné aktivity na počítači, a kvôli tomu je model veľmi komplikovaný. CPU pracuje s pamäťou na rôznych úrovniach a to registre, L1/L2/L3 cache, RAM, HDD, od najrýchlejšieho k najpomalšiemu. Tieto rozličné úložiská umožňujú rýchly prístup k dátam keď ich CPU vyžaduje. Taktiež to ale znamená, že tento proces je veľmi ťažký na pochopenie a nie je triviálne ho využiť. Vo väčšine prípadov stačí nechať CPU aby sa staralo o predikciu načítavania pamäte samo.

Predikcia je mechanizmus, ktorý zabezpečuje aby procesor mal v správnom čase dostupné správne prostriedky. Tento proces je veľmi aktívny a veľmi dôležitý pri podmienených skokoch (v C/C++ jazyku sú to IF klauzuly), kde v prípade zlej predpovede častokrát musíme čakať na dané zdroje. Preto často ak máme veľmi rýchle výpočty, treba si uvedomiť ako ktoré rozhodnutie a vetvenie kódu ovplyvní rýchlosť behu.

GPU je alternatívou k CPU a predstavuje omnoho viac špecializovaný komponent. Bežné GPU má stovky, dokonca až tisícky samostatných malých procesorov, zatiaľ čo bežné CPU ich má 2 alebo 4. GPU je jeden veľký zapuzdrený systém, s vlastnou, veľmi rýchlou pamäťou. Tento výpočtový model sa snaží čo najviac paralelizovať samotné výpočty. Jedno jadro GPU procesora je veľmi malé, a dokonca zdieľa prostriedky ako registre alebo inštrukčné jednotky v malom bloku. Dopad práce s pamäťou alebo rozhodovania je omnoho väčší pri GPU algoritmoch, presne z tohto dôvodu. Zatiaľ čo CPU sa stará o správu pamäte väčšinu času samo, pri GPU implementáciách si musí programátor byť vedomý každého detailu. Správne zarovnávanie pamäte je často kľúčové na to aby sme boli schopní využiť všetky dostupné zdroje GPU procesora. Pre porovnanie, výkon CPU býva okolo 60 GFLOPS (floating point operations per second), a GPU okolo 4 TFLOPS, čo znamená skoro 100x väčší výkon.

Charakter práce s obrazom umožňuje veľmi vysokú úroveň paralelizácie v algoritmoch. Preto je často potrebné každý algoritmus analyzovať tak, aby sme boli schopní posúdiť, aké je jeho využitie. Sú algoritmy ktoré sú paralelizovateľné na povedzme 50-100 vláknach naraz. Pre tieto algoritmy nemá zmysel uvažovať nad GPU implementáciou, keďže tie potrebujú naraz pustiť tisícky vlákien, ak chceme plne využiť dostupné prostriedky. Medzi ďalšie požiadavky patria napríklad možnosť zarovnávania pamäte a málo vetvenia alebo skupinové vetvenie kódu. Len po dosiahnutí týchto požiadaviek má zmysel začať uvažovať o omnoho komplexnejšej ale častokrát niekoľkonásobne rýchlejšej GPU implementácií.

1.2 Deskriptory a klasifikátory

Deskriptory a klasifikátory nám umožňujú zobrať ľubovoľný obraz/video a z neho dostať nejaké dáta a úvahy. Väčšinou je potrebné testovať ich rôzne kombinácie a zistiť, ktorá funguje na danú úlohu najlepšie, v rámci výkonnosti a rýchlosti.

Obrazový deskriptor je algoritmus na vytvorenie popisu nejakých vizuálnych vlastností obrazu alebo videa. V dnešnej dobe komunikačných technológií a internetu je potrebné rýchlo a spoľahlivo spracovávať a analyzovať nám dostupné dáta. Deskriptory poskytujú informácie o dátach, ktoré nie sú zrejmé na prvý pohľad a taktiež sa snažia popísať informácie vo formáte pochopiteľnom pre počítač. Bez týchto informácií by neboli možné mnohé dnešné aplikácie.

Rozdeľujú sa na dve skupiny [1]:

1. Všeobecné informačné deskriptory – hlavné typy popisujú podľa farby, tvaru, regionov, textury a pohybu
2. Informačné deskriptory špecifickej domény – riešia už nejakú špecifickú úlohu napríklad detekcia chodcov alebo sledovanie pohybu áut na parkovisku

Príklady využití deskriptorov sú napríklad multimédiá, kde chceme aby nám systém odporúčal obsah, ktorý by sme chceli konzumovať, triedenie súborov, aby sme

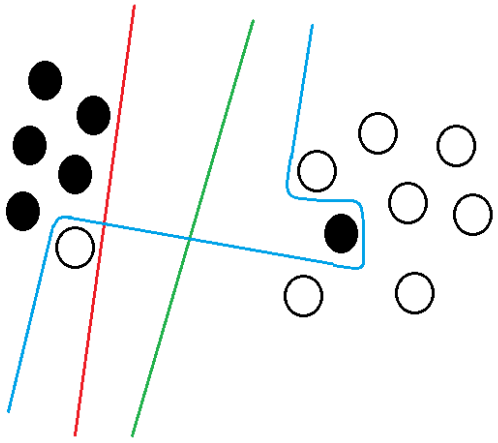
nemuseli pomenúvať súbory a aby to program urobil za nás, alebo aj ako medzistupeň pre komplexnejšie systémy, napríklad samoriadiace autá alebo rozšírená realita.

Klasifikácia je proces počas ktorého rozpoznáme, odlíšime a pochopíme rôzne typy objektov. Lineárny klasifikátor dokáže urobiť rozhodnutie pomocou hodnôt objektu lineárnou kombináciou charakteristík. Charakteristiky vstupného objektu na klasifikáciu sa volajú hodnoty čŕt a hodnoty objektu sa zväčša dodávajú klasfikátoru vo vektore čŕt získaných deskriptorom.

Klasifikátor vezme vektor čŕt a transformuje ho do bodu v N-dimenzionálnom priestore, a nejaká funkcia *f*, ktorá rozdelí tento priestor na dve časti. Podľa toho do ktorej časti patrí náš bod, je aj výsledná klasifikácia. Výhodou lineárnych klasifikátorov je ich jednoduchosť, čo následne umožňuje vysokú výpočtovú priepustnosť dát.

Klasifikátory sú metódy z oblasti strojového učenia. Medzi príklady klasifikátorov patria napríklad Bayesovsky klasifikátor, Fisherov lineárny diskriminant alebo SVM (Support Vector Machine).

Klasifikátory väčšinou vyžadujú určitý tréning, pri ktorom sa zoberie optimalizačný algoritmus, ktorý sa snaží minimalizovať chybu. Do tohto algoritmu sa dodajú sa vstupy a výstupy pre danú trénovaciu sadu a ako výsledok máme matematický model, ktorý dokáže rozhodovať o daných dátach.



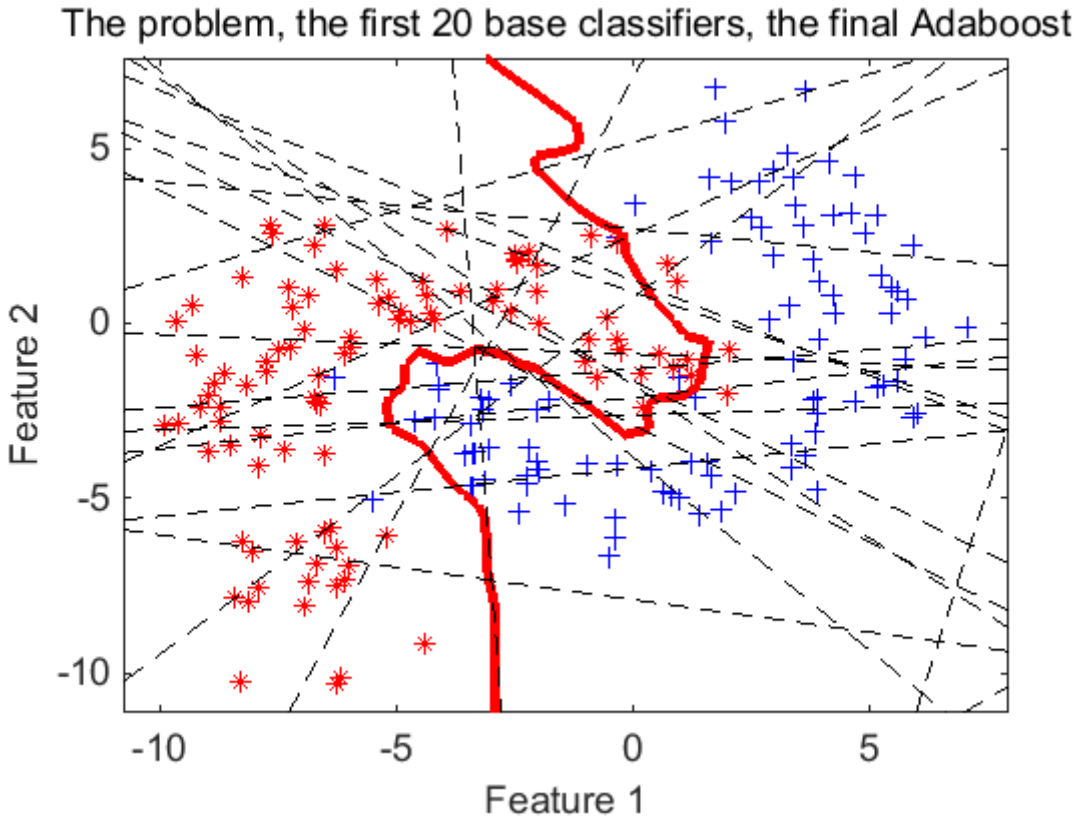
Obrázok 2 – Klasifikátor

Na obrázku 2 vidíme dve triedy dát, biele a čierne, a 3 rôzne klasifikátory vo forme kriviek. Najlepší klasifikátor je zelený, pretože najvšeobecnejšie rozdeľuje dve triedy dát, bez toho aby sa pretrénoval. Červený nerozdeľuje triedy rovnomerne

a v reálnej situácií nemusí fungovať správne. Modry klasifikátor je bežný príklad pretrénovania, tzn. natrénovaný model príliš tesne modeluje trénovacie dáta a neberie ohľad na šum v dátach, ktorý predstavuje osamotená čierna trieda medzi bielymi (a naopak).

1.3 AdaBoost a slabé klasifikátory

AdaBoost je meta-algoritmus pre strojové učenie. Väčšina algoritmov využíva klasifikátory tak, že natrénuje jeden silný klasifikátor, ktorý rozhodne o nejakej úlohe. Takýmto algoritmom je napríklad SVM. AdaBoost ide o krok ďalej. Nepracuje so samotnými dátami, ale narába s týmito klasifikátormi. Väčšinou sa používajú tzv. slabé klasifikátory, ktorých šanca urobiť chybu je menšia ako 50%. AdaBoost priradí týmto slabým klasifikátorom určitým spôsobom váhu, podľa toho ako veľmi daný klasifikátor ovplyvní finálny verdikt. Ako výsledok tohto procesu je jeden silný klasifikátor, ktorý využijeme na danú úlohu.



Obrázok 3 – Vizualizácia klasifikácie pomocou AdaBoostu

Ako slabé klasifikátory sa zvyčajne používajú rozhodovacie stromy rôznych typov. V práci Dalal a Triggs [2] používali ako slabé klasifikátory SVM klasifikátory, za účelom zjednodušenia viacdimenzionálnej úlohy na lineárnu.

Samotný AdaBoost, tak ako väčšina algoritmov strojového učenia, funguje v dvoch módoch, tréningový a testovací. V prípade AdaBoostu sa v trénovacom móde natrénuje model vážením slabých klasifikátorov a ten sa následnej uloží v nejakej forme(zvyčajne XML) na permanentné médium pre budúce využitie. V testovacom móde sa tento model načíta a následne použije na dátach o ktorých potrebujeme rozhodnúť, či patria do triedy A alebo B. Trénovací mód musí byť rýchly, aby tréning bol vykonaný v rozumnom čase ale taktiež je potrebné aby bol kvalitný. Testovací mód vyžaduje omnoho väčší dôraz na rýchlosť. V prípade že využívame AdaBoost na detekciu objektov v obraze, rozhodnutie sa volá miliónkrát pre jeden Full HD obrázok. Samotný AdaBoost a jeho rozhodovanie preto musí byť rýchle aby bolo možné ho využiť v aplikáciách v reálnom čase.

Spôsob akým dostaneme váženie závisí od modifikácie AdaBoostu. Medzi hlavné odnože patria:

a. Diskrétny AdaBoost – vstupná hodnota rozhodne o liste rozhodovacieho stromu, ten môže mať hodnotu -1 alebo 1, a porovná sa voči nejakej hranici buď jednotlivo alebo ako suma pre všetky vstupné hodnoty
b. Real AdaBoost – vstupná hodnota rozhodne o prechode po rozhodovacom strome, listová hodnota je odhad pravdepodobnosti, že hodnota patrí do danej triedy, z tých sa pre celý vektor urobí suma a porovná sa voči nule, ak > 0 tak patrí do triedy 1 a naopak
c. GentleBoost – limituje krok upravovania váh slabých klasifikátorov pri tréningu, tak aby algoritmus nenastavil ako krok nekonečno, čo spôsobuje nárast chyby
d. LogitBoost – aplikácia logistickej regresie na AdaBoost
e. Skoré ukončenie – pri tréningu nastavíme počet slabých klasifikátorov, ktoré budeme ceniť. V tomto prípade ale máme sekundárne kritérium kedy prestať pridávať slabé klasifikátory – napríklad ak bola dosiahnutá

hladina kvality výsledkov a viac klasifikátorov by len spôsobilo pretrénovanie/pomalší beh algoritmu.

f. Pruning – využíva určitý spôsob odstraňovania tých slabých klasifikátorov, ktoré neposkytujú dostatočnú rozhodovaciu hodnotu, alebo nejaké iné kritérium.

Najbežnejšie používaným slabým klasifikátorom v AdaBooste sú rozhodovacie stromy. Rozhodovacie stromy sú jednoduché a rýchle, čo dobre funguje v spojení s potrebou generovania veľkého poštu slabých klasifikátorov. Taktiež je potrebné mať slabý klasifikátor, ktorý je konzistentne lepší ako náhodný výber. Rozhodovacie stromy toto dokážu bez hociakého nastavovania parametrov, čo znamená, že sú omnoho jednoduchšie na použitie ako napríklad SVM. Je veľmi jednoduché upravovať odchýlku a skreslenie modelu pomocou nastavovania hĺbky stromov. AdaBoost zvykne znižovať skreslenie ale taktiež odchýlku čo môže spôsobovať pretrénovanie.

1.4 AdaBoost a spojenie s deskriptormi

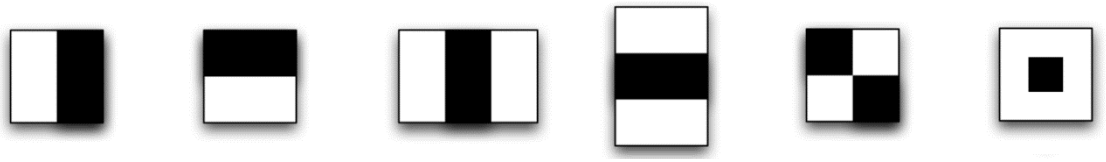
Najväčší problém využitia AdaBoostu v praxi je potreba dobrého deskriptoru, ktorý správne vyjadrí nami hľadané objekty v obraze. Za predpokladu, že nami používaná implementácia AdaBoostu je maximálne optimalizovaná, zostáva už len nájsť deskriptor, ktorý dokáže vygenerovať veľa rôznych čŕt, ktoré sú zároveň rýchle a taktiež ľahko nepretrénujú trénovaný model.

V minulosti už boli publikované práce s deskriptormi za použitia AdaBoostu v spojení s Haarovými vlnkami [3], neskôr taktiež v spojení s HOG deskriptorom, a SVM ako slabým klasifikátorom [4], v spojení s Local Binary Patterns a taktiež v spojení s aproximáciami HOG deskriptora [5] [6].

Keďže napriek využitiu ideálnych deskriptorov na danú úlohu sa dokáže jeden silný klasifikátor ľahko pretrénovať, používajú sa tzv. kaskády klasifikátorov. Tento prístup má niekoľko výhod. Umožňuje používať rôzne typy klasifikátorov, s rozličnými rýchlosťami a odlišnými filtračnými schopnosťami. Na začiatku kaskády sa zvyknú používať čo najrýchlejšie klasifikátory, a čo najpresnejšie klasifikátory na konci kaskády.

1.4.1 Haarové vlnky

Jednoduchá Haarová vlnka (v angl. Haar-like feature) v oblasti detekcie objektov sa dá definovať ako rozdiel súm pixelov čiernych a bielych oblastí v danom obdĺžnikovom okne a tieto oblasti môžu byť rôznych veľkostí a tvarov. Najjednoduchšie vlnky sa skladajú z dvoch obdĺžnikových oblastí, a komplexnejšie z troch alebo štyroch oblastí



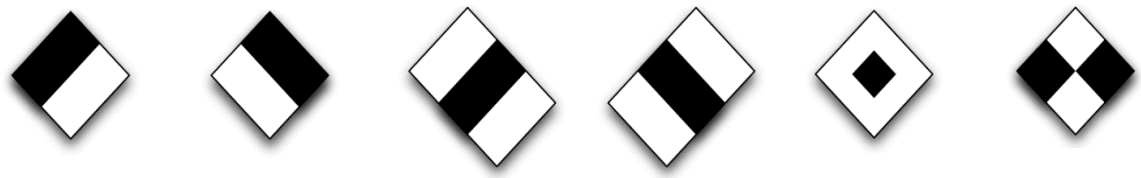
Obrázok 4 – Základné Haarové vlnky

Prvýkrát boli použité a pomenované Violom a Jonesom pri snahe vytvoriť detektor tvárí. Ich meno pochádza z Haarových vlniek (v angl. Haar Wavelet) z matematiky, ktoré sú postupnosťami funkcií vytvárajúcich štvorcové grafy, na ktoré sa tieto črty na detekciu objektov podobajú. Hlavným poznatkom bol fakt, že ľudská tvár sa dá rozdeliť na obdĺžnikové časti, v ktorých keď spočítame hodnoty pixelov, dostaneme dostatočnúinformáciu na to aby sme mohli rozhodnúť či tam tvár je alebo nie je.

Obrázok 5 – Aplikácia Haarových vlniek na obrázok

Dôvod prečo sa Haarové vlnky dajú využiť v aplikáciách v reálnom čase je, že pomocou integrálneho obrazu je možné podstatne zjednodušiť zdroje potrebné na výpočet jednej vlnky. Integrálny obraz (taktiež summed-area table) spôsobí, že na každú Haarovú vlnku je potrebných maximálne 9 prístupov do pamäte, 6 pre dvoj obdĺžnikovú, 8 pre troj obdĺžnikovú a 9 pre štvoro obdĺžnikovú.

Neskôr sa začali taktiež používať naklonené Haarové vlnky (obrázok 4), ktoré dostaneme otočením ľubovoľnej vlnky o 45 stupňov. Napriek tomu, že sú úspešné pri popise niektorých typov objektov, v bežnej praxi sa nezvyknú využívať, kvôli problémom so zaokrúhľovaním a s výpočtovou rýchlosťou.



Obrázok 6 – Naklonené Haarové Vlnky

Hlavným problémom využitia Haarových vlniek v praxi je ich náhodnosť a počet prístupov do pamäte. Ako bolo spomenuté v kapitole 1.2, v dnešnej dobe je rýchlosť procesorov podstatne vyššia ako rýchlosť pamäte. V prípade, že chceme optimalizovať prácu s pamäťou, je potrebné načítavať hodnoty, ktoré sú v pamäti uložené za sebou, aby bolo možné ich načítať viac naraz (radič zbernice naraz zvykne prenášať 128-bitov a bežná hodnota je 32-bit), a taktiež je potrebné čo najviac znížiť počet prístupov do pamäte. V praxi je 4 až 9 prístupov do pamäte počas klasifikácie veľa, pretože vtedy pristupujeme na veľa náhodných miest, ktoré nie je možné ani vektorizovať, ani skupinkovať tak ako sa to dá pri rovnomerných výpočtoch, napríklad počas výpočtu integrálneho obrazu. Kvôli tomu je klasifikácia pomocou Haarových vlniek problematická, pokiaľ chceme veľmi rýchlu implementáciu.

1.4.2 Histogram of oriented gradients

Histogram orientovaných gradientov, ďalej HOG, je deskriptor využívaný v oblasti detekcie objektov na rôzne úlohy. Je omnoho komplexnejší ako Haarová vlnka a črty, ktoré poskytuje, sú všetky vypočítané v jednom jednotnom algoritme. Hlavnou

myšlienkou HOG deskriptora je vyjadrenie detekčného okna podľa orientácií gradientov, ktoré dokážu dobre popísať hrany v obraze bez toho, aby sme potrebovali pracovať s jednotlivými pixelmi.

Prvýkrát bol tento termín použitý v práci Dalala a Triggsa [2], a bol použitý v spojení s SVM klasifikátorom na detekciu chodcov. Autori sa snažili navrhnúť algoritmus na popis objektov v reálnom čase. Kvôli dobrému vyjadreniu tvaru a lokálneho výzoru objektu, je HOG invariantný voči geometrickej a fotometrickej rotácií, čo spôsobuje že je jedinečne výkonný pri detekcií ľudí, za predpokladu, že sú v relatívne vzpriamenom postoji.

Samotný výpočet HOG deskriptoru má niekoľko krokov, a v práci Dalala a Triggsa [2] boli detailne rozobraté najlepšie verzie podkrokov. Výsledný algoritmus sa skladá z nasledujúcich častí:

1. Výpočet magnitúdy a orientácie gradientu v každom bode obrazu
2. Vytvorenie buniek, ktoré obsahujú histogram orientácií gradientov pixelov v danej bunke
3. Normalizácia buniek v rámci väčších blokov

Diskrétny gradient ľubovoľného pixelu vypočítame aplikovaním nejakej derivačnej masky v horizontálnom a vertikálnom smere. V základnej práci HOGu bola vybraná maska [-1,0,1], kvôli jej jednoduchosti, a nulovému posunu hrany. Tieto hodnoty dosadíme do vzorca na výpočet magnitúdy

$$\nabla i(x,y) = \frac{\partial i(x,y)}{\partial x}\hat{x} + \frac{\partial i(x,y)}{\partial y}\hat{y}$$

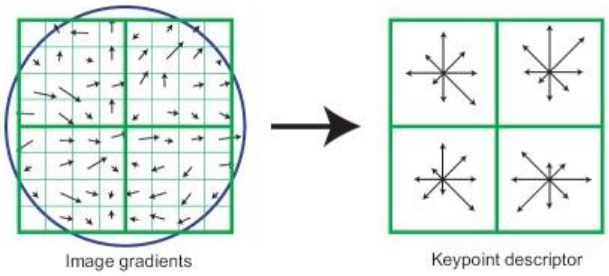
a na výpočet orientácie gradientu

$$\theta = \operatorname{atan2}(\frac{\partial i(x,y)}{\partial y}, \frac{\partial i(x,y)}{\partial x})$$

kde $\frac{\partial i(x,y)}{\partial x}$ je gradient v smere x(horizontálnom), $\frac{\partial i(x,y)}{\partial y}$ je gradient v smere y(vertikálnom).

Histogram orientácií gradientov v jednotlivej bunke dostaneme rozdelením celého 360 stupňového koláča na niekoľko košov, ktoré obsahujú uniformnú časť. V základnej práci bolo odporúčané používať 9 košov. Taktiež sa odporúča používať

bezznamienkové histogramy, ktorých hodnoty sú od 0 po 180 stupňov namiesto 0 po 360 stupňov, pretože dávajú lepšie výsledky. Za veľkosť bunky sa v základnej práci volí štvorec 3x3 pixelov. Samotná magnitúda gradientov sa váži do dvoch najbližších košov podľa orientácie gradientu.

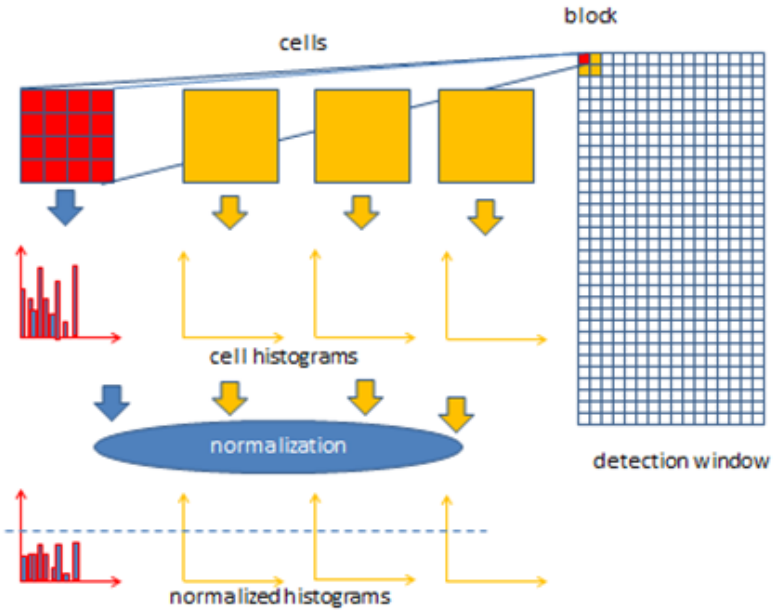


Obrázok 7 – Transformácia pixelov na bunkový histogram

Zdroj: "Distinctive Image Features from Scale-Invariant Keypoints," Lowe, IJCV, 2004

Normalizácia buniek v blokoch je potrebná aby bol výsledný deskriptor viac invariantný voči nasvieteniu a tieňom. Z dostupných normalizačných vzorcov bola zvolená L2-sqrt norma so vzorcom $x/\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2 + \varepsilon^2}$ kde epsilon predstavuje nejakú malú normalizačnú konštantu pre zamedzenie delenia nulou. Suma v menovateli predstavuje sumu všetkých hodnôt pixelov v danom bloku, ktorou normalizujeme všetky hodnoty v bloku. V základnej práci bola zvolená veľkosť blokov 6x6 pixelov.

Problém s výpočtom hodnôt HOGu je ten, že krok 2 a 3 majú príliš veľa prístupov do pamäte (treba spočítať všetky hodnoty buniek/blokov do rôznych uskupení) a krok 3 sa nedá dobre paralelizovať, pretože každý blok môže počítať bez synchronizácie maximálne jedno vlákno. Na jedno detekčné okno býva malý počet blokov, čo spôsobí, že pre relatívne malé obrázky do 1000x1000 nebude možné vytvoriť až také veľké množstvo paralelných výpočtov, čo môže spôsobiť, že rozdiel medzi CPU a GPU algoritmami sa bude podstatne zmenšovať.



Obrázok 8 – Ilustrácia fungovania normalizácie

Zdroj: "Distinctive Image Features from Scale-Invariant Keypoints," Lowe, IJCV, 2004

1.4.3 Aproximácie HOG deskriptora - HistFeat

Predošlé dva popísané deskriptory podávali dobré výsledky v rámci detekcie, ale mali pár nedostatkov, ktoré znemožňovali rýchlu implementáciu. Čiastočne sa dal tento problém obísť kaskádovým klasifikovaním. Neskôr bola ale snaha zobrať poznatky z Haarových vlniek a HOG deskriptora a pomocou nich navrhnúť ich verziu, ktorá by umožňovala omnoho rýchlejšiu detekciu.

Autori analyzovali Haarové vlnky a usúdili, že je potrebné zredukovať počet prístupov do pamäte v klasifikačnom kroku z maximálnych 9 na 1. Ako možnosť bolo spomenuté predpočítavanie samotných hodnôt čŕt z integrálneho obrazu tak aby klasifikačný krok vyžadoval menej prístupov.

Autori sa ale vydali cestou využitia poznatkov z HOG deskriptora, a to že gradient poskytuje dobrú lokálnu informáciu o obraze. Prvým takýmto deskriptorom je HistFeat [7]. Prvý krok pre výpočet orientácie gradientu jednotlivých pixelov limitujú na 8 košov, čo umožňuje ukladať hodnotu v 3ch bitoch. Magnitúdu pixelov počítajú sčítaním absolútnych hodnôt, namiesto presnejšej Euklidovskej vzdialenosti pre maximalizáciu rýchlosti. Bunky sú nastavené na fixnú 4x4 veľkosť, a histogram každej

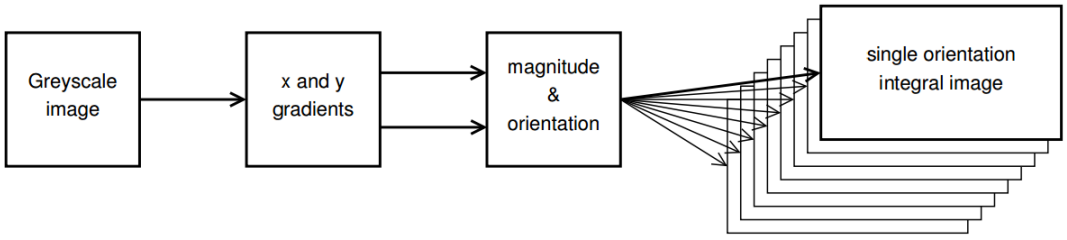
bunky je uložený v jednej 32-bit premennej. Na každú magnitúdu aplikujeme nejakú hraničnú hodnotu, čo nám dá 0 alebo 1, následne sčítame všetky orientácie, a v prípade že všetkých 16 hodnôt bunky ukazuje rovnakým smerom (všetky patria do toho istého koša) osekáme hodnotu 16 na 15 čo umožní uložiť výsledok do jednej 4-bit premennej. Koncept normalizácie blokov sa kvôli rýchlosti (podstatne redukuje možnosti paralelizácie) v tomto deskriptore vôbec nepoužíva.

V klasifikačnej fáze využívali „a posteriori“ tabuľku, ktorú adresovali pomocou 2 z 8 histogramových hodnôt. Prístup do pamäte bol limitovaný na jednu 32-bit hodnotu. Tieto zmenu spôsobili, že HistFeat deskriptor je omnoho rýchlejší ako jednoduchšie Haarové vlnky.

1.4.4 LiteHOG a S-HOG

Na HistFeat nadväzuje ďalšia práca od rovnakých autorov v ktorej sa snažia vytvoriť deskriptor s lepšou popisovacou schopnosťou. Autori vychádzali z predpokladu, že HistFeat je príliš jednoduchý na to aby plne využil výpočetnú kapacitu a preto je ho možné rozšíriť. Jeho algoritmus síce optimalizoval prístupy do pamäte natoľko aby rýchlosť zbernice nebola problémom, čo ale odhalilo, že sa nevyužívajú všetky výpočetné zdroje. Pokračovanie v rozvoji deskriptorov predstavovalo snahu využiť tieto výpočetné zdroje.

Pri LiteHOGu hlavná zmena oproti HistFeat je využitie Fisherovej diskriminačnej analýzy (ďalej FDA) na transformáciu 8-rozmerného priestoru na 1-rozmerný. Hodnoty histogramu predstavujú súradnice v 8-rozmernom priestore. Táto výsledná hodnota sa použije ako vstup pre AdaBoost, rovnako ako v prípade Haarových vlniek. Autori ale zistili, že ak počítame FDA pre všetkých 8 hodnôt, tak algoritmus nie je obmedzený pamäťou ale výpočtami. Preto navrhli upravenú verziu LiteHOG+, kde výber počtu hodnôt je variabilný od 1-8. Táto verzia je nielen omnoho viac vyrovnaná v oblasti výpočty/pamäť ale taktiež má lepšie popisovacie vlastnosti, čiže podáva aj lepšie výsledky. Autori taktiež navrhli úpravu pre AdaBoost, kde ak dve črty majú rovnako dobrú rozhodovaciu hodnotu, vyberieme tú ktorá je rýchlejšia, koncept aplikovateľný na LiteHOG+.



Obrázok 9 – Algoritmus SHOG deskriptora

Zdroj: "Large scale sign detection using HOG feature variants," Overett, Petersson, NICTA, 2011

Ďalšou variantou tohto konceptu je S-HOG a FDA-HOG. Oproti predošlému deskriptoru neosekávame magnitúdu gradientu ale využívame integrálny obraz na rýchly výpočet histogramov. Pre každý z 8 košov histogramu vypočítame jeden integrálny obraz, ktorý nám následne umožní rýchlo vypočítať histogramy v rámci bunky. S-HOG považuje za jednu črtu ľubovoľnú jednu hodnotu z hociktorého z ôsmych košov. FDA-HOG vloží do rovnice FDA hodnoty histogramov a vytvorí z 8 orientácií jednu lineárnu transformáciu.

1.5 Viola-Jones kaskáda

V roku 2001 bol vyvinutý framework na detekciu objektov dvoma vedcami Paulom Viola a Michaelom Jonesom. Bol to prvý kaskádový framework, ktorý podával konkurencie schopné výsledky v reálnom čase. Framework je schopný práce na rôznych typoch objektov, ale hlavným cieľom bolo preukázať výsledky na úlohe detekcie tvárií [3].

Kaskáda je súbor niekoľkých za sebou idúcich krokov. Každý krok kaskády obsahuje vlastný natrénovaný model, ktorý sa skladá z niekoľkých slabých klasifikátorov. Pozitívny nález je taký, ktorý prejde cez všetky kroky kaskády úspešne. Takýto prístup značne urýchľuje rozhodovanie a taktiež umožňuje využitie rôznorodých klasifikátorov pri tej istej úlohe. V prvých krokoch kaskády býva jeden alebo niekoľko rýchlych klasifikátorov, ktoré podstatne znížia počet negatívnych vzoriek a umožnia aby sme mohli na koniec kaskády efektívne využiť pomalší a presnejší klasifikátor bez veľkého spomalenia. Viola-Jones kaskáda sa skladá zo za sebou uložených AdaBoost klasifikátorov a jej klasifikácia sa skladá z dvoch krokov, z výpočtového

a z klasifikačného. Počas výpočtového kroku sa predpočítajú všetky možné hodnoty pre dané vzorky (zväčša globálne pre celý obrázok) a pri klasifikačnom kroku robíme rozhodnutia na základe vopred vypočítaných dát.

Táto prvá verzia frameworku využívala Haarové vlnky ako deskriptor. Na ich výpočet bol využívaný integrálny obraz. Keďže je ich možné generovať veľmi veľký počet, dajú sa z nich dobre robiť rôzne kroky kaskády. V každom kroku sa vyhodnotil stanovený počet vlniek, vybrali sa tie s najlepšími vlastnosťami a tie sa použili vo výslednom klasifikátore. Framework rozširoval AdaBoost o rôzne nastavenia, ktoré umožňujú lepšie natrénovanie a to napríklad skoré ukončenie alebo bootstrapping negatívnych obrázkov do ďalšieho kroku kaskády.

Tento framework bol neskôr použitý ako hlavný stavebný kameň pre mnohé rozšírenia a nové pokusy. Bolo urobených mnoho implementácií Viola-Jones kaskády napríklad pre MATLAB a knižnicu OpenCV. V knižici OpenCV bola neskôr pridaná podpora pre HOG deskriptor a LBP deskriptor a taktiež tam sú multi-scale detekčné algoritmy pre CPU (Haar,HOG,LBP) a pre GPU (Haar, LBP). Implementácia Viola-Jones kaskády taktiež existuje v balíčku Multi-boost.

Hlavnou výhodou OpenCV CPU implementácie je jednoduchá rozšíriteľnosť. Tréningový aj testovací režim majú interface, ktorý po implementácií určitých krokov umožňuje plne využívať vlastne naprogramované deskriptory. GPU časť, ktorá podporuje len testovací režim, ale vyžaduje nízkoúrovňovú implementáciu šitú priamo na mieru, takže nie je možné jednoducho naimplementovať ďalší deskriptor pre GPU.

1.6 OpenCV

OpenCV je knižnica programovacích funkcií, ktorá slúži hlavne na prácu s obrazom v reálnom čase. Je dostupná pod BSD licenciou. Možno ju používať na viacerých platformách, napríklad Windows, Linux, OS X, Android, iOS. Obsahuje moduly na úpravu videa, rozpoznávanie objektov, sledovanie pohybu, segmentáciu obrazu, rozšírenú realitu, mobilnú robotiku a taktiež obsahuje podporu na paralelné výpočty na CPU aj GPU.

Prvotne bola knižnica vyvíjaná pobočkou Intelu v Nižnom Novgorode, odvtedy sa k podpore pridala Willow Garage a Itseez. V auguste 2012 prevzala vývoj knižnice nezisková organizácia OpenCV.org [8], ktorá udržuje stránku pre developerov a používateľov, vrátane online dokumentácie [9].

Väčšina knižnice je napísaná v jazyku C++. Taktiež je dostupná stará verzia v jazyku C. Jazyky Java, Python, MATLAB/OCTAVE majú dostupné úplné wrappery na C++ jadro a taktiež existujú verzie pre C#, Perl a Ruby, ktoré boli vyvinuté za účelom rozšírenia záujmu o túto knižnicu. Od roku 2010 sa taktiež vyvíjajú CUDA verzie dostupných algoritmov a od roku 2012 pre OpenCL.

V tejto práci sa hlavne čerpá z modulu na detekciu objektov, Viola-Jones kaskády, ktorá je dostupná ako separátna kompilovateľná aplikácia a z GPU modulov pre CUDA platformu.

2. PRAKTICKÁ ČASŤ

2.1 Testovanie

Všetky časy a výsledky boli robené na nasledovnej zostave:

Intel i7 4700 3.4 GHz + 24 GB 1600 MHz RAM

Nvidia Geforce MSI GTX 970 4GB

Samsung Evo 850 SSD

2.1.1 Dataset

Detekcia a rýchlosti boli merané na datasete z videa futbalového zápasu. Zábery v datasete pochádzajú z 30 fps videa s rozlíšením 2048x1536. V rámci projektu Analýza obrazu v zimnom a letnom semestri 2015/2016 boli anotovaný hráči na záberoch s frekvenciou 2 fps. Pre potrebu tejto práce boli vysekané pozitívne a negatívne dáta

Obrázok 10 – Príklad snímku z videa datasetu

Obrázok 11 – Príklad anotovania

Obrázok 12 – Príklad výsekov

pomocou týchto anotácií. Príklad záberu z videa je na obrázku 10, na obrázku 11 je zobrazený spracovaný záber vrátane anotácií a na obrázku 12 je možné vidieť konkrétne vyseknuté vzorky popredia (hŕačov), ktoré sa v tejto práci používajú na trénovanie. Anotácie sú uložené v XML súbore.

2.2 Porovnávanie implementácií AdaBoostu

V rámci výberu implementácie AdaBoostu boli porovnávané 3 implementácie:

1. OpenCV 3.0 AdaBoost
2. Viola-Jones Kaskáda
3. MultiBoost

Cieľom tejto podrobnej analýzy bolo vybrať implementáciu, ktorá je vhodná na experimentovanie s novými deskriptormi. Hlavné kritériá porovnávania sú rozšíriteľnosť, rýchlosť a už existujúce prostriedky na trénovanie a testovanie detekcie.

2.2.1 OpenCV 3.0 implementácia

Táto implementácia je úplne všeobecná implementácia cielená na experimentovanie nielen v rámci počítačového videnia. Disponuje štvorma variantmi trénovacej časti a to DiscreteBoost, RealBoost, LogitBoost a GentleBoost. Využíva rozhodovacie stromy ako slabý binárny klasifikátor a umožňuje len jeden spôsob ako ukončiť tréning: počtom slabých klasifikátorov. Sú tu dostupné dva módy vyhodnocovania a to buď sumovaním hodnôt alebo voľbou väčšiny.

Implementácia nedisponuje žiadnymi prostriedkami na výpočet deskriptorov a taktiež nemá žiadne optimalizácie dodávania hodnôt čŕt do algoritmu. V testovacom móde táto implementácia tiež nemá žiadne veľké možnosti, je možné buď testovať veľa vzoriek vektorov naraz, ale keďže je potrebné ich predpočítať je to pamäťovo náročné, alebo testovať po jednej vzorke, čo je zase pomalé. Implementácia je veľmi rýchla, ale keďže nedisponuje všetkými požadovanými prostriedkami vyžaduje veľa práce aby mohla byť použitá v práci s obrazom v reálnom čase.

Táto implementácia bola využívaná pri prvotnej analýze a prvotnom experimentovaní s dostupnými možnosťami. Bola vybratá kvôli jej jednoduchosti a kvôli úplnosti jej dokumentácie.

2.2.2 Viola-Jones implementácia

Táto implementácia je súčasťou väčšieho frameworku a okrem implementácie AdaBoostu, je taktiež implementáciou Viola-Jones kaskády [3]. Vyvinutá bola pomocou OpenCV 2.0 implementácie, ktorá je predkom 3.0, preto je ale ich štruktúra podstatne odlišná. Dokumentácia je dostupná len na úrovni používateľa, čo spôsobuje, že analýza je veľmi náročná. Trénovací mód je separátna aplikácia od testovacieho módu, ktorý je súčasťou knižnice. Implementácia obsahuje všetky súčasti 3.0 implementácie potrebné pre prácu s obrazom. Ako slabý klasifikátor sú taktiež používané rozhodovacie stromy.

Implementácia obsahuje výpočet čŕt, optimalizácie výpočtov, a dokonca aj rozhrania umožňujúce rozšírenie o ďalšie deskritptory. Testovací mód disponuje metódami pre detekciu vo veľkých obrázkoch a taktiež disponuje GPU implementáciami pre niektoré jej súčasti. Kód tejto implementácie je písaný so zámerom využitia v reálnom čase.

Táto implementácia bola využívaná neskôr, pri viac komplexných experimentoch. Pri prechode od 3.0 implementácie k tejto boli vykonané porovnávania, ktoré podporili toto rozhodnutie a tým sa budeme venovať v ďalšej kapitole.

2.2.3 MultiBoost

Počas analýzy prvých dvoch implementácií bola analyzovaná aj knižnica MultiBoost v rámci spolupráce s Projekt 1 Analýza Obrazu. Táto knižnica podporuje boosting ako klasifikátor viac tried, oproti bežnému AdaBoostu, ktorý je binárnym klasifikátorom. Tento prístup má svoje výhody aj nevýhody, ktoré sú ale predmetom hlbšieho skúmania.

Tak ako VJ implementácia, táto knižnica umožňuje kaskádový prístup tréningu a testovania. Obsahuje integráciu výpočtu čŕt a taktiež rozhrania na implementácie vlastných deskriptorov. Oproti VJ implementácií taktiež podporuje rozhrania na implementácie vlastných slabých klasifikátorov.

Problémom pre využitie je nedostupnosť optimalizácií testovacieho kroku. Zatiaľ čo VJ implementácia disponuje rozhraniami pre rýchle implementácie detekcie v obraze, MultiBoost je viac smerovaný na všeobecný tréning modelov, tak ako 3.0 implementácia, čo znamená, že hlavný dôraz nie je práca s obrazom. Štruktúra knižnice je ešte viac komplexnejšia ako VJ a preto je do budúcnosti potrebná analýza náročnosti úpravy testovacej časti pre rýchlu detekciu v obraze.

2.3 Analýza a rozšírenie OpenCV 3.0 implementácie AdaBoostu

Začiatky analýzy prebiehali súbežne so začiatkami experimentovania s dostupnými prostriedkami s cieľom oboznámiť sa lepšie s vnútorným dianím algoritmov. Úplne prvou úlohou bola implementácia detekcie futbalistov v obraze pomocou AdaBoostu.

Ako bolo spomenuté v minulej kapitole, pri prvých pokusoch som využíval OpenCV 3.0 implementácia AdaBoostu, pretože bola najjednoduchšia na pochopenie pre začiatočníka v oblasti Boostingu. Táto implementácia vyžadovala vlastnú implementáciu načítavania obrazu a taktiež úpravu vstupov na správny formát. AdaBoost ako taký vyžaduje vstupné vzorky a ich odpovede pri tréningu, pomocou ktorých sa natrénuje model a ten je možné neskôr používať pri testovaní.

Na testovanie sa implementoval jednoduchý algoritmus, ktorý podáva AdaBoostu jednu vzorku vyseknutého okna rovnakej veľkosti ako boli vstupné vzorky pri tréningu. Pomocou anotačného nástroja som vysekal testovaciu sadu vzoriek 96x160 pixelov, pri ktorých sme vedeli aké výstupy majú podávať. Tento algoritmus bol potrebný, keďže 3.0 implementácia nemá multiškálovú detekciu v obraze a treba testovať osobitné výseky.

Pri prvých pokusoch tréningu som používal priamo pixely samotného výseku obrázku ako vstupy pre AdaBoost. Keďže ale samotné intenzity farieb (v tomto prípade greyscale) podávajú nie veľmi dobrú informáciu o objektoch v obraze, takto natrénovaný model podával len okolo 92% úspešnosť rozhodovania o pozitívnych nálezoch a 20% chybu v rozhodovaní pri negatívnych nálezoch. Tieto hodnoty boli vyhodnotené podľa spomínaného testu.

Ako vidíme, tieto čísla nie sú postačujúce na správnu detekciu v obraze, pretože pri jednom 2048x1536 obrázku sa vyhodnocuje detekčné okno 4223299 krát pri veľkosti okna 24x40 pixelov. Z toho vyplýva, že je potrebné využívať deskriptor, ktorý lepšie popíše obraz ako samotné hodnoty pixelov.

Kvôli rýchlosti a jednoduchosti som pokračoval s implementáciou Haarových vlniek. Inšpiroval som sa už dostupnou implementáciou vo Viola-Jones kaskáde, o ktorej bolo overené, že je rýchla a využiteľná. Naprogramované boli aj základné vlnky aj rozšírené (naklonené).

Súbežne s implementáciou Haarových vlniek sa začalo experimentovať s multiškálovou detekciou veľkých obrázkov. Analyzované boli verzie detectMultiScale HOG deskriptoru v OpenCV a Viola-Jones kaskády, ktoré už boli dostupné. Bolo potrebné naimplementovať systém, ktorý dynamicky pracuje s rôznymi škálami obrázku. Prvá verzia tohto algoritmu naznačila, že je potrebné začať optimalizovať algoritmus a užšie ho spájať s výpočtom vstupov pre AdaBoost. Pre obrázok 2048x1536 táto verzia nestihla zbehnúť v časovom limite 30 minút.

Hlavný problém bol v zdieľaných výpočtoch medzi prechodmi z jedného detekčného okna na druhé. Prvotná verzia používala drahé alokácie pamäte a resizovanie okien aby dostala správnu dĺžku vstupného vektora pre AdaBoost. Preto som odstránil resizovanie a alokácie, tým že som pre každú škálu vždy predpočítal všetky výpočty cez integrálny obraz (predtým sa počítal pre výsek, teraz pre celý jeden krok) a taktiež som predalokoval na začiatku algoritmu všetku vyžadovanú pamäť. Tieto dve základné optimalizácie zrýchlili beh algoritmu na 15 sekúnd.

Čím hlbšie pokračoval vývoj vlastnej verzie kaskády a multiškálového algoritmu, tým viac zrejmé bolo ako veľmi sa inšpirujem Viola-Jones kaskádou, ktorá mala všetky tieto prvky dostupné. Pri porovnaní VJ detectMultiScale s mojou vlastnou implementáciou, som zistil že VJ kaskáda s hĺbkou 28 krokov a 3-5 slabých klasifikátorov v jednom kroku, beží okolo 1,1 sekundy na jednom vlákne, zatiaľ čo môj vlastný beží 15 sekúnd.

Moja implementácia mala nasledovné problémy:

- OpenCV 3.0 implementácia nevyhadzuje nepoužívané črty z modelu a je jej treba dodávať vždy rovnako dlhý vektor – tzn. počítame zbytočné črty, na druhej strane VJ počíta črty len ak ich potrebuje, tzn. ak aj nezídeme do nejakej vetvy rozhodovacieho stromu, daná črta sa nevyhodnotí
- Kaskádový prístup spôsobuje, že cez prvý krok prejde menej ako polovica okien, čo znamená, že nie vždy sa vyhodnocujú všetky slabé klasifikátory kaskády, zatiaľ čo 3.0 vždy musí vyčísliť všetky slabé klasifikátory a až potom je schopná rozhodnúť
- Spôsob akým je navrhnutý detekčný mód OpenCV 3.0, spôsobuje, že inicializačný krok sa pustí pri každom volaní, zatiaľ čo VJ ho urobí raz na začiatku.

Po týchto nálezoch začalo byť jasné, že postup ktorým sa to vydáva je preimplementovanie schopností VJ kaskády do môjho vlastného programu. Taktiež vďaka snahe implementovať vlastné verzie sa dostavilo väčšie pochopenie existujúcich zdrojových kódov a implementácií čo umožnilo urobiť rozhodnutie a pustiť sa radšej do hlbokej analýzy a potenciálneho využitia VJ kaskády na naše účely.

Napriek tomu treba spomenúť, že v prípade ak chceme urobiť porovnávací framework medzi implementáciami, treba ísť týmto smerom vlastných implementácií. V prípade ak by sme chceli v budúcnosti do hĺbky skúmať využitie knižnice MultiBoost prípadne inej knižnice, je potrebné mať všeobecné implementácie a rozhrania, ktoré umožnia jednoduché zapojenie pre naše potreby za cenu pomalšieho behu.

2.4 Analýza OpenCV 2.0 implementácie Viola-Jones kaskády

Ako už bolo spomenuté v teoretickej časti, Viola-Jones kaskáda je framework slúžiaci na tréning kaskádových modelov. V OpenCV knižnici sa nachádza rozsiahla implementácia, ktorá umožňuje veľmi jednoducho natrénovať svoj vlastný model na vlastných dátach, pomocou už dostupných deskriptorov.

Implementácia má dva módy: tréningový a detekčný. Tréningový mód má za úlohu čo najjednoduchšie umožniť natrénovanie modelu. Tento mód je pripravený ako samostatná aplikácia, ktorá dokáže načítať obrázky, nastaviť parametre pre tréning a následne natrénovať kaskádu podľa vybraných parametrov.

V tréningovom móde sú dostupné tri deskriptory – Haarové vlnky, HOG deskriptor a LBP deskriptor. Prvé dva deskriptory sú ordinálne, a LBP deskriptor je kardinálny. Haarové vlnky sú veľmi dobré pre trénovanie celej kaskády, pretože je možné vygenerovať veľké množstvo rôznych vlniek a taktiež sú unikátne vhodné na detekciu tvárí. HOG deskriptor na druhej strane je podstatne pomalší ako Haarové vlnky a taktiež nedokáže vygenerovať tak veľké množstvo rôznych hodnôt, ale tieto samotné hodnoty majú väčšiu popisnú hodnotu ako jedna Haarová vlnka, čo znamená, že sú lepšie na vytvorenie jedného kvalitného kroku kaskády. Pre porovnanie pre 20x50 výsek je možné vygenerovať viac ako 400000 vlniek, zatiaľ čo HOG deskriptor umožní vygenerovať len okolo 6000 rôznych hodnôt. LBP deskriptor, alebo taktiež local binary pattern deskriptor, využíva hodnoty postavené na úplne inom základe, a ten nie je podstatný pre túto prácu. Oplatí sa spomenúť, že oproti Haarovým vlnkám je výpočet rýchlejší ale taktiež trochu menej kvalitný.

V detekčnom móde sú dostupné dva deskriptory – Haarové vlnky a LBP deskriptor. Vo verzií OpenCV 3.0 ešte nie je dostupná detekčná implementácia HOG deskriptora. Detekčný mód je dostupný aj na CPU aj na GPU. Tu to začne byť trochu komplikované. Vo verzií OpenCV 2.0 bola prepísaná Viola-Jones kaskáda a bol modernizovaný zápis modelu. Je možný zápis do nového aj starého modelu, ale zo starého modelu nie je možné kaskádu reštartovať aby pokračoval tréning ďalej. Keďže je omnoho zložitejšie naimplementovať GPU verziu algoritmu, detekčný mód Viola-Jones kaskády dokáže na GPU detekovať objekty len ak je model zapísaný v starej verzií modelu. Našťastie to nie je problém, keďže je možné kaskádu reštartovať a prikázať zápis do starého modelu, a naopak existuje utilita, ktorá transformuje modelu do nového formátu. V prípade LBP deskriptora GPU verzia podporuje novú verziu modelu.

GPU implementácie sú podstatne komplikovanejšie ale taktiež omnoho rýchlejšie ako CPU verzie. Pre jeden 2048x1536 obrázok trvá detekcia na CPU 0.76 sekundy a detekcia na GPU 0.11 sekundy. Ako vidieť GPU implementácie sú

niekoľkonásobne rýchlejšie. Je potrebné spomenúť dva dôležité fakty. CPU implementácia obsahuje voliteľný kód používajúci knižnicu TBB – Threading building blocks od Intelu, ktorá je voľne dostupná a umožňuje veľmi efektívnu paralelizáciu na CPU. Zdrojové kódy je potrebné skompilovať s touto knižnicou, už zbalené distribúcie knižnice nemajú túto možnosť zapnutú. GPU implementácia vyžaduje buď grafickú kartu podporujúcu OpenCL platformu alebo grafickú kartu podporujúcu CUDA platformu. Taktiež ako v prípade TBB je potrebné si skompilovať knižnicu OpenCV s touto možnosťou zapnutou.

Hlavným dôvodom, prečo Viola-Jones kaskáda je dodnes používaná, je jednoduchosť jej rozšírenia. Oba módy, aj tréningový aj detekčný obsahujú rozhrania, ktoré umožňujú relatívne bezzásahovo rozšíriť kaskádu o ďalšie deskriptory. Kvôli zložitosti a rôznym požiadavkám módov je pre každý mód odlišné rozhranie.

V testovacom móde sa stará rozhranie o zobrazenie nastavení, ich načítanie a taktiež ich zápis do modelu. Následne sa stará o výpočet deskriptoru vo výpočtovom kroku a sprístupnenie hodnôt v klasifikačnom kroku. Pre úspešnú implementáciu nového deskriptoru je potrebné zdediť nasledovné dve triedy:

- CvFeatureParams – táto trieda sa stará o parametre pre daný deskriptor. V prípade Haarových vlniek to je napríklad ktorý set vlniek sa má používať, či základný, rozšírený alebo rozšírený aj s naklonenými vlnkami.
- CvFeatureEvaluator – táto trieda sa stará o samotný výpočet deskriptoru a jeho sprístupnenie počas klasifikačnej fázy. Taktiež sa stará o zápis vybraných čŕt do modelu kaskády.

Trieda CvFeatureParams má nasledujúce virtuálne metódy, ktoré je možné preťažiť:

- printDefaults – vypíše na obrazovku všetky možné nastavenia deskriptoru pri nezadaní žiadneho parametra do kaskády
- printAttrs – vypíše na obrazovku vybrané parametre pri úspešnom spustení tréningu kaskády

- scanAttrs – načíta z príkazového riadku zadané parametre pre daný deskriptor
- init – inicializuje všetky potrebné premenné potrebné pre začiatok trénovania kaskády
- write – zapíše zvolené parametre kaskády do natrénovaného modelu
- read – načíta zapísané parametre kaskády z natrénovaného modelu v prípade pokračovania tréningu

Je potrebné spomenúť, že v prípade ak nepotrebujeme preťažovať ani jednu z týchto metód, stále musíme vytvoriť potomka tejto triedy a nastaviť mu správne meno cez konštruktor u predka. Napríklad ak náš deskriptor je statický a nemá žiadne nastavenia, ako v prípade implementácie HOG deskriptora.

Trieda CvFeatureEvaluator má nasledovné virtuálne metódy:

- init – inicializuje všetky potrebné premenné pre začiatok trénovania podľa parametrov. Taktiež sa v tomto kroku zvykne vykonávať všetka alokácia potrebná pre začiatok tréningu.
- generateFeatures – vygeneruje všetky rôzne črty dostupné pre danú veľkosť výseku pre daný deskriptor. V prípade Haarových vlniek sú to státisíce až milióny, v prípade HOG deskriptora tisícky.
- writeFeatures – zapíše všetky črty vybrané tréningom do natrénovaného modelu kaskády
- setImage – nastaví na danú pozíciu výsek s ktorým chceme pracovať. Táto metóda prestavuje výpočtový krok výpočtu deskriptora kaskády v tréningovom móde. Táto metóda predpočíta všetky možné hodnoty potrebné pre klasifikáciu. V prípade Haarových vlniek sa v tomto kroku počíta integrálny obraz a v prípade HOG deskriptora sa počíta celý deskriptor naraz.
- operator() – vypočíta alebo len vráti hodnotu špecifickej črty pre daný výsek. Táto metóda prestavuje klasifikačnú časť výpočtu deskriptora kaskády v tréningovom móde.

Trieda CvFeatureEvaluator taktiež zvykne obsahovať vlastnú podtriedu s názvom Feature, ktorá sa stará o lokalizovaný výpočet čŕt a lokalizovaný zápis čŕt na disk. Samozrejme je možné implementovať túto triedu bez tejto podtriedy, ale v prípade exportovania vlastnej implementácie je potrebné čo najviac dodržiavať štýl programovania stanoveného v tréningovom móde. V implementácí potomka triedy je potrebné naimplementovať všetky spomenuté metódy a taktiež je potrebné správne volať niektoré metódy predka v preťažených metódach aby bol zabezpečený správny priebeh tréningu kaskády.

V detekčnom móde sa rozhranie stará o načítanie čŕt a nastavení deskriptora podľa modelu. Toto rozhranie je využívané počas metódy detectMultiScale na výpočet hodnôt deskriptora a ich indexovanie. Rozhranie predstavuje trieda FeatureEvaluator. Potomok tejto triedy má za úlohu čo najrýchlejšie a najefektívnejšie počítať a sprístupňovať hodnoty čŕt.

Trieda FeatureEvaluator má nasledovné virtuálne metódy:

- read – načíta dáta pre črty a deskriptor z modelu
- clone – vytvorí kópiu FeatureEvaluatora
- getFeatureType – vráti typ deskriptora
- setImage – táto metóda sa normálne nepreťažuje. Stará sa o správu rôznych škál obrazu, v ktorom chceme detekovať objekty. Taktiež sa tu nachádza alokácia pamäte pre dané škály. V prípade že všeobecná alokácia nám nestačí máme možnosť pozmeniť túto metódu.
- setWindow – oznámi deskriptoru s ktorým výsekom okna práce pracujeme. V tejto metóde sa prepočítavajú offsety pre pamäťové prístupy.
- getMats/getUMats – vráti dáta v ktorých sa nachádza predpočítaný deskriptor
- calcOrd/calcCat – momentálne nevyužívané funkcie. Sú prítomné pre prípad, že by sme chceli implementovať deskriptor s kategorickými a ordinálnymi hodnotami a pridať podporu pre tento mechanizmus do kaskády.
- create – vytvorí novú inštanciu partikulárneho FeatureEvaluatora

- computeChannels – táto metóda predstavuje výpočtový krok kaskády. Slúži na výpočet všetkých hodnôt deskriptora tak ako v trénovacom móde na to slúži metóda setImage. Hlavný rozdiel oproti trénovacému módu je potrebná logika pre prácu s rôznymi škálami obrazu.
- computeOptValues – táto metóda predstavuje klasifikačný krok kaskády. Slúži na sprístupnenie danej črty pre práve nastavený výsek.

Implementácia detekčného rozhrania je podstatne náročnejšia kvôli potrebnej réžií pre rôzne škály obrazu. Pokročilejšia logika pri alokácií pamäte spôsobuje, že toto rozhranie je taktiež náročnejšie na pochopenie. Potomkovia FeatureEvaluator v prípade Haar a LBP implementácií používajú vnútorne dve triedy: Feature a OptFeature. Trieda Feature predstavuje to isté čo v trénovacom móde a taktiež slúži na načítanie z modelu. Trieda OptFeature slúži na samotný výpočet čŕt počas klasifikácie. Dôvod prečo je použitá druhá separátna trieda sú offsety. Pri pohybe v obraze prostredníctvom detekčných okien, je potrebné správne posúvať prístupy do pamäte. Preto sa pre každú Feature vygeneruje jedna OptFeature, ktorej vždy pri posune do ďalšieho okna prepočítame offsety. Dôvod prečo sa tento proces nerobí priamo pri výpočte črty je vnútorná optimalizácia procesora. Ak prepočítame všetky črty naraz, procesor v spojení s kompilátorom môže preorganizovať výpočty tak aby bežali omnoho rýchlejšie ako keby sa tieto výpočty robili izolovane pri prístupe k črte.

Napriek tomu, že Viola-Jones kaskáda je veľmi obsiahly framework, je pár vecí, ktoré chýbajú v tejto modernej kaskáde. Jednou takouto chýbajúcou časťou je možnosť využiť rôzne deskriptory v tej istej kaskáde. Viaceré skupiny, ktoré pracovali na detekcií obrazu pomocou kaskádového frameworku využívali jednoduchšie deskriptory na začiatku kaskády, napríklad Haarové vlnky, a komplikovanejšie deskriptory na konci kaskády, napríklad HOG deskriptor [6].

Ďalšou takouto chýbajúcou súčasťou je zapojenie ľubovoľného slabého klasifikátora. Táto implementácia má dostupný jeden slabý klasifikátor – rozhodovacie stromy. Treba ale spomenúť, že tento slabý klasifikátor má implementovanú optimalizáciu, v prípade že stromy degenerujú na pne, kedy je klasifikácia podstatne rýchlejšia. Toto je v kontraste voči knižnici MultiBoost, ktorá obsahuje jednoduché

rozhrania na rozšírenie o ďalší slabý klasifikátor. Ďalšie slabé klasifikátory boli taktiež odporúčané v práci SHOGu pri testovaní kaskády, a to konkrétne SRB learner [6].

Trénovací mód kaskády má niekoľko nedostatkov, ktoré podstatne spomaľujú priebeh tréningu. Počas validačného kroku, kedy kontrolujeme či kaskáda má pokročiť do ďalšieho kroku, kontrolujeme odozvu doteraz natrénovanej časti kaskády, či zodpovedá parametrom ktoré sme nastavili na začiatku tréningu. Tento proces prebieha v jednom vlákne, zatiaľ čo je ho možné perfektne paralelizovať.

Ďalším nedostatkom trénovacieho módu je vysekávanie negatívnych vzoriek pre tréning. Ako vstup kaskády sú obrázky čistého pozadia, čiže neobsahujú žiadne pozitívne nálezy hľadaných objektov. Tento mechanizmus slúži na ľahké získavanie pozadí vysekaním z týchto negatívov. Problém nastáva neskôr v priebehu kaskády kedy akceptujeme menej ako 0.0001 false-negatives, tzn. nesprávnych nálezov. V tomto bode vysekávanie negatívnych vzoriek trvá hodiny a v prípade rigorózneho trénovania dokonca až dni.

Posledným viditeľným problémom implementácie trénovacieho módu je nekonzistencia využitia TBB knižnice. Táto knižnica sa aktivuje použitím #define HAVE_TBB, kedy paralel_for cyklus využije tbb::paralel_for na paralelizáciu výpočtov. Keďže ale táto knižnica sa stále vyvíja, kód trénovacieho módu používa „paralel_for“, ktorý je zadefinovaný v tejto samostatnej aplikácií, a ak je zadefinovaný, používa HAVE_TBB. Taktiež ale existuje „paralel_for_“ s podčiarkovníkom navyše, a tento je zdedený z hlavnej knižnice OpenCV. Treba si preto dať pozor ako kompilujeme knižnicu a túto separátnu aplikáciu, pretože to môže spôsobiť, že nie plne využívame paralelizačné možnosti.

2.5 Spustenie kaskády

Spustenie Viola-Jones kaskády vyžaduje oboznámenie sa s parametrami a prípravu určitých dát. Samotný tréning kaskády je aplikácia s názvom opencv_traincascade.exe, ktorá sa buď nachádza v distribuovaných balíčkoch, alebo taktiež je možné si túto aplikáciu osobitne skompilovať, ako už bolo spomínané v predošlých kapitolách.

Medzi hlavné tri vstupy patria pozitívne vzorky, negatívne vzorky a priečinok na výstupy kaskády. Negatívne vzorky sú obrázky pozadia, tzn. obrázky v ktorých nie sú žiadne hľadané objekty

Pozadia sa zvyknú dodávať v čo najväčších obrázkoch, pretože kaskáda dokáže vyrábať výseky, ktoré následne použije ako negatívne vzorky správnych veľkostí počas behu. Tieto obrázky pozadí sa uložia do textového súboru, ktorý obsahuje relatívne/absolútne cesty k nim. Jeden riadok predstavuje jeden obrázok pozadia.

Obrázok 13 – Príklad pozadia

Pozitívne vzorky je možné predpripraviť rôznymi spôsobmi. Mnou zvolený spôsob bol vytvoriť súbor info.dat, ktorý obsahuje zoznam všetkých obrázkov s cestami tak ako pri negatívnych vzorkách. Navyše ale tento súbor obsahuje pozíciu hľadaného objektu v danej vzorke. Jeden obrázok/vzorka môže obsahovať viac objektov naraz. Tento súbor sa následne pošle do programu opencv_createsamples.exe, ktorý je taktiež pribalený alebo kompilovateľný. Výsledkom je jeden binárny vec súbor, a ten obsahuje všetky potrebné dáta a popisy ako ich má kaskáda načítať. Ako vstupný parameter pre vytváranie vzoriek je nastavenie veľkosti detekčného okna, keďže vzorky sa budú resizovať na tento rozmer. Táto vedľajšia aplikácia má ďalšie vstupné parametre, ale tie nie sú potrebné, keďže bol dostupný vlastný anotačný nástroj.

Samotná kaskáda má viacero dôležitých parametrov:

- numPos – počet použitých pozitívnych vzoriek pri tréningu. Je potrebné nastaviť také číslo, aby po rôznych krokoch kaskády nedošli pozitívne vzorky. Na začiatku každého kroku sa hľadajú vzorky, ktoré by prešli až do daného kroku kaskády. V prípade, že nám dôjdu vzorky, aplikácia skončí chybou, keďže nemôže pokračovať v tréningu.
- numNeg – počet použitých negatívnych vzoriek pri tréningu. Platí rovnaké pravidlo ako pri pozitívnych vzorkách, nesmú nám dôjsť výseky pozadí. Keďže sa ale zvyknú dodávať pozadia prostredníctvom veľkých obrázkov, v našom prípade 2048x1536 rozlíšenie, máme k dispozícií milióny potenciálnych výsekov. Treba ale spomenúť, že v neskorších krokoch kaskády, zvýšenie tohto parametra výrazne spomalí beh tréningu, kvôli chýbajúcej optimalizácií.
- numStages – jedno zo zastavovacích kritérií. Tréning kaskády sa zastaví a finálny model sa zapíše na disk ak prekročíme maximálny počet krokov.
- precalcValBufSize a precalcIdxBufSize – veľkosti bufferov v Mb, ktoré špecifikujú, koľko rôznych čŕt a indexov možno predpočítať pred započatím tréningu kroku kaskády. Čím viac hodnôt sa predpočíta, tým rýchlejší je trénovací proces.
- baseFormatSave – tento parameter špecifikuje uloženie do starého formátu aplikácie opencv_haarcascade, z ktorej opencv_traincascade vznikla. Tento starší formát je jediný spôsob ako spustiť detekciu pomocou Haarových vlniek na CUDA GPU.
- acceptanceRatioBreakValue – ďalšie zo zastavovacích kritérií. Počas vysekávania pozadí, počítame koľko hodnôt kaskáda spotrebovala aby dosiahla aspoň numNeg počet vzoriek. V prípade že pomer počtu použiteľných výsekov a počtu všetkých vyskúšaných výsekov klesne pod hodnotu tohto parametra, tréning kaskády skončí. Tento parameter má za úlohu zabezpečiť aby sme netrénovali príliš veľa, čo by mohlo spôsobiť pretrénovanie modelu kaskády.

- featureType – parameter špecifikuje deskriptor, ktorý chceme použiť na generovanie čŕt. Dostupné možnosti sú HAAR, HOG, a LBP. V ďalších kapitolách bude popisovaný náš vlastný SHOG.
- w – šírka detekčného okna. Musí odpovedať nastaveniu vec súboru, ktorý bol výstupom aplikácie opencv_createsamples.
- h – výška detekčného okna. Taktiež musí odpovedať nastaveniu vec súboru.
- Bt – typ použitej modifikácie AdaBoostu. Dostupné možnosti sú DiscreteBoost(DAB), RealBoost(RAB), GentleBoost(GAB) a LogitBoost(LAB)
- mode – v prípade Haarových vlniek, môžeme špecifikovať aké typy vlniek použijeme. Basic – vlnky z práce Viola-Jones, Core – všetky bežné normálne vlnky, Tilted – všetky vlnky vrátane naklonených
- minHitRate – minimálne percento akceptovania pozitívnych vzoriek v jednom kroku kaskády. V prípade, že nastavíme 0.995, zo vstupných 400 pozitívnych vzoriek, musí krok kaskády akceptovať 99.5% z nich. Celkový hitrate kaskády dostaneme ako minHitRate*(počet krokov kaskády). Východzie nastavenie je 0.995.
- maxFalseAlarmRate – maximálne percento nesprávnych rozhodnutí na pozadiach. Východzie nastavenie je 0.5, tzn. z 1000 vstupných negatívnych vzoriek pre daný krok, musíme rozhodnúť o menej ako 50%, že sú to pozitívny nález. Celkový falseAlarmRate kaskády rozhodne o tom, koľko nesprávnych nálezov budeme dostávať pri detekcií. Dobrý celkový falseAlarmRate pre celú kaskádu je okolo 0.00001. Tento parameter dokáže pri príliš nízkom čísle spôsobovať pretrénovanie kaskády.
- weightTrimRate – tento parameter rozhoduje či použijeme trimming a s akou hodnotou.
- maxDepth – maximálna hĺbka rozhodovacích stromov. Východzie nastavenie je 1, čo prestavuje peň. Tie sú v kaskáde optimalizované tak,

aby rozhodovanie prebiehalo čo najrýchlejšie, keďže nie je potrebné používať nijaké cykly.

- maxWeakCount – tento parameter špecifikuje maximálny počet slabých klasifikátorov na jeden krok kaskády. Čím väčší je, tým lepšie je možné natrénovať krok kaskády ale aj tým pomalší je krok kaskády.

Rôzne nastavenia kaskády umožňujú prispôsobenie tréningu kaskády na danú úlohu. Za účelom testovania vlastného deskriptora voči Haarovým vlnkám som zvolil východzie parametre kaskády, okrem nasledujúcich parametrov: -w a -h som nastavil na rozmer 20px a 50px, pretože vo vstupných vzorkách bežný vzpriamený futbalista má tento pomer strán. Menšie detekčné okná nemajú zmysel, pretože by nebolo možné vygenerovať dostatočný počet čŕt. -bt – bola zvolená modifikácia AdaBoostu RealBoost. V prácach, ktoré sme spomínali v teoretickej časti, to bola vždy zvolená modifikácia [4] [5].-numPos bolo nastavené na 400, aby sme mali aj detailný model a aj relatívne rýchly tréning (do niekoľko hodín). Ostatné parametre nie sú potrebné pre vyhodnotenie vlastného deskriptora.

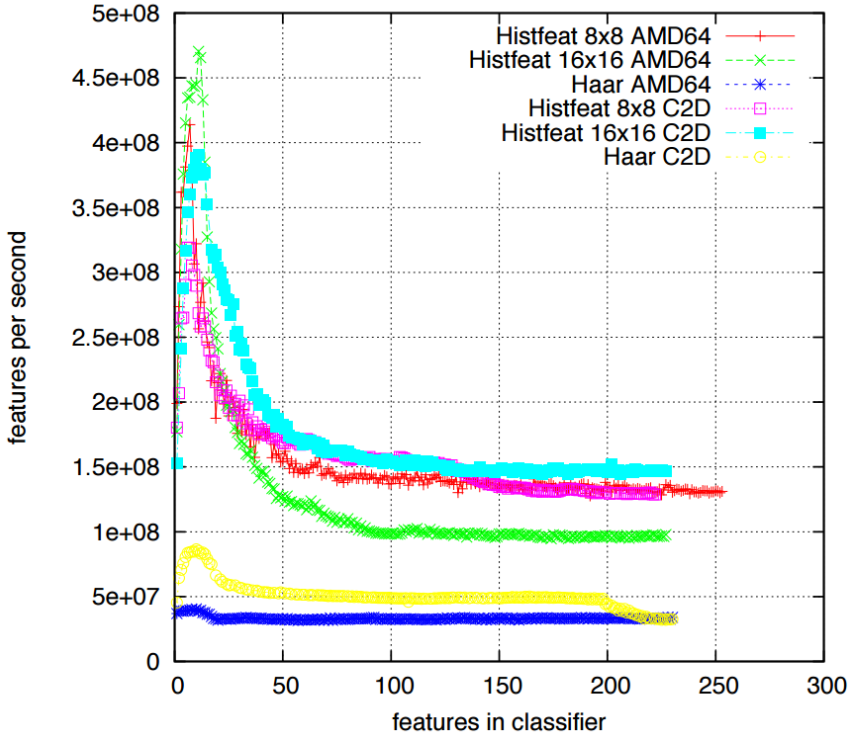
2.6 Výber deskriptora na našu úlohu

Ako už bolo spomínané v teoretickej časti, dôležitými parametrami pre výber deskriptora sú pre nás rýchlosť a kvalita detekcie v spojení s AdaBoostom. Prvé aplikácie boostingu s spojení s deskriptormi, boli viac menej úspešné v dobe, kedy počítače a možnosti boli omnoho pomalšie. Práca Viola-Jones, z ktorej vzišla ich kaskáda a taktiež Haarové vlnky, definovala detekcie v reálnom čase ako algoritmus, ktorý dokáže spracovať aspoň 2 snímky videa za sekundu. Aj keď tento výsledok v prípade kaskády závisí od rôznych parametrov, ako počet krokov kaskády, počet slabých klasifikátorov a pod., tento cieľ sa im podarilo dosiahnuť. O čo viac, tento framework sa v rozšírených formách používa dodnes.

Ďalší dôležitý krok vo vývoji rýchlosti deskriptorov bol HOG deskriptor, ktorý bol najprv použitý samostatne (Dalal-Triggs) [2] a neskôr v spojení s AdaBoost kaskádou a SVM klasifikátorom (Zhu et al.) [4]. Prvá iterácia dokázala spracovať jeden 320x240 obrázok za sekundu (jeden obrázok potreboval vyhodnotenie okolo 800 detekčných

okien) a podľa dnešných potrieb by nebola využiteľná v bežných aplikáciách. Druhá iterácia spojila koncept HOG deskriptora s kaskádou podobnou Viola-Jones frameworku, čo znamenalo podstatné zníženie priemerného počtu blokov na jedno detekčné okno. Dalal-Triggs verzia musela v priemere vyhodnotiť 105 blokov HOG deskriptora na jedno detekčné okno, zatiaľ čo Zhu et al. musela vyhodnotiť v priemere 4.6 blokov HOG deskriptora. Už len v tomto jednom kroku to je viac ako 20x zrýchlenie konceptu, čo v úplnom porovnaní rýchlostí znamenalo dosiahnutie 70x rýchlejšej klasifikácie. Obe tieto práce testovali výsledky na detekcií ľudí. Zhu et al. taktiež porovnali natrénovanú Viola-Jones kaskádu s Haarovými vlnkami voči vlastnej HOG kaskáde. S použitím rovnakého datasetu, výsledný klasifikátor mal veľmi nízky hit rate okolo 50%, čo spôsobilo, že nebol použiteľný na danú úlohu. HOG kaskáda s rovnakými parametrami dosahovala 88% hit rate. Zhu et al. taktiež vykonali štatistickú analýzu Haarových vlniek a HOG deskriptora, ktorá ukázala, že HOG deskriptor popisuje obraz omnoho stabilnejšie.

Tieto poznatky vytvorili základ pre Pettersson et al. [7], kde pokračovali v úprave HOG deskriptora za použitia kaskády. Aj keď HOG deskriptor vykazoval omnoho lepšie popisovacie vlastnosti, stále bol pomalší oproti veľmi jednoduchým Haarovým vlnkám. Pettersson et al. sa postavili k HOG deskriptoru ako k Haarovým vlnkám, tzn. hodnoty histogramov sú chápané ako jedna črta, a je ich možné tak aj adresovať. Ako slabý klasifikátor boli použité *a posteriori*, ktoré zobrali hodnoty histogramov danej bunky a urobili rozhodnutie o triede objektu. Tento unikátny prístup k HOG deskriptoru podstatne zrýchlil výpočtový krok vynechaním normalizácie blokov, zvýšil počet čŕt, ktoré deskriptor vie vygenerovať, zmenšením bunky a zmenšením posunu a taktiež zamenil omnoho pomalší SVM klasifikátor za podstatne jednoduchšie *a posteriori* tabuľky. Na obrázku 14 môžeme vidieť o koľko rýchlejší je výpočet HistFeat čŕt. Vysoký nárast na ľavej strane je spôsobený cachovaním hodnôt čŕt, kedy v prípade dostatočne veľkej cache, sú prístupy k hodnotám čŕt omnoho rýchlejšie. Pettersson et al. pripisovali tieto výsledky hlavne redukcií prístupov do pamäte počas klasifikačného kroku, tzn. z 4 až 9 prístupov na 1 a taktiež presunutiu všetkých výpočtov na výpočtový krok, kedy kompilátor a procesor môžu vektorizovať/paralelizovať výpočty.



Obrázok 14 – Porovnanie rýchlosti Haarových vlniek a HistFeat klasifikátora

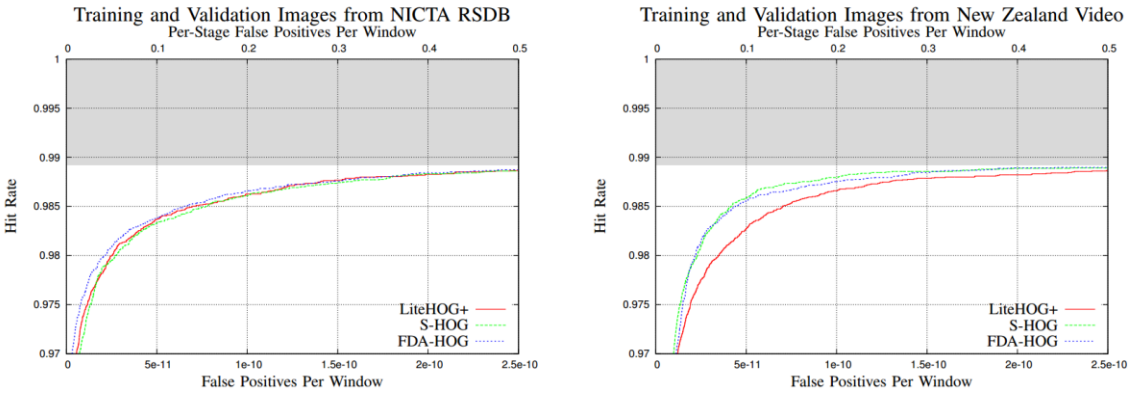
Zdroj: The Histogram Feature – A Resource-Efficient Weak Classifier, Pettersson et al., NICTA 2008

V ďalšej práci nadväzujúcej na poznatky z HistFeat deskriptora, Overett et al. [5] vyhodnotili HistFeat ako úspech ale taktiež poznamenali, že HistFeat nevyužíva všetky výpočtové zdroje procesora. HistFeat je perfektne stavaný na detekciu na začiatku kaskády, kedy sú potrebné extrémne rýchle črty. Pokiaľ ale používame HistFeat na opačnom konci kaskády, klasifikátor má tendenciu sa pretrénovať, pretože musíme použiť príliš veľké číslo čŕt v posledných krokoch kaskády.

Cieľom ich nasledujúcich pokusov bolo vytvoriť komplikovanejší deskriptor, ktorý by bolo možné použiť v tandéme s HistFeat deskriptorom na konci kaskády. Takýmito pokusmi boli LiteHOG a LiteHOG+, a neskôr SHOG a FDA-HOG z nadväzujúcej práce na túto. [6]

V rámci našej práce nie je potrebné spomínať LiteHOG pretože z výsledkov Overett et al. je LiteHOG+ deskriptor na celej čiare lepší. LiteHOG+ využíva poznatky HistFeat, a to rovnaký výpočet histogramu, s tým že jednotlivé hodnoty histogramov nedávame do *a posteriori* tabuľky, ale pomocou Fisherovej diskriminačnej analýzy (FDA), dostaneme z N-rozmerného priestoru, ktorý je adresovaný cez koše histogramu,

1-dimenzionálnu odpoveď, ktorú môžeme použiť ako vstupnú hodnotu pre AdaBoost. Tento deskriptor podáva veľmi dobré detekčné výsledky na detekcií chodcov, ale aj na iných úlohách vykazuje zlepšenie oproti kaskáde natrénovanej výhradne na HistFeat deskriptore.



Obrázok 15 – Porovnanie výsledkov jedného silného kroku kaskády LiteHOG+, SHOG a FDA-HOG

Zdroj: "Large scale sign detection using HOG feature variants," Overett, Petersson, NICTA, 2011

V práci SHOG a FDA-HOG deskriptorov, Overett-Petersson sa snažili nájsť alternatívu k LiteHOG+ deskriptoru. V tejto práci sa výhradne zaoberajú nájdením deskriptora, ktorý by bol použiteľný na konci kaskády, kde treba robiť detailné rozhodnutia. Hlavnou zmenou týchto deskriptorov oproti LiteHOG+ je ukladanie celých hodnôt histogramov. V HistFeate a LiteHOG+ sa hodnoty gradientov osekávajú na {0,1} podľa stanovenej hranice, zatiaľ čo v SHOGu a FDA-HOGu sa berú celé hodnoty a tie sa následne ukladajú do integrálneho obrazu, aby bolo veľmi jednoduché počítať hodnoty buniek histogramov. Tento proces umožňuje používať jednu orientáciu histogramovej bunky ako jednu črtu pre AdaBoost v prípade SHOGu a taktiež v spojení s FDA vytvoriť z celého histogramu popisnejšiu hodnotu. Na obrázkoch možno vidieť, že SHOG a FDA-HOG majú porovnateľné výsledky, ale taktiež sú podstatným a viditeľným vylepšením voči LiteHOG+. Tieto poznatky umožnili posúdiť rôzne deskriptory a preto som sa rozhodol implementovať SHOG deskriptor do Viola-Jones kaskády v OpenCV.

2.7 Implementácia deskriptora

Cieľom tohto procesu je implementácia SHOG deskriptora v trénovacom a testovacom móde. Implementáciu som rozdelil do troch krokov tak aby bolo čo najjednoduchšie verifikovať jej výpočty: prototyp výpočtu, trénovacie rozhranie, testovacie rozhranie.

Výpočet SHOG deskriptora sa skladá z troch krokov + alokácia pamäte. Polia potrebné na tento výpočet sú vcelku jednoduché. Prvé pole je potrebné na uloženie gradientov pre každú orientáciu. Jeho rozmery sú (počet riadkov)*(počet stĺpcov)*(počet orientácií=8). Druhé pole je potrebné na výpočet integrálneho obrazu a jeho rozmery sú (počet riadkov + 1)*(počet stĺpcov + 1)*(8). Tretie pole je potrebné na uloženie výsledku SHOGa. Jeho rozmery sú trochu viac komplikované, (počet riadkov/krok riadkov – výška bunky)*(počet stĺpcov/krok stĺpcov – šírka bunky). V prípade SHOG deskriptora, krok je nastavený na statických [1,1] a rozmery bunky na [4,4]. Napriek tomu som tento výpočet zakomponoval v rámci dobrého a nastaviteľného dizajnu pomocou #define.

Prvý krok výpočtu je výpočet gradientu. Gradient v horizontálnom a vertikálnom smere nie je potrebné nikam ukladať, takže priamo z neho vypočítam orientáciu a magnitúdu. Orientácia je daná vzorcom (dy < 0) * 4 + (dx < 0) * 2 + (absol(dy) > absol(dx)) * 1, kde dx predstavuje gradient v horizontálnom smere a dy gradient vo vertikálnom smere. Magnitúda je daná vzorcom absol(dx) + absol(dy). Magnitúdu zapisujem do prvého poľa na danú pozíciu v obraze podľa vypočítanej orientácie.

Druhý krok výpočtu je výpočet integrálneho obrazu. V tomto prípade je v knižnici OpenCV dostupná implementácia, ktorú pomocou trochy objektového zaobaľovania našich polí, viem použiť bez hocijakej potreby kopírovania pamäte. Integrálny obraz počítam paralelne pomocou tbb::paralel_for. V jednom vlákne počítam jednu orientáciu integrálneho obrazu. Výsledok je zapísaný do druhého nami naalokovaného poľa.

Tretí krok výpočtu je výpočet histogramových buniek. Pomocou integrálneho obrazu sú na výpočet ľubovoľnej bunky potrebné 4 hodnoty. Majme bunku so

súradnicami x,y a rozmermi a,b. Bunku vypočítame nasledujúcim vzorcom: integrálnyObraz[x][y] + integrálnyObraz[x + a][y + b] - integrálnyObraz[x + a][y] - integrálnyObraz[x][y + b].

Vďaka implementácií prototypu, pridanie tohto výpočtu je len otázka zmeny cieľového poľa a perzistencia cieľového poľa. Preto som sa mohol venovať potrebám trénovacieho rozhrania. SHOG nemá žiadne parametre, takže táto možnosť sa mohla preskočiť. Bolo potrebné vymyslieť adresovanie čŕt SHOG deskriptora. Ja som zvolil ako adresovanie súradnice v rámci poľa histogramu. Z toho mi vychádzajú tri parametre, keďže pole je segmentované ako trojrozmerné pole: x, y, orientácia. Zvyšok implementácie testovacieho rozhrania je len použitie logiky nájdenej v implementáciách rozhraní pre Haarové vlnky a LBP deskriptor.

Rozhranie v detekčnom móde je viac náročné na implementáciu, pretože je viac náročné na pochopenie. V princípe výpočet deskriptora je identický, tam kde v trénovacom rozhraní zapisujeme do modelu, tu čítame z modelu ale tam podobnosť končí.

Tento mód obsahuje dva ďalšie koncepty, OptFeature a SetWindow. OptFeature sú črty, ktoré sa dajú znovu využiť na adresovanie v rámci poľa histogramu aj v prípade viacerých škál (tréning pracuje len s jednou škálou). Preto sú možné dva prístupy, buď a) je možné používať riedke polia, a takýmto adresovaním predstierať, že každá škála má rovnaké rozmery alebo b) treba prepočítavať OptFeature. V referenčných implementáciách je využitá prvá možnosť, ja som sa rozhodol pre druhú možnosť, keďže výsledný algoritmus je omnoho jednoduchší na pochopenie a dopad na výkon je zanedbateľný.

SetWindow je metóda, ktorá počas klasifikačného kroku nám povie, kde práve v obraze sme a odkiaľ máme počítať hodnoty. Tu je použitý jednoduchý prístup prepočítavania smerníka. Táto logika zabezpečí, že napriek tomu že posielame črte len samotný smerník na pole, tento smerník nezačína v ľavom hornom rohu obrazu, ale v ľavom hornom rohu detekčného okna. Táto logika spôsobí, že reálne treba OptFeature prepočítať medzi škálami, a nie pri prechode medzi detekčnými oknami, a taktiež nie je potrebné vykonávať aritmetiku výpočtu miesta v poli. Aj keď tento

spôsob adresovania poľa je veľmi rýchly, má to za výsledok, že je celkom zložitý na pochopenie, ak čitateľ nemá presné znalosti algoritmu.

Po implementácií výpočtu deskriptora, a implementácií rozhraní, nasleduje vyhodnotenie činnosti deskriptora. Počas implementovania som pracoval s Visual Studio performance analyzerom, aby som našiel potenciálne spomalenia algoritmu. Hlavným problémom s ktorým som sa stretol bolo klonovanie FeatureEvaluator, ktoré robí OpenCV interne. Bolo potrebné zaobaliť polia do std::shared_ptr, aby sa klonoval len smerník, a nie celé pole.

Počas implementácie vlastného deskriptora sa mi taktiež podarilo podstatne zrýchliť vysekávanie negatívnych vzoriek počas tréningu kaskády. Pôvodný algoritmus na vysekávanie z obrázkov pozadí, som nahradil využitím detekčného módu kaskády, čo spôsobilo, že samotný krok vysekávania sa znížil z niekoľkých hodín na konštantný čas, podľa počtu obrázkov pozadí. Jediný problém, je potreba záložného mechanizmu, v prípade, že detekčný mód nevráti dosť negatívnych vzoriek, keď už kaskáda je veľmi dobre natrénovaná. Na to sa dá využiť samotný pôvodný pomalý vysekávač.

2.8 Test deskriptora

Implementácia SHOG deskriptora bude porovnávaná s implementáciou Haarových vlniek pomocou VJ kaskády. Hlavné kategórie sú rýchlosť detekcie a kvalita detekcie.

Problém s vyhodnotením rýchlosti detekcie je, že v prípade kaskádového prístupu, rýchlosť závisí od počtu krokov kaskády, hĺbky stromov a počtov slabých klasifikátorov. Tieto rôzne parametre priamo závisia od úlohy počas ktorej sa snažíme rozpoznávať objekty.

Vyhodnocovanie rýchlosti som preto vykonával na podobne natrénovaných modeloch, kde boli veľmi podobné počty spomínaných parametrov. Pre obrázky veľkosti 2048x1536 trvala multiškálová detekcia v prípade Haarových vlniek v priemere 0.15 sekundy, zatiaľ čo v prípade SHOG deskriptora trvala okolo 0.5 sekundy.

Problém s takýmto porovnávaním je ale ten, že vo výsledku nám nie vždy záleží ako rýchlo dokážeme vypočítať jednu hodnotu slabého klasifikátora, ale ako dlho trvá celá detekcia s podávaním rovnakých výsledkov. Preto som taktiež vykonal druhý rýchlostný test, kde Haarové vlnky a SHOG deskriptor podávali podobné výsledky detekcie a vtedy trvala detekcia rovnakých 0.19 sekúnd pre Haarové vlnky, ale pre SHOG bola podstatne pomalšia, okolo 1.3 sekundy.

Dôvod prečo nastal tento stav je, že výpočet SHOGu vyžaduje v každom kroku kaskády väčší počet slabých klasifikátorov. To spôsobuje viac vyhodnocovania počas detekčného kroku, aj keď je potrebné omnoho menej výpočtov v tomto kroku oproti Haarovým vlnkám. Treba ale spomenúť, že autori práce SHOG deskriptora [6] navrhovali tento deskriptor hlavne na jednoduchšie architektúry, čo znamená podstatnú nevýhodu na CPU.

Analýza rýchlosti pomocou profilera ukázala dva slabé body v ktorých by bolo možné zrýchliť beh algoritmu. Prvý problém je, že SHOG implementácia viac využíva vlákna ako implementácia Haarových vlniek. V prípade statickej alokácie vlákien, by bolo pravdepodobne možné znížiť overhead, ktorý predstavuje vytváranie vlákien počas výpočtu deskriptora. Na druhej strane to ale taktiež znamená, že sa tento výpočet ľahko paralelizuje oproti Haarovým vlnkám, čo bolo aj cieľom nášho hľadania.

Druhý problém je klonovanie detekčných objektov. V prípade Haarových vlniek, potomok triedy FeatureEvaluator je celkom jednoduchý a jeho klonovanie nevyžaduje žiadne zložitejšie operácie. Keďže je ale SHOG viac komplikovaná štruktúra, klonovanie tejto triedy predstavuje určité spomalenie samotného algoritmu, ktoré by bolo možné zásahmi do volaní AdaBoostu podstatne znížiť.

Na test rýchlosti priamo nadväzuje test kvality detekcie. Na toto bola využívaná časť aplikácie, ktorá porovnáva nájdené obdĺžniky detekovaných objektov so výstupným XML anotačného nástroja. Tento program porovnáva prelínanie nájdených obdĺžnikov s obdĺžnikmi, ktoré boli anotované v nástroji a podľa toho vyhodnotí štatistiku štyroch hodnôt: Počet všetkých objektov, počet správnych nálezov, počet nenájdených objektov a počet chybných nálezov (false-positives).

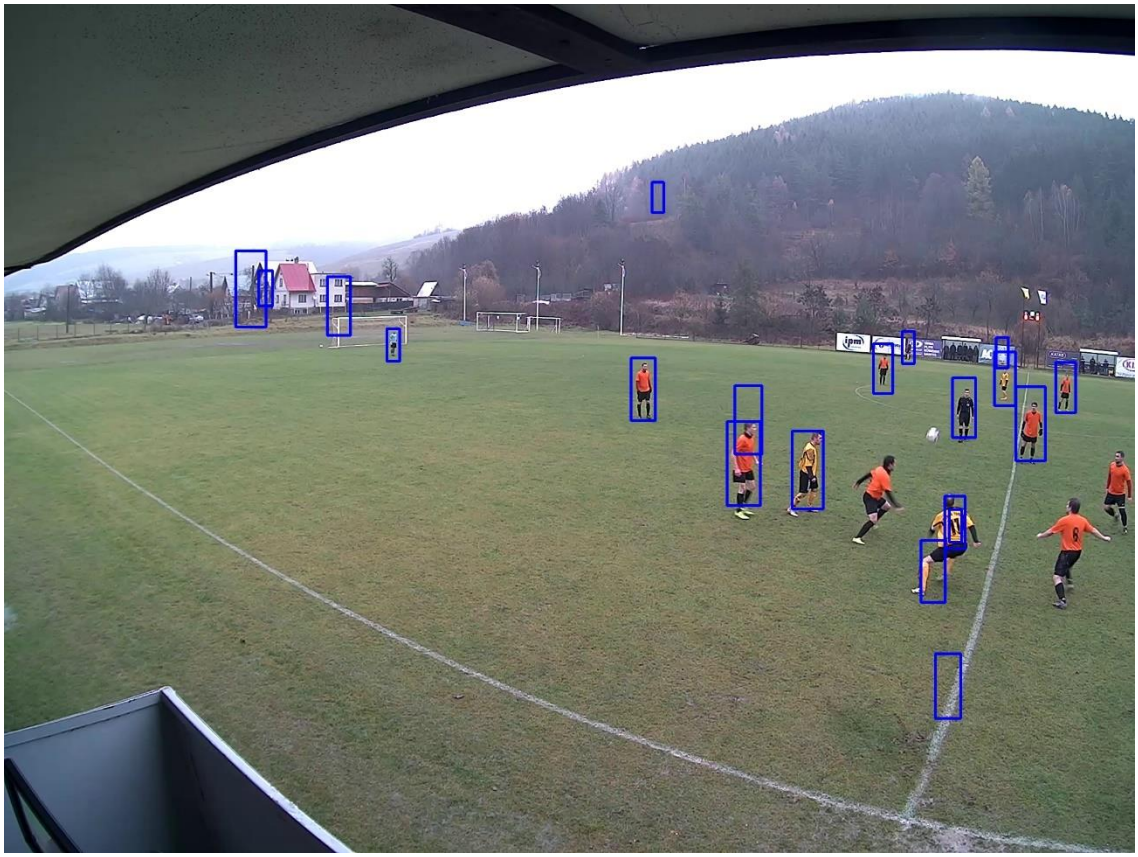
Deskriptor\Hodnota		Počet nájdených objektov	Počet nenájdených objektov	Počet chybných nálezov	Počet všetkých nájdených objektov
Haarové vlnky		1698	1382	1154	2913
SHOG		2027	1053	1598	3663

Tabuľka 1 – Výsledky detekcie

Prvá bola vyhodnotená kaskáda Haarových vlniek. Táto kaskáda detekovala v obraze viac ako 55% objektov, s pomerom chybných nálezov (false-positive rate) okolo 10^{-5} čo sú relatívne dobré výsledky. Pravdepodobne v prípade úpravy zgrupovania obdĺžnikov by bolo možné zlepšiť tieto výsledky prispôsobením na túto úlohu.

Obrázok 16 – Príklad detekcie kaskády Haarových vlniek

Rovnaký test s rovnakými parametrami bol vyhodnotený pre kaskádu SHOG deskriptora. V jeho prípade jeho úspešnosť bola 65% a porovnateľný počet chybných nálezov ako pri Haarových vlnkách. V prípade SHOG deskriptora, má natrénovaná kaskáda väčší problém s čiastočne otočenými hráčmi, v momentoch kedy nie sú v zpriamenej polohe. Tento poznatok je v súlade s vyhodnotením Dalal-Triggsa o HOG deskriptore [2]. Počet chybných nálezov je možné znížiť detailnejším tréningom.



Obrázok 17 – Príklad detekcie kaskády SHOG deskriptora

Celkovo môžeme vyhodnotiť, že SHOG deskriptor podáva konkurencieschopné výsledky pri porovnaní s Haarovými vlnkami. Je pomalší, ale taktiež je možnosť lepšej paralelizácie a pripravenia jeho hodnôt pre klasifikačný krok. Je to deskriptor, ktorý možno využiť pri rôznych úlohách detekcie obrazu, a taktiež je vhodný na experimentovanie s kaskádou. Podľa mňa by bolo vhodné v budúcnosti preskúmať jeho vlastnosti v kaskáde, ktorá bude používať rôzne typy deskriptorov naraz.

2.9 Možnosti rozšírenia

Do budúcnosti existujú určité možnosti ako pokračovať v testovaní, prípadne hľadať využiteľnejšie možnosti. Keďže náš vlastný deskriptor je implementovaný na CPU, prináša to určité limity na rýchlosť a na distribúciu výpočtov. V oblasti počítačového videnia sa bežne využívajú klastre GPU procesorov, ktoré podávajú väčší výkon. S dizajnom GPU algoritmov taktiež prichádza inherentná distribúcia výpočtov na iné zariadenia, keďže častokrát keď je algoritmus paralelizovateľný, môže taktiež bežať na rôznych zariadeniach naraz, napríklad ak máme viacero GPU procesorov v jednom počítači.

Viola-Jones kaskáda v OpenCV disponuje CUDA implementáciou detectMultiScale v spojení s Haarovými vlnkami. Napriek tomu, že táto implementácia je nízkoúrovňová, a neexistuje jednoduché rozhranie do ktorého by bolo možné zapojiť vlastný výpočet deskriptora, má táto implementácia niekoľko častí, ktoré sa dajú znova použiť.

Medzi tieto súčasti patria:

- Réžia výpočtu rôznych škál
- Optimalizovaný výpočet integrálneho obrazu na GPU pomocou operácie parallel scan prefix sum
- Vyhodnocovanie slabých hypotéz a rozhodovanie o danom detekčnom okne v kaskáde
- Export nájdených obdĺžnikov z GPU na CPU

Spomínané súčasti by sa dali znova využiť pri implementácií SHOG deskriptora na GPU. Vtedy by bolo potrebné presne pochopiť mapovanie do pamäte, keďže existujúca implementácia seká pamäť tak, aby čítanie z nej bolo čo najrýchlejšie. Taktiež by bolo potrebné implementovať načítanie nových modelov do pamäte, ale na túto úlohu by sa dala využiť časť implementácie LBP deskriptora na GPU. V mojej vlastnej práci [10] som skúmal rýchlosť implementácií HOG deskriptora na GPU, a keďže najväčším problémom bola normalizácia, ktorú SHOG nepoužíva, GPU implementácia by priniesla podstatné zrýchlenie. Taktiež by bolo možné doimplementovať do implementácie detectMultiScale Haarových vlniek dve funkcie –

načítavanie nových OpenCV 2.0 modelov kaskády a zgrupovanie hypotéz. Na zgrupovanie hypotéz sa momentálne využíva CPU funkcia.

Implementované rozhrania VJ kaskády taktiež umožňujú rovnaké výpočty vykonávať na platforme OpenCL. Celý algoritmus je potrebné prispôsobiť na prácu s UMat objektami, ktoré v princípe predstavujú surovú pamäť GPU. Existujúce implementácie Haarových vlniek a LBP deskriptora majú tieto potrebné úpravy, a preto je možné sa nimi inšpirovať v prípade potreby SHOG deskriptora na OpenCL platforme.

Okrem iného je možné zlepšiť výsledky z minulej kapitoly rozšírením samotného boostingového algoritmu kaskády. Medzi už spomínané zlepšenia patrí napríklad implementácia SRB learnera, použitého v práci SHOG deskriptora [6]. Medzi ďalšie možnosti patria napríklad:

- implementácia tzv. polovičnej črty – v prípade, že máme viacero čŕt, ktoré podávajú rovnaké zlepšenie detekcie, vyberieme tú, ktorá je najrýchlejšia
- implementácia Fisherovej diskriminačnej analýzy – umožní rozšíriť SHOG na FDA-HOG, ktorý potenciálne môže byť vhodný na iné úlohy

3. ZÁVER

Cieľom tejto práce bolo zhodnotiť momentálnu situáciu a zhodnotiť možnosti implementácie deskriptora, ktorý je možné používať na detekciu objektov pomocou AdaBoostu. Hlavné parametre pri hľadaní vhodného deskriptora boli paralelizovateľnost a schopnosť generovať veľký počet slabých klasifikátor.

So snahou pochopiť dôležitosť algoritmov, ktoré je možné upraviť na rôzne výpočtové modely, sme sa venovali procesorom. Prešli sme rôzne dôvody, kvôli ktorým moderné procesory obmedzujú dizajn rýchlych algoritmov. Oboznámili sme sa s deskriptormi a klasifikátormi. Medzi deskriptormi sme do hĺbky rozoberali deskriptory v minulosti používané s boostingovými kaskádami a to najmä Haarové vlnky, HOG deskriptor a jeho úpravy. Taktiež sme sa oboznámili s AdaBoostom a Viola-Jones kaskádou. Nakoniec sme popísali knižnicu OpenCV.

V praktickej časti sme analyzovali dostupné implementácie AdaBoostu a myšlienkový proces pri ich výbere. Popísali sme postupy pri oboznamovaní sa s algoritmami, ktoré sú potrebné pri detekcií objektov a z toho sme dospeli k zmene používanej AdaBoost implementácie. Potom sme sa do hĺbky venovali OpenCV implementácií Viola-Jones kaskády, za účelom využitia všetkých jej poznatkov a jej rozšíriteľnosti. Taktiež bolo spomenutých niekoľko slabých bodov tejto implementácie, a niekoľko možností vylepšenia. Následne sme sa venovali rozboru deskriptorov a porovnávali sme ich s našimi kritériami. Detailne bola popísaná implementácia SHOG deskriptora. V kapitole testovania sme zhodnotili, že SHOG podáva porovnateľné výsledky s už implementovanými Haarovými vlnkami. Nasledovalo zhodnotenie možných vylepšení deskriptora a kaskády a akým smerom sa môže posúvať ďalšie zlepšovanie detekcie.

ZOZNAM POUŽITÝCH SKRATIEK

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

FDA – Fisherova Diskriminačná Analýza

GPU – Graphics Processing Unit

HD – High-Definition

HOG – Histogram of Oriented Gradients

LBP – Local Binary Pattern

RAM – Random Access Memory

SRB – Smoothed Response Binning

SVM – Support Vector Machine

TBB – Threading Building Blocks

ZOZNAM POUŽITÝCH OBRÁZKOV

ZOZNAM POUŽITÝCH TABULIEK

ZOZNAM POUŽITEJ LITERATÚRY

[1] J. S. A. J. SIROT, „RELATING VISUAL AND SEMANTIC IMAGE DESCRIPTORS,“ Centre for Digital Video Processing, Dublin City University, Dublin 9, 2004.

[2] N. D. a. B. Triggs, „Histograms of Oriented Gradients for Human Detection,“ INRIA, Rhˆone-Alps, 655 avenue de l’Europe, Montbonnot 38334, 2005.

[3] M. J. Paul Viola, „Robust Real-time Object Detection,“ CVPR, Cambridge, 2001.

[4] S. A. M.-C. Y. K.-T. C. Qiang Zhu, „Fast Human Detection Using a Cascade of Histograms of Oriented Gradients,“ MITSUBISHI ELECTRIC RESEARCH LABORATORIES, 201 Broadway, Cambridge, Massachusetts 02139, 2006.

[5] L. P. A. a. N. P. Gary Overett, „Boosting a Heterogeneous Pool of Fast HOG Features for Pedestrian and Sign Detection,“ NICTA, Locked Bag 8001, Canberra, 2009.

[6] G. O. a. L. Petersson, „Large Scale Sign Detection using HOG Feature Variants,“ NICTA, Locked Bag 8001, Canberra, Australia, 2011.

[7] L. P. a. L. A. Niklas Pettersson, „The Histogram Feature – A Resource-Efficient Weak Classifier,“ NICTA, Locked Bag 8001, Canberra, 2008.

[8] OpenCV.org, „OpenCV,“ OpenCV.org, [Online]. Available: opencv.org. [Cit. 4 5 2015].

[9] OpenCV.org, „OpenCV Documentation,“ OpenCV.org, [Online]. Available: http://docs.opencv.org. [Cit. 4 5 2015].

[10] F. Kajánek, „Výpočtovo efektívna implementácia HOG deskripto-ra s využitím mnohojadrových grafických proceso-rov,“ UNIZA, Žilina, 2015.

PRÍLOHY

Príloha A – CD médium obsahujúce

- všetky zdrojové kódy, ktoré boli upravované.
- návod na kompiláciu zdrojových kódov
- pripravený priečinok na spustenie tréningu kaskády
- bakalársku prácu