

UNIVERZITA KOMENSKÉHO, BRATISLAVA  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

OVĽÁDANIE POČÍTAČOVÝCH APLIKÁCIÍ  
POMOCOU GEST RUKY

Bakalárska práca

2012

Michal Hozza

UNIVERZITA KOMENSKÉHO, BRATISLAVA  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

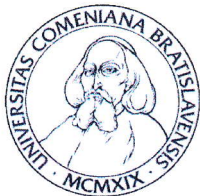
OVLÁDANIE POČÍTAČOVÝCH APLIKÁCIÍ  
POMOCOU GEST RUKY

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

<b>Úvod</b>	<b>1</b>
<b>1 Platforma, jazyk a knižnice</b>	<b>2</b>
<b>2 Neurónové siete</b>	<b>3</b>
2.1 Jednoduchý spojitý perceptrón . . . . .	3
2.2 Jednovrstvová neurónová sieť . . . . .	4
2.3 Viacvrstvová dopredná neurónová sieť . . . . .	4
<b>3 Návrh</b>	<b>6</b>
3.1 Základný algoritmus . . . . .	6
3.1.1 Rozdielový obraz . . . . .	6
3.1.2 Predspracovanie vstupného obrazu . . . . .	7
3.1.3 Segmentácia . . . . .	7
3.1.4 Predspracovanie segmentov . . . . .	8
3.2 Návrh architektúr neurónových sietí . . . . .	9
3.2.1 Požiadavky na architektúru neurónovej siete . . . . .	9
3.2.2 Typ 1: Viac vrstvová dopredná neurónová sieť . . . . .	10
3.2.3 Typ 2: Upravená verzia viac vrstvovej doprednej neurónovej siete	10
3.2.4 Typ 3: Rekurentná neurónová sieť . . . . .	11
<b>4 Experimenty</b>	<b>13</b>
4.1 Trénovanie . . . . .	13
4.2 Testovanie a vyhodnocovanie neurónových sietí . . . . .	13
4.3 Trénovacie a testovacie dáta . . . . .	14
4.4 Porovnanie architektúr neurónových sietí . . . . .	14
4.5 Porovnanie rôznych typov dát . . . . .	15
<b>5 Implementácia</b>	<b>17</b>
5.1 Neurónové siete . . . . .	17
5.1.1 Perceptron . . . . .	17

5.1.2	ContinuousPerceptron . . . . .	19
5.1.3	RecurrentPerceptron . . . . .	19
5.1.4	NeuralLayer . . . . .	20
5.1.5	DistributedNeuralLayer . . . . .	20
5.1.6	RecurrentLayer . . . . .	20
5.1.7	DistributedRecurrentLayer . . . . .	21
5.1.8	NeuralNetwork . . . . .	21
5.1.9	DistributedNeuralNetwork . . . . .	21
5.1.10	RecurrentNetwork . . . . .	21
5.1.11	DistributedRecurrentNetwork . . . . .	21
5.2	Triedy pre obrázky . . . . .	22
5.3	Získanie obrazu z webkamery . . . . .	23
5.4	Spracovanie obrazu . . . . .	23
5.4.1	Trieda ImageProcessor . . . . .	23
5.5	Rozpoznanie ruky . . . . .	25
5.6	Rozpoznanie gesta . . . . .	25
5.7	Simulácia stlačenia kláves . . . . .	25
5.8	Pomocné programy . . . . .	25
5.9	Problémy a ich riešenia . . . . .	26
	<b>Záver</b>	<b>27</b>
	<b>Literatúra</b>	<b>28</b>



# 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 . . .	8
3.3	Upravená dopredná neurónová sieť . . . . .	10
3.4	Rekurentný neurón . . . . .	11
5.1	Neurónové siete - Class diagram . . . . .	18

# Zoznam tabuliek

4.1	Veľkosti sád vstupov . . . . .	14
4.2	Porovnanie úspešnosti NS pri rôznych počtoch neurónov . . . . .	15
4.3	Porovnanie úspešnosti upravenej NS pri rôznych počtoch neurónov . . .	15
4.4	Porovnanie úspešnosti rekurentnej NS pri rôznom umiestnení rekurencie	16
4.5	Porovnanie úspešnosti NS pri rôznych dátach . . . . .	16

# 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

## 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. Navyše je objektovo orientovaný a umožňuje jednoduché použitie komplexných dátových štruktúr.

Programovanie a prácu nám uľahčia už hotové knižnice. Využijeme knižnicu *Qt* (multiplatformová knižnica umožňujúca 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 umožňuje simulovanie kláves v X servri).

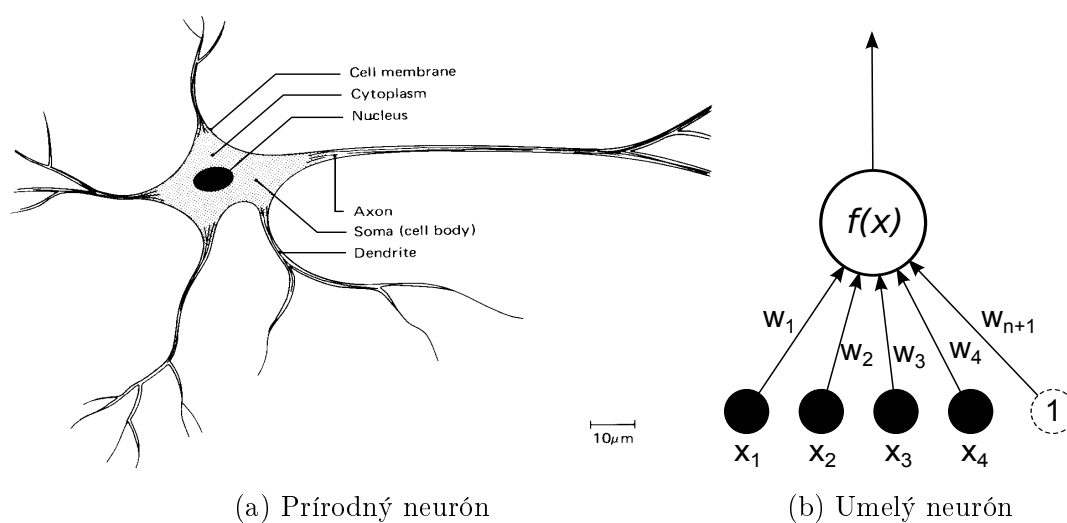
Aplikácia využíva vlákna, na paralelizáciu a urýchlenie niektorých výpočtov.

# 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 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 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 ostávajú 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 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

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 môžeme vrstvy pospájať.

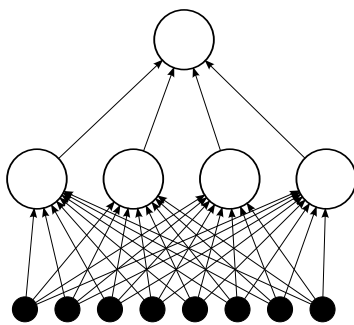
Neuróny vo vrstve nie sú nijako pospájané a každý teda operuje nezávisle, takže z praktického hľadiska je to skupina neurónov pracujúcich nad tým istým vstupom.

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

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 trénujú algoritmom backpropagation<sup>1</sup> [Hay99, Fed11].

---

<sup>1</sup>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.

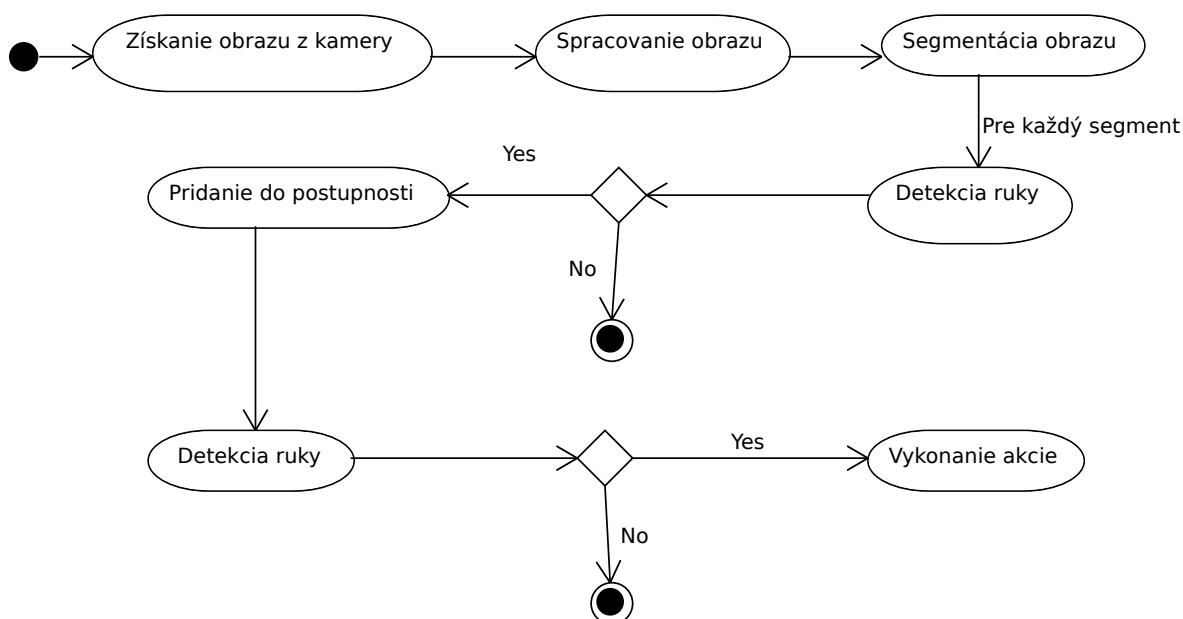
### 3.1 Základný algoritmus

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 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 hodnoty  $0 \dots 255$ . 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  $11 \times 11$ . 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

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. 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  $128 \times 128$  - a normalizuje sa.

#### Normalizácia dát

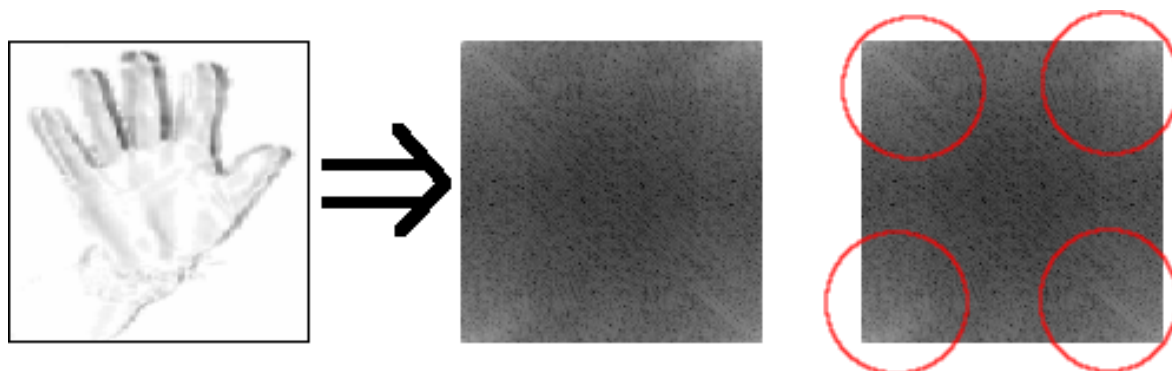
Ideálne vstupy pre neurónovú sieť sú z intervalu  $\langle 0, 1 \rangle$ . Pixle čiernobielych obrázkov majú hodnoty  $\{0 \dots 255\}$ . Pri obrázkoch zvolíme pre farbu pozadia hodnotu 0 a pre objekt ostatné hodnoty. Z tohto dôvodu chceme, aby rozdiel v normalizovanej hodnote medzi 0 a 1 bol najväčší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



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

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 ??) a vyberieme najvhodnejšiu z nich, ktorú potom použijeme v aplikácii.

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<sup>1</sup> siete.

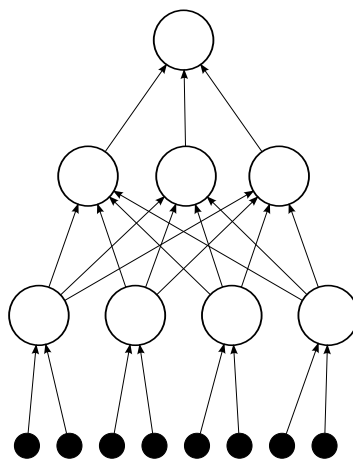
### 3.2.2 Typ 1: Viac vrstvomá dopredná neurónová sieť

Viac vrstvomá dopredná neurónová sieť (obr. 2.2) je implementáciu klasického viacvrstvomého 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 vrstvomovej 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 ( $4 \times 4$ ) a ku každej časti sme pridelili niekoľko neurónov. Každý neurón spracúva len vstupy z jeho časti (obr. 3.3).

---

<sup>1</sup>vzhľadom na počet váh

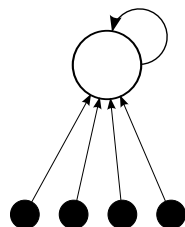
Jeho výhodou je zvýšenie rýchlosti pri rovnakom počte neurónov - zníži sa počet váh vstupných neurónov. Náš vstup má rozmer  $128 \times 128 = 16384$ , čo nie je malé číslo. Keď ho rozdelíme na 16 častí, jednotlivé časti budú mať veľkosť  $32 \times 32 = 1024$ . Takto namiesto toho aby každý vstupný neurón počítal s 16384 vstupmi počíta len s 1024, čo je  $16\times$  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. Môžeme 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 šírla 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 v niektorej vrstve 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 updatnúť 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 - v rámci 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 detekovali 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:  $\frac{1}{2}(t - o)^2$ , kde  $t$  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, 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 trénovaciu a testovaciu množinu.

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.

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.

	Testovacia množina		Trénovacia množina	
Sada	ruky	ostatné	ruky	ostatné
1. sada				
2. sada				

Tabuľka 4.1: Veľkosti sád vstupov

### 4.4 Porovnanie architektúr neurónových sietí

**Viac vrstvomá 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 dvojvrstvomá sieť na tento problém stačí a tretia vrstva nepomáha.

V tabuľke 4.2 je porovnanie úspešnosti na testovacích dátach pre rôzne počty neurónov a priemerná kvadratická chyba.

Keď si porovnáme tabuľku 4.4 s tabuľkou 4.2 zistíme, že rozdiel v úspešnosti nie je veľký. Preto sme sa rozhodli, že použijeme radšej túto architektúru. Trénovanie aj klasifikácia je značne rýchlejšia.

TODO!!!

TODO!!!



	Testovacia množina		Trénovacia množina		
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 4.2: Porovnanie úspešnosti NS pri rôznych počtoch neurónov

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

Tabuľka 4.3: Porovnanie úspešnosti upravenej NS pri rôznych počtoch neurónov

## 4.5 Porovnanie rôznych typov dát

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

Tabuľka 4.4: Porovnanie úspešnosti rekurentnej NS pri rôznom umiestnení rekurencie

	Testovacia množina		Trénovacia množina		
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 4.5: Porovnanie úspešnosti NS pri rôznych dátach

# 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 vrstvomý 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

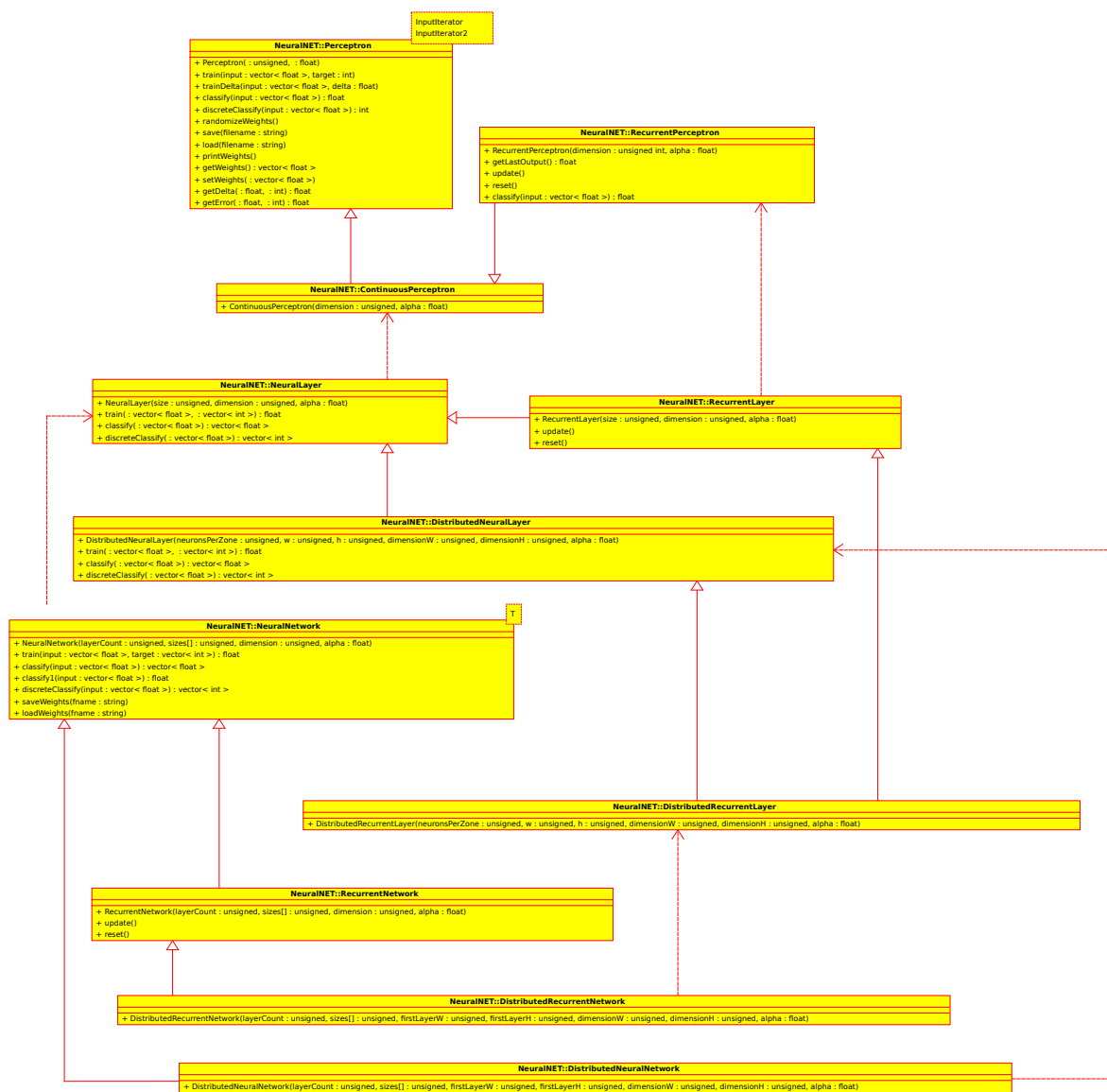
*Perceptron* je abstraktná trieda, ktorá zovšeobecňuje rôzne typy perceptrónov, ktoré sa líšia rôznymi aktivačnými funkciami.

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 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* =  $\delta$  z učiaceho pravidla:  $\Delta w_i = \alpha \delta x_i$ <sup>1</sup> [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.

---

<sup>1</sup>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 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>, vector<float>)` 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 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 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.

#### 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

TODO!!!

### 5.1.10 RecurrentNetwork

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

### 5.1.11 DistributedRecurrentNetwork

TODO!!!

## 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 *HCIImage*, 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 kockrbatosť, ktorá môže mať výrazný vplyv pri použití Fourierovej transformácie.

### Maskovanie

Metóda `mask(HCIImage *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"<sup>2</sup>, 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)`. 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 odstráni<sup>3</sup> 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.

## 5.4 Spracovanie obrazu

### 5.4.1 Trieda ImageProcessor

Trieda *ImageProcessor* má na starosti spracovanie obrazu z webkamery, jeho segmentáciu a posunutie na ďalšie spracovanie triede *HandRecognizer*. *ImageProcessor*

<sup>2</sup>známu z grafických programov ako je GIMP, či Photoshop

<sup>3</sup>začierni

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 &coloring`), 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.

### 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  $O(k^2 \cdot w \cdot h)$ . 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  $O(w \cdot h)$  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 ??, 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.

## 5.5 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!!!**

## 5.6 Rozpoznanie gesta

**TODO!!!**

## 5.7 Simulácia stlačenia kláves

Knižnica *Xtst* poskytuje funkciu *XTestFakeKeyEvent*, ktorá nám umožňuje 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 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 tréningovú/testovaciu sadu. **TODO!!!**

### **Skripty**

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

## **5.9 Problémy a ich riešenia**

**TODO!!!**

# Záver

Tu bude záver. **TODO!!!**

# Literatúra

- [Fed11] Dominika Fedáková. Aplikované využitie neurónových sietí, 2011.
- [Hay99] S. Haykin. *Neural Networks: A Comprehensive Foundation*. IEEE, 1999.
- [kap] Kapture. <http://kapture.berlios.de/>. [Online; accessed 13-May-2012].
- [KBP<sup>+</sup>97] V. Kvasnička, L'. Beňušková, J. Pospíchal, I. Farkaš, P. Tiňo, and A. Král'. *Úvod do teórie neurónových sietí*. Iris, 1997.
- [Per09] Peter Perešíni. Fourierova transformácia a jej použitie, 2009.