

UNIVERZITA KOMENSKÉHO, BRATISLAVA
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

OVLÁDANIE POČÍTAČOVÝCH APLIKÁCIÍ
POMOCOU GEST RUKY

Bakalárska práca

2012

Michal Hozza

UNIVERZITA KOMENSKÉHO, BRATISLAVA
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

OVLÁDANIE POČÍTAČOVÝCH APLIKÁCIÍ
POMOCOU GEST RUKY

Bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Školiace pracovisko: Katedra Informatiky
Školiteľ: RNDr. Marek Nagy, PhD.

Bratislava, 2012
Michal Hozza



Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Michal Hozza
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Ovládanie počítačových aplikácií pomocou gest ruky

Cieľ: Pomocou počítačových neurónových sietí rozpoznávať jednoduché dynamické gestá prezentované rukou a zaznamenané webovou kamerou. Porovnať dva prístupy k rozpoznávaniu: cez doprednú a rekurentnú neurónovú sieť. Realizovaný rozpoznávač prepojiť na ovládanie aplikácií.

Poznámka: C++, Qt, Linux

Vedúci: RNDr. Marek Nagy, PhD.

Dátum zadania: 13.10.2011

Dátum schválenia: 20.10.2011

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

Ďakujem vedúcemu bakalárskej práce Marekovi Nagyovi za cenné rady a pripomienky, bez ktorých by táto práca asi nevznikla, priateľke, blízkym priateľom a rodine za morálnu podporu.

Michal Hozza

Abstrakt

Táto bakalárska práca sa zaoberá návrhom a implementáciou aplikácie na ovládanie počítača pohybovými gestami ruky. Popisujeme návrh základného algoritmu a návrh vhodnej architektúry umelej neurónovej siete na rozpoznanie ruky. Ukážeme si základné metódy predspracovania obrazu, ktoré nám pomôžu zvýšiť úspešnosť rozpoznávania. Ďalej popisujeme jednotlivé triedy a ich metódy a implementáciu aplikácie.

Kľúčové slová: neurónové siete, umelá inteligencia, počítačové videnie, rozpoznávanie gest, rozpoznávanie ruky

Abstract

This bachelor thesis deals with a design and implementation of application for controlling computer by moving hand gestures. We describe design of basic algorithm and design of suitable architecture of artificial neural network for hand recognition. We will show basic methods of image preprocessing, that will help us increase success of recognition.

Next, we describe classes and their methods and implementation of the application.

Key words: neural networks, artificial intelligence, computer vision, gesture recognition, hand recognition

Obsah

Úvod	1
1 Platforma, jazyk a knižnice	2
2 Neurónové siete	3
2.1 Jednoduchý spojitý perceptrón	3
2.2 Jednovrstvová neurónová sieť	4
2.3 Viacvrstvová dopredná neurónová sieť	4
3 Návrh	6
3.1 Základný algoritmus	6
3.1.1 Rozdielový obraz	6
3.1.2 Predspracovanie vstupného obrazu	7
3.1.3 Segmentácia	8
3.1.4 Predspracovanie segmentov	8
3.2 Návrh architektúr neurónových sietí	9
3.2.1 Požiadavky na architektúru neurónovej siete	10
3.2.2 Typ 1: Viacvrstvová dopredná neurónová sieť	10
3.2.3 Typ 2: Upravená verzia viacvrstvovej doprednej neurónovej siete	10
3.2.4 Typ 3: Rekurentná neurónová sieť	11
4 Experimenty	13
4.1 Trénovanie sietí	13
4.2 Testovanie a vyhodnocovanie neurónových sietí	13
4.3 Trénovacie a testovacie dáta	14
4.4 Porovnanie architektúr neurónových sietí	14
4.4.1 Typ 1: Viacvrstvová dopredná neurónová sieť	14
4.4.2 Typ 2: Upravená verzia viacvrstvovej doprednej neurónovej siete	15
4.4.3 Typ 3: Rekurentná neurónová sieť	15
4.4.4 Zhrnutie	17

4.5	Porovnanie úspešnosti pri rôznom spôsobe predspracovania	17
4.5.1	Vplyv Fourierovej transformácie	17
4.5.2	Rozdielový vs. pôvodný obraz	18
4.5.3	Zhrnutie	18
5	Implementácia	19
5.1	Neurónové siete	19
5.1.1	Perceptron	19
5.1.2	ContinuousPerceptron	20
5.1.3	RecurrentPerceptron	22
5.1.4	NeuralLayer	22
5.1.5	DistributedNeuralLayer	22
5.1.6	RecurrentLayer	23
5.1.7	DistributedRecurrentLayer	23
5.1.8	NeuralNetwork	23
5.1.9	DistributedNeuralNetwork	24
5.1.10	RecurrentNetwork	24
5.1.11	DistributedRecurrentNetwork	24
5.2	Triedy pre obrázky	24
5.3	Získanie obrazu z webkamery	27
5.4	Spracovanie obrazu - Trieda ImageProcessor	27
5.4.1	Krok 1: Príprava	27
5.4.2	Krok 2: Algoritmus rozpitia	27
5.4.3	Krok 3: Segmentácia	28
5.5	Rozpoznanie ruky	28
5.6	Rozpoznanie gesta	31
5.6.1	GestureRecognizer	31
5.6.2	Gesture	33
5.7	Simulácia stlačenia klávesu	33
5.8	Pomocné programy	33
5.9	Problémy a ich riešenia	34
5.9.1	Multivláknovosť	34
5.9.2	Viacnásobné dedenie	35
5.10	Screenshoty aplikácie	35
	Záver	36
	Literatúra	37

Zoznam obrázkov

2.1	Prírodný vs. umelý neurón	3
2.2	Dopredná neurónová sieť	5
3.1	Základný algoritmus	7
3.2	Fourierova transformácia a jej typické vlastnosti pre obrázok ruky . . .	9
3.3	Upravená dopredná neurónová sieť	11
3.4	Rekurentný neurón	11
5.1	Neurónové siete - Class diagram	21
5.2	Zdrojový kód funkcie <code>HImage<T>::getAdaptiveFloodFillSelectionMask()</code>	26
5.3	Zdrojový kód funkcie <code>ImageProcessor::expandPixelsX()</code>	29
5.4	Zdrojový kód funkcie <code>ImageProcessor::segment()</code>	30
5.5	Zdrojový kód funkcie <code>GestureRecognizer::getGesture()</code>	32
5.6	Zdrojový kód funkcie <code>GestureRecognizer::removeNoise()</code>	32
5.7	Správne rozpoznaná ruka a jej predchádzajúca trajektória	35
5.8	Správne rozpoznávaný iný objekt	35

Zoznam tabuliek

4.1	Velkosti sád vstupov	14
4.2	Porovnanie úspešnosti doprednej NS pri rôznych počtoch neurónov . . .	15
4.3	Porovnanie úspešnosti upravenej NS pri rôznych počtoch neurónov . . .	16
4.4	Porovnanie úspešnosti rekurentnej NS pri rôznom umiestnení rekurencie	16
4.5	Porovnanie úspešnosti rekurentnej NS pri rôznych počtoch neurónov . .	17
4.6	Porovnanie úspešnosti NS na dátach Fourierovej tr. a rozdielového obrázku	18
4.7	Porovnanie úspešnosti NS na dátach rozdielového obrázku, pôvodného obrázku a ich Fourierovych transformácií	18

Úvod

Notebooky sú často využívané na prezentácie, či už prednášok spracovaných v rôznych aplikáciách, alebo jednoduchého zobrazenia fotiek fotiek. Pri prezentáciách sa často využíva diaľkový ovládač, aby prezentujúci nemusel sedieť pri počítači alebo k nemu stále chodiť.

V dnešnej dobe väčšina notebookov obsahuje webovú kameru, takže sa ponúka otázka, či sa nedá kamera využiť na elimináciu potreby ovládača. Počítač by sa mohol ovládať pomocou pohybu ruky, ktorý by sa snímal webkamerou.

V čase, keď vznikol nápad písať túto prácu¹ sa takéto riešenia v praxi nevyužívali, hoci niečím podobným sa už zaoberali viacerí. Nenašli sme však žiadne dokončené riešenia, ktoré by boli voľne prístupné.

Ešte pred dokončením tejto práce bol uvedený na trh inteligentný televízor, ktorý bolo možné ovládať kombináciou hlasu a pohybu ruky, čo naznačuje, že táto myšlienka nie je zlá.

Narozdiel od spracovania v televízore, kde sa ruka používala na ovládanie kurzora myši, my budeme pohyby ruky interpretovať ako gestá a následne prekladať na príkazy pre počítač - pomocou simulácie stlačení klávesov.

Práca je zameraná na problém rozpoznania pohybujúcej sa ruky v obraze, ktorý je získaný z webkamery. Rozpoznávanie realizujeme pomocou neurónových sietí. Popíšeme návrh architektúry siete a rôzne spôsoby predspracovania dát, aby sme dosiahli čo najlepšiu úspešnosť rozpoznávania. Ukážeme, aké prístupy môžu pomôcť a čo sa stane ak do siete pridáme rekurentné spojenia.

Nakoniec popíšeme implementáciu aplikácie, ktorá bude rozpoznávať jednoduché gestá ruky a bude umožňovať ovládať iné aplikácie rozpoznávanými gestami.

Výsledný produkt sa bude dať použiť napríklad na ovládanie prezentačnej aplikácie alebo prehliadača fotiek, či prehrávača videa.

¹v roku 2010

Kapitola 1

Platforma, jazyk a knižnice

V tejto kapitole sa budeme venovať použitej platforme, jazyku a knižniciam. Vysvetlíme dôvody, prečo sme si ich vybrali.

Vhodnou platformou pre vývoj aplikácií, ktoré pracujú s perifériami je Linux, pretože má dostupné otvorené ovládače a vynikajúcu podporu. V Linuxe sa pracuje s perifériami veľmi pohodlne, každé zariadenie má vlastný súbor, ktorý sa nachádza v `/dev/`.

Linux je písaný hlavne v jazyku C, preto prevažná väčšina knižníc pre Linux je dodávaná aj s hlavičkovými súbormi pre jazyk C a C++. Jazyk C++ je preto veľmi vhodný pre programovanie na tejto platforme. Jazyk C++ má taktiež vynikajúcu podporu, kvalitný kompilátor a je objektovo orientovaný, čo umožňuje jednoduché použitie komplexných dátových štruktúr. Vďaka vysokej podpore C++ na rôznych platformách, nie je problém po prispôbení niektorých častí aplikácie, preniesť ju aj na iné platformy.

Programovanie a prácu nám uľahčia už hotové knižnice. Na užívateľské rozhranie použijeme framework *Qt*, ktorý je multiplatformový a umožňuje jednoduchú prácu s oknami a ďalším GUI. Okrem toho obsahuje aj triedy umožňujúce ľahkú správu vlákien a mnohé iné.

Ďalej využijeme linuxovú knižnicu *v4l2*¹, ktorá je určená na prácu s webovou kamerou a je štandardnou súčasťou väčšiny Linuxových distribúcií.

Na počítanie Fourierovej transformácie využijeme rýchlu knižnicu *fftw3* a na simuláciu stlačení klávesov knižnicu *Xtst*, ktorá umožňuje simuláciu udalostí v X-serveri.

Naša aplikácia využíva niekoľko vlákien, aby mohla paralelizovať niektoré výpočty. Správa vlákien sa vykonáva pomocou *Qt* frameworku. Paralelizuje sa hlavne predspracovanie obrázkov a segmentov(kapitola 5.4).

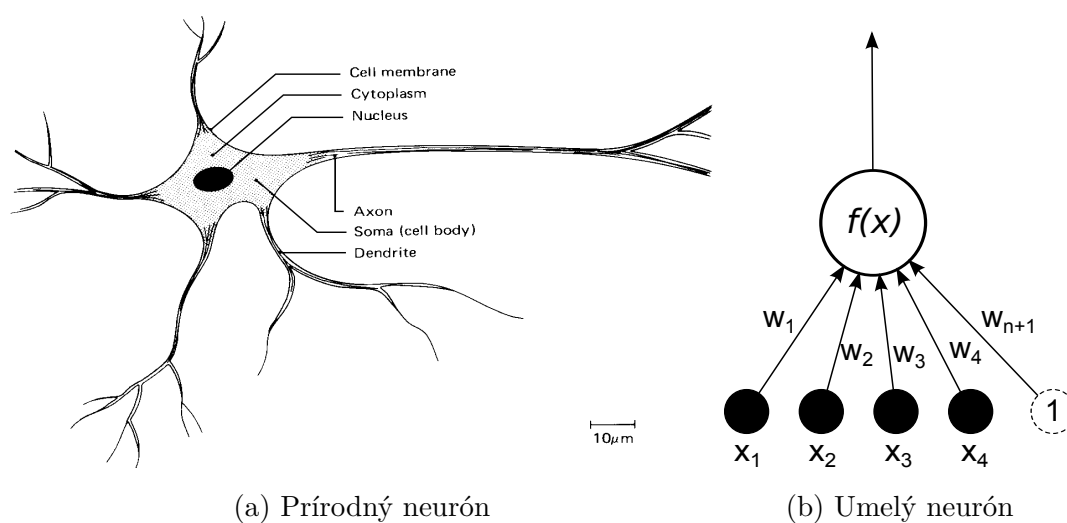
¹video for linux 2

Kapitola 2

Neurónové siete

Umelé neurónové siete sú významnou časťou tejto práce, preto im venujeme samostatnú kapitolu. Vysvetlíme, čo sú neurónové siete a ako fungujú. Popíšeme základné typy neurónových sietí, ktoré používame v aplikácii.

2.1 Jednoduchý spojitý perceptrón



Obr. 2.1: Prírodný vs. umelý neurón

Jednoduchý perceptrón je inšpirovaný nervovou bunkou - *neurónom*. Vstupy umelého neurónu zodpovedajú *dendritom*, výstup zodpovedá *axónu*. Transformáciu vstupu na výstup zabezpečuje aktivačná funkcia.

Keď si neurón predstavíme ako orientovaný graf (Obr. 2.1b), za vrcholy položíme vstupy, výstup a „telo“ neurónu, vzniknú nám 2 typy hrán.

1. Synaptické – vstup \rightarrow neurón – lineárna input-output väzba, kde pôvodný signál x_i prenasobíme váhou synapsy w_i , a tým dostaneme výsledný signál x'_i .
2. Aktivačné – neurón \rightarrow výstup – nelineárna input-output väzba, kde y dostaneme dosadením $\sum x'_i$ do aktivačnej funkcie.

K daným vstupom treba pridať ešte jeden vstup – *bias*, ktorý má vždy hodnotu 1. Má význam pri vstupe so samými nulami, v takomto prípade hodnoty na synapsách ostajú nezmenené a perceptrón by sa tento vzor nevedel naučiť.

Nech n je počet vstupov a f je aktivačná funkcia. Výsledný signál y dostaneme takto:

$$y = f\left(\sum_{i=1}^{n+1} w_i x_i\right) \quad x_{n+1} = 1$$

Aktivačnou funkciou spojitého perceptrónu je sigmoida:

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

Z matematického hľadiska robí takýto perceptrón zobrazenie $\mathbb{R}^n \rightarrow (0, 1)$.

2.2 Jednovrstvová neurónová sieť

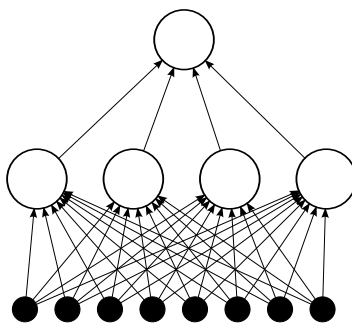
Môžeme spojiť niekoľko perceptrónov do jednej vrstvy. Získame tým väčší rozmer výstupu a teda môžeme vstupy klasifikovať do viacerých tried. Hodí sa to aj pri použití vo viacvrstvových sieťach, kde môžeme vrstvy pospájať.

Neuróny vo vrstve nie sú nijako pospájané, čo znamená, že každý operuje nezávisle, takže z praktického hľadiska ide o skupinu neurónov pracujúcich nad tým istým vstupom.

Vrstva robí zobrazenie $\mathbb{R}^n \rightarrow (0, 1)^m$, kde n je rozmer vstupu a m je počet neurónov vo vrstve.

2.3 Viacvrstvová dopredná neurónová sieť

Viacvrstvová neurónová sieť vznikne spojením niekoľkých vrstiev. Nižšia vrstva tvorí vstup pre vyššiu (obr. 2.2). Pôvodný vstup je vstupom pre najnižšiu vrstvu.



Obr. 2.2: Dopredná neurónová sieť

Viacvrstvová sieť umožňuje riešiť problémy, ktoré jedna vrstva riešiť nedokáže. Jedným z nich je napríklad funkcia *xor* [Hay99, s. 197]. Vo všeobecnosti sa viacvrstvová sieť dokáže naučiť aj súvislosti medzi vstupmi.

Viacvrstvové neurónové siete trénujeme pomocou algoritmu *backpropagation*¹ [Hay99, Fed11].

¹algoritmus spätného šírenia chyby

Kapitola 3

Návrh

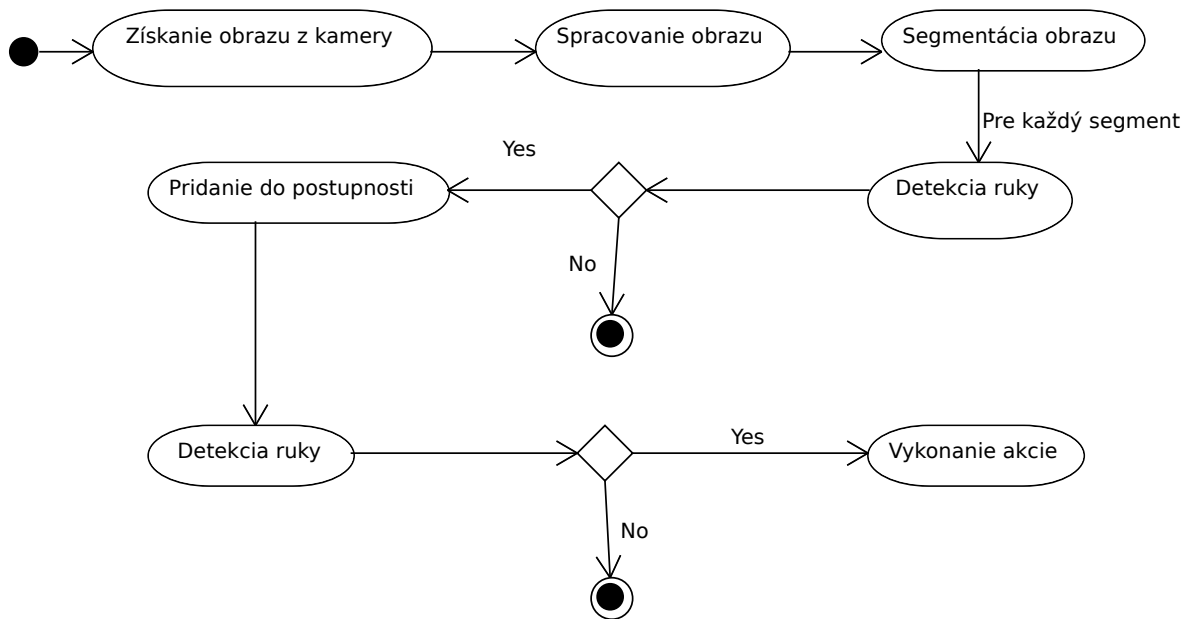
V tejto kapitole predstavíme základný algoritmus a postupne sa budeme venovať jeho jednotlivým častiam. Ukážeme metódy na zlepšenie úspešnosti klasifikátora a porovnáme rôzne prístupy.

3.1 Základný algoritmus

Základný algoritmus (Obr. 3.1) je nasledovný. Najskôr sa získa obraz z webkamery. Ten sa potom spracuje a následne sa rozdelí na jednotlivé pohybujúce sa segmenty. Každý z týchto segmentov sa predloží klasifikátoru, ktorý rozhodne, či daný segment je, alebo nie je ruka. Nakoniec je zo všetkých segmentov vybraný ten, o ktorom si je klasifikátor najviac istý, že je ruka (a zároveň prevyšuje danú hranicu). Z vybraného segmentu sa vypočíta bod, ktorý sa berie ako pozícia ruky. Tento bod je pridaný do postupnosti bodov, o ktorých sa ďalej rozhodne, či tvoria nejaké gesto. Ak klasifikátor gesta detekuje gesto, vykoná sa príslušná akcia – simulácia stlačenia niektorého klávesu – a postupnosť sa vymaže. Ak sa dlhšiu dobu v obraze nevykoná žiadna zmena a klasifikátor gesta nezistí žiadne gesto, postupnosť sa vymaže tiež.

3.1.1 Rozdielový obraz

Základ pre predspracovanie a segmentáciu tvorí takzvaný **rozdielový obraz** (Obr. 3.2, prvý štvorec). Rozdielový obraz je obraz, ktorý vznikne odčítaním 2 po sebe idúcich čiernobielych obrázkov v absolútnej hodnote. Tento obraz obsahuje zmeny - pohybujúce sa objekty. Statické objekty sa tam nevyskytnú, čo nám umožní ich veľmi ľahko odfiltrovať. V tomto obraze sa vyskytnú obrisy pohybujúcich sa objektov, pretože ku zmenám dochádza najviac na hranách. Podľa rýchlosti pohybu môžu byť obrisy hrubšie, alebo tenšie.



Obr. 3.1: Základný algoritmus

3.1.2 Predspracovanie vstupného obrazu

Odfiltrovanie šumu

Odfiltrovanie šumu zabezpečí hranica – *threshold*. Všetky pixle svetlejšie ako určitá konštanta, budú vykreslené bielou. Ideálna hranica je taká, ktorá potlačí šum, ale zachová čo najviac z ostatných zmien – čiže by mala byť najmenšia možná. Pohybujeme sa v odtieňoch šedej, čiže v rozmedzí hodnôt $0 \dots 255$. Praktické testy ukázali, že vhodnou hodnotou pre hranicu je 7.

Rozpitie pixlov

Kvôli segmentácii potrebujeme, aby jednotlivé segmenty boli súvislé, čiže aby obrys ruky tvoril jeden celok. Lahko sa nám však môže stať, že obrys ruky je niekde prerušený – nedostatočná zmena, prípadne iné dôvody. Predpokladáme však, že všetky časti jedného segmentu sú dostatočne blízko. Spojiť segmenty nám pomôže rozpitie pixlov.

Okolo každého pixla zafarbíme štvorec s veľkosťou 11×11 . Hodnota 11 pre stranu štvorca sa ukázala, vzhľadom na rozlíšenie 400×300 , ako najvhodnejšia. Príliš veľké hodnoty spájajú aj časti, ktoré nepatria do toho istého segmentu, príliš malé zase nespoja časti, ktoré sú ďalej od seba.

3.1.3 Segmentácia

Rozpítý obrázok si teraz vieme predstaviť ako graf, pričom hrana je medzi každými 2 susediacimi pixlami (v štyroch smeroch). Segmentácia je vlastne len nájdenie komponentov v tomto grafe. Na to môžeme použiť napríklad prehľadávanie do šírky.

Potrebuje ešte nájsť opísaný obdĺžnik. Spravíme to tak, že nájdeme najľavejší, najpravejší, najvrchnejší a najspodnejší bod segmentu.

3.1.4 Predspracovanie segmentov

V tejto časti sa budeme zaoberať jednotlivými segmentami, ktoré budeme predkladať neurónovej sieti, aby nám rozlíšila, či je to ruka alebo nie.

Každý nájdený obdĺžnik sa naškáluje (pomocou bilineárnej interpolácie¹) na veľkosť vstupu pre neurónovú sieť - v našom prípade 128×128 - a normalizuje sa.

Normalizácia dát

Ideálne vstupy pre neurónovú sieť sú z intervalu $\langle 0, 1 \rangle$. Pixle čiernobielych obrázkov majú hodnoty $\{0 \dots 255\}$. Pri obrázkoch zvolíme pre farbu pozadia hodnotu 0 a pre objekt ostatné hodnoty. Z tohto dôvodu chceme, aby rozdiel v normalizovanej hodnote medzi 0 a 1 bol najväčší a postupne klesal. Preto sme za normalizačnú funkciu zvolili:

$$f(x) = \frac{1}{1+x}$$

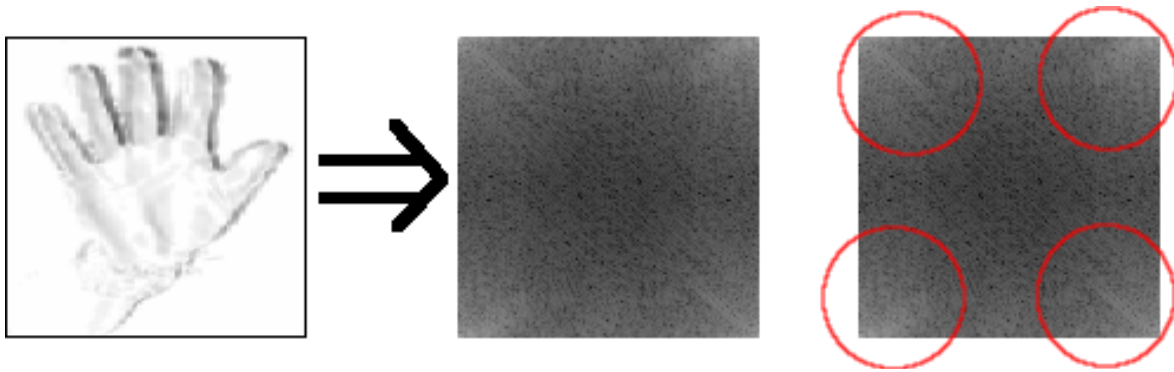
Pre normalizáciu Fourierovej transformácie sme zvolili tú istú funkciu. Hodnoty vo Fourierovej transformácii môžu byť veľmi veľké, preto chceme eliminovať vplyv príliš veľkých hodnôt. Navyše nám to umožňuje stlačiť hodnoty do intervalu $\langle 0, 1 \rangle$ aj bez toho, aby sme poznali maximálnu hodnotu.

Fourierova transformácia

Fourierova transformácia zvykne často pomáhať, keď sa použije na predspracovanie dát pri tréovaní obrazových alebo zvukových vzoriek. Preto sme sa aj my rozhodli vyskúšať, aký bude mať vplyv na úspešnosť. Navyše Fourierove transformácie obrázkov rúk mali určité typické vlastnosti (obr. 3.2 - označené červenými kružnicami).

Fourierova transformácia bola použitá na segment ako celok, potom bola prevedená do reálnych čísiel ako absolútna hodnota z komplexného čísla a následne normalizovaná.

¹kapitola 5.2



Obr. 3.2: Fourierova transformácia a jej typické vlastnosti pre obrázky ruky

Použitie pôvodného namiesto rozdielového obrázka

Nevýhodou rozdielového obrázka je to, že zmena spôsobená pohybom sa v ňom vyskytne dvakrát. Raz na mieste, kam sa objekt posunul a raz na mieste odkiaľ sa posunul. Táto zmena je navyše závislá od rýchlosti pohybu ruky, čo nie je celkom ideálne pre neurónovú sieť a zvyšuje nároky na veľkosť trénovacej množiny. Toto sme chceli eliminovať tým, že sa vyberie ruka z pôvodného obrázka podľa farby. Táto ruka tam bude vždy len raz. Nakoniec sa ukázalo, že tento prístup má viac zlých vlastností ako dobrých.

Pri vyberaní obrázka treba mať nastavené správne parametre, podľa ktorých sa rozhoduje čo pridať do výberu a čo nie. Tieto parametre veľmi závisia od vlastností osvetlenia. Osvetlenie sa môže meniť aj pri pohybe ruky, čo veľmi sťažuje nastavenie správnych parametrov. Pred použitím aplikácie by sa aplikácia musela nakalibrovať, čo znižuje komfort jej použitia. Pri zmene osvetlenia by ju bolo treba opäť prekalibrovať, čo by mohlo byť z hľadiska použiteľnosti neprípustné.

Ďalším problémom je správne tipnutie bodu, ktorý patrí ruke, aby sa z neho mohla odštartovať selekcia. Pokiaľ by bola v danom obdĺžniku len dlaň, tak by nebolo až také ťažké správne trafiť – je takmer isté, že kúsok pod stredom obrázka sa nachádza dlaň. Bohužiaľ sa často stáva, že užívateľ pohne nielen rukou, ale aj predlaktím a segmentačný algoritmus zaradí do segmentu aj predlaktie. Potom môže dôjsť k tomu, že bod ruky netrafieme a algoritmus nemá šancu ruku vyselektovať.

Pri tomto spôsobe by sa nám veľmi zredukovala množina správnych rúk a tento prístup by sa dal použiť iba ako pomôcka, nie ako hlavné kritérium.

3.2 Návrh architektúr neurónových sietí

V tejto kapitole popíšeme rôzne architektúry sietí, ktoré sme vyskúšali, porovnáme ich vlastnosti a úspešnosť pri riešení problému rozpoznania ruky. Vyberieme vhodnú

architektúru, ktorú potom použijeme v našej aplikácii.

3.2.1 Požiadavky na architektúru neurónovej siete

Naším cieľom je vytvoriť vhodnú architektúru neurónovej siete, ktorá bude rozhodovať o danom vstupe, či zodpovedá ruke alebo nie. Navrhujeme niekoľko typov architektúr, ktoré neskôr porovnáme (kapitola 4.4) a vyberieme najvhodnejšiu z nich, ktorú potom použijeme v aplikácii.

Neurónová sieť má rozdeliť vstupy do 2 tried – na tie, ktoré zodpovedajú rukám a na ostatné. Vo všetkých architektúrach na to využijeme jeden výstupný neurón.

Budeme sa snažiť dosiahnuť čo najlepšiu úspešnosť a čo najvyššiu rýchlosť, čiže najmenšiu zložitosť² siete.

3.2.2 Typ 1: Viacvrstvová dopredná neurónová sieť

Viacvrstvová dopredná neurónová sieť (obr. 2.2) je implementáciou klasického viacvrstvého perceptrónu, ktorý sme popísali v kapitole 2.3. Je zložená z viacerých vrstiev neurónov, pričom signál sa šíri len z nižšej vrstvy na vyššiu.

Vstupom každého neurónu sú výstupy neurónov z predošlej vrstvy. Prvá vrstva dostane na vstupe vstup siete.

Ďalšie typy budú odvodené z tohto typu s tým, že na nich budú vykonané nejaké optimalizácie z hľadiska výkonu, alebo k nim bude pridaná nejaká ďalšia informácia.

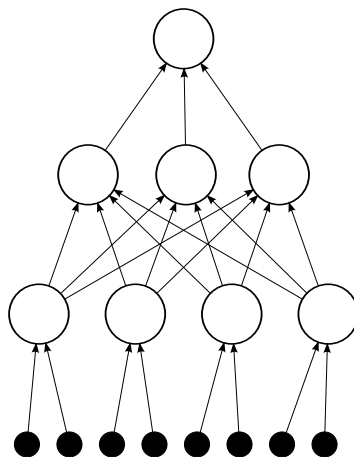
3.2.3 Typ 2: Upravená verzia viacvrstvovej doprednej neurónovej siete

Pri tomto type sme upravili spodnú (vstupnú) vrstvu siete. Vstup sme rozdelili na 16 častí (4×4) a ku každej časti sme pridelili niekoľko neurónov. Každý neurón spracúva len vstupy z jeho časti (obr. 3.3).

Výhodou tohto typu je zvýšenie rýchlosti pri rovnakom počte neurónov – zníži sa počet váh vstupných neurónov. Naš vstup má rozmer $128 \times 128 = 16384$, čo nie je malé číslo. Keď ho rozdelíme na 16 častí, jednotlivé časti budú mať veľkosť $32 \times 32 = 1024$. Takto namiesto toho, aby každý vstupný neurón počítal s 16384 vstupmi počíta len s 1024 vstupmi, čo je $16 \times$ menej. Skupina neurónov pridelená danej časti sa stará len o príznaky zo svojej časti a nie je ovplyvňovaná ostatnými časťami.

Nevýhodou je skutočnosť, že neuróny sú fixne pridelené na jednotlivé vstupy. V pôvodnej sieti si neuróny samé vyberali, ktoré časti vstupu sú pre ne najvýznamnejšie a

²vzhľadom na počet váh



Obr. 3.3: Upravená dopredná neurónová sieť

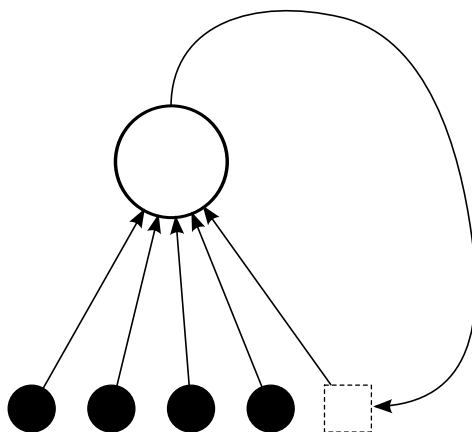
mohli tak lepšie pokryť vstup. Môžeme to však vykompenzovať miernym zvýšením počtu neurónov aby sme dosiahli optimálny pomer výkonu a úspešnosti.

3.2.4 Typ 3: Rekurentná neurónová sieť

V predchádzajúcich podkapitolách sme sa zaoberali doprednými sieťami (feedforward), v ktorých sa informácia šíri len smerom od vstupov k výstupu. V rekurentných sieťach máme navyše rekurentné spojenia, cez ktoré sa informácia prenáša v čase. Informácia z jednotlivých neurónov môže byť v ďalšom kroku použitá ako vstupná informácia pre neuróny.

Naša architektúra rekurentnej neurónovej siete vychádza z upravenej doprednej neurónovej siete, s tým, že namiesto obyčajného neurónu používame rekurentný neurón (obr. 3.4).

Rekurentný neurón obsahuje spätnú väzbu. Spätná väzba sa tvári ako ďalší vstup



Obr. 3.4: Rekurentný neurón

a obsahuje posledný aktualizovaný výstup toho istého neurónu. Po aktivácii neurónu môžeme aktualizovať poslednú aktivačnú hodnotu. Ak to neurobíme, hodnota ostane taká, ako bola predtým. Neurón môžeme aj resetovať, vtedy sa hodnota vynuluje.

Tento typ siete sme navrhli s ohľadom na už existujúce typy sietí a na architektúru aplikácie. Umožňuje nám bez väčšieho zásahu použiť túto sieť tak, aby nám v každom kroku vedela poskytnúť informáciu, či sa jedná o ruku alebo nie. Oproti predošlým typom sietí má navyše informáciu o tom ako reagovala sieť na predošlé obrázky rúk.

Našu jednoduchú rekurentnú neurónovú sieť trénujeme tiež algoritmom *backpropagation*, s tým, že sieti predkladáme postupnosti – v rámci nej sú dáta vždy v tom istom poradí. Rekurentný vstup aktualizujeme iba pri kladnej odozve, čo nám umožní odfiltrovať zlé obrázky z postupnosti a nestratiť informáciu o tom, ako sieť reagovala na predchádzajúce ruky. Ruky v rámci postupnosti sa väčšinou od seba navzájom líšia len málo. Po každej postupnosti resetujeme rekurentný vstup na 0, čím sa sieť dostane do východzieho stavu, v ktorom by mala reagovať na novú ruku, ktorá môže byť iná ako doteraz videné ruky.

Kapitola 4

Experimenty

V tejto kapitole porovnáme jednotlivé typy architektúr neurónových sietí, porovnáme ich úspešnosť a ukážeme, ktorá z nich je najvhodnejšia. Ďalej sa budeme venovať predspracovaniu segmentov, ktoré priamo ovplyvňuje vlastnosti trénovania a úspešnosť sietí.

4.1 Trénovanie sietí

Trénovanie dopredných sietí sme robili pomocou algoritmu *backpropagation* - algoritmu spätného šírenia chyby [Hay99]. Sieti sme postupne predkladali dáta s informáciou, či sa jedná o ruku alebo nie. Pred každou epochou sme dáta náhodne preusporiadali.

Pri rekurentných sieťach sme predkladali postupnosti dát, pričom sme zachovávali poradie obrázkov v postupnosti. Rekurentné vstupy sme aktualizovali len v prípade, že sme rozpoznali ruku. Po každej postupnosti sme rekurentné vstupy resetovali na 0. Takto simulujeme správanie siete v reálnej aplikácii. Aj pri tomto type sietí sme mali k dispozícii informáciu či sa jedná o ruku, alebo nie.

4.2 Testovanie a vyhodnocovanie neurónových sietí

Pri trénovaní sa snažíme minimalizovať kvadratickú chybu. Preto aj pri vyhodnocovaní úspešnosti sietí budeme používať túto metriku.

Kvadratická chyba sa počíta takto:

$$e = \frac{1}{2}(t - o)^2$$

t je cieľová hodnota a o je výstup siete. V tabulkách budeme uvádzať priemernú kvadratickú chybu, ktorú budeme počítať ako súčet všetkých kvadratických chýb deleno počet testovacích vstupov.

4.3 Trénovacie a testovacie dáta

Na generovanie dát sme použili upravenú verziu našej aplikácie (kapitola 5.8), ktorá umožňovala ukladanie vysegmentovaných obrázkov na disk bez toho, aby došlo k výraznému spomaleniu aplikácie.

Trénovacie dáta sme rozdelili na dve sady. Každá sada bola rozdelená na dve disjunktné množiny – trénovaciú a testovaciú. Ku každému vstupu sme mali informáciu, či sa jedná o ruku, alebo nie.

Pomocou prvej sady sme vyhodnocovali vlastnosti rôznych typov architektúr neurónových sietí a vplyv Fourierovej transformácie. Obsahovala dáta, ktoré boli rozdelené do postupností. To nám umožňovalo trénovať nimi rekurentné neurónové siete. Dopredné siete tieto dáta brali ako jednotlivé obrázky. Na jednu postupnosť pripadá v priemere približne 11 obrázkov. Sada obsahuje rozdielové obrázky¹ a k nim zodpovedajúce Fourierove transformácie.

Druhá sada bola špecializovaná na vyhodnocovanie použitia pôvodného vs. rozdielového obrázka. Z tejto sady boli vyňaté tie obrázky rúk, ktoré sa nepodarilo algoritmu flood-fill správne vyselektovať. Porovnávali sme teda úspešnosti v ideálnych prípadoch. Sada obsahuje rozdielové obrázky, k nim zodpovedajúce pôvodné obrázky a Fourierove transformácie zodpovedajúce obom typom.

Sada	Testovacia množina		Trénovacia množina	
	ruky	ostatné	ruky	ostatné
1. sada	168	478	245	568
2. sada	96	174	149	165

Tabuľka 4.1: Veľkosti sád vstupov

Keďže obrázkov rúk je v dátach menej ako ostatných a sieť, ktorá by na všetko povedala že to ruka nie je by mala úspešnosť $> 60\%$, percentuálnu úspešnosť počítame ako **priemer percentuálnej úspešnosti** na rukách a na ostatných obrázkoch.

4.4 Porovnanie architektúr neurónových sietí

4.4.1 Typ 1: Viacvrstvová dopredná neurónová sieť

Experimentálne sme zistili, že dvojvrstvová sieť na tento problém stačí a tretia vrstva nepomáha – nedarilo sa nám ju natréňovať. V tabuľke 4.2 uvádzame úspešnosti pre 2-vrstvovú sieť na dátach Fourierových transformácií.

¹kapitola 3.1.1

Počet neurónov	Testovacia množina		Trénovacia množina	
	úspešnosť(%)	chyba	úspešnosť(%)	chyba
30	64,3	0,1036	99,8	0,001
40	63,22	0,0956	99,8	0,001
45	64,19	0,098	99,8	0,001
47	65,07	0,099	100	0,0004
55	63,46	0,099	100	0,0004
60	63,48	0,099	100	0,0003

Tabuľka 4.2: Porovnanie úspešnosti doprednej NS pri rôznych počtoch neurónov

Môžeme si všimnúť, že pridávanie neurónov donekonečna nemusí pomôcť a zbytočne veľa neurónov môže aj uškodiť.

4.4.2 Typ 2: Upravená verzia viacvrstvovej doprednej neurónovej siete

Pri tomto type siete sa nám podarilo natrénovať 2 aj 3-vrstvové architektúry, v tabuľke 4.3 sa teda budú vyskytovať obe. Vrstvy sú písané od najvrchnejšej po najspodnejšiu. Pri spodnej vrstve je v zátvorke počet neurónov na jednu časť obrázka.

Všimnime si že tretia vrstva nám v úspešnosti veľmi nepomohla, hoci skrátila dobu tréovania – stačila asi 1/4 epoch oproti 2 vrstvám. Keďže je v praxi pridanie ďalšej vrstvy spomalením, dve vrstvy budú v tomto prípade lepšie.

Oproti sieťam typu 1, vidíme určitý nárast úspešností o 2,4 percentuálnych bodov, pričom sieť obsahuje menej váh, čiže sa rýchlejšie trénuje aj rýchlejšie počíta.

4.4.3 Typ 3: Rekurentná neurónová sieť

Pri tomto type sietí sme mali viacero možností ako zvoliť rekurentné neuróny. Mohli sme ich nechať len na spodnej vrstve, alebo sme ich mohli dať do celej neurónovej siete.

V tabuľke 4.4 vidíme, že lepšie si počína sieť s rekurenciou len na spodnej vrstve. Obe siete sú 2-vrstvové, v zátvorke je počet neurónov na jednu časť dát – tak ako v predošlej kapitole. Informácie z vyšších vrstiev sú pre sieť mätúce. V ďalších testoch (tabuľka 4.5) teda budeme vychádzať z architektúry, ktorá má rekurentné neuróny len na spodnej vrstve. Vyskúšame 2 aj 3-vrstvové architektúry.

Ani v tomto prípade nám nepomohla ďalšia vrstva. Pri 2-vrstvových architektúrach rekurencia mierne pomohla.

Počet neurónov	Testovacia množina		Trénovacia množina	
	úspešnosť(%)	chyba	úspešnosť(%)	chyba
48(3)	66,39	0,0872	99,39	0,0131
112(7)	66,56	0,1015	99,65	0,004
176(11)	67,47	0,0957	99,39	0,0112
240(15)	64,89	0,1207	99,83	0,0016
320(20)	64,12	0,1108	99,91	0,0007
7; 48(3)	66,32	0,0947	99,53	0,0038
7; 112(7)	67,05	0,1042	99,21	0,0053
7; 176(11)	67,37	0,1038	99,56	0,0023
7; 208(13)	65,8	0,1035	99,36	0,0052
7; 272(17)	66,14	0,1032	98,95	0,0078
7; 320(20)	65,84	0,1156	99,91	0,0009
11; 112(7)	64,1	0,1108	99,91	0,0004
12; 144(9)	66,27	0,0978	99,74	0,003
12; 480(30)	67,24	0,0983	99,53	0,0023
13; 208(13)	63,86	0,1001	99,33	0,0036
17; 112(7)	62,52	0,1043	100	0,0023

Tabuľka 4.3: Porovnanie úspešnosti upravenej NS pri rôznych počtoch neurónov

Rekurentná vrstva	Testovacia množina		Trénovacia množina	
	úspešnosť(%)	chyba	úspešnosť(%)	chyba
spodná(11)	67,88	0,1106	98,8	0,005
spodná(20)	67,01	0,109	99,47	0,0029
všetky(11)	61,84	0,1055	99,82	0,0026
všetky(20)	65,07	0,1116	99,47	0,0033

Tabuľka 4.4: Porovnanie úspešnosti rekurentnej NS pri rôznom umiestnení rekurencie

Počet neurónov	Testovacia množina		Trénovacia množina	
	úspešnosť(%)	chyba	úspešnosť(%)	chyba
112(7)	65,63	0,1049	98,54	0,0056
176(11)	67,88	0,1106	98,8	0,005
240(15)	65,16	0,1173	99,53	0,0028
320(20)	67,01	0,109	99,47	0,0029
7; 176(11)	66,13	0,1107	99,74	0,0018
12; 144(9)	65,73	0,0937	98,87	0,0083

Tabuľka 4.5: Porovnanie úspešnosti rekurentnej NS pri rôznych počtoch neurónov

4.4.4 Zhrnutie

Vylepšeniami architektúry sa nám podarilo zvýšiť úspešnosť o **2,81** percentuálnych bodov a nájsť architektúru s úspešnosťou **67,88%**. Ukázalo sa, že je lepšie použiť 2-vrstvovú sieť ako 3-vrstvovú. Ďalej sme zistili, že keď budú mať neuróny prístup len k časti obrázka, môžeme ich použiť viac a dosiahnuť lepšiu úspešnosť. Nakoniec sa nám podarilo ukázať, že pridaním rekurentných spojení na spodnú vrstvu dokážeme ešte trochu zlepšiť úspešnosť.

Úspešnosť na rukách sa pohybovala okolo 40-50% a na ostatných obrázkoch 80-90%. Sieť teda dokáže dobre odfiltrovať zlé obrázky, pričom má pomerne dobrú úspešnosť aj na rukách.

4.5 Porovnanie úspešnosti pri rôznom spôsobe spracovania

4.5.1 Vplyv Fourierovej transformácie

Pre porovnanie Fourierovej transformácie a rozdielových obrázkov sme použili 3-vrstvovú upravenú neurónovú sieť. Vybrali sme ju preto, lebo 2-vrstvovú sa nám na rozdielových obrázkoch nepodarilo natrénovať. Ani pri tejto architektúre sa nám však nedarilo sieť natrénovať na úspešnosť viac ako 71%. Už z toho vidno výhodu Fourierovej transformácie. Počty neurónov sme vybrali také, ktoré dopadli v predošlom teste najlepšie (tabuľka 4.3) – [7; 112(7)] a [7; 176(11)].

Z tabuľky 4.6 je vidieť zlepšenie o 2 percentuálne body pri použití Fourierovej transformácie. Okrem toho nám Fourierova transformácia dáva potenciál zdvihnúť úspešnosť pri väčšom množstve trénovacích dát a širšie možnosti tréningu neurónových sietí.

Typ dát	Testovacia množina		Trénovacia množina	
	úspešnosť(%)	chyba	úspešnosť(%)	chyba
Obrázok(7)	65,16	0,1052	70,71	0,0864
Fourierova tr.(7)	67,05	0,1042	99,21	0,0053
Obrázok(11)	65,06	0,096	70,76	0,0887
Fourierova tr.(11)	67,37	0,1038	99,56	0,0023

Tabuľka 4.6: Porovnanie úspešnosti NS na dátach Fourierovej tr. a rozdielového obrázku

4.5.2 Rozdielový vs. pôvodný obraz

Pri tomto teste sme použili na tréovanie a testovanie *sadu 2* (kapitola 4.3), v ktorej boli tréovacie a testovacie dáta podobnejšie, preto je aj úspešnosť vyššia. Využili sme 3-vrstvovú architektúru upravenej doprednej neurónovej siete s počtami neurónov [7; 176(11)], z podobných dôvodov ako v predošlej kapitole.

Typ dát	Testovacia množina		Trénovacia množina	
	úspešnosť(%)	chyba	úspešnosť(%)	chyba
Rozdielový obr.	76,64	0,071	92,9	0,0303
Pôvodný obr.	66,51	0,123	77,46	0,0782
Rozdielový – FT	81,78	0,0821	95,06	0,0128
Pôvodný – FT	74,73	0,1156	90,13	0,0279

Tabuľka 4.7: Porovnanie úspešnosti NS na dátach rozdielového obrázku, pôvodného obrázku a ich Fourierových transformácií

Z tabuľky 4.7 je vidieť, že úspešnosť siete na dátach pôvodného obrázku je značne nižšia ako na rozdielovom obrázku a to aj pri použití Fourierovej transformácie. Z toho vyplýva že tento typ dát nie je vhodný.

4.5.3 Zhrnutie

Z tabuliek vidíme, že Fourierova transformácia pomáha nielen pri tréovaní, ale aj pri celkovej úspešnosti neurónovej siete. Sieť je možné rýchlejšie a lepšie natréovať a tréovať aj na väčších dátach.

Dáta získané na základe pôvodného obrázku sa ukázali ako nevhodné, sieť sa na nich veľmi ťažko učí a ani Fourierova transformácia na týchto dátach nedosahuje takú vysokú úspešnosť ako z rozdielového obrázku. Preto sme sa rozhodli v aplikácii tento prístup nepoužiť.

Kapitola 5

Implementácia

V tejto kapitole predstavíme triedy a popíšeme implementačné detaily jednotlivých častí algoritmu.

Budeme sa venovať základným triedam, riešeniam elementárnych vecí, detailom spracovania obrazu, rozpoznávaniu ruky a gesta. Podrobne si popíšeme implementáciu neurónových sietí.

Spomenieme aj niektoré problémy a ich riešenia a možnosti paralelizácie.

5.1 Neurónové siete

Naša implementácia neurónových sietí kopíruje vrstvomý model. Základnou jednotkou je spojený perceptrón, skupinu perceptrónov zastrešuje vrstva a skupinu vrstiev neurónová sieť.

Ku každej z týchto úrovní máme všeobecnú triedu a rôzne implementácie. Na obrázku 5.1 je class diagram nášho riešenia.

5.1.1 Perceptron

Perceptron je abstraktná trieda, ktorá zovšeobecňuje rôzne typy perceptrónov, ktoré sa líšia rôznymi aktivačnými funkciami.

Obsahuje trénovací algoritmus pre perceptróny, ale aj podporu algoritmu *backpropagation* a klasifikačný algoritmus. Okrem toho obsahuje užitočné funkcie na randomizáciu váh, ukladanie a načítanie váh.

Pri volaní metód, ktoré majú ako parameter vstup siete si perceptrón sám pridá *bias*¹, takže ho nie je nutné pridávať k vstupu.

Možnosť vytvorenia rôznych typov perceptrónov zabezpečujú abstraktné metódy

¹kapitola 2.1

- `float activationFunction(const vector<float>* input),`
- `float derivativeFunction(float x),`

ktoré treba v potomkovi definovať.

Dôležité metódy

- `void train(vector<float> input, int target)`
Jednoduché tréningovanie perceptrónov.
Vstup: *input* - vstup siete, *target* - požadovaný výstup
- `void trainDelta(vector<float> input, float delta)`
Delta tréningovanie pre *backpropagation* algoritmus.
Vstup: *input* - vstup siete, *delta* = δ z učiaceho pravidla: $\Delta w_i = \alpha \delta x_i^2$ [Hay99, s. 74]
- `float classify(vector<float> input)`
Klasifikačný algoritmus.
Vstup: *input* - vstup siete
Výstup: reálne číslo z intervalu (0, 1) - výstup siete.
- `void prepare(vector<float>* input);`
Predpríprava vstupu pre spracovanie v perceptróne - pridáva *bias*.
Túto metódu treba volať tam, kde metóda dostáva užívateľský vstup (bez *biasu*).
Vstup: **input* - pointer vstup siete
Výstup: metóda upraví priamo vektor, ktorý jej bol daný ako vstup.

5.1.2 ContinuousPerceptron

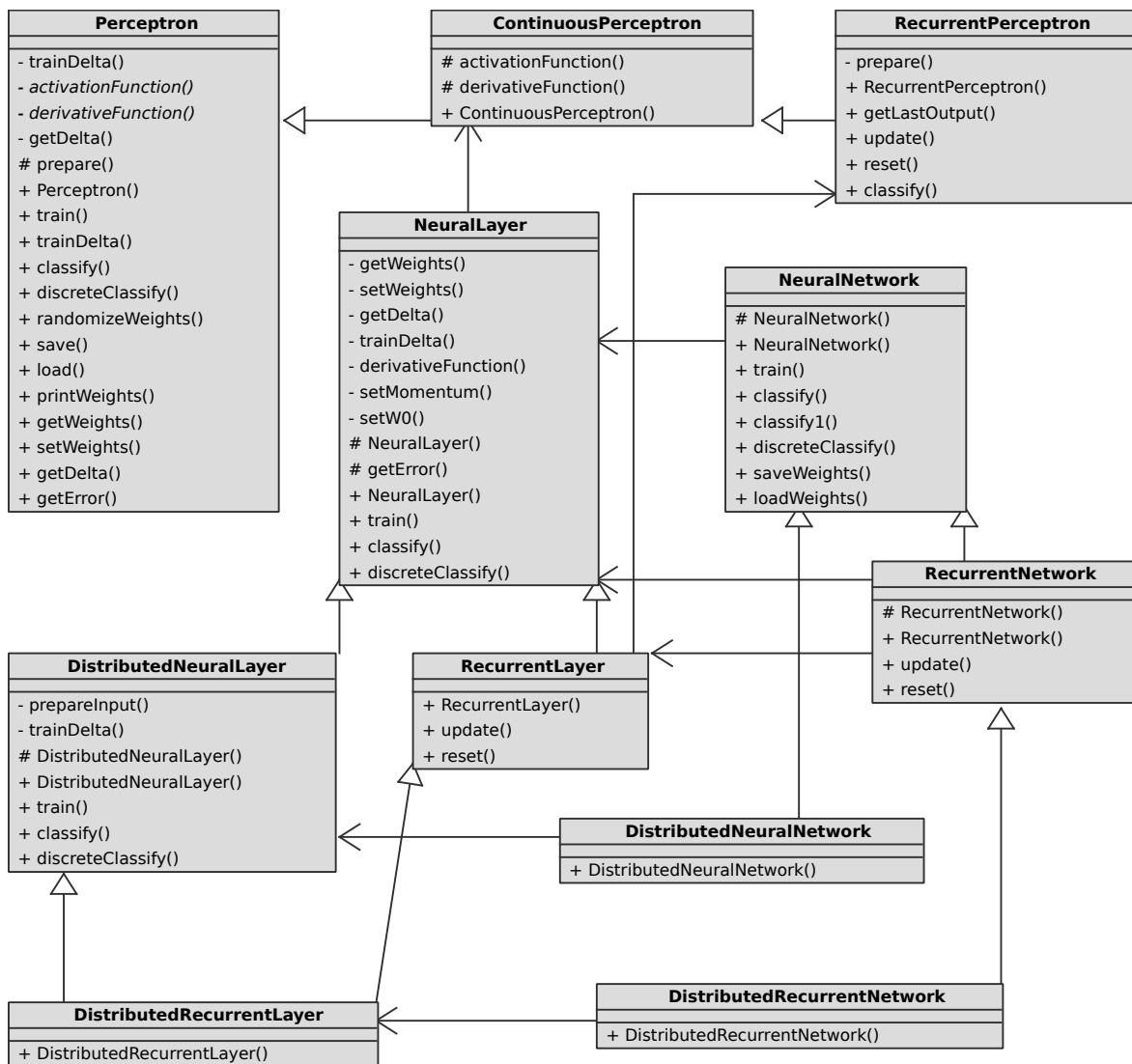
Trieda *ContinuousPerceptron* je implementácia spojitého perceptrónu. Dedí od triedy *Perceptron* Definuje aktivačnú funkciu

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

a jej deriváciu

$$f'(x) = \frac{e^x}{(e^x + 1)^2}$$

²používame mierne inú notáciu ako je v [Hay99]



Obr. 5.1: Neurónové siete - Class diagram

5.1.3 RecurrentPerceptron

Trieda *RecurrentPerceptron* rozširuje triedu *ContinuousPerceptron* o rekurentný vstup a metódy na manipulovanie s ním – `void update()` a `void reset()` na aktualizáciu a resetovanie rekurentného vstupu.

5.1.4 NeuralLayer

NeuralLayer je trieda, ktorá zoskupuje spojité perceptróny do vrstvy. Umožňuje trénovanie na viacrozmerne výstupy a podporuje *backpropagation* algoritmus.

Táto trieda poskytuje základnú implementáciu – predkladá celý vstup každému neurónu. Je zároveň základnou triedou pre vrstvy neurónov, od ktorej potom dedia iné typy vrstiev. Zmena vlastností triedy sa deje v konštruktore, kde je možné určiť typy jednotlivých neurónov a ich parametre.

Dôležité metódy

- `float train(vector<float> input, vector<int> target)`
Jednoduché trénovanie jednotlivých perceptrónov.
Vstup: *input* - vstup siete, *target* - požadované výstupy
Výstup: chyba na danom vstupe.
- `void trainDelta(vector<float> input, vector<float> delta)`
Delta trénovanie pre *backpropagation* algoritmus.
Vstup: *input* - vstup siete, *delta* – delty pre jednotlivé neuróny - podobne ako v kapitole 5.1.1
- `vector<float> classify(vector<float> input)`
Klasifikačný algoritmus.
Vstup: *input* - vstup siete
Výstup: pole čísel z intervalu (0, 1) - výstup siete.

5.1.5 DistributedNeuralLayer

Trieda *DistributedNeuralLayer* dedí od triedy *NeuralLayer*. Poskytuje podobnú funkcionálnu ako *NeuralLayer*, ale vstup je rozdelený na $n \times m$ častí a každý neurón má k dispozícii len jednu časť.

Okrem konštruktora bolo v tomto prípade nutné upraviť aj metódy `float train(vector<float>, v` a `vector<float> classify(vector<float>)`, tak, aby trénovanie a klasifikácia prebiehala na príslušnej časti vstupu.

5.1.6 RecurrentLayer

Trieda *RecurrentLayer* dedí od triedy *NeuralLayer*. Namiesto spojitých perceptrónov používa rekurentné. Navyše pridáva metódy `void update()` a `void reset()`, ktoré umožňujú aktualizáciu a resetovanie rekurentného vstupu neurónov.

5.1.7 DistributedRecurrentLayer

Trieda *DistributedRecurrentLayer* dedí od triedy *RecurrentLayer* a *DistributedRecurrentLayer*. Tu sa ukazuje sila nášho objektovo orientovaného modelu ako aj sila viacnásobného dedenia v C++. Táto trieda obsahuje len konštruktor, ktorý volá konšuktory rodičovských tried. Prakticky bez námahy vieme takto vytvoriť nový typ siete spojením dvoch existujúcich.

Spojením dvoch typov vrstiev získavame máme nový typ vrstvy, ktorá obsahuje rekurentné neuróny, pričom každý z nich má k dispozícii len časť vstupu.

5.1.8 NeuralNetwork

Trieda *NeuralNetwork* zoskupuje vrstvy do viacvrstvovej siete. Implementuje viacvrstvovú sieť s vrstvami typu *NeuralLayer* a zároveň slúži ako základná trieda pre neurónové siete, od ktorej potom dedia ostatné siete.

Poskytuje implementáciu *backpropagation* algoritmu, ukladanie a načítanie váh do/zo súboru.

Dediace triedy väčšinou mierne menia architektúru napríklad použitím iného typu vrstvy. Tieto zmeny sa zvyknú robiť v konštruktore.

Dôležité metódy

- `float train(vector<float> input, vector<int> target)`
Backpropagation algoritmus.
Vstup: *input* - vstup siete, *target* - požadované výstupy
Výstup: chyba na danom vstupe.
- `vector<float> classify(vector<float> input)`
Klasifikačný algoritmus.
Vstup: *input* - vstup siete
Výstup: pole čísel z intervalu (0, 1) - výstup siete.

5.1.9 DistributedNeuralNetwork

Trieda *DistributedNeuralNetwork* dedí z *NeuralNetwork*. Jedinou zmenou je, že ako spodnú vrstvu používa *DistributedNeuralLayer*.

5.1.10 RecurrentNetwork

Trieda *RecurrentNetwork* je potomkom triedy *NeuralNetwork*. Ako spodnú vrstvu používa *RecurrentLayer*. Okrem toho implementuje metódy `update()` a `reset()`, ktoré aktualizujú a resetujú rekurentnú vrstvu.

5.1.11 DistributedRecurrentNetwork

Trieda *DistributedRecurrentNetwork* dedí z *RecurrentNetwork*. Ako spodnú vrstvu používa *DistributedRecurrentLayer*.

5.2 Triedy pre obrázky

Pre účely aplikácie potrebujeme špecifickú triedu na obrázky, ktorá spĺňa nasledujúce vlastnosti:

- Rýchly prístup k jednotlivým pixlom.
- Vystrihnutie(crop)
- Škálovanie
- Maskovanie
- Floodfill

Okrem toho potrebujeme, aby obsahovala aj pomocné metódy, ktoré sú potrebné na niektoré algoritmy a testovanie, konverziu do rôznych formátov, ukladanie na disk a iné.

Trieda *HCIImage* je abstraktná trieda, ktorá implementuje základné veci, ktoré sú nezávislé na type pixlu. Od tejto triedy potom dedia triedy *GrayScaleImage* a *ColorImage*. Tieto implementujú metódy, ktoré sú závislé na type pixlu (odtiene šedej alebo farebný pixel).

Vystrihnutie

Na vystrihnutie slúži metóda `copy(QRect r)`, ktorá berie ako parameter obdĺžnik, ktorého obsah potom vráti ako výsledok.

Škálovanie

Metóda `scale(unsigned w, unsigned h)` preškáluje obrázok na novú veľkosť - $w \times h$ pixlov.

Využíva bilinéarnu interpoláciu – vypočíta súradnice nového bodu a pomocou susedných bodov z pôvodného obrázka vypočíta farbu.

Bilineárna interpolácia má niekoľko výhod. Je stále dosť rýchla, pričom eliminuje kostrbatosť, ktorá môže mať výrazný vplyv pri použití Fourierovej transformácie.

Maskovanie

Metóda `mask(HCImage *mask, bool invert = false)` aplikuje masku na obrázok. Hodnotu každej farby zníži podľa hodnoty masky v danom bode.

Maskovanie použijeme na odstránenie nepodstatných častí obrázka, aby neovplyvňovali neurónovú sieť. Využívame bitové operátory na urýchlenie výpočtu.

Floodfill selekcia

Techniku *floodfill selekcie* si môžeme predstaviť ako „čarovnú paličku“³, alebo nástroj na selekciu susedných pixlov s podobnými farbami. Techniku *floodfill* používame na získanie masky v metóde `getAdaptiveFloodFillSelectionMask(int sx, int sy, int threshold, float originalFactor, float changeFactor)` (Obr. 5.2). Metóda berie niekoľko parametrov – pozíciu bodu z ktorého má floodfill začať, hranicu podobnosti farieb, vplyv originálnej farby a zmeny. Referenčná farba sa získa ako priemer začiatočného bodu a jeho okolitých bodov.

Metóda vráti obrázok, ktorý sa dá použiť ako maska, ktorá odstráni⁴ pixle, ktoré neboli vybraté pomocou techniky *floodfill*.

Algoritmus funguje ako prehľadávanie do šírky, pričom do susedného pixlu pôjde len v prípade, že ich rozdiel je menší ako hranica. V prípade farebného obrázka sa tento rozdiel počíta po zložkách a menší musí byť každý z nich. Pri prejdení do nasledujúceho pixlu sa upraví referenčná farba podľa nasledujúceho vzorca: $refcolor = reference \cdot originalFactor + (refcolor \cdot changeFactor + color \cdot (1 - changeFactor)) \cdot (1 - originalFactor)$, kde *reference* je začiatočná referenčná farba, *refcolor* je aktuálna referenčná farba a *color* je farba pixlu.

³známu z grafických programov ako je GIMP, či Photoshop

⁴zacierni

```
1 template <class T>
2 HImage<T>* HImage<T>::getAdaptiveFloodFillSelectionMask( unsigned sx,
3     unsigned sy, int treshold, float originalFactor, float changeFactor)
4 {
5     T reference = getAverageColor(sx,sy);
6     ImageBuffer b;
7     b.resize(width()*height(),0);
8     queue<pair<pair<unsigned, unsigned>,T> > f;
9     f.push(make_pair(make_pair(sx,sy),reference));
10    while(!f.empty())
11    {
12        unsigned x = f.front().first.first;
13        unsigned y = f.front().first.second;
14        T refcolor = f.front().second;
15        f.pop();
16        if(x>=width()) continue;
17        if(y>=height()) continue;
18
19        T color = pixel(x,y);
20        if(!similar(refcolor,color,treshold) || b[x+y*w]!=T(0)) continue;
21        b[x+y*w]=0xffffffff;
22        refcolor = reference*originalFactor + (refcolor*changeFactor+color
23            *(1-changeFactor))*(1-originalFactor);
24
25        f.push(make_pair(make_pair(x+1,y),refcolor));
26        f.push(make_pair(make_pair(x-1,y),refcolor));
27        f.push(make_pair(make_pair(x,y+1),refcolor));
28        f.push(make_pair(make_pair(x,y-1),refcolor));
29    }
30    HImage<T> * maskImage = create(b,width(),height());
31    return maskImage;
32 }
```

Obr. 5.2: Zdrojový kód funkcie HImage<T>::getAdaptiveFloodFillSelectionMask()

5.3 Získanie obrazu z webkamery

Na získanie obrazu z webkamery používam triedu, ktorá pochádza z programu *Kapture* [kap12], a ktorú sme upravili pre potreby využitia v našej aplikácii. Program je šírený pod GNU GPL licenciou.

Webkamera vracia obraz vo formáte MJPEG(Motion JPEG), teda postupnosť obrázkov vo formáte JPEG. Úlohou našej triedy je vždy, keď je buffer pripravený, pretransformovať obrázok do niektorej z tried na obrázky. JPEG kóduje farby vo formáte YUV, preto ich musíme pretransformovať do RGB. V prípade čiernobieleho obrázka iba zoberieme zložku Y.

5.4 Spracovanie obrazu - Trieda ImageProcessor

Trieda *ImageProcessor* má na starosti predspracovanie obrazu z webkamery, jeho segmentáciu a posunutie na ďalšie spracovanie triede *HandRecognizer*. *ImageProcessor* využíva vlákna na paralelizáciu úloh a jeho práca je rozdelená do niekoľkých krokov, ktoré nemôžu byť vykonané paralelne. Hlavnou metódou je metóda `processImage` (`const GrayScaleImage &image, const ColorImage &colorimg`), ktorá spravuje vlákna a spúšťa jednotlivé kroky.

Spúšťanie vlákien je realizované pomocou funkcie `QtConcurrent::run(Function function, ...)` z knižnice *Qt*, ktorá umožňuje spustiť funkciu s danými parametrami v novom vlákne. Jednotlivé metódy sú upravené tak, aby vedeli upravovať aj len určitú časť obrázka. Spúšťajú sa v rôznych vláknach s rôznymi časťami obrázka.

5.4.1 Krok 1: Príprava

Príprava spočíva vo vytvorení rozdielového obrázka. Pamätáme si pôvodný obrázok a dostaneme nový. Spočítame rozdiel a zapamätaný obrázok potom nahradíme novým.

Na začiatku je pôvodný obrázok celý čierny a v prvom rozdielke toho bude veľa, ale je nízka pravdepodobnosť, že to bude zle vyhodnotený. V prípade zlého vyhodnotenia však 1 obrázok stále nestačí na gesto.

5.4.2 Krok 2: Algoritmus rozpitia

Označíme si dĺžku strany štvorca k .

Triviálny algoritmus rozpitia – pre každý pixel vyrobíme okolo neho čierny štvorček – je časovo náročný – má zložitosť $O(k^2.w.h)$. Preto potrebujeme efektívnejší algoritmus. Efektívny algoritmus rozpitia treba robiť v 2 krokoch. Najskôr sa rozpije v X -ovom

smere a potom v Y -ovom.

Algoritmus pre X -ový smer (Obr. 5.3) funguje nasledovne: Ideme postupne po riadkoch a pre každý riadok si pamätáme pokiaľ máme kresliť čiernu. Na začiatku je to 0. Pozeráme sa vždy o $\frac{k}{2}$ pixlov ďalej ako kreslíme. Vždy, keď vidíme pixel inej farby ako bielej, tak hranicu – pokiaľ máme kresliť čiernou – posunieme na hodnotu o k väčšiu ako je y súradnica pixla, ktorý kreslíme. Pokiaľ sme pred hranicou, kreslíme čiernou, inak bielou.

Keďže riadky sa spracovávajú nezávisle, obrázok sa dá rozdeliť na niekoľko častí, ktoré sa dajú spracovať paralelne, čo je ďalšou výhodou efektívnejšieho algoritmu.

Pre Y -ový smer je to analogické.

Tento algoritmus má zložitosť $O(w.h)$ v praxi pre $k = 11$ je asi $5\times$ rýchlejší.

5.4.3 Krok 3: Segmentácia

Ako sme už spomenuli v kapitole 3.1.3, obrázok si reprezentujeme ako graf, na ktorý následne použijeme algoritmus prehľadávania do šírky.

Algoritmus postupne spustíme z každého bodu obrázka, ktorý nie je prázdny a zároveň sme ho predtým žiadnym predošlým behom algoritmu nenavštívili. Každý beh nám vráti obdĺžnik, ktorý ak spĺňa veľkostné obmedzenia, tak je vložený do fronty na ďalšie spracovanie.

Táto časť je realizovaná len jedným vláknom, keďže obrázok nie je možné vhodne rozdeliť.

5.5 Rozpoznanie ruky

Rozpoznanie ruky má na starosti trieda *HandRecognizer*. Beží naraz v niekoľkých vláknach – každé vlákno spracováva len jeden segment. Trieda *HandRecognizer* vyberie z fronty obdĺžnik čakajúci na spracovanie. Podľa rozmerov a pozície vystrihne príslušnú časť obrázka.

Na vystrihnutý obrázok sa aplikuje Fourierova transformácia (kapitola 3.1.4) a normalizuje sa (kapitola 3.1.4). Výsledok je potom predložený ako vstup neurónovej siete, ktorá vráti hodnotu $> 0,5$ ak to rozpoznala ako ruku, ináč vráti hodnotu $\leq 0,5$.

```
1 void ImageProcessor::expandPixelsX(int sy, int ex, int ey, GrayScaleImage
2   * imgIn, GrayScaleImage * imgOut)
3 {
4   int sx = 0;
5   for(int y = sy; y < ey; y++)
6   {
7     int endblack = 0;
8     for(int x = sx; x < sx + PIXEL_RADIUS && x < ex; x++)
9     {
10      uchar c = imgIn->pixel(x, y);
11      if(c != WHITE)
12      {
13        endblack = x + PIXEL_RADIUS;
14      }
15    }
16    for(int x = sx; x < ex; x++)
17    {
18      int xx = x + PIXEL_RADIUS;
19      if(xx < ex)
20      {
21        uchar c = imgIn->pixel(xx, y);
22        if(c != WHITE)
23        {
24          endblack = xx + PIXEL_RADIUS;
25        }
26      }
27      if(x < endblack)
28      {
29        imgOut->setPixel(x, y, 0);
30      }
31      else
32      {
33        imgOut->setPixel(x, y, 0xFF);
34      }
35    }
36  }
```

Obr. 5.3: Zdrojový kód funkcie ImageProcessor::expandPixelsX()

```
1 QRect ImageProcessor::segment(unsigned sx, unsigned sy, uchar color,
   GrayScaleImage * image, QRect rect)
2 {
3     queue<pair<unsigned, unsigned> > f;
4     f.push(make_pair(sx, sy));
5     while(!f.empty())
6     {
7         unsigned x = f.front().first;
8         unsigned y = f.front().second;
9         f.pop();
10        if(x>=image->width()) continue;
11        if(y>=image->height()) continue;
12        if(image->pixel(x,y)!=0) continue;
13        image->setPixel(x,y, color);
14        if((int)x<rect.left()) rect.setLeft(x);
15        if((int)x>rect.right()) rect.setRight(x);
16        if((int)y<rect.top()) rect.setTop(y);
17        if((int)y>rect.bottom()) rect.setBottom(y);
18        f.push(make_pair(x+1,y));
19        if(x>0)
20            f.push(make_pair(x-1,y));
21        f.push(make_pair(x,y+1));
22        if(y>0)
23            f.push(make_pair(x,y-1));
24    }
25    if(rect.height()>(3 * rect.width())/2)
26        rect.setHeight(3*rect.width()/2);
27    return rect;
28 }
```

Obr. 5.4: Zdrojový kód funkcie ImageProcessor::segment()

5.6 Rozpoznanie gesta

Pokiaľ sme v aktuálnom frame našli ruku, pridáme jej pozíciu do postupnosti a skontrolujeme, či postupnosť netvorí nejaké gesto. Pre každé gesto máme jednoduchú triedu, ktorá ho rozpoznáva. Každá takáto trieda dedí od triedy *Gesture*. Správu gest zastrešuje trieda *GestureRecognizer*

5.6.1 GestureRecognizer

Trieda umožňuje pridávať gestá, pridávať body do gesta, resetovať gesto, zistiť ktoré z pridaných gest zodpovedá postupnosti a vymazať šum z postupnosti gesta.

Zistenie gesta prebieha tak, že sa postupne postupnosť predloží každému rozpoznávaču, až kým ho niektorý z nich nerozpozna (Obr. 5.5).

Odstránenie šumu odstráni body, ktoré sú príliš ďaleko od predošlého bodu (Obr. 5.6).

Dôležité metódy

- `void addGesture(Gesture * g)`
Pridá gesto do zoznamu rozpoznávaných gest.
Vstup: *g* - pointer na rozpoznávač gesta
- `void addPoint(QPoint p)`
Pridá bod do postupnosti
Vstup: *p* - bod
- `void resetGesture()`
Zmaže všetky body z postupnosti
- `void removeNoise()`
Zmaže body, ktoré sa vychylujú od ostatných
- `Gesture * getGesture()`
Vráti pointer na rozpoznávač gesta, ktorý rozpoznal dané gesto
Výstup: Pointer na rozpoznávač gesta, alebo *NULL*, ak nebolo rozpoznané žiadne gesto.

```
1 Gesture * GestureRecognizer::getGesture()
2 {
3     removeNoise();
4     for(vector<Gesture *>::iterator it = gestures.begin(); it != gestures.
5         end(); it++)
6     {
7         if((*it)->check(points)) return (*it);
8     }
9     return NULL;
10 }
```

Obr. 5.5: Zdrojový kód funkcie GestureRecognizer::getGesture()

```
1 void GestureRecognizer::removeNoise()
2 {
3     if(points.size() < 3) return;
4     unsigned f = 1, a = 1;
5     while(f < points.size() && a < points.size())
6     {
7         if(point_subtract(points[f-1], points[a]) < MAX_POINT_DISTANCE)
8         {
9             points[f] = points[a];
10            f++;
11        }
12        a++;
13    }
14    points.resize(f);
15 }
```

Obr. 5.6: Zdrojový kód funkcie GestureRecognizer::removeNoise()

5.6.2 Gesture

Dôležité metódy

- `bool check(vector<QPoint>)`
Zistí či daná postupnosť bodov zodpovedá gestu.
Vstup: vector bodov.
Výstup: *true* ak postupnosť zodpovedá gestu, ináč *false*
- `void action()`
Vykoná akciu gesta(simulácia klávesy)

5.7 Simulácia stlačenia klávesu

Knižnica *Xtst* poskytuje funkciu `XTestFakeKeyEvent`, ktorá nám umožňuje simulovať stlačenie klávesu. Do nej vložíme pointer na display, kód klávesu, či je stlačený, alebo nie (funkciu treba použiť 2x - raz na stlačenie a raz na pustenie klávesu) a aktuálny čas.

5.8 Pomocné programy

Upravená verzia aplikácie

Upravená verzia aplikácie je kópiou aplikácie, ktorá umožňuje jednoduché ukladanie dát na disk.

Aby sme dostali čo najrealistickejšie obrázky, potrebovali sme, aby aplikácia bežala takmer tak rýchlo ako pôvodná. Zápis na disk je však časovo náročná operácia, preto sme si obrázky ukladali do buffera a zapisovali dávkovo. Zapisovali sme čo najmenšie množstvo dát, preto sme dáta Fourierových transformácií⁵ vytvárali až dodatočne ďalšou aplikáciou.

CreateFFT

CreateFFT je program, ktorý umožňuje vytvorenie Fourierovej transformácie z obrázka. Ako parameter berie cestu k obrázku a vyrobí 2 súbory – normalizovanú Fourierovu transformáciu vhodnú pre neurónové siete a obrázok Fourierovej transformácie vhodný pre posúdenie človekom.

Použitie: `./CreateFFT image.ppm`

⁵kapitola 3.1.4

NeuralNet

NeuralNet je tréningová aplikácia pre neurónové siete. Využíva rovnaké triedy pre neurónové siete ako hlavná aplikácia. Obsahuje ľahko upraviteľný algoritmus tréningu a funkcie na jednoduché načítanie dát.

Dáta máme v 2 oddelených adresároch. V jednom sú dáta zodpovedajúce rukám a v druhom ostatné. Programu poskytneme tieto adresáre a on si z nich vyrobí tréningovú/testovaciu sadu.

Pri rekurentnej neurónovej sieti je dôležitý aj názov súboru, z ktorého sa získa informácia do ktorej postupnosti a framu prvok patrí.

Použitie: `./NeuralNet architektura pocet_epoch vypisy typ_dat cesta_ruky cesta_ostatne vahy_vstup vahy_vystup`

- architektura n – dopredná, r – rekurentná, c – z konzoly
- pocet_epoch - počet epoch alebo 0 pre testovanie
- vypisy - miera výpisov do konzoly
- typ_dat - 0 – Fourierove transformácie, 1 – obrázky s čiernym pozadím, 3 - obrázky s bielym pozadím
- cesta_ruky - cesta k dátam rúk
- cesta_ostatne - cesta k ostatným dátam
- vahy_vstup - súbor z ktorého sa načítajú váhy
- vahy_vystup - súbor do ktorého sa uložia váhy

5.9 Problémy a ich riešenia

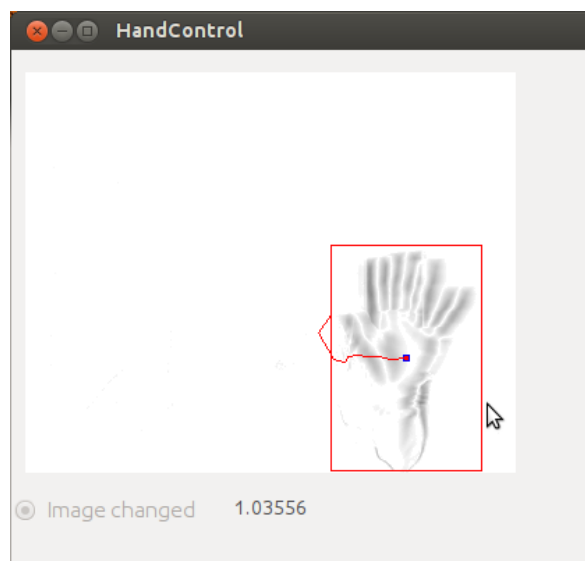
5.9.1 Multivláknovosť

Použitie viacerých vlákien prinieslo problémy v prípade, že viac vlákien pristupovalo naraz k tomu istému zdroju. Problémom bola napríklad fronta, do ktorej sa ukladali segmenty. Jedno vlákno pridávalo do fronty a ostatné vyberali. Takéto problémy sme vyriešili vhodným umiestnením mutexov. Problémy robila aj knižnica *fftw3*, kde alokovanie a uvoľnenie pamäti bolo tiež treba ošetriť mutexami.

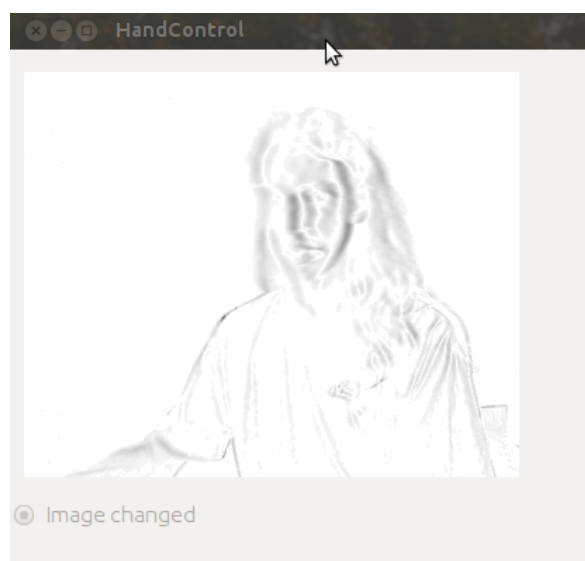
5.9.2 Viacnásobné dedenie

Pri implementácii neurónových sietí sme sa stretli s problémom diamantového dedenia, keď jedna trieda dedí od 2 tried ktoré majú spoločného predka. Spoločného predka sme museli pri dedení označiť ako `virtual` a pretypovávať sme museli pomocou `dynamic_cast`.

5.10 Screenshoty aplikácie



Obr. 5.7: Správne rozpoznaná ruka a jej predchádzajúca trajektória



Obr. 5.8: Správne rozpoznaný iný objekt

Záver

V práci sme popísali návrh aplikácie pre ovládanie počítača pomocou gest ruky. Popísali sme si algoritmus rozpoznávania gesta a podrobnejšie sme sa zaoberali jeho jednotlivými časťami. Stručne sme si vysvetlili, čo sú umelé neurónové siete a ako ich použijeme na rozpoznanie ruky v obraze. Zaoberali sme sa aj technickými časťami aplikácie, ako je preklad gesta na stlačenie klávesy a získavanie obrazu z webovej kamery.

Navrhli sme vhodnú architektúru umelej neurónovej siete na rozpoznanie ruky a vyskúšali sme rôzne metódy predspracovania. Porovnali sme úspešnosti jednotlivých prístupov a vybrali sme tie, ktoré sa ukázali byť nápomocné. Nakoniec sme si popísali implementáciu aplikácie a rozobrali sme niektoré problémy, ktoré sa vyskytli pri jej implementácii.

Podarilo sa nám ukázať, že Fourierova transformácia pomáha na zvýšenie úspešnosti, a že rozdielový obraz, ktorý vznikne rozdielom dvoch po sebe idúcich obrázkov a obsahuje obrysy pohybujúcich sa objektov je vhodným zdrojom informácie pre neurónovú sieť. Ukázali sme, že umelé neurónové siete sa dajú použiť na úlohu rozpoznávania ruky. Natrénovali sme neurónovú sieť, ktorá má úspešnosť 67,88 % na testovacej vzorke a otestovali sme aplikáciu v praxi. Aplikácia je v praxi použiteľná, rozpoznávanie gest dosahuje úspešnosť približne 80%.

Momentálne aplikácia rozpoznáva 4 základné gestá – pohyb rukou doprava, doľava, hore a dole. Snažili sme sa ju zoptimalizovať a na bežných systémoch funguje plynule a bez problémov. Oproti komerčným riešeniam, ktoré sa dodávajú aj so špecifickým hardvérom má aplikácia nevýhodu v nízkej kvalite obrazu z kamery a to, že kamera je len jedna a obraz je dvojrozmerný. Komerčné riešenia majú často 2 kamery a senzor hĺbky, vďaka čomu majú verný 3D obraz scény. Aj napriek tomu však naša aplikácia funguje v bežných svetelných podmienkach celkom dobre.

Aplikácia poskytuje ešte mnoho priestoru na vylepšenie. Dal by sa použiť napríklad *Kalmanov filter* na zvýšenie presnosti gesta a elimináciu zlých rozhodnutí siete, zlepšiť rozpoznávanie gest a pridať ďalšie, aj komplexnejšie gestá. Rozpoznávanie sa dá napojiť aj na kvalitnejší algoritmus sledovania objektov. V budúcnosti by sme chceli vyskúšať aj použitie *Skrytých Markovových modelov* namiesto neurónových sietí.

Literatúra

- [Fed11] Dominika Fedáková. Aplikované využitie neurónových sietí, 2011.
- [Hay99] S. Haykin. *Neural Networks: A Comprehensive Foundation*. IEEE, 1999.
- [kap12] Kapture.
<http://kapture.berlios.de/>, 2012. [Online; accessed 13-May-2012].
- [KBP⁺97] V. Kvasnička, L'. Beňušková, J. Pospíchal, I. Farkaš, P. Tiňo, and A. Král'.
Úvod do teórie neurónových sietí. Iris, 1997.
- [Nok12] Nokia. *Documentation / Qt Developer Network*, 2012.
<http://qt-project.org/doc/>[Online; accessed 30-May-2012].
- [Per09] Peter Perešíni. Fourierova transformácia a jej použitie, 2009.
- [RFW12] Ashley Walker Robert Fisher, Simon Perkins and Erik Wolfart. Fourier Transform.
<http://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm>, 2012. [Online; accessed 30-May-2012].
- [Wik12a] Wikipedia. Bilinear interpolation.
http://en.wikipedia.org/wiki/Bilinear_interpolation, 2012. [Online; accessed 30-May-2012].
- [Wik12b] Wikipedia. Breadth-first search.
http://en.wikipedia.org/wiki/Breadth-first_search, 2012. [Online; accessed 30-May-2012].
- [Wik12c] Wikipedia. Fourier transform.
http://en.wikipedia.org/wiki/Fourier_transform, 2012. [Online; accessed 30-May-2012].