

Rozšíření systému rozpoznávání ručně psaného písma

Handwriting recognition system extension

Bc. Michal Hájek



ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Michal HÁJEK**
Osobní číslo: **A09470**
Studijní program: **N 3902 Inženýrská informatika**
Studijní obor: **Informační technologie**

Téma práce: **Rozšíření systému rozpoznávání ručně psaného písma**

Zásady pro vypracování:

1. Zpracujte literární rešerši na dané téma.
2. Rozšiřte stávající systém rozpoznávání ručně psaného písma o inteligentní rozpoznávání slov oproti existujícímu slovníku.
3. Převeďte zdrojové kódy vytvořené v prostředí Matlab do programovacího jazyka C#.
4. Navrhněte nový algoritmus učení, ve kterém bude využito více jader procesoru pro výpočet (multithreading).
5. Dílčím cílem práce je také rozšířit stávající databázi vzorů potřebných pro naučení neuronové sítě a navrhnout nový systém sběru vzorů a jejich zpracování.
6. Poslední částí bude vytvoření nového GUI aplikace.

Rozsah diplomové práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. ZELINKA, Ivan. Umělá inteligence I. Volume 1. Zlín : Vutium, Brno, 1998. 126 s. ISBN 80-214-1163-5
2. GONZALES, Rafael C, WOODS, Richard E, EDDINS, Steven L. Digital image processing using MATLAB. Upper Saddle River : Pearson/Prentice Hall, 2004. 609 s. ISBN 0-13-008519-7
3. NAGEL, Christian, et al. C Sharp 2008 Programujeme profesionálně. Brno: Computer Press, 2009. 1904s. ISBN 978-80-251-2401-7
4. MAREŠ, Amadeo. 1001 tipů a triků pro C Sharp. Brno: Computer Press, 2008. 360s. ISBN 978-80-251-2125-2
5. LECUN, Yann, et al. Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE. 1998, Volume: 86 Issue:11 , s. 2278-2324

Vedoucí diplomové práce:

Ing. Jiří Pálka

Ústav elektroniky a měření

Datum zadání diplomové práce:

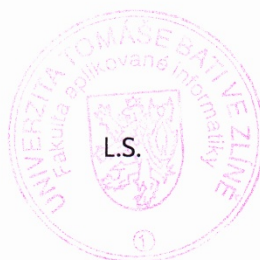
24. února 2011

Termín odevzdání diplomové práce:

18. května 2011

Ve Zlíně dne 24. února 2011


prof. Ing. Vladimír Vašek, CSc.
děkan




doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

ABSTRAKT

Práce se zabývá rozpoznáváním ručně psaného tiskacího písma. Hlavní náplní práce sběr informací týkající se tohoto oboru, převážně o využití neuronových sítí. Zabývá se problémy zpracování dokumentů, rozpoznání jednotlivých znaků a následné vyhledání celých slov oproti slovníku. V praktické části byla vytvořena aplikace pro detekci znaků, napojení s neuronovou sítí v prostředí MATLAB a vyhledání proti existujícímu slovníku.

Klíčová slova: neuronové sítě, neocognitron, backpropagation, zpracování obrazu, detekce hran, rozpoznání písma

ABSTRACT

The work deals with the recognition of handwritten printed writing. The main responsibility of gathering information regarding this field, mainly on the use of neural network. It deals with the problems of document processing, recognition of individual characters and subsequent search for whole words against the dictionary. In the practical application was created for the detection of characters connected with neural network in MATLAB and search against the existing dictionary.

Keywords: neural network, Neocognitron, Backpropagation, image processing, edge detection, writing recognition

PODĚKOVÁNÍ

Chtěl bych poděkovat mojí rodině, za to, že mi umožnila studovat a rozvíjet moje znalosti, přátelům, jež mi pomáhali udržet chladnou hlavu a dobrou náladu a v neposlední řadě bych chtěl poděkovat Ing. Jiřímu Pálkovi za odborné vedení práce a podání pomocné ruky v nejtěžších chvílích.

„Nikdy jsem nedopustil, aby škola stála v cestě mému vzdělání.“

Mark Twain

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

PODĚKOVÁNÍ.....	5
ÚVOD.....	10
I TEORETICKÁ ČÁST.....	11
1 ROZPOZNÁNÍ TEXTU	12
1.1 HISTORIE	12
1.2 STRUKTURA	13
1.2.1 Vstupní data	13
1.2.2 Předzpracování dat	13
1.2.3 Rozpoznávací algoritmy.....	13
1.2.4 Poprocesní zpracování	14
1.3 VYUŽITÍ	15
2 DETEKCE HRAN V OBRAZE A JEJICH REPREZENTACE.....	16
2.1 DIGITÁLNÍ OBRAZ.....	16
2.1.1 Vektorový obraz.....	16
2.1.2 Rastrový obraz	16
2.2 DETEKCE HRAN	16
2.2.1 Cannyho hranový detektor	18
2.2.2 Houghova transformace	19
2.2.3 Reprezentace hran	20
2.2.3.1 Řetězcové kódy.....	21
3 NEURONOVÉ SÍTĚ	22
3.1 DĚLENÍ SÍTÍ, ZÁKLADNÍ POJMY	23
3.1.1 Podle počtu vrstev	23
3.1.2 Podle algoritmu učení	23
3.1.3 Podle stylu učení	24
3.2 PŘENOSOVÉ FUNKCE	24
3.2.1 Funkce perceptron.....	24
3.2.2 Binární funkce	25
3.2.3 Funkce logistická (sigmoida)	25
3.2.4 Funkce hyperbolický tangens.....	26
3.3 VÍCEVRSTVÁ SÍŤ S ALGORITMEM BACKPROPAGATION	26
3.3.1 Aktivační fáze	27
3.3.2 Adaptační fáze.....	28
3.3.3 Chybová funkce a globální minimum	28
3.4 NEOCOGNITRON	28
3.4.1 Struktura sítě	29
3.4.2 Propojení sítě.....	30
3.4.3 Buňky	32

3.4.3.1	S-buňky	32
3.4.3.2	V-buňky	33
3.4.3.3	C-buňky	33
3.4.4	Učení Neocognitronu	34
3.4.5	Vybavování Neocognitronu	34
4	C# 2008	36
4.1	PŘEDDEFINOVANÉ DATOVÉ TYPY	37
4.1.1	Referenční typy	38
4.1.2	Hodnotové typy	38
4.2	ŘÍZENÍ BĚHU PROGRAMU	40
4.2.1	Příkaz if	40
4.2.2	Příkaz switch	41
4.2.3	Cyklus for	42
4.2.4	Cyklus while	42
4.2.5	Cyklus do...while	43
4.2.6	Cyklus foreach	43
5	LINQ	45
5.1	CO JE LINQ?	45
5.2	IMPLEMENTACE LINQ	46
5.2.1	LINQ pro objekty	46
5.2.2	LINQ pro ADO.NET	47
5.2.3	LINQ pro XML	48
5.3	SYNTAXE LINQ	48
5.3.1	Klauzule from	50
5.3.2	Klauzule where	51
5.3.3	Klauzule select	51
5.3.4	Klauzule group a into	52
5.3.5	Klauzule orderby	52
5.3.6	Klauzule join	53
5.4	DEGENEROVANÉ DOTAZOVACÍ VÝRAZY	53
5.5	ZPRACOVÁNÍ VÝJIMEK	53
6	VYHLEDÁVACÍ ALGORITMY	55
6.1	NAIVNÍ ALGORITMUS	55
6.2	KNUTH-MORRIS-PRATTŮV ALGORITMUS (KMP)	56
6.3	BOYER-MOOREŮV ALGORITMUS (BM)	58
6.4	QUICKSEARCH	59
II	PRAKTICKÁ ČÁST	61
7	NÁVRH ŘEŠENÍ	62
7.1	PŘEVOD ZDROJOVÝCH KÓDŮ	64
7.2	PARALELIZACE ALGORITMU NEURONOVÉ SÍTĚ	65
8	REALIZACE APLIKACE	66
8.1	KNIHOVNA AForge	66
8.2	ZPRACOVÁNÍ OBRAZU	66
8.2.1	Zjištěné vady algoritmu	69

8.3	NEURONOVÁ SÍŤ.....	70
8.3.1	Vícevrstvá neuronová síť s algoritmem Backpropagation.....	70
8.3.2	Neocognitron.....	71
8.4	VYHLEDÁVACÍ ALGORITMUS PRO VYHLEDÁNÍ SLOV VE SLOVNÍKU	73
8.4.1	Požadavky	73
8.4.2	Realizace	74
8.5	POPIS GUI APLIKACE A OVLÁDÁNÍ.....	79
9	NÁVRH FORMULÁŘE	81
9.1	DATABÁZE MNIST	81
9.2	NÁVRH A VYTVOŘENÍ VLASTNÍ DATABÁZE.....	81
9.3	SBĚR DAT	83
10	BUDOUCNOST PROJEKTU	84
	ZÁVĚR	86
	ZÁVĚR V ANGLIČTINĚ.....	88
	SEZNAM POUŽITÉ LITERATURY.....	90
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	91
	SEZNAM OBRÁZKŮ	92
	SEZNAM TABULEK.....	93

ÚVOD

Znaky a písmo existuje již od pradávna. Vzniklo při potřebě zaznamenávat informace trvalým a neměnným způsobem. Písmo nám umožňuje nahradit informaci obrazovým znakem. Tento znak může reprezentovat písmeno, slovo větu či celý příběh. Písmo se napříč celou historií vyvíjelo a usadilo se až do dnešní podoby. Dnes se reprezentuje informace jednoho znaku grafickým symbolem. Jenže samotné zaznamenávání by bylo k ničemu, pokud by jsme nebyli schopni informaci opět získat.

Člověk se naučí číst již v útlém věku. Jde o osvojení schopnosti, kdy přiřadí grafickému symbolu příslušný znak. V dnešní době je ale potřeba zpracovávat velké množství dat. S vyplňováním formulářů se člověk setkává všude, od posílání poštovních zásilek, až po komunikaci s úřady. Tyto formuláře je potřeba zpracovávat elektronicky.

S rozvojem techniky a jejího výkonu se začaly objevovat aplikace, které byly schopny přiřadit obrazů význam. První algoritmy dokázaly rozpoznávat strojově psané znaky. Vznikly systémy označované jako OCR (optické rozpoznání znaků). Tyto systémy se zdokonalovaly, řešily se velikosti znaků, jejich částečná rotace a další problémy. Jenže tyto algoritmy nedokázaly účinně rozpoznávat deformované či ručně psané znaky.

Tomuto odvětví pomohl rozvoj neuronových sítí. Tyto sítě se snaží technickou cestou napodobit fungování lidského mozku. S dostupným výpočetním výkonem také roste možnost využití složitějších systémů sítí. Díky těmto vlivům začaly vznikat ICR systémy (inteligentní rozpoznávání písma). Využívající modely neuronových sítí a s větší či menší úspěšností se začaly učit rozpoznávat jednotlivé znaky jako lidský mozek.

I. TEORETICKÁ ČÁST

1 ROZPOZNÁNÍ TEXTU

Rozpoznání ručně psaného textu je specifickým oborem vědní disciplíny optického rozpoznávání znaků (Optical Character Recognition). OCR je v podstatě technologie, která slouží k zpracování a převedení textu z bitmapového formátu do digitální formy. Jedná se tedy o digitalizaci dat.

1.1 Historie

Technologie rozpoznání znaků je poměrně mladým oborem, avšak určité náznaky můžeme pozorovat již na počátku 20. století. Mezi první zmínky patří rok 1914, kdy byl v USA vydán patent na mechanický přístroj sloužící mimo jiné i k rozpoznávání rukopisů. Dále byly ve 30. letech zaznamenány patenty na mechanické přístroje, které pomocí šablony a fotodetektoru rozpoznávaly určité znaky. Tyto přístroje však mají s dnešním oborem OCR jen pramálo společného.

Skutečné počátky bychom měli datovat do 60. let 20. století, které souvisí s rozvojem výpočetní techniky. Je zde snaha o rychlé a efektivní zpracování bankovních dokumentů jako jsou šeky, cenné papíry, kreditní karty, dále zpracování poštovních zásilek atd. Úplně první systém vytvořila společnost Reader's Digest již v roce 1955. Ten sloužil pro převod ručně psaného textu na děrové štítky, pro další zpracování počítačem. V roce 1965 vzniká v USA standart písma OCR-A, což je v podstatě první zjednodušené písmo, umožňující strojové čtení. V témže roce začala využívat technologii OCR americká poštovní společnost United States Postal Service na třídění zásilek. Vznikají také první slavné systémy jako IMB 1287, který byl představen na světové výstavě v New Yorku 1965.

V 70. letech se objevují první komerční systémy a dochází k postupnému rozvoji OCR. Avšak až do 90. let je díky vysoké pořizovací ceně jejich využití omezeno pouze na několik málo velikých společností. V posledních 20 letech došlo v důsledku prudkého rozvoje výpočetní techniky jednak ke zvýšení efektivity systémů, a také především ke snížení jejich ceny, což umožnilo masové rozšíření OCR systémů mezi běžné uživatele. Díky tomu dnes existuje celá škála oborů, kde je využíváno těchto systémů[7].

1.2 Struktura

1.2.1 Vstupní data

Vstupem celého rozpoznávacího systému jsou různé druhy vstupních dat, které má systém zpracovávat. U rozpoznání ručně psaného textu se vstupní data dělí do dvou základních skupin.

On-line data – text je zapisován přímo v digitální formě pomocí speciálních zařízení, jako jsou digitalizační tablety, dotykové displeje. Rozpoznání on-line dat zaznamenává v posledních letech raketový rozvoj, díky rozmachu zařízení jako jsou kapesní počítače, dotykové mobilní telefony a další přístroje s dotykovým displejem.

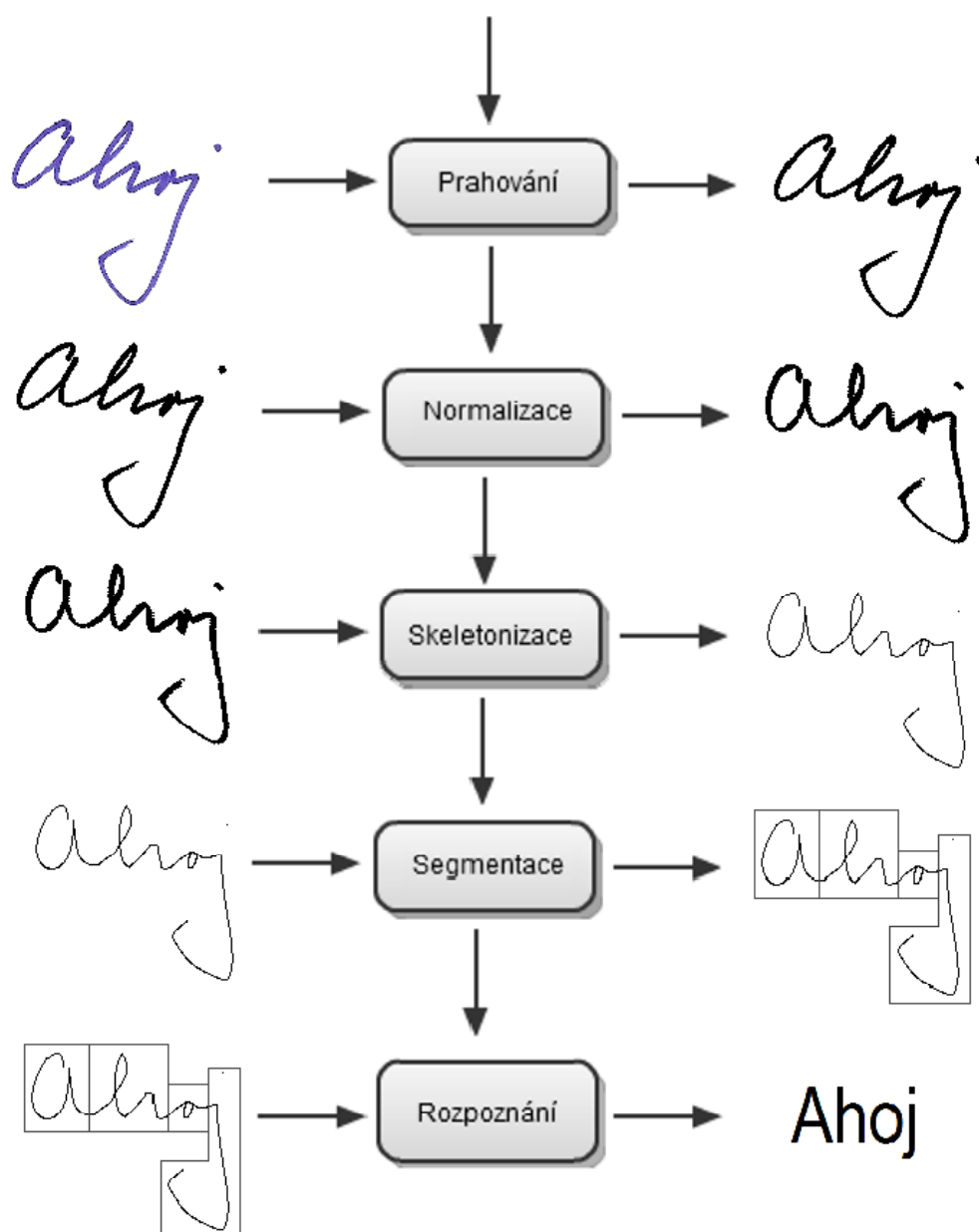
Off-line data – jsou klasická data napsaná na papíře. Pro jejich zpracování je tedy nejprve nutné tyto data převést do digitální formy pomocí skeneru.

1.2.2 Předzpracování dat

Přímá vstupní data však mají stále množství nedostatků, které stěžují jejich rozpoznání příslušnými algoritmy. Na řadu tedy musí přijít tzv. *preprocesní zpracování*, které vstupní data upraví do takové podoby, která je příhodná pro rozpoznávací algoritmy. Preprocesní zpracování má větší váhu u dat získaných off-line, avšak některé jeho části najdou uplatnění i u zpracovávání on-line dat. Mezi techniky preprocesního zpracování patří jednoduché techniky *prahování* sloužící k oddělení znaků od pozadí. Techniky *segmentace* pro izolaci jednotlivých znaků. Dále *skeletonizace* pro vypreparování kostry daných znaků a v neposlední řadě *normalizační* techniky pro výslednou úpravu znaků do požadované podoby. Tímto předzpracováním vstupních dat značně ulehčíme práci samotných algoritmů a dosáhneme vyšší účinnosti rozpoznání[2].

1.2.3 Rozpoznávací algoritmy

Jádrem OCR systémů jsou již samotné metody pro rozpoznání. Dnes se nejběžněji používají algoritmy založené na neuronových sítích, skrytých Markových modelech, klasifikátorech minimální vzdálenosti apod. Tyto metody mají obvykle velikou úspěšnost rozpoznání u dobře napsaných dat. U ručně psaných dat je rozpoznání komplikovanější díky značné rozmanitosti jednotlivých rukopisů. Při zpracování tedy může dojít u různých systémů k chybné klasifikaci znaku nebo k neschopnosti znak rozpoznat, proto je zde potřeba určité následné úpravy získaných dat.



Obr. č. 1: Schéma ideálního zpracování dat

1.2.4 Poprocesní zpracování

Tímto se dostáváme k další části systémů, kterou je poprocesní zpracování dat. Zde pracujeme se znalostí kontextu výsledného textu. Mezi nejběžnější nástroje patří kontrola pravopisu (spell checking). K chybné kvalifikaci může dojít jednak díky zhoršení kvality dat. Některé znaky jsou si však velmi podobné, proto je může algoritmus rozpoznání snadno zaměnit. Typickým příkladem jsou např. „5“ a „S“ nebo „rn“ a „m“ aj. Tato snadno vzniknutelná záměna je díky znalosti kontextu také snadno identifikovatelná a opravitelná. Některé znaky je bez znalosti kontextu nemožné efektivně rozpoznat např. „0“ a „O“. Dnešní robustní systémy se bez poprocesního zpracování neobejdou[2].

1.3 Využití

Možností využití systémů pro rozpoznání znaků je dnes nepřehledné množství. Klasické rozpoznání znaků OCR nachází značné využití při digitalizaci textů, čehož se hojně využívá například v knihovnách apod. Specifičtější rozpoznání ručně psaného textu můžeme také využít k digitalizaci textu, jako jsou rukopisy, což je zde však spíše okrajová záležitost. Hlavní pole působnosti těchto systémů je především tam, kde se zpracovává velké množství ručně psaných dat, což znamená např. na poštách pro identifikaci PSČ, dále v bankovníctví pro identifikaci šeků, v pojišťovnictví, pro zpracování různých dotazníků apod.

U rozpoznání on-line dat zaznamenávají tyto systémy v posledních letech masivní rozšíření a lze předpokládat značný rozvoj i v budoucnosti. Systémy pro rozpoznání ručně psaného textu se stávají dnes již běžným standardem u tabletových PC, kapesních počítačů, mobilních telefonů a v podstatě u většiny zařízení vybavených dotykovým displejem. Tyto systémy jsou dnes běžnou součástí balíčků operačních systémů, což je masově rozšiřuje ke koncovým uživatelům[7].

2 DETEKCE HRAN V OBRAZE A JEJICH REPREZENTACE

Tato kapitola se zabývá detekcí a reprezentací hran v obraze. Na samotné rozpoznávání znaky není použita, ale je potřebná pro zpracování formulářů, získávání oblastí, kde jsou vyplněny znaky či srovnání samotného zpracovávaného dokumentu.

2.1 Digitální obraz

Pro zpracování obrazu digitální cestou je potřeba formulář digitalizovat. Jedná se o převod funkce obrazu do jeho diskrétní podoby. Tím se ale ztrácí část informace. Množství ztracené informace lze ovlivnit volbou rozlišení a hustotou rozlišení označovanou jako DPI (Dots Per Inch). Tato kombinace nám udává velikost obrazu a jeho jemnost.

Samotný obraz může být digitálně zpracován dvěma základními způsoby. A to jako vektorový obraz nebo jako rastrový obraz.

2.1.1 Vektorový obraz

Vektorové obrazy jsou definovány soustavou rovnic popisující jednotlivé křivky obrazových informací. Tato reprezentace obrazu je výhodná pouze u obrazů s malým počtem objektů. Při popsání běžné fotografie vektorovým obrazem by šlo o neuvěřitelně velké množství rovnic a jejich zpracování by bylo velmi náročné. Vektorové obrazy mají výhodu v minimální ztrátě informace. Lze je bez ztráty informace zmenšovat a zvětšovat.

2.1.2 Rastrový obraz

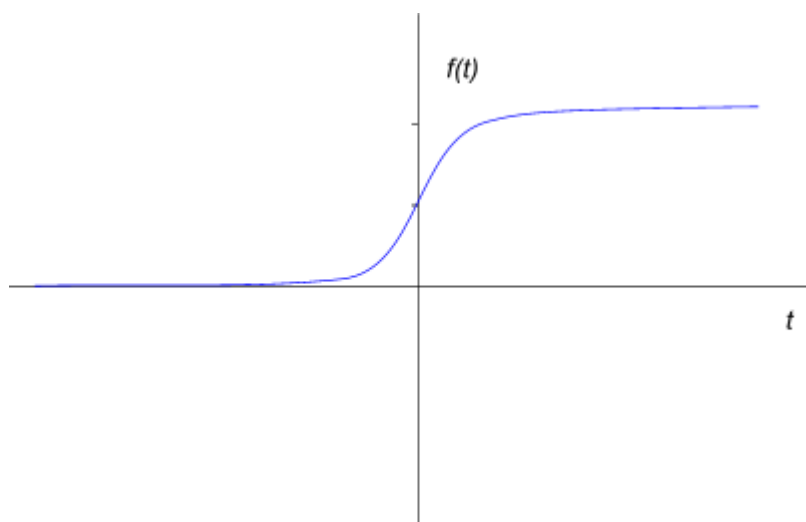
Rastrový obraz je reprezentován mřížkou položenou na skutečný obraz a vypočítáním hodnoty jednotlivých pixelů. Podle hustoty mřížky (DPI) se určuje také ztráta informace. Každý bod je reprezentován v mřížce jako hodnota barvy počtená jako průměr malé oblasti. V rastrovém formátu lze uchovávat poměrně velké množství obrazových informací. Nevýhodou tohoto formátu je ztráta informací, která vzniká při zmenšení obrazu a nemožnost obnovení informací při jeho zvětšení.

2.2 Detekce hran

Detekce hran v obraze je disciplína, při které se v rastrovém obraze hledá skupina pixelů, kde se značným způsobem mění jas obrazu. Takovou skupinu pixelů lidské oko vnímá jako hranu objektu. Typickým příkladem takové hrany je tabulka formuláře. Zde je přechod

nejviditelnější, protože se ve většině případů jedná o přechod mezi bílou barvou a černou barvou.

Samotný přechod v obraze lze zobrazit přechodovou funkcí, kterou získáme pohybem v obraze. Takové přechody vyhledávají algoritmy zvané jako hranové detektory (edge detector)[8].



Obr. č. 2: Přechodová jasová funkce

Hranových detektorů existuje velké množství. Tyto detektory se liší ve spoustě parametrů a možností nastavení. Mezi základní parametry se řadí způsob nalezení hrany, reprezentace hrany, citlivost rozpoznání či odolnost proti šumu a jiným vadám obrazu.

V základu se hranové detektory dělí podle rozhodování o hraně, a to na detektory využívající první derivaci a detektory využívající druhou derivaci jasové funkce. V prvním případě je vypočtený hranový gradient porovnán s prahem a tím je určeno, zda se jedná o hranu či ne. Algoritmy využívající druhé derivace jasové funkce porovnávají změnu v polaritě, a jestli je tato změna dostatečně významná.

Největší změna v jasové funkci nastává, pokud přecházíme hranu ve směru kolmém na tuto hranu. V praxi se ovšem toho nevyužívá a provádí se výpočet jen ve dvou, případně ve čtyřech směrech. Výpočet gradientu se v praxi provádí jako konvoluční filtrování obrazu. Jednotlivé hranové filtry se potom liší v jádrech filtrů. Zde jsou v maticích uváděny hodnoty, které určují, jaké body se pro výpočet použijí a také velikost váhy bodu v tomto výpočtu. Velikost matic prozradí ovlivňuje odolnost proti šumu a jiným obrazovým vadám. Některé nejznámější detektory jsou uvedeny v následujícím přehledu[8]:

	G_x	G_y
Robertsův	$\begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
Prewittové	$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
Sobelův	$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$
Robinsonův	$\begin{bmatrix} 1 & 1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ 1 & -2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
Kirshův	$\begin{bmatrix} 3 & 3 & -5 \\ 3 & 0 & -5 \\ 3 & 3 & -5 \end{bmatrix}$	$\begin{bmatrix} -5 & -5 & -5 \\ 3 & 0 & 3 \\ 3 & 3 & 2 \end{bmatrix}$
Frei-Chenův	$\begin{bmatrix} 1 & 0 & -1 \\ \sqrt{2} & 0 & -\sqrt{2} \\ 0 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & -\sqrt{2} & -1 \\ 0 & 0 & 0 \\ 1 & \sqrt{2} & 1 \end{bmatrix}$
Prewittové(5x5)	$\begin{bmatrix} 2 & 1 & 0 & -1 & -2 \\ 2 & 1 & 0 & -1 & -2 \\ 2 & 1 & 0 & -1 & -2 \\ 2 & 1 & 0 & -1 & -2 \\ 2 & 1 & 0 & -1 & -2 \end{bmatrix}$	$\begin{bmatrix} -2 & -2 & -2 & -2 & -2 \\ -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$

Metody založené na druhé derivaci jasové funkce se snaží nacházet průchody této derivace nulou. Využívají toho, že je snazší nalézt průchod nulou, nežli extrém funkce. Bohužel druhá derivace je ještě citlivější na šum, než první, proto je vhodné její výpočet kombinovat s takovým vyhlazením, které odstraní maximální množství šumu a při tom nepoškodí hrany. Tyto metody často pracují s Laplaceovým operátorem – Laplaciánem, který aproximuje druhou všesměrovou derivaci jasové funkce. K výpočtu se používá tedy pouze jedna matice. K jeho nevýhodám patří, že má dvojité odezvy na hrany odpovídající tenkým liniím v obraze.

2.2.1 Cannyho hranový detektor

Cannyho hranový detektor je znám jako optimální hranový detektor. Je navržen tak, aby splňoval tři základní požadavky na hranový detektor.

- minimální počet chyb - musí nalézt všechny hrany v obraze a odezva na místa, které nejsou hranami, musí být nulová

- přesnost - co nejpřesnější určení polohy detekované hrany
- jednoznačnost - každá hrana má být detekována pouze jednou, nesmí docházet například ke zdvojování hran

Samotný postup detekce hran se skládá z několika vlastních kroků:

1. eliminace šumu Gaussovým filtrem - na obraz je aplikován Gaussův šum, jen je jednoduše aplikovatelný filtr pomocí konvoluční masky
2. aplikace Sobelova operátoru - na obraz je aplikována konvoluční maska Sobelova detektoru, zde se nalezne směr a velikost gradientů hran
3. proces ztenčení - proces při kterém jsou detekované hrany ztenčeny a umístěny na správné místo, princip této fáze probíhá tak, že jako bod hrany je označen jen takový bod, jehož sousední body v okolí kolmém na směr gradientu (směr gradientu je znám – vrací jej Sobelův detektor) mají hodnotu gradientu nižší
4. prahování s hysterezí - je to poslední krok zpracování obrazu, jeho významem je určit a ohodnotit význam nalezených hran, hrany pocházející ze šumu mají gradient menší jako hrany pravé, Crannyho detektor používá dva prahy, překročí-li hodnota gradientu vyšší práh je hrana uznána, nepřekročí-li hodnota gradientu nižší práh, hrana uznána není, je-li hodnota gradientu v rozmezí mezi prahy je hrana uznána jen, pokud sousedí s jinou hranou

Crannyho hranový detektor je v současné době považován za vrchol hledání "nejlepšího" hranového detektoru[2].

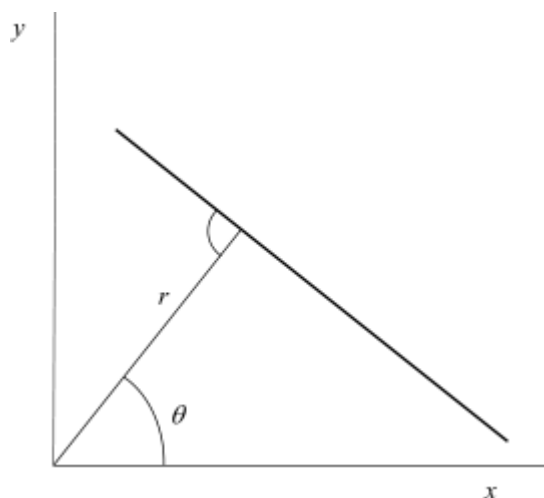
2.2.2 Houghova transformace

Houghova transformace je metoda, která nám pomáhá naleznou parametrický popis objektů v obraze. Nevýhodou je musíme dopředu znát analytický popis tvaru objektu, který hledáme. Proto se tato metoda používá pro nalezení známých objektů, jako jsou přímky, kružnice, elipsy a z nich složené obrazce jako čtverce, trojúhelníky a podobně. Největší výhodou této metody je její robustnost a odolnost proti šumu či porušeným datům. Metoda používá převod z kartézského souřadnicového systému do polárního. Detekce přímek dokumentu je využita například při detekci natočení dokumentu a pomocí ní lze dokument správně otočit a zpracovat.

Pro ukázkou je uveden parametrický popis přímky:

$$r = x \cdot \cos\theta + y \cdot \sin\theta$$

kde r je délka normály od přímky k počátku souřadnic a θ je úhel mezi normálou a osou x



Obr. č. 3: Parametrický popis přímky

Samotná transformace probíhá tak, že parametry x a y každého vstupního bodu obrazu, který představuje bod úsečky nebo přímky, jsou dosazeny do rovnice (1.1). Za hodnotu θ je postupně dosazována každá možná hodnota – interval $\langle 0, 360 \rangle$ (matematicky se jedná o nekonečnou množinu hodnot reálných čísel představujících úhel θ , ale v případě implementace jsou dosazovány hodnoty dle nastavení citlivosti Houghovy transformace) a hodnota r se dopočítává. Tak v parametrickém prostoru vznikají křivky, pro každý vstupní bod jedna. Pokud jsou vstupní body kolineární, křivky parametrického prostoru se protínají nejčastěji v bodě představujícím parametry hledané přímky.

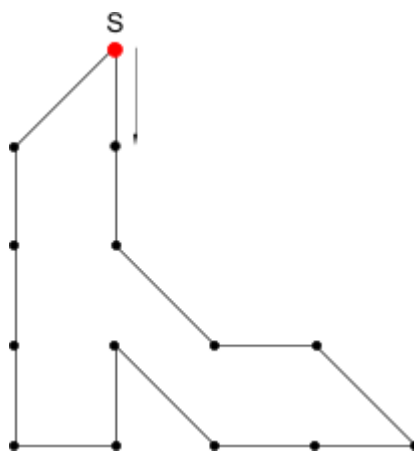
2.2.3 Reprezentace hran

Reprezentace hran nám poskytuje aparát, pomocí něhož lze ukládat a prezentovat data, poskytovaná hranovými detektory. Zobrazení těchto hran je ve většině případů totiž nedostačující a je potřeba tyto hrany nadále zpracovávat a ukládat.

Pokud se nám podaří získat kontury z obrazu, vyvstává problém, jak tyto kontury budeme nadále zpracovávat. Jedním z největších kritérií zpracování těchto kontur je jejich povaha. Pokud potřebujeme ukládat data, kde je předpoklad dlouhých přímek či malý počet ostrých ohybů je výhodné využít ukládání pomocí matematických funkcí. Pokud je potřeba zpracovávat rastrové data, kde se dá očekávat velké množství ostrých změn směru, zatížení šumem a vad obrazů či velmi nepravidelné tvary je výhodnější užívat řetězcových kódů[8].

2.2.3.1 Řetězcové kódy

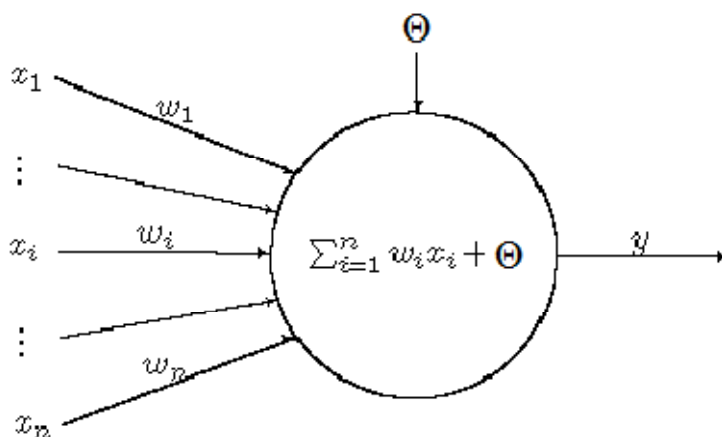
Princip řetězcových kódů spočívá v tom, že cestu z jednoho bodu do sousedního bodu popisované kontury lze označit – například číslicí. Pokud uvažujeme, že se pohybujeme na úrovni jednotlivých pixelů, musíme podle zvolené provázanosti takto označit 4 resp. 8 směrů. Freemanův řetězcový kód pracuje s 8mi směry a s jejich označením číslicemi od 0 do 7. Tyto směry si můžeme představit jako možnosti tahu s dámou na šachovnici. Potom řetězcový kód kontury s počátkem v bodě S bude vypadat takto {4,4,3,2,3,6,6,7,4,6,0,0,0,1}.



Obr. č. 4: Kontura řetězcového kódu

3 NEURONOVÉ SÍTĚ

Historie vzniku prvních umělých neuronových sítí se řadí do první poloviny 20. století. Ve 40. letech pak W. S. McCulloch a W. Pitts sestavili neuron, který se prakticky ve stejné podobě používá dodnes. První síť Perceptron ovšem nebyla schopna řešit lineárně neseparabilní úlohy, což způsobilo útlum zájmu. Poté byly objeveny sítě s šířením chyb, které jsou schopny řešit i problémy lineárně neseparabilní a význam neuronových sítí tak roste dodnes.



Obr. č. 5: Model umělého neuronu

V současné době lze využít neuronové sítě ve velkém počtu odvětví, jako např.:

- identifikaci radarových či sonarových signálů
- predikce chování
- optimalizace
- filtrace
- klasifikace

a v mnoha dalších.

Mezi největší výhody neuronových sítí oproti klasickému analytickému programování je odolnost proti odchylkám. V případě klasického programování musí být vstupy v přesném tvaru či z definovaného rozsahu. Nelze tedy efektivně vyhodnocovat například tvary, které se mohou lišit pouze v drobnostech, ale natolik velkých, že se dostanou mimo rozsah. Neuronové sítě oproti tomu jsou schopny vyhodnocovat i hodnoty podobné. Lze je tedy efektivně využít tam, kde je potřeba identifikovat rozdílné prvky se stejnými vlastnostmi [1]. Základní srovnání vlastností udává následující tabulka:

Tab. č. 1: Rozdíl mezi PC a NS

Neuronová síť	Počítač
Je učena nastavováním vah, prahů a struktury	Je programován instrukcemi, (if, then, go to,...)
Paměťové a výkonné prvky jsou uspořádány spolu	Proces a paměť pro něj jsou separovány
Paralelismus	Sekvenčnost
Tolerují odchylky od originálních informací	Netolerují odchylky
Samoorganizace během učení	Neměnnost programu

3.1 Dělení sítí, základní pojmy

3.1.1 Podle počtu vrstev

Dělení podle počtu vrstev znamená, že rozlišujeme, z kolika vrstev se síť skládá. Existují sítě s jednou, dvěma či více vrstvami. Sítě, jež používají jednu či dvě vrstvy, mají ve většině případů speciální učicí algoritmy a topologii. Takové sítě jsou sestavovány k přesně danému účelu. Jedná se o sítě typu Hopfieldova, Kohonenova či ART. Pro vícevrstvé sítě se používá, ve většině případů, Backpropagation algoritmus učení.

3.1.2 Podle algoritmu učení

Základní dělení podle algoritmu učení je učení s učitelem a bez učitele.

Učení s učitelem je proces, kdy se na vstup posílají vstupní data a na výstupu se kontrolují hodnoty. Tyto hodnoty jsou porovnávány s požadovaným výstupem a podle toho jsou přepočítávány váhy. Tímto způsobem se přizpůsobuje odezva neuronové sítě dle požadavků.

Učení bez učitele je proces, kdy se neuronové síti poskytne soubor vstupních dat a nejsou jí dodány požadované výsledky. Jedná se o sítě, které si výstup tvoří samy. Je pravděpodobné, že síť začne kategorizovat prvky do skupin podle jejich společných vlastností a podle jejich podobnosti. Prvky s malou odchylkou si vytvoří novou kategorii

teprve až poté, co jich bude větší množství. Pokud jich bude pouze malá část, budou přiřazeny ke kategorii, od které mají nejmenší odchylku [1].

3.1.3 Podle stylu učení

Styl učení v podstatě znamená, jak se přistupuje k zjištění vah sítě. V případě, že se jedná o zjištění výpočtem, pak mluvíme o deterministickém učení.

Jestliže jsou však váhy získávány pomocí generátoru náhodných čísel, pak mluvíme o stochastickém stylu učení. Tento způsob získání vah sítě se obvykle používá jen při startu sítě.

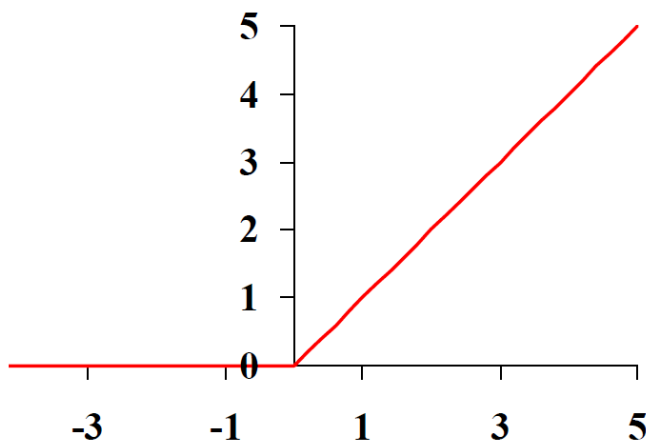
3.2 Přenosové funkce

Pro správný chod neuronu a neuronových sítí je důležité, jakou přenosovou funkci zvolíme. Přenosová funkce udává, jaká bude odezva na výstupu na vstupní podnět. Jsou různé druhy funkcí, u kterých obecně platí, že jejich hodnota má být v intervalu -1 až +1 a že mají být spojité (sigmoida, hyperbolický tangens,...), nebo s nespojitostí prvního druhu (binární funkce 0-1).

3.2.1 Funkce perceptron

Je to lineární funkce, která byla použita v Rosenblattově první neuronové síti a která díky své lineárnosti byla schopna řešit jen problémy lineárně separabilní. To v konečném důsledku vedlo k úpadku zájmu o neuronové sítě[1].

$$F(P) = x; \forall x > 0; jinak x = 0$$

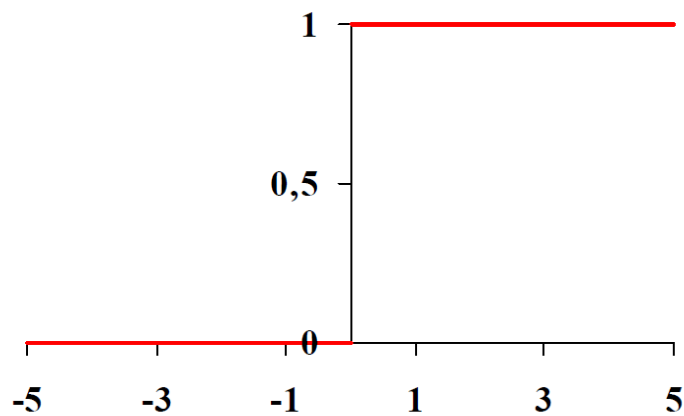


Obr. č. 6: Přenosová funkce perceptron

3.2.2 Binární funkce

Je to funkce, která může nabývat pouze dvou hodnot 0 a 1. Její využití je nejčastěji u rozhodovacích funkcí, typicky ano/ne.

$$F(B) = 1; \forall x > 0; \text{jinak } x = 0$$

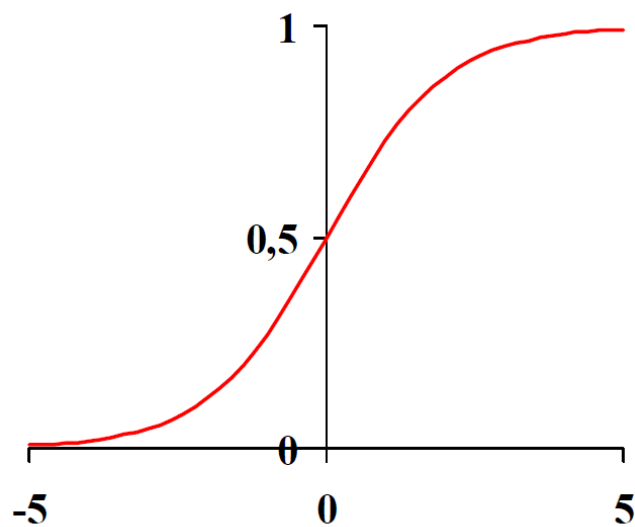


Obr. č. 7: Přenosová binární funkce

3.2.3 Funkce logistická (sigmoida)

Je to jedna z nejvíce používaných funkcí, která byla odvozena jako aproximace přenosové funkce biologického neuronu.

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

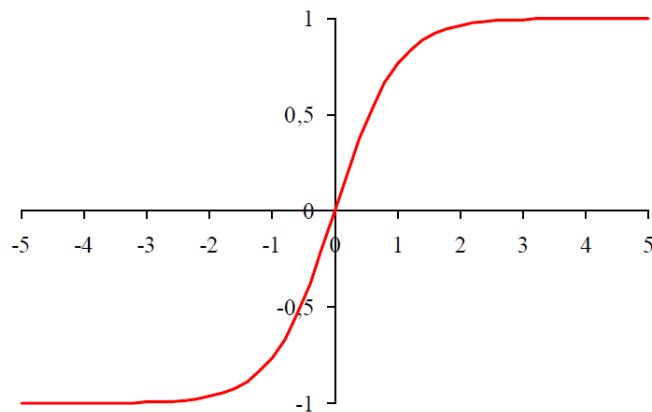


Obr. č. 8: Přenosová funkce logistická

3.2.4 Funkce hyperbolický tangens

Funkce hyperbolický tangens je obdoba logistické funkce, s tím rozdílem, že může nabývat hodnot -1 až +1, což znamená, že poskytuje mimo jiné i větší lineární úsek okolo počátku, což má také svůj význam.

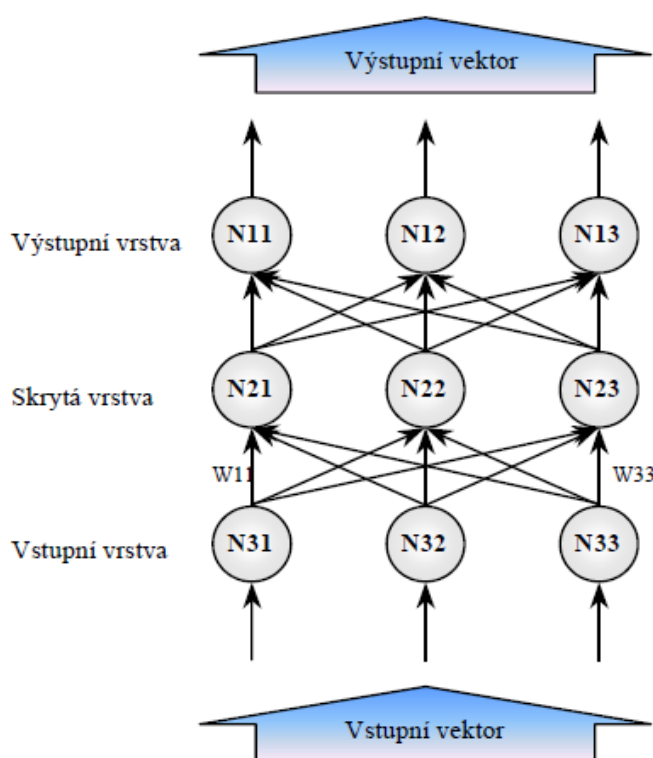
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Obr. č. 9: Přenosová funkce hyperbolický tangens

3.3 Vícevrstvá síť s algoritmem Backpropagation

Základní algoritmus, pomocí kterého se příslušná vícevrstvá neuronová síť může je Backpropagation. Někdy bývá v literatuře síť využívající tento algoritmus chybně nazývána jako síť Backpropagation.



Obr. č. 10: Síť neuronů pro algoritmus backpropagation

Backpropagation je pouze algoritmus, který byl vytvořen pro učení vícevrstvých neuronových sítí s učitelem. Tento algoritmus opravuje-nastavuje váhy jednotlivých spojů zpětným chodem tak, aby jejich velikosti byly z hlediska řešeného problému pokud možno optimální - hledá se globální minimum chybové funkce. Nastavení vah tedy probíhá v opačném směru, než jakým se šíří vstupní informace. U tohoto algoritmu opět rozeznáváme dvě fáze - aktivační a adaptační[1].

3.3.1 Aktivační fáze

Při inicializaci sítě se musí nastavit váhy na vhodnou hodnotu, což se dělá obvykle pomocí generátoru náhodných čísel. Váhy se pohybují v rozmezí 0,5 až -0,5. Kromě použití generátoru náhodných čísel existují i jiné metody pro prvopočáteční nastavení vah jako simulované žíhání a genetické algoritmy.

V této fázi se informace obsažená ve vstupním vektoru šíří z vstupu na výstup, jak ukazuje Obr. 2.5. Vstupní vektor je ve vstupní vrstvě rozvětven a každá „větev“ - spoj je ohodnocen tzv. vahou (hodnota spoje * váha). V každém neuronu, do kterého vstupují takto ohodnocené spoje, se provede jejich součet. Takto získané číslo je použito jako argument přenosové funkce, jejíž výsledná hodnota je výstupem z neuronu, a slouží jako

vstupní hodnota do dalších neuronů ve vyšší vrstvě. Pokud se jedná o vrstvu výstupní, pak výstupní hodnota z neuronů je vlastní výstupní vektor.

Aktivační fáze je používána při učení a vybavování sítě. Je to tedy aktivita, při které se vstupní informace dostane na výstup a je zmodifikována momentální množinou vah a přenosovými funkcemi ve vlastních neuronech[1].

3.3.2 Adaptační fáze

Výstupní vektor (odezva sítě na vstupní vektor) je porovnána s požadovaným originálem a rozdíl mezi oběma vektory je použit pro výpočet nových vah tak, že se nejprve opraví váhy u spojů, které vstupují do výstupní (nejvyšší) vrstvy. Pak jsou opraveny váhy u nižší vrstvy, atd... Až se dosáhne vrstvy vstupní, je tato fáze ukončena a opakuje se fáze aktivační.

Při každém porovnávání výstupní odezvy s požadovaným originálem se daný rozdíl uchová v paměťové proměnné a sumarizuje se s dalšími postupně získanými rozdíly. Takto získané číslo za celou trénovací množinu (epochu) se nazývá globální chyba. Tato globální chyba je po každé epoše kontrolována s chybou, kterou zadal uživatel a pokud je nižší, než chyba zadaná, pak je síť naučena a učení končí. Algoritmus tedy hledá globální minimum chybové funkce[1].

3.3.3 Chybová funkce a globální minimum

Vlastní globální chyba je pozice na tzv. chybové ploše, která je dána všemi vahami a jejich možnými hodnotami. Vlastní chybová plocha je mnohdy komplikovaná plocha s mnoha lokálními minimy, přičemž pozice neuronové sítě na této ploše je dána aktuálním stavem vah. Z toho je jasné, že pokud startujeme síť na nové učení s nově náhodně nastavenými vahami, jsme při každém startu pokaždé v jiném bodě chybové plochy. Problém vlastního hledání je, že obvykle nevíme, zda momentální minimum, kterého jsme dosáhli, je globální. V případě, že si nejsme jisti tím, že se jedná o lokální minimum, musíme použít některou z metod, která nám toto minimum pomůže opustit. Jednou z těchto metod je např. simulované žíhání.

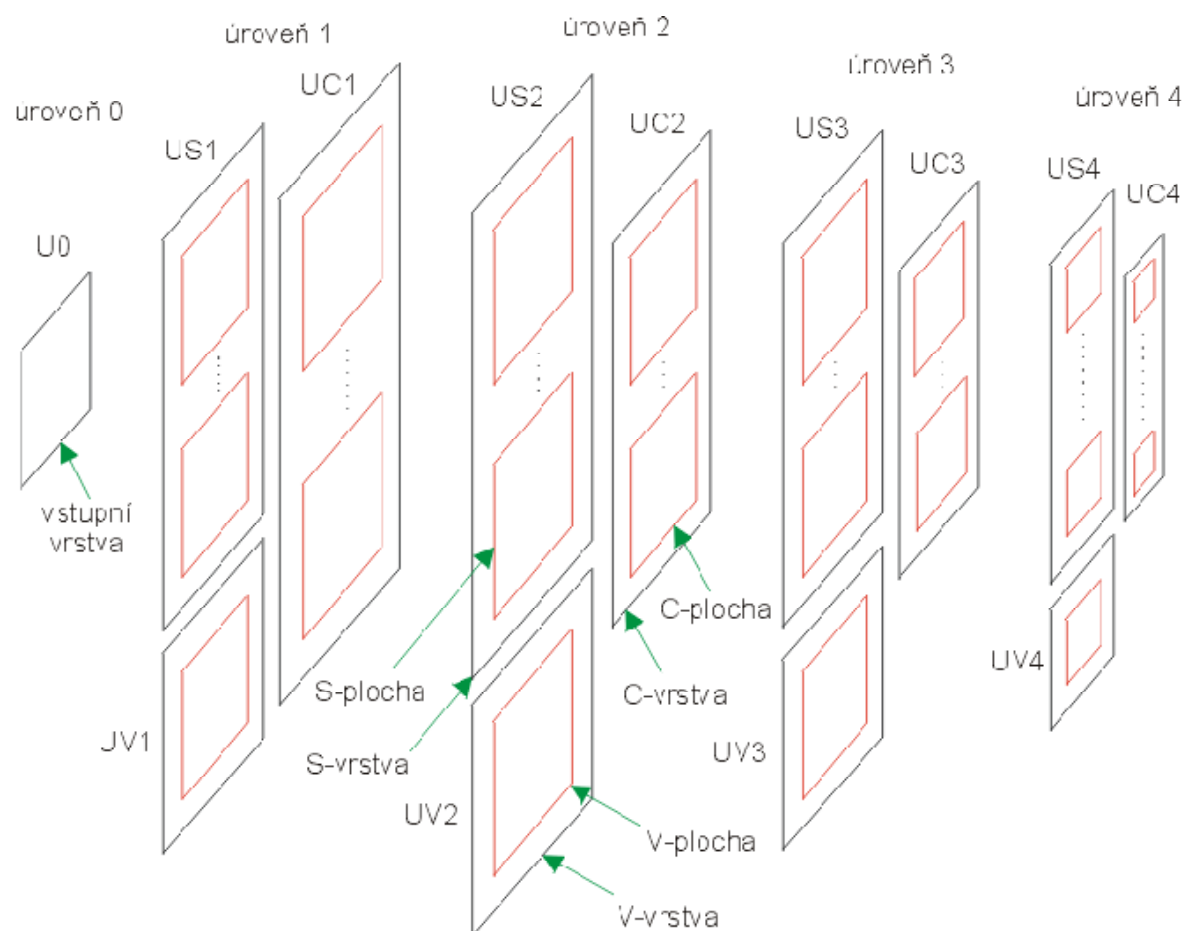
3.4 Neocognitron

Neocognitron je typ vícevrstvé hierarchické neuronové sítě, které se využívá pro rozpoznávání ručně psaných znaků. Síť navrhl japonský profesor **Kunihiko Fukushima** v

roce 1979. Během dlouhé doby od jejího uveřejnění, prošla síť mnohými modifikacemi a dnes existuje několik různých verzí. Původní verze byla bez učitele, později však byla vytvořena verze s učitelem, pro kterou je nutno vytvořit speciální soubor učících znaků.

3.4.1 Struktura sítě

Jak bylo uvedeno, jedná se o hierarchickou síť. Tato hierarchie spočívá v tom, že v nižších úrovních síť detekuje nejjednodušší příznaky. S každou další úrovní jsou tyto příznaky složitější. Jednotlivé úrovně obsahují vždy tři vrstvy: S-vrstvu, C-vrstvu, V-vrstvu. Výjimkou je pouze nultá úroveň obsahující pouze vstupní vrstvu, která ukládá vstupní informace. Jednotlivé vrstvy se skládají z určitého počtu ploch, které jsou dle příslušné vrstvy buď S, C nebo V. Jednotlivé plochy tvoří dvojrozměrné pole buněk. Síť Neocognitron obsahuje čtyři základní typy buněk: S-buňky, C-buňky, V-buňky a receptorové buňky. Základní stavební prvky, tedy jednotlivé buňky, již pracují s reálnými nezápornými hodnotami[7].



Obr. č. 11: Struktura sítě Neocognitron

Všechny plochy obsažené v jedné vrstvě mají stejně veliké pole buněk. Stejně tak jsou si rozměrově totožné V-plochy a S-plochy stejné úrovně. Příklad počtu ploch a příslušných buněk pro vrstvy jednotlivých úrovní je zobrazen v následující tabulce (Tab. 1). Tento konkrétní příklad je pro klasickou aplikaci neuronové sítě Neocognitron na rozpoznávání ručně psaných znaků, konkrétně číslic 0 až 9 a písmen velké abecedy A až Z.

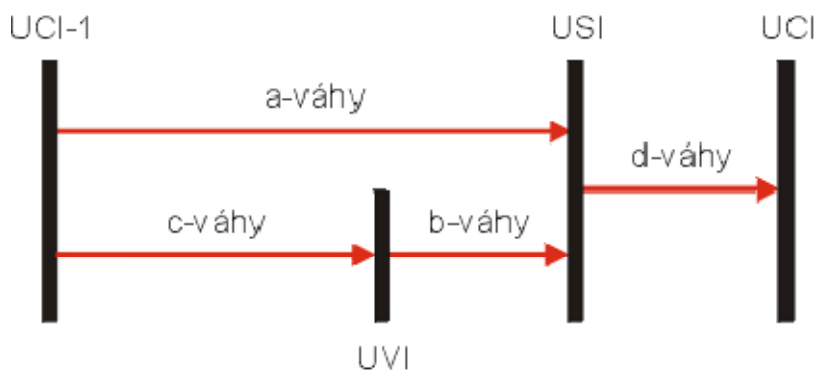
Tab. č. 2: Počet ploch a buněk sítě Neocognitron

Vrstva	Rozměr pole buněk	Počet ploch	Počet buněk
U0	19 x 19	-	361
US1	19 x 19	12	4332
UV1	19 x 19	1	361
UC1	21 x 21	8	3528
US2	21 x 21	80	35 280
UV2	21 x 21	1	441
UC2	13 x 13	33	5 577
US3	13 x 13	97	16 393
UV3	13 x 13	1	169
UC3	7 x 7	64	3 136
US4	3 x 3	46	414
UV4	3 x 3	1	9
UC4	1 x 1	35	35
Celkem	-	380	70 045

3.4.2 Propojení sítě

Sítě Neocognitron se vyznačují vysokou hustotou spojů mezi jednotlivými buňkami. Každá buňka je spojena se skupinou buněk, tzv. *připojovací oblastí*, předchozí vrstvy. Tato připojovací oblast má nejčastěji rozměr 3x3 nebo 5x5 buněk. Buňky vstupní vrstvy nebo C-vrstvy předchozí úrovně jsou propojeny s buňkami na S-vrstvě a V-vrstvě. Dále jsou

jednotlivé V-buňky spojeny se S-buňkami příslušné vrstvy. Nakonec jsou S-buňky spojeny s C-buňkami. Každé propojení v síti má určitou váhu. Zde existují čtyři typy vah: a-váhy, b-váhy, c-váhy a d-váhy[5].



Obr. č. 12: Váhy jednotlivých spojení

a-váhy

Jsou modifikovány učením. Spojují připojovací oblast C-ploch předešlé úrovně s příslušnými S-buňkami následující úrovně. Váhy jsou sdílené, což znamená, že všechny S-buňky v jedné S-ploše mají stejnou hodnotu a-váhy.

b-váhy

Také jsou modifikovány učením. Spojují V-buňky s příslušnými S-buňkami. B-váhy jsou také sdílené, takže všem S-buňkám stejné S-plochy přísluší stejná hodnota b-váhy. K jedné S-buňce přísluší pouze jedna b-váha, jelikož je počet V a S-buněk v jedné ploše totožný.

c-váhy

Hodnota c-vah je určena pevně při konstrukci sítě. Spojují připojovací oblast C-buněk předchozí úrovně s V-buňkami úrovně následující. Tyto váhy bývají nejčastěji nastaveny tak, že nejméně omezují přenos informací ze středu připojovací oblasti a postupně k jejím okrajům tento přenos tlumí.

d-váhy

Jsou spojením mezi připojovací oblastí S-buněk a příslušnou C-buňkou. Váhy jsou pevně nastaveny při konstrukci a stejně jako c-váhy tlumí přenos informace směrem k okrajům připojovací oblasti.

3.4.3 Buňky

Jak již bylo zmíněno, v neuronových sítích Neocognitron jsou čtyři druhy buněk. Prvními a nejjednoduššími buňkami jsou **receptorové buňky**, které se nachází ve vstupní vrstvě. Jejich funkcí je uchovávat informaci o jednom pixelu vstupního obrazu. Počet receptorových buněk nám tedy udává maximální možné rozlišení vstupního vzoru.

3.4.3.1 S-buňky

Dalším druhem buněk jsou S-buňky. Jejich úkolem je detekce příznaků na předem daných pozicích ve vrstvě. Excitační informaci získává každá S-buňka ze svých připojovacích oblastí C-ploch předešlé úrovně a dále získává také inhibiční informaci od příslušné V-buňky. Tato informace udává průměrnou aktivitu v příslušné oblasti.

Formálně se výstupní hodnota S-buňky vypočítá dle vztahu[7]:

$$u_{Sl}(n, k) = r_l(k) \cdot \varphi \left[\frac{1 + \sum_{\kappa=1}^{\kappa_{Cl}-1} \sum_{v \in A_l} a_l(v, \kappa, k) \cdot u_{Cl-1}(n + v, k)}{1 + \frac{r_l(k)}{1 + r_l(k)} \cdot b_l(k) \cdot u_{Vl}(n)} \right]$$

kde:

l je číslo úrovně,

n souřadnice buňky,

k číslo plochy v rámci vrstvy,

v souřadnice buňky v rámci připojovací oblasti,

κ_{Cl} počet C-ploch v C-vrstvě,

A_l připojovací oblast S-buňky,

r_l selektivita,

φ prahová přenosová funkce,

a_l a-váha,

b_l b-váha,

u_{Vl} výstupní hodnota V-buňky.

Důležitým je zde především parametr r_l , neboli **selektivita**. Její velikost lineárně ovlivňuje význam inhibiční vstupní složky, čímž se ovlivňuje schopnost rozlišovat deformované

vzory. Při vysoké hodnotě selektivity se zvýší přesnost rozpoznání, avšak sníží se schopnost reagovat na odlišnější vzory. Naopak snižováním selektivity roztřídíme i deformované vzory, avšak sníží se přesnost roztřídění. Každá S-vrstva může mít specifickou hodnotu selektivity. Správné nastavení této hodnoty je nezbytné pro optimální funkci neuronové sítě.

3.4.3.2 V-buňky

Hlavním úkolem je předat informaci o průměrné aktivitě připojovací oblasti C-ploch, příslušné S-buňce. Výstupní hodnota V-buňky je popsána vztahem[7]:

$$u_{Vl}(n) = \sqrt{\sum_{\kappa=1}^{\kappa_{Cl-1}} \sum_{v \in Al} c_l(v) \cdot u_{Cl-1}^2(n+v, \kappa)}$$

kde:

c_l je c-váha.

3.4.3.3 C-buňky

Úkolem C-buněk je zajistit odolnost proti natočení a posunutí vzorů. Aktivita buňky je přímo úměrná d-váze a hodnotě S-buněk v připojovací oblasti. Připojovací oblasti jednotlivých C-buněk na S-vrstvě se překrývají, takže jedna S-buňka ovlivňuje více C-buněk. Tímto je zaručena určitá odolnost, jelikož výsledkem C-plochy je rozmazaný obraz vzoru ze S-plochy. Formálně je výstupní hodnota C-buňky popsána vztahem:

$$u_{Cl}(n, k) = \psi \left[\sum_{\kappa \in K_{Sl}} \sum_{v \in Dl} d_l(v) \cdot u_{Sl}(n+v, \kappa) \right]$$

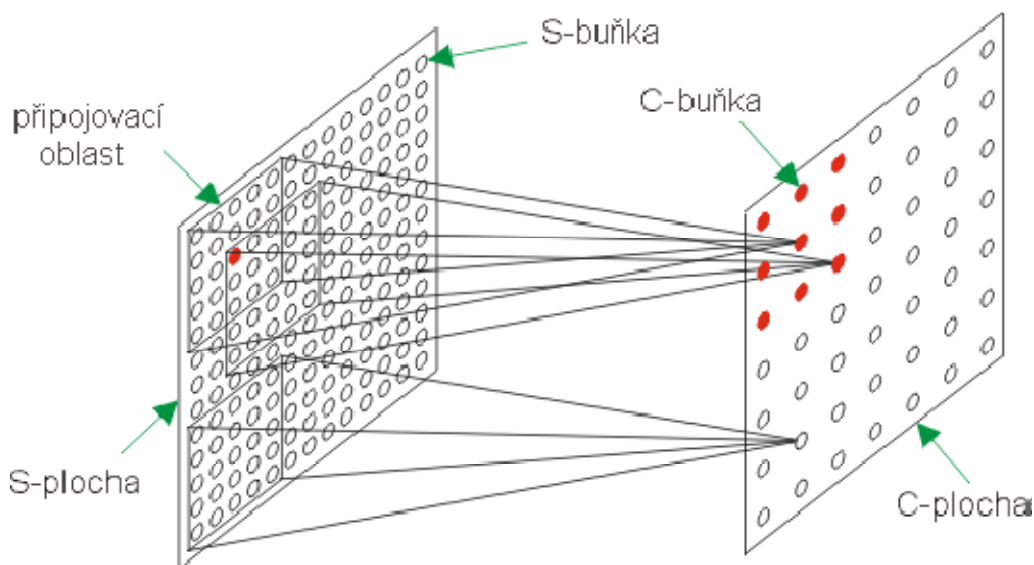
kde:

d_l je d-váha,

K_{Sl} počet S-ploch v S-vrstvě,

D_l připojovací oblast C-buňky,

ψ přenosová funkce.



Obr. č. 13: Ukázka překrytí připojovacích oblastí C-buněk.

3.4.4 Učení Neocognitronu

Učení spočívá v předkládání trénovacích vzorů a nastavování modifikovaných vah (a-vah, b-vah) tak, aby síť byla schopna úspěšně rozpoznat tyto vzory. Pro učení je tedy potřeba vytvořit trénovací vzory, které by měly přímo odpovídat rozpoznávaným znakům.

Na začátku učení jsou vynulovány veškeré a-váhy a b-váhy. Učení probíhá od nejnižších vrstev. Nejprve se tedy učí jednotlivé S-plochy S-vrstvy první úrovně. Ve vybrané S-ploše zvolíme jednu buňku tzv. *semínko* (seed) a na vstup se vloží požadovaný vzor. Pro semínko určíme příslušné váhy. Jelikož jsou váhy v jedné S-ploše sdílené, automaticky se dle vah semínka nastaví ostatní buňky S-plochy. Obdobně pokračujeme pro další S-plochy. Poté co jsou naučeny všechny S-plochy dané S-vrstvy, pokračuje se na S-vrstvu vyšší úrovně.

Pro každou S-plochu je obvykle jeden trénovací vzor, ale může jich být i více. Pro každou úroveň obsahuje trénovací vzor jen určitý příznak, charakteristickou část daného vzoru, kterou bude síť v daném kroku rozpoznávat. Příznaky by měly být nastaveny tak, aby pro danou úroveň detekovaly to, co jednotlivé znaky rozděluje[5].

3.4.5 Vybavování Neocognitronu

Vybavování je samotným procesem rozpoznávání předložených vzorů. Výsledkem je určení kategorie, do které daný vzor patří. Nejprve tedy vložíme vzor na receptorové buňky vstupní vrstvy. Následně se určí hodnota V-buněk V-vrstvy první úrovně. Poté

mohou být vypočteny hodnoty S-buněk, které detekují nejjednodušší příznaky. Dále dojde k zpracování C-vrstvou, která obraz rozmaže a poté se opět postup analogicky opakuje ve vyšší vrstvě. Výstupem C-buněk nejvyšší vrstvy je míra podobnosti předloženého vzoru s kategorií, již daná C-buňka reprezentuje[5].

4 C# 2008

C# je poměrně nový programovací jazyk, který byl navržen pro práci s .NET Frameworkem. Pomocí jazyka C# lze například napsat dynamickou webovou stránku, klasickou aplikaci s podporou režimů online i offline. Jazyk C# nabízí možnost pro tvorbu téměř libovolných typů programů nebo komponent, které lze použít přímo v systému Windows. Celá technologie .NET Frameworku obsahuje řadu možností, jak programátorům ulehčuje práci. Zde je uveden základní přehled[3]:

- **Objektově orientované programování:** Platforma .NET Framework a jazyk C# jsou již od začátku kompletně založeny na objektově orientovaných principech.
- **Dobrý návrh:** Knihovna základních tříd je od základu navržena vysoce intuitivním způsobem.
- **Jazyková nezávislost:** V technologii .NET jsou všechny jazyky, jako Visual Basic .NET, C#, J# a spravované C++, překládány do společného *zprostředkujícího jazyka*. To znamená, že jazyky mohou spolupracovat způsobem, který dříve nebyl k dispozici.
- **Lepší podpora dynamických webových stránek:** Prostředí ASP sice poskytovalo značnou pružnost, ale zároveň nebylo příliš efektivní, protože používalo interpretované skriptovací jazyky. Nedostatečný objektově orientovaný návrh také často vedl ke vzniku nepřehledného kódu APS. Technologie .NET nabízí integrovanou podobu webových stránek pomocí nové technologie ASP.NET. v rámci ASP.NET je kód stránek překládán a lze jej psát v jazyce vysoké úrovně kompatibilním s technologií .NET, jako je např. C#, J# nebo Visual Basic.
- **Účinný přístup k datům:** Sada komponent .NET, souhrnně označovaná jako ADO.NET, zajišťuje efektivní přístup k relačním databázím a různým zdrojům dat. Jsou také k dispozici komponenty, které poskytují přístup k systému souborů a k adresářům. Zejména je do technologie .NET integrována podpora jazyka XML, což umožňuje manipulaci s daty, která lze importovat nebo exportovat na jiné platformy než Windows.
- **Sdílení kódu:** Technologie .NET zcela mění způsob sdílení kódu mezi aplikacemi. Zavádí koncepci **sestavení**, která nahrazují tradiční knihovny DLL. Sestavení mají formální funkce pro správu verzí a je možné, aby vedle sebe existovaly různé verze sestavení.

- **Zlepšené zabezpečení:** Každé sestavení může také obsahovat integrované informace o zabezpečení, jež mohou přesně určovat, který uživatel nebo která kategorie uživatelů či procesů smí volat určité metody definovaných tříd. Tak lze získat velmi podrobnou kontrolu nad tím, jak lze zaváděná sestavení používat.
- **Instalace s nulovým dopadem:** Existují dva typy sestavení: sdílená a soukromá. Sdílená sestavení jsou společné knihovny dostupné všem programům, zatímco soukromá sestavení jsou určena pouze k použití s konkrétním softwarem. Soukromé sestavení je zcela soběstačné, takže se zjednodušuje proces instalace. Nevznikají žádné položky registru. Stačí umístit příslušné soubory do správné složky v systému souborů.
- **Podpora webových služeb:** Technologie .NET zahrnuje plně integrovanou podporu vývoje webových služeb, které lze vytvářet stejně snadno jako libovolný jiný typ aplikací.

Programování v samotném jazyku C# lze rozdělit do několika kapitol.

- Deklarace proměnných
- Inicializace a obor proměnných
- Předdefinované datové typy jazyka C#
- Výčty
- Jmenné prostory
- Metoda `Main()`
- Základní možnosti překladače C# pro příkazový řádek
- Realizace vstupních a výstupních konzolových operací pomocí třídy `System.Console`
- Práce s komentáři a dokumentačními nástroji
- Direktivy preprocesoru
- Doporučené pokyny a konvence správného programování v C#

Následující kapitoly se některým tématům budou věnovat podrobněji.

4.1 Předdefinované datové typy

Datové typy jsou v programovacím jazyku C# rozdělené na dvě části:

- Hodnotové typy

- Referenční typy

4.1.1 Referenční typy

Referenční typy jsou takové proměnné, jež neobsahují samotnou hodnotu, ale pouze odkaz na hodnotu. Tyto typy jsou uloženy na různých místech paměti. Hodnotové typy se ukládají do oblasti označované jako zásobník a referenční typy jsou umístěny v oblasti s názvem řízená halda.

Následující kód ukazuje použití referenčního typu. V případě tohoto kódu je předpokládáno, že je definovaná třída s názvem *Vector*. *Vector* je referenční typ a obsahuje členskou proměnnou typu *int* s názvem *Value*.

```
Vector x, y;  
x = new Vector();  
x.Value = 30;  
y = x;  
Console.WriteLine(y.Value);  
y.Value = 50;  
Console.WriteLine(x.Value);
```

Je důležité si uvědomit, že po spuštění tohoto kódu bude existovat pouze jeden objekt typu *Vector*. Proměnné *x* i *y* odkazují na umístění v paměti, kde je tento objekt uložen. Protože *x* a *y* jsou proměnné referenčního typu, deklarace každé z těchto proměnných pouze vyhradí odkaz – nevytvoří instanci objektu daného typu. Přesně to odpovídá deklaraci ukazatele v jazyce C++ nebo objektového odkazu v jazyce Visual Basic. Ani v jednom případě není ve skutečnosti vytvořen objekt. Chcete-li vytvořit objekt, musí se použít klíčové slovo *new*, jak bylo ukázáno. Vzhledem k tomu, že proměnné *x* a *y* odkazují na stejný objekt, změny proměnné *x* ovlivní proměnnou *y* a naopak. Program tedy zobrazí číslo 30 a poté číslo 50.

4.1.2 Hodnotové typy

Hodnotové typy jsou typy, jež přímo obsahují hodnotu. Tyto proměnné jsou uloženy v zásobníku. Jejich použití je jednoduché:

```
int i, j;  
i = 20;  
j = i;
```

V uvedeném příkladu budou obě proměnné *i* a *j* obsahovat stejnou hodnotu.

Vestavěné hodnotové typy reprezentují základní hodnoty celočíselných typů[3]:

Tab. č. 3: Hodnotové typy C#

Název	Typ systému CTS	Popis
sbyte	System.SByte	8bitové celé číslo se znaménkem
short	System.Int16	13bitové celé číslo se znaménkem
int	System.Int32	32bitové celé číslo se znaménkem
long	System.Int64	64bitové celé číslo se znaménkem
byte	System.Byte	8bitové celé číslo bez znaménka
ushort	System.UInt16	16bitové celé číslo bez znaménka
uint	System.UInt32	32bitové celé číslo bez znaménka
ulong	System.UInt64	64bitové celé číslo bez znaménka
float	System.Single	32bitové reálné číslo s jednoduchou přesností
double	System.double	64bitové reálné číslo s dvojitou přesností
decimal	System.Decimal	128bitové celé číslo uložené v desítkové soustavě s vysokou přesností
bool	System.Boolean	true nebo false
char	System.Char	Představuje jeden 16bitový znak v kódování Unicode
object	System.Object	Kořenový typ, od kterého jsou odvozeny všechny další typy v systému CTS
string	System.String	Znakový řetězec v kódování Unicode

Datový typ `System.String` je výjimkou v jazyku C#. Chová se stejně jako hodnotové typy, lze do něj vkládat hodnotu přímým přiřazením hodnoty do proměnné. Navzdory tomuto vkládání je referenčním datovým typem uloženým v haldě. I přes tuto vlastnost je datový typ upraven tak, že se chová jako hodnotový typ.

4.2 Řízení běhu programu

V této kapitole budou rozebrány nejzákladnější a nejpotřebnější příkazy pro řízení běhu programu. Bez těchto příkazů bychom se dostaly k situaci, že by se všechny programy prováděly sekvenčně a nebylo by možno reagovat na podněty.

4.2.1 Příkaz if

Pro účely podmíněného větvení převzal jazyk C# z jazyků C a C++ konstrukci IF...ELSE. Syntaxe by měla být dostatečně intuitivní pro každého, kdo někdy programoval v procedurálním jazyce:

```
if (podmínka)
    příkaz (příkazy)
else
    příkaz (příkazy)
```

Pokud je potřeba v některém z příkazů vnořených do if provést několik příkazů, je nutné tyto příkazy spojit do bloku pomocí složených závorek ({...}).

Použitá syntaxe se podobá jazykům C++ a Java, ale liší se od jazyka Visual Basic. Není zde totiž žádná klauzule EndIf. V jazyce C# se předpokládá, že každá klauzule if obsahuje právě jeden příkaz či jeden blok příkazů. Pokud není potřeba vyhodnocovat podmínku opačnou, není potřeba ani uvádět klauzuli else.

Pokud je potřeba vyhodnotit více podmínek, lze zkombinovat klauzule else if:

```
if (podmínka1)
    příkaz1
else if (podmínka2)
    příkaz2
else if (podmínka3)
    příkaz3
.
.
.
```

Počet příkazů else if, které lze přidat do příkazu if, není nijak omezen.

4.2.2 Příkaz switch

Příkaz switch...case se hodí při výběru jedné výkonné větve ze skupiny větví, které se vzájemně vylučují. Pro programátory v jazycích C++ a Java to není nic nového a podobá se to i Select Case ve Visual Basicu.

Příkaz switch začíná klíčovým slovem switch, za kterým následuje sada klauzulí case. Pokud hodnota výrazu za klíčovým slovem switch nabývá jedné z hodnot uvedených vedle klíčového slova case, provede se kód bezprostředně následující za touto klauzulí case. Jedná se o jednu ze situací, kdy se nemusí spojovat příkazy do bloků pomocí složených závorek. Místo toho se označuje konec kódu pro každou z klauzulí case příkazem break. Do příkazu switch se může také zahrnout implicitní klauzule default, která je prováděna, když se výraz nerovná žádné z hodnot uvedených v klauzulích case. Lze to vidět v následující ukázce kódu[3]:

```
switch (integerA)
{
    case 1:
        Console.WriteLine(„integerA =1“);
        break;
    case 2:
        Console.WriteLine(„integerA =2“);
        break;
    case 3:
        Console.WriteLine(„integerA =3“);
        break;
    default:
        Console.WriteLine(„integerA není 1,2 ani 3“);
        break;
}
```

Důležité je, že hodnoty u návěští case musí být zadány konstantními výrazy: proměnné zde nejsou povoleny.

4.2.3 Cyklus for

Cyklus for v jazyce C# umožňuje opakovat skupinu příkazů. Před každým opakováním se testuje platnost určité podmínky (tzv. podmínky opakování). Používá se syntaxe

```
for (inicializátor; podmínka; krok)
    příkaz (nebo blok příkazů, tzv. tělo cyklu)
```

kde:

- Jako inicializátor slouží podmínka vyhodnocená před prvním spuštěním cyklu. Obvykle se inicializuje lokální proměnná jako čítač cyklů (označuje se také jako parametr nebo řídící proměnná cyklu).
- Podmínka (podmínka opakování) je výraz, který se kontroluje před každou další iterací (opakováním) cyklu. Aby proběhla další iterace, musí být výsledkem vyhodnocení podmínky hodnota true.
- Krok označuje výraz, který se vyhodnotí po každé iteraci (zpravidla dojde ke zvýšení čítače cyklu). Cyklus končí, když bude mít podmínka hodnotu false.

Cyklus for je tzv. cyklus podmínkou na počátku, protože podmínka opakování se vyhodnocuje před provedením příkazů těla cyklu. Má-li tedy podmínka opakování na počátku hodnotu false, tělo cyklu se vůbec neprovede.

Cyklus for se dokonale hodí k opakování příkazu nebo bloku příkazů, kde je předem určen počet opakování. Následující příklad představuje typické použití cyklu for. Tento kód vypíše všechna celá čísla od 0 do 99:

```
for ( int i = 0; i < 100; i = i+1)
{
    Console.WriteLine(i);
}
```

Cykly for se dost často vnořují do sebe. Díky tomuto vnoření lze například procházet dvourozměrná pole.

4.2.4 Cyklus while

Cyklus while se shoduje s cyklem while v jazycích C++ a Java a s cyklem While...Wend v jazyce Visual Basic. Podobně jako cyklus for je i while cyklem s podmínkou na počátku. Syntaxe je podobná, ale cyklus while obsahuje pouze jeden výraz[4]:


```
while(podmínka)
    přímka (nebo blok příkazů);
```

Na rozdíl od cyklu for se cyklus while nejčastěji používá k opakování příkazu nebo bloku příkazů, když není před začátkem cyklu znám počet opakování. Příkaz uvnitř cyklu while obvykle při určité iteraci nastaví logický příznak na hodnotu false a tím způsobí ukončení cyklu, jak to dokládá následující ukázka:

```
bool condition = false;
while (!condition)
{
    DoSomeWork();
    condition = CheckCondition();
}
```

4.2.5 Cyklus do...while

Cyklus do...while je variantou cyklu while s podmínkou na konci. Plní stejnou funkci a má stejnou syntaxi jako cyklus do...while v jazycích C++ a Java a funkčně se shoduje s cyklem Loop...While v jazyce Visual Basic. To znamená, že testovací podmínka cyklu se vyhodnotí po provedení těla cyklu. Díky tomu jsou cykly do...while vhodné v situacích, kdy je nutno provést blok příkazů alespoň jednou jako je uvedeno v následujícím příkladu:

```
bool condition;
do
{
    MustBeCalledAtLeastOnce();
    Condition = CheckCondition();
} while (condition);
```

4.2.6 Cyklus foreach

Cyklus foreach je posledním z příkazů cyklů v C#. zatímco jiné varianty cyklů byly k dispozici již v prvních verzích jazyků C a C++, příkaz foreach je novinkou (vypůjčenou z jazyka Visual Basic), která je mimochodem velmi vítaná.

Cyklus foreach umožňuje iterovat(projít) jednotlivé položky v kolekci. Prozatím se nebudeme starat o to, co kolekce obnáší. Nyní si pouze uvedeme, že se jedná o objekt, který obsahuje jiné objekty. Následující část kódu zobrazuje použití cyklu foreach[4]:

```
foreach (int temp in arrayOfInts)
{
    DoSomething();
}
```

5 LINQ

Lze najít několik popisů integrovaného jazyka pro dotazování (Language Integrated Query, LINQ), mezi nimi figurují např. tyto:

- LINQ je jednotný programovací model pro libovolný druh dat. LINQ umožňuje dotazovat se na data a pracovat s nimi v konzistentním modelu, nezávisle na datovém zdroji.
- LINQ je dalším nástrojem pro začlenění dotazů SQL do kódu.
- LINQ je další datovou vrstvou pro abstrakci dat.

Všechny tyto definice jsou do určité míry správné, ale každá z nich se zaměřuje pouze na jeden aspekt LINQ. LINQ umí mnohem více, než jen začlenit dotazy SQL, používá se mnohem snáze než „jednotný programovací model“ a zdaleka není jen další množinou pravidel pro modelování dat.

5.1 Co je LINQ?

LINQ je programovací model, který zavádí dotazy jako prvořadý princip do všech jazyků Microsoft :NET. Úplná podpora LINQ však vyžaduje určitá rozšíření používaného jazyka. Tato rozšíření zvyšují efektivitu vývojářů a poskytují kratší, smysluplnější a jasnější syntaxi pro manipulaci s daty.

LINQ nabízí metodologii, která zjednodušuje a sjednocuje implementaci libovolného typu přístupu k datům. LINQ nenutí používat specifickou architekturu, využívá implementaci několika existujících systémů pro přístup k datům, např.:

- RAD/prototyp
- Klient/server
- N-vrstev
- Chytrý klient

LINQ se poprvé objevil v září 2005 jako technický náhled. Od té doby se vyvinul z rozšíření Microsoft Visual Studio 2005 do integrální součásti .NET Frameworku 3.5 a Visual Studio 2008, které byly vydány v listopadu 2007. První vydaná verze LINQ přímo podporovala několik datových zdrojů [6].

V současné době slouží tato technologie jako přístup k několika druhům datových zdrojů:

- LINQ pro objekty
- LINQ pro ADO.NET
 - LINQ pro SQL
 - LINQ pro datové sady
 - LINQ pro entity
- LINQ pro XML

5.2 Implementace LINQ

LINQ je technologie, která zastřešuje mnoho datových zdrojů. Některé z těchto zdrojů jsou součástí implementací LINQ, které Microsoft poskytuje jako součást .NET Frameworku, jenž dále obsahuje LINQ pro entity.

Každá z těchto implementací je definována pomocí množiny rozšiřujících metod obsahujících operátory pro to, aby LINQ mohl pracovat s konkrétním datovým zdrojem. K těmto funkcím se přistupuje pomocí významných jmenných prostorů.

5.2.1 LINQ pro objekty

LINQ pro objekty slouží pro manipulaci s kolekcemi objektů, které lze navzájem provázat a vytvořit tak graf. Z určitého úhlu pohledu je LINQ pro objekty výchozí implementací, kterou používá dotaz LINQ. LINQ pro objekty lze zpřístupnit pomocí jmenného prostoru *System.Linq*.

Bylo by chybou domnívat se, že dotazy LINQ pro objekty se omezují jen na kolekce dat generovaných uživatelem. Nesprávnost tohoto předpokladu lze odhalit pohledem na následující výpis, který ukazuje dotaz LINQ nad informací získanou ze souborového systému. Seznam všech souborů v daném adresáři se načítá do paměti a poté filtruje v dotazu LINQ podle velikosti souboru[4].

```
string tempPath = Path.GetTempPath();
DirectoryInfo dirInfo = new DirectoryInfo( tempPath );
var query =
    from f in dirInfo.GetFiles()
    where f.Length > 10000
    orderby f.Length descending
    select f;
```

5.2.2 LINQ pro ADO.NET

LINQ pro ADO.NET obsahuje různé implementace LINQ, které pracují s relačními daty. Obsahuje také další technologie, jež jsou specifické pro jednotlivé trvalé vrstvy:

- **LINQ pro SQL** Obsluhuje mapování mezi vlastními typy v .NET a schématem fyzické tabulky.
- **LINQ pro Entity V mnoha** směrech se podobá LINQ pro SQL. Ale namísto práce s fyzickou databází jako trvalou vrstvou používá konceptuální datový model entit (Entity Data Mode, EDM). Výsledkem je abstraktní vrstva, která je nezávislá na fyzické datové vrstvě.
- **LINQ pro datové sady Umožňuje** v LINQ dotazy do objektu typu *DataSet*

LINQ pro SQL a LINQ pro Entity se v mnohém podobají, protože oba typy přistupují k relační databázi a pracují s entitami objektů v paměti, které reprezentují externí data. Hlavním rozdílem je, že operují na odlišné úrovni abstrakce. Zatímco LINQ pro SQL se váže na fyzickou strukturu databáze, LINQ pro entity pracuje nad konceptuálním modelem (obchodní entity), jenž se může od fyzické struktury (tabulky v databázi) výrazně odlišovat.

Důvodem pro tyto odlišné možnosti přístupu k relačním datům v LINQ je skutečnost, že pro přístup k databázi se dnes používají různé modely. Některé organizace provozují veškerý přístup přes uložené procedury včetně všech dotazů do databáze a vůbec nepoužívají dynamické dotazy. Mnoho dalších využívá uložené procedury na vkládání, aktualizace či mazání dat a pro dotazy vytváří dynamicky příkazy SELECT. Někteří chápou databázi jako jednoduchý objekt v trvalé vrstvě, zatímco další do databáze vkládají určitou obchodní logiku prostřednictvím spouští a uložených procedur. Jazyk LINQ se snaží nabídnout pomoc a vylepšení přístupu k databázi, aniž by někoho nutil přijímat jediný všeobsahující model[6].

5.2.3 LINQ pro XML

LINQ pro XML obsahuje poněkud odlišnou syntaxi, která pracuje s daty XML, a umožňuje dotazy a práci s daty. Obzvláště významnou podporu LINQ pro XML nabízí Visual Basic 2008, jehož integrální součástí jsou výrazy XML. Tato rozšířená podpora zjednodušuje kód potřebný pro manipulaci s daty. Ve Visual Basicu 2008 je možno napsat následující kód:

```
Dim book =  
    <Book Title="Programování LINQ">  
        <%= From person In team _  
            Where person.Role = „Autor“ _  
            Select <Autor><%= person.Name %></Autor> %>  
    </Book>
```

Odpovídající dotaz v C# 3.0 by měl takovýto tvar:

```
dim book =  
    new XElement( "Book",  
        new XAttribute( "Title", "Programování LINQ" ),  
        from person in team  
        where person.Role == "Author"  
        select new XElement( "Author", person.Name ) );
```

5.3 Syntaxe LINQ

Integrovaný jazyk pro dotazování umožňuje vývojářům dotazovat se a spravovat sekvence položek (objekty, entity, databázové záznamy, uzly XML apod.) v jejich softwarovém řešení pomocí běžné syntaxe a jednoho programovacího jazyka bez ohledu na charakter zpracovávaných položek. Klíčovou vlastností LINQ je jeho integrace v běžně používaných programovacích jazycích, kterou umožňuje používání společné syntaxe pro všechny druhy obsahu.

LINQ vychází z množiny dotazovacích operátorů definovaných jako rozšiřující metody, které pracují s jakýmkoliv objektem implementujícím rozhraní *IEnumerable<T>* nebo *IQueryable<T>*.

Tento přístup dělá z LINQ obecné dotazovací prostředí, protože rozhraní *IEnumerable<T>* či *IQueryable<T>* implementuje mnoho kolekcí či typů a jakýkoliv vývojář si může napsat

svou vlastní implementaci. Tato dotazovací infrastruktura je rovněž hojně rozšiřitelná. S danou architekturou rozšiřujících metod mohou vývojáři specializovat chování metody na základě typu dat, na která se dotazují. Například LINQ pro SQL i LINQ pro XML mají specializované operátory LINQ, které umí zpracovávat relační data resp. uzly XML.

Jako příklad je níže uveden krátký zdrojový kód pro jazyk C# 3.0. Jde o vyfiltrování záznamů z pole objektů podle klíčového slova.

```
using System;
using System.Linq;
using System.Collection.Generic;

public class Developer
{
    public string Name;
    public string Language;
    public int Age;
}

class App
{
    static void Main()
    {
        Developer[] developers = new Developer[] {
            new Developer {Name = „Paolo“, Language = „C#“},
            new Developer {Name = „Marco“, Language = „C#“},
            new Developer {Name = „Frank“, Language = „VB.NET“}};
        var developersFiltered =
            from d in developers
            where d.Language == „C#“
            select d.Name;
    }
}
```

Tučná část kódu ve výše uvedeném příkladu se nazývá dotazovací výraz. V některých implementacích LINQ se reprezentaci těchto dotazů v paměti se říká strom výrazu.

Uvedený dotazovací výraz vypadá podobně jako příkaz SQL, ale jeho styl se poněkud odlišuje. Ukázkový výraz se skládá z příkazu výběru:

```
select d.Name
```

aplikovaného na množinu položek:

```
from d in developers
```

kde klauzule *from* míří na jakoukoliv instanci třídy, která implementuje rozhraní *IEnumerable<T>*. Na výběr se aplikuje konkrétní podmínka filtrování:

```
where d.Language == „C#“
```

Tyto klauzule se překládají v kompilátorech jazyků do volání rozšiřujících metod, které se sekvenčním způsobem aplikují na cíl dotazu. Hlavní knihovna LINQ, definovaná v knihovně *System.Core.dll*, definuje množinu rozšiřujících metod seskupených podle cíle a účelu. Knihovna například obsahuje třídu s názvem *Enumerable* definovanou ve jmenném prostoru *System.Linq*, která definuje rozšiřující metody aplikovatelné na instance typů implementujících rozhraní *IEnumerable<T>*.

Tento zápis ale ne vždy vyhovuje potřebám a také přehlednosti. Proto existuje ještě jeden způsob zápisu dotazu. Tento zápis využívá tzv. lambda výrazů. Výše uvedený zápis lze přepsat jako následující dotaz:

```
IEnumerable<string> developersFilter =  
    developers  
    .Where(d => d.Language == „C#“)  
    .Select(d => d.Name);
```

5.3.1 Klauzule from

Definuje datový zdroj dotazu či poddotazu a proměnnou intervalu, jež určuje všechny jednotlivé elementy ve zdroji, na něž má dotaz směřovat. Datovým zdrojem může být libovolná instance nějakého typu, jenž obsahuje rozhraní *IEnumerable*, *IEnumerable<T>* nebo *IQueryable<T>*, které implementuje rozhraní *IEnumerable<T>*.

Kompilátor jazyka odvodí typ proměnné intervalu z typu datového zdroje. Jestliže je například datový zdroj typu *IEnumerable<string>*, proměnná intervalu bude typu *string*.

V případech, kdy se nepoužívá silně typový datový zdroj, měly by se pro proměnné intervalu explicitně stanovit typ[6].

V dotazech lze použít vícero klauzulí `from` a spojovat tak různé datové zdroje. V tomto případě lze zvolit i prvek pomocí, kterého se oba datové zdroje svážou, tak jako ukazuje následující část kódu.

```
var ordersQuery =  
    from c in customers  
    from o in c.Orders  
    select new { c.Name, o.IdOrder, o.EuroAmount };
```

5.3.2 Klauzule `where`

Klauzule `where` udává filtrovací podmínku, jež se aplikuje na datový zdroj. Predikát vyhodnotí logickou podmínku pro každou položku v datovém zdroji a vybere pouze ty položky, kde má podmínka hodnotu `true`. V jednom dotazu lze použít více klauzulí `where` nebo klauzuli `where` s více predikáty, které je možné kombinovat pomocí logických operátorů `&&`, `||` a `!` (AND, OR, NOT).

Na níže uvedené části kódu lze vidět základní použití klauzule `where`:

```
var ordersQuery =  
    from c in customers  
    from o in c.Orders  
    where o.EuroAmount > 200  
    select new { c.Name, o.IdOrder, o.EuroAmount };
```

5.3.3 Klauzule `select`

Klauzule `select` určuje, jak bude vypadat výstup z dotazu. Jde o projekci, která určuje, co se má vybírat z výsledků všech předchozích klauzulí a výrazů. Ve Visual Basicu 2008 není klauzule `select` povinná. Není-li zadána, vrací dotaz typ vycházející z proměnné intervalu stanovené pro aktuální oblast působnosti dotazu. Ve výše uvedené části kódu byla klauzule `select` použita pro vytvoření nového anonymního typu složeného z vlastností či členů proměnných intervalů v dané oblasti.

5.3.4 Klauzule group a into

Klauzuli group lze použít na seskupení výsledků podle klíče. Je možné ji použít jako alternativu ke klauzuli from a umožňuje pracovat s klíči s jednou hodnotou i více hodnotami. V následujícím dotazu je uvedena ukázka, kdy se seskupují vývojáři podle programovacího jazyka:

```
var developersGrouped =  
    from d in developers  
    group d by d.Language;  
  
foreach (var group in developersGrouped)  
{  
    Console.WriteLine(„Jazyk: {0}“, group.Key);  
    Foreach (var item in group)  
    {  
        Console.WriteLine(„\t{0}“, item.Name)  
    }  
}
```

Klíčové slovo into se používá na ukládání výsledků příkazů select, group či join do dočasné proměnné. Tuto konstrukci lze použít, pokud je potřeba provést nad výsledky dodatečné dotazy[4].

5.3.5 Klauzule orderby

Klauzule orderby, jak již název napovídá, umožňuje vzestupně či sestupně třídit výsledky dotazu. Řazení lze provádět pomocí jednoho či více klíčů, které kombinují různé směry řazení. Následující část kódu vrací objednávky řazené dle parametru EuroAmount

```
var ordersSorted =  
    from c in customers  
    from o in c.Orders  
    orderby o.EuroAmount  
    select new { c.Name, o.IdOrder, o.EuroAmount };
```

5.3.6 Klauzule join

Klíčové slovo join umožňuje spojovat různé datové zdroje na základě členů zdrojů, u nichž lze zjišťovat rovnost. Pracuje to podobně jako vnitřní spojování tabulek v SQL. Nemůže porovnávat položky pomocí operátorů <, > či !=. Srovnávání lze provádět pouze pomocí speciálního slova equals, jenž má odlišné chování od operátoru ==, protože záleží na pořadí operandů.

V níže uvedeném kódu lze vidět spojení dvou datových zdrojů:

```
var categoriesAndProducts =  
    from c in categories  
    join p in produkts on c.IdCategory equals p.IdCategory  
    select new {  
        c.IdCategory,  
        CategoryName = c.Name,  
        Product = p.Description};
```

5.4 Degenerované dotazovací výrazy

Někdy je potřeba postupně procházet elementy datového zdroje bez jakéhokoliv filtrování, řazení, sdružování či vlastních projekcí. Dotaz, který vrací stejný výsledek jako původní datový zdroj, se nazývá degenerovaný dotazovací výraz.

5.5 Zpracování výjimek

Dotazovací výrazy se mohou ve své definici odkazovat na externí metody. Někdy mohou tyto metody selhat. Následující část kódu ukazuje volání metody DoSomething

```
var query =  
    from d in developers  
    let SomethingResult = DoSomething(d)  
    select new { d.name, SomethingResult };
```

Nejprve je nutno zvážit, zdali je potřeba v definicích dotazů volat vlastní metody. V případě, kdy je toto nutné je tedy potřeba zabalit kód do bloků try-catch.

Obecně je vkládání definici dotazovacího výrazu do bloku try-catch zbytečné. Navíc by se mělo ze stejného důvodu vyhnout přímému používání výsledků metod či konstruktorů jako datových zdrojů pro dotazovací výrazy a namísto toho přiřadit jejich výsledky do proměnných a toto přiřazení vložit do bloku try-catch. V následující části kódu je vidět správné uspořádání, jež by mělo být používáno[6].

```
var query =  
    from d in developers  
    let SomethingResult = DoSomething(d)  
    select new { d.Name, SomethingResult };  
try { foreach (var item in query) {...}};  
catch {...}
```

6 VYHLEDÁVACÍ ALGORITMY

V dnešní době, kdy se většina dokumentů (ať už na úřadech nebo v běžných podmínkách) vede na počítačích, a tedy v elektronické podobě, je potřeba mít dostatečně silné nástroje k tomu, aby se v daných dokumentech vyhledal nějaký text (slovo, věta apod.) v co nejkratším možném čase. Proto se řadu let vyvíjí různé algoritmy, které tento úkol s odlišnými úspěchy plní. Tyto algoritmy jsou běžnému uživateli počítače schovány v jiných větších celcích, jako jsou různé textové editory, kde je občas nutné nějaké to slovo v textu vyhledat (potažmo zaměnit za jiné). Dalším příkladem mohou být vyhledávací systémy v knihovnách, vyhledávání v databázích, všelijaké vyhledávací servery apod. Principy vyhledávání vzorků v textu se také používají i v jiných oborech než v informatice, příkladem může být například hledání vzorků v DNA šroubovici. Z toho vyplývá, že vyhledávání v textu je poměrně rozsáhlý problém, o jehož užitečnosti jistě není sporu. Proto je určitě užitečné uvést si nějaké příklady algoritmů, které tento problém řeší, popsat je a porovnat.

Algoritmy se mohou dělit do několika možných kategorií podle toho, za jakým účelem se používají nebo podle svého původu. Můžeme se tedy setkat s algoritmy, které vyhledávají pouze jeden vzorek v daném textu, v jiných případech je možno vyhledávat množinu více vzorků. Rozdíl je také v tom, že některé algoritmy naleznou pouze první výskyt vzorku v textu, jiné naleznou všechny výskyty. Existují algoritmy, které se inspirovaly i jinými obory informatiky jako je např. teorie automatů[9].

6.1 Naivní algoritmus

Naivní algoritmus spadá mezi nejjednodušší algoritmy. Jeho princip spočívá v procházení textu znak za znakem a porovnávání. Pokud najde shodu prvního znaku, kontroluje znak další. V případě, že najde neshodu, pokračuje dále ve vyhledávání. Pokud najde shodu na všech místech tak vrátí pozici. Tento algoritmus by se dal zapsat následujícím algoritmem v pseudokódu.

```
(1)  n = length(T)
(2)  m = length(P)
(3)  for s = 1 to n-m+1
(4)    if T[s..s+m-1] == P[1..m]
(5)      then print("Vzorek se nachází na pozici", s)
```

První dvě řádky obstarávají pouze uložení délky obou textových řetězců do proměnných n a m . Posouvání vzorku pod textem zajišťuje cyklus, který začíná na řádce (3). Proveďte se přesně $(n-m+1)$ -krát, což je počet pozic, na kterých se může vzorek vyskytovat. Řádka (5) jen informuje o nalezení vzorku v textu. Pro určení časové složitosti je klíčová řádka číslo 4. Zde se provádí porovnávání vzorku s daným textem. Tento pseudozápis může ve skutečnosti být `while` cyklus, který porovnává jednotlivé znaky, dokud nenarazí na neshodu nebo na konec vzorku, v tomto případě se vykoná řádka (5). Je snadné nahlédnout, že v nejhorším případě (to je že vždy projdu všechny znaky ve vzorku) se `while` cyklus provede m -krát. Časová složitost v nejhorším případě je tedy $O((n-m+1)*m)$.

6.2 Knuth-Morris-Prattův algoritmus (KMP)

Mezi prvními, kteří si uvědomili, že informace, které získává naivní algoritmus svým porovnáváním znak po znaku, mohou být velmi cenné pro návrh efektivního algoritmu, byl právě Knuth se svými společníky Morrisem a Pratem. Jejich nápad spočíval v tom, že pokud se tyto informace využijí správným způsobem, může se vzorek nad prohledávaným textem posouvat i o více než pouze o jeden znak doprava. Tím se významně zkrátí doba potřebná k prohledání textu. Také je zbytečné se v prohledávaném textu vracet ke znakům, které již byly analyzovány, tak jak to činí naivní algoritmus. Toto vrácení spočívá ve skutečnosti, že pokud při porovnávání vzorku s daným textem narazím na neshodu, vrátím se zpět na začátek vzorku a ten posunu o jedno místo doprava. Tato činnost je zřejmě zbytečná, neboť já již mám informaci o předchozích znacích, stačí ji pouze dostatečně využít.

Posun je nezávislý na prohledávaném textu. Jeho velikost určuje tzv. prefixová funkce. Díky prefixové funkci si před spuštěním vlastního vyhledávacího algoritmu předpočtu hodnoty posunů pro jednotlivé pozice ve vzorku do nějaké tabulky. Mnoho efektivních vyhledávacích algoritmů používá podobné předpočítané tabulky, které se později v průběhu vyhledávání používají. Tedy jak je patrné algoritmus KMP bude mít dvě fáze. V první fázi si z daného vzorku vypočítáme potřebné hodnoty posunů. Druhá fáze bude uskutečňovat vlastní vyhledávání[9].

Samotný algoritmus tedy pracuje v několika fázích. První fází je výpočet prefixové funkce. Následuje fáze druhá, vyhledávání. Princip je velice jednoduchý. Na začátku se zarovná

vzorek vůči textu a začne se porovnávat. Pokud se narazí na neshodu, podívá se do tabulky prefixové funkce s indexem, který odpovídá pozici ve vzorku, kde došlo k neshodě. S tabulky zjistí číslo, které udává znak vzorku který se, když příslušně zarovná vzorek k textu, bude nacházet přesně nad znakem v textu, který byl příčinou neshody (vzorek se tedy posune doprava). Vzorek se bude tímto způsobem posouvat doprava, dokud nenarazí na jeho začátek nebo nebude moci pokračovat od dané pozice dalším porovnáváním. Takto pokračuje, dokud nenarazí na konec textu. Pokud vzorek v textu najde, oznámí to a pokračuje dále.

```
(1)  n = length(T)
(2)  m = length(P)
(3)  pi = Prefix_func(P)
(4)  q = 0
(5)  for i = 1 to n
(6)    while (q > 0 && P[q+1] != T[i]) q = pi[q]
(7)    if P[q+1] == T[i] then q = q+1
(8)    if q = m then
(9)      print("Vzorek se nachází na pozici", i-m+1)
(10)   q = pi[q]
```

Prefix_func(P)

```
(1)  m = length(P)
(2)  pi[1] = 0
(3)  k = 0
(4)  for q = 2 to m
(5)    while (k > 0 && P[k+1] != P[q]) k = pi[k]
(6)    if P[k+1] == P[q] then k = k+1
(7)    pi[q] = k
(8)  return pi
```

Nyní si určíme časovou složitost algoritmu. Nejprve výpočet prefixové funkce. Základem je, že vnitřní while cyklus se provede nejvýše tolikrát jako cyklus vnější. Platí totiž, že každým průchodem vnitřním cyklem se proměnná k sníží, neboť $pi[k] < k$. Současně se proměnná k může zvýšit nejvýše jednou v každém kroku vnějšího cyklu (díky řádce (6)). Z toho evidentně plyne, že počet průběhů vnitřním cyklem je menší roven počtu kroků vnějšího cyklu. Odtud již plyne, že časová složitost výpočtu prefixové funkce je $O(m)$.

Obdobnou úvahou dojdeme k tomu, že časová složitost vyhledávací fáze je $O(n)$. Časová složitost celého algoritmu je tedy $O(m+n)$, což je výrazně lepší než u naivního algoritmu[9].

6.3 Boyer-Mooreův algoritmus (BM)

Jedná se o algoritmus podobný naivnímu algoritmu.

```
(1)  n = length(T)
(2)  m = length(P)
(3)  l = Last_Occur_func(P,m,e)
(4)  g = Good_suff_func(P,m)
(5)  s = 0
(6)  while (s <= n-m)
(7)    j = m
(8)    while (j > 0 && P[j] == T[s+j]) j = j-1
(9)    if j == 0
(10)      then print("Vzorek se nachází na pozici", s+1)
(11)      s = s+g[0]
(12)    else s = s+max(g[j], j-l[T[s+j]])
```

Odlišnosti jsou v tom, že vzorek se s textem porovnává zprava doleva, tedy odzadu (u naivního algoritmu se porovnávání provádí zleva doprava). Pokud narazím na začátek vzorku, je jasné že jsem v textu našel jeho výskyt. Zde je další rozdíl, neboť při takovém nálezu neposunu vzorek o jedno místo doprava, ale o nějakou hodnotu $g[0]$. Pokud narazím na neshodu opět posunu vzorek, ale posun nemusí mít nutně velikost jedna jako u naivního algoritmu. Ve skutečnosti je tento posun mnohdy mnohem větší. Další odlišností je, že v případě naivního algoritmu (a vlastně i v případě Knuth-Morris-Prattova algoritmu) se zpracoval každý znak prohledávaného textu aspoň jednou (u naivního algoritmu i mnohokrát). U Boyer-Mooreova algoritmu se díky tomu, že vzorek procházím od konce, a díky tomu, že vzorek v případě neshody posunu mnohdy o více než jedno písmeno doprava, na některé znaky v prohledávaném textu vůbec nedostane (přeskočí se).

Aby se tohoto úspěchu dosáhlo, používá algoritmus dvě heuristiky (v kódu jsou reprezentovány zatím záhadnými symboly g a l). Jelikož jde o heuristiky, dá se očekávat, že se časová složitost v nejhorším případě oproti naivnímu algoritmu příliš nezlepší. Naštěstí jsou tyto heuristiky tak efektivní a úspěšné, že v běžné praxi dosahují velmi

dobrých výsledků. Jak bylo řečeno výše, spousta znaků v textu se díky těmto heuristikám může jednoduše přeskočit, aniž by byly nějakým způsobem zpracovány. Na následujícím obrázku si ukážeme, jaká je základní myšlenka obou heuristik. Jen pro zajímavost v angličtině se nazývají bad-character heuristic (tedy něco jako heuristika špatného znaku. Z toho se dá odvodit, že heuristika bude nějakým způsobem souviset se znakem, který v textu způsobil neshodu.) a good-suffix heuristic (tomu v češtině odpovídá asi heuristika dobré přípony. Opět je zjevné, že bude souviset s příponami vzorku)[9].

6.4 Quicksearch

V roce 1990 publikoval člověk jménem D. M. Sunday algoritmus Quicksearch, který se od předchozích výrazně liší ve dvou věcech. Je rychlejší a mnohem jednodušší. Některé jeho rysy jsou podobné jako u Boyer-Mooreova algoritmu. U BM algoritmu totiž v nejlepším případě přeskočíme tolik znaků kolik je délka samotného vzorku. U Quicksearch se, jak uvidíme později, nejčastěji přeskakuje $m+1$ znaků (kde m je délka vzorku). Znamená to, že časová složitost v průměrném případě je méně než $O(n)$ a blíží se k $O(n/(m+1))$. To je důležité zvláště pro delší vzorky, kde je urychlení opravdu markantní. U Boyer-Mooreova algoritmu se díky tomu, že vzorek porovnáváme odzadu, v textu vracíme, což může způsobit problémy s paměťovým bufferem, které byly popsány v úvodu Knuth-Morris-Prattova algoritmu. Quicksearch nic takového nedělá, takže se jeví jako bezproblémový. Má však jiné nevýhody, které jsou popsány v další kapitole.

Myšlenka tohoto algoritmu je opravdu velice jednoduchá. Na začátku jako obvykle zarovnáme vzorek k prohledávanému textu. Stejně jako u naivního algoritmu budeme text a vzorek porovnávat znak po znaku. Postup se ale liší v případě, že objevíme neshodu mezi jednotlivými písmeny. V tomto okamžiku se podíváme na znak, který se nachází v textu přímo za koncem vzorku (testový znak). Pokud se tento znak ve vzorku na žádném místě neobjevuje, žádný posun, který by umístil jakékoli písmeno vzorku nad testový znak, nebude platný. S klidným svědomím tedy můžeme celý vzorek přemístit až za testový znak. To představuje posun o $m+1$ znaků, kde m je velikost vzorku. Tato vzdálenost je mnohem lepší než v případě předchozích algoritmů (včetně BM algoritmu, kde byl posun v tomto případě pouze m).

Pokud se testový znak ve vzorku na nějakém místě nachází (případně na více místech), posuneme vzorek o nejmenší vzdálenost takovou, že se testový znak bude shodovat se znakem v nově posunutém vzorku, který je zarovnán k testovému znaku. Většinou to bude

představovat posun o více než jeden znak. Dalším porovnáním zjistíme, jestli byl posun správný a vzorek se na této pozici již nachází. Jinak se posuneme stejným způsobem dále[9].

```
(1)  n = length(T)
(2)  m = length(P)
(3)  shift = Comp_shift(P,e)
(4)  pat = 1
(5)  s = 0
(6)  while (pat <= m && pat+s <= n)
(7)    if P[pat] == T[pat+s]
(8)      then pat = pat+1
(9)      else s = s+shift[T[s+m+1]]
(10)      pat = 1
```

Comp_shift(P,e)

```
(1)  for každý znak a z abecedy e
(2)    shift[a] = m+1
(3)  for i = 1 to m
(4)    shift[P[i]] = m-i+1
(5)  return shift
```

II. PRAKTICKÁ ČÁST

7 NÁVRH ŘEŠENÍ

Výsledná aplikace bude pracovat tak, že na vstupu bude obrázek s textem v podporovaném formátu, dostatečném rozlišení a velikostí rozpoznávaných znaků. Výstupem bude textový dokument obsahující převedené slova.

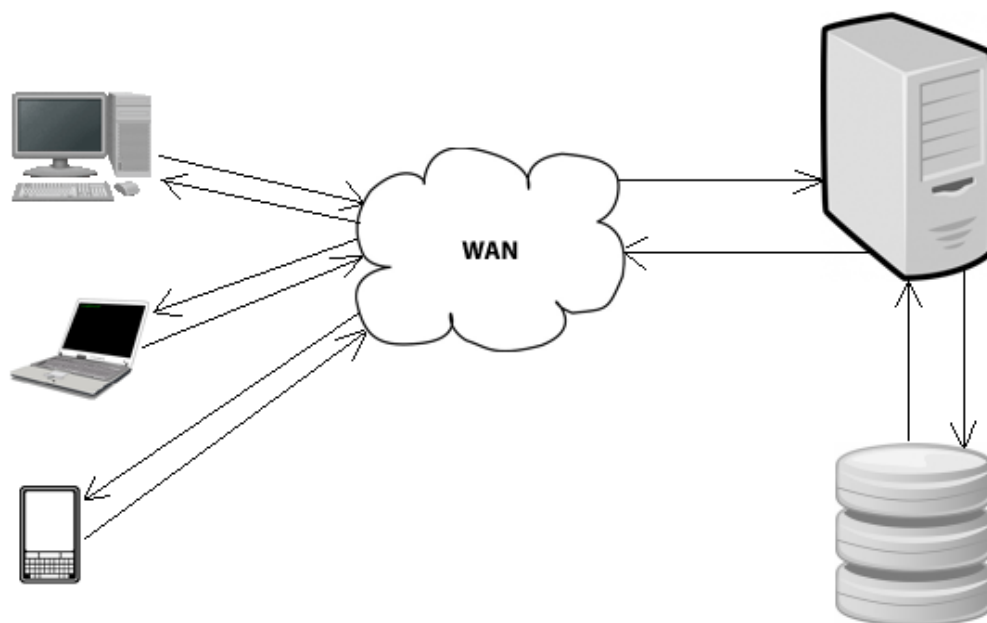
Tento proces se bude skládat z jednotlivých kroků:

- načtení vstupního obrázku
- zpracování obrázku a příprava pro detekci pozice znaků
- detekce pozic znaků
- zpracování jednotlivých znaků neuronovou sítí
- zpracování výstupu z neuronové sítě - pravděpodobnosti rozpoznání
- vyhledání známých slov ve slovníku s ohledem na pravděpodobnost rozpoznání
- korekce slov uživatelem
- uložení výstupního dokumentu nebo odeslání údajů do databáze

Samotná aplikace poté musí dodržet následující vlastnosti:

- program by měl dovolit možnost multiplatformního rozšíření
- systém by měl ochránit neuronovou síť proti jejímu odcizení
- vlastní databáze nemůže být na klientské aplikaci, potřeba dodržet malou velikost
- modifikace neuronové sítě či úprava dat v databázi by se měly projevit na všech klientech

Z těchto uvedených důvodů byl vymyšlen návrh řešení v cloudu. Toto řešení nám dovoluje na straně klienta vytvořit aplikaci, která bude objemově malá, lze ji tedy případně naportovat na mobilní zařízení, a také se usnadní její přepsání na jiné platformy. Oddělení databáze a neuronové sítě dále způsobí, že budeme mít plnou kontrolu nad parametry sítě, což lze využít k jejímu doučování či úpravách, a také databází, kde můžeme editovat samotné data. Nevýhodou tohoto systému se jeví distribuce aplikací. Bude-li provedena zásadní změna na databázi či NS, může nastat problém, kdy se starší klienti pokouší připojit např. na neexistující tabulku. Toto chování lze omezit kontrolou aktualizací při každém spuštění aplikace. Navrhovaný systém komunikace je uveden na následujícím obrázku.



Obr. č. 14: Návrh řešení v cloudu

Nyní když je vyřešena komunikace a ochrana aplikace rozebereme jednotlivé kroky samotné aplikace.

Načtení vstupního obrázku bude probíhat u klienta. Tento obrázek bude nejčastěji získán ze scanneru na straně klienta. Další možností je obrázek získat vyfocením digitálním fotoaparátem. Tento postup, ale sebou nese nevýhody ve formě špatného osvětlení, ostření znaků a rozlišení. Může se stávat, že díky špatnému osvětlení bude malý kontrastní odstup mezi znaky a pozadím a nedojde tedy ke správné detekci znaků. Také zde vzniká problém s rozlišením obrázku. Pokud bychom chtěli fotoaparátem nahradit scanner s rozlišením 300dpi, potřebovali bychom na stranu A4 fotoaparát s rozlišením 8,7 MPix. Toto je značně omezující v případě mobilních telefonů s fotoaparátem.

Zpracování obrázků bude obsahovat několik operací. Pro detekci znaků je potřeba obrázek nejprve převést do stupňů šedi a poté jeho barvy invertovat. Hranová detekce slov vyžaduje pro Cannyho algoritmus bílé znaky a černé pozadí (probíhá detekce bílých ploch ohraničených černou barvou).

Pro detekci pozic znaků není potřeba znát jejich tvar či velikost. V algoritmu je pouze potřeba zjistit výřez obrázku, ve kterém se znak nachází. Tento výřez bude standardně obdélníkový a uvnitř bude obsažen jeden znak. Tyto znaky ovšem musí být samostatné. Pokud by se jednalo o spojené znaky (typickým příkladem je psací písmo) detekovalo by

se slovo jako jeden znak. V tomto případě půjde o znaky tiskací a velké. Tyto znaky by se dotýkat neměly, a pokud tento případ nastane, bude znak špatně rozeznán. Detekované znaky je také potřeba odfiltrovat od malých objektů, ve většině případů šumu vygenerovaného scannerem.

Pokud již jsou znaky detekovány je potřeba je z obrázku vyříznout a upravit jejich velikost. Neuronová síť je v tomto případě nastavena na vstupní data v rozměru 28x28 pixelů. Jakmile jsou jednotlivé znaky zpracovány, odesílají se neuronové síti. Ta je po jednotlivých znacích zpracovává a poskytuje ke každému znaku seznam znaků s jakou pravděpodobností je rozeznán.

Po rozpoznání znaku dostaneme seznam znaků s pravděpodobností, s jakou byl znak rozeznán správně. Tyto pravděpodobnosti je potřeba zpracovat a pro každou pozici v textu vytvořit seznam znaků na tom místě. Zde se také určí, zdali znak má vysokou pravděpodobnost a bude brán v úvahu pouze s nejvyšší pravděpodobností nebo v případě nízké pravděpodobnosti bude zanedbán.

Po získání jednotlivých znaků z NS a načtení jejich pravděpodobností je potřeba rozpoznat jednotlivé slova. Vyhledání slov se bude provádět algoritmem, který bude brát v úvahu i pravděpodobnosti jednotlivých znaků. Dle výsledků vyhledání bude nabídnuto uživateli řešení rozpoznání textu.

Po vyhledání slov je celý text předán uživateli a ten provede korekci textu. V případě nerozpoznaných slov či znaků je potřeba zajistit, aby uživatel měl možnost tyto slova opravit či doplnit.

V závěru celého postupu je potřeba pouze text uložit v zařízení klienta či jeho odeslání do databáze v případě, že se jedná o formulář s vyplněnými údaji.

7.1 Převod zdrojových kódů

Jedním z cílů této práce byl převod zdrojových kódů vytvořených v prostředí MATLAB do programovacího jazyka C#. Po dohodě s vedoucím práce se realizovat nebude a to z důvodů vysokých časových nároků. Zdrojové kódy obsahují značné množství funkcí, které v jazyce C# nejsou dostupné a musely by se kompletně přeprogramovat. Licence programu MATLAB je drahá, ale při použití pouze na serveru se náklady rozpočítají mezi klienty. Díky ponechání kódů v prostředí MATLAB se dále získá přístup k dalším aktualizacím prostředí a optimalizace jeho kódu. V případě přeprogramování do C# by ladění

výkonu znamenalo velkou časovou náročnost na lidské zdroje a nebylo by možno udržet krok s vývojem samotného prostředí MATLAB. Dalším důvodem pro ponechání zdrojových kódů v prostředí MATLAB je obsažená paralelizace. Nemusí se tedy řešit složité zpracování paralelizace v C#, kde je krom samotného stanovení paralelních částí, nutno řešit přístup k jednotlivým datovým strukturám.

7.2 Paralelizace algoritmu neuronové sítě

Díky ponechání zdrojových kódů v prostředí MATLAB máme přístup k silnému výpočetnímu prostředí. První paralelní zpracování bylo dostupné již ve verzi 2007a, zde jej bylo potřeba ručně zapnout v nastavení. Od verze MATLAB 2008a je již paralelizace zapnuta automaticky a prostředí MATLAB se samotné stará o rozdělení pro více jader. MATLAB poskytuje paralelizaci pro většinu funkcí v něm obsaženou a to i pro zpracovávání základních operací. Níže je uveden příklad základních funkcí, podporujících paralelizaci v prostředí MATLAB:

- 1) Trigonometrie: ACOS(x), ACOSH(x), ASIN(x), ASINH(x), ATAN(x), ATAND(x), ATANH(x), COS(x), COSH(x), SIN(x), SINH(x), TAN(x), TANH(x), HYPOT(x,y), TAND(x)
- 2) Exponenciální funkce: EXP(x), POW2(x), SQRT(x)
- 3) Operátor x^y
- 4) Operace s maticemi: DET(X), RCOND(X), HESS(X), EXPM(X), LU(X), QR(X)
- 5) Operace s poli: LOGICAL(X), ISINF(X), ISNAN(X), INT8(X), INT16(X), INT32(X)
- 6) Další funkce: UNWRAP(x), CEIL(x), FIX(x), FLOOR(x), MOD(x,N), ROUND(x), INV(X), LSCOV(X,x), LINSOLVE(X,Y)

8 REALIZACE APLIKACE

8.1 Knihovna AForge

Knihovna AForge je opensource .NET knihovna, obsahují velké množství nástrojů. Tato knihovna je volně dostupná ke stažení a zde je popis hlavních částí:

- AForge.Imaging - knihovna pro zpracování obrazu a filtrování
- AForge.Vision - nástroje pro počítačové vidění
- AForge.Video - knihovna pro zpracování videa
- AForge.Neuro - knihovna obsahující funkce pro práci s NS
- AForge.Genetic - knihovna obsahující evoluční algoritmy
- AForge.Fuzzy - fuzzy logika
- AForge.Robotics - knihovna poskytující funkce pro některé modelové kity

V této práci je využito pouze knihovny AForge.Imaging poskytující nám nástroje pro zpracování obrazu a detekci objektů v obrazu.

8.2 Zpracování obrazu

Samotné zpracování obrazu bude probíhat vždy na straně klienta. Tento způsob byl zvolen z důvodu poměrně velké výpočetní zátěže. V případě, že by se daná zátěž akumulovala na serveru, mohlo by často docházet k velkému zpoždění a výpadkům. Také náročnost na přenesená data by byla větší. Přenos celého naskenovaného dokumentu by trval déle než přenos samostatných znaků.

Pro samotnou detekci je potřeba načtený obraz prvně zpracovat. Zpracování probíhá v následujících krocích:

- aplikace filtru na obrázek
- převedení na černobílou
- inverze barev
- detekce objektů a nastavení filtrů

Před začátkem zpracování obrazu je potřeba nainstalovat knihovny AForge. Tyto knihovny lze stáhnout jako zdrojové kódy a ty pak zpracovat v programu. To má výhody v tom, že mohou být obsaženy pouze funkce, které jsou použity. To program dělá menším a není potřeba žádných dalších souborů. Druhou cestou je použít zkompileované knihovny a ty

pouze k programu přiložit a používat je. Byla vybrána druhá cesta a pro přidání mezi resource je potřeba ve zdrojovém kódu nastavit používání těchto knihoven. To ukazuje následující část kódu.

```
using AForge.Imaging.Filters;  
using AForge.Imaging;  
using AForge;
```

Poté jsou již k dispozici všechny potřebné funkce pro práci s obrázky a jejich úpravami.

Jako první zpracování je na obraz aplikován barevný filtr. V případě černobílého dokumentu je tento krok zbytečný, ale v případě barevně skenovaného dokumentu je to krok důležitý. Dále se dá například využít pro odfiltrování formuláře. Bude-li formulář vytištěn červenou barvou a vyplněn modrou, tak je možnost tímto jednoduchým filtrem odfiltrovat červený formulář a ponechat pouze modré písmo.

```
ColorFiltering filter = new ColorFiltering();  
    filter.Red = new IntRange(240, 255);  
    filter.Green = new IntRange(240, 255);  
    filter.Blue = new IntRange(240, 255);
```

Výše uvedený kód ukazuje deklaraci a nastavení barevného filtru. Tento filtr je definován rozsahem pro každou z RGB složek. Zde lze upravovat, jaké barvy filtr bude propouštět a jaké propouštět nebude. Použití tohoto filtru poté spočívá pouze v zavolání funkce aplikace s parametrem obrázku, na který bude filtr aplikován.

```
Bitmap objectsImage = null;  
Bitmap image = new Bitmap(o.FileName);  
objectsImage = filter.Apply(image);
```

Zde je vidět aplikace filtru na obrázek. Po aplikaci toho prvního filtru již nadále nepotřebujeme pracovat s vlastnostmi obrázku. Jediné co je potřeba tak samotná obrazová data. Pro tento účel se vytvoří nový objekt, který ponese pouze informace o pixelech. Aby nedošlo ke změně původního obrázku, použije se funkce pro uzamčení bitů.

```
BitmapData objectsData = objectsImage.LockBits(new Rectangle(0, 0,  
image.Width, image.Height), ImageLockMode.ReadOnly,  
image.PixelFormat);
```

Po uzamknutí pixelů je možno nadále pracovat pouze s obrazovými daty. V tuto chvíli se na obraz aplikuje černobílý filtr. Při detekci objektů není potřeba nést barevnou informaci.

```
GrayscaleBT709 grayscaleFilter = new GrayscaleBT709();  
UnmanagedImage grayImage = grayscaleFilter.Apply(new  
UnmanagedImage(objectsData));
```

Obrázek se nyní nachází ve stavu, kdy by na něm mělo být pouze písmo a to jen v odstínech šedi. V tomto stavu je možno na obrázek aplikovat filtr inverze.

```
Invert filterI = new Invert();  
filterI.ApplyInPlace(grayImage);
```

Zde je vidět rozdílná aplikace filtru oproti aplikaci barevného filtru. Je možné použít obě dvě metody. Použijeme-li metodu Apply() výstupem bude opět obrázek ve vstupním tvaru. Použije-li se metoda ApplyInPlace() provedou se změny přímo v obrázku. Záleží tedy pouze na zvolení metody, která se více hodí k řešení problému.

Obrázek se nyní nachází ve stavu, kdy obsahuje černé pozadí a bílé (či světle šedé) písmo. Tyto znaky jsme tedy již schopni detekovat. To provedeme pomocí třídy BlobCounter, kterou nám poskytuje knihovna AForge.

```
BlobCounter blobCounter = new BlobCounter();  
Rectangle[] rects = null;  
  
blobCounter.FilterBlobs = true;  
blobCounter.MinHeight = 5;  
blobCounter.MinWidth = 5;  
blobCounter.ProcessImage(grayImage);  
rects = blobCounter.GetObjectsRectangles();
```

Naskenovaný obrázek obsahuje šum. S tímto musíme počítat a je tedy potřeba odfiltrovat malé objekty. U většiny scannerů lze šum vidět jako malé čtverce, které mají většinou rozměry 2x2 pixely. Zřídka kdy lze vidět šum ze scanneru v jiném tvaru či velikosti. Navíc tento šum vzniká většinou jen v případě skenování velkých bílých ploch bez textu. Bude-li dokument míň popsán, bude tedy obsahovat více šumu. Tento šum se dá jednoduše odstranit velikostním filtrem. Tento filtr je dostupný přímo v třídě BlobCounter. Upravuje se nastavením hodnot. V této ukázce je filtr zapnutý a filtruje objekty menší jak 5x5 pixelů. Bohužel tento filtr také dokáže odfiltrovat tečky za větami či čárky. Mezi další filtrování lze využít definice maximálních rozměrů či barvu jako práh detekce. Pokud jsou všechny filtry nastaveny, pak se obrázek nechá zpracovat. Zpracování probíhá po zavolání metody ProcessImage(Image). Tato metoda obrázek zpracuje a uloží veškeré nalezené údaje. Třída

BlobCounter poskytuje několik typů informací. Mezi základní informace patří počet nalezených objektů, z těch pokročilejších je to například seznam obdélníků kolem objektů, polygony kolem objekty, detekované linie či pořadí detekovaných objektů.

Nyní když je k dispozici seznam obdélníků jednotlivých objektů můžeme provést jejich vyřiznutí.

```
Bitmap image = new Bitmap(Image);

image = grayscaleFilter.Apply(image);

int i = 0;
foreach (Rectangle r in rects)
{
    Bitmap bmp = new Bitmap(r.Width, r.Height);
    bmp.SetResolution(image.HorizontalResolution,
image.VerticalResolution);
    Graphics g = Graphics.FromImage(bmp);
    g.DrawImage(image, 0, 0, r, GraphicsUnit.Pixel);
    bmp.Save(SelectedPath + "\\" + i + ".jpg");
    i++;
}
```

Zde je otevřen znovu obrázek bez úprav. Na něj je aplikován černobílý filtr. Pro zpracování neuronovou sítí není potřeba barevná informace. Poté pro každý detekovaný obdélník je vytvořen nový obrázek. Do tohoto obrázku je pomocí .NET třídy Graphics vyřiznut obdélník obsahující detekovaný objekt. Každý takový výřez je poté uložen do vybrané složky pro zpracování MATLABem.

8.2.1 Zjištěné vady algoritmu

Během realizace a testování výše uvedeného algoritmu bylo zjištěno několik vad, které je potřeba odstranit.

- detekce znaků s nízkým kontrastem
- kompatibilita se všemi formáty
- chybná detekce malých objektů
- párování písmen s diakritikou

Detekce znaků u dokumentů se špatným kontrastem je v současném systému neřešitelná. Opravou by bylo velmi citlivé nastavení prahu mezi znaky a pozadím, ale to je velmi zdoluhavý proces a navíc by se dal využít jen pro jeden dokument a pro další by se celý proces musel opakovat. Další možností je zvýšení kontrastu pomocí k tomu vytvořených algoritmů. Toto ale nebylo testováno a není to obsahem této práce.

Ve vytvořeném ukázkovém programu není vytvořena kompatibilita se všemi obrazovými formáty. Knihovna AForge poskytuje filtry a nástroje, které dokáží pracovat pouze s 8 bitovou hloubkou nebo 24 bitovou hloubkou. Většina formátů ale využívá 32 bitovou hloubku. To je způsobeno tím, že se udávají hodnoty tří barevných složek RGB a také alfa kanálu, který udává průhlednost. Takové formáty je potřeba upravit a konvertovat do použitelného formátu.

Díky nastaveným filtrům při detekci objektů nelze detekovat objekty menší jak 5x5 pixelů. Z tohoto důvodu se můžou špatně detekovat objekty menší. Typicky se jedná o desetinné čárky, tečky či různé znaky jako je např. středník a dvojtečka. Toto je potřeba dopracovat a vyřešit.

Posledním větším problémem současného algoritmu jsou čárky a háčky nad písmeny. Tyto znaky jsou detekovány samostatně. Je potřeba je detekovat, že se jedná o znak diakritiky a poté tento znak připojit k vlastnímu znaku, ke kterému patří. Toto je možno odstranit v případě formuláře. Zde se nebudou detekovat jednotlivé pozice znaků, ale budou se vybírat políčka formuláře. Znak tedy bude načten i s diakritikou.

8.3 Neuronová síť

8.3.1 Vícevrstvá neuronová síť s algoritmem Backpropagation

V programovém prostředí MATLABu byl vytvořen algoritmus realizující vícevrstvou neuronovou síť. Bylo zde možné zvolit počet skrytých vrstev, počty neuronů ve skrytých vrstvách, typy přenosových funkcí, nastavovat různé metody učení či řadu dalších parametrů.

Zde vznikly dva odlišné typy neuronové sítě. První zahrnoval jednu jedinou síť s počtem výstupních neuronů odpovídající počtu klasifikačních skupin. Druhý typ představoval množinu několika jednoduchých sítí, které měli pouze jeden výstup. Počet všech potřebných kombinací byl dán vztahem $\frac{n!}{2(n-2)!}$, kde n je počet znaků. Předložený vzor

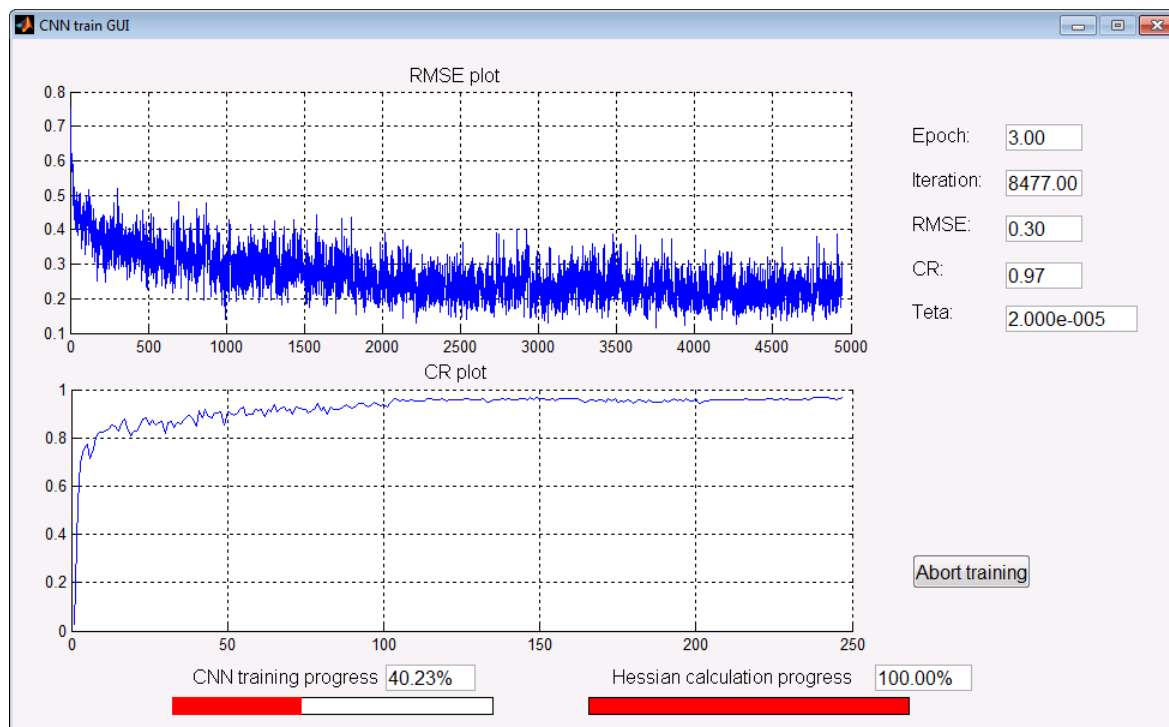
se porovnával napříč celou množinou subsítí. Tato metoda byla podstatně časově náročnější na učení, zvláště při větším počtu znaků.

Klíčovým atributem při rozpoznávání znaků pomocí neuronové sítě je to, co vlastně předložit síti na vstup (feature extraction). Hlavním úkolem této extrakce je získat množinu charakteristik, která maximalizuje úspěšnost klasifikace a která bude mít co možná nejmenší počet elementů. Během řešení tohoto problému byly postupně vyzkoušeny různé možnosti, týkaly se jak statistických (zónování, projekce, profily, průchody nulou), tak i strukturálních (histogramy, Fourierova transformace) charakteristik, byly vyzkoušeny i různé momentové charakteristiky. Začal jsem od nejjednodušších, tj. klasické použití 1 bitové reprezentace obrazu, kdy tzv. feature vektor byl reprezentován hodnotami 0 nebo 1 dle vzoru a pokračoval postupnými úpravami vstupního vzoru k výše uvedeným. Bylo provedeno množství testovacích měření za účelem najít takovou kombinaci, která by byla nejvhodnější z hlediska správného rozpoznání. Důležitou a sledovanou vlastností navržené sítě byla schopnost učit se a následně pak správně klasifikovat předkládané neznámé vzory. V tomto případě se síť učila celkem obstojně, při předložení vzorů z trénovací množiny jich bylo úspěšně klasifikováno přes 90 %, avšak po předložení vzorů neznámých (z testovací množiny) tato úspěšnost klesla pod 50 %, což bylo naprosto nevyhovující. Dalším testováním bylo zjištěno, že síť chybně klasifikuje v důsledku i malých změn vstupního vzoru např. posuv, sklon, změna měřítko či různé rotace. Pro eliminaci těchto běžně vyskytujících se jevů byl vytvořen algoritmus, který odstraňoval sklon a určitým způsobem normalizoval předkládané vzory. Úspěšnost se poté zlepšila pouze o několik jednotek procent, ale vzhledem k časové náročnosti algoritmů a faktu, že byly testovány číslice, tj. pouze 10 klasifikačních skupin, jsem tento přístup zavrhl. Mezitím byla navržena vlastní databáze vzorů obsahující velká písmena české abecedy včetně diakritiky.

8.3.2 Neocognitron

V programovém prostředí MATLABu byl vytvořen algoritmus realizující konvoluční síť. Opět bylo možné zvolit počet skrytých vrstev, počty buněk v jednotlivých vrstvách, typy přenosových funkcí a řadu dalších parametrů. Při těchto testech byla využita jak databáze MNIST, tak i vlastní databáze. Ukázka průběhu učení při použití databáze MNIST: parametr RMSE představuje střední kvadratickou chybu, parametr CR úspěšnost průběžné klasifikace, který se v průběhu učení mění (roste). Lze tedy pozorovat, jestli se síť během učení zlepšuje nebo se již dále neučí. Z časových důvodů byl tento testovací vzorek

poměrně malý, cca do 500 vzorů. Vzhledem k tomuto omezení se výsledná úspěšnost klasifikace naučené sítě mohla lišit, proto se po každém naučení sítě provedlo její testování, jehož výsledkem byla zmíněná úspěšnost.



Obr. č. 15: Příklad učení neuronové sítě (úspěšnost 97 % při rozpoznávání testovacího vzorku), při konečném testování tato síť vykazovala úspěšnost 97,4 %

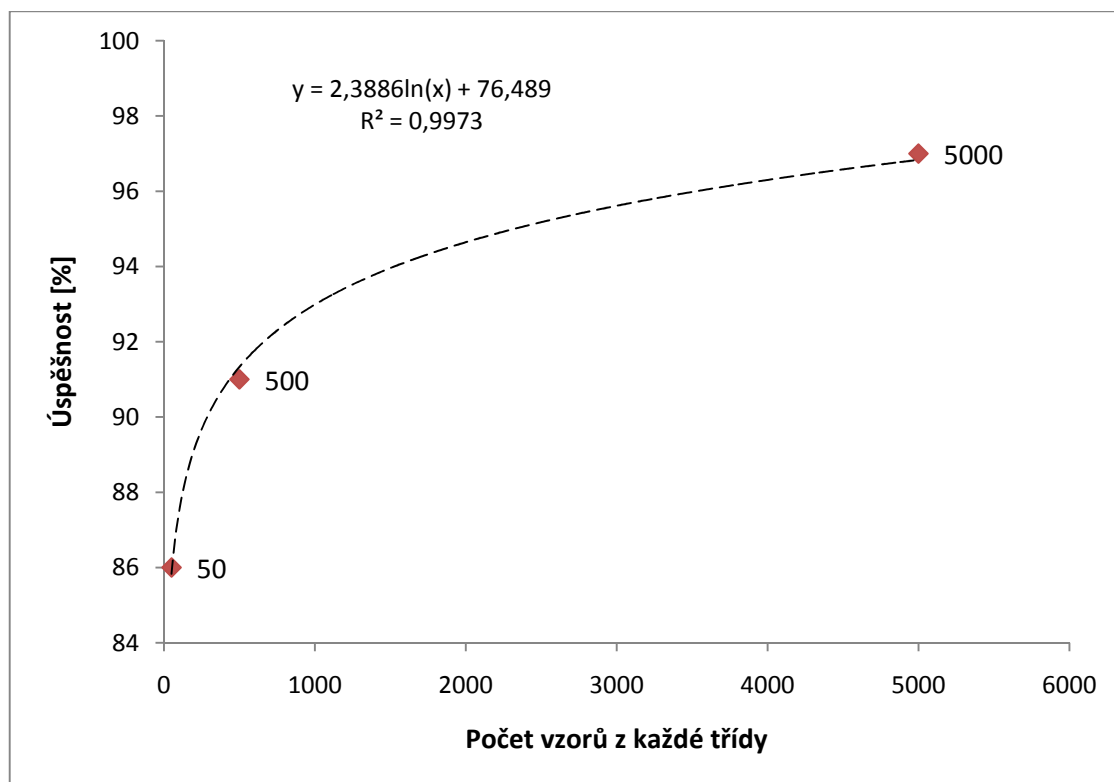
V následující tabulce jsou shrnuty výsledky učení a testování sítě pro databázi MNIST.

Tab. č. 4: Konvoluční síť (100 neuronů), 500 vzorů = 500 vzorů od jedné klasifikační skupiny

	Úspěšnost na konci učení	Úspěšnost při konečném testu	Doba učení
500 vzorů	86 %	85,6 %	0,4 hodiny
5000 vzorů	91 %	91,9 %	2 hodiny
50000 vzorů	97 %	97,4 %	10 hodin

Pokud budeme předpokládat a uvažovat logaritmickou závislost úspěšnosti na počtu vzorů, tak by se dalo vypočítat, že pro 100% úspěšnost by bylo v tomto případě potřeba přibližně 19 tisíc vzorů od každé třídy. Tento počet je ovšem z hlediska praktické realizace

nevhodný, nehledě na dobu učení potřebnou k naučení sítě. Uvedený výpočet je pouze orientační vzhledem k malému počtu provedených měření. Dalším faktorem, který má na úspěšnou klasifikaci vliv, je kvalita trénovací množiny. Pokud budeme klasifikovat např. číslice, velmi častým problémem je správné rozlišení číslic „7“ a „2“. V mnoha případech má problém s klasifikací i lidský mozek, proto nejen z tohoto důvodu je úspěšnost 100 % nereálná.



Obr. č. 16: Vztah mezi úspěšností klasifikace a počtem vzorů braných k naučení sítě

8.4 Vyhledávací algoritmus pro vyhledání slov ve slovníku

8.4.1 Požadavky

Pro použití s vyhledáváním znaků, které nemají vždy 100% jistotu správného rozeznání, musí být vytvořen algoritmus, který toto bere v úvahu. Základní požadavky na algoritmus jsou uvedeny níže

- algoritmus musí brát v úvahu pravděpodobnost znaku
- uživatel by měl mít možnost změnit rozhodovací prahy algoritmu
- vrácené slova by měli mít vypočtenou pravděpodobnost z jednotlivých znaků

Výstupem z neuronové sítě je pro každý detekovaný znak soubor znaků s pravděpodobnostmi. Pravděpodobnost znaku určuje, s jakou jistotou byl znak rozeznán. Pro vyhledávání je důležité vědět, jestli se rozeznáný znak má brát v úvahu se 100% jistotou nebo jestli se má znak zanedbat.

Pro rozlišení hranic, kdy se pravděpodobnost znaku bere jako 100% nebo naopak, kdy má být znak zanedbán, jsou brány prahy. Defaultně tyto prahy jsou nastaveny na 80% a 20%, ale každý uživatel musí mít možnost tyto prahy změnit.

Pro navracená slova se bude vypočítávat pravděpodobnost jako průměr pravděpodobností jednotlivých znaků. Bude to sloužit jako ukazatel jistoty rozeznání slova.

8.4.2 Realizace

Pro práci se slovy musela být vytvořena nová třída. V této třídě bude uloženo slovo a i další vlastnosti tohoto slova jako délka či pravděpodobnost.

```
class Slovo
{
    string _text;
    int _delka;
    double _pravdepodobnost;

    public Slovo(string text, int delka, double pravdepodobnost)
    {
        _text = text;
        _delka = delka;
        _pravdepodobnost = pravdepodobnost;
    }

    public string Text
    {
        get { return _text; }
        set { _text = value; }
    }

    public int Delka
```



```
{
    get { return _delka; }
    set { _delka = value; }
}
public double Pravdepodobnost
{
    get { return _pravdepodobnost; }
    set { _pravdepodobnost = value; }
}
}
```

Jako další byla vytvořena třída pro reprezentaci jednoho znaku. V této třídě bude opět znak a pravděpodobnost, ale je potřeba uchovávat i pozici znaku ve slově.

```
class Znak
{
    string _znak;
    int _pozice;
    double _pravdepodobnost;

    public Znak(string znak, int pozice, double pravdepodobnost)
    {
        _znak = znak;
        _pozice = pozice;
        _pravdepodobnost = pravdepodobnost;
    }

    public string Symbol
    {
        get { return _znak; }
        set { _znak = value; }
    }

    public int Pozice
    {
        get { return _pozice; }
    }
}
```

```
        set { _pozice = value; }
    }

    public double Pravdepodobnost
    {
        get { return _pravdepodobnost; }
        set { _pravdepodobnost = value; }
    }
}
```

Pro samotné vyhledávání jsou použity struktury `List<>`. Pro znak potřebujeme znát i pozici znaku ve slově z důvodu, že na jedné pozici se může nacházet více znaků. Tím je vyhledávání specifické.

Jako první stupeň filtrace slov je použito omezení na délku. Ze slovníku se vyberou pouze slova o délce detekovaného slova. Délku zjistíme dle počtu detekovaných objektů.

```
List<Slovo> nalezeno = new List<Slovo>();
nalezeno = Slovník.Where(a => a.Delka == rects.Length).ToList();
```

Zde je využito pro filtrování technologie LINQ. Tato technologie může být využita nad jakoukoliv třídou a je podporována Frameworkem .NET 3.5 a vyšším. Při této první filtraci nám v seznamu nalezeno zůstanou slova pouze o hledané délce. Další možností je například filtrování podle typu slov. Pokud víme, že na daném místě hledáme jméno, tak si zavoláme slovník, která obsahuje jména.

Když už je slovník vyfiltrován podle délky a není potřeba kontrolovat slova kratší či delší je možno přistoupit k vlastnímu filtrování dle nalezených znaků. To provádí níže uvedený cyklus.

```
for (int i = 0; i < rects.Length; i++)
{
    nalezeno = Vyhledat(i, Znaky.Where(a=>a.Pozice==i).ToList(),
nalezeno);
}
```

Ten pro každou pozici ve slově zavolá vyhledávací algoritmus a předá mu následující parametry:

- Pozice, na které se vyhledává

- List všech znaků, které se mohou na dané pozici vyskytovat
- List slov, ve kterých se vyhledává

Kompletní algoritmus vyhledání na jedné pozici jednoho znaku je uveden níže

```
private List<Slovo> Vyhledat(int pozice, List<Znak> hledat,
List<Slovo> kde)
{
    List<Slovo> temp = new List<Slovo>();

    hledat =
hledat.Where(a=>a.Pravdepodobnost>=Convert.ToDouble(tSpodniHranice
.Text)).ToList();
    if (hledat.Where(a => a.Pravdepodobnost >=
Convert.ToDouble(tHorniHranice.Text)).Count() > 0)
    {
        hledat = hledat.Where(a => a.Pravdepodobnost >=
Convert.ToDouble(tHorniHranice.Text)).Select(a => new
Znak(a.Symbol, a.Pozice, 100)).ToList();
    }

    temp =
        (from k in kde
         from h in hledat
         where k.Text.Substring(pozice, 1) == h.Symbol
         select new Slovo(k.Text, k.Delka, (k.Pravdepodobnost +
h.Pravdepodobnost)/2)).ToList();

    return temp;
}
```

Celý princip algoritmu vyhledávání je postaven na postupném filtrování. Tím se provádí porovnání na stále menším počtu vzorků a algoritmus vyhledání se zrychluje, aniž by bylo omezeno množství údajů ním nalezených.

Pro každou pozici znaku se vyhledání provede v několika krocích. Prvním krokem je odfiltrování všech znaků, které jsou pod spodním prahem. Takové znaky nejsou brány v úvahu a jsou ignorovány. stará se o to tato část kódu.

```
hledat =  
hledat.Where(a=>a.Pravdepodobnost>=Convert.ToDouble(tSpodniHranice  
.Text)).ToList();
```

Dalším krokem je kontrola znaků, které mají větší pravděpodobnost než je horní hranice. U takových znaků se předpokládá, že jsou rozpoznány správně. Pokud takový znak existuje, všechny znaky pod horní hranici jsou ignorovány a do vyhledání postupují pouze tyto znaky. O to se stará následující část kódu.

```
if (hledat.Where(a => a.Pravdepodobnost >=  
Convert.ToDouble(tHorniHranice.Text)).Count() > 0)  
{  
    hledat = hledat.Where(a => a.Pravdepodobnost >=  
Convert.ToDouble(tHorniHranice.Text)).Select(a => new  
Znak(a.Symbol, a.Pozice, 100)).ToList();  
}
```

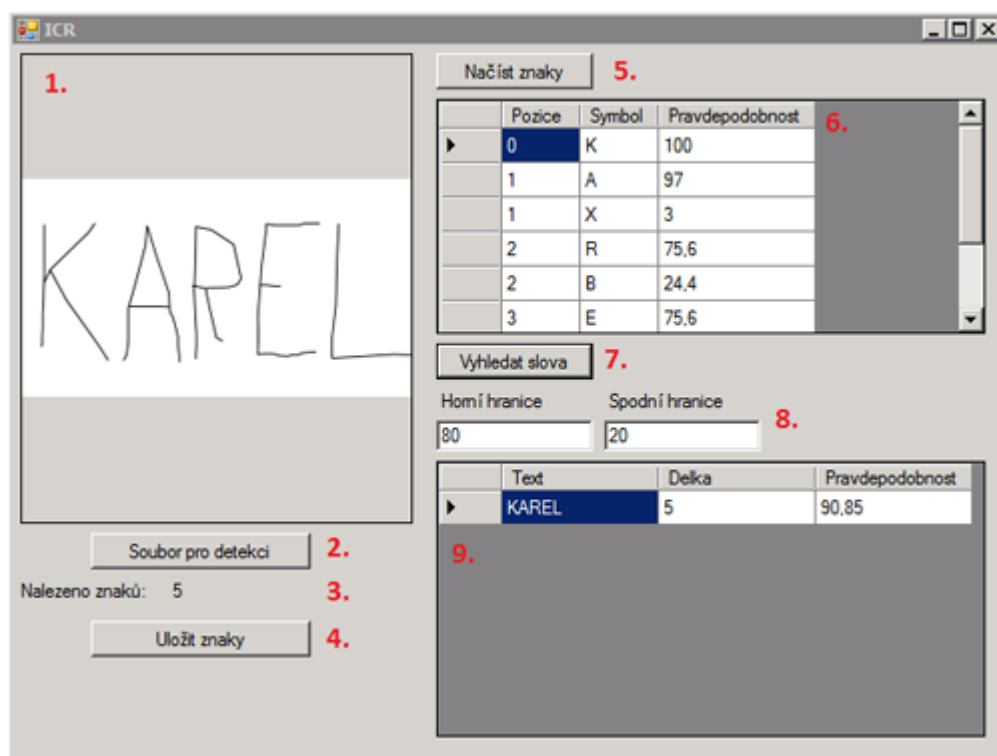
Pokud jsou znaky a slova připraveny, tak se provede filtrace. K té je využito opět technologie LINQ, ale tentokrát je poskládán dotaz jiným zápisem, který dovoluje provést složitější operace.

```
temp =  
    (from k in kde  
     from h in hledat  
     where k.Text.Substring(pozice, 1) == h.Symbol  
     select new Slovo(k.Text, k.Delka, (k.Pravdepodobnost +  
h.Pravdepodobnost)/2)).ToList();
```

Zde se vyfiltrují slova, které na hledané pozici obsahují dané znaky. Tím se získá menší seznam slov, který již obsahuje pouze možné kandidáty. Každým takovým vyfiltrováním na pozici se objem slov, ve kterých se hledá, zmenšuje a vyhledávání je rychlejší. Také použití technologie LINQ umožňuje rychlejší vyhledávání slov.

8.5 Popis GUI aplikace a ovládání

Pro aplikaci bylo vytvořeno následující GUI:



Obr. č. 17: Grafické rozhraní aplikace

1. Náhled otevřeného dokumentu
2. Výběr souboru k otevření a detekci znaků
3. Informace o počtu detekovaných objektů
4. Výběr složky pro uložení rozřezaných objektů
5. Výběr složky pro načtení rozpoznaných znaků
6. Náhled rozpoznaných znaků, zobrazuje znak, pozici a pravděpodobnost
7. Vyhledání slov ve slovníku
8. Nastavení horního a spodního prahu pravděpodobnosti
9. Výpis nalezených slov s určenou pravděpodobností

Ovládání aplikace je velmi intuitivní.

První je nutno otevřít soubor s naskenovaným textem. To lze jednoduše provést kliknutím tlačítka č.2 - *Soubor pro detekci*. Tím se otevře jednoduchá dialog pro výběr souboru na otevření. V něm lze procházet složky a standardním způsobem vyhledat soubor.

Po načtení souboru se automaticky provede detekce znaků. O počtu těchto znaků je uživatel informován pomocí prvku č.3. Zde je uveden počet detekovaných obrazců.

Následujícím krokem je uložení vyřízlých znaků. To probíhá po stisku tlačítka č.4 *Uložit znaky*. Po stisku tohoto tlačítka dojde k vytvoření dialogového okna pro výběr adresáře. V tomto okně se zvolí adresář, kde budou uloženy jednotlivé rozpoznané znaky očíslované podle pozice detekce.

Tímto krokem je dokončena první fáze, která probíhá na straně klienta. Obrázky jednotlivých znaků jsou nyní připraveny pro zpracování pomocí neuronové sítě. Jakmile proběhne zpracování, tak dalším krokem je načtení dat a výpis nalezených slov.

Načtení dat lze provést po stisku tlačítka č.5 *Načíst znaky*. Po stisku tohoto tlačítka se vytvoří dialog pro výběr složky obsahující rozpoznané znaky. Jednotlivé znaky jsou uloženy v textových souborech, kde číslo souboru označuje pozici znaku ve slově a jednotlivé znaky jsou uloženy v textových souborech s pravděpodobnostmi jejich rozpoznání.

Jakmile jsou jednotlivé znaky načteny, zobrazí se jejich výpis v části č.6. Zde je vždy uvedeno pořadí znaku ve slově, o který znak se jedná a také s jakou pravděpodobností byl znak rozeznán.

Posledním krokem je vyhledání slov ze slovníku dle odpovídajících pravděpodobností. První je nutno nastavit odpovídající horní a dolní mez. V případě, že má písmeno na pozici větší pravděpodobnost než je horní mez, je bráno v potaz pouze toto písmeno, ostatní jsou zanedbány a odpovídajícímu písmenu je přiřazena pravděpodobnost 100%. Pokud se pravděpodobnost písmena nachází v intervalu mezi horní a dolní mezí, je toto písmeno započítáno. V takovém případě může být znaků na jedné pozici i více. Berou v úvahu všechny. Pokud je pravděpodobnost nějakého písmene pod hranicí dolní meze, tak je takové písmeno zanedbáno a nebere se v potaz.

Vyhledání ve slovníku se provede pomocí stisku tlačítka č.7 *Vyhledat slova*. Po dokončeném vyhledávání se zobrazí seznam navrhaných slov jako výsledek vyhledávání. U vyhledaných slov je ještě zobrazena pravděpodobnost, s jakou je slovo rozeznáno.

9 NÁVRH FORMULÁŘE

9.1 Databáze MNIST

Jedná se o databázi ručně psaných číslic obsahující 60 000 vzorů pro trénování a 10 000 vzorů pro testování. Tyto vzory byly získány od přibližně 250 různých autorů. Databáze je vhodná pro testování učících algoritmů při rozpoznávání a klasifikaci vzorů (ručně psaných číslic). Číslice byly velikostně normalizovány a vycentrovány. Výhodou je, že vzory z databáze nevyžadují předzpracování či další úpravy. Jednotlivé vzory v databázi jsou uloženy v podobě černobílého obrazu, který je následně normalizován na velikost 20x20 pixelů se zachováním poměru stran. Po normalizaci je obraz díky anti-aliasing technice transformován do odstínů šedi. Poslední operací je vycentrování těžiště obrazu na střed výsledného obrazu o velikosti 28x28 pixelů. V programovém prostředí MATLABu byl vytvořen algoritmus pro načtení všech vzorů z uvedené databáze, jejich nezbytné úpravy a takto připravené vzory představovaly vstupní data pro trénování a testování neuronové sítě.

9.2 Návrh a vytvoření vlastní databáze

Databáze MNIST obsahovala dostatečné množství vzorů, avšak pouze pro číslice a to navíc s některými odlišnostmi od česky psaných číslic (zejména číslice 1 a 7). Pro ověření funkce navržených algoritmů pro rozpoznání vzorů byl navržen formulář, který zahrnoval číslice, znaky české abecedy (velká písmena) včetně diakritických znamének, lomítko a pomlčku.

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

A	Á	B	C	Č	D	Ď	E	É	Ě	F	G	H	I	Í	J	K	L	M	N	Ň	O	Ó	P	Q	R
A	Á	B	C	Č	D	Ď	E	É	Ě	F	G	H	I	Í	J	K	L	M	N	Ň	O	Ó	P	Q	R
A	Á	B	C	Č	D	Ď	E	É	Ě	F	G	H	I	Í	J	K	L	M	N	Ň	O	Ó	P	Q	R
A	Á	B	C	Č	D	Ď	E	É	Ě	F	G	H	I	Í	J	K	L	M	N	Ň	O	Ó	P	Q	R

Ř	S	Š	T	Ť	U	Ú	Ů	V	W	X	Y	Ý	Z	Ž	/	-
Ř	S	Š	T	Ť	U	Ú	Ů	V	W	X	Y	Ý	Z	Ž	/	-
Ř	S	Š	T	Ť	U	Ú	Ů	V	W	X	Y	Ý	Z	Ž	/	-
Ř	S	Š	T	Ť	U	Ú	Ů	V	W	X	Y	Ý	Z	Ž	/	-

Obr. č. 18: Ukázka vyplněného formuláře pro vlastní databázi

Každý formulář obsahoval celkem 53 vzorů, které daný respondent vyplnil celkem 3x. Vzniklo tedy 159 vzorů. Celkem bylo takto získáno 88 unikátních formulářů, což v konečném důsledku představuje 13 992 vzorů. Byl proto vytvořen algoritmus zpracovávající tyto naskenované formuláře, ze kterých byl extrahován každý vzor a následně vytvořena databáze všech vzorů.

Při zpracovávání dat z formuláře vznikaly chyby. Tyto chyby byly způsobeny částečnou rotací naskenovaného formuláře, nemožnost detekce jeho obrácení a chyby vzniklé skenováním.

Z těchto důvodů bylo potřeba navrhnout nový typ formuláře. Základním úkolem bylo vyřešit rotaci dokumentu. Z tohoto důvodu byly do formuláře doplněny tři objekty. Tyto objekty byly voleny tak, aby měli velkou váhu (jednotlivou plochu). Tyto objekty musí být tři, aby se dalo detekovat libovolné otočení.

Další chyby byly vytvářeny při rozřezávání podle jednotlivých políček pomocí C# a AForge, díky přerušeným liniím. To bylo způsobeno šumem vznikajícím při skenování. Z tohoto důvodu bylo potřeba zvolit silnější ohraničení buněk formuláře. V tomto případě již nehrozí přerušení linií v takové míře.

Poslední úpravou na dokumentu formuláře bylo pouze přidání políček pro číslování formuláře. Toto bylo přidáno pouze z evidenčních důvodů.

The image shows a handwritten form with three tables of characters and a small box for form number. The tables are filled with handwritten characters, likely representing a character set or a specific encoding. The first table has 10 columns (0-9) and 4 rows. The second table has 26 columns (A-O) and 4 rows. The third table has 26 columns (P-Z) and 4 rows. The small box is labeled 'Číslo formuláře NEVYPLŇOVAT!' and contains the number 11872.

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

A	Á	B	C	Č	D	Ď	E	É	Ě	F	G	H	I	Í	J	K	L	M	N	Ň	O
A	Á	B	C	Č	D	Ď	E	É	Ě	F	G	H	I	Í	J	K	L	M	N	Ň	O
A	Á	B	C	Č	D	Ď	E	É	Ě	F	G	H	I	Í	J	K	L	M	N	Ň	O
A	Á	B	C	Č	D	Ď	E	É	Ě	F	G	H	I	Í	J	K	L	M	N	Ň	O

Ó	P	Q	R	Ř	S	Š	T	Ť	U	Ú	Ů	V	W	X	Y	Ý	Z	Ž	/	-	
Ó	P	Q	R	Ř	S	Š	T	Ť	U	Ú	Ů	V	W	X	Y	Ý	Z	Ž	/	-	
Ó	P	Q	R	Ř	S	Š	T	Ť	U	Ú	Ů	V	W	X	Y	Ý	Z	Ž	/	-	
Ó	P	Q	R	Ř	S	Š	T	Ť	U	Ú	Ů	V	W	X	Y	Ý	Z	Ž	/	-	

Obr. č. 19: Ukázka vyplněného rozšířeného formuláře.

Každý formulář opět obsahuje 53 vzorů, který každý respondent vyplní 4x. Těchto formulářů bylo vyplněno a získáno 56. Tím byla databáze rozšířena o dalších 11872 vzorů.

9.3 Sběr dat

Pro vyplnění formuláře je nutné každý vytisknout a vyplnit ručně. To je značně omezující pro sběr a zpracování těchto dat. Základem tohoto vyplňování nemůže být vyplňování přes Internet, bez fyzického vytištění a vyplnění. Důvodem je to, že většina lidí nedokáže pomocí myši vytvořit znak, který by se podobal jejich písmu. To lze jen v případě, kdy je počítač vybaven tabletem a je k dispozici fyzická tužka. Sběr vyplněných formulářů je tedy omezen pouze na dvě možnosti:

- Sběr vytištěných vyplněných formulářů
- Sběr naskenovaných vyplněných formulářů

Další možností pro získávání dat je využití znaků získaných pro rozpoznávání. Tento postup by musel obsahovat interakci uživatele, který odsouhlasí, že rozpoznáný je správný a lze jej použít pro učení sítě. V takovém případě by učící základna pro jednotlivé znaky byla různá.

10 BUDOUCNOST PROJEKTU

Pro budoucí použití je potřeba dopracovat ještě několik částí:

- Úprava vstupního formuláře
- Neuronovou síť
- GUI aplikace

Úpravou vstupního formuláře není v tomto případě myšlen formulář pro učení. Je myšleno zpracování vyplněného formuláře pro rozpoznávání. V této části je potřeba vyřešit jednak podobu vstupního formuláře, podle jednotlivých požadavků na získávané data. Tak i případné korekce špatného skenování. To lze vyřešit například pomocí podobných objektů, jež byly použity pro formulář, který slouží pro získání dat pro učení neuronové sítě. Formulář nejspíše bude používat detekci podle čtverců jednotlivých políček. Toho lze dosáhnout jednoduchou úpravou algoritmu. Je také potřeba zajistit správné pořadí detekovaných znaků či jejich přímé umístění do jednotlivých typů polí. Toto jsou ovšem již úpravy, které se týkají konkrétních formulářů.

Další nutnou úpravou pro zpracování je vytvoření rozhraní mezi prostředím MATLAB a .NET. Toto rozhraní by mělo umožňovat využití všech komunikačních prvků, které nám poskytuje .NET Framework a bude komunikovat s neuronovou sítí v prostředí MATLAB.

Další drobnou úpravou je úprava vyhledávacího algoritmu. Tento algoritmus pracuje v pořádku, ale bude jej potřeba částečně modifikovat pro specifikaci vyhledávání. Pokud budeme hledat slova o známé délce, jako jsou například jména, adresy a podobně, tak algoritmus není potřeba modifikovat. Jen bude nutno specifikovat tabulky v databázi s příslušnými daty, ve kterých bude potřeba vyhledávat. Pokud, ale bude potřeba vyhledávat jednotlivé slova v dlouhém textu bez mezer, kde není možnost rozeznávat délky jednotlivých slov, bude potřeba modifikace algoritmu, aby vyhledával bez délky slova, a bude nutno vyřešit slévání slov.

Poslední modifikací programu je úprava GUI aplikace. Současné GUI bylo vytvořeno primárně pro testování jednotlivých algoritmů. Lze tak jednoduše kontrolovat výsledky každého jednotlivého kroku. Ovšem z pohledu uživatele je toto zbytečnost. Uživatel nepotřebuje vidět, kolik mu algoritmus detekoval znaků či pravděpodobnost jednotlivých znaků po rozpoznání. Je potřeba vytvořit editor, kde se zobrazí rozpoznaný text či formulář. V tomto editoru by se měla nacházet, krom samotného rozpoznaného textu,

možnost výběru slov. To je potřeba v případě, že systém vrátí několik podobných slov. Uživatel musí mít možnost z těchto slov vybrat a zvolit to správné.

Další možností rozšíření je možnost učení z rozpoznávaného textu. Až proběhne celý postup a uživatel odsouhlasí rozpoznané slovo za správné, je možnost tyto údaje odeslat zpátky na server. Tím se získá kombinace skenovaného znaku a správné hodnoty. Tyto kombinace by bylo ideální ukládat a v době nízké zátěže serveru, což je převážně v noci, pouštět učení sítě na těchto nově získaných znacích. Tím se bude neuronová síť neustále rozvíjet a získávat nové informace a také se tím bude zvyšovat její přesnost. Ta bude stoupat až do hodnoty pod 100%.

ZÁVĚR

Od naskenování dokumentu až po získání hodnot a textů je nutno provést celou řadu kroků. Mezi čtyři hlavní se řadí zpracování naskenovaného dokumentu, rozpoznání znaků, zpracování oproti slovníku a zobrazení uživateli.

V první části této práce jsou zpracovány teoretické informace potřebné splnění všech kroků. Je zde obsažena kapitola o samotném postupu zpracování obrazu, v další kapitole je zpracována teorie o použitých neuronových sítích a následně zpracovány kapitoly o použitém programovacím jazyce, technologii LINQ a vyhledávacích algoritmech. Všechny tyto informace jsou potřeba pro pochopení této problematiky.

Druhá část práce je věnována praktické stránce. V kapitolách 8.1 – 8.3 je popsána realizace algoritmu pro detekci znaků a jejich vyřezání z naskenovaného dokumentu. Zde bylo zjištěno několik zásadních překážek. První z nich je kvalita dokumentu. Ta má zásadní vliv na správnou detekci znaků. Pokud je dokument ofocen a obsahuje málo kontrastní data, je detekce téměř nemožná. Druhou je detekce českých znaků s diakritikou na čistém papíře. Jedná se o problém oddělení diakritiky od jednotlivých znaků. Řešením je používat formuláře, tj. dokumentu, kde se hodnoty vpisují do předem vytvořených polí. Všechny zjištěné vady algoritmu jsou uvedeny v kapitole 8.3 a také je uveden návrh řešení.

Kapitola 8.4 je věnována neuronovým sítím. Zde jsou otestovány dvě sítě na rozpoznávání znaků. První z nich je vícevrstvá síť s algoritmem učení Backpropagation. Tato síť má jednoduchou strukturu, ale její výsledky jsou nepřesvědčivé. Po naučení sítě a otestování na trénovací množině, bylo správně rozpoznáno přes 90% vzorků. Při použití testovací množiny, která obsahovala vzorky jiné než trénovací množina, bylo správně rozpoznáno méně než 50% vzorků. Tato síť byla shledána nevhodnou pro použití. Druhou testovanou sítí byla síť Neocognitron. Tato síť je složitější na realizaci, strukturu i časovou náročnost při učení. Při naučení sítě ze souboru 500 vzorků bylo dosaženo správného rozpoznání 86% vzorků z učící skupiny a při testování sítě bylo dosaženo rozpoznání 85,6% vzorků. Testováním bylo zjištěno, že při zvýšení počtu vzorků v učícím souboru na 50000, lze dosáhnout správného rozpoznání 97,4% vzorků z testovací množiny. Doba učení roste úměrně s počtem vzorků. Výsledky dosažené v testování jsou zobrazeny v Tab. č. 4.

Vyhledávacímu algoritmu je věnována kapitola 8.5. Zde je popsána realizace algoritmu vytvořeného pro vyhledávání ve slovníku. Algoritmus využívá technologie LINQ a pracuje na postupné filtraci slov dle rozpoznávaných znaků. Díky postupné filtraci slov ze slovníku je

vyhledávání pomocí algoritmu rychlé, protože s každým cyklem se zmenšuje soubor slov ve slovníku, nad kterým se provádí operace.

Pro uvedené algoritmy bylo vytvořeno GUI, které umožňuje sledovat správnost jednotlivých kroků. Toto GUI je zobrazeno na Obr. č. 17.

Dále byl upraven formulář pro získávání vzorků pro učení neuronové sítě. Do tohoto formuláře byly přidány prvky usnadňující zjištění správnosti polohy a upraven, aby se omezilo poškození vzniklé šumem při skenování. Také bylo navrženo průběžné doučování sítě s využitím dat pro rozpoznání.

Poslední kapitola 10. je věnována budoucnosti projektu. V ní jsou navrženy postupy a problémy k řešení. Tyto byly zjištěny při realizaci algoritmu a jeho následném testování.

ZÁVĚR V ANGLIČTINĚ

From the scanned document to obtain the values and texts must be done a number of steps. The four main steps is scanned document processing, character recognition, compared to the dictionary and display to user.

In the first part of this work are theoretical information necessary for compliance all steps. There is included a chapter on the actual imaging process, the next chapter is drawn on the theory of neural networks and subsequently processed chapter for used programming languages, LINQ technology and search algorithms. All this information is needed to understand the issue.

The second part is devoted to the practical aspect. In chapters 8.1 to 8.3 is described implementation of the algorithm to detect signs and carved from a scanned document. Here were found several major obstacles. The first of these is quality of document. It is essential to detect the correct characters. If photocopies of the document and contains low-contrast data, it is almost impossible to detect. The second is the detection Czech accented characters on plain paper. The problem is with department of accented characters. The solution is to use the form, ie paper, where the values entered in the pre-built boxes. All the deficiencies algorithm are given in Section 8.3 and the proposed solution is presented.

Chapter 8.4 is devoted to neural networks. Here are tested two networks for character recognition. The first one is a multilayer network with Backpropagation learning algorithm. This network has a simple structure, but its results are inconclusive. After learning the network and test training set were correctly detected over 90% of the samples. When using a test set, which included samples of non-training set, was recognized less than 50% of the samples. This network was found to be unsuitable for use. The second network was tested Neocognitron. This network is more difficult to implement, structure and time required for learning. The learning network of 500 samples were set to correctly identify 86% of samples from learning group and testing network to achieve recognition 85.6% of samples. Tests found that increasing the number of samples in the learning set to 50000, can be achieved by correct identification of 97.4% of samples from the test set. Learning time increases proportionally with the number of samples. The results achieved in testing are shown in Tab. No. 4

Search algorithm is a devoted to chapter 8.5. Here is described the implementation of the algorithm developed for searching the dictionary. The algorithm uses the LINQ technology

and working at the gradual filtering of words according to recognized characters. Due to the progressive filtering of the words in the dictionary using search algorithm faster, since each cycle reduce set of words in the dictionary, which performs the operation.

For these algorithms was created a GUI that allows to monitor the accuracy of each step. This GUI is illustrated in Fig. No 17

In addition, was created a modified form for obtaining samples for neural networks. In this form, the added features to facilitate detection accuracy position and adapted to limit the damage noise incurred during scanning. It was also suggested by ongoing tutoring network using data for recognition.

The last chapter 10 is devoted to the future of the project. In her practice and are designed to solve problems. These were identified during the implementation of the algorithm and its subsequent testing.

SEZNAM POUŽITÉ LITERATURY

- [1] IVAN, Zelinka. *Umělá inteligence I. : Volume 1*. Brno : Zlín: Vutium, 1998. 126 s. ISBN 80-214-1163-5.
- [2] GONZALES, Rafael C; WOODS, Richard E; EDDINS, Steven L. *Digital image processing using MATLAB*. Upper Saddle River : Pearson/Prentice Hall, 2004. 609 s. ISBN 0-13-008519-7.
- [3] NAGEL, Christian, et al. *C Sharp 2008 Programujeme profesionálně..* Brno : Computer press, 2009. 1904 s. ISBN 978-80-251-2401-7.
- [4] MAREŠ, Amadeo. *1001 tipů a triků pro C Sharp*. Brno : Computer press, 2008. 360 s. ISBN 978-80-251-2125-2.
- [5] LECUN, Yann, et al. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*. 1998, Volume: 86 Issue: 11, s. 2278-2324.
- [6] PIALORSI, Paolo; RUSSO, Marco. *Microsoft LINQ Kompletní průvodce programátora*. Brno : Computer press, 2009. 615 s. ISBN 978-80-251-2735-3.
- [7] RŮŽEK, Václav. *Algoritmy pro rozpoznání ručně psaných znaků*. Zlín, 2010. 66 s. Bakalářská práce. Univerzita Tomáše Bati ve Zlíně.
- [8] BERÁNEK, Jan. *Metody detekce a reprezentace hran v obraze*. Brno, 2007. 42 s. Bakalářská práce. Vysoké učení technické v Brně.
- [9] BENEŠ, Miroslav. *Vyhledávání v textu* [online]. 2001 [cit. 2011-04-09]. Dostupné z WWW: <<http://htmltolatex.sourceforge.net/samples/sample3.html>>.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

.NET	soubor knihoven a technologií poskytovaných společností Microsoft
apod.	a podobně
atd.	a tak dále
DPI	Dots per Inch – rozlišení obrazu, udává počet bodů na jednotku délky
LINQ	Language Integrated Query – integrovaný dotazovací jazyk pro prostředí .NET
MNIST	veřejně přístupná databáze vzorků ručně psaných tiskacích písmen a znaků, anglický zdroj bez diakritiky a menší odlišnosti
MPix	Mega pixel – rozlišení obrazu, udává počet pixelů na celé ploše obrazu
NS	Neuronová síť
OCR	Optical Character Recognition – hromadné označení technologie rozpoznávání znaků
PC	Personal Computer – osobní počítač
SQL	Structured query language – dotazovací jazyk používaný především v databázích
tzv.	tak zvaně, tak zvaný
XML	Extensible markup language – strukturovaný formát textově uložených dat

SEZNAM OBRÁZKŮ

Obr. č. 1: Schéma ideálního zpracování dat	14
Obr. č. 2: Přechodová jasová funkce	17
Obr. č. 3: Parametrický popis přímky	20
Obr. č. 4: Kontura řetězcového kódu	21
Obr. č. 5: Model umělého neuronu	22
Obr. č. 6: Přenosová funkce perceptron.....	24
Obr. č. 7: Přenosová binární funkce	25
Obr. č. 8: Přenosová funkce logistická	25
Obr. č. 9: Přenosová funkce hyperbolický tangens	26
Obr. č. 10: Síť neuronů pro algoritmus backpropagation	27
Obr. č. 11: Struktura sítě Neocognitron.....	29
Obr. č. 12: Váhy jednotlivých spojení	31
Obr. č. 13: Ukázka překrytí připojovacích oblastí C-buněk.	34
Obr. č. 14: Návrh řešení v cloudu	63
Obr. č. 15: Příklad učení neuronové sítě (úspěšnost 97 % při rozpoznávání testovacího vzorku), při konečném testování tato síť vykazovala úspěšnost 97,4 %	72
Obr. č. 16: Vztah mezi úspěšností klasifikace a počtem vzorů braných k naučení sítě	73
Obr. č. 17: Grafické rozhraní aplikace.....	79
Obr. č. 18: Ukázka vyplněného formuláře pro vlastní databázi	82
Obr. č. 19: Ukázka vyplněného rozšířeného formuláře.	83

SEZNAM TABULEK

Tab. č. 1: Rozdíl mezi PC a NS	23
Tab. č. 2: Počet ploch a buněk sítě Neocognitron	30
Tab. č. 3: Hodnotové typy C#.....	39
Tab. č. 4: Konvoluční síť (100 neuronů), 500 vzorů = 500 vzorů od jedné klasifikační skupiny	72