|  |  |
| --- | --- |
| Žilinská univerzita v Žilinežilinská univerzita v Žiline  Fakulta riadenia a informatikyfakulta riadenia a informatiky | |
| diplomová práca  Študijný program:  **Informačné systémy – Spracovanie dát**  Podnázov práce | |
| **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, 2017Bakalárska práca | |
|  |  |

|  |
| --- |
| Žilinská univerzita v Žilinežilinská univerzita v žiline  Fakulta riadenia a informatikyfakulta riadenia a informatiky |
| diplomová práca  Študijný program:  **Informačné systémy – Spracovanie dát**  Podnázov práce |
| **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  Bakalárska práca |

Original zadania – scan

Čestné vyhlásenie

Poďakovanie

Abstrakt

Contents

[Contents 7](#_Toc480362916)

[Úvod 9](#_Toc480362917)

[1. Teoretická časť 10](#_Toc480362918)

[1.1 Počítačové videnie 10](#_Toc480362919)

[1.2 Rýchlosť výpočtov 10](#_Toc480362920)

[1.3 Deskriptory/Klasifikátory 11](#_Toc480362921)

[1.4 AdaBoost + slabé klasifikátory 13](#_Toc480362922)

[1.5 Adaboost a spojenie s deskriptormi 15](#_Toc480362923)

[1.5.1 Haarové vlnky 15](#_Toc480362924)

[1.5.2 Histogram of oriented gradients 17](#_Toc480362925)

[1.5.3 Aproximácie HOG deskriptora - HistFeat 19](#_Toc480362926)

[1.5.4 LiteHOG a FDA-HOG 20](#_Toc480362927)

[1.6 Viola-Jones kaskáda 21](#_Toc480362928)

[1.7 OpenCV 22](#_Toc480362929)

[2. Praktická časť 23](#_Toc480362930)

[2.1 Testovanie 23](#_Toc480362931)

[2.1.1 Dataset 23](#_Toc480362932)

[2.2 Porovnávanie implementácií AdaBoostu 24](#_Toc480362933)

[2.2.1 OpenCV 3.0 implementácia 25](#_Toc480362934)

[2.2.2 Viola-Jones implementácia 25](#_Toc480362935)

[2.2.3 MultiBoost 26](#_Toc480362936)

[2.3 Popis našich vlastných pokusov 26](#_Toc480362937)

[2.4 Analýza Viola-Jones kaskády 26](#_Toc480362938)

[2.5 Implementácia rozšírenia na CPU 26](#_Toc480362939)

[2.6 Možnosti rozšírenia na GPU 26](#_Toc480362940)

[3. Záver 27](#_Toc480362941)

Úvod

Teoretická časť

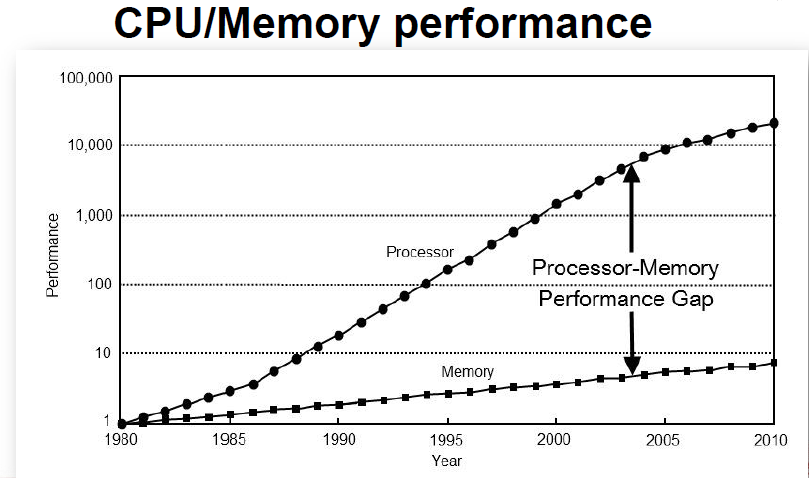
Počítačové videnie

Počítačové videnie je obor, ktorý sa zaoberá získavaním informácií z digitálneho obrazu a videa. Snaha je zanalyzovať, navrhnúť a naimplementovať činnosti, ktoré dokáže robiť ľudský zrakový systém. Pochopenie obrazu pre tento obor znamená transformáciu digitálneho obrazu na popisy sveta s ktorými dokážu pracovať iné myšlienkové procesy. Za účelom využitia vizuálnych dat možno použiť rôzne techniky z geometrie, fyziky, štatistiky a teorie učenia. Pod dátami, ktoré počítačové videnie používa, môžu byť ľubovoľné videá, rôzne pohľady z rôznych kamier, multi-dimenzionálne dáta zo zdravotných skenerov alebo aj len jednoduchá fotografia. Medzi podobory počítačového videnia patria napríklad rekonštrukcia scény, detekcia udalostí, sledovanie videa, detekcia objektov, učenie, indexovanie alebo aj predpovedanie pohybu.

Rýchlosť výpočtov

V počítačovom videní bolo vždy potrebné rýchle spracovanie dát. Vo väčšine prípadov uvažujeme, že najlepšie riešenie je nájsť taký popis alebo také vyhodnotenie, ktoré nám dodá čo najlepší výsledok. V tejto a taktiež v mnohých iných oblastiach je potrebné optimalizovať nielen dosiahnutý výsledok, ale aj rýchlosť algoritmu.

V dnešnej dobe je jasným trendom pri zvyšovaní výkonu, zvyšovať počet jadier v procesore, či už ide o CPU alebo GPU. Kvôli tomu sú potrebné algoritmy, ktoré dokážu bežať paralelne a bez prílišnej synchronizácie medzi procesmi. Dostupný hardware umožňuje nielen zdieľané výpočty na jednom fyzickom procesori ale aj taktiež medzi rôznymi počítačmi/grafickými kartami.

Ďalším a narastajúcim faktorom pri rýchlosti algoritmu je využívanie pamäte. Zatiaľčo FLOPs (floating point operations per second) procesorov narastajú podľa Moorovho zákona, rýchlosť pamäte narastá omnoho pomalšie, čo spôsobuje, že veľa algoritmov má úzky profil v oblasti využitia pamäte. Tento problém je veľmi výrazný hlavne na GPU architektúrach, kde individuálne vlákna zdieľajú pamäťové zbernice a taktiež cache pamäť. Na CPU je tento problém čiastočne eliminovaný komplikovanou logikou predikcie a L1/L2/L3 cache pamäťami. Jedným z bežných problémov spôsobujúcich uzky profil pri práci s pamäťou sú časté izolované prístupy do pamäte.

Obrázok -Rozdiel medzi nárastom výkonu CPU a pamäte

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

Vo výsledku tieto faktory spôsobujú, že niektoré algoritmy sú ťažko využiteľné v praxi. Často je možné vidieť algoritmy, ktoré sú len malou zmenou oproti pôvodnému konceptu a napriek tomu tento obmenený algoritmus už je možné využiť pri výpočtoch v reálnom čase. Ako príklad je úprava HOG deskriptoru na LiteHOG verziu (TODO pridať referenciu) alebo snaha aproximovať niektoré medzistupne výpočtu HOG deskriptoru pomocou integrálneho obrazu (TODO pridať referenciu na BC)

Deskriptory/Klasifikátory

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

Obrazový deskriptor je popis nejakých vizuálnych vlastností obrazu alebo videa, alebo taktiež algoritmus alebo aplikácia, ktorá tento popis vytvorí. V dnešnej dobe komunikačných technológií a internetu je potrebné rýchlo a spoľahlivo spracovávať a analyzovať dáta nám dostupné. Presne preto sú potrebné deskriptory, aby sme mali dostupné data na riešenie rôznych úloh.

Rozdeľujú sa na dve skupiny:

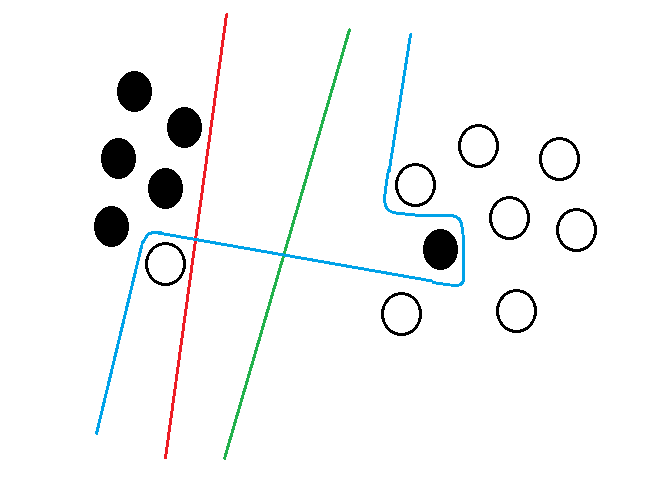
1. Všeobecné informačné deskriptory – hlavné typy popisujú podľa farby, tvaru, regionov, textury a pohybu
2. Informačné deskriptory špecifickej domény – riešia už nejakú špecifickú úlohu napríklad detekcia chodcov alebo sledovanie pohybu áut na parkovisku

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 popis 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.

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 výpočetnú priepustnosť dát.

Klasifikátory nachádzajú využitie v oblasti strojového učenia, kde pomáhajú pri rozhodovaní v jednoduchých ale aj komplexných systémoch. 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 výstup máme matematický model, ktorý dokáže rozhodovať o daných dátach.

Obrázok - Klasifikátor

Na obrázku 2 máme 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 je priliš tesne urobený na trénovacie dáta a neberie ohľad na štatistické chyby, ktoré predstavuje osamotená čierna trieda medzi bielymi (a naopak).

AdaBoost + slabé klasifikátory

AdaBoost je meta-algoritmus pre strojové účenie. 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, a on nepracuje so samotnými dátami, ale pracuje s týmito klasifikátormi. Väčšinou sa používajú tzv. slabé klasifikátory, ktorých šanca urobiť chybu je menšia ako 50%, že rozhodnutie, ktoré urobia je správne. 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 TODO)

Ako slabé klasifikátory sa zvyčajne používajú rozhodovacie stromy rôznych spôsobov. V práci (TODO pridať prácu ktorá vylepšovala hog od dalal triggsa) používali ako slabé klasifikátory SVM klasifikátory, za účelom zjednodušenia viacdimenzionálnej úlohy na lineárnu.

Samotný AdaBoost, tak ako väčšina machine-learning algoritmov, funguje v dvoch módoch, tréningový a testovací. V prípade AdaBoostu, v trénovacom móde sa natrénuje model vážením 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. Zatiaľčo pri trénovacom móde nám záleží len na kvalite natrénovaného modelu, pri testovacom móde nám taktiež záleží na rýchlosti rozhodovania. V prípade že využívame AdaBoostu na detekciu objektov v obraze, rozhodnutie sa volá milióny krát 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:

1. Diskrétny AdaBoost – vstupná hodnota sa prenásobí hodnotou v liste rozhodovacieho stromu, a porovná sa voči nejakej hranici buď jednotlivo alebo ako suma pre všetky vstupné hodnoty
2. 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
3. GentleBoost – limituje krok upravovania cien pri tréningu, tak aby algoritmus nenastavil ako krok nekonečno, čo spôsobuje nárast chyby
4. LogitBoost – aplikácia logistickej regresie na AdaBoost
5. 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.

TODO: prečo sa používajú rozhodovacie stromy

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

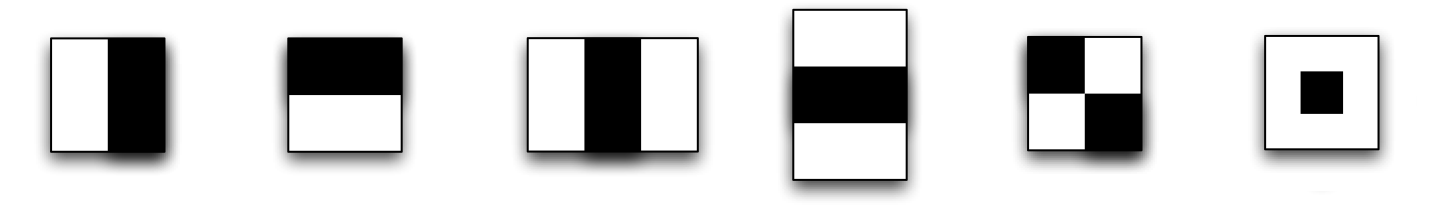
Adaboost a spojenie s deskriptormi

Najväčší problém využitia AdaBoostu v praxi je potreba dobrého deskriptoru, ktorý správne vyjadrí nami hľadané objekty v obraze. Za prepokladu, 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 vypracované pokusy s deskriptormi za použitia AdaBoostu, a to v spojení s Haarovými vlnkami (Viola Jones práca referencia), neskôr taktiež v spojení s HOG deskriptorom, a SVM ako slabým klasifikátorom (pokračovanie hoga referencia aziati), v spojení s Local Binary Patterns a taktiež v spojení s aproximáciami HOG deskriptora (litehog a fda hog referencie).

Keďže napriek využitiu ideálnych deskriptorov na danú, sa dokáže jeden klasifikátor ľahko pretrénovať, používajú sa tzv. kaskády klasifikátorov. Je to niekoľko klasifikátorov za sebou, ktoré filtrujú všetky testovacie vstupy, tak aby na konci sme mali len správne výsledky. Tento prístup má niekoľko výhod. Umožňuje používať rôzne typy klasifikátorov, s rôznymi rýchlosťami a rôznymi filtračnými schopnosťami. Na začiatku kaskády sa zvyknú používať čo najrýchlejšie klasifikátory, a na konci kaskády čo najpresnejšie klasifikátory.

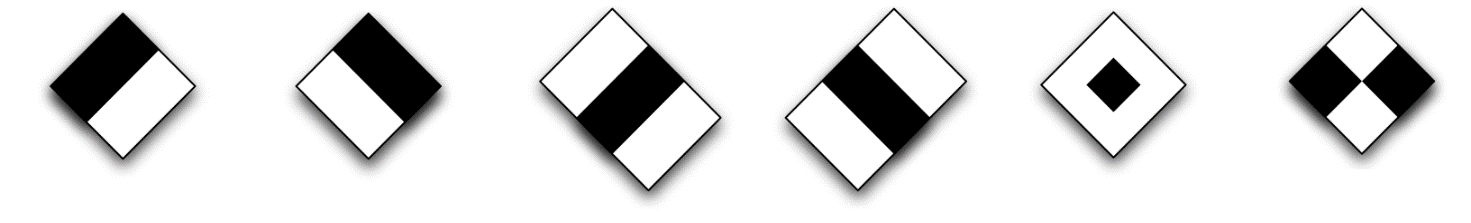
Haarové vlnky

Jednoduchá Haarová vlnka (v angl. Haar-like feature) v oblasti detekcie objektov sa dá definovať ako rozdiel súm pixelov 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 (obrazok 1). 

Prvý krát boli použité a pomenované Violom a Jonesom pri snahe vytvoriť detektor tvárií. Ich meno pochádza z Haarových vlniek (v angl. Haar Wavelet) z matematiky, ktoré sú postupnosťami funkcií vytvárajúce 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 dostatočnú informáciu na to aby sme mohli rozhodnúť či tam tvár je alebo nie je(obrazok 3).

Obrázok - 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 dvojobdĺžnikovú, 8 pre trojobdĺžnikovú a 9 pre štvorobdĺžnikovú.

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 výpočetnou rýchlosťou.

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 rýchlosť procesorov je 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 znížiť čo najviacej počet prístupov do pamäte. Preto v praxi 9 prístupov do pamäte je viac na jednu črtu ako by sme chceli, a kedže vlnky sú zväčša náhodne roztrúsené po detekčnom okne, nie je možné ich naskladať za sebou. (TODO: pridať vysvetlenie prečo je 9 veľa)

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

Prvý krát bol tento termín použitý v práci Dalala a Triggsa (referencia), 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.

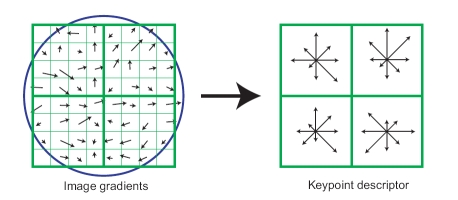
Samotný výpočet HOG deskriptoru má niekoľko krokov, a v práci Dalala a Triggsa (referencia) boli detailne rozobraté najlepšie verzie podkrokov. Výsledný algoritmus sa skladal z nasledujúcich častí:

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 magnitúdu

a na orientáciu gradientu

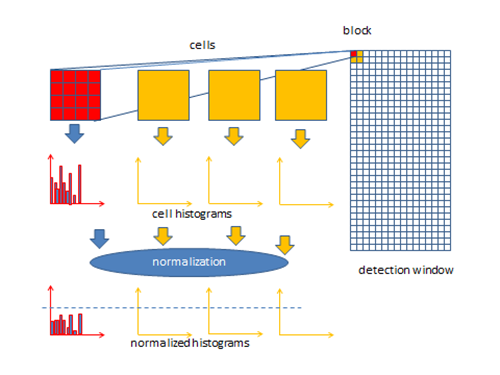
kde je gradient v smere x(horizontálnom), je gradient v smere y(vertikálnom).

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 odporúčas 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 - 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 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. (obrazok algoritmu)

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 je jedine CPU algoritmus reálna voľba, a aj pri veľkých obrázkoch nárast výkonu nie je dostatočne veľký na to aby bol dôvod vyvíjať omnoho náročnejší GPU algoritmus.

Obrázok - Ilustrácia fungovania normalizácie

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

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.

Prvým takýmto deskriptorom je HistFeat(referencia na pracu). 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. Prvý krok pre výpočet gradientu limitujú počet orientácií na 8 košov, čo umožňuje ukladať hodnotu v 3ch bitoch. Magnitúdu počítajú sčítaním absolútnych hodnôt, namiesto presnejšen Euklidovskej vzdianosti 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 osekáme hodnotu 16 na 15 čo umožní uložiť výsledok do jednej 4-bit premennej. Koncept normalizácie blokov sa v tomto deskriptore vôbec nepoužíva.

V klasifikačnej fáze využívali „a posteriori“ tabuľku, ktorú adresovali 2ma 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.

LiteHOG a FDA-HOG

Na HistFeat nadväzuje ďalšia práca od rovnakých autorov v ktorej sa snažia vytvoriť viac komplikovaný 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.

Pri LiteHOGu hlavná zmena oproti HistFeat je využitie Fisherovej diskriminačnej analýzy (dalej FDA) na transformáciu 8-rozmerného priestoru, súradnice predstavujú histogramové hodnotu, na lineárny priestor. 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+.

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. Obe TODO dopisať

Viola-Jones kaskáda

V roku 2001 bol vyvinutý framework na detekciu objektov dvoma vedcami Paulom Viola a Michaelom Jonesom. Bol to prvý 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. (referencia na VJ prácu).

Táto prvá verzia frameworku využívala Haarové vlnky ako deskriptor. Keďže je ich možné generovať veľmi veľký počet, dajú sa z nich dobre robiť rôzne kroky kaskády. Na ich výpočet bol využívaný integrálny obraz. 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.

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 [7], ktorá udržuje stránku pre developerov a používateľov, vrátane online dokumentácie [8].

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.

Praktická časť

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

Dataset

Detekcia a rýchlosti boli merané na datasete z videa futbalového zápasu. Rozlíšenie videa 2048x1536, 30 fps z ktorých bolo 2 fps oanotované v rámci projektu Analýza obrazu v zimnom a letnom semestri 2015/2016. Pre potrebu tejto práce boli vysekané pozitívne a negatívne dáta pomocou týchto anotácií. Príklady datasetu:

Anotácie sú uložené v XML súbore, príklad anotovania:

C:\Users\killerwife\AppData\Local\Microsoft\Windows\INetCache\Content.Word\20170416225623629.pngC:\Users\killerwife\AppData\Local\Microsoft\Windows\INetCache\Content.Word\20170416225343332.pngC:\Users\killerwife\AppData\Local\Microsoft\Windows\INetCache\Content.Word\20170416225622523.pngPríklad výseku ktorý je použitý pri trénovaní a testovaní:

Porovnávanie implementácií AdaBoostu

V rámci výberu implementácie AdaBoostu boli porovnávané 4 implementácie:

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

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

OpenCV 3.0 implementácia

Táto implementácia je úplne všeobecná implementácia cielená na experimentovanie nielen v rámci počítačového videnia. Disponuje 4ma variantmi trénovacej časti a to DiscreteBoost, RealBoost, LogitBoost a GentleBoost. Využíva rozhodovacie stromy ako slabý 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.

Viola-Jones implementácia

Táto implementácia je súčasťou väčšieho frameworku a okrem implementácie AdaBoostu, je taktiež implementáciou Viola-Jones kaskády (referencia). 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.

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.

Popis našich vlastných pokusov

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 holé 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 normálne vlnky aj 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. Táto prvá verzia nezbehla do 30tich 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 hĺbkovej 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 ceny pomalšieho behu.

Analýza Viola-Jones kaskády

Implementácia rozšírenia na CPU

Možnosti rozšírenia na GPU

Záver