Univerzita Komenského, Bratislava
Fakulta Matematiky, Fyziky a Informatiky

Ovládanie počítačových aplikácií pomocou gest ruky

Bakalárska práca

2012

Michal Hozza

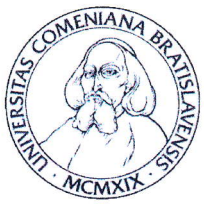
Univerzita Komenského, Bratislava
Fakulta Matematiky, Fyziky a Informatiky

Ovládanie počítačových aplikácií pomocou gest ruky

Bakalárska práca

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

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 RNDr. Marekovi Nagyovi za cenné rady a pripomienky, priateľke, blízkym priateľom a rodine za morálnu podporu.

Michal Hozza

Abstrakt

Tu bude abstrakt TODO!!!

Kľúčové slová: neuronove siete, umela inteligencia, počítačové videnie, rozpoznávanie gest

Abstract

Tu bude anglicka verzia abstraktu - vznikne az ked budem mat finalnu slovensku verziu TODO!!!

Key words: neural networks, AI, computer vision, gesture recognition

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Notebooky sú často využívané na prezentácie, či už na nejakej prednáške, alebo prezentácie fotiek. Pri prezentovaní 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 vzniká 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 prácu[1] sa takéto riešenia v praxi nevyužívali, hoci niečim 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 ukazuje, že táto myšlienka nie je zlá.

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

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

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

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

[1] v 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 sú dostupné otvorené ovládače a vynikajúca podpora. S perifériami sa pracuje v Linuxe veľmi pohodlne, každé zariadenie má vlastný súbor, ktorý sa nachádza v `/dev/`.

Keďže Linux je písaný hlavne v jazyku C, prevažná väčšina knižníc pre Linux sa dodáva aj s hlavičkovými súbormi pre jazyk C a C++. Preto je jazyk C++ veľmi vhodný pre programovanie pre túto platformu. Okrem toho má vynikajúcu podporu, kvalitný kompilátor a je objektovo orientovaný, čo umožňuje jednoduché použitie komplexných dátových štruktúr. Vďaka vysokej podpore na rôznych platformách, nie je problém po prispôsobení niektorých častí aplikácie, preportovať 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 vláken 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 kláves knižnicu *Xtst*, ktorá umožňuje simuláciu udalostí v X-serveri.

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

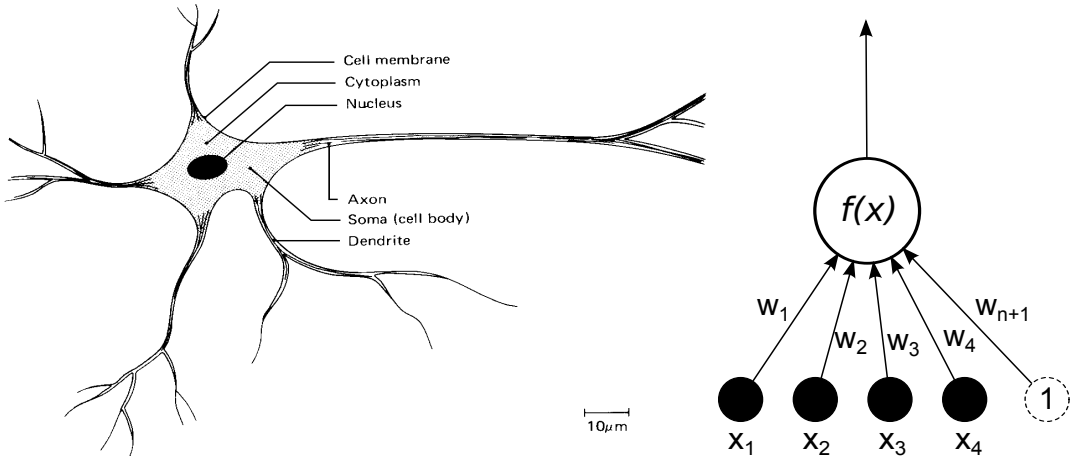
[1] video for linux 2

Kapitola 2

Neurónové siete

Keďže *umelé neurónové siete* sú významnou časťou tejto práce, venujeme im samostatnú kapitolu. Popíšeme čo sú neurónové siete, ako fungujú a základné typy neurónových sietí, ktoré používame v aplikácií.

2.1 Jednoduchý spojitý perceptrón



(a) Prírodný neurón (b) Umelý neurón

Obr. 2.1: Prírodný vs. umelý neurón

Jednoduchý perceptrón je inšpirovaný nervovou bunkou - neurónom. Vstupy umelého neurónu zodpovedajú *dendridom*, 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 → neurón – lineárna input-output väzba, kde pôvodný signál x_i prenásobíme váhou synapsy w_i a tým dostaneme výsledný signál x_i'.

2. Aktivačné – neurón → výstup – nelineárna input-output väzba, kde y dostaneme dosadením $\sum x_i'$ do aktivačnej funkcie.

K vstupom treba pridať ešte jeden vstup – *bias* – ktorý má vždy hodnotu 1. Jeho význam je pri vstupe so samými nulami, pretože vtedy hodnoty na synapsách ostanú nezmenené a perceptrón by sa tento vzor nevedel naučiť.

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

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

Aktivačnou funkciou spojitého je sigmoida:

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

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

2.2 Jednovrstvová neurónová sieť

Niekoľko perceptrónov vieme spojiť 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 môžme vrstvy pospájať.

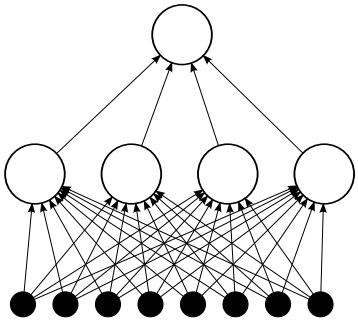
Neuróny vo vrstve nie sú nijako pospájané a každý teda operuje nezávisle, takže z praktického hľadiska je to skupina neurónov pracujúcich nad tým istým vstupom.

Vrstva robí zobrazenie $\mathbb{R}^n \to (0, 1)^m$, 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.

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.



Obr. 2.2: Dopredná neurónová sieť

Viacvrstvové neurónové siete sa trénujeme algoritmom backpropagation[1] [Hay99, Fed11].

[1] algoritmus spätného šírenia chyby

Kapitola 3

Návrh

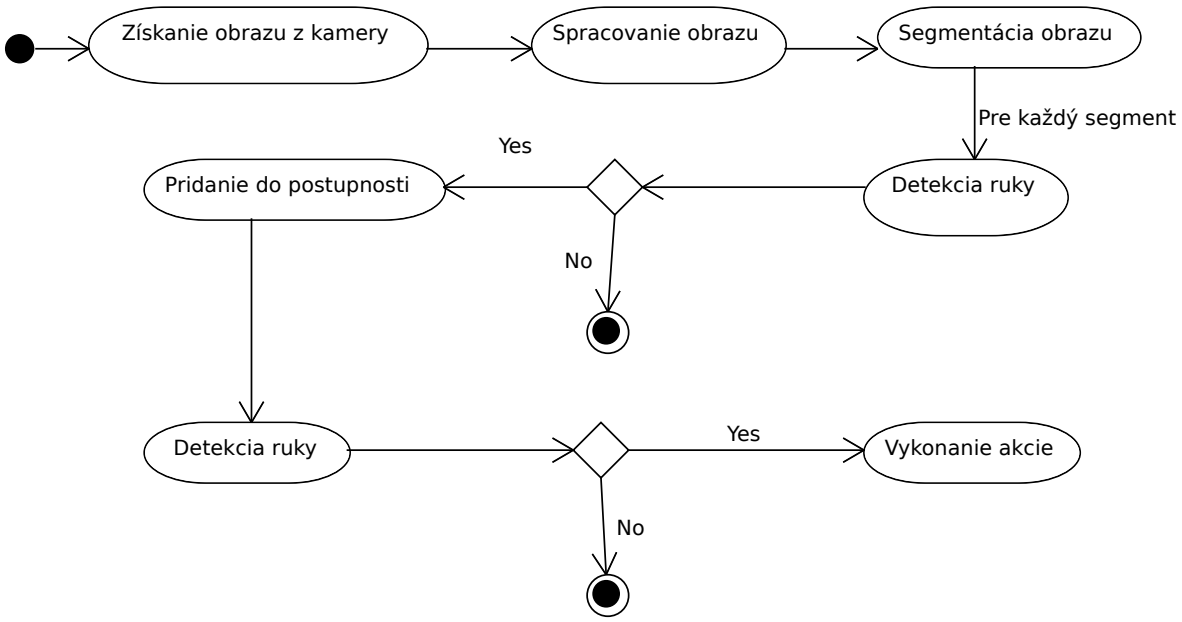
V tejto kapitole predstavíme základný algoritmus a postupne sa budeme venovať jeho jednotlivým častiam. Budeme sa venovať metódam na zlepšenie úspešnosti klasifikátora a porovnáme rôzne prístupy.

3.1 Základný algoritmus

Základný algoritmus (Obr. 3.1) vyzerá nasledovne: Najskôr sa získa obraz z webkamery. Ten sa potom spracuje a následne sa rozsegmentuje na jednotlivé pohybujúce sa objekty. Každý z týchto segmentov sa predloží klasifikátoru, ktorý rozhodne, či daný segment je, alebo nie je ruka. Zo všetkých segmentov je vybratý ten, o ktorom si je klasifikátor najviac istý, že je ruka (a zároveň spĺňa danú hranicu). Z vybratého segmentu sa vypočíta bod, ktorý je braný ako pozícia ruky. Tento bod je pridaný do postupnosti bodov, o ktorých sa ďalej rozhodne, či tvoria niektoré gesto. Ak klasifikátor gesta detekuje nejaké gesto, vykoná sa príslušná akcia - simulácia stlačenia niektorej klávesy - a postupnosť sa vymaže. Ak sa dlhšiu dobu v obraze nevykonala žiadna zmena a klasifikátor gesta nezistil žiadne gesto, postupnosť sa tiež vymaže.

3.1.1 Rozdielový obraz

Základ pre predspracovanie a segmentáciu tvorí takzvaný **rozdielový obraz**. 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 teda nevyskytnú, čo nám umožní ich veľmi ľahko odfiltrovať. V tomto obraze sa vyskytnú obrysy pohybujúcich sa objektov, pretože ku zmenám dochádza najviac na hranách. Podľa rýchlosti pohybu môžu byť obrysy hrubšie, alebo tenšie.



Obr. 3.1: Základný algoritmus

3.1.2 Predspracovanie vstupného obrazu

Odfiltrovanie šumu

Odfiltrovanie šumu zabezpečí hranica – *treshold*. 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 hodnoty $0 \ldots 255$. Praktické testy ukázali, že vhodnou hodnotou pre hranicu je 7.

Rozpitie pixlov

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

Z každého pixla spravíme štvorec s veľkosťou 11×11. Hodnota 11 pre stranu štvorca sa ukázala 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

Rozpitý 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.

Potrebujeme ešte nájsť opísaný obdĺžnik. To spravíme 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 povedala, či je to ruka, alebo nie.

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

Normalizácia dát

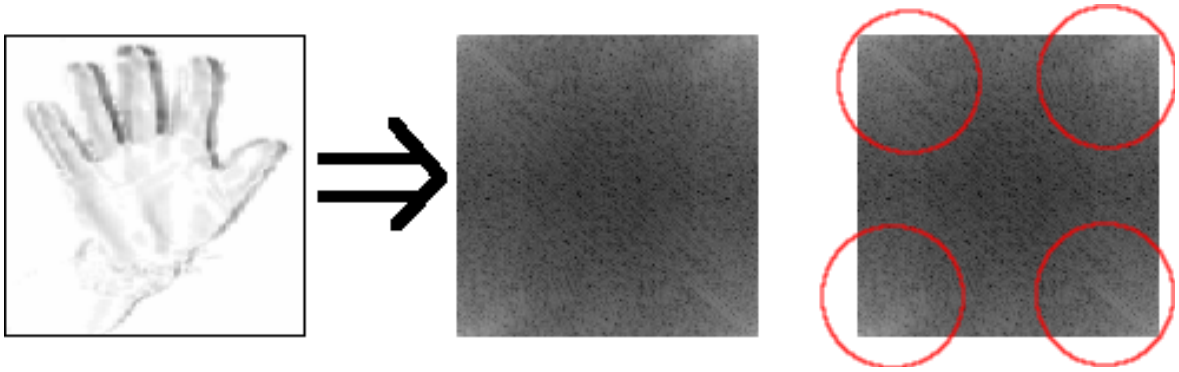
Ideálne vstupy pre neurónovú sieť sú z intervalu $\langle 0, 1\rangle$. Pixle čiernobielych obrázkov majú hodnoty $\{0 \ldots 255\}$. Pri obrázkoch zvolíme pre farbu pozadia hodnotu 0 a pre objekt ostatné hodnoty. Z tohto dôvodu chceme, aby rozdiel v normalizovanej hodnote medzi 0 a 1 bol najväčší a 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 trénovaní 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).



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

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

Použitie pôvodného namiesto rozdielového obrázka

Nevýhodou rozdielového obrázka je, že zmena spôsobená pohybom sa v ňom vyskytne dvakrát. Raz na mieste, kam sa objekt posunul a raz na mieste odkiaľ sa posunul. Navyše je táto zmena závislá od rýchlosti pohybu ruky, čo nie je celkom ideálne pre neurónovú sieť, a zvyšuje nároky na veľkosť trénovacej množiny. Toto sme chceli eliminovať tak, že sa vyberie ruka z pôvodného obrázka podľa farby. Táto ruka tam bude vždy len raz. Bohužiaľ 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. Navyše 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.

Ďalší problém je správne tipnúť bod, ktorý patrí ruke, aby sa z neho mohla odštartovať selekcia. Pokiaľ by bola v danom obdĺžniku len dlaň, tak nie je až také ťažké sa správne trafiť - je takmer isté, že kúsok pod stredom obrázka bude dlaň. Bohužiaľ často sa stane, že užívateľ pohne nielen rukou, ale aj predlaktím a segmentačný algoritmus zaradí do segmentu aj predlaktie. Potom sa môže stať, že bod ruky netrafíme a algoritmus nemá šancu ruku vyselektovať.

Takto 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 si popíšeme rôzne architektúry sietí, ktoré sme vyskúšali a porovnáme ich vlastnosti a úspešnosť pri riešení problému rozpoznania ruky a vyberieme vhodnú architektúru, ktorú potom použijeme v našej aplikácii.

3.2.1 Požiadavky na architektúru neurónovej siete

Našim cieľom je vytvoriť vhodnú architektúru neurónovej siete, ktorá bude rozhodovať o danom vstupe, či zodpovedá ruke alebo nie. Navrhneme niekoľko typov architektúr,

ktoré neskôr porovnáme (kapitola **??**) a vyberieme najvhodnejšiu z nich, ktorú potom použijeme v aplikácií.

Neurónová sieť má rozdeliť vstupy do 2 tried - tie, ktoré zodpovedajú rukám a ostatné. Na to využijeme vo všetkých architektúrach jeden výstupný neurón.

Budeme sa snažiť dosiahnuť čo najlepšiu úspešnosť a čo najvyššiu rýchlosť, čiže najmenšiu zložitosť[1] siete.

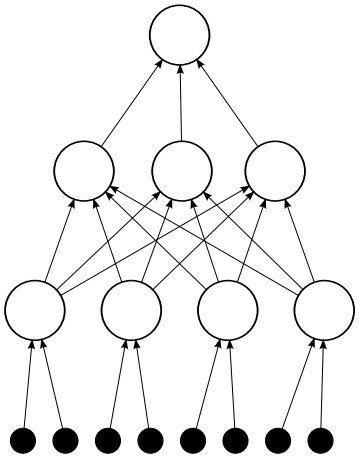
3.2.2 Typ 1: Viac vrstvová dopredná neurónová sieť

Viac vrstvová dopredná neurónová sieť (obr. 2.2) je implementáciu klasického viacvrstvového perceptrónu, ktorý sme popísali v kapitole 2.3. Je zložená z viacerých vrstiev neurónov, pričom signál sa šíri len zo spodnejšej vrstvy na vyššiu.

Ako vstup každého neurónu berú výstupy neurónov z predošlej vrstvy. Prvá vrstva dostane na vstup vstup siete.

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

3.2.3 Typ 2: Upravená verzia viac vrstvovej doprednej neurónovej siete



Obr. 3.3: Upravená dopredná neurónová sieť

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).

[1] vzhľadom na počet váh

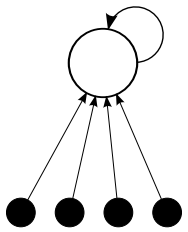
Jeho výhodou je zvýšenie rýchlosti pri rovnakom počte neurónov - zníži sa počet váh vstupných neurónov. Náš vstup má rozmer $128 \times 128 = 16384$, čo nie je malé číslo. Keď ho rozdelíme na 16 častí, jednotlivé časti budú mať veľkosť $32 \times 32 = 1024$. Takto namiesto toho, aby každý vstupný neurón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

Nevýhodou je to, že neuróny sú fixne pridelené na jednotlivé vstupy. V pôvodnej sieti si neuróny sami vyberali, ktoré časti vstupu sú pre nich najvýznamnejšie a mohli tak lepšie pokryť vstup. Môžme to však vykompenzovať miernym zvýšením počtu neurónov aby sme tak 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 šírila 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).



Obr. 3.4: Rekurentný neurón

Rekurentný neurón obsahuje navyše spätnú väzbu. Spätná väzba sa tvári ako ďalší vstup a obsahuje posledný updatenutý výstup toho istého neurónu. Po aktivácii neurónu môžeme updatnuť poslednú aktivačnú hodnotu. Ak to neurobíme, hodnota ostane taká, ako bola predtým. Neurón môžeme aj resetnúť, vtedy sa hodnota vynuluje.

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

Našu jednoduchú rekurentnú neurónovú sieť trénujeme tiež algoritmom *backpropagation* s tým, že sieti predkladáme postupnosti - vrámci nej sú dáta vždy v tom istom poradí. Pri kladnej odozve updatujeme rekurentný vstup, ináč nie, čo nám umožní odfiltrovať zlé obrázky z postupnosti, a nestratíme informáciu o tom ako sieť reagovala na poslednú ruku. Ruky v rámci postupnosti sa väčšinou od seba navzájom líšia len málo. Po každej postupnosti zresetujeme rekurentný vstup na 0, čím sa sieť dostane do východzieho stavu, kedy by mala reagovať na novú ruku - 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 aj predspracovaniu segmentov, ktoré priamo ovplyvňuje vlastnosti trénovania a úspešnosť sietí.

4.1 Trénovanie

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 updatli len v prípade, že sme rozpoznali ruku. Po každej postupnosti sme rekurentné vstupy resetli na 0. Takto simulujeme správanie siete v reálnej aplikácii.

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:

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

t je cieľová hodnota a o je výstup siete. V tabuľkách 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.

Trénovacie dáta sme rozdelili na dve sady. Každá sada bola rozdelená na dve disjunktné množiny – trénovaciu a testovaciu.

Prvou sadou 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 obrázky[1] a k nim zodpovedajúce fourierove transformácie.

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

	Testovacia množina		**Trénovacia množina**	
Sada	**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: Viac vrstvová dopredná neurónová sieť

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

[1] kapitola 3.1.1

	Testovacia množina		Trénovacia množina	
Počet neurónov	**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 viac vrstvovej doprednej neurónovej siete

Pri tomto type siete sa nám podarilo natrénovať 2 aj 3-vrstvové architektúry, v tabuľke 4.3 sa budú teda vyskytovať obe. Vrstvy sú oddelené „;" a 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 trénovania - stačila asi 1/4 epoch oproti 2 vrstvám. Keďže ale v praxi je pridanie ďalšej vrstvy spomalením, tak dve vrstvy budú v tomto prípade lepšie.

Oproti sieťam typu 1, vidíme určitý nárast úspešností približne o 2%, 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.

	Testovacia množina		Trénovacia množina	
Počet neurónov	**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

	Testovacia množina		Trénovacia množina	
Rekurentná vrstva	**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

	Testovacia množina		Trénovacia množina	
Počet neurónov	**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 zdvihnúť úspešnosť o **2,81%** 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 zvýšiť 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 predspracovania

4.5.1 Vplyv Fourierovej transformácie

Pre porovnanie fourierovej transformácie a rozdielových obrázkov sme použili 3-vrstvovú upravenú neurónovú sieť. Vybrali sme ju pre to, že 2-vrstvovú sa nám nepodarilo natrénovať. Ani pri tejto architektúre sa nám však nedarili sieť natrénovať na úspešnosť viac ako 71%. Už z toho je vidieť výhoda fourierovej transformácie. Počty neurónov sme vybrali tie, 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% pri použití Fourierovej transformácie. Okrem toho nám Fourierova transformácia dáva potenciál zdvihnúť úspešnosť pri väčšom množstve trénovacích dát a širšie možnosti trénovania.

	Testovacia množina		Trénovacia množina	
Typ dát	**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 trénovanie a testovanie *sadu 2* (kapitola 4.3) v ktorej boli trénovacie 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.

	Testovacia množina		Trénovacia množina	
Typ dát	**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 Fourierovych transformácií

Z tabuľky 4.7 je vidieť, že úspešnosť siete na dátach pôvodného obrázka 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 nie len pri trénovaní, ale aj pri celkovej úspešnosti neurónovej siete. Sieť je možné rýchlejšie a lepšie natrénovať a trénovať na väčších dátach.

Dáta získané na základe pôvodného obrázka 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 obrázka. 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 spacovania 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 vrstvový model. Základnou jednotkou je spojitý 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

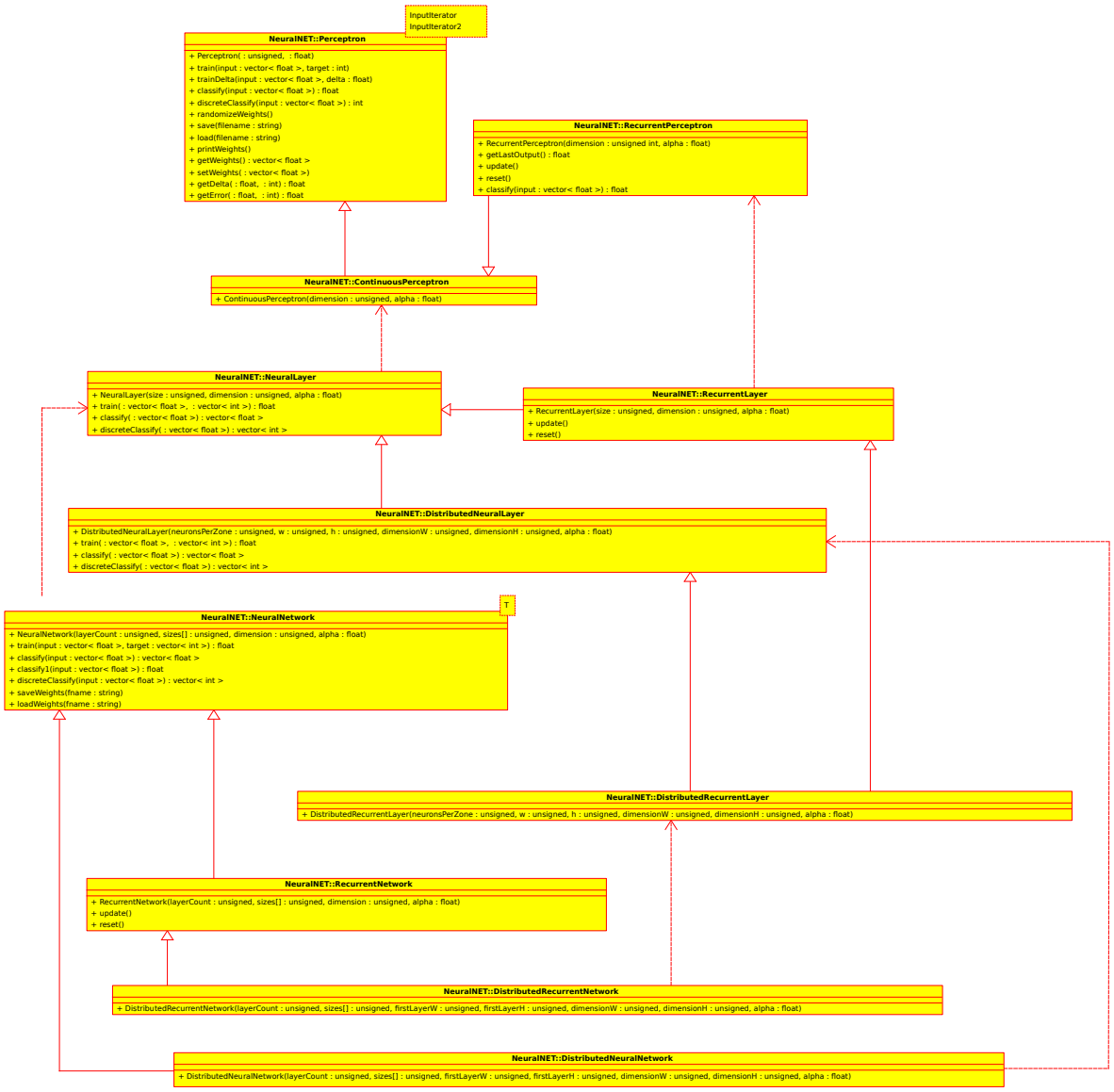
Perceptron je abstraktná trieda, ktorá zovšeobecňuje rôzne typy perceptrónov, ktoré sa líšia rôznymi aktivačnými funkciami.

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

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

Možnosť vytvorenia rôznych typov perceptrónov zabezpečujú abstraktné metódy

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



Obr. 5.1: Neurónové siete - Class diagram

- `float derivativeFunction(float x)`,

ktoré treba definovať.

Dôležité metódy

- `void train(vector<float> input,int target)`
  Jednoduché trénovanie perceptrónov.
  Vstup: *input* - vstup siete, *target* - požadovaný výstup

- `void trainDelta(vector<float> input, float delta)`
  Delta trénovanie pre *backpropagation* algoritmus.
  Vstup: *input* - vstup siete, *delta* $= \delta$ z učiaceho pravidla: $\Delta w_i = \alpha \delta x_i$[1] [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

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

5.1.3 RecurrentPerceptron

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

[1] používame mierne inú notáciu ako je v [Hay99]

5.1.4 NeuralLayer

NeuralLayer je trieda, ktorá zoskupuje spojité perceptróny do vrstvy. Umožňuje trénovanie na viacrozmerné 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 funkcionalitu 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 umožnujú update a reset 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 konštruktory 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 up-

datujú 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 konverzia do rôznych formátov, ukladanie na disk a iné.

Trieda *HCImage*, je abstraktná trieda ktorá implementuje základné veci, ktoré sú nezávislé na type pixelu. Od tejto triedy potom dedia triedy *GrayScaleImage* a *ColorImage*. Tieto implementujú metódy, ktoré sú závislé na type pixla.

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 bilineárnu 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"[2], alebo nástroj na selekciu susedných pixlov s podobnými farbami. Techniku *floodfill* používame na získanie masky v metóde `getAdaptiveFloodFillSelectionMask(int sx, int sy, int treshold, float originalFactor, float changeFactor)` (Obr. 5.2). Metóda berie nekoľko 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 odstráni[3] pixle, ktoré neboli vybraté pomocou techniky *floodfill*.

Algortimus funguje ako prehľadávanie do šírky, pričom do susedného pixla 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 pixla 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 pixla.

5.3 Získanie obrazu z webkamery

Na získanie obrazu z webkamery používam triedu, ktorá pochádza z programu *Kapture* [kap] 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, zobrať pretransformovať obrázok do niektorej z tried na obrázky. JPEG kóduje farby vo formáte YUV, tieto teda musíme pretransformovať do RGB. V prípade čiernobieleho obrázka iba zoberieme zložku Y.

[2] známu z grafických programov ako je GIMP, či Photoshop

[3] začierni

```
template <class T>
HCImage<T>* HCImage<T>::getAdaptiveFloodFillSelectionMask(unsigned sx,
    unsigned sy, int treshold, float originalFactor, float changeFactor)
{
  T reference = getAverageColor(sx,sy);
  ImageBuffer b;
  b.resize(width()*height(),0);
  queue<pair<pair<unsigned,unsigned>,T> > f;
  f.push(make_pair(make_pair(sx,sy),reference));
  while(!f.empty())
  {
      unsigned x = f.front().first.first;
      unsigned y = f.front().first.second;
      T refcolor = f.front().second;
      f.pop();
      if(x>=width()) continue;
      if(y>=height()) continue;

      T color = pixel(x,y);
      if(!similar(refcolor,color,treshold) || b[x+y*w]!=T(0)) continue;
      b[x+y*w]=0xffffffff;
      refcolor = reference*originalFactor + (refcolor*changeFactor+color
          *(1-changeFactor))*(1-originalFactor);

      f.push(make_pair(make_pair(x+1,y),refcolor));
      f.push(make_pair(make_pair(x-1,y),refcolor));
      f.push(make_pair(make_pair(x,y+1),refcolor));
      f.push(make_pair(make_pair(x,y-1),refcolor));
  }
  HCImage<T> * maskImage = create(b,width(),height());
  return maskImage;
}
```

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

5.4 Spracovanie obrazu

5.4.1 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. Hlavná metóda je metóda `processImage (const GrayScaleImage &image, const ColorImage &colorimg)`, ktorá spravuje vlákna a spúšťa jednotlivé kroky.

Spúštanie vláken 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.

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 rozdieli toho bude veľa, ale je nízka pravdepodobnosť, že to bude zle vyhodnotené. V prípade zlého vyhodnotenia ale 1 obrázok stále nestačí na gesto.

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. 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 čiernu posunieme na hodnotu o k väčšiu ako je y súradnica pixla ktorý kreslíme. Pokiaľ sme pred hranicou, kreslíme čiernu, inak bielu. Tento algoritmus má zložitosť $O(w.h)$ v praxi pre $k = 11$ asi $5\times$ rýchlejší.

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

Pre Y-ový smer je to analogické.

```
void ImageProcessor::expandPixelsX(int sy, int ex, int ey, GrayScaleImage
    * imgIn, GrayScaleImage * imgOut)
{
  int sx = 0;
  for(int y = sy;y<ey;y++)
  {
    int endblack = 0;
    for(int x = sx;x<sx+PIXEL_RADIUS && x<ex;x++)
    {
      uchar c = imgIn->pixel(x,y);
      if(c!=WHITE)
      {
        endblack = x+PIXEL_RADIUS;
      }
    }
    for(int x = sx;x<ex;x++)
    {
      int xx = x+PIXEL_RADIUS;
      if(xx<ex)
      {
        uchar c = imgIn->pixel(xx,y);
        if(c!=WHITE)
        {
          endblack = xx+PIXEL_RADIUS;
        }
      }
      if(x<endblack)
      {
         imgOut->setPixel(x,y,0);
      }
      else
      {
         imgOut->setPixel(x,y,0xFF);
      }
    }
  }
}
```

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

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 ešte 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.

```
QRect ImageProcessor::segment(unsigned sx, unsigned sy, uchar color,
    GrayScaleImage * image, QRect rect)
{
  queue<pair<unsigned,unsigned> > f;
  f.push(make_pair(sx,sy));
  while(!f.empty())
  {
       unsigned x = f.front().first;
       unsigned y = f.front().second;
       f.pop();
       if(x>=image->width()) continue;
       if(y>=image->height()) continue;
       if(image->pixel(x,y)!=0) continue;
       image->setPixel(x,y,color);
       if((int)x<rect.left()) rect.setLeft(x);
       if((int)x>rect.right()) rect.setRight(x);
       if((int)y<rect.top()) rect.setTop(y);
       if((int)y>rect.bottom()) rect.setBottom(y);
       f.push(make_pair(x+1,y));
       if(x>0)
         f.push(make_pair(x-1,y));
       f.push(make_pair(x,y+1));
       if(y>0)
         f.push(make_pair(x,y-1));
  }
  if(rect.height()>(3*rect.width())/2)
    rect.setHeight(3*rect.width()/2);
  return rect;
}
```

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

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 sieti, ktorá vráti hodnotu $> 0,5$ ak to rozpoznala ako ruku, ináč vráti hodnotu $\leq 0,5$.

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 nerozpozná (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 vychyľujú 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.
        end();it++)
5   {
6     if((*it)->check(points)) return (*it);
7   }
8   return NULL;
9 }
```

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 kláves

Knižnica *Xtst* poskytuje funkciu `XTestFakeKeyEvent`, ktorá nám umožnuje simulovať stlačenie klávesy. Do nej vložíme pointer na display, kód klávesy, či je stlačená alebo nie(funkciu treba použiť 2x - raz na stlačenie a raz na pustenie klávesy) 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 išla 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 transformácií[4] robili 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`

[4] kapitola 3.1.4

NeuralNet

NeuralNet je trénovacia aplikácia pre neurónové siete. Využíva rovnaké triedy pre neurónové siete. Obsahuje ľahko upraviteľný algoritmus trénovania a funkcie na jednoduché načítanie dát.

Dáta máme v 2 oddelených adresároch v jednom sú dáta zodpovedajúce rukám, a v druhom ostatné. Programu potom dáme tieto adresáre a on si z nich vyrobí trénovaciu/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

Skripty

Pre zjednodušenie práce sme si napísali rôzne skripty na jednoduché spúštanie trénovania, testovania a na rozdelovanie vstupov do príslušných adresárov. TODO!!!

5.9 Problémy a ich riešenia

TODO!!!

5.10 Screenshoty aplikácie

TODO!!!

Záver

Tu bude záver. TODO!!!

Literatúra

[Fed11] Dominika Fedáková. Aplikované využitie neurónových sieti, 2011.

[Hay99] S. Haykin. *Neural Networks: A Comprehensive Foundation*. IEEE, 1999.

[kap] Kapture. `http://kapture.berlios.de/`. [Online; accessed 13-May-2012].

[KBP+97] V. Kvasnička, L'. Beňušková, J. Pospíchal, I. Farkaš, P. Tiňo, and A. Král'. *Úvod do teórie neurónovỳch sietí*. Iris, 1997.

[Per09] Peter Perešíni. Fourierova transformácia a jej použitie, 2009.