Rozpoznávání ručně psaných symbolů

Recognition of handwritten symbols

Bc. Milan Medlík

2010

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

akademický rok: 2009/2010

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Milan MEDLÍK**
Osobní číslo:	**A08410**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Rozpoznávání ručně psaných symbolů**

Zásady pro vypracování:

1. **Vypracujte literární rešerši, která bude shrnovat používané způsoby pro automatizované rozpoznávání ručně psaných znaků**
2. **Vyberte, případně navrhněte, a podrobně popište algoritmus pro rozpoznávání ručně psaných samostatných symbolů z bitmapové předlohy. Vstupem algoritmu bude bitmapa obsahující jeden symbol a podmnožina ASCII znaků, v níž má symbol být obsažen. Výstupem pak bude rozpoznaný symbol a míra správnosti rozpoznání (jak moc si je program jistý výsledkem)**
3. **Algoritmus realizujte formou programu spustitelného v prostředí MS Windows. Program zpracujte tak, aby rozpoznávací rutiny byly snadno začlenitelné do jiných programových projektů.**
4. **Otestujte výsledný program a zhodnoťte jeho klady a zápory.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **SOBOTKA, B. Grafické formáty. České Budějovice : Kopp, 1996. 156 s. ISBN 8085828588.**
2. **SURESH, R.M., ARUMUGAM, S. Fuzzy technique based recognition of handwritten characters. Image and Vision Computing. 2007, vol. 25, no. 2, s. 230–239. Dostupný z WWW: http://www.sciencedirect.com.**
3. **PESSOA, Lúcio F. C., MARAGOS, Petros. Neural networks with hybrid morphological/rank/linear nodes: a unifying framework with applications to handwritten character recognition. Pattern Recognition. 2000, vol. 33, no. 6, s. 945–960. Dostupný z WWW: http://www.sciencedirect.com.**
4. **CHALUPA, Petr. Programové vybavení pro vizuální vyhodnocování testů. Zlín, 1999. 55 s. Vysoké učení technické v Brně. Vedoucí diplomové práce Ing. Tomáš Sysala.**

Vedoucí diplomové práce: **Ing. Petr Chalupa, Ph.D.**
Ústav řízení procesů

Datum zadání diplomové práce: **19. února 2010**

Termín odevzdání diplomové práce: **8. června 2010**

Ve Zlíně dne 19. února 2010

L.S.

prof. Ing. Vladimír Vašek, CSc.
děkan

prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

ABSTRAKT

Tato práce pojednává o principech rozpoznávání ručně psaného písma. Jejím cílem je vytvoření knihovny, na jejímž vstupu jsou grafická data s ručně psaným symbolem a jejíž výstupem bude rozpoznaný symbol v počítačové čitelné podobě. Značný díl teoretické části, která mimo jiné obsahuje historii vývoje OCR, je také věnována programovacím jazykům, rozhraním a knihovnám, které byly pro vývoj knihovny použity. V závěru teoretické části je několik slov k neuronovým sítím. Praktická část začíná organizací projektů, a poté popisem ovládání demonstračního programu, jenž je také součástí práce. Úkolem tohoto demonstračního programu je zobrazovat výsledky knihovny. Práce se proto zabývá popisem jeho metod a knihovnou zprostředkovávající napojení na knihovnu. Nejdůležitější částí je popis samotné knihovny. Nachází se zde přehled jednotlivých funkcí a metod jimi užívanými. Nechybí ani popis principů doplněn obrázky. Podstatná část je věnována využití neuronové sítě. Popisem experimentu a nastavení parametrů při jejich trénování. V závěru je ohodnocen výsledek a jeho další možné vize, jakým směrem by se mohl tento projekt vydat do budoucnosti.

Klíčová slova:

OCR, rozpoznávání písma, Neuronová síť, C/C++, C#, Gtk-sharp

ABSTRACT

The thesis deals with the principles of recognition of hand-written script. The aim is to create a library, its input includes graphical data with a hand-written symbol and its output will be a discerned symbol in a computer-legible form. A large portion of the theoretical part which includes beside others the history of OCR development also deals with the programming languages, interfaces and libraries that were used for the development of the library. In the conclusion of the theoretical part, neuronal nets are discussed, too. The practical part starts with organisation of the projects, after that the description of the demonstration programme control which is included in the thesis as well follows. The role of the demonstration programme is to represent the results of the library. For this reason the thesis deals with the description of its methods and the library mediating the connection with the library. The most important part is the description of the library itself. There is a summary of separate functions and methods used by them. A description of principles supplemented with pictures is also included. A large proportion takes notice of the use of a neuronal net, the setting of its parameters during the training and the description of the experiments. At the end, the result and its possible visions are evaluated to discuss which direction the project may go in the future.

Keywords:

OCR, handwriting recognition , artificial neural network, C/C++, C#, Gtk-sharp

Děkuji svému vedoucímu, panu Ing. Petru Chalupovi, Ph.D za jeho nápady v oblasti řešení mé práce. Dále děkuji Ing. Radku Červinkovi, za jeho věcné rady a připomínky. V neposlední řadě také děkuji rodině, přátelům a všem, kteří mě v tomto těžkém období provázeli a všemožně podporovali. Děkuji.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce
  využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu
  využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

§ že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.

§ že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

…….

podpis diplomanta

OBSAH

ÚVOD

Jedna z mála věcí, která spojuje čtenáře této diplomové práce, je schopnost číst. Písmo je nejvýznamnější vynález lidstva. Díky písmu můžeme mluvit o civilizaci. Jsme vzdělaní a tak dosahujeme pokroku. Písmo se zajisté také podílelo na tom, že z nejednotných kmenů se staly národy. Bez písma bychom totiž nemohli sdělovat a uchovávat myšlenky příštím generacím.[0]

Už jen málokomu z nás utkvěl v paměti první rok ve školní lavici, kde jsme se učili rozpoznávat písmenka, slabiky, věty. Za pouhý rok jsme byli schopni přečíst prakticky cokoliv. Proto si jen těžko dokážeme představit, jaký zázrak to vlastně je, že se děti naučí rozumět mluvenému a později i psanému slovu. Až když se lidé pokusili to samé naučit i stroje, zjistili, že to zdaleka tak samozřejmé není. Není se čemu divit. Asi před dvaceti tisíci lety se objevily první malby, ale až o sedmnáct tisíc let později začíná lidstvo k vyjadřování používat písmo!

Jen pro zajímavost, původ slova ***psáti*** pochází ze staroslověnského ***pisatb***, tj. podobné latinskému ***pingere***. Po etymologickém rozboru zjistíme, že obojí původně znamenalo malovat. Též latinské ***scribere*** znamenalo původně vrytí.

Tolik let samostatného vývoje písma však svědčí o tom, že vznik a vývoj písma je nesmírně dlouhý a složitý proces. I když se nám zdá, že žijeme v době, která je spolehlivému strojovému nebo chcete-li počítačovému čtení velmi blízko, tato zařízení se však zdaleka nenaučila číst za rok! (více v části o historii OCR). Tato diplomová práce se zabývá zhodnocením metod a využitím některé z nich k realizaci rozpoznávání ručně psaných symbolů.

Tuto prácí jsem si zvolil z důvodu její zajímavosti. Jako člověk jsem si chtěl zodpovědět několik otázek: „Jak tyto programy fungují uvnitř?“, „Je jejich realizace opravdu natolik složitá, aby odpovídala ceně, kterou účtují společnosti za jejich komerční produkty?“, „Mohu se jim alespoň trochu přiblížit?“. Jako člověk-vývojář, jsem si položil otázky ohledně programové implementace: „Je neuronová síť opravdu tak všemocná, jak se prezentuje? Dokáže se naučit při jakýchkoliv vstupech generovat požadované výstupy?“, „Nakolik složitý bude proces učení?“, „Nakolik bude výsledek odpovídat hledanému výsledku?“

V hlavě jsem měl spoustu nápadů, jak této práce využít k užitku větší masy lidí, například napojením tohoto modulu na webovou stránku a nabízet tak rozpoznávaní online.V neposlední řadě jsem také uvažoval o použití nastupující technologie OpenCL pro výpočet podobnosti symbolů. V době, kdy píši tyto řádky znám odpovědi na některé z nejtíživějších otázek. Vy se však dozvíte odpovědi k jednotlivým otázkám, a nakolik jsou uspokojivé, až v závěru této práce. Součástí jsou také útrapy, které mě během experimentů potkaly a také to, proč některé mé vize nebyly nakonec začleněny do výsledného projektu.

I. TEORETICKÁ ČÁST

1 ÚVOD DO OCR

Tato technologie se zaměřuje na digitalizaci tištěných textů, s nimiž pak lze pracovat jako s normálním počítačovým textem. Jde o převod z analogové podoby do digitální. Počítačový program převádí obraz buď automaticky, nebo se musí naučit rozpoznávat znaky. Převedený text je téměř vždy v závislosti na kvalitě předlohy třeba podrobit důkladné korektuře, protože OCR program nerozezná všechna písmena správně. Optickým rozpoznáváním obrazu se v současné době zabývá mnoho komerčních programů, jejichž cena odpovídá kvalitě převedeného textu.

1.1 Historie OCR

Historie OCR se začíná psát v Německu roku 1929. Tehdy si Gustav Tauschek nechává patentovat jeho přístroj na rozpoznávání symbolů. Do konstrukce svého zařízení se pouští až po roce 1933, kdy je patentován také v USA. Hotové dílo pracuje převážně na mechanickém základě. Stroj obsahoval šablony jednotlivých znaků, pokud se šablona pěkně překrývala s daným znakem (což posoudil fotoreceptor, který tak viděl jenom bílou), prohlásil systém znaky za shodné.[1]

Další vývoj se udál v roce 1950. Krypto-analytik David H. Shepard pracoval na automatizaci přepisování dat do strojové formy. Množství tištěného materiálu bylo velké, a proto se Shepard se svým přítelem Harvem Cookem pokusili sestavit stroj, který si o rok později nechali patentovat pod jednoduchým názvem „Aparát na čtení“. David Shepard si s Williamem Lawlessem založili v roce 1952 firmu Intelligent Machines Research Corporation, aby tak pomohli komerčnímu úspěchu jejich mašinky nazvané "Gismo". Společníci Shepard a Harve měli štěstí. O jejich zařízení projevila zájem společnost IBM. Firma IBM nakonec jejich vynález odkoupila i s patenty. Později opět firma IBM dala mladé firmě úkol vyrobit stroj, který by byl schopný rozpoznávat ručně psané číslice. Mezitím IBM nadále rozvíjela systém rozpoznávání a poprvé jej také pojmenovala slovy Optical Character Recognition. V dnešní době probíhá rozpoznávání znaků a symbolů vesměs v digitální podobě. Přesto se název označující spíše opticko-mechanickou cestu udržel až do dnešních dnů.[1]

1.2 První komerční využití - předplatné a kreditky

Jako první si aplikaci tohoto systému zakoupila firma Readers Digest. Starší bratříček Gisma od firmy IMR jim pomáhal v oddělení pro předplatné. Významným odběratelem byla také kalifornská společnost Standard Oil Company, která OCR začala využívat pro čtení obtisknutých čísel kreditních karet na účtech. Systémy pro rozpoznávání znaků používá americká pošta od roku 1965. Využívala přístroje navržené Jacobem Rabinowem, velice plodným americkým vynálezcem, který je kromě čtecích zařízení zodpovědný také například za vylepšení v magnetickém ukládání informací. Evropské pošty začaly systémů OCR využívat v roce 1971. Využití této technologie vyústilo 13. ledna 1973, kdy byl představen čtecí stroj pro nevidomé, navržený Rayem Kurzweilem. Tento přístroj sebou přinesl objev dvou vynálezů: CCD plochého scaneru a textového syntetizéru řeči.[1]

1.3 Současnost

V současnosti je rozpoznávání strojem psaného textu v latince považováno za velice dobře zvládnuté a vyřešené. Můžeme se s ním setkat například v aplikacích na scanování dokumentů, jejichž typická přesnost rozpoznávaní je až 99 procent. Naprosté přesnosti lze dosáhnout až lidským zásahem. V ostatních oblastech rozpoznávaní, jako jsou rozpoznávání ručně psaného písma nebo písma v jiné znakové sadě, například skládající se z více znaků než má latinka, jsou i nadále předmětem aktivního výzkumu. Přesnost výsledku lze zlepšit zahrnutím slovníkové metody, pomocí které se program snaží doplnit písmeno podle slov, která jsou přípustná v aktuálním rozpoznání. Znalost gramatiky pomáhá určit, zda rozpoznávané slovo je podstatné jméno nebo sloveso Tato metoda se osvědčila, neboť dokáže nepřesnost snížit z 5 procent (95procentní úspěšnosti) až na nepřesnost pod 1 procento.[2]

1.4 On-line vs. Off-line rozpoznávání

On-line rozpoznávání je často zaměňováno s pojmem rozpoznávání znaků. OCR je implementací Off-line, kdy systém rozpoznává znaky pevného statického tvaru, kdežto on-line rozpoznávání snímá pohyb ruky při psaní rukou. S on-line rozpoznáváním se nejčastěji setkáváme v Tablet-PC, Palm OS a jim podobným. Jejich algoritmy analyzují snímaná gesta rukou a na základě toho můžeme říct, zda tato čára byla vedena zprava doleva nebo obráceně, můžeme také určit její rychlost. Využívají také toho, že uživatele musí naučí určité tahy pro určité symboly. Tyto metody bohužel nelze použít u softwaru,

který scanuje dokumenty ze statické podoby jako je papír. U ručně psaného textu je úspěšnost rozpoznání 80 – 90 procent. Zdánlivě velká míra úspěšnosti se bohužel projevuje až desítkami chyb na jedné straně naskenovaného textu. Tvary jednotlivých znaků ručně psaného písma samy o sobě neobsahují dostatek informací, aby byly rozpoznány s více než 98 procentní úspěšností. Je jednodušší rozpoznávat celá slova a porovnávat je slovníkovou metodou. Je třeba pochopit, že technologie OCR, která je využívána u skenovacích aplikací může být unikátní, patentovaná a nesnadno kopírovatelná, přestože vychází ze základů OCR. Pro složitější problémy rozpoznávání jsou navrhnuty inteligentní systémy využívající umělé neuronové sítě.[2]

1.5 reCAPTCHA

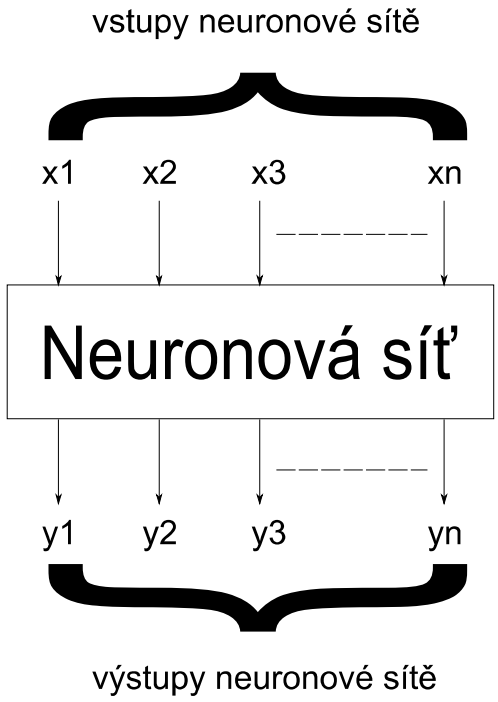
Tato technika vychází z metody CAPTCHA, což je metoda, která si bere za cíl na webu automaticky rozeznat člověka od robota. Setkáváme se s ní například během přispívaní na internetových fórech nebo při vyplňování různých formulářů. Tento test spočívá zpravidla v zobrazení obrázku s deformovaným textem tak, aby i roboti s pokrokovými metodami OCR rozpoznávaní nedokázali text obrázku rozpoznat. Člověk ovšem tento text rozezná a je vyzván k přepisu textu z tohoto obrázku do počítače pro ověření.

Vlastní funkce systému je založena na analýze naskenovaného textu pomocí dvou OCR programů - v případě, že se programy neshodnou, je nejasné slovo převedeno do CAPTCHA. Toto slovo je poté zobrazeno pro kontrolu se slovem již známým. Systém je nastaven tak, že pokud člověk napíše kontrolní slovo v pořádku, nejasné slovo je poté také považováno za správné. Identifikace OCR programem je ohodnocena hodnotou 0,5 bodu a každá interpretace člověkem má hodnotu jednoho bodu. Jakmile hodnota identifikace slova dosáhne 2,5 bodu je slovo považováno za správně identifikované.[3]

Wikipedie hovoří o **reCAPTCHA** jako o bezplatné službě, která pomáhá s digitalizací tištěných médií, jako jsou knihy nebo časopisy. ReCAPTCHA systém byl vyvinut na Carnegie Mellon univerzitě v USA, která používá jako základ CAPTCHA systém. reCAPTCHA aktuálně pomáhá s digitalizací archivu New York Times. Dvacet let archivu New York Times již bylo digitalizováno a zbývajících 110 let má být digitalizováno pomocí reCAPTCHA do konce roku 2010. Systém uvádí, že zobrazuje 30 milionů OCR obrázků každý den (údaj z prosince 2007) a systém reCAPTCHA je používán weby jako Twitter, Facebook nebo TicketMaster.

2 NEURONOVÉ ŠÍTĚ

Poměrně novým prostředkem pro zpracování dat jsou technické neuronové sítě. Lze říci, že neuronová síť realizuje zobrazení ze vstupního vektorového prostoru do vektoru prostoru výstupního. I když to není z obrázku č. 1 zcela patrné, dimenze těchto prostorů mohou být obecně různé.



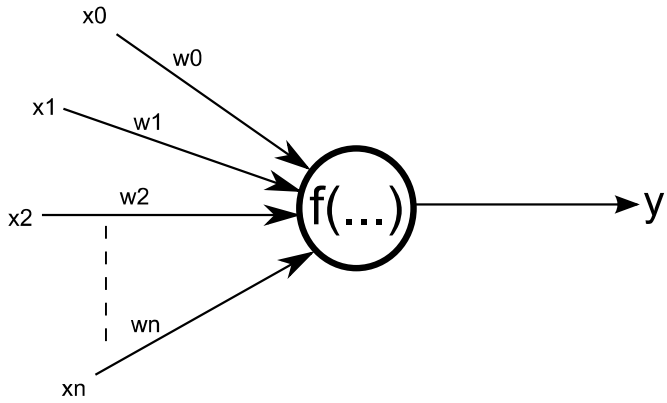
Obrázek 1Vstupy a výstupy neuronové sítě

Neuronové sítě mohou být aplikovány v obecném chápání velmi různorodým způsobem. V oblasti našeho zájmu jde především o použití jako klasifikátoru – pak vstupní vektor reprezentuje úsek signálu nebo obrazu (případně vhodně transformovaný) a výstup sítě naznačuje, jak byl tento úsek zatříděn. Využít lze také toho, že některé typy sítí mají schopnost optimalizace a použít jich pro restauraci signálu podle vhodného kritéria. Důležitou aplikací některých typů sítí je využití jako asociativní paměti, schopné vybavit informaci na základě pouze neúplného zadání. Výběrem vhodné architektury sítě lze dosáhnout toho, že v jistém místě je přenášena informace tzv. úzkým hrdlem, což může mít za následek, že při v podstatě neovlivněném informačním obsahu se přenáší jen zlomek původního rozsahu dat – dochází tedy k redukci dat.

Výše uvedené funkce lze ovšem realizovat i klasickými prostředky. Podstatný rozdíl je ve způsobu návrhu, zatímco klasické prostředky jsou přesně definovány pro jistý účel, aby bylo možné exaktně analyzovat jejich účinnost. Neuronové sítě jsou vždy pouze rámcem

reprezentovaným zvolenou strukturou sítě. Způsob, jakým s nimi experimentujeme, podstatně ovlivňuje způsob jejich práce a jimi realizované výsledky. Je také důležité, jakých algoritmů se používá pro opravu jejich parametrů v tzv. procesu učení a jak interpretujeme výstupní veličiny, resp. chování sítí.[4]

V posledních dvaceti letech způsobilo nevídaný rozmach této oblasti zpracování dat. Způsobila vesměs pozitivní vlastnost současnosti, a to skutečnost, že simulační programy jsou v současnosti dostupné a umožňují využití neuronových sítí i úplným laikům. Klasické architektury neuronových sítí téměř zaručují dosažení nějakého výsledku v konkrétním experimentu. Není však zaručeno, že to je výsledek dobrý nebo dokonce optimální. To souvisí především s tou skutečností, že přes značné pokroky i v teorii neuronových sítí není dosud možné interpretovat učením získané soubory vah ve vztahu k žádoucím nebo i dosaženým výsledkům – nedokážeme vysvětlit, proč síť dala za jistých okolností (vstupů, způsobu učení, atd.) takový či jiný výsledek rozhodnutí apod. Formalizovaná analýza či dokonce syntéza sítí je za této situace omezena jen na některé pravděpodobnostní charakteristiky, jejichž vztah k řešení úlohy bývá ovšem značně odtažitý. Odtud pramení i jistá nedůvěra zejména teoretiků k těmto přístupům. [5]



Obrázek 2 Technický neuron schematicky

V počátku vznikali technické sítě jako jisté zjednodušené modely biologických sítí. Je potřeba říci, že podobnost je značně vzdálená a že technické sítě se rozvíjejí již dlouhou dobu zcela nezávisle na skutečných vlastnostech svých biologických vzorů. Dnešní modely biologických neuronů, které berou v úvahu transportní jevy v buněčných membránách, různé mechanismy šíření vzruchů v nervových vláknech a podobně. Jsou nesrovnatelně složitější než procesní prvky v technických sítích, které z historického důvodu nesou stejný název. V této práci používám pojmy neuron a neuronová síť v čistě technickém smyslu, bez nároku na správnost analogie s biologickými neuronovými sítěmi.[5]

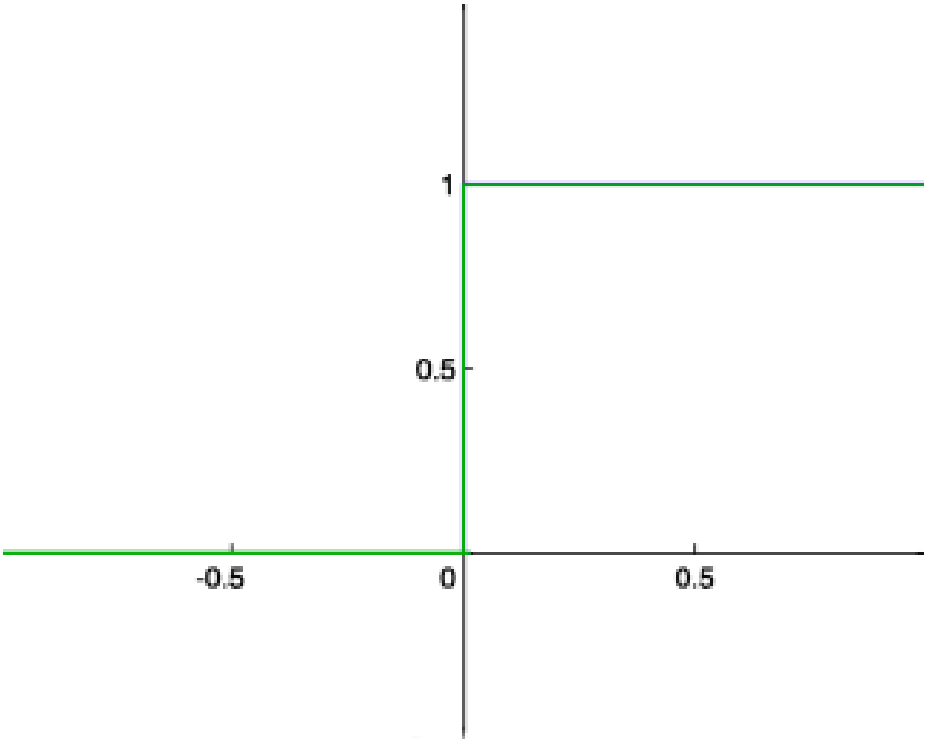
2.1 Neuron

Pro lepší představivost technického neuronu je schématicky zobrazen na obrázku č.2 Zkládá se z několika vstupů (x_0 … x_n) vah přiřazeným jednotlivým vstupům (w_0 … w_n) a jeho výstupu *y*. *Matematicky je Neuron* definován jako procesní prvek, který má N vstupů a jeden výstup *y* a je charakterizován rovnicí č 1:

$$y = S(\sum_{i=1}^{N}(w_i x_i) + \Theta)$$

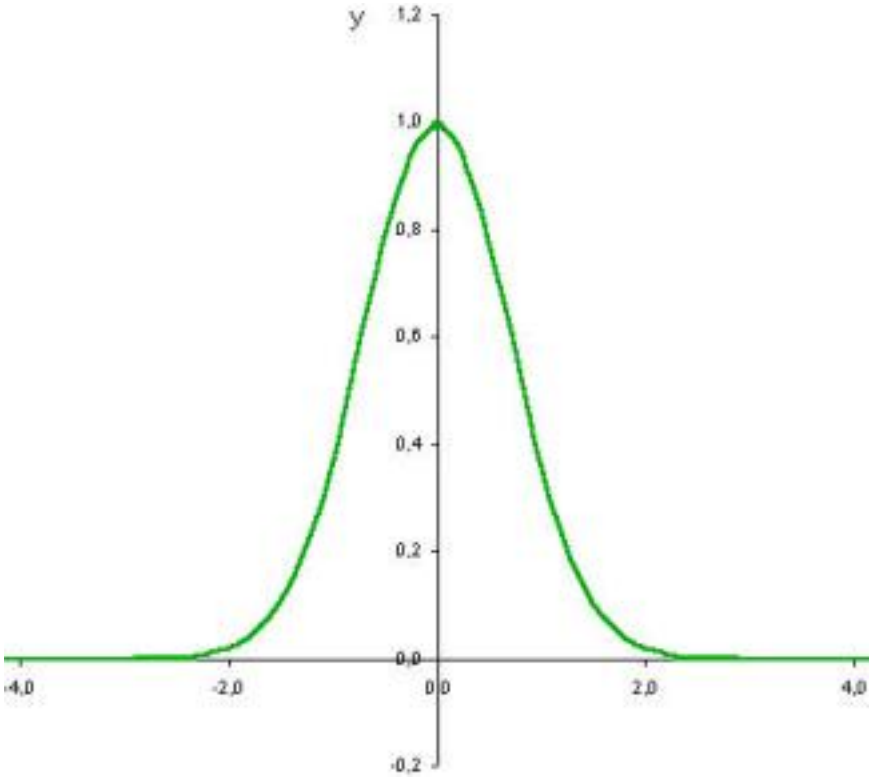
Rovnice 1 přechodová funkce neuronu

udávající jeho výstup *y* pro vstupní vektor $x = [x_1, x_2, x_3, \ldots, xN]$ přičemž $w = [w_1, w_2, w_3, \ldots w_N]$ je vektor aktuálních vah, Θ je aktuální práh neuronu, S je zvolená, ale dále již neměnná zpravidla nelineární funkce, která se nazývá *charakteristika neuronu* Argument této funkce se nazývá *aktivace* neuronu. Typické používané průběhy charakteristik v neuronech přináší následující obrázek č. 3 až obrázek č. 7.

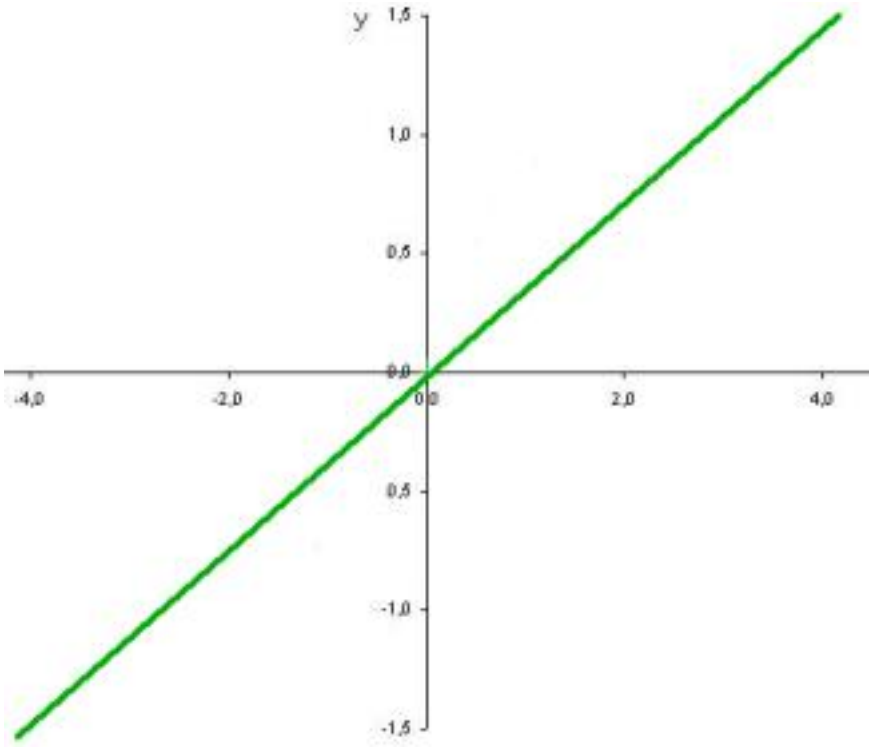


Obrázek 3 Skoková přenosová funkce

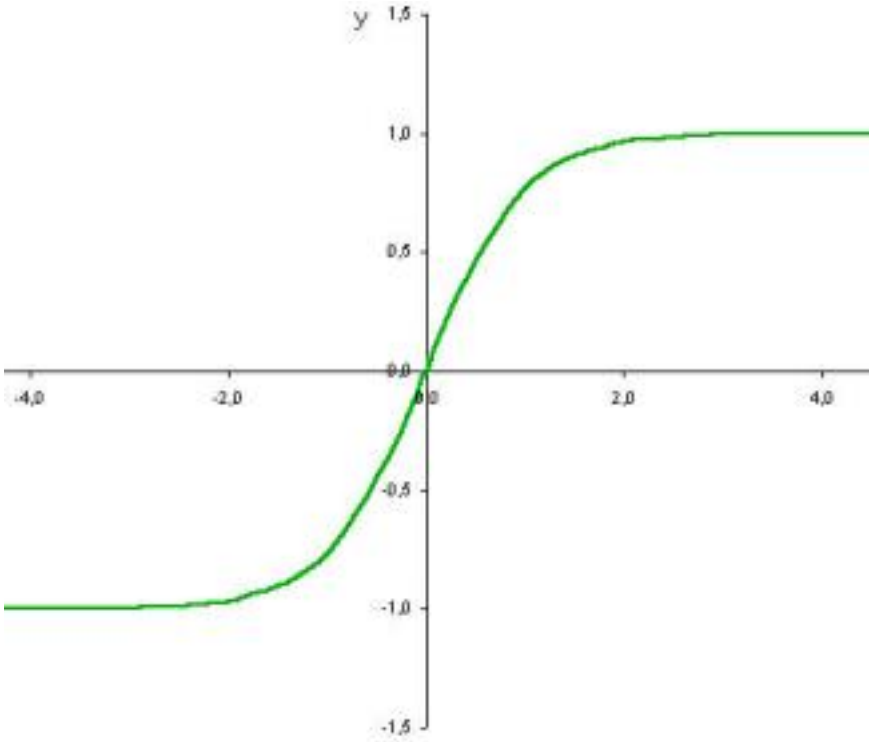
..



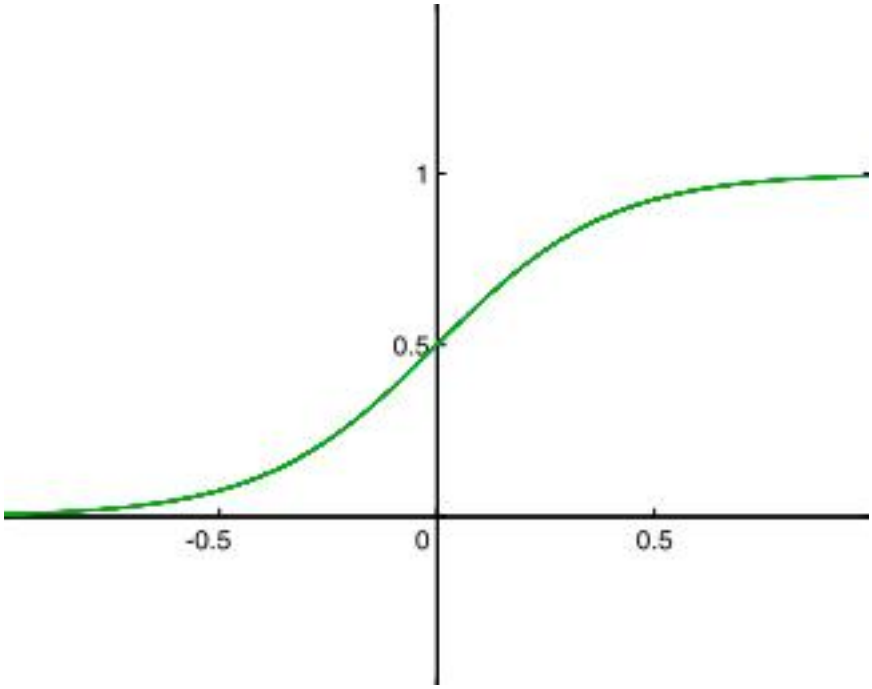
Obrázek 4 Přenosová funkce radialní báze



Obrázek 5 Lineární přenosová funkce



Obrázek 6 Přenosová funkce hyperebolické tangenty



Obrázek 7 Sigmodialní přenosová funkce

2.2 Učení NS

Řízené učení neuronu předpokládá, že existuje tzv. učební množina dvojic (x, y_d) reprezentujících požadovanou korespondenci, y_d je tedy požadovaná odezva neuronu na vstupní vektor x. Učení pak spočívá v tom, že se porovnává skutečná odezva neuronu s odezvou žádoucí a na základě toho je vhodně upravován vektor vah w.

Byla navržena řada algoritmů učení. Mnohé mají heuristický charakter. Patrně nejstarší Habbovo pravidlo pro neurony s booleovskými vstupy i výstupy, ilustrující filosofii takových postupů. Je-li y = 1, pak:

- výsledek správný → posílit váhy buzených vstupů.
- výsledek nesprávný → oslabit váhy buzených vstupů.
- Hodnoty vah nebuzených vstupů, se nemění.

Je-li y = 0, váhy se nemění.

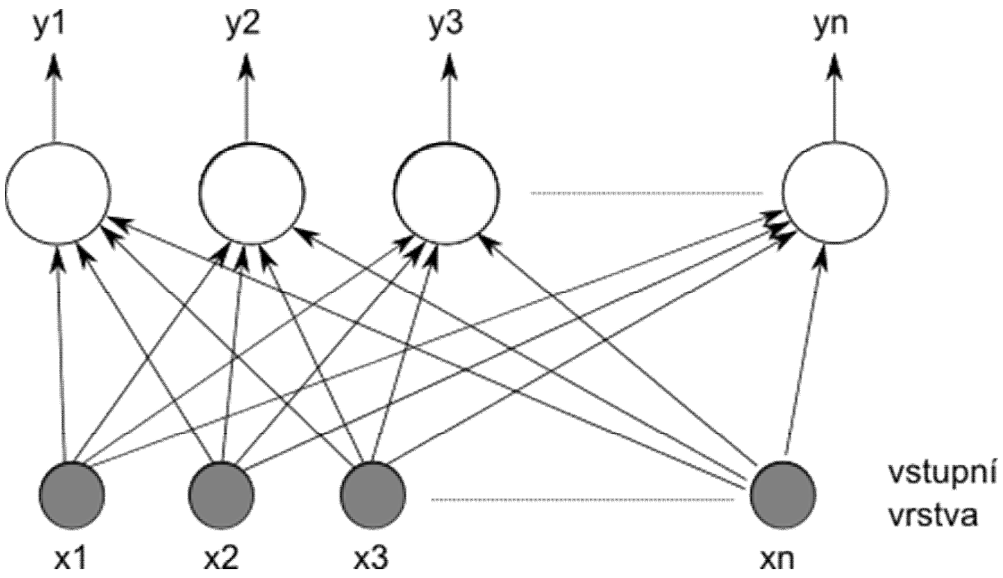
Oprava se uskutečňuje vždy po ukončení jedné epochy učení, tj. ukázání všech učebních vektorů, kdy je k dispozici součet, znamenající gradient. Vycházejíce z náhodného počátečního odhadu tak postupně získáváme zlepšující se odhady a jsou-li v poslední epoše správně zatříděny všechny učební vektory, popř. je-li podíl chybně zatříděných, dostatečně malý, iterace se ukončí. Přístup je pozoruhodný tím, že byl nalezen přesný formalizovaný postup optimalizace přesto, že charakteristika neuronu je silně nelineární a dokonce nespojitá.[4]

2.3 Jednovrstvé a vícevrstvé sítě

Koncepčně nejjednodušší neuronovou sítí je jednovrstvý perceptron. Jak je vidět na obrázku č. 8, jde pouze o M paralelně pracujících neuronů, z nichž každý nezávisle realizuje transformaci vstupního vektoru, $y_j = f({}_jw^t\, x)$, podle svého vektoru vah ${}_jw$.

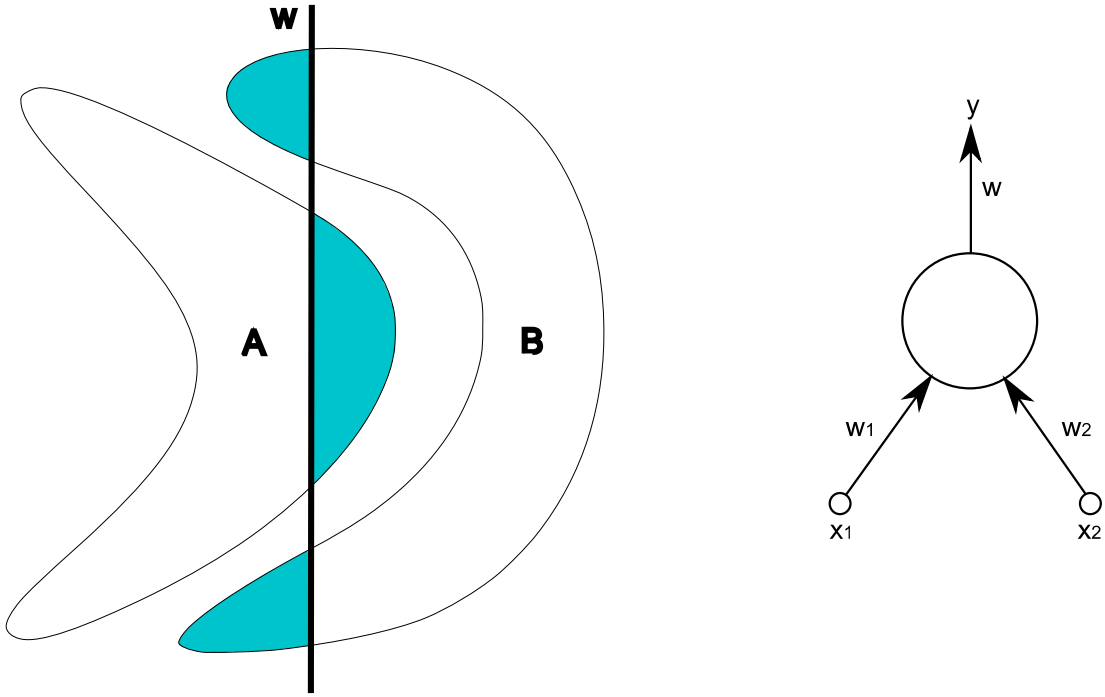
Charakteristiky všech neuronů bývají v jedné síti shodné. Je zřejmé, že taková síť realizuje nějaké zobrazení z prostoru $\mathbf{R}^n$ do $\mathbf{R}^m$. Pokud jde o neurony se spojitými charakteristikami nebo mezi stejně dimenzionálními binárními prostory, pokud půjde o neurony booleovské. U sítě spojitě pracujících neuronů je častou konfigurací kaskáda jednovrstvého perceptronu a navazující jiné sítě, která označí výstup y_k, jenž nabyl maximální hodnoty, např. při soutěživém učení to může být interpretováno za jistých předpokladů tak, že vektor ${}_k\mathbf{w}$ je optimální aproximací přirozeného vstupního vektoru **x**. Uveďme, že takto byl

zorganizován mj. Historický Rosenblattův perceptron, kde vstupy neuronů byly náhodně propojeny s množinou primárních vstupů, tzv. receptorů. V případě sítě s binárními neurony jde o realizaci booleovského M, N-pólu, jehož funkce se ovšem v průběhu učení mění.

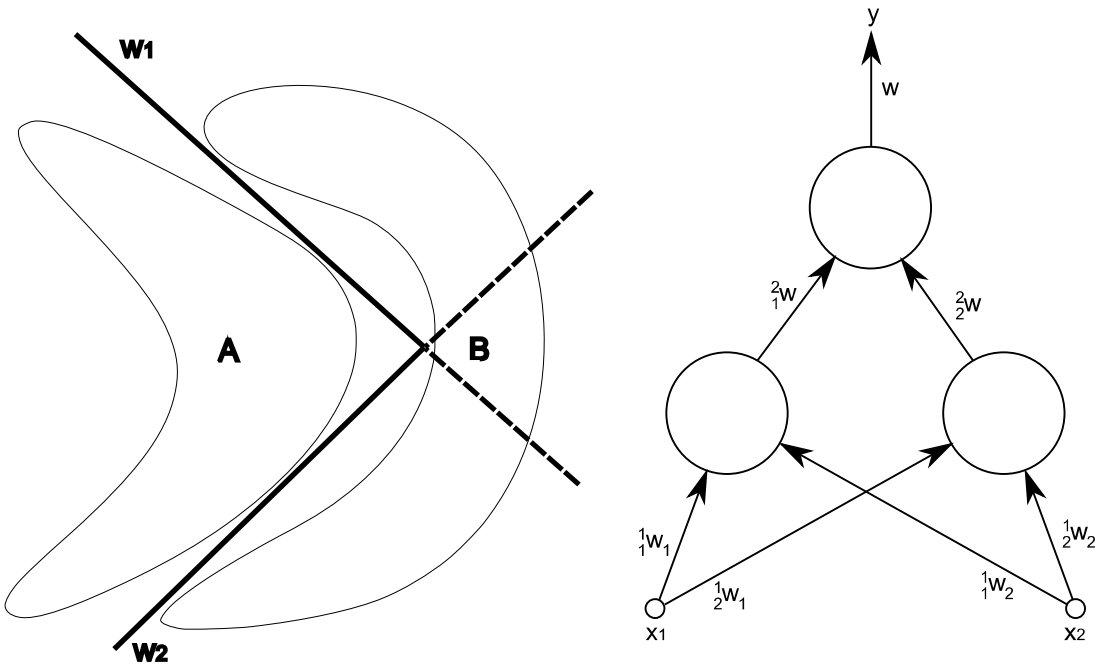


Obrázek 8 Jednovrstvý percepron

Na nejjednodušším případě obecné sítě s N = 2 a jedním binárním výstupem lze demonstrovat omezení jednovrstvého perceptronu a naopak rozšíření možnosti vrstvením. V obrázku č. 9, jež charakterizuje klasifikátor mezi dvěma množinami A a B s cílem NS je rozlišit, jestli prvek patří do B nebo do A – i při optimální poloze dělící přímky zůstávají jisté oblasti bodů, které budou klasifikovány chybně. Zmnožení neuronů ve vrstvě nepřináší samo o sobě žádoucí efekt – jedna přímka nemůže jakkoli pružně reagovat. Na dalším obrázku č. 10 je zobrazena dvojvrstvá síť, která tento nedostatek odstraňuje: součiny vytvářejí oblasti, které jsou průniky polorovin odpovídajících klasifikačním možnostem jednotlivých neuronů prvé vrstvy. Průnikem dostatečného množství polorovin lze zřejmě aproximovat libovolnou konvexní oblast.



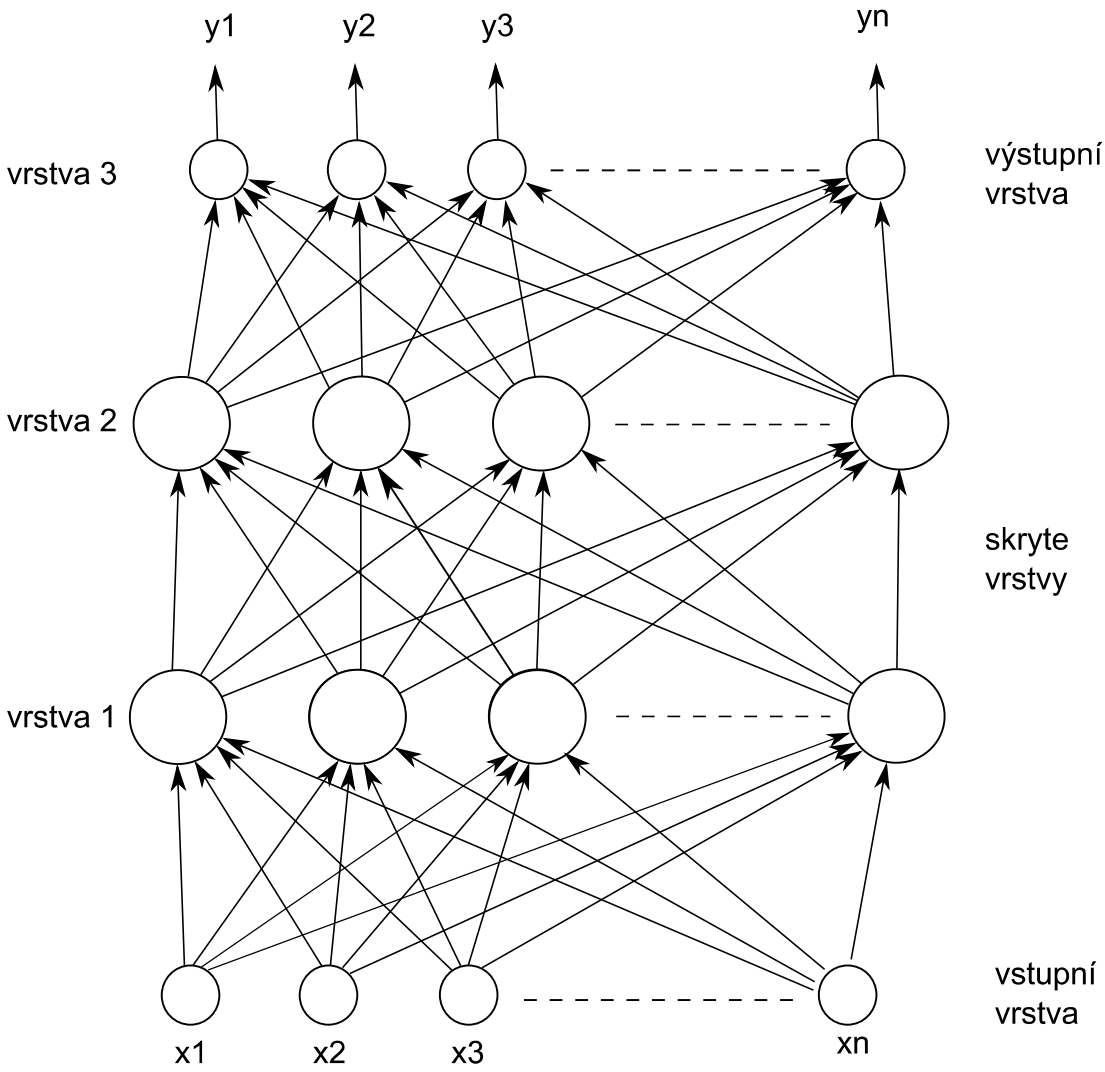
Obrázek 9 Jednovrstvá síť jako klasifikátor



Obrázek 10 Dvou vrstvá síť jako klasifikátor

2.4 Dopředné sítě

Předchozí příklad byl zvláštním případem vícevrstvého perceptronu, jehož obecnou podobu naznačuje obrázek č. 11. Můžeme zde rozlišit nejvyšší vstupní vrstvu M neuronů pod ní zvenčí nepřístupné tzv. skryté vrstvy a nejníže vrstvu N vstupních uzlů. Síť je obecně plně propojená, tedy výstup kteréhokoli neuronu (přirozeně s výjimkou výstupních) je spojen se vstupy všech neuronů následující vrstvy. Tato architektura, kterou budeme označovat jako dopřednou síť, je v praxi nejpoužívanějším typem neuronové sítě. Uvedené označení vyplývá z faktu, že signály postupují v síti pouze směrem od vstupní k výstupní vrstvě. Označení téže architektury je síť se zpětným šířením (back propagation network), které se odvozuje od práce systému. V rámci architektury jsou jednotlivé sítě odlišeny svou strukturou, tedy počtem vrstev a počty neuronů v nich.[4]



Obrázek 11 obecná dopředná síť

3 POUŽITÉ PROGRAMOVACÍ TECHNOLOGIE

3.1 C a C++

Jazyk C++ vychází z jazyka C, který je strukturovaný, relativně minimalistický, kompilovaný programovací jazyk. Tento jazyk spolu s jazykem PASCAL pomohl zavést období strukturovaného programování. Díky schopnosti adresovat hardwarové komponenty, jako jsou řídící komunikační porty a ovladače disků, se během 70-80 let jazyk C těšil velké popularitě. Mnoho vysokých škol a mnoho vládních i nevládních organizací v USA na něm budovali své projekty. Bohužel při realizaci těchto projektů použili své vlastní variace tohoto jazyka, který vznikal v Bell laboratořích firmy AT&T na konci 60let. Na tento podmět byl sestaven v roce 1983 v institutu ANSI výbor za účelem udržení vzájemné kompatibility rozdílných implementaci a vytvoření specifikací pro ně. Tato iniciativa vyvrcholila v roce 1989 normou C89 známou spíše jako ANSI C. [6]

Jazyk C++ přebírá z jazyka C tradici efektivity, účelnost, přenositelnost a rychlost. Jeho objektový přístup přináší novou metodiku programování, která je navržena tak, aby si poradila se stupňováním obtížnosti moderních úkolů programování. Jazyk C++ tak, jako jazyk C také začíná svůj život v Bell Labs a to již na počátku 80let, ale první oficiální normy (standardizace) se dočkává až v roce 1998. C++ působí dojmem dualismu, kde C++ dává jazyku C objektový přístup a tím schopnost uvést představy do spojitosti, které jsou zahrnuty v problému. C část dává jazyku C++ možnost, aby se dostal blíže k hardwaru.[7]

Možná by se dalo říci, že v dnešní době jsou tyto jazyky dávno překonány novějšími jazyky s lepší strukturou a progresivnějším přístupem k objektovému modelu, zjednodušením některých principů, ale i několik desetiletí po svém vzniku tyto jazyky stále ovlivňují IT segment ve velké míře. C++ je stále velkým hráčem na PC platformě. Je užíván tam, kde je preferován výkon. Jedná se aplikace, které jsou určeny ke zpracování, výpočtu nebo zobrazovaní určitých dat, pokud možno v co nejkratším časovém horizontu. V dnešní době jsou to z uživatelského pohledu především multimediální programy, jako hry nebo různé aplikace na zpracování zvuku a videa. Tyto aplikace jsou i přes rychlý pokrok v oblasti hardwaru stále limitovány právě rychlostí tohoto hardwaru, proto je zde potřeba velká provázanost s hardwarem a v neposlední řadě co největší optimalizace kódu. V této oblasti je stále kam pokračovat a tyto jazyky budou kráčet s nimi. Dovolím si tvrdit, že až do doby virtuální reality (pokud se tedy neobjeví nějaký výkonnější a jednodušší vývojový model).

3.2 C Sharp

Jazyk C Sharp vznikl ve vývojových centrech americké společnosti Microsoft. Vývojoví architekti vycházeli ze syntaxe C, C++ a JAVA. Na vývoji spolupracovalo také několik programátorů, kteří opustili konkurenční společnost Borland, kde se podíleli na vývoji IDE Borland Delphi a zanesli i do tohoto jazyka pár svých myšlenek. Přestože jazyk C# je součástí .NET Framework je standardizován. V současné době (duben 2010) již ve verzi 4.0.[8]

Křížek v názvu jazyka C# je odvozen z hudební notace. Označuje zvýšení noty o půl tónu. V tomto případě označuje notu cis, tedy C zvýšené o půl tónu. C# lze využít k tvorbě databázových programů, webových aplikací a stránek, webových služeb, formulářových aplikací ve Windows, softwaru pro mobilní zařízení (PDA a mobilní telefony) atd.[9]

3.3 O projektu Mono

Projekt Mono byl původně vyvíjen společností Ximian, kterou nedávno převzal Novell. Tento projekt si klade za cíl implementovat všechny klíčové komponenty technologie MS .NET, které jsou součástí standardu ECMA - překladač jazyka C#, runtime s podporou JIT, ale i těch, které nejsou standardizované, což je dost kontroverzní krok vzhledem k tomu, že existuje určité riziko neočekávaných modifikací ze strany Microsoftu, nebo dokonce patentových sporů (na území Spojených států).

Projekt již dosáhl první stabilní verze 1.0. Překladač (mcs) je napsán v C#, díky tomu je možné jej provozovat na libovolné platformě (x86, s390, SPARC, HPPA, StrongARM a PowerPC s operačními systémy GNU/Linux, FreeBSD, Windows), pro kterou existuje třeba jen interpreter IL, případně i vhodný runtime s podporou JIT. To, že je překladač napsán v C#, nepřináší jen filozofický problém, co bylo dřív, jestli vejce nebo slepice, ale taky velmi nepříjemně zvýšenou dobou kompilace. Kód generovaný překladačem je plně přenositelný a spustitelný i v jiných prostředích než jen Mono. U knihoven je bohužel situace o malinko horší, protože místy se implementace rozchází s knihovnami Microsoftu. Obvykle se jedná o extrémní případy, které nejsou nikde specifikovány a vývojáři si je po svém vyřešili. Jedná se třeba o různé zpracování konce řádků. Takovéto triviální problémy pak dokáží aplikaci odladěnou na jedné platformě diskvalifikovat pro použití na druhé platformě. Jako referenční bude asi vždy brána specifikace společnosti Microsoft. Přesto věřím, že se situace v budoucnu zlepší a tyto problémy budou minulostí.[10]

3.3.1 Rozšíření Mona

Mono obsahuje dvě docela zajímavá rozšíření. Prvním je implementace některých vlastností z C# 2.0, zejména se jedná o podporu generických typů, kterou má zatím pouze Mono. Další zajímavostí je možnost začlenit runtime do aplikací napsaných v jazyce C a tím jednoduše rozšířit aplikace o řadu nových schopností, počínaje "jednoduchým skriptováním" a konče psaním celých částí v managed kódu. Proto je zvažován jazyk C#, jako budoucí hlavní jazyk desktopového prostředí GNOME (verze 3 nebo spíše 4). Není divu - hlavní vývojář projektu Mono, Miguel de Icaza, stojí také za projektem GNOME.

Na druhou stranu Mono nepodporuje jednu z docela žádaných vlastností - generování dokumentace ze zdrojového kódu. Vývojáři to vysvětlují tím, že online dokumentace se neumí pořádně vypořádat s komentáři ve více jazycích, a proto používají vlastní systém – monodoc.[10]

3.4 Knihovna FANN

FANN je zkratka anglických slov *Fast Artificial Neural Network.* Jedná se, jak název napovídá, o knihovnu, která implementuje vícevrstvé umělé neuronové sítě. Na stránkách této knihovny se můžete dozvědět, že je naprogramovaná v jazyce C a pod open-source licencí GPL. Umožňuje funkcionalitu na více operačních systémech jako je MS Windows nebo GNU/Linux atd. Mně se bohužel nepodařilo funkcionalitu pod GNU/Linux ověřit, neboť v aktuálně dostupné verzi 2.1 při trénování neuronové sítě hlásil systém špatnou segmentaci paměti. Dále umožňuje operace jak v pevné, tak v pohyblivé řadové čárce. Obsahuje také funkci pro vytvoření neuronové sítě, její ukládání a načítaní do souboru a v neposlední řadě také funkce pro snadnou práci s trénovacími daty a jejich soubory. Je snadno použitelná, univerzální a je možné ji provázat s PHP, C++, .NET, Ada, Python, Delphi, Octave, Ruby, Prolog Pure Data a Mathematica. Referenční příručku doprovází knihovnu příklady a doporučeními, jak knihovnu používat. Grafické uživatelské rozhraní je ke knihovně také k dispozici.[11]

Z mého pohledu tato knihovna je poměrně dobře zpracovaná, ale její velká nevýhoda tkví v nevyužití jakékoliv optimalizace pro vícejádrové procesory. Největší slabost této „single thread“ architektury se projevuje při trénování neuronové sítě, protože tento proces je delší a využití všech jader v procesoru by tento proces mohlo několikanásobně urychlit. Při

realizaci praktické části této diplomové práce s touto knihovnou jsem se pokusil tento nedostatek odstranit pomocí knihovny OpenMP

3.5 Knihovna OpenMP

O knihovně OpenMP se můžeme dočíst třeba na wikipedii tyto řádky: OpenMP je tvořen soustavou direktiv pro překladač a knihovních procedur pro paralelní programování. V současné době je OpenMP standardem pro programování počítačů se sdílenou pamětí a výrazně usnadňuje vytváření vícevláknových programů. OpenMP podporuje programovací jazyky Fortran, C a C++. První OpenMP standard pro FORTRAN 1.0 byl publikován v roce 1997. Rok poté byl uvolněn standard pro C/C++. Standard verze 2.0 byl uvolněn pro FORTRAN v roce 2000 a pro C/C++ v roce 2002. Aktuální je verze 2.5, která byla jako kombinovaná pro jazyky C/C++/FORTRAN uvolněna v roce 2005.[12]

II. PRAKTICKÁ ČÁST

4 REALIZACE

4.1 Hirerarchie

Tato část bude pojednávat o realizaci praktické části. Nejprve bylo nutné celý projekt rozložit do určitých hierarchických částí, proto se celá práce skládá z pěti logických celků. Každý celek má svou specifickou úlohu, kterou se pokusím v této části rozepsat podrobněji. Správu nad jednotlivými celky je vedena programem MS Visual Studio, kde jsou vedeny jako jednotlivé projekty.

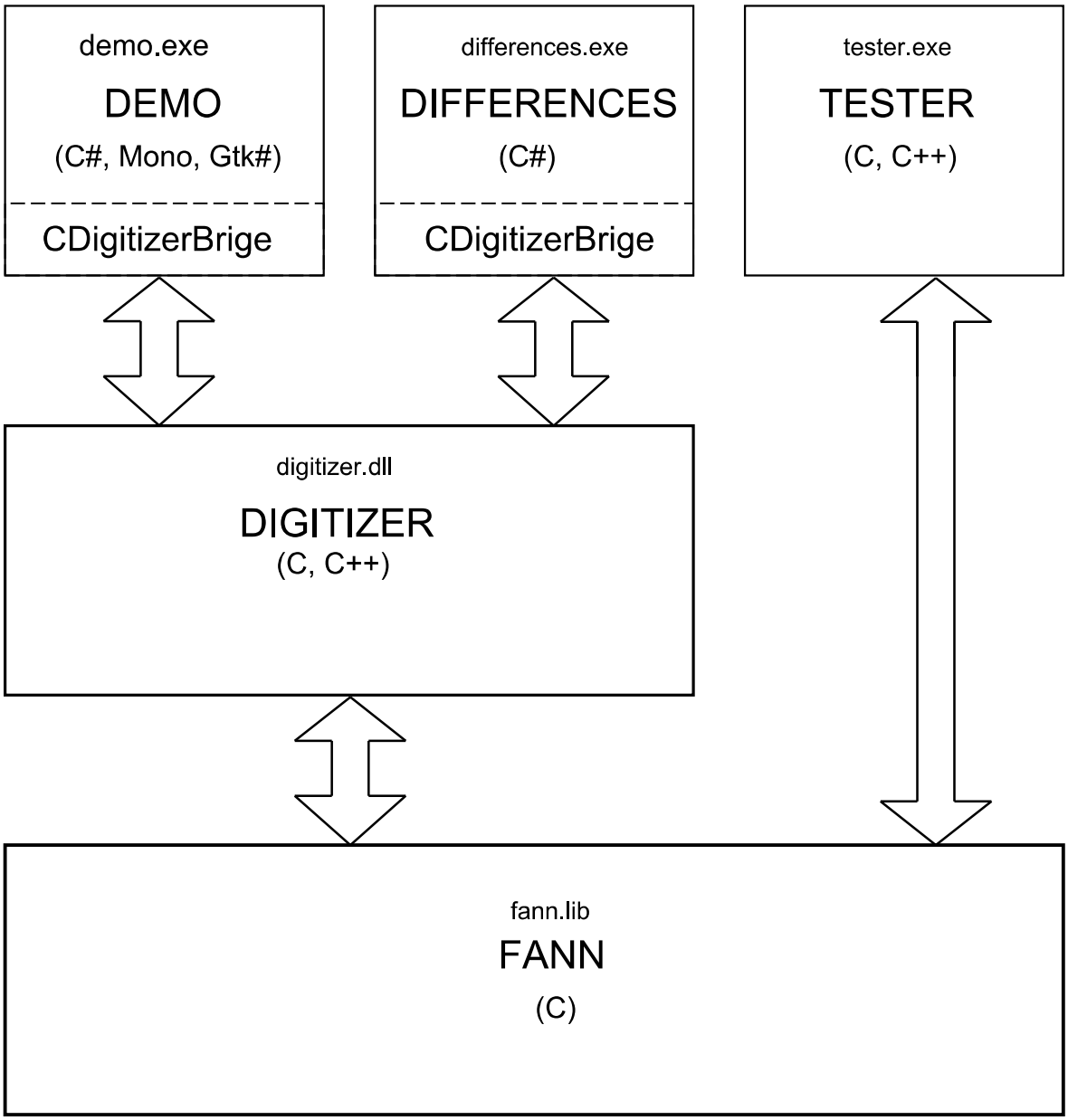
Vývojové prostředí MSVC jsem si zvolil pro jeho výkonnost, týkající se návrhu a zpracování aplikací. Za jeho kvalitou stojí dlouhé roky vývoje u společnosti Microsoft. Preference tohoto nástroje také souvisí s tím, že jej používám několik let a v současné době je také mým každodenním pomocníkem při výkonu zaměstnání.

V počáteční fázi vývoje této práce bylo také experimentováno s vývojovým prostředím MonoDevelop. Toto open-source IDE poskytuje téměř stejné programátorské pohodlí jako MSVC, včetně možnosti debuggingu naprogramované aplikace. Bohužel právě debug mód toto docela povedené vývojové prostředí sráží na kolena. V projektech jsem často pracoval s poli *integer* čísel a ty nebylo možné v debug módu zobrazit nebo dokonce debagovat. Mluvím o MonoDevelop ve verzi 2.1 je tedy možné, že v této době je k dispozici novější verze, kde je debug mód v tomto prostředí zcela funkční.

Je třeba říci, že v MonoDevelop lze výše zmíněné celky (projekty) také docela dobře zpravovat. Autoři se i zde snažili o maximální kompatibilitu s komerčními nástroji od firmy Microsoft. Bohužel i v tomto případě jsem narazil na problém kompatibility s projektem v jazyce C, kde se *.vcproj* nedokázal napojit na prostředí sub-projektu v MonoDevelop. Přes všechny útrapy považuji MonoDevelop za velice zralý produkt, který lze bez nadsázky označit jako plnohodnotná náhrada za komerční produkty, přinejmenším pro amatérský a studentský vývoj aplikací.

Zpět k úvodu a zmíněné hierarchii projektů. Hirearchije mezi těmito projekty ukazuje obrázek č.12. V horní úrovni jsou projekty, jež jsou přímo spustitelné a tudíž vázány přímo na uživatele. Jsou to *demo.exe differences.exe tester.exe* Střední vrstva představuje samotnou knihovnu pro rozpoznávání písma *digitizer.dll* a spodní část představuje knihovnu pro operaci s neuronovou sítí *FANN*.

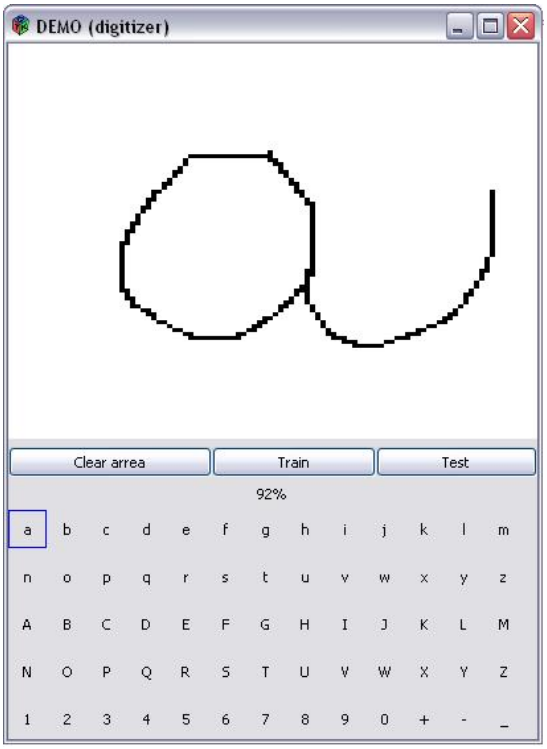
Také si můžete všimnout, že napojení programu *demo.exe* a *differences.exe* na knihovnu *digitizer* je realizováno pomocí třídy *CDigitizerBrige.* Tato třída je pouhý interface nebo chcete-li zapouzdření funkcí obsahující knihovnu *digitizer.dll.* Díky této vrstvě mohou moduly, respektive třídy, v jazyku C# přistupovat přímo k instancím funkcí naprogramovaných v C++ a přilinkovány do .dll knihovny. Dále budu v jednotlivých kapitolách obecně popisovat jednotlivé části. Nejdůležitější části jako je *demo* a *digitizér* proberu důkladněji v nesledujících kapitolách.



Obrázek 12 Projektová hierarchie

4.2 Demo

Jak název tohoto programu napovídá, jedná se o demonstrační aplikaci výsledné diplomové práce. Jejím hlavním úkolem je zpracovat uživatelský vstup v podobě obrazových dat představující symboly k rozpoznání knihovnou *digitizer.dll* a zobrazit její výstup. Dalším úkolem této aplikace je zařazování vzorku pro trénování NS. Jedná se o GUI program napsaný v jazyce C#. Tento jazyk byl zvolen pro jeho jednoduchost a lepší přizpůsobitelnost k tvorbě GUI aplikací. Konkrétněji se o tomto jazyku rozepisuji v jeho samostatném oddíle v teoretické části práce.



Obrázek 13 Spuštená aplikace demo.exe

Platforma .NET ve Windows má pro tvorbu okenních aplikací implementováno rozhraní WinForms. Autor ovšem zaexperimentoval se svobodným rozhraním Gtk#, které je součástí také svobodné platformy Mono, o které se opět více rozepisuji v teoretické části. Toto se bohužel ukázalo, jako ne příliš perspektivní cesta, protože tato technologie není tak vyzrála, jak se původně mohlo zdát a bohužel celou práci spíše zkomplikovala. Jednalo se o věci, které se oproti WinForms v Gtk# řeší poněkud krkolomněji. Další velký oříšek byla dokumentace k tomuto rozhraní. Přece jen to není tak využívaná technologie jako WinForms, a proto se mnoho internetových fór zabývající se touto problematikou odkazovala právě na WinForms.

Aplikace se spouští příkazem ***demo.exe,*** poté se uživateli zobrazí program jako vidíme na obrázku 13. Samotný program se skládá z kreslící plochy, tří funkčních a několika interakčních tlačítek. Podrobněji se rozepíši o jednotlivých funkcích v samostatné kapitole.

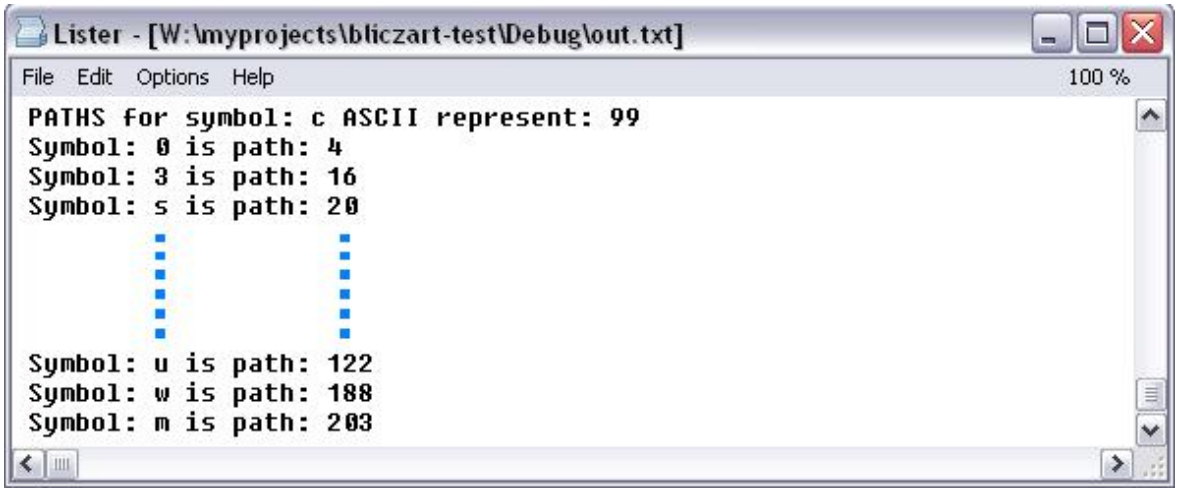
4.3 Differences

Taktéž tento program je programován v jazyku C#. Jedná se o konzolovou aplikaci *differences.exe*. Tento program má spíše výpomocný charakter, a proto se o něm rozepíši pouze v této části a dále již nebude obšírněji zmiňován. Jeho úkolem je shromáždit vzorky, které figurují v NS jako vzorové z adresaře ***symples-head***/. U těchto vzorků vypočítá (pomocí rovnice č. 1) jednotlivou podobnost vůči sobě navzájem. Výsledek je zapsán do souboru *out.txt.* Samotné výsledky přibližují, nakolik je vhodně zvolené předzpracování vzorků pro NS.

Rovnice 2 ohodnocení mezivýsledku

$$z = \sum_{i=0} |x_i - y_i|$$

O samotné předzpracovaní vzorků pro NS se stará ***digitizér,*** proto bude podrobněji popsán v kapitole o realizaci *digitizeru*. Hlavním cílem předzpracování dat je rozbor bitmapových dat na serii čísel. V tomto případě jde o šestnáct čísel, které se snaží co nejspecifičtěji charakterizovat daný symbol. Tento program tedy pouze porovnává vzorky mezi sebou a pomocí vážené sumace se snaží zjistit, jak číselně blízko jsou si vstupní vzorky. Obrázek č. 12 zobrazuje první a poslední tři vypočtené rozdíly pro symbol „c“.



Obrázek 14 ukázka souboru *out.txt* první a poslední tři vypočtené rozdíly pro symbol „c“

Jak jsem již výše zmiňoval výstupem je textový soubor *out.txt*, který obsahuje informaci o měřeném vzorku a míry shody tohoto vzorku s ostatními vzorky. Čím je číslo menší, tím větší shoda mezi symboly nastává. Obecně by nemělo dojít, aby v nějakém mezivýsledku byla obsažena nula. Tato možnost by znamenala pro neuronovou síť nemožnost rozlišit mezi těmito vzorky. Obrázek č. 12 demonstruje vzdálenost ostatních symbolů od symbolu „c“. Je zněj patrné, že nejblíže mu je symbol „0“ a nejdále symbol „m“.

4.4 Digitizer

Prozatím byli zmiňovány pouze spustitelné aplikace. Tento celek je však zapouzdřen v knihovně a není proto samostatně spustitelný. Ostatní programy, jako *demo.exe* a *differences.exe*, k němu přistupují přes rozhraní *CDigitizerBrige*, o kterém se zmiňuji v kapitole *demo*. Tato knihovna je stěžejní část celé této práce. Jde o knihovnu, která zpracovává obrazová data v podobě bitmapy a snaží se na základě natrénované neuronové sítě rozlišit, o jaký symbol a s jakou pravděpodobností jde.

Pro realizaci projektu knihovny jsem si vybral jazyk C/C++ pro jednoduché zapouzdření knihovny *FANN*, která je rovněž v tomto jazyku. Mezi jeho další přednosti patří přenositelnost a také to, že jsem v něm vytvořil již několik projektů a dobře se v něm tudíž orientuji.

Původní myšlenka vycházela z použití hrubé síly, kdy je celá bitmapa transformována do předem definované velikosti 32x32 pixelu, kde každý pixel představoval vstup neuronové sítě. Tato metoda nebyla příliš účinná, neboť při transformaci docházelo k deformaci symbolů. Dále rohové pixely byly téměř vždy nulové a nevyužité. Tyto nevyužité pixely také zabíraly místo při ukládání vzorků a bylo tak neefektivní. Tuto metodu jsem zvolil spíše z hlediska testovacího. Zajímalo mě, jak se s takovými daty NS vyrovná a jaké bude podávat výsledky. Po několika testech se potvrdilo, že tato cesta je nerealizovatelná.

Proto jsem zvolil metodu ohodnocování ručně psaných symbolů. Nějakou sérii čísel, která by co nejlépe popisovala daný symbol. Ohodnocovaný algoritmus je založen na počtu průniků virtuální přímkou, proloženou přes ručně psaný symbol. Počet těchto protnutí se ukládá jako vstup do NS. V původní verzi se ohodnocování provádí pomocí virtuálních přímek. Virtuální přímka je každá řada nebo sloupec po sobě jdoucích pixelů. Opakující se výsledky jsou brány jako jedna a tatáž přímka, a proto jsou do vstupních dat zahrnuta jen jednou.

4.5 Tester

Jednoduchý konzolový program, jehož úkolem je trénování NS na základě vstupních vzorků. Naprogramován opět v jazyku C/C++, pro jeho snadné propojení s knihovnou FANN. Knihovna FANN je využívána k vytvoření sítě, načtení trenovacích dat a nakonec k uložení natrénované sítě do souboru *digitezer.net*. Parametry sítě jsou napevno nastaveny v kódu. Nejlepších výsledků bylo dosaženo při síti s jednou skrytou vrstvou. Tato skrytá vrstva má 295 neuronů. Aktivační funkce pro vnější vrstvy byla zvolena lineární funkce a pro vnitřní vrstvy byla zvolena funkce signum. Samotný program se spouští příkazem *tester.exe*, poté je zahájeno trénování, během kterého jsou zobrazovány mezivýsledky chyby.

4.6 FANN

Tento sub-projekt, který je neodlučitelnou součástí projektu, je knihovna volně ke stáhnutí na adrese *http://leenissen.dk/fann/*. Distribuována je pod licencí GPL. Tato licence umožňuje zásah od zdrojových kódů a tím pádem modifikaci knihovny. Jak se zmiňuji v teoretické části, jejím velkým nedostatkem je nepřipravenost pro vícejádrové procesory. Jelikož celá aplikace byla tvořena na procesoru *AMD Phenom II X3*, který má k dispozici tři samostatná jádra, rozhodl jsem se tedy toho využít a provést alespoň nejdůležitější úpravy k využití všech jader. K realizaci této schopnosti jsem využil knihovnu OpenMP, která, jak se také v teoretické části zmiňuji, sdružuje soubor direktiv pro rozdělení práce pro více jader.

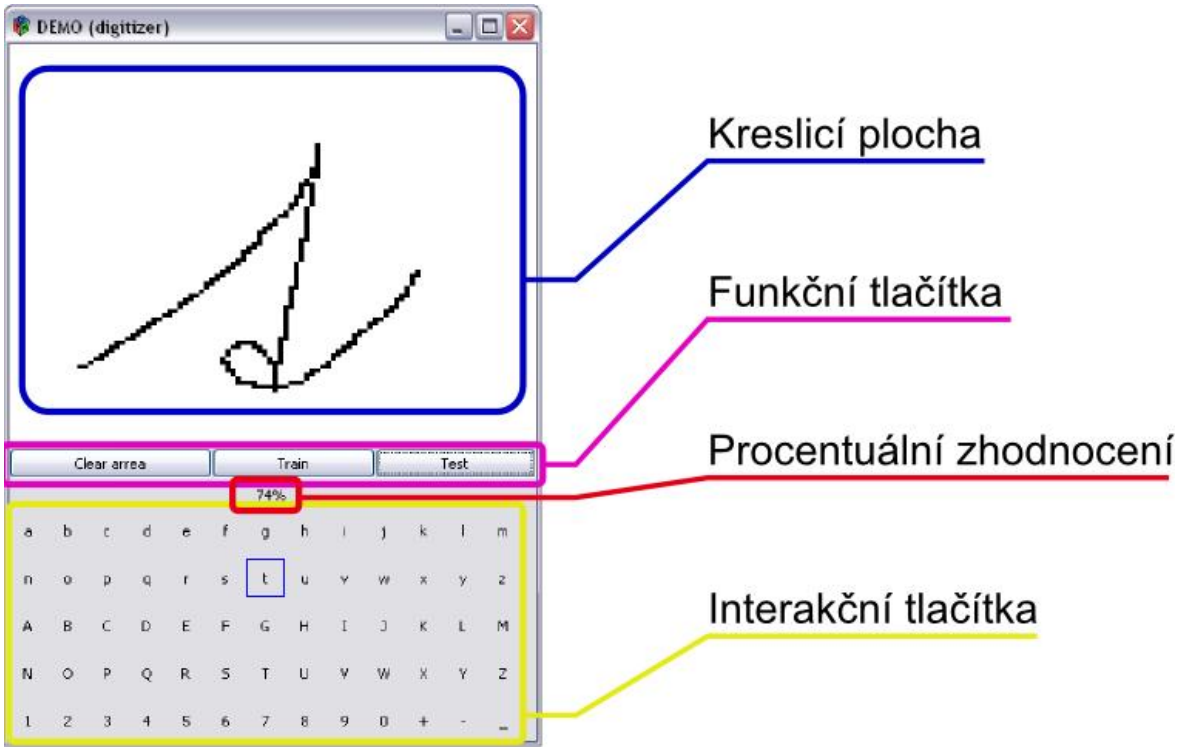
Nejdůležitější část optimalizace se nachází ve funkci *fann_propagation_MSE(struct fann *ann)*. Nejdůležitější, protože zde se nejvíce projevila celková rychlost trénování. Kompletní paralelizacice knihovny by vyžadovala mnohem více času. Bohužel do konečné fáze se mě nepovedlo provedené úpravy dostatečně odladit. Proto do finální fáze nebyla vůbec zařazena a na přiloženém CD je spíše jako rarita. Největším problémem při převodu do OpenMP je, že se autor FANN knihovny snažil co nejvíce využít adresových ukazatelů. Tyto ukazatele jsou použity také v podmínce pro ukončení cyklů. Tímto se snaží dosáhnout co největší efektivity jazyka C, ale pro nasazení OpenMP jsou v cyklech nutné podmínky numerické.

```
/* INTERNAL FUNCTION
   Propagate the error backwards from the output layer.

   After this the train_errors in the hidden layers will be:
   neuron_value_derived * sum(outgoing_weights * connected_neuron)
*/
void fann_backpropagate_MSE(struct fann *ann)
{
.
.
      first_neuron_it = layer_it->first_neuron;
      int v, max = (int)(last_neuron - layer_it->first_neuron);
      #pragma omp parallel for
            firstprivate(max,first_neuron_it, first_neuron)
            private(neuron_it) schedule(dynamic)
      for(v = 0;v < max; v++)
      {
            #pragma omp critical
            {
                  neuron_it = first_neuron_it + v;
            }
        .
        .
      }
.
.
}
```

5 DEMO

V předchozí kapitole o knihovně *FANN* jsme již naťukli implementaci OpenMP do knihovny *FANN*. Nyní je na řadě popis implementace programu, jenž se stará o vizualizaci a zpracování uživatelských vstupů *dema*. Než se však pustíme do samotné implementace, je nutné seznámit se nejprve s jednotlivými částmi programu z uživatelského pohledu. Proto si nejprve objasníme jednotlivé části programu a jeho samotné ovládaní.



Obrázek 15 Popis programu demo (s nakresleným symbolem „t“)

5.1 Ovládaní programu demo

Po spuštění programu demo se zobrazí malé okno, které je možné rozdělit do čtyř funkčních celků. Tyto celky popisuje obrázek č. 15.

5.1.1 Kreslící plocha

V této oblasti lze kreslit jednotlivé symboly. Samotné kreslení je velice jednoduché a intuitivní. Při stisku levého nebo pravého tlačítka na myši a jejím pohybem se v aktuálním bodě nakreslí pixel černé barvy. Tyto symboly jsou součásti bitmapy pro *digitizer* a tudíž zobrazují výsledný tvar bitmapy, která se bude testovat nebo vkládat jako nový vzorek pro trénování. To, zda se bude bitmapa testovat nebo se využije k trénování, určují *funkční tlačítka.*

5.1.2 Funkční tlačítka

Funkční tlačítka jsou tři. Jejich popis a funkci přibližuje tabulka č 1.

Tabulka 1 Popis funkce funkčních tlačítek

Jméno	Funkce tlačítka
Clear	Vyčistí plochu a umožní kreslení nového symbolu.
Train	Zařadí nakreslený symbol do trénovací množiny pro NS. Před samotným stisknutím tohoto tlačítka je nutné zvolit, o jaký symbol se jedná pomocí interakčních tlačítek (více v sekci Interakční tlačítka). Při stisku trenovacího tlačítka se kreslící plocha smaže.
Test	Pošle nakreslená data do digitizer.dll a zobrazí výsledek pomocí interakčních tlačítek (více v sekci Interakční tlačítka) a v pruhu pro procentuální zhodnocení zobrazí, s jakou pravděpodobností odpovídá nakreslená data nalezenému znaku.

5.1.3 Procentuální zhodnocení

Jak bylo uvedeno výše v této části okna jsou zobrazovány procentuální odhady výsledku.

5.1.4 Interakční tlačítka

Jednou z nejdůležitější části programu jsou interakční tlačítka. Tato tlačítka, jak již bylo výše uvedeno, slouží jako doplňková k trénování sítě a jako zobrazení výstupu. Těchto tlačítek je sedmdesát a každé představuje určitý symbol, kterým je označen. V pravém spodním rohu je tlačítko nesoucí symbol „_“. Pokud během rozpoznávání dojde k označení právě tohoto tlačítka, symbolizuje to, že NS daný symbol nerozpoznala a její výsledek je zcela mimo rozsah. Při označení tohoto tlačítka a pokusu trénování, se trénování neprovede! Při stisku tlačítka dojde k jeho označení. Odznačit jej lze pouze při překliknutí na jiné interakční tlačítko. Označené tlačítko lze poznat podle toho, že je modře orámováno, například na obrázku č. 15 je označeno tlačítko se symbolem „t“.

5.2 Změna chování programu Demo

Doposud jsme mluvili o programu *demo* jako o prostředí, ve kterém se dají symboly kreslit. Tento program ovšem má skrytou vlastnost, která umožňuje již nakreslené vzorky vyvolat. Toto nastavení bylo implementováno, aby nebylo nutné kreslit vzorky nové, při každé logické změně v knihovně digitizér.

Za logickou změnu považuji jako změnu ohodnocování symbolů v knihovně. To znamená, že při stejném vstupu se změnil výstup. Během tvorby tohoto programu a její knihovny byl

neustále měněn počet a metody ohodnocování, které se projevovaly mimo jiné také proměnlivým počtem výstupních čísel.

Jen zmíním, že samotná knihovna si také ukládá vstupní symboly pro pozdější trenování, ale pouze výstup po procesu ohodnocování, a proto při změně v této části kódu by bylo nutné všechna data přetransformovat podle nové logiky, což by v některých případech mohlo být realizovatelné, ale bohužel by se našlo mnohem více případů, kde by bez znalosti vstupních dat být realizovatelné nemohly. Proto si program *demo* uchovává každý vstup, který je označen k trénování.

Příkaz	Popis	Příklad
-samples_all	Speciální režim, který vyvolá zpět již nakreslené vzorky	demo.exe -samples_all
-only_char chars	Příkaz omezení výstupu pouze na zadané vzorky	demo.exe -only_char abcd

Tabulka 2 Parametry programu demo

Další z možností ovlivnit chování programu, je přiřadit mu množinu znaků, definující rozsah výstupu. V případě, že knihovna ohodnotí vstupní znak jako znak, který není v této množině, nevrátí jej jako výsledek, naopak vybere z množiny nejvíce podobný výsledku.

Jak je vidět v tabulce č. 2 k tomuto omezení dojde přidáním parametru *–only_char* a seznam symbolů, které mohou byt výstupem knihovny pro program demo. V tabulce je také uveden příklad, jež odpovída omezení výstupu pouze na symboly „a“, „b“, „c“ a „d“.

5.2.1 Vyvolání již nakreslených vzorků

Pro spuštění v tomto režimu je nutné napsat (jak ukazuje tabulka č. 2) parametr příkazové řádky: *demo.exe –sample_all*

Po tomto příkazu se program spustí a v oblasti kreslící plochy bude již zobrazen symbol. Tento symbol je načten ze složky *samples/*. Tento symbol můžete zkusit „digitalizovat“ tlačítkem *Test*, jehož funkce se v tomto režimu nikterak nezměnila. Naopak tlačítko *Train* daný symbol přidá do databáze vzorků a zobrazí nesledující symbol. V případě, že se objeví prázdná kreslící plocha, byl předchozí vzorek ve složce poslední. Vzorky jsou skládány abecedně podle jména souboru (toto setřídění děla *.NET* funkce *Directory.GetFiles*). To tedy znamená, že vzorky, jejichž ASCI hodnota je 100, budou první. ASCI kód „100“ představuje symbol „d“, tudíž budou následovat symboly „e“, „f“, „g“, …

5.2.2 Formát vstupních dat

Všechna trenovací data se ukládají v podobě souborů do adresáře ***samples/*** . Tyto soubory jsou pojmenovány podle následujícího klíče:

XXX_NAME.sample

Kde XXX představuje ASCI kód znaku. Může být buď dvojciferné nebo trojciferné číslo.

NAME za podtržítkem představuje náhodně vygenerované jméno souboru. Toto jméno má proměnlivý počet znaků.

Poslední část za tečkou „sample“ je pevná a neměnná.

Uložení dat v tomto souboru je ve formátu jak zobrazuje tabulka č. 3, kdy první tři bajty představují hlavičku a za ní následují data. Formát hlavičky je následující tabulky:

Pozice:	Velikost:	Popis:
1. bajt	1 bajt	šířka bitmapy
2. bajt	1 bajt	výška bitmapy
3. bajt	1 bajt	bitová hloubka (tato hodnota je vždy 8)
4. bajt	neomezena	bitmapová data

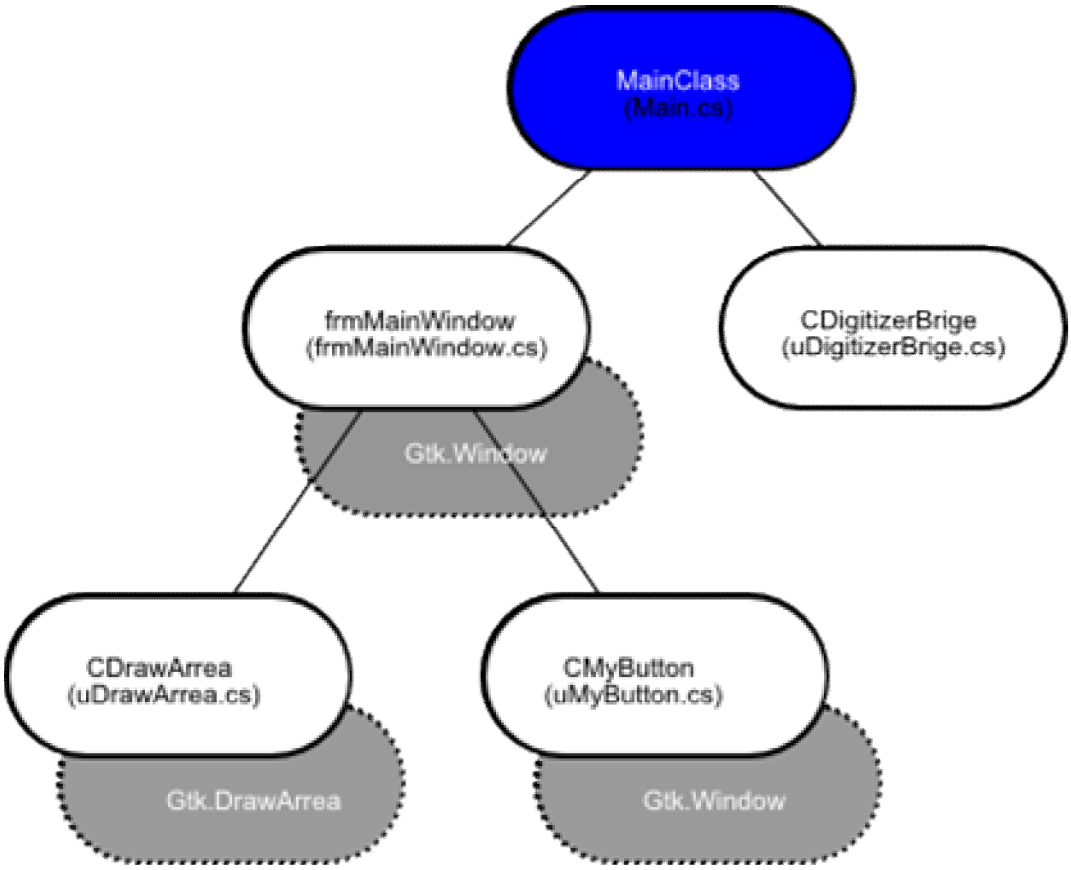
Tabulka 3 Formát souboru pro uchovávání vzorků.

Samotná data představují po sobě jdoucí čísla nesoucí hodnotu „1“ nebo „0“. Tato data pochází, respektive se ukládají, do ***booleanovského*** vektoru ***CDrawArrea.fbMatrix,*** o kterém bude řeč v následující kapitole, konkrétně o implementaci třídy ***CDrawArrea.***

5.3 Implementace programu demo

Konečně se dostáváme k implementaci programu ***demo,*** jak bylo uvedeno v části o hierarchickém rozdělení. Tento sub-projekt je naprogramován v jazyku C# a o jeho vizuální prvky jsou řešeny pomocí rozhraní Gtk#. Složení jednotlivých objektů je vidět na obrázku č. 16.

V hlavní třídě ***MainClass*** obsahuje pouze statickou funkci ***Main,*** která se spouští v jazyku C# jako první. V této funkci dochází k inicializaci ***digitizeru*** a zobrazení hlavního okna ***frmMainWindow***.



Obrázek 16 hierarchie tříd projektu demo

5.3.1 frmMainWindow

Tato třída se stará o celé okno, které reprezentuje program demo. Jejím hlavním úkolem je rozmístění komponent, jako je *CDrawArrea*, funkční tlačítka a interakční tlačítka. Zároveň u všech zachytává jejich uživatelskou odezvu v podobě kliknutí. Jejím nepřímým úkolem je také to, že připravuje data pro *digitizér* a naopak zobrazuje výsledek z *digitizeru*.

5.3.2 CMyButton

V úvodu jsem zmiňoval nedotaženost GUI rozhraní Gtk# . Nejvíce je to poznat v této části, kdy je potřeba, aby se tlačítko po kliknutí změnilo barvu. U WinForms zcela banální záležitost, nastavení jedné property a věc je hotová. Bohužel u Gtk# jsem něco takového hledal marně. Nakonec jsem celou záležitost musel řešit přetížením funkce *onExposeEvent* z *Gtk.Button*. V této přetížené funkci se vykresluje rámeček reprezentující označení na základě vnitřní proměnné *pbActive*. Tato proměnná se nastavuje na hodnotu *true*, při kliku na toto tlačítko. Naopak při stisku jiného interakčního tlačítka je nastaveno zpět na hodnotu *false*.

```
protected override bool OnExposeEvent(Gdk.EventExpose args)
{

      base.OnExposeEvent(args);
      // draw from matrix
      Gdk.Window oWin = this.GdkWindow;
      Gdk.Rectangle oArea = args.Area;
      //oArea.Right -= 2;
      // fill empty place
      Gdk.Rectangle oA = new Rectangle(oArea.X, oArea.Y,
                                       oArea.Width - 2, oArea.Height - 2);

      if (pbActive)
            oWin.DrawRectangle(Style.BackgroundGC(StateType.Active),
            false, oA);
      else

            oWin.DrawRectangle(Style.BackgroundGC(Gtk.StateType.Normal),
            false, oA);

       return true;
}
```

5.3.3 CDrawArrea

Třída *CDrawArrea* vychází ze stejného principu jako *CMyButton.* Také zde je přetížena funkce *OnExposeEvent*, kde se vykreslují tmavě body, které byly označeny uživatelem jako černé (stisknutí levého nebo pravého tlačítka na myši nad touto oblastí, více v uživatelské sekci). K zapamatování si bodů, které uživatel označil černě, slouží bitový vektor. Tento vektor se jmenuje *fbMatrix*. Jedná se o jednorozměrný vektor, ve kterém je druhý rozměr (tak aby se jednalo o matici s osami x a y) dopočítáván klasickým způsobem, kde určitý počet prvků představuje pevně dané body pro osu x a každý další jejich násobek je o úroveň zvýšena osa y. Pro přístup do vektoru demonstruje nesledující ukázka kódu z funkce *CDrawArrea.OnExposeEvent.*

```
// opakuj pro všechna x
for (int iX = 0; iX != ciMatrixResX; iX++)
      // opakuj pro všechna y
      for (int iY = 0; iY != ciMatrixResY; iY++)
      {
            // iX           - reprezentuje pozici na souřadnici x
            // iY           - reprezentuje pozici na souřadnici y
            // ciMatrixResX - šířka matice ( na x-sové souřadnici )
            if (fbMatrix[iX + iY * ciMatrixResX])
                  // pokud je pozice označena vykresli na této
                  // pozici černý čtvereček
                  oWin.DrawRectangle(this.Style.BlackGC, true,
                              iX * fiPointSizeX, iY * fiPointSizeY,
                              fiPointSizeX, fiPointSizeY);
      }
```

Je nutné zmínit, že tato třída přetěžuje ještě dvě funkce pro odchycení zmačknutí a uvolnění *OnButtonPressEvent,* respektive *OnButtonReleaseEvent* tlačítka na myši. Při zmáčknutí tlačítka na myši proběhne nastavení proměnné *fbButtonPress* na *true.* Při jeho uvolnění se tato proměnná nastaví zpět na *false*. Poslední přetíženou funkcí je funkce *OnMotionNotyfiEvent,* pro odchytávání pohybu myši. V poslední jmenované funkci se provede zápis do vektoru *fbMatrix,* podle aktuálně zjištěné pozice myši, a to pouze v případě, že proměnná *fbButtonPress* je nastavena na *true*.

5.3.4 CDigitizerBrige

Poslední stěžejní třída programu *demo*, jejíž implementace se také využívá v programu *differences*. Tato třída má dvě logické části. Jedna část, která nese data pro *digitizer*:

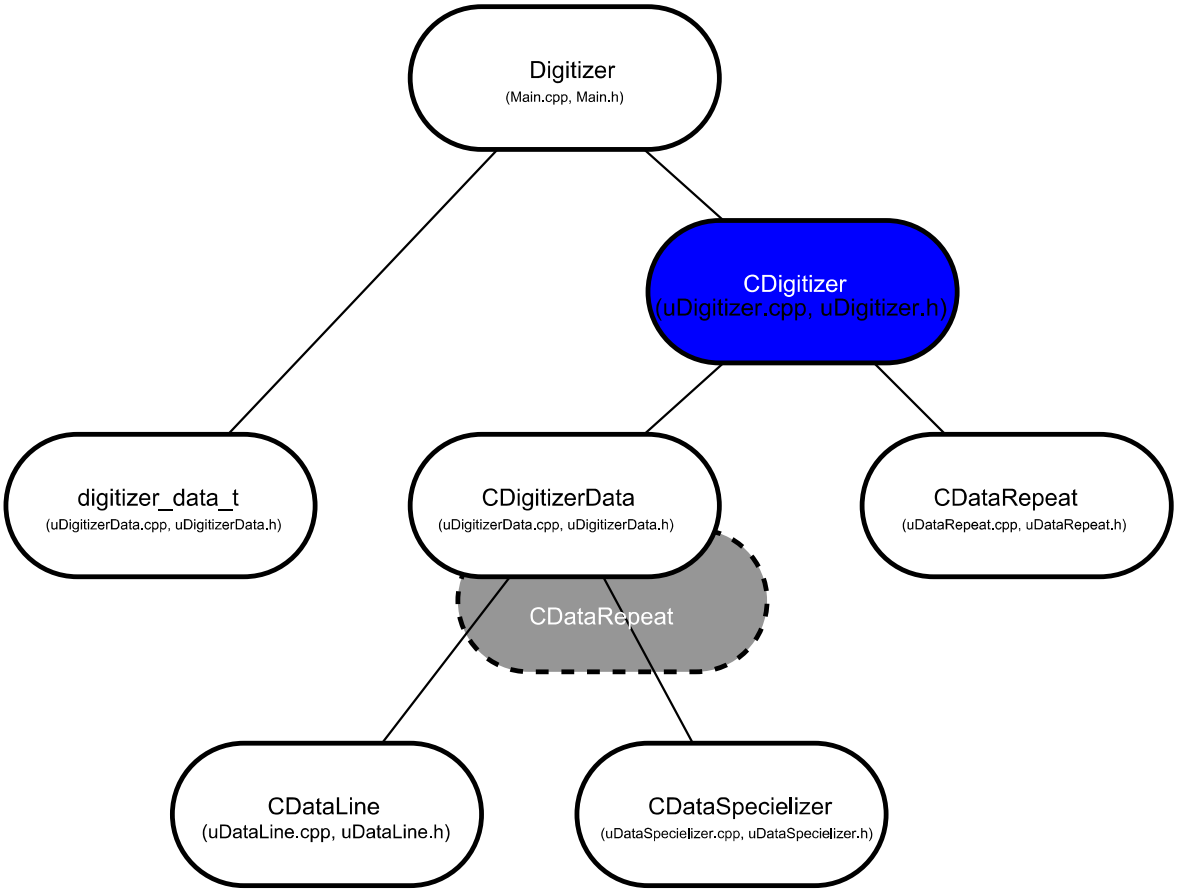
- iRexX – šířka bitmapy
- iResY – výška bitmapy
- iBpp – bitová hloubka (vždy obsahuje číslo 8)
- ppbData – vektor s daty
- fGraphResult – vrací knihovna informaci o shodě v procentech

Druhá část je souborem statických funkcí, které linkuje na sebe z knihovny *digitizer.dll*. Jsou to tyto funkce:

- gInit – inicializace knihovny
- giDigitize – funkce pro otestování vzorku
- gTrain – funkce pro přidání vzorku
- gSample – to samé jako gTrain, ale vzorek se nerozkládá standardním způsobem
- gDestroy – funkce pro uvolnění vnitřních proměnných

6 DIGITIZER

Pokud čtete tyto řádky, tak jste se prokousali do nejdůležitější části. Popisu samotné knihovny *digitizer*. V této kapitole budou popsány principy rozparsování symbolu z bitmapy do vektoru čísel popisující daný symbol. Ve finální podobě jde o sérii šestnácti po sobě jdoucích čísel. K pochopení celého procesu, jakým se tato čísla v knihovně sestavují, je nutné se seznámit s jednotlivými částmi knihovny.



Obrázek 17 schéma objektu v knihovně digitizer

Nejsvrchovanější objekt *Digitizer* na obrázku č. 17 není ve své logické podstatě objektem jako takovým, pouze zpřístupňuje funkce pro *CDigitizerBrige*, kterým jsme zakončovali předchozí kapitolu o programu *Demo.* Jednotlivé funkce a jejich popis naleznete v tabulce č. 4:

Tabulka 4 Popis přístupných funkcí

Funkce:	Popis a parametry
int gInit();	
	Inicializuje knihovnu před samotným použitím **Parametry:** žádné
int giDigitize(digitizer::digitizer_data_t * p_data, char *p_cChars, int iNumChar)	
	Převadí bitmapová data do ASCII kódu **Parametry:** p_data - bitmapa ve stanoveném formátu p_cChars - množina povolených znaků, může být NULL iNumChar - velikost množiny povolených znaků
void gTrain(digitizer::digitizer_data_t * p_data, int iResult)	
	Přidává vzorek v podobě bitmapového vzrou do trenovacích dat **Parametry:** p_data - bitmapa ve stanoveném formátu iResult - definování odezvy na bitmapu
void gSample(digitizer::digitizer_data_t * p_data, int iResult)	
	Přidává vzorek v podobě bitmapového vzoru mezi samply **Parametry:** p_data - bitmapa ve stanoveném formátu iResult - definování odezvy na bitmapu
int gDestroy();	
	Uvolní knihovnu v případě, že už nebude dále použita **Parametry:** žádné

Klíčovou roli u výše jmenovaných funkcí sehrává třída *CDigitizer*. Konkrétně funkce *gInit* a funkce *gDestroy* se stará o inicializaci této třídy, respektive její uvolnění z paměti. Ostatní uvedené funkce se pouze odkazují na členské funkce, již inicializované třídy *CDigitizer*. Vstupem a zároveň výstupem těchto funkcí je struktura *digitizer_data_t,* jejíž cílem je vytvořit stejný paměťový prostor jako má *CDigitizerBridge*. Tabulka č. 5 zobrazuje její proměnné.

Tabulka 5 vstupní a výstupní data digitizeru (digitizer_data_t)

digitizer_data_t	
Název proměnné	Popis
int piWidth;	šířka bitmapy
int piHeight;	výška bitmapy
int iBpp;	hloubka bitmapy (vždy 8)
unsigned char *ppcData;	data bitmapy (po řádcích)
float fGraphResult;	míra rozlišení v procentech

6.1 Začlenění knihovny digitizér

Pro začlenění do vlastního projektu je potřeba několik souborů, jak nám ukazuje tabulka č. 6. Jsou to zejména zejména hlavičkové soubory knihovny, které umožňují přístup k jednotlivým funkcím. Dále knihovna samotná v podobě dynamické knihovny nebo přilinkováním statické knihovny. Dva soubory ve formátu FANN, jeden nesoucí parametry neuronové sítě, druhý obsahuje dvojice výstupu ohodnocení neuronovou sítí a požadovaného výstupu v podobě ASCII kódu znaku. Pokud není připojena dynamická knihovna, je nutné také přilinkovat knihovnu FANN, pro ovládání těchto souborů.

Tabulka č. 6 Potřebné soubory pro použití knihovny digitizer.dll

Soubor/y	Význam
uCropData.h uDataLine.h uDataRepeat.h uDataSpecializer.h uDigitizer.h	Hlavičkové soubory knihovny, výchozím hlavičkovým souborem je soubor uDigitizer.h
digitizer.dll (digitizer.lib)	Samotná knihovna
digitizer.net	soubor nesoucí parametry neuronové sítě
SAMPLES.train	soubor nesoucí ohodnocené bitmapy a odkazy na jejich ASCII znak
fann.lib (a jeho hlavičkové soubory)	Knihovna FANN pro načtení neuronové sítě a souboru SAMPLES.train

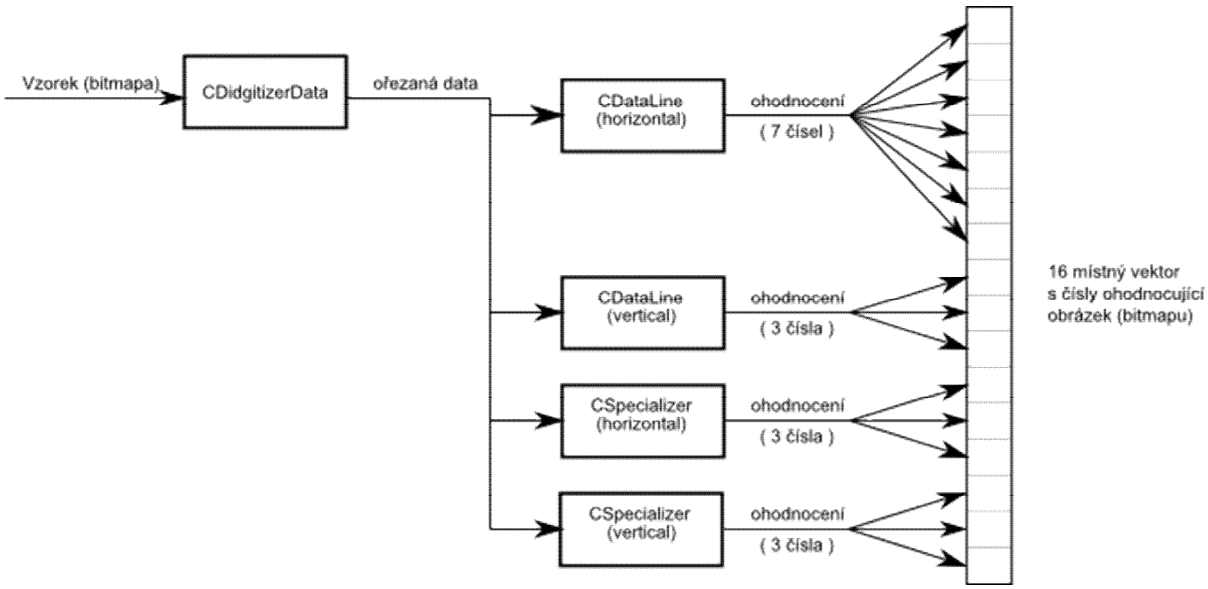
V samotném zdrojovém kódu je nutné uvést hlavičkový soubor *uDigitizer.h* a určit *namespace* na *digitizer*.

```
#include "../digitizer/uDigitizer.h"
using namespace digitizer;
.
.
// inicializace knihovny
gInit();
.
.
// otestovaní symbolu
iResultASCII = giDigitize(&oBitmap, NULL, 0);
.
.
// ukončení knihovny
gDestroy();
```

Inicializovat knihovnu funkcí *gInit* a samotné funkcí *giDigitizer,* jejíž vstupem je objekt *digitizer_data_t,* jehož popis je v tabulce č. 5. Po provedení všech prací s knihovnou je vhodné zavolat funkci *gDestroy,* ve které si knihovna provede de-inicializaci. Všechny funkce jsou popsané v tabulce č. 4. Vstup funkce a celý příklad nalezneme v příloze č. 2.

6.2 Ohodnocování bitmap

Při popisu organizace dat je nutné si připomenout, že vstupem do knihovny *digitizer* je bitmapa neboli dvojrozměrný vektor. Tento vektor byl v ranných fázích vývoje po lehkých úpravách zároveň vstupem neuronové sítě ve formátu samotné bitmapy. Princip spočíval v ořezání matice a její transformace na matici o předem dané velikosti. Experimentoval jsem s maticemi 8x8 , 16x16 a v poslední řadě 32x32. Výsledky však byly nepřesvědčivé, a proto se obrazová data ohodnocují sérií čísel. Jak tato čísla vznikají, naznačuje obrázek č. 18 se schématem vnitřních úprav. Tento postup bude v dalších schématech označován jako „*DIGITIZER*“.



Obrázek 18 schéma ohodnocováni bitmapy

Nejprve jsou data ořezána na nejmenší možný obrazec tak, aby byl jeho tvar zachován. Dále se pak algoritmus třídy *CDataLine* snaží protnout sedmi čarami, které jsou situovány horizontálně. Počet protnutí každé z čar je uložen po sobě do výstupního vektoru. Stejný princip je zopakován se třemi vertikálními čarami (směr a tvar čar bude zmíněn v kapitole o *CDataLine*). O poslední dvě operace se ve vertikálním a horizontálním směru postará *CDataSpecializer*, který rovněž bude blíže rozebrán v samostatné kapitole.

Takto získaná data knihovna uchovává dvojím způsobem:

- Základní vzorky, ze kterých ostatní vychází tzv. „samples“
- Klasické vzorky pro trénování NS, říkejme jim „data“

Vzorky nazývané „samples“, jsou ohodnocená data z pečlivě vybraných vzorků obsahující jako výstup ASCII kód (vstupem je výsledek ohodnocení *digitizérem*) znaku, který reprezentují. Každý *ASCII* znak je jedinečný. Nemůže se tedy v tomto souboru objevit více než jednou. Tato data slouží jako šablona pro ohodnocení výstupu neuronové sítě a přiřazení tomuto výstupu konkrétní *ASCII* kód.

SAMPLES

Vstup	Digitizer 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Výstup	ASCII kód 1

DATA

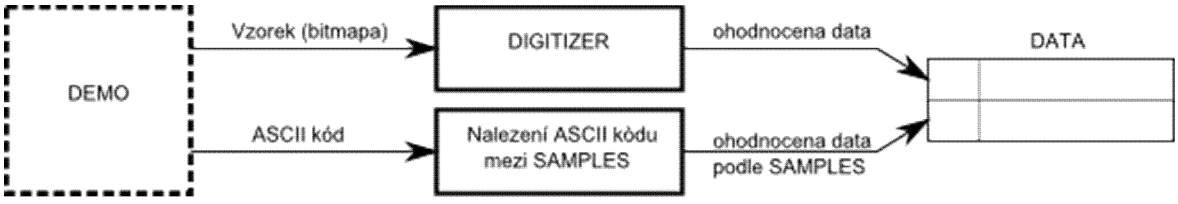
Vstup	Digitizer 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Výstup	SAMPLES.Výstup 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Obrázek 19 organizace dat u „samples“ a u „data“

Druhý mnohem obsáhlejší typ „data“ má jako vstupní data ohodnoceni *digitizérem* a jako výstupní data obsahuje vstupní data „sample“. Názorně je to vidět na obrázku č. 19. Uvedená čísla naznačují velikost vektoru ve vstupní nebo výstupní části. Vzorky „samples“ mají na vstupu čísel šestnáct a na výstupu pouze jedno, „data“ mají na vstupu i na výstupu šestnáct čísel.

6.3 Funkce *gTrain*

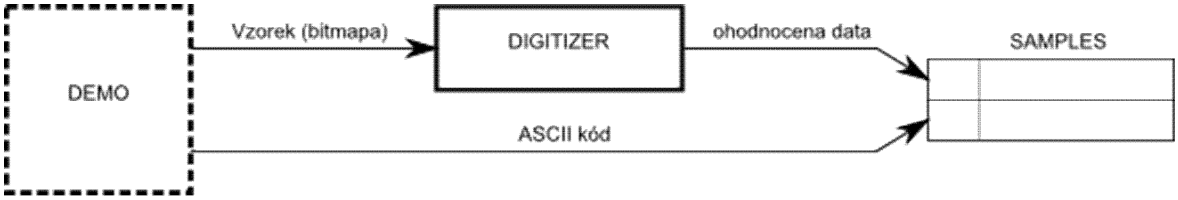
Schéma organizace dat podle funkce *gTrain* probíhá podle obrázku č. 20. Vzorek je ohodnocen *digitizérem* a tato ohodnocená data jsou přivedena na vstup. Pro výstup jsou prohledány všechny vzorky z kategorie „samples“ a vstupní data z tohoto nalezeného vzorku jsou přidána na výstup právě tvořených dat.



Obrázek 20 Schéma organizace dat podle funkce gTrain

6.4 Funkce gSample

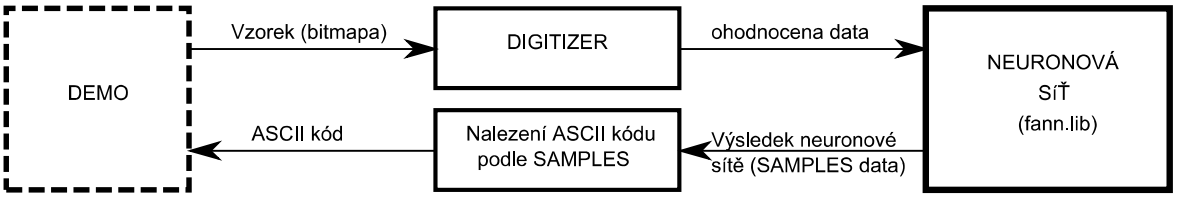
Schéma organizace dat podle funkce *gSample* probíhá podle obrázku č. 21. Jak je vidět tato funkce je oproti funkci *gTrain* zjednodušena o část prohledávaní mezi „samples“ vzorky a *ASCII kód* je zařazen přímo na výstup. Nutno dodat, že tato funkce se z programu demo nevolá. Její největší využití je programem *Differences*, protože tento se stará o zařazení klasických vzorků a vzorků jako „samples“.



Obrázek 21 Schéma organizace dat podle funkce gSample

6.5 Funkce gDigitize

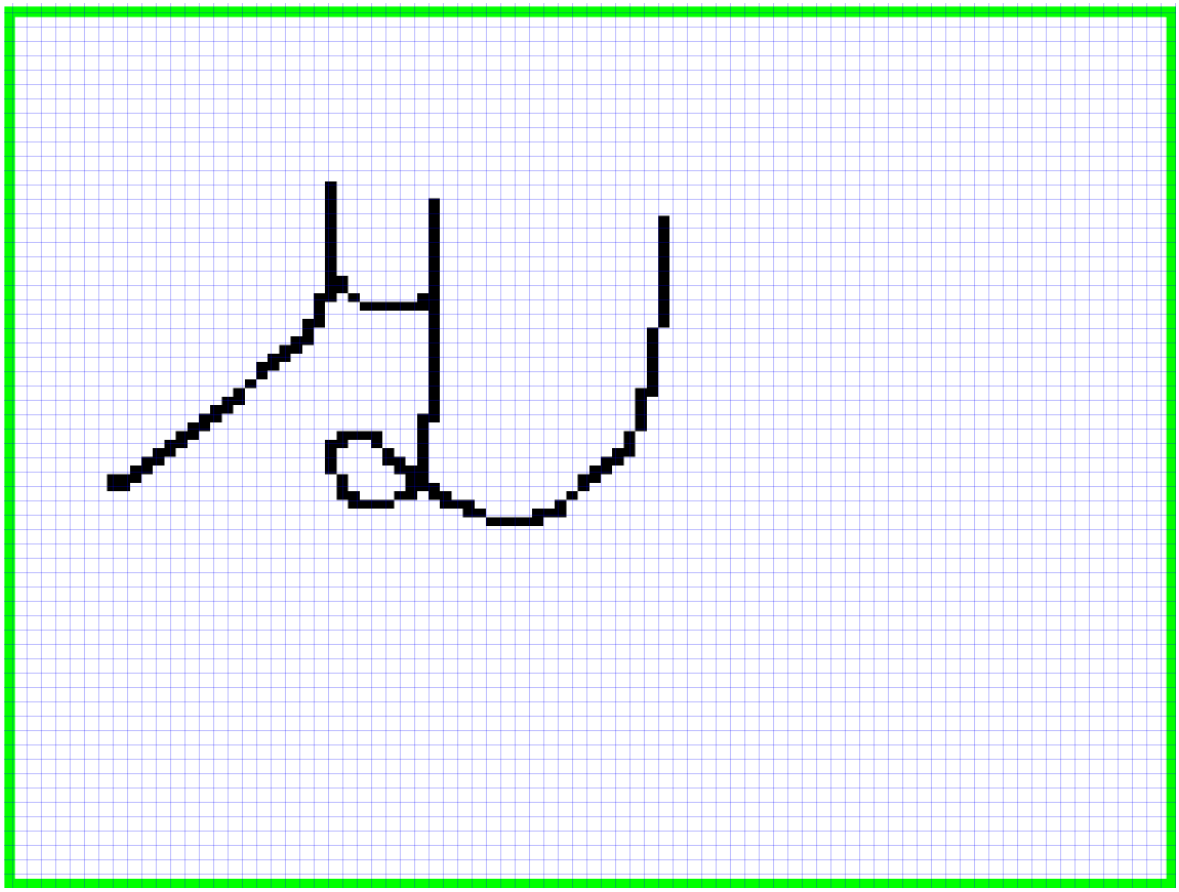
Nejdůležitější funkcí, která se z programu demo volá je nepochybně funkce *gDigitize*. Její schéma zachycuje obrázek č. 22. Hlavním rozdílem oproti předešle jmenovaným funkcím je, že ASCII kód není vstupem funkce, ale jejím výstupem. Na vstupu jsou pouze obrazová data, která se zpracují standardním způsobem v *DIGITIZERU*, jeho výsledek je poslán na otestování neuronovou sítí, která vrátí výsledek v podobě data „sample“. Pak se jen statisticky dohledá nejvíce podobný vzorek a podle něho vrátí ASCII kód.



Obrázek 22 schéma uskupení funkce gDigitize

6.6 Třída CDigitizer

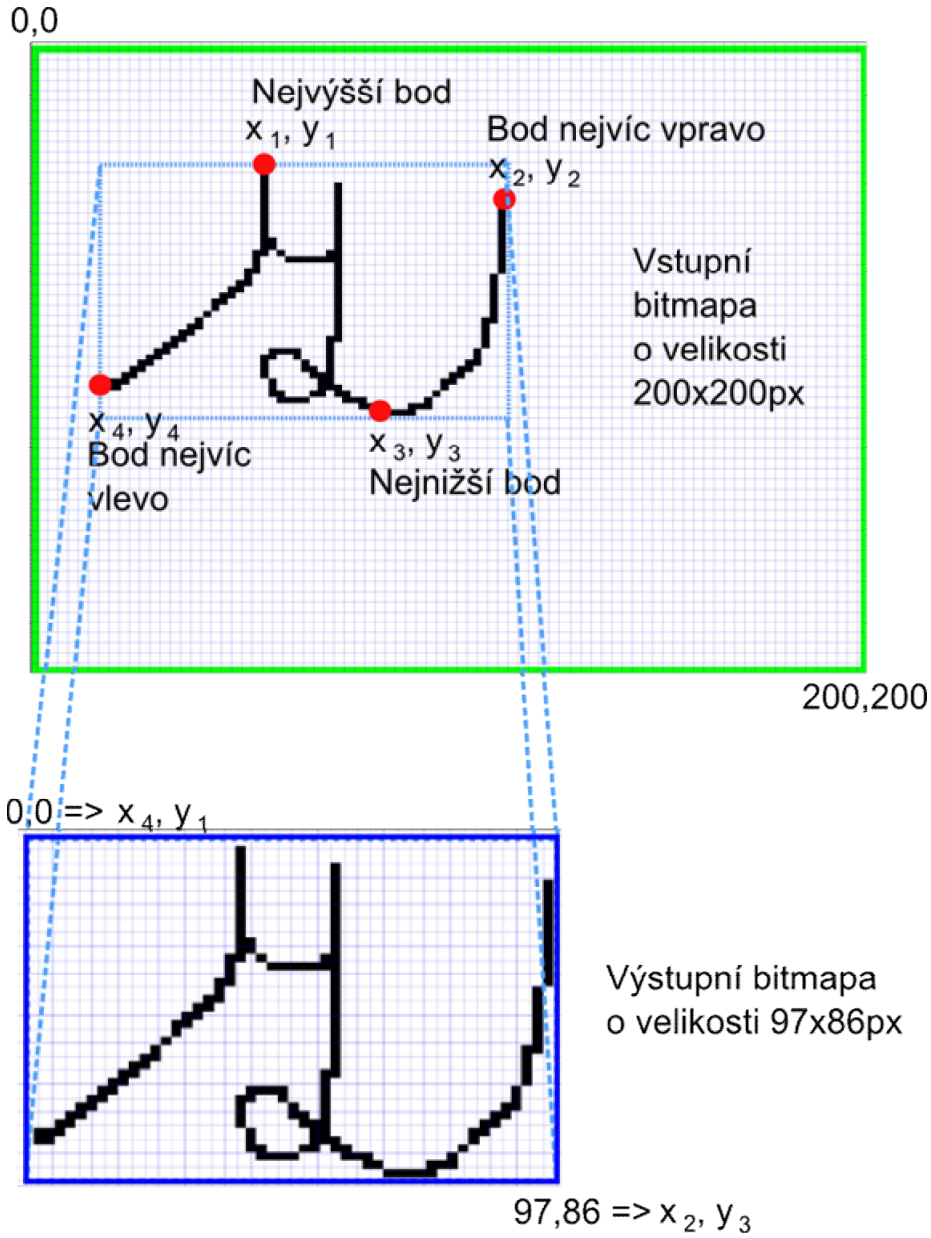
Tato třída je stěžejním bodem celého projektu, protože právně v ní dochází k přeměně bitmapových dat na data vektorová a následné otestování na neuronové síti. V kapitole o organizaci dat nastiňuji, že výsledkem digitalizace je šestnáct celočíselných číslic, ale jak tato čísla vznikají, poodhalím až teď.



Obrázek 23 Bitmapa s písmenem „z“ před ořezáním.

6.7 Ořezání matice

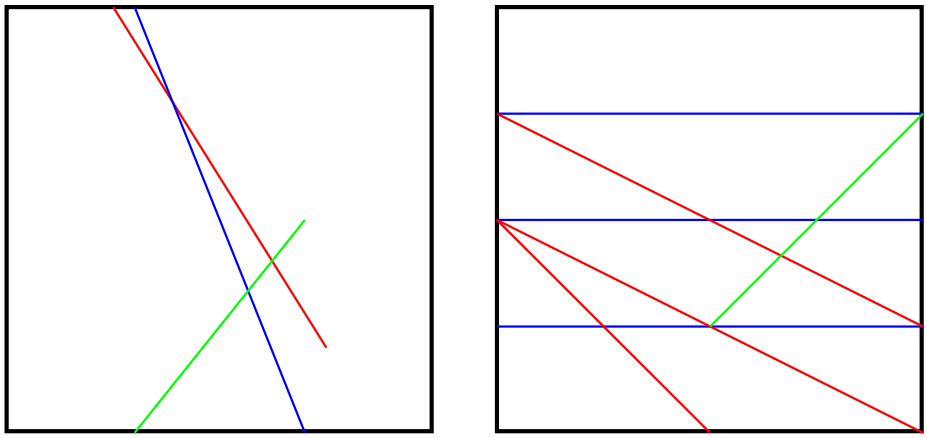
Program demo pracuje v kreslící oblasti s maticí 200x200 pixelů. Tato bitová matice je posílána nezměněná do *digitizeru*. Po nakreslení symbolu do kreslící plochy však nejsou všechny pixely využity a při posílání do NS by mohlo docházet ke zkreslení při změně pozice a zbytečné matení NS. Mějme například symbol malé „z“, mírně situovaný v levém horním rohu jak prezentuje obrázek č. 23. V první fázi jsou nalezeny nejkrajnější pixely. Jejich pozice jsou označeny pozicí *x* a *y* a indexovány *1* až *4*. Z těchto pixelů se sestaví ořezaná matice, která bude jak je vidět na obrázku č.24 mít počátek x_4, y_1 a konec x_2, y_3 Proces ořezání se provádí autonomně v konstruktoru *CDigitizerData*. Tento konstruktor má vstupní parametr *digitizer_data_t,* jež nese data z programu *Demo*. Ořezaná data jsou připravena projít ohodnocovacím algoritmem z třídy *CDataLine* a *CDataSpecializer,* o níž bude řeč v nesledující kapitole.



Obrázek 24 příklad ořezání bitmapy s písmenem „z“

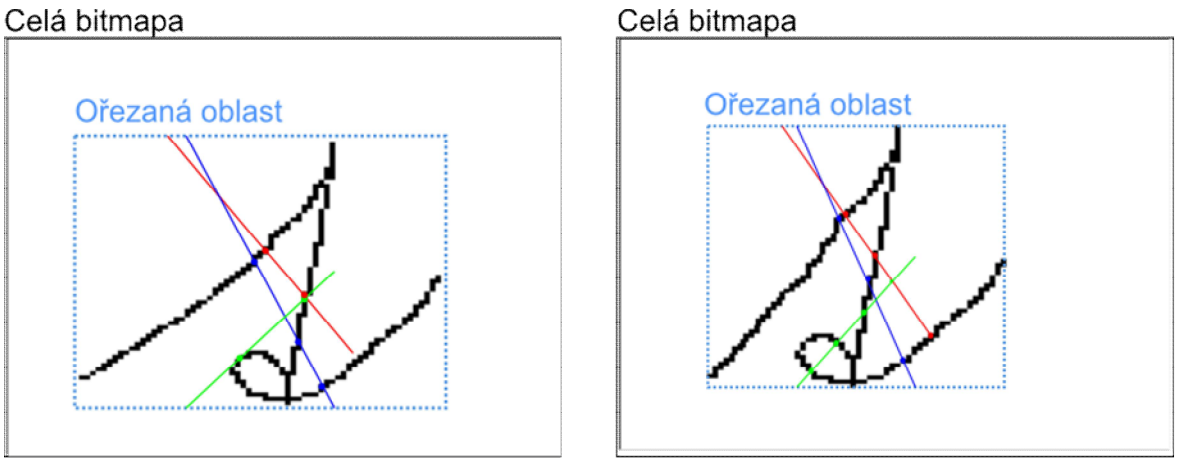
6.8 Třída CDataLine

Třída *CDataLine* má za úkol oříznutý obrázek na vstupu proložit virtuálními čarami a počet protnutí s obrazovými daty zapsat do výstupního vektoru. Jak bylo uvedeno výše těchto čar je deset (sedm horizontálních a tři vertikální). Jejich tvar naznačuje obrázek č. 25. Barevné rozlišení čar slouží jen pro lepší představu.



Obrázek 25 tvar čar v CDataLine (zleva vertikální a horizontální)

Následující obrázek č. 26 v jeho levé části zobrazuje, jak takové proložení čar může vypadat. Z tohoto obrázku je také velice patrné, jak dopadnou jednotlivé výsledky. Nejvíce protnutí signalizuje modrá čára a to tři. Červená se zelenou mají po dvou protnutích.

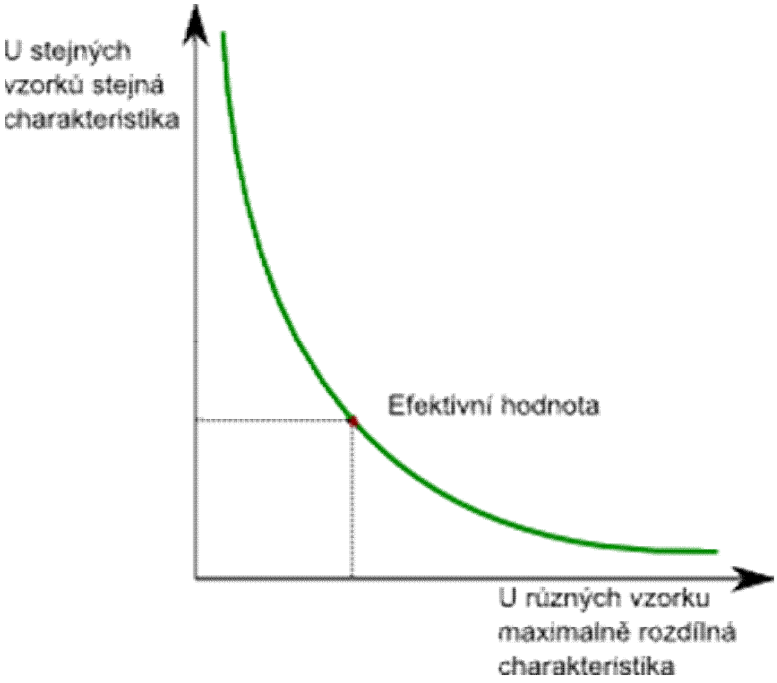


Obrázek 26 Demonstrace proložení čar (vertikální)

Bohužel tato technologie při malé změně selhává. Pro příklad vezmeme stejný symbol „t“, jen malinko upravený (v tomto případě klín, který svírá symbol „t“ trochu zostříme) a provedeme stejné proložení jako v předchozím případě. Toto srovnání nám přináší pravá část obrázku č. 26. Modrá čára signalizuje stále tři protnutí, avšak zelená a červená signalizují o protnutí víc, tedy po třech. Tyto odchylky se také projevují v horizontálním proložení a při malé změně může dojít zcela k jiným výsledkům, byť se jedná o tentýž symbol. Tato skutečnost mě velmi potrápila. Podařilo se mi ji jen částečně odstranit speciálním algoritmem. Odtud tedy název třídy *CDataSpecializer*. Než však vkročíme do vod algoritmů *CDataSpecializer,* chtěl bych se zmínit, jak tuto třídu CDataLine co možná nejlépe zdokonalit.

6.8.1 Možnosti zdokonalení

Celá třída stojí a padá na rozmístění horizontálních a vertikálních čar. Proto počet rozmístění a směr čar určuje vypovídající hodnotu výsledného vektoru čísel charakterizující bitmapu, které jsou vstupem NS. S těmito čarami bylo experimentováno spíše v počátku tohoto projektu. Později se od nich ustoupilo a ke konci byly přejaty v takovém stavu, v jakém byly opuštěny. Ale bylo by velmi zajímavé vytvořit program, který by se pokusil najít hodnotu pro počet rozmístění a směr, které by charakterizovaly vstupní obrazec co nejefektivněji.



Obrázek 27 Grafické vyjádření efektivní hodnoty.

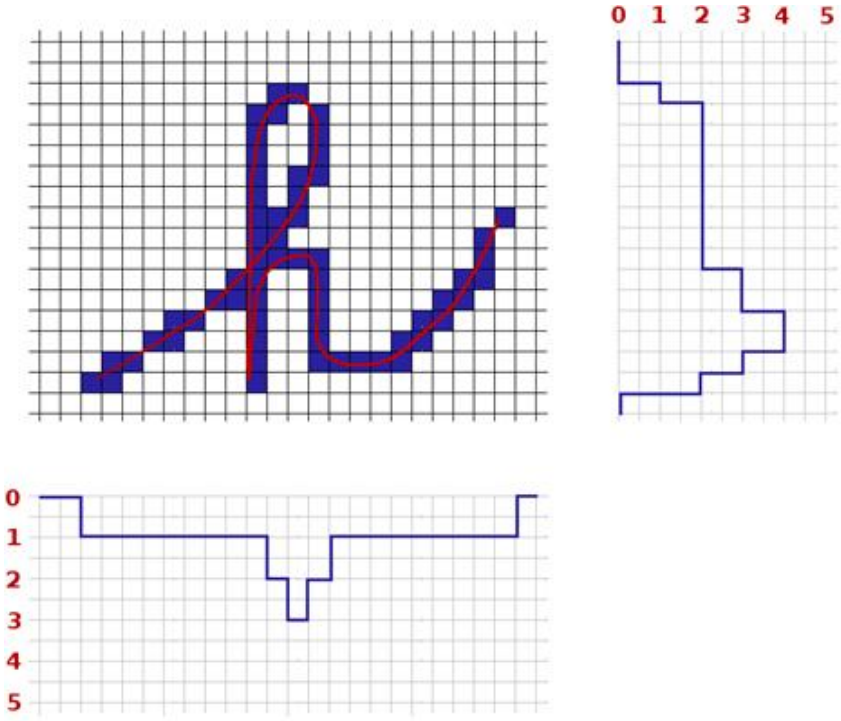
Efektivní hodnota je v tomto smyslu nejlepší poměr mezi:

- U stejných vzorků s co možná nejstejnější charakteristikou
- U různých vzorků s co možná nejrozdílnější charakteristikou

Efektivní hodnota je naznačena v obrázku č. 27. Zmiňovaný program by mohl mnohem rychleji prozkoumat všechny možné kombinace a to v reálném čase. Bohužel k sestavení takového pomocného programu, který by hledal parametry čar je časově náročné a do dne uzavření této práce se již nevešel.

Třída CDataSpecializer

Tato třída prošla bouřlivým vývojem. Jak bylo uvedeno v předešlé kapitole, malá změna může diametrálně ovlivnit výsledek. Tato třída má v základu podobný přistup jako *CdataLine*, jen neprokládá vstupní obrazec několika čarami, ale její každý řádek, respektive její každý sloupec, je proloženou čarou. Celé schéma algoritmu najdeme v příloze P1.



Obrázek 28 Grafické zobrazení parsování ve sepecializeru

Jeho funkce spočívá v projetí všech řádků (nebo sloupců podle vertikálního nebo horizontálního parsování) a na každém z nich provede součet protnutých bodů (dva body vedle sebe se počítají jako jedno protnutí). Další omezující podmínkou je, že výchylka, která se projeví jen na jednom řádku nebo sloupci se nezapisuje. Toto je zejména kvůli tomu, když uživatel píše symbol příliš rychle, tak se u souvislých čar vznikají nevyplněná místa, která by mohla značně ovlivnit výsledek. Jak takové ohodnocení může vypadat graficky ukazuje obrázek č. 28.

Tabulka 7 Přiřazení počtu proložených čar, které protnuli parsovanou bitmapu

Pozice	Hodnota protnutí:
1.	1x nebo 2x
2.	3x nebo 4x
3.	5 a vícekrát

Samotné uložení výsledku má svůj specifický formát a záleží na jeho nastavení. V části o organizaci na obrázku č. 20 se schématem seskupení dat je zobrazeno, že jsou to tři čísla z vertikálního a tři čísla z horizontálního ohodnocení.

Tyto čísla symbolizují počet proložených čar, které protnuly parsovaný obrazec. První číslo symbolizuje, jak ukazuje tabulka č. 7, počet čar, které protnuly bitmapu dvakrát nebo třikrát. Druhé číslo představuje počet čar, které protnuly třikrát nebo čtyřikrát a poslední číslo charakterizuje počet proložených čar, které protnuly bitmapu pětkrát a vícekrát.

6.9 CDataLine vs. CDataSpecializer

Rozdíl mezi těmito třídami je, že *CDataLine* podává různé výsledky při stejném symbolu, při jeho změně poměru šířky k výšce. Ukázali jsme si na příkladu symbolu „t“, na obrázku č. 26 v jeho porovnání levé a pravé časti. Na tomto obrázku se změnil poměr šířky k výšce, díky zostření úhlu v klínu písmene „t“.

Třída *CDataSpecializer* podává při této změně stejné výsledky ohodnocení. Její slabinou je bohužel jiné naklonění písmene, se kterým se lépe vyrovnává třída *CDataLine*. Součástí projektu jsou i algoritmy pro odhalení zkosení symbolů a jeho narovnání. Tyto algoritmy ale nepodávají uspokojivé výsledky a do uzávěrky DP se mi je nepovedlo dotáhnout do takové podoby, aby mohly být začleněny mezi algoritmy ohodnocující daný symbol.

7 EXPERIMENTY S NEURONOVOU SÍTÍ

V předchozích kapitolách jsme si popsali, jak data vznikají. Nyní si povíme, co s nimi. Úkolem neuronové sítě je vstupní data transformovat do takové podoby, která by odpovídala co nejvíce vzorkům v souboru „samples“. Tento výsledek bude nakonec porovnán se souborem „samples“ a vybere podle nejvíce podobného jeho ASCII kód, který je návratovou hodnotou.

7.1 Výsledky

Po dlouhém experimentovaní s neuronovou sítí bylo dosaženo nejlepších výsledků s těmito parametry neuronové sítě:

- Skryté vrstvy: 2
- Počet vstupních neuronů: 16
- Počet výstupních neuronů: 16
- Počet neuronů v 1. skryté vrstvě: 56
- Aktivační funkce u skrytých vrstev: FANN_ELLIOT
- Aktivační funkce u vnějších vrstev: FANN_LINEAR
- Trénovací algoritmus: FANN_TRAIN_RPROP

7.2 Vývoj knihovny na základě experimentů

Během mé práce jsem experimentoval s několika typy vstupu a výstupů neuronové sítě. V prvních fázích byly vstupem celé bitmapy. Bitmapa byla transformována na pevně danou velikost. Nejprve to byla velikost 24x24. Výsledky NS mě přivedly k experimentům ještě s maticemi o velikostech 16x16 a 32x32. Tyto experimentální neuronové sítě měly pouze jeden výstup reprezentující ASCII kód v podobě celého čísla. Později tento ASCII kód v dvojkové soustavě, tedy osm výstupů. Ani to však nevedlo k uspokojivým výsledkům.

Tyto neuspokojivé výsledky mě jednak vedly k malému zklamání, neboť jsem od NS očekával příliš. A jednak bylo potřeba vynaložit větší úsilí a neuronové síti práci trochu odlehčit, a to v podobě několika čísel, které by dokázaly bitmapu charakterizovat stejně dobře jako ona sama. O této charakterizující technice pojednávám v kapitole o organizaci dat.

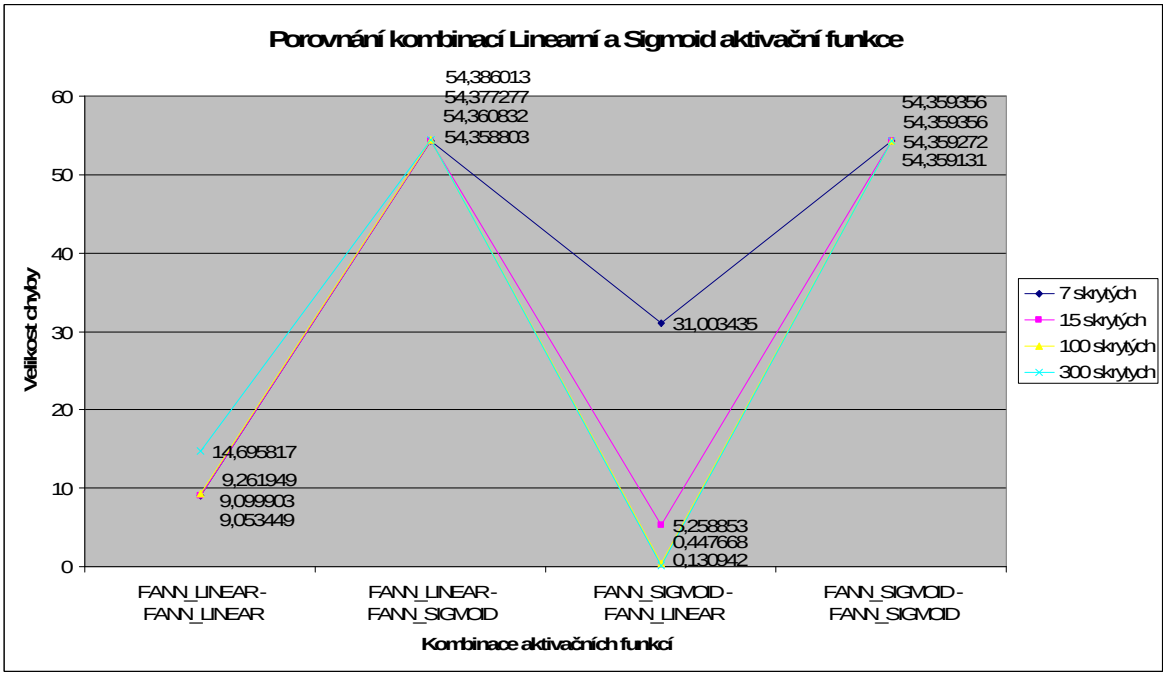
Chci ovšem zmínit, že ani tato část se nevyhnula bez experimentů. Měnil se počet vstupů (tedy charakterizujících čísel) a měnil se počet výstupů. Nejprve šlo pouze o čísla vycházející z třídy *CDataLine*. U těchto čísel se měnil počet čar a tedy výstupů (výstupů pro *CDataLine* vstupů pro neuronovou síť). Ukázalo se, že tato cesta nevede ke kýženým výsledkům. Přemýšlel jsem tedy dál. Jakou charakteristiku vymyslet, aby se nenechala zmást při malé obměně charakterizovaného symbolu. Postupnými úpravami a vylepšeními původní třídy CDataLine vznikla třída nová, nazvaná CDataSpecializer. Nejprve podávala pro charakterizaci deset výstupů. Pět pro vertikální a pět pro horizontální směr. Časem se opět projevilo, že tento model má své nedostatky, proto je v poslední fázi jejich kombinace.

Výstup u těchto zmiňovaných modelů byl kombinován jedna nebo osm podle toho, jestli byl výstupem celé číslo ASCII znaku nebo jeho dvojková reprezentace. V poslední fázi se neuronová síť pouze snaží o rekonstrukci vstupních dat na nejpravděpodobnější charakter. Tento je potom statisticky porovnán s ostatními a podle nich je vybrán výsledný ASCII kód. Konkrétnější rozbor této charakteristiky je popsán v kapitole o organizaci dat.

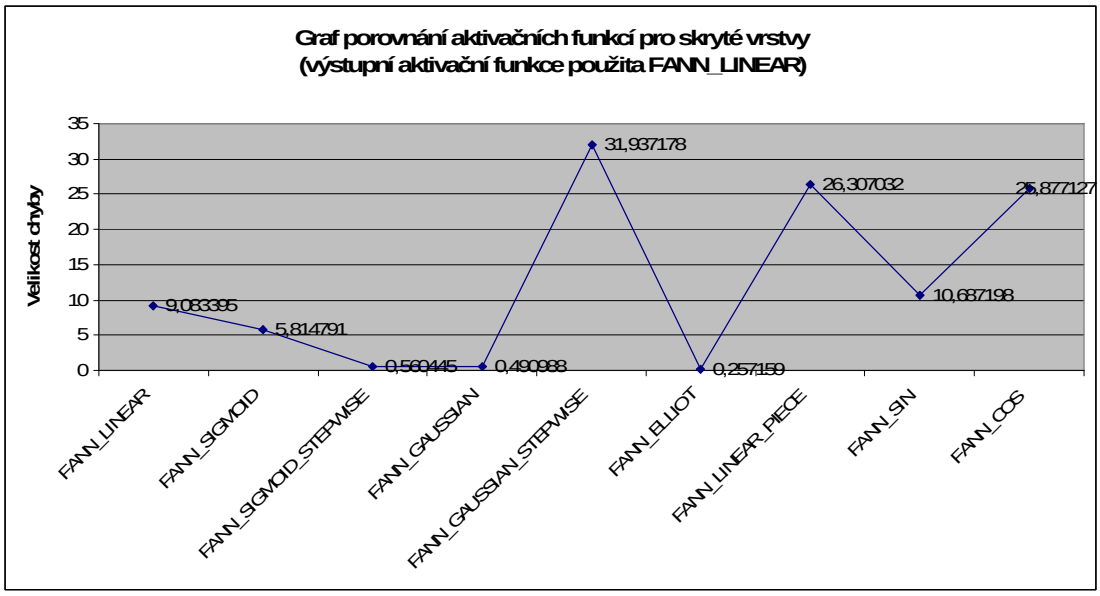
7.3 Porovnání výsledků v grafech

Při experimentech s knihovnou *fann.lib* je zobrazována hodnota chyby. Jedná se o číslo udávající průměrný rozdíl mezi jejím výstupem a jejím požadovaným výstupem. Smyslem trenování tedy bylo dosáhnout co nejefektivnější hodnoty této chyby. Příliš velká hodnota znamená nesprávné ohodnocení. Naopak příliš malá hodnota vede k přetrénování sítě, což znamená špatnou reakci na neznámé vzorky. Všechny následující uvedené grafy ukazují velikost chyby po 10 tisíci epochách.

Při trénování je nejdůležitější vhodná volba aktivačních funkcí u vnitřních a výstupních vrstev. Graf č. 1 zobrazuje vzájemné kombinace FANN_LINEAR a FANN_SIGMOID. Z výsledku vyplývá, že počet neuronů ve skryté vrstvě od dosažení určitého počtu nehraje příliš velkou roli. Lze z něj vyčíst, že nejlepší výsledky dosahuje síť s FANN_LINEAR na výstupní vrstvě. Zohlednil jsem to v dalším testování, jak naznačuje graf č. 2. Výstupní vrstva ponechána na FANN_LINEAR a na vnitřní vrstvě jsou kombinace různých aktivačních funkcí. Jak je vidět nejlepší výsledky podávají FANN_SIGMOID_STEPWISE, FANN_GAUSIAN a úplně nejlépe FANN_ELLIOT.

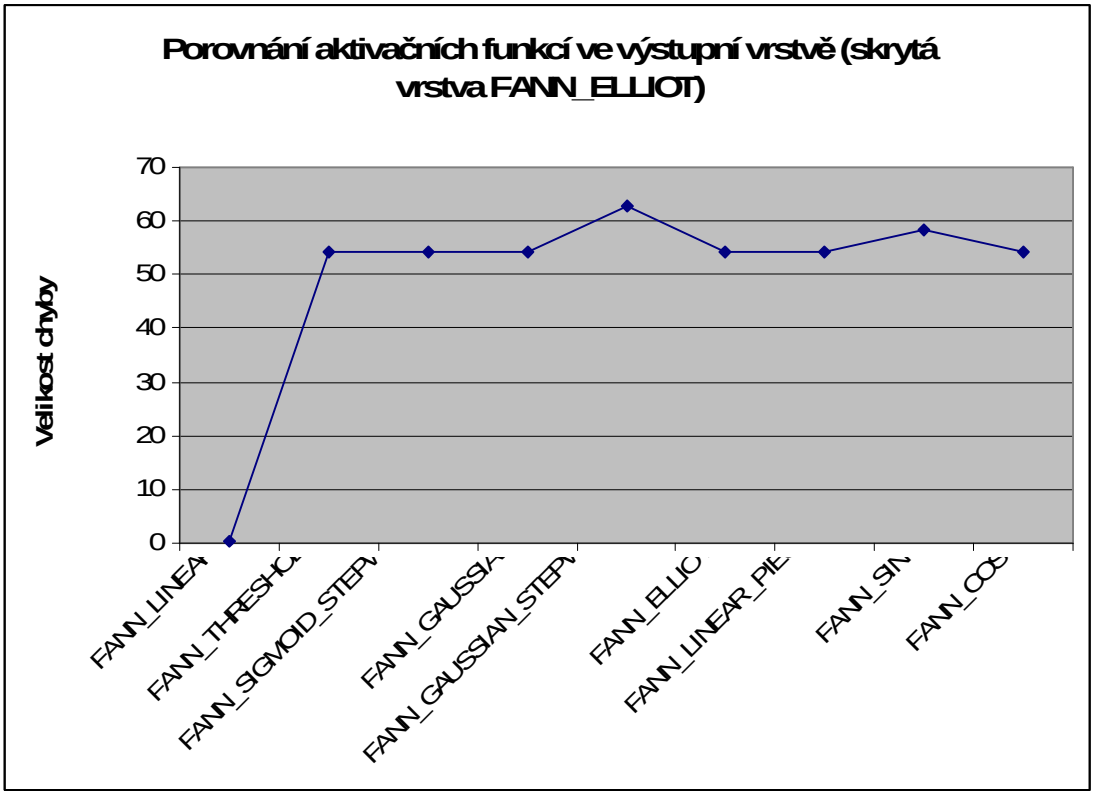


Graf č 1 vhodnost kombinací aktivační funkce



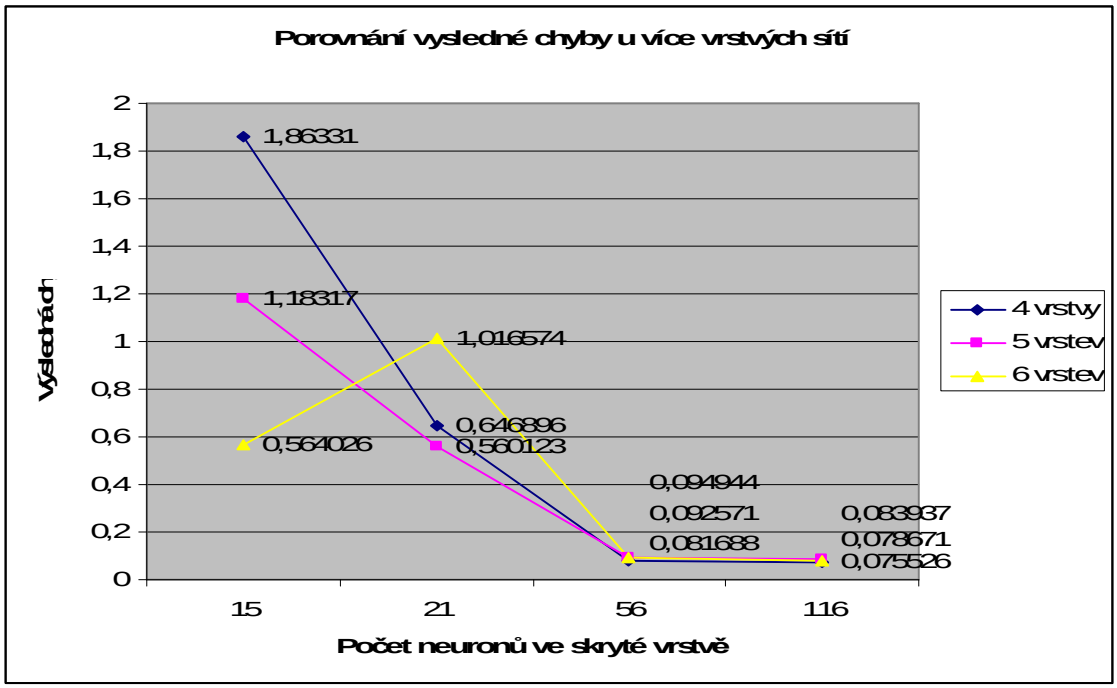
Graf 2 Porovnání různých aktivačních funkcí ve vnitřní vrstvě

Z předchozího grafu tedy vyplynulo jako nejúčinnější kombinace FANN_ELLIOT ve vnitřní vrstvě a FANN_LINEAR ve výstupní vrstvě. Graf č. 3 se snaží zmapovat, jestli není pro aktivační funkci FANN_ELLIOT jiná vhodnější aktivační funkce ve výstupní vrstvě, což se nepotvrdilo. Funkce FANN_LINEAR podává nejlepší výsledky.



Graf 3 kombinace různých aktivačních vrstev ve výstupní vrstvě

Doposud byla testována třívrstvá síť. Poslední graf nám nabízí srovnání vícevrstvé sítě tedy čtyř, pěti a šestivrstvé sítě. Tyto sítě dosahují lepšího výsledku (při stejných parametrech) než sítě s jednou skrytou vrstvou.



Graf 4 Vzájemné porovnání vícevrstvých sítí.

V předchozích grafech je srovnáváno i se sítěmi obsahující čtyři sta neuronů ve skryté vrstvě (takové trénování pro deset tisíc epoch trvalo kolem 4-5h). Podobné časové náročnosti dosahovalo trénování šestivrstvé sítě se 116 neurony. Například trénování pro šestivrstvou síť a 400 neuronů ve vnitřní vrstvě by mohlo trvat i několik dní. Proto není uvedeno srovnání se srovnatelnou třívrstvou sítí. Všechna uvedená trénování byla prováděna na pětistechdvanácti vzorcích.

7.4 Budoucnost projektu

Rozlišovací schopnost by se dala zdokonalit především v algoritmické části. Zejména nalézt co nejefektivnější pozici, směr a počet proložených čar. V tomto případě navrhuji nějaký program, který by hrubou silou otestoval bod po bodu. Nebo nějakým chytrým způsobem, kde by byl vstupem neuronové sítě opět soubor bitmap, ale výsledkem by byla právě pozice, směr a počet proložených čar. Nemalé zlepšení by jistě přinesl i odladění algoritmu pro narovnání zešikma psaných symbolů. Takto narovnané symboly by byly lépe ohodnocovány algoritmy ve třídě *CDataSpecializer*. Nejlepší výsledky by mohly přinést algoritmy, které by bitmapová data převedla na vektory, neboli na soubor čar a křivek, a na jejich základě ohodnocována.

I na straně neuronové sítě by mohlo dojít ke zlepšení při kompletní úpravě na multi-jádrové procesory. Celkový trénink neuronové sítě by se značně urychlil a bylo by možné pružněji hledat vhodné parametry.

Při prvopočátcích tvorby tohoto projektu jsem měl několik vizí. Mezi ně patřilo především napojení této knihovny na webový server, jako je např. *Apache,* a v rámci něho napsat nějakou webovou aplikaci a celý finální produkt tak nabízet online. Jednalo by se o stránku, na kterou by se poslal nascanovaný text ve formě obrázku a výsledkem by byl elektronický, počítačově editovatelný text. K takovému cíli od rozpoznávání jednotlivých symbolů do rozlišení celé stránky textu je však cesta ještě příliš daleká…

ZÁVĚR

V zadání této práce bylo zpracování literární rešerše na téma rozpoznávání ručně psaných symbolů. Tato rešerše se nachází v teoretické části práce. Začíná historií rozpoznávání symbolů obecně až po použití nejčastějších technik. Dále je v teoretické části rozepsána principielní funkce neuronových sítí. Závěr této části je věnován programátorským technikám použitých v praktické části.

Praktická část je převážně zaměřena na tvorbu knihovny, jejímž úkolem je rozpoznávání ručně psaných symbolů z bitmapy. Nejvíce prostoru je věnováno popisu algoritmů uvnitř této knihovny. Jejich úkolem je rozpoznat symbol a míru správnosti na základě bitmapové předlohy. Součástí vstupu je také podmnožina ASCII znaků, ze kterých je možné symbol vybírat.

Tato knihovna je ve formě MS Windows DLL, proto je součástí práce také popis jejího rozhraní, na kterém se dá knihovna snadno začlenit do jiných programových projektů.

Původní myšlenkou bylo vytvoření také knihovny využívající rozhraní *OpenCL*, která umožňuje paralelní výpočet nejen na CPU ale také za využití grafické karty. Složitost a časová náročnost takového úkolu by však sama o sobě vydala na diplomovou práci, proto jsem se poohlédnul po jednodušším řešení. V tomto případě knihovna *fann*, která zapouzdřuje část, ve které se pracuje s neuronovou sítí. Je naprogramována v jazyku C/C++, proto i celá knihovna je v jazyku C/C++, aby bylo zapouzdření knihovny *fann*, co nejjednodušší.

Nemálo času zabral samotný návrh knihovny. Tento návrh představoval rozvržení tříd a uskupení dat uvnitř knihovny. Postupem času se měnil z jednoduché struktury do poměrně organizovaného systému tříd. Úkolem těchto tříd je ohodnotit vstupní data v podobě bitmapy obsahující symbol, sérií čísel, která tento symbol nejlépe popisuje.

Největším úskalím mé práce ovšem bylo nastavení neuronové sítě tak, aby podávala rozumné výsledky na dané vstupy. Jak se ukázalo, neuronová síť není opravdu všemocná a i přes její, z principu jednoduchý návrh, je obtížné dosáhnout takové konfigurace, která podává dobré výsledky.

Během mé práce jsem experimentoval s několika typy vstupů a výstupů neuronové sítě. V prvních fázích byly vstupem celé bitmapy. Další experimenty probíhaly pouze sérií čísel, jež původní bitmapu nějakým způsobem charakterizovala. Způsob charakterizace a počet

charakterizujících čísel se během vývoje měnil. Výstupem neuronové sítě byl povětšinou ASCII kód rozpoznaného symbolu. Tento výstup byl v posledních fázích upraven tak, že se pouze snaží o rekonstrukci vstupních dat na nejpravděpodobnější charakter. ASCII kód je poté získán až z výsledného charakteru.

Úkolem DP bylo také vyhotovení programu pro prezentaci funkčnosti knihovny. Přestože je v zadání diplomové práce pouze prostředí MS Windows, rozhodl jsem se demonstrační program vytvořit v rozhraní Gtk# na platfomě Mono. Tato platforma umožňuje kompatibilitu se systémem GNU/Linux. Bohužel, jak se později ukázalo, nebyla to dobrá volba a vývoj programu spíše zkomplikovala.

Pokud jde o hodnocení funkčnosti platformy Mono jako takové, dovolím si tvrdit, že i přes všechna úskalí, které jsem s rozhraním Mono a její knihovnou Gtk# měl, se mohou stát základem nejednoho komerčního projektu. Jako velice dobře zvládnuté je podle mne i IDE prostředí MonoDevelop. Toto prostředí si sice ještě v komerčním sektoru nedovedu představit, ale jako základ nějakého menšího amatérského projektu nabízí v poměru cena/výkon velice povedený nástroj.

Výsledkem mé práce tedy je *.dll* knihovna, jejíž cílem je rozpoznat symbol z bitmapové předlohy. Tyto předlohy jsou ohodnoceny algoritmy, založené na principu počtu protnutí čarami různé délky, umístění a směru. Během mé práce jsem se seznámil s technickými neuronovými sítěmi. Jejich úkolem bylo samotné rozpoznání symbolu na základě ohodnocené předlohy.

Mezi výhody mé knihovny jsou zejména v jednoduchosti zapouzdření do jiného programového projektu. Změnou souboru s neuronovou sítí (*digitizér.net*), lze snadno změnit charakter výsledků bez zásahu do kódu knihovny.

Mezi současné nevýhody bohužel patří slabá rozpoznávací schopnost. Je to způsobeno převážně ohodnocovací schopností vnitřních algoritmů, kdy při stejném symbolu může dojít k jinému ohodnocení. Tyto algoritmy však mohou být ještě vylepšeny, zejména zvolením vhodného počtu, umístění a směru protínacích čar nebo narovnání sešikmeného symbolu do svislé pozice. Svým velkým dílem se na výsledku také podílí neuronová síť. U ní záleží na nalezení co nejvhodnější konfigurace. Tyto úpravy a vylepšení mohouou být nadále součástí dalšího výzkumu.

ZÁVĚR V ANGLIČTINĚ

The assignment of the thesis was to work on a literary search for the topic Recognition of hand-written symbols. This search can be found in the theoretical part of the thesis. It starts with the history of recognition of symbols generally and continues up to the use of the most frequent technologies. Further in the theoretical part, the function of neuronal nets is described. The conclusion of the thesis deals with programming technologies used in the practical part.

The practical part is aimed mainly to the creation of a library and its role to discern hand-written symbols from a bitmap. The largest part deals with the description of algorithms within the library. Their role is to discern the symbol and the extent of the correctness on the base of a bitmap template. The input includes also the subset of ASCII symbols, of which a symbol may be selected.

The library is in the form of MS Windows DLL and that is why the description of its interface on which the library can be easily integrated in other programme projects is also included in the thesis.

The original idea was to create also a library using the interface *OpenCL* which allows a parallel calculation besides CPU also on the graphic card. The complexity and time demands of such a task would be worth of a thesis itself and that was the reason why I looked for a simpler solution. In this case it is the library ***fann*** which capsulizes the part in which the neuronal net is dealt with. It is programmed in the language C/C++ and that is why the whole library is in C/C++, so that the capsulization of the library ***fann*** is as simple as possible.

The design of the library itself took a great deal of time. The design comprised allocating classes and grouping the data within the library. In the course of time it evolved from a simple structure into a rather organised system of classes. The role of the classes is to evaluate the input data in the shape of a bitmap including the symbol, a series of numbers which best describe the symbol.

The worst pitfall of my thesis was setting the neuronal net so that it provides reasonable results to the given inputs. As it turned out, the neuronal net is not omnipotent at all and in spite of its technically simple design it is complicated to achieve such a configuration which brings good results.

During my work I experimented with several types of inputs and outputs of a neuronal net. In the first stages, the inputs comprised of whole bitmaps. Other experiments continued only through series of numbers which were characteristic for the original bitmap in some way. The way of characterization and the number of characterizing numbers changed during the process. The output of the neuronal net was mostly an ASCII code of a discerned symbol. The output was modified in the last stages so that it only tries to reconstruct the output data to the most probable character. ASCII code is then acquired from the resulting character.

The aim of the thesis was also developing a programme for a presentation of the library functionality. In spite of the fact that the thesis assignment included only the MS Windows environment, I decided to create a demonstrative programme in the Gtk# interface on the Mono platform. This platform allows a compatibility with the system GNU/Linux. Unfortunately, as it turned out later, it was not a good choice and the development of the programme became more complicated.

As for the evaluation of the Mono platform functionality, I dare to say that in spite of all pitfalls that I went through with the Mono interface and its Gtk# library they may become a base for many commercial projects. I also find the IDE environment MonoDevelop well-handled. I still cannot imagine this environment in the commercial sector but, as a base for a smaller amateur project, it seems to be a successful tool in the ratio price/performance.

Thus the result of my work is a ***.dll*** library whose aim is to discern a symbol from a bitmap template. Those templates are evaluated by algorithms based on the principle of a number of intersections of lines of different lengths, positions and directions. During my work I learned about technical neuronal nets. Their task was recognition of a symbol on the base of an evaluated template.

The advantages of my library include the simplicity of capsulization into a different programme project. Via the change of a file with a neuronal net (*digitizér.net*), it is very well possible to change the character of the results without interference into the library code.

One of the disadvantages is the weak recognition ability. It is caused mainly by the evaluating ability of the inner algorithms where the same symbol can be evaluated differently. Those algorithms can be improved though, especially by choosing the right number, position and direction of the intersecting lines or by straightening a pitched

symbol into the vertical position. The neuronal net shares a great deal of the result. The important thing is finding the most convenient configuration. Those amendments and improvements can be examined into the future.

SEZNAM POUŽITÉ LITERATURY

[0] Vývoj písma [online] [cit 15.4.2010] <http://www.phil.muni.cz/~dchytkova/>

[1] Technet [online] 24.11.2010 [cit 15.4.2010] Jak se naučil počítač číst milion knížek ročně <http://technet.idnes.cz/jak-se-pocitac-naucil-cist-milion-knizek-rocne-fo8-/tec_technika.asp?c=A071123_182221_tec_technika_pka>

[2] Wikipedie otevřená encyklopedie dostupná [online] 20.4.2010 [cit 29.4.2010] OCR <http://en.wikipedia.org/wiki/Optical_character_recognition>

[3] Wikipedie otevřená encyklopedie dostupná [online] 29.3.2010 [cit 29.4.2010] ReCaptcha <http://en.wikipedia.org/wiki/ReCAPTCHA>

[4] Číslicová filtrace, analýza a restaurace signálů (Jiří Jan)

[5] Wikipedie otevřená encyklopedie dostupná [online] 30.3.2010 [cit 3.5.2010] Neuronová síť <http://cs.wikipedia.org/wiki/Neuronov%C3%A1_s%C3%AD%C5%A5>

[6] Mistrovství v C++ (Stephan Prata)

[7] Wikipedie otevřená encyklopedie dostupná [online] 27.4.2010 [cit 4.5.2010] C programovaci jazyk z WWW: <http://cs.wikipedia.org/wiki/C_%28programovac%C3%AD_jazyk%29>

[8] Programujeme v C# profesionálně (*Simon Robinson, K. Scott Allen,…)*

[9] Wikipedie otevřená encyklopedie dostupná [online] 3.5.2010 [cit 4.5.2010] C Sharp programing language <http://cs.wikipedia.org/wiki/C_Sharp>

[10] Microsoft .NET a linux dostupné z www: <http://www.linuxexpres.cz/software/microsoft-net-a-linux?highlightWords=linux+pro+nevidom%C3%A9>

[11] Kihovna FANN dostupná z www: <http://leenissen.dk/fann/>

[12] Wikipedie otevřená encyklopedie dostupná [online] 28.3.2010 [cit 6.5.2010]. OpenMP. dostupné z WWW: <http://cs.wikipedia.org/wiki/OpenMP>

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

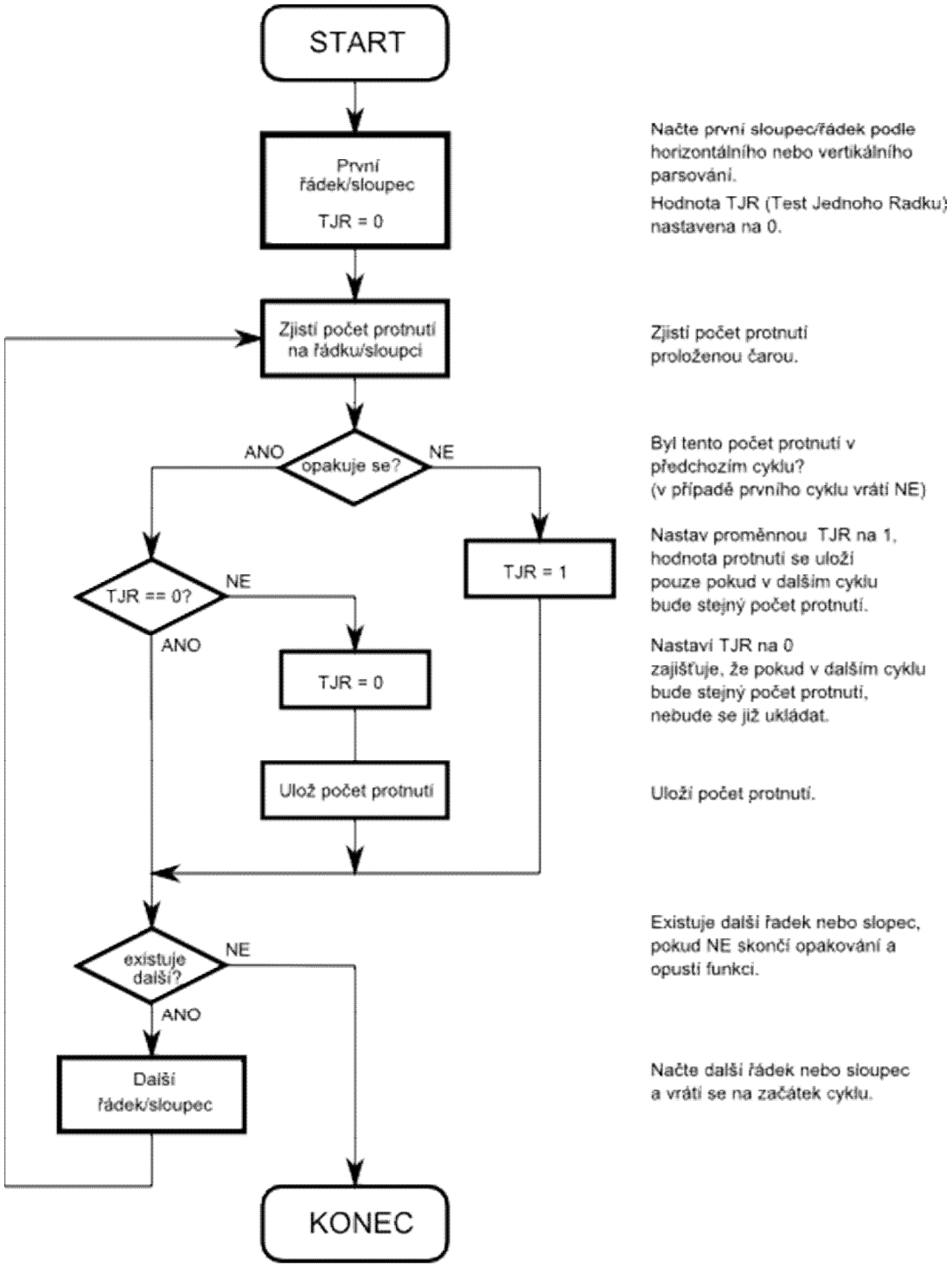
OCR	Optical charakter recongnition (optické rozpoznávání znaků)
NS	Neuronová síť
GUI	Graphical user interface (Grafické uživatelské prostředí)
IDE	Integrated develpment interface (Integrované vývojové prostředí)
FANN	Fast artificial neural network (název knihovny pro práci s neuronovou sítí)
MSVC	MS Visual C++ Studio
OpenCL	Open computing language (standart pro paralelní programování)
ASCII	American National Standards Institute
CPU	Central Processing Unit

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: SCHÉMA PARSOVÁNÍ VE SPECIELIZÉRU



PŘÍLOHA P2: PŘÍKLAD VYUŽITÍ DIGITIZERU

```
#include <stdio.h>

#include "../digitizer/uDigitizer.h"

using namespace digitizer;

int main(int argc, char * argv[])
{
	int iResultASCII = 0;

	// bitmapa se symbolem "+"
	unsigned char pcData[] = {0, 0, 1, 0, 0,
	                          0, 0, 1, 0, 0,
	                          1, 1, 1, 1, 1,
	                          0, 0, 1, 0, 0,
	                          0, 0, 1, 0, 0};
	// vstupní bitmapa
	digitizer_data_t oBitmap;
	// nastaveni parametrů
	oBitmap.piWidth = 5;
	oBitmap.piHeight = 5;
	oBitmap.iBpp = 8;
	oBitmap.ppcData = pcData;

	// inicializace knihovny
	gInit();

	// otestovaní bitmapy na celé množině
	iResultASCII = giDigitize(&oBitmap, NULL, 0);
	printf("Result char (ASCII) is: %c (%i)\nResult precize: %f%%",
	                        (char)iResultASCII, iResultASCII,
	                        oBitmap.fGraphResult);

	// otestovaní bitmapy na omezené množině symbolů „a“,“b“ nebo “c“
	iResultASCII = giDigitize(&oBitmap, "abc", 3);
	printf("Result char (ASCII) is: %c (%i)\nResult precize: %f%%",
	                        (char)iResultASCII, iResultASCII,
	                        oBitmap.fGraphResult);

	// ukončení knihovny
	gDestroy();
}
```

Příklad představuje dvojí otestování vstupní bitmapy. Jednou s neomezenou množinou výstupů a podruhé je omezen výstup na množinu znaků „a“, „b“ nebo „c“. Výsledek tedy v tomto případě bude jeden z těchto tří nebo nula (nerozlišeno).