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

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 web-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 web-kamerou.

V tejto práci sa budem zaoberať 2 prístupmi k rozpoznávaniu a budem porovnávať ich úspešnosť. Prvým z nich je oddelený prístup, kde budem zvlášť rozpoznávať ruku pomocou doprednej neurónovej siete a zvlášť gestá nakreslené touto rukou. V druhom budem rozpoznávať gesto z postupnosti obrazov pomocou rekurentnej neurónovej siete. Okrem toho sa budem venovať predspracovaniu obrazu rôznymi metódami tak, aby som zlepšil kvalitu rozpoznávania.

Výsledný produkt sa bude dať použiť na transformáciu gest na klávesové skratky a tým na ovládanie prezentačnej aplikácie.

Kapitola 1

Technológie

V tejto kapitole sa budeme venovať použitým technológiám, použitému jazyku a dôvodom, 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 velmi 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. Navyše je objektovo orientovaný a umožnuje jednoduché použitie komplexných dátových štruktúr.

Programovanie nám uľahčia už hotové knižnice. Využijeme knižnicu *Qt* (multiplatformová knižnica umožnujúca vytváranie okien, správu vlákien a mnohé iné), video for linux 2(*v4l2* - na prácu z webkamerou), *fftw3* (rýchla knižnica počítajúca fourierovu transformáciu) a *Xtst*(knižnica ktorá okrem iného umožnuje simulovanie kláves v X servri).

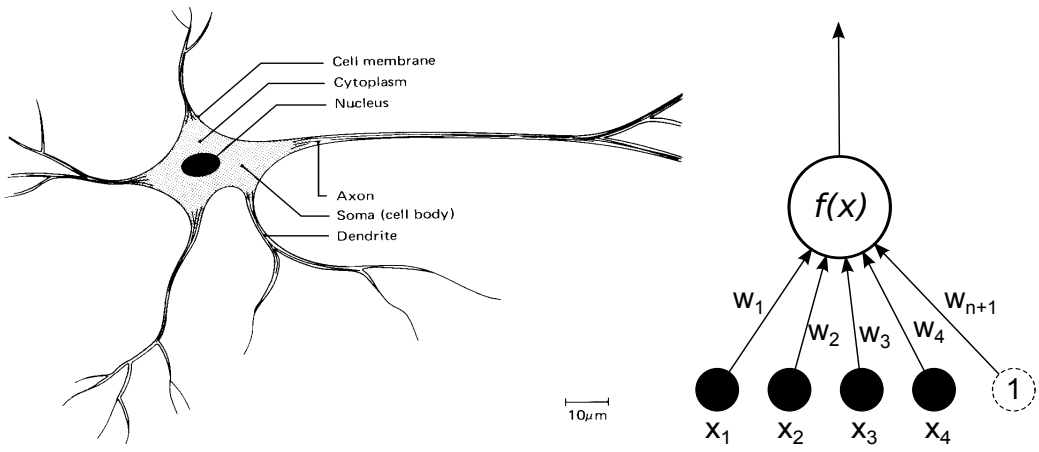
Aplikácia využíva vlákna, na paralelizáciu a urýchlenie niektorých výpočtov.

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 perceprtón



(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 Vrstva neurónovej siete

Niekoľko perceptrónov vieme spojiť do jednej vrstvy. Získame tým vačší rozmer výstupu a teda môžeme vstupy klasifikovať do viacerých tried. Hodí sa to aj pri použití vo viacvrstvových sieťach.

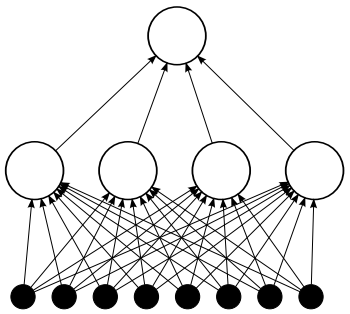
Neuróny vo vrstve nie sú nijako pospájané a každý teda operuje nezávisle, takže z praktického hľadiska je to skupina neurónov pracujúcich nad tým istým vstupom.

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.

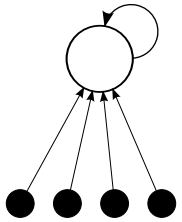


Obr. 2.2: Dopredná neurónová sieť

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

2.4 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.



Obr. 2.3: Rekurentný neurón

Existuje veľké množstvo architektúr neurónových sietí, niektoré môžete nájsť v [Hay99, s. 755]. My sme si vybrali oveľa jednoduchší model. Namiesto jednoduchých spojitých perceptrónov použijeme rekurentné neuróny (obr. 2.3).

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.

[1] algoritmus spätného šírenia chyby

Našu jednoduchú rekurentnú neurónovú sieť trénujeme tiež algoritmom back-propagation s tým, že sieti predkladáme postupnosti - vždy v tom istom poradí. Pri kladnej odozve updatujeme rekurentný vstup, ináč nie. Po každej postupnosti zresetujeme rekurentný vstup na 0.

Kapitola 3

Návrh

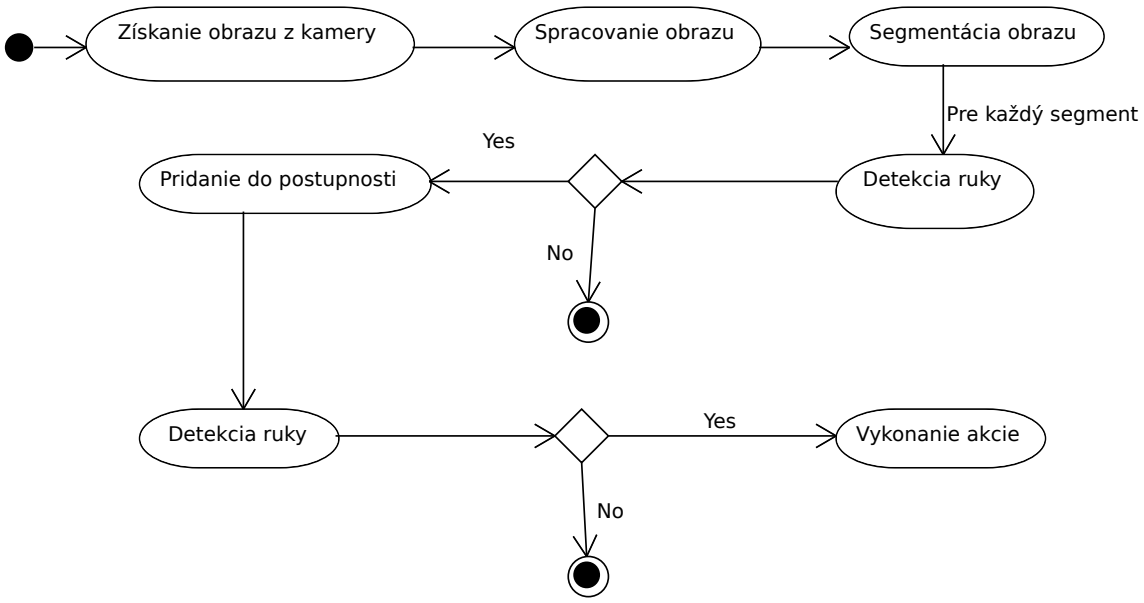
V tejto kapitole predstavíme základný algoritmus a postupne sa budeme venovať jeho jednotlivým častiam. Budeme sa venovať metódam na zlepšenie úspešnosti klasifikátora a porovnáme rôzne prístupy.

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 Predspracovanie a segmentácia

Základ pre predspracovanie a segmentáciu tvorí takzvaný **rozdielový obraz**. Rozdielový obraz je obraz, ktorý vznikne odčítaním 2 posebe 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



Obr. 3.1: Základný algoritmus

najviac na hranách. Podľa rýchlosti pohybu môžu byť obrysy hrubšie, alebo tenšie. Našim cieľom je čo najlepšie zachytiť pohybujúcu sa ruku.

Dohodneme sa, že najmenšia zmena (žiadna) bude znázornená bielou a najväčšia čiernou.

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.

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.

3.1.2 Spracovanie segmentov

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

TODO!!!

3.2 Návrh architektúry neurónovej siete

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 Cieľ

Našim cieľom je vytvoriť vhodnú architektúru neurónovej siete, ktorá bude rozhodovať o danom vstupe, či zodpovedá ruke alebo nie. Vyskúšame niekoľko typov architektúr, porovnáme ich a následne vyberieme najvhodnejšiu, ktorú potom použijeme.

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

3.2.2 Trénovanie

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

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 detekovali ruku. Po každej postupnosti sme rekurentné vstupy resetli na 0. Takto simulujeme správanie siete v reálnej aplikácii.

Na trénovanie sme použili dáta fourierovych transformácií (viď 3.3.4). Množina dát obsahovala cca 800 vzorov, ktoré boli pri trénovaní rekurentnej neurónovej siete rozdelené do približne 70 postupností.

3.2.3 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 počita takto: $\frac{1}{2}(t-o)^2$, kde t je cieľová hodnota a o je výstup siete. V tabuľkách budeme uvádzať priemernú kvadratickú chybu, ktorú budeme počitať ako súčet všetkých kvadratických chýb deleno počet testovacích vstupov.

Na testovanie sme mali množinu cca 650 vzorov (60 postupností), ktoré sme nepoužívali na trénovanie.

3.2.4 Viac vrstvová dopredná neurónová sieť

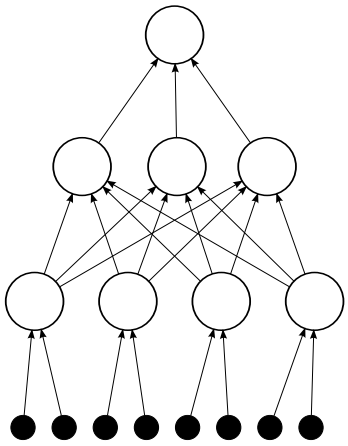
Viac vrstvová dopredná neurónová sieť (obr. 2.2) je neurónová sieť zložená z viacerých vrstiev neurónov, pričom signál sa šíri len zo spodnejšej vrstve na vyššiu. Viac o tomto type siete nájdete v kapitole 2.3.

Experimentálne sme zistili, že dvojvrstvová sieť na tento problém stačí a tretia vrstva nepomáha.

V tabuľke 3.1 je porovnanie úspešnosti na testovacích dátach pre rôzne počty neurónov a priemerná kvadratická chyba.

	Testovacia sada		**Trénovacia sada**		
Počet neurónov	**úspešnosť**	**chyba**	**úspešnosť**	**chyba**	**čas**
40	81,11%	0,0818			
45	85.55%	0,0643			
50	85,19%	0,0667			
60	85,19%	0,0634			

Tabuľka 3.1: Porovnanie úspešnosti NS pri rôznych počtoch neurónov



Obr. 3.2: Upravená dopredná neurónová sieť

3.2.5 Viac vrstvová dopredná neurónová sieť - upravená verzia

V upravenej verzii sme upravili spodnú vrstvu siete - tú, ktorá dostáva vstup. Vstup je rozdelený na 16 častí a ku každej časti je pridelených niekoľko neurónov. Každý neurón spracúva len vstupy z jeho časti (obr. 3.2).

Jej výhodou je rýchlosť. Náš vstup má rozmer $128 \times 128 = 16384$, čo nie je malé číslo. Rozdelíme ho na 16 častí s veľkosťou $32 \times 32 = 1024$. Takto namiesto toho aby každý vstupný neurón počítal s 16384 vstupmi počíta len s 1024, čo je 16× rýchlejšie. 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.

	Testovacia sada		**Trénovacia sada**		
Počet neurónov	**úspešnosť**	**chyba**	**úspešnosť**	**chyba**	**čas**
48; 12	83,33%	0,0745			
112; 12	84,07%	0,0714			

Tabuľka 3.2: Porovnanie úspešnosti upravenej NS pri rôznych počtoch neurónov

Keď si porovnáme tabuľku 3.3 s tabuľkou 3.1 zistíme, že rozdiel v úspešnosti nie je veľký. Preto sme sa rozhodli, že použijeme radšej túto architektúru. Trénovanie aj klasifikácia je značne rýchlejšia.

TODO!!!

3.2.6 Rekurentná neurónová sieť

Architektúra našej rekurentnej neurónovej siete vychádza z upravenej doprednej neurónovej siete, s tým, že namiesto obyčajného neurónu používame v niektorej vrstve rekurentný neurón (obr. 2.3). Viac o tomto type siete nájdete v kapitole 2.4.

	Testovacia sada		**Trénovacia sada**		
Rekurentná vrstva	**úspešnosť**	**chyba**	**úspešnosť**	**chyba**	**čas**
spodná	83,33%	0,0745			

Tabuľka 3.3: Porovnanie úspešnosti rekurentnej NS pri rôznom umiestnení rekurencie

TODO!!!

3.3 Metódy na zlepšenie úspešnosti klasifikátora ruky

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

3.3.1 Trénovacia a testovacia množina

TODO!!!

3.3.2 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.

TODO!!!

3.3.3 Pôvodný vs. rozdielový obrázok

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.

Toto sme chceli eliminovať tak, že sa vyberie ruka z pôvodného obrázka podľa farby. Táto ruka tam bude vždy len raz. Bohužiaľ tento prístup mal 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 osvetlenia. Navyše osvetlenie sa môže meniť pri pohybe ruky, čo veľmi stažuje nastavenie správnych parametrov. Pred použitím aplikácie by sa aplikácia musela nakalibrovať, čo znižuje komfort jej použitia.

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.

Rozhodujúcim problémom však bolo to, že úspešnosť siete na dátach, ktoré ani neobsahovali zle vybraté ruky bola aj tak nižšia ako u rozdielového obrázka (tabuľka 3.4). Preto sme sa rozhodli radšej pridať ďalšie dáta do trénovacej množiny pre rozdielové obrázky. Pri vyhodnocovaní tejto časti sme použili menšiu trénovaciu a testovaciu sadu, ktorá obsahovala cca 350 trénovacích a 300 testovacích vzorov. Sada neobsahovala zle vybraté ruky.

3.3.4 Fourierova transformácia

Fourierova transformácia zvykne často pomáhať, keď sa použie na predspracovanie dát pri trénovaní obrazových alebo zvukových vzoriek. Preto sme sa aj my rozhodli vyskúšať aký bude mať vplyv na úspešnosť.

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

Konvergencia chyby pri trénovaní bola značne rýchlejšia a použitie transformácie umožnilo dosiahnuť menšiu chybu na trénovacej množine.

TODO!!!

	Testovacia sada		Trénovacia sada		
Typ dát	**úspešnosť**	**chyba**	**úspešnosť**	**chyba**	**čas**
Rozdielový obr.	83,33%	0,0745			
Pôvodný obr.	68,52%	0,1174			
Rozdielový – FT	75,18%	0,0895			
Pôvodný – FT	70.37%	0,1158			

Tabuľka 3.4: Porovnanie úspešnosti NS pri rôznych dátach

Kapitola 4

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.

4.1 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 4.1 je class diagram nášho riešenia.

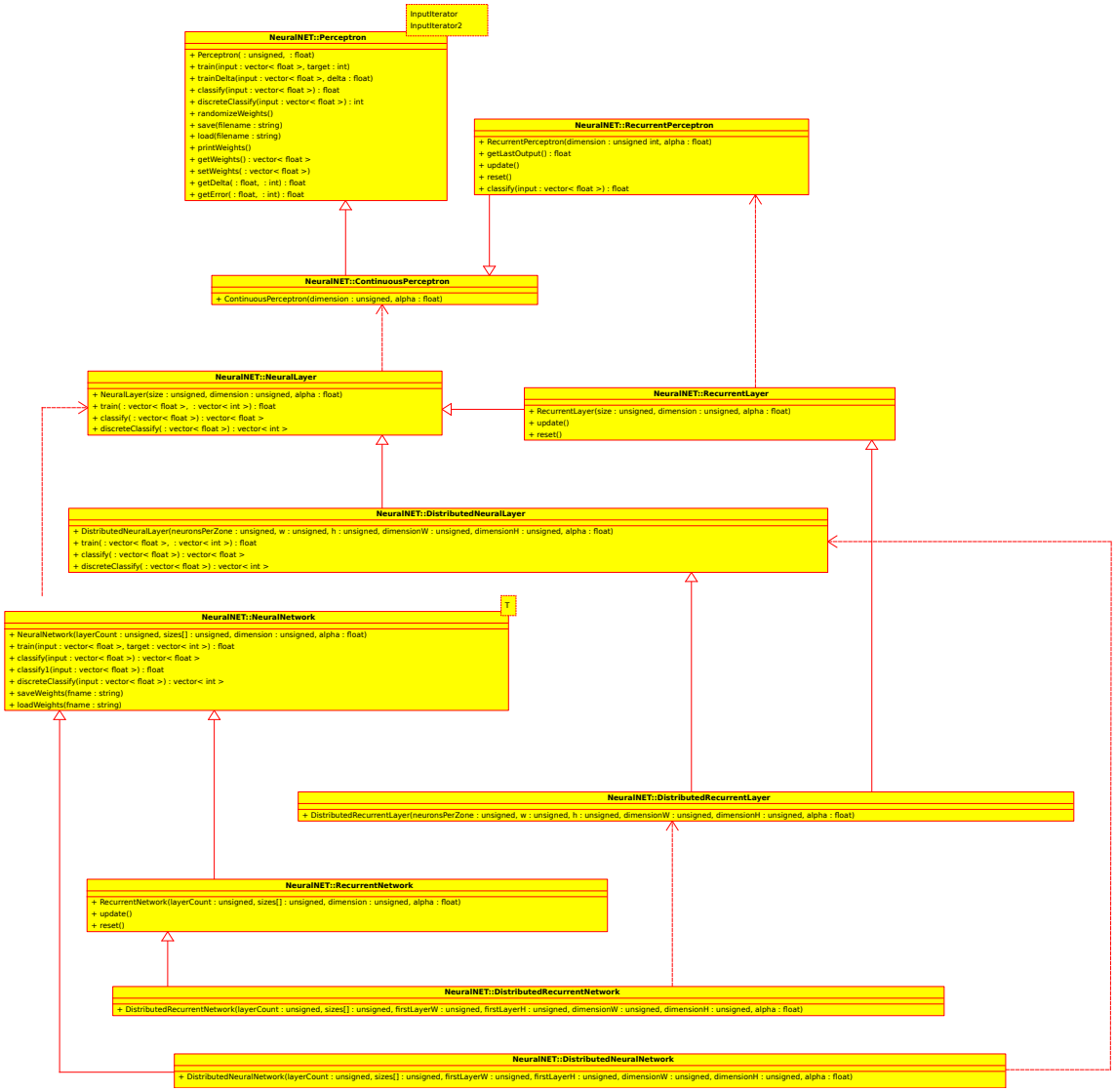
4.1.1 Perceptron

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



Obr. 4.1: Neurónové siete - Class diagram

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

4.1.2 ContinuousPerceptron

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

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

a jej deriváciu

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

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

4.1.3 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.

4.1.4 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 4.1.1

- `vector<float> classify(vector<float> input)`
  Klasifikačný algoritmus.
  Vstup: *input* - vstup siete
  Výstup: pole čísel z intervalu $(0, 1)$ - výstup siete.

4.1.5 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>` a `vector<float> classify(vector<float>)`, tak, aby trénovanie a klasifikácia prebiehala na príslušnej časti vstupu.

4.1.6 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.

4.1.7 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.

4.1.8 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.

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.

4.1.9 DistributedNeuralNetwork

TODO!!!

4.1.10 RecurrentNetwork

Trieda *RecurrentNetwork* je potomkom triedy *NeuralNetwork*. TODO!!!

4.1.11 DistributedRecurrentNetwork

TODO!!!

4.2 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)`. 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.

4.3 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

4.4 Spracovanie obrazu

4.4.1 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 rozdieli toho bude veľa. To nám však nevadí. TODO!!!

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 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.

Krok 3: Segmentácia

Ako sme už spomenuli v kapitole 3.1.1, v časti segmentácia, 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.

4.5 Rozpoznanie ruky

Rozpoznanie ruky má nastarosti 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. TODO!!!

4.6 Rozpoznanie gesta

TODO!!!

4.7 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.

4.8 Pomocné programy

Upravená verzia aplikácie

Upravená verzia aplikácie umožňuje jednoduché ukladanie dát na disk. TODO!!!

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.

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. TODO!!!

Skripty

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

4.9 Problémy a ich riešenia

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.