VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

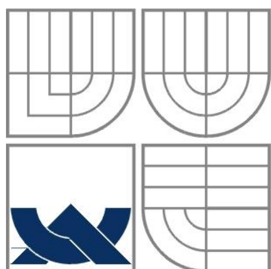
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# IMPLEMENTACE ALGORITMU SVM V FPGA

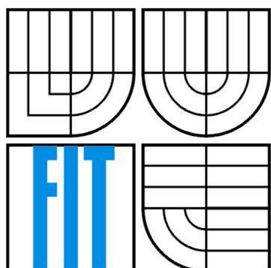
DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE Bc. JAN KRONTORÁD
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# IMPLEMENTACE ALGORITMU SVM V FPGA

IMPLEMENTATION OF SVM ALGORITHM IN FPGA

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE Bc. JAN KRONTORÁD
AUTHOR

VEDOUCÍ PRÁCE Dr. Ing. OTTO FUČÍK
SUPERVISOR

BRNO 2009

*Zadání diplomové práce/9177/2008/xkront00*

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačových systémů Akademický rok 2008/2009

# Zadání diplomové práce

Řešitel: **Krontorád Jan, Bc.**

Obor: Počítačové systémy a sítě

Téma: **Implementace algoritmu SVM v FPGA**

Kategorie: Počítačová architektura

Pokyny:

1. Prostudujte dostupnou literaturu na téma programovatelných logických obvodů FPGA a algoritmu SVM (Support Vector Machine) pro zpracování obrazu.
2. Popište daný algoritmus v jazyku VHDL a funkci ověřte simulací.
3. Algoritmus implementujte na desce DX64, která je dostupná na FIT.
4. Vyhodnoťte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Splnění prvních dvou bodů zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Fučík Otto, Dr. Ing.**, UPSY FIT VUT

Datum zadání: 22. září 2008

Datum odevzdání: 26. května 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

doc. Ing. Zdeněk Kotásek, CSc.
*vedoucí ústavu*

## Abstrakt

Tato diplomová práce se zabývá algoritmy pro trénování klasifikátoru SVM a jejich realizací v hradlovém poli FPGA. Jsou zde uvedeny základní informace o klasifikátoru, jeho trénování a uvedeny dva trénovací algoritmy. Oblíbený algoritmus SMO a algoritmus vhodný pro realizaci trénování v hardwaru.

## Abstract

This master's thesis deals with algorithms for learning SVM classifiers on hardware systems and their implementation in FPGA. There are basics about classifiers and learning. Two learning algorithms are introduced – SMO algorithm and one hardware-friendly algorithm.

## Klíčová slova

Klasifikátor, učení, SVM, FPGA

## Keywords

Classifier, learning, SVM, FPGA

## Citace

Krontorád Jan: Implementace algoritmu SVM v FPGA, diplomová práce, Brno, FIT VUT v Brně, 2009

# Implementace algoritmu SVM v FPGA

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Dr. Ing. Otto Fučíka.
Další informace mi poskytli Ing. Martin Žádník.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jan Krontorád
26.5.2009

© Jan Krontorád, 2009
*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Obsah

1 Úvod ... 3
2 Klasifikace ... 4
2.1 Klasifikátor ... 4
2.1.1 Vektor příznaků ... 5
2.1.2 Učení ... 5
2.2 Lineární binární klasifikátor ... 6
3 SVM ... 9
3.1 Lineární SVM ... 9
3.1.1 Neseparovatelná úloha ... 12
3.1.2 Klasifikace lineárního SVM ... 14
3.2 Nelineární SVM ... 15
3.2.1 Explicitní mapování do jiného prostoru ... 16
3.2.2 Jádrové funkce ... 17
3.2.3 Běžné jádrové funkce ... 18
3.2.4 Klasifikace nelineárního SVM ... 19
4 Trénování SVM ... 21
4.1 SMO algoritmus ... 21
4.1.1 Optimalizace dvou koeficientů ... 21
4.1.2 Heuristika pro výběr koeficientů ... 23
4.1.3 Výpočet posunutí ... 24
4.1.4 Shrnutí ... 24
4.2 HW-friendly trénování ... 24
4.2.1 VLSI algoritmy ... 25
4.2.2 Shrnutí ... 27
5 Implementace trénování ... 29
5.1 Aritmetika ... 29
5.1.1 Reprezentace parametrů ... 29
5.1.2 Základní výpočet ... 30
5.1.3 Kompletní architektura ... 31
5.2 Dekompozice trénování ... 31
5.2.1 Komponenta `update_b` ... 32
5.2.2 Komponenta `mm_Qa` ... 33
5.2.3 Komponenta `sum_ay` ... 34
5.2.4 Komponenta `RAM_Q_a` ... 34
5.2.5 Testování ... 35

5.2.6 Vizualizační SW ..... 36
5.2.7 Test ..... 37
6 Platforma Uni1P/DX64 ..... 39
6.1 Popis ..... 39
6.1.1 CICB ..... 40
6.1.2 Funkční jednotka `TOP_FU` ..... 41
6.1.3 Syntéza ..... 42
7 Závěr ..... 43

# 1 Úvod

Výpočetní systémy v dnešní době nejsou využívány k prostým numerickým výpočtům, ale jsou stále častěji nasazovány na problémy zpracování signálů – DSP – digital signal processing. Těmito signály mohou být záznamy zvuku, obrazu nebo jiných měřitelných veličin.

Tento diplomový projekt je zaměřen na klasifikátor SVM, který se často uplatňuje právě v aplikacích zpracování obrazu. Zpracování a rozpoznávání obrazu má široké uplatnění v různých oborech lidské činnosti. Bezpečnostní aplikace mohou pomocí kamer sledovat pohyb osob, vyhodnocovat rizikové chování a upozornit na ně obsluhu systému. V dopravě mohou podobné aplikace sledovat bezpečnost provozu v tunelech, pomáhat k plynulosti provozu ve spojení se systémy řízení křižovatek nebo pátrat po odcizených vozidlech. V automobilovém průmyslu jsou rozpracovány projekty pro předcházení nehodám, kdy systém může hlídat dodržování bezpečné vzdálenosti nebo vybočení ze směru jízdy při mikrospánku řidiče. Poslední oblastí, která zde bude uvedena jako příklad použití, je zdravotnictví. Medicínské aplikace a zobrazovací metody jako třeba radiologie, magnetická rezonance nebo počítačová tomografie.

Všechny výše zmíněné příklady použití mohou být (nebo jsou) realizovány ve formě tzv. vestavěných systémů. Takový systém a jeho části jsou pak vyrobeny a optimalizovány pro konkrétní aplikaci a typicky pak přináší vyšší poměrný výkon aplikace ku spotřebě energie, rychlejší zpracování oproti klasickému osobnímu počítači, realizujícímu tutéž činnost a často menší rozměry celého systému. S vývojem a masivním užíváním programovatelného hardwaru pak dále dochází ke snižování nákladů na vývoj a výrobu systému. Použitím tzv. IP cores lze využívat již existující komponenty systému (komunikace po sběrnici, výpočetní bloky) a urychlit tak celkový čas vývoje. Při využití programovatelného hardwaru hraje také velkou roli možnost modifikovat, vylepšovat a opravovat funkce realizovaného systému.

Diplomový projekt realizuje návrh architektury pro trénování SVM klasifikátoru a její realizaci s využitím programovatelných hradlových polí FPGA. V první části tohoto textu je čtenář seznámen se základní myšlenkou klasifikátoru. Následují kapitoly věnované seznámení s klasifikátorem SVM a s vybranými algoritmy pro trénování. Poslední části jsou věnovány návrhu systému a jeho realizaci pro fyzický vestavěný systém.

# 2 Klasifikace

Klasifikace obecně je proces, při kterém jsou vstupní objekty, typicky z reálného světa, rozdělovány do předem definovaných skupin - tříd. Support Vector Machine je jednou z mnoha metod strojového učení klasifikátoru, proto je tato kapitola určena k bližšímu seznámení s teorií klasifikátoru a klasifikace.

Člověk sám běžně provádí klasifikaci každý den a používá při tom stanovená pravidla, nebo znalosti a zkušenosti, získané během života. Třeba při česání jablek je schopen na základě získaných zkušeností rozeznat zralá a shnilá jablka, čímž je rozdělí – klasifikuje – do dvou tříd: vyhovující a nevyhovující. Dalším příkladem jednoduché klasifikace může být rybář, který po ulovení ryby a jejím změření nebo zvážení rozhodne, zda si ji může na základě stanovených pravidel ponechat, nebo zda ji musí pustit zpět do vody.

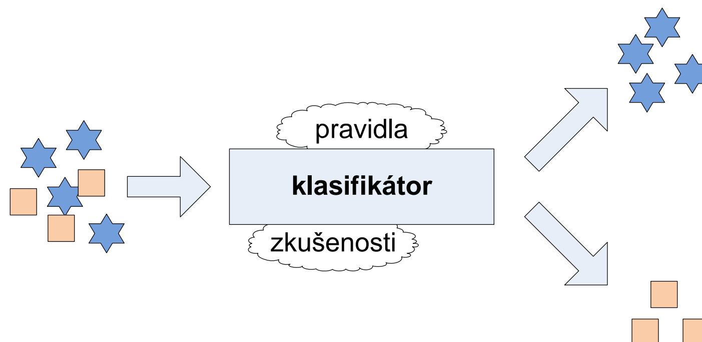
Propracované metody klasifikace používají bankovní instituce při prověřování žadatelů o půjčku. Na základě informací, získaných od klienta, jako jsou např. věk, profese, rodinný stav, měsíční příjem atd. a jejich porovnáním s obrovskou zkušeností banky se vyhodnotí, zda klient nepatří k nějaké rizikové skupině, kde by hrozilo nesplácení půjčky. Pokud si představíme, že zkušenost banky může představovat statisíce poskytnutých úvěrů v minulosti, je jasné, že rozhodnutí na základě tak velkého množství informací není schopen provést jeden člověk, ale je třeba nějaké automatizované metody. Zkušenost banky tedy může reprezentovat databáze informací o minulých klientech a automatizovaná metoda může být počítačový program, který vyhodnotí informace od nového klienta a klasifikuje jej jako potencionálně spolehlivého nebo nespolehlivého.

## 2.1 Klasifikátor

Pojmem klasifikátor se rozumí určitý stroj, který provádí klasifikaci, ať je realizován fyzicky nebo ne. Jednoduchý fyzický klasifikátor může mechanicky třídit výrobky podle velikosti a tvaru. Propracovanější fyzické klasifikátory, doplněné o snímače (různá čidla, kamery) a výpočetní systém, mohou kontrolovat kvalitu průmyslových výrobků - zda mají správnou hmotnost, zda neobsahují praskliny apod. podle požadavků majitele takového stroje. Jiné klasifikátory mohou na poštách třídit zásilky podle detekovaného poštovního směrovacího čísla a směřovat je směrem k adresátovi.

Jako příklad klasifikátoru, který nemusí být realizován jako fyzický stroj, může posloužit systém, který zpracovává nějaké abstraktní objekty, jako je třeba obraz nebo zvuk. Z databáze obrázků lesa, nebo ze vstupní kamery, která snímá les a je připojená k tomuto

systému, lze např. automaticky (strojově) vyhodnocovat, zda les nehoří. V případě, že by systém klasifikoval v některém z obrázků požár, uvědomí svoji obsluhu, která po ověření zajistí jeho likvidaci. Při zpracování zvuku pak může systém ze vstupní nahrávky detekovat řečníka, jazyk, kterým hovoří, nebo pořizovat textový záznam hovoru.




*Obrázek 1: klasifikátor*

### 2.1.1 Vektor příznaků

Jak fyzické tak i abstraktní objekty, které chceme třídit, musíme nějakým vhodným způsobem reprezentovat. Zvolení vhodného způsobu pro danou aplikaci nemusí být triviální a přímo ovlivňuje kvalitu funkce klasifikátoru [4]. Jelikož je klasifikátor reprezentován výpočetním systémem, jedinou možnou reprezentací je sada čísel. Těmto číslům se říká vektor příznaků – *feature vector*.

Vektor příznaků tedy obsahuje údaje z reálného světa, které můžeme změřit nějakými čidly, získat zpracováním signálů zvuku/obrazu, nebo získat od určité osoby vyplněním dotazníku. Změřené hodnoty můžeme přímo reprezentovat ve vektoru příznaků jako číslo. Při zpracování obrazu a zvuku se provádí filtrace, která vstupní signál zbaví šumu a následně se z tohoto signálu provede extrakce příznaků – *feature extraction*. Extrakce příznaků u signálu by tedy měla snížit velikost reprezentace objektu při zachování podstatných vlastností, které budou použity k trénování a následně i ke klasifikaci. Ze vstupního barevného obrázku můžeme zjistit histogram barevných odstínů, můžeme jej převést na obrázek v odstínech šedi a následně aplikovat nějaký obrazový filtr pro nalezení hran, rohů, geometrických objektů, případně provést určitou transformaci. Po dokončení zpracování tak získáme sadu příznaků, která reprezentuje objekt zachycený na obrázku.

### 2.1.2 Učení

Pro reálné úlohy, které nejsou založeny pouze na použití rozhodovacích pravidel veličin, které můžeme přesně definovat a popsat, je třeba, aby klasifikátor získal zkušenosti, stejně jako člověk. Pokud by byl automatický systém použit např. ke kontrole jakosti jablek na

běžícím pásu v továrně a jejich následném třídění, je třeba, aby klasifikátor byl naučen (natrénován) k rozpoznávání vyhovujících a nevyhovujících jablek.

Pro trénování se často používá metoda *Supervised learning* - učení s učitelem. V tomto případě máme k dispozici velké množství objektů, u kterých známe jejich zařazení do třídy. Tyto objekty převedeme na vektory příznaků a k samotnému trénování použijeme množinu, která obsahuje dvojice $(\vec{x}, Y)$, kde $\vec{x} = (x_{0,} x_1, ..., x_n)$ je vektor příznaků a $Y$ určuje třídu, do které patří objekt s tímto vektorem. Potom pomocí vybraného algoritmu, který je závislý na konkrétním typu použitého klasifikátoru, provedeme trénování, spočívající v generování nějakých rozhodovacích struktur nebo nastavení parametrů klasifikátoru.

Trénování může být ukončeno různými podmínkami. Např. překročením maximálního stanoveného počtu trénovacích iterací, dosažením minimální stanovené chyby klasifikace na trénovacích datech, malou změnou parametrů klasifikátoru mezi dvěma iteracemi, nebo jinak. Po dokončení trénování máme tedy klasifikátor, který je schopen s určitou pravděpodobností řešit funkci $f(\vec{x}) = Y$, kde $\vec{x}$ je vektor příznaků neznámého vstupního objektu a $Y$ je odezva klasifikátoru – třída klasifikovaného objektu.

## 2.2 Lineární binární klasifikátor

Nyní se budeme zabývat lineárním binárním klasifikátorem a lineárně separovatelnými vektory příznaků - daty. Pojem „data“ bude v následujícím textu využit k označení vektoru příznaků. Binární klasifikátor je schopen rozlišit mezi dvěma třídami a lineárně separovatelná data jsou taková, která lze v prostoru příznaků oddělit obecně hyperplochou. Ve dvourozměrném prostoru (vektor příznaků se skládá ze dvou čísel) to znamená, že lze mezi vektory různých tříd (body v ploše) nakreslit přímku, v trojrozměrném prostoru lze mezi vektory dvou tříd (body v 3D prostoru) vložit plochu a obecně v $N$ rozměrném prostoru lze mezi vektory příznaků vložit hyperplochu rozměru $N-1$.

Pokud tedy máme vektory příznaků $\vec{p_i}$, kde $\vec{p_i} \in \mathbb{R}^N, \forall i$ a vektory obou tříd jsou lineárně oddělitelné, lze je oddělit objektem (hyperplochou) dimenze $N-1$, která vyhovuje rovnici $\vec{w} \cdot \vec{x} - b = 0$, kde $\vec{w} \in \mathbb{R}^N$, $\vec{w}$ je normálový vektor oddělující hyperplochy a $b$ je posunutí (v literatuře *bias* nebo *threshold*) oddělující hyperplochy od počátku souřadného systému. Pro příklad trojrozměrného prostoru ($N = 3$) je oddělující hyperplochou dvourozměrná hyperplocha – rovina s rovnicí $\vec{w} \cdot \vec{x} - b = 0$, kde $\vec{w} \in \mathbb{R}^3$.

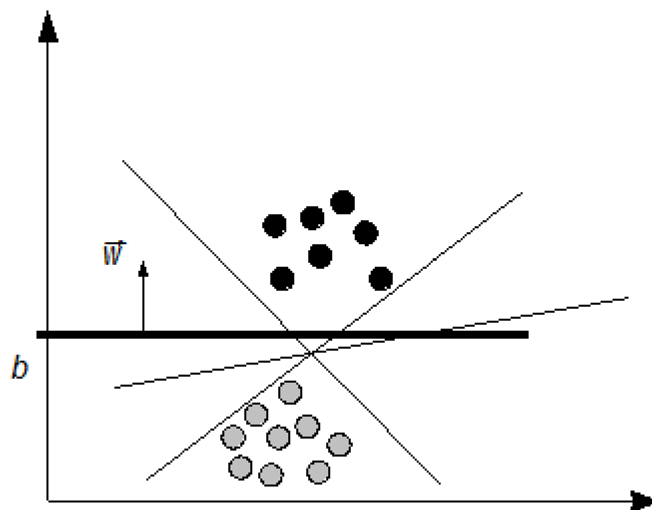
Binární klasifikátor je nejjednodušší klasifikátor, který rozděluje objekty do dvou tříd. Tyto třídy se často označují jako pozitivní třída a negativní třída, nebo jako +1 a -1. Pomocí jednoduchých binárních klasifikátorů lze pak vytvořit klasifikátor více třídový [4]. Obecně je tedy klasifikátor funkce

$$f:\vec{x} \subseteq \mathbb{R}^N \rightarrow \mathbb{R} \tag{1}$$

kde $\vec{x}$ je vektor vlastností klasifikovaného objektu a je mu přiřazena pozitivní hodnota, pokud $f\left(\vec{x}\right) \geq 0$, jinak hodnota negativní. Binární klasifikátor lze také zapsat následujícím způsobem:

$$\operatorname{sign}\left(f\left(\vec{x}\right)\right) = \begin{cases} +1 & \text{pro} \quad f\left(\vec{x}\right) \geq 0 \\ -1 & \text{jinak} \end{cases} \tag{2}$$

Geometricky je možné klasifikátor zobrazit jako na následujícím obrázku.




*Obrázek 2: Lineární klasifikátor*

Na obrázku 2 jsou zobrazena trénovací data – vektory příznaků dvou tříd. Běžnou metodou, jak stanovit oddělující hyperplochu (v tomto případě přímku) je například perceptronový algoritmus. Jedná se o iterační proces hledání vektoru $\vec{w}$ a posunutí $b$, jehož řešením mohou být různé přímky [3], oddělující trénovací vektory.

Všechny hyperplochy na předchozím obrázku dobře klasifikují trénovací množinu dat. V praxi ale není většinou realizovatelné trénovat klasifikátor na všech možných datech, proto je dobré vybrat optimální hyperplochu tak, aby drobný posun klasifikovaných objektů

mimo hranice natrénovaných oblastí nezpůsoboval chybu v klasifikaci. Tím zajistíme poměrně významnou vlastnost klasifikátoru – generalizaci výkonnosti [3].

Na základě předchozího tvrzení lze usoudit, že takový klasifikátor by měl mít co největší hranici mezi klasifikací rozdílných tříd. Ideální možností je v tomto případě zvolit hyperplochu vyznačenou tučnou čarou. To však nejsou jednoduché metody strojového učení (perceptronový algoritmus) schopny zajistit [3].

# 3 SVM

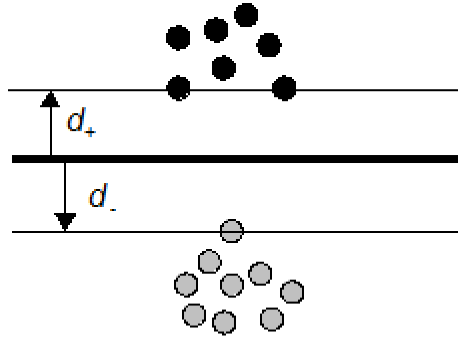
Support Vector Machine je metoda strojového učení určená pro klasifikaci a regresi. Počátky této metody sahají do roku 1963, kdy Vladimir Vapnik prezentoval algoritmus pro nalezení optimální dělící roviny lineárního binárního klasifikátoru. V roce 1992 Vapnik s kolegy představil metodu, jak implicitně převést nelineární klasifikační problém na lineární problém v jiném prostoru (*feature space*) pomocí jádrových funkcí (*kernel trick*). V tomto prostoru, který mívá podstatně vyšší dimenzi než původní, může existovat lineární dělící rovina mezi vzorky trénovacích dat dvou tříd a lze tedy použít algoritmu na nalezení optimální dělící roviny lineárního problému. V 90. letech minulého století došlo k rozvoji metody SVM a byla použita k různým účelům zpracování signálů jako např. rozpoznávání ručně psaných číslic poštovních směrovacích čísel, detekce obličeje v obrazu, nebo detekce mluvčího. Tyto příklady použití jsou převzaty z [1].

V této kapitole bude představena základní myšlenka SVM, kdy se při trénování hledá optimální dělící rovina mezi dvěma třídami. Optimální proto, že se maximalizuje vzdálenost dělící roviny k nejbližším trénovacím vzorkům, čímž se zlepšuje kvalita výsledného natrénovaného klasifikátoru. Následně bude ukázáno použití jádrových funkcí k transparentnímu převedení nelineárních klasifikačních problémů na lineární.

## 3.1 Lineární SVM

Nejjednodušším případem SVM je lineární klasifikátor, který je trénovaný na lineárně separovatelných datech.

Nechť máme $N$ trénovacích vzorků $\{\vec{x_i}, y_i\}$, $i = 1,...N$, $y_i \in \{-1,1\}$, $\vec{x_i} \in \mathbb{R}^d$ dat. Trénovací data jsou ve formě dvojice vektoru příznaků $\vec{x_i}$ a klasifikační třídy $y_i$ daného vektoru. Předpokládejme, že máme hyperplochu, která odděluje trénovací data obou tříd. Pak body $\vec{x}$, které leží na oddělující hyperploše splňují rovnici $\vec{w} \cdot \vec{x} + b = 0$, kde $\vec{w}$ je normálový vektor dané hyperplochy, $|b| / \|\vec{w}\|$ je vzdálenost hyperplochy od počátku souřadného systému a $\|\vec{w}\|$ je euklidovská norma vektoru. Pokud si stanovíme $d_+$ a $d_-$ jako vzdálenost oddělující hyperplochy k nejbližšímu bodu pozitivní, respektive negativní třídy, pak je výraz $d_+ + d_-$ roven šířce hranice, mezi oběma třídami. Pro lineárně separovatelný případ hledá SVM algoritmus takovou hyperplochu, která má tuto šířku hranice maximální.




*Obrázek 3:vzdálenost hyperplochy od nejbližších trénovacích dat*

Pro další výklad je důležitá vlastnost, že můžeme měnit velikost normálového vektoru nalezené hyperplochy $\|\vec{w}\|$ a v závislosti na něm hodnotu posunutí $b$ tak, že nedojde k ovlivnění funkce klasifikátoru. Změna se provede tak, aby nejbližší body pozitivní třídy a nejbližší body negativní třídy ležely na hyperplochách daných rovnicemi (3) a (4).

$$\vec{w} \cdot \vec{x}_{positive} = +1 \tag{3}$$

$$\vec{w} \cdot \vec{x}_{negative} = -1 \tag{4}$$

Podle obrázku 3 budou trénovacímu vektoru příznaků $\vec{x}$ pozitivní třídy odpovídat dva tmavé body a pro negativní třídu jeden světlý bod. Velikost okraje mezi dvěma třídami pak tedy bude

$$\frac{1}{\|\vec{w}\|} + \frac{1}{\|\vec{w}\|} = \frac{2}{\|\vec{w}\|} \tag{5}$$

a pro všechny trénovací vektory $\vec{x}$ bude platit

$$\begin{aligned} \vec{w} \cdot \vec{x} + b &\geq +1, \quad \forall \vec{x} \in \text{pozitivní třída} \\ \vec{w} \cdot \vec{x} + b &\leq -1, \quad \forall \vec{x} \in \text{negativní třída} \end{aligned} \tag{6}$$

Rovnice (6) lze sloučit do jednoho vztahu

$$y_i \left( \vec{w} \cdot \vec{x}_i + b \right) \geq +1 \tag{7}$$

kde $y_i \in \{-1,+1\}$ je klasifikační třída, přidělená trénovacímu vektoru $\vec{x_i}$. Z rovnice (5) plyne, že šířka hranice mezi třídami bude tím větší, čím menší bude velikost normálového vektoru hyperplochy $\|\vec{w}\|$. Je tedy nutno vyřešit následující problém:

$$\begin{aligned} &\text{minimalizovat} \quad && \frac{1}{2}\|\vec{w}\|^2 \\ &\text{vzhledem k} \quad && y_i\left(\vec{w}\cdot\vec{x_i}+b\right)\geq +1 \end{aligned} \tag{8}$$

což je konvexní kvadratický optimalizační problém (*QP problem*) s lineárními podmínkami [1] a lze jej řešit s použitím metod *Lagrangeových násobitelů* a principu *duality*. V rovnici (8) by bylo možné minimalizovat pouze normu $\|\vec{w}\|$. Nicméně výraz z (8) má stejné minimum a pro další výpočet zjednodušuje odvození. Výchozím lagrangianem je tedy

$$L_P\left(\vec{w},b,\alpha\right)=\frac{1}{2}\|\vec{w}\|^2-\sum_i \alpha_i\left(y_i\left(\vec{w}\cdot\vec{x_i}+b\right)-1\right) \tag{9}$$

kde $\alpha_i \geq 0$ jsou lagrangeovi násobitele, $\vec{w}$ je normálový vektor hledané hyperplochy, $b$ je hodnota posunutí hyperplochy, $\vec{x_i}$ jsou trénovací vektory a $y_i$ jsou třídy přiřazené daným trénovacím vektorům. Řešením tohoto lagrangianu jsou rovnice (10) a (11) s podmínkami (12).

$$\vec{w}=\sum_i \alpha_i y_i \vec{x_i} \tag{10}$$

$$0=\sum_i y_i\alpha_i \tag{11}$$

$$\alpha_i\left(y_i\left(\vec{w}\cdot\vec{x_i}+b\right)-1\right)=0, \quad \forall i \tag{12}$$

Když dále rovnici (10) dosadíme do výchozího lagrangianu $L_P$ (9), dostaneme duální lagrangian

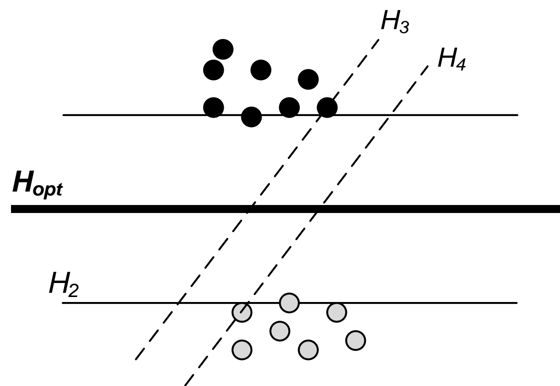
$$L_D\left(\alpha\right)=\sum_i \alpha_i-\frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \vec{x_i}\cdot\vec{x_j} \tag{13}$$

který již není závislý na hledaných parametrech $\vec{w}$ a $b$ a budou se hledat takové násobitele $\alpha$, které budou splňovat podmínky:

$$\alpha_i \geq 0 \quad (14)$$

$$\sum_i \alpha_i y_i = 0 \quad (15)$$

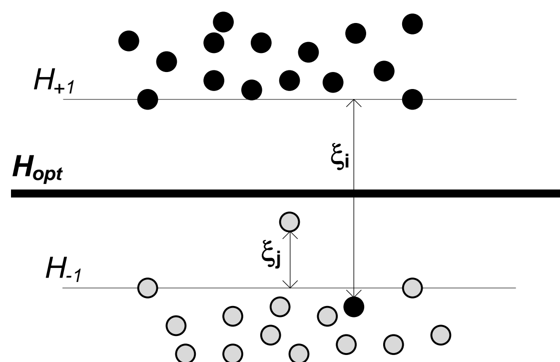
Po nalezení lagrangeových násobitelů $\alpha$ pak rovnice (10) stanovuje hodnotu normálového vektoru $\vec{w}$ optimální hyperplochy a z rovnice (12) lze vyjádřit hodnotu posunutí $b$. Lagrangeovy koeficienty pro body, ležící na hraničních hyperplochách jsou větší než 0. Tyto body jsou současně nazývány jako *Support Vectors* – podpůrné vektory a odezva klasifikátoru (hodnota reálné klasifikační funkce) na tyto body je +1 resp. -1. Pro ostatní trénovací body jsou Lagrangeovy koeficienty nulové. Na obrázku 4 je optimální nalezená hyperplocha $H_{opt}$ a podpůrné vektory leží na hraničních rovnoběžných hyperplochách $H_1$.a $H_2$.




*Obrázek 4: rozdílné velikosti šířky okrajů*

### 3.1.1 Neseparovatelná úloha

Hledání optimální oddělující hyperplochy, způsobem popsaným v předchozí části je většinou problematické. Pro úlohy, které nejsou ze své podstaty lineární, existuje způsob transformace na lineární úlohu. Tento způsob je popsán níže. Pokud je úloha lineárně řešitelná, nebo je transformovaná na lineární, tak stále může existovat v trénovacích datech jistý šum, který několik málo trénovacích vzorků zařadí do opačné třídy. Hledání maximální možné dělící šířky mezi trénovacími daty funguje pouze v případě, že jsou striktně lineárně oddělitelná. Pokud tedy trénovací data vypadají jako na následujícím obrázku 5, nejsme schopni pomocí maximalizace šířky oddělovacího pásma najít řešení [1], protože trénování maximalizací dělícího pásma nepřipouští chybu v trénovací množině.




*Obrázek 5: lineární úloha s volným okrajem*

Obrázek 5 ukazuje úlohu, která může být lineárně řešitelná, avšak s jistou tolerancí u některých trénovacích vzorků. Každému trénovacímu vzorku je přiřazena hodnota $\xi$, která udává, jak daleko se může daný trénovací vzorek dostat mimo hranice vymezené hyperplochami $H_{+1}$ a $H_{-1}$. Jedná se o *soft margin* úlohu – úlohu s volným okrajem. Trénovací data pak musejí odpovídat následujícím rovnicím a podmínce:

$$\vec{x_i} \cdot \vec{w} + b \geq +1 - \xi_i \quad \text{pro } y_i = +1 \tag{16}$$

$$\vec{x_i} \cdot \vec{w} + b \leq -1 + \xi_i \quad \text{pro } y_i = -1 \tag{17}$$

$$\xi_i \geq 0 \forall i \tag{18}$$

Rovnice (16), (17) a (18) lze shrnout do následující rovnice:

$$y_i((\vec{w_i} \cdot \vec{x_i}) + 1) - 1 + \xi_i \geq 0 \forall i \tag{19}$$

Hodnoty $\xi_i$ mohou být interpretovány jako měřítko, jak moc se může daný bod dostat mimo povolená pásma nad hyperplochou $H_{+1}$ a pod hyperplochou $H_{-1}$. Různé možnosti shrnuje následující tabulka:

| | |
|---|---|
| $\xi_i = 0$ | Trénovací vzorek $\vec{x_i}$ je korektně umístěn; je správně klasifikován a leží mimo oddělovací zakázané pásmo. Tomuto případu odpovídá většina trénovacích vzorků z obrázku 5. |
| $0 \leq \xi_i < 1$ | Trénovací vzorek $\vec{x_i}$ je sice korektně klasifikován, ale nachází se uvnitř oddělovacího pásma. Tomuto případu odpovídá jeden světlý trénovací vzorek z obrázku 5, který se nachází mezi optimální |

|  |  |
|---|---|
|  | hyperplochou $H_{opt}$ a hraniční $H_{-1}$. |
| $\xi_i \geq 1$ | Trénovací vzorek $\vec{x_i}$ není správně klasifikován. Tomuto případu odpovídá jeden tmavý bod z obrázku 5, který je umístěn ve špatné klasifikační polorovině. |

*Tabulka 1*

Jelikož $\xi_i$ přesahuje hodnotu 1 jen v případě, že je daný trénovací vzorek špatně klasifikován, pak $\sum_i \xi_i$ všech trénovacích vzorků udává horní hranici počtu špatně klasifikovaných trénovacích vzorků po ukončení trénování. Pokud tedy bude předešlá suma rovna hodnotě 10, pak je možné připustit, že po ukončení trénování bude existovat až 10 spatně klasifikovaných trénovacích vzorků. Nebudeme tedy minimalizovat $\frac{1}{2}\left\|\vec{w}\right\|^2$ z rovnice (8), ale $\frac{1}{2}\left\|\vec{w}\right\|^2 + C\left(\sum_i \xi_i\right)$, kde $C$ je parametr stanovený uživatelem před trénováním. Z minimalizační úlohy dané rovnicí (13) a podmínkami (14) a (15) se pak stane úloha minimalizovat lagrangian

$$L_D = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \vec{x_i} \cdot \vec{x_j} \qquad (20)$$

za podmínek

$$0 \leq \alpha_i \leq C \qquad (21)$$

$$\sum_i \alpha_i y_i = 0 \qquad (22)$$

Jedinou změnou je tedy podmínka (21), kde jsou hodnoty lagrangeových koeficientů shora omezeny uživatelem stanovenou konstantou $C$. Čím vyšší bude konstanta $C$, tím větší bude při trénování penalizace, za špatně klasifikovaný trénovací vzorek dat a proces trénování se bude více snažit tento bod správně klasifikovat.

### 3.1.2 Klasifikace lineárního SVM

Pokud máme zkonstruovaný a natrénovaný lineární klasifikátor, máme k dispozici hodnotu vektoru $\vec{w}$ a posunutí $b$ optimální oddělující hyperplochy. Tyto hodnoty byly stanoveny výše popsaným postupem po výpočtu lagrangeových koeficientů $\alpha$ daného minimalizačního problému.

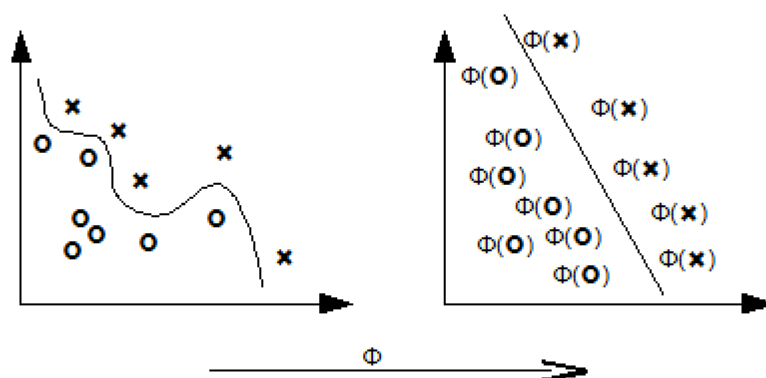
Ke klasifikaci neznámého bodu $\vec{x}$, reprezentovaného vektorem příznaků, tedy stačí spočítat následující funkci:

$$f\left(\vec{x}\right) = \vec{w} \cdot \vec{x} + b \tag{23}$$

Pokud je výsledkem funkce číslo větší nebo rovno 0, pak se jedná o vzorek z pozitivní třídy, pokud je výsledkem funkce číslo menší než 0, pak se jedná o vzorek z negativní třídy. Připomeňme, že pokud je výsledkem funkce +1 nebo -1, pak se jedná o bod, ležící na jedné z hraničních hyperploch $H_1$ nebo $H_2$. Výpočetně se tedy při klasifikaci jedná o operace typu MAC (násob a sčítej – skalární součin vektorů). Normálový vektor optimální dělící hyperplochy $\vec{w}$ je skalárně vynásoben vektorem příznaků klasifikovaného objektu $\vec{x}$. K tomto výsledku je přičtena hodnota posunutí $b$ hyperplochy a kladná nebo záporná hodnota určuje, zda je klasifikovaný vzorek z pozitivní nebo negativní třídy.

## 3.2 Nelineární SVM

Lineární klasifikátory mají velké omezení použitelnosti právě v aplikovatelnosti pouze na lineárně separovatelné úlohy. Reálné úlohy jsou často velmi komplexní a hledání vhodných vektorů příznaků tak, aby byly klasifikované třídy lineárně oddělitelné, je velice obtížné. Jinými slovy, v reálných aplikacích bývá rozhodovací funkce nelineární [3]. Obecnou myšlenkou k řešení nelineárních úloh je převést úlohu ze vstupního prostoru do cílového prostoru (typicky podstatně vyšší dimenze) pomocí nelineární transformace (mapování) tak, aby v cílovém prostoru byla úloha lineárně řešitelná. Na obrázku 6 je znázorněno takové mapování $\Phi$ mezi dvěma dvourozměrnými prostory.

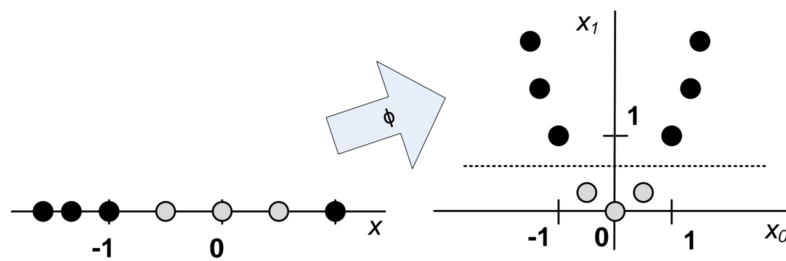



*Obrázek 6: mapování z jednoho prostoru do jiného*

Pokud nalezneme tedy vhodné mapování $\Phi$, můžeme tak použít trénovací vektory ve formě $\Phi\left(\vec{x_i}\right)$ a převést takovou úlohu na lineární, kterou jsme schopni řešit. Cílový prostor, do kterého je úloha převedena se v literatuře označuje jako *feature space* – prostor příznaků.

### 3.2.1 Explicitní mapování do jiného prostoru

Pro ilustraci reálného mapování do jiného prostoru je zde uvedena situace na obrázku 7. Vlevo jsou zobrazena trénovací data z jednorozměrného prostoru – tedy dvojice $(x, y)$, kde $x \in \mathbb{R}$ je jednorozměrný vektor příznaků a $y \in \{-1, +1\}$ reprezentuje třídu černých a světlých bodů.




*Obrázek 7: transformace z 1D do 2D prostoru*

V pravé části obrázku jsou zobrazeny totožné trénovací body, na které byla aplikovaná transformace. Po transformaci vektorů dostaneme dvojice $((x_0, x_1), y)$, kde $(x_0, x_1)$ je pozice trénovacího vektoru v novém prostoru a označení třídy $y$ zůstává nezměněno. Transformace, která je schopna převést tuto úlohu na lineární může být například tato:

$$\begin{aligned} &\Phi : L \to H \\ &L \approx \mathbb{R}, H \approx \mathbb{R}^2 \\ &x \in L \to (x, x^2) \in H \end{aligned} \tag{24}$$

$L$ je označení původního prostoru úlohy a $H$ je označení pro vícerozměrný prostor příznaků. Obvykle platí, že $\dim(H) \gg \dim(L)$. Způsob, jak převedeme vektory příznaků z původního prostoru do prostoru příznaků je uveden na třetím řádku v rovnici (24).

V praxi je obtížné hledat mapování $\Phi$, protože dimenze prostoru příznaků $H$, ve kterém je možno úlohu lineárně řešit, bývá vysoká. Dále dochází k zvyšování výpočetní náročnosti úměrně s poměrem velikostí původního prostoru a prostoru příznaků. Proto se využívá vlastností jádrových funkcí, které při srovnatelné výpočetní složitosti trénování

implicitně převedou problém do vysokodimenzionálního prostoru, který je často nekonečný [3].

### 3.2.2 Jádrové funkce

Při hledání optimální hyperplochy oddělující dvě lineárně separovatelné třídy je využito minimalizace pomocí Lagrangeovy metody. Tato metoda je závislá na trénovacích datech pouze tak, že se trénovací data vyskytují ve formě skalárního součinu mezi dvěma obecně různými trénovacími vektory. V kapitole 3.1 je odvozen Lagrangian (20), a je hledáno jeho řešení v závislosti na omezujících podmínkách (21) a (22).

Lagrangian z rovnice (20) je závislý na skalárních součinech vektorů příznaků trénovacích dat $\vec{x_i} \cdot \vec{x_j}$. Nyní předpokládejme mapování z počátečního prostoru, který má dimenzi rovnu velikosti vektoru příznaků, do jiného euklidovského prostoru vyšší dimenze: $\Phi : \mathbb{R}^d \rightarrow H$. Potom bude trénovací algoritmus závislý na skalárním součinu transformovaných vektorů příznaků - $\Phi\left(\vec{x_i}\right) \cdot \Phi\left(\vec{x_j}\right)$. Pokud ale budeme mít jádrovou funkci (*kernel function*) $K$ takovou, že $K\left(\vec{x_i}, \vec{x_j}\right) = \Phi\left(\vec{x_i}\right) \cdot \Phi\left(\vec{x_j}\right)$, potom budeme moci v trénovacím algoritmu použít tuto funkci, namísto skalárního součinu transformovaných vektorů příznaků.

$$L_D(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \Phi\left(\vec{x_i}\right) \cdot \Phi\left(\vec{x_j}\right) \tag{25}$$

$$L_D(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K\left(\vec{x_i}, \vec{x_j}\right) \tag{26}$$

Často je cílový euklidovský prostor vysoké dimenze (tisíce), nebo nekonečné [1] a nebylo by pak výhodné pracovat s takto dlouhými vektory příznaků z vysokorozměrného prostoru. Dále by nastal problém, jak vybrat vhodnou transformaci $\Phi$ tak, aby trénovací vektory byly v cílovém prostoru lineárně oddělitelné. Pokud tedy explicitní transformaci $\Phi$ a skalární násobení transformovaných vektorů nahradíme jádrovou funkcí v trénovacím algoritmu, budeme mít k dispozici SVM klasifikátor pracující ve vysokodimenzionálním prostoru, který jsme schopni natrénovat ve stejném čase, jako bychom měli lineární klasifikátor v původním prostoru. Rovnice (26) ukazuje konečný lagrangian, který je třeba vyřešit.

### 3.2.3 Běžné jádrové funkce

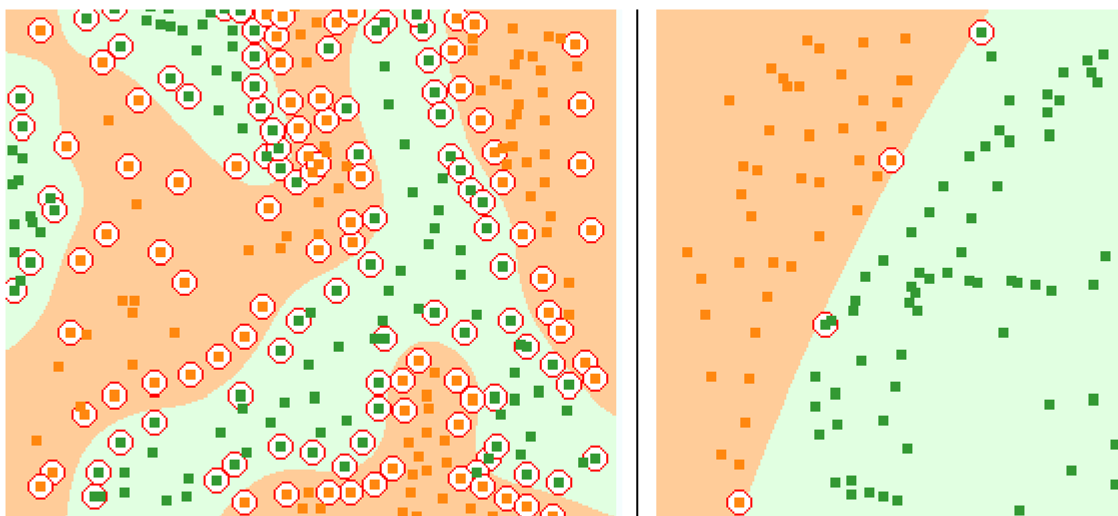
Běžným typem obecně používaných jádrových funkcí pro aplikace "*pattern recognition*" jsou následující [1].

$$K\left(\vec{x}_i, \vec{x}_j\right) = \left(\vec{x}_i \cdot \vec{x}_j + c\right)^p \tag{27}$$

$$K\left(\vec{x}_i, \vec{x}_j\right) = \exp\left(\frac{-\left\|\vec{x}_i - \vec{x}_j\right\|^2}{2\sigma^2}\right) \tag{28}$$

$$K\left(\vec{x}_i, \vec{x}_j\right) = \tanh\left(\kappa \vec{x}_i \cdot \vec{x}_j - \delta\right) \tag{29}$$

Polynomiální jádrová funkce (27) vytvoří polynomiální klasifikátor stupně $p$. Gaussovská radiální bázová jádrová funkce RBF (28) vytvoří gaussovský klasifikátor, který má tolik center, kolik je podpůrných vektorů po ukončení trénování. Podpůrné vektory jsou tedy centry tohoto klasifikátoru a lagrangeovi koeficienty $\alpha_i$ jsou váhy těchto center. Rovnice (29) pak reprezentuje jádrovou funkci pro dvouvrstvou sigmoidovou neuronovou síť.



*Obrázek 8: vlevo klasifikátor s RBF jádrem, vpravo s polynomiálním*

Na obrázku 8 jsou vidět dva nelineární klasifikátory. Zvýrazněné body reprezentují podpůrné vektory. Zatímco pro polynomiální klasifikátor je zapotřebí malé množství podpůrných vektorů, pro RBF jádro a složité problémy je třeba velké množství podpůrných vektorů. SVM klasifikátor s RBF jádrovou funkcí je často používaný, jelikož je schopen ve vysokodimenzionálním prostoru lineárně oddělit trénovací data (natrénovat se), která jsou

v původní dimenzi velmi složitá, lineárně neseparovatelná. Obrázek 8 ukazuje trénovací data a oblasti, které natrénovaný klasifikátor klasifikuje do třídy označené oranžově a do třídy označené zeleně.

### 3.2.4 Klasifikace nelineárního SVM

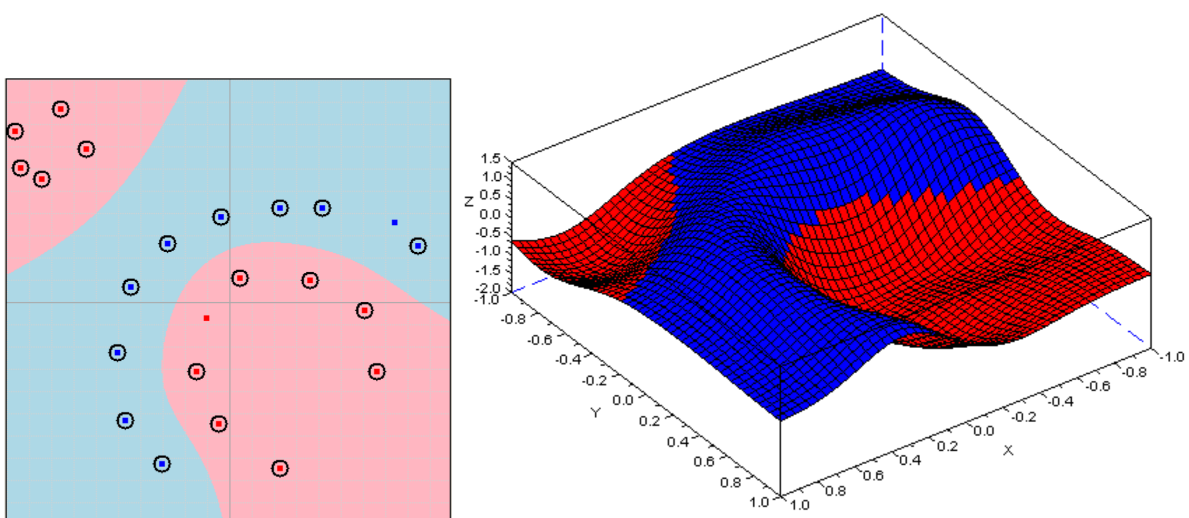
Nelineární Support Vector Machine, přestože je realizována vysokodimenzionálním lineárním SVM, typicky nezná parametr normálového vektoru $\vec{w}$ vysokodimenzionální hyperplochy. Proto nelze ke klasifikaci použít rovnici (23) jako u lineárního SVM.

Trénování SVM proběhlo na datech původní dimenze a SVM pracuje ve vysokodimenzionálním prostoru $H$, díky implicitní transformaci pomocí jádrové funkce. Rozhodovací funkcí pro takový klasifikátor je tedy

$$f\left(\vec{x}\right)=\sum_{i=1}^{N_s}\alpha_i y_i \Phi\left(\vec{s_i}\right)\cdot\Phi\left(\vec{x}\right)+b=\sum_{i=1}^{N_s}\alpha_i y_i K\left(\vec{s_i},\vec{x}\right)+b \tag{30}$$

kde $N_s$ je celkový počet podpůrných vektorů, $\alpha_i$ jsou lagrangeovi násobitele těchto vektorů a $\vec{s_i}$ jsou podpůrné vektory – *support vectors*. Výraz $K\left(\vec{s_i},\vec{x}\right)$ pak reprezentuje jádrovou funkci s operandy konkrétního podpůrného vektoru a klasifikovaného vzorku.

Výpočetní náročnost nelineárních klasifikátorů tedy není konstantní, ale je závislá na počtu podpůrných vektorů. Dále musí mít nelineární klasifikátor k dispozici pozice podpůrných vektorů v prostoru, což jsou vlastně vektory příznaků pro tyto body.




*Obrázek 9: 2D nelineární klasifikátor*

Na obrázku 9 je zobrazen nelineární SVM klasifikátor s jádrem RBF. Vlevo jsou ukázány trénovací vektory dvou tříd a barevně jsou odlišeny části plochy, ve kterých klasifikátor klasifikuje danou třídu. Vpravo je zobrazena klasifikační funkce téhož klasifikátoru. Tam, kde je plocha (hodnota dvourozměrné klasifikační funkce) ve výšce do 0, je klasifikována červená třída, tam kde je výška plochy nad 0, je klasifikována modrá třída.

# 4 Trénování SVM

Trénování a trénovací algoritmus SVM je základním prvkem pro konstrukci klasifikátoru. K trénování existuje celá řada softwarových knihoven, které implementují různé algoritmy, avšak metod, jak trénovat SVM klasifikátor v hardwaru (např. v hradlovém poli) zatím mnoho popsáno není.

Analytické řešení trénování SVM je možné pouze pokud je trénovací množina velmi malá, nebo v případě většího lineárního problému, pokud předem známe, které body se stanou podpůrnými vektory [1]. Pro obecné použití byly vyvinuty a nadále se rozvíjí metody, které využívají různých numerických metod a heuristik. V této kapitole jsou představeny dvě metody trénování SVM, které byly použity při řešení tohoto projektu.

## 4.1 SMO algoritmus

Sequential Minimal Optimization (SMO) je algoritmus pro řešení kvadratického optimalizačního problému (*QP problem*), který odpovídá trénování SVM klasifikátoru. Je schopen řešit velké problémy jejich dekompozicí na nejmenší možné menší QP podproblémy [2].

Nejmenším možným problémem během optimalizace je řešení dvou lagrangeových násobitelů v jednom kroku. Dva násobitele proto, že musejí splňovat podmínky linearity (14) a (15) a úpravou jednoho koeficientu je třeba zajistit splnění těchto podmínek upravením druhého koeficientu. Řešení dvou lagrangeových koeficientů za lineárních podmínek je zároveň možné provést jednoduše analyticky. Není třeba žádné numerické metody.

Výhodou tohoto algoritmu je, že ani pro velké problémy nemá nereálné požadavky na paměť, nepoužívá velkých matic řešených numericky a tím není tak citlivý na chyby přesnosti v numerických metodách.

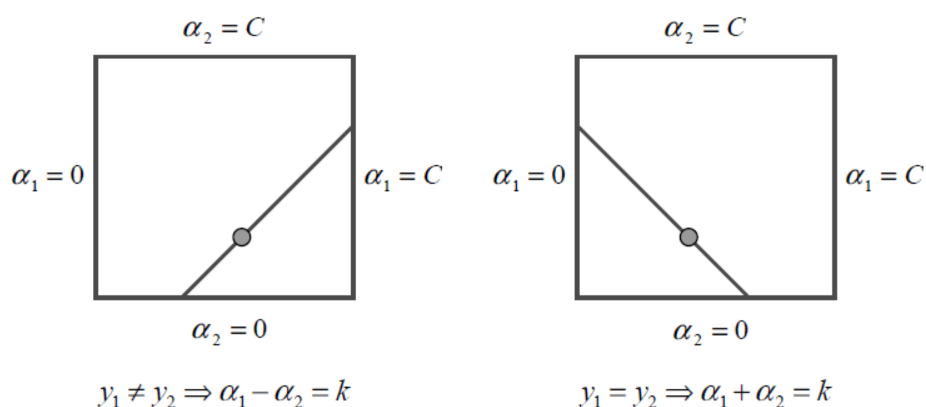
SMO algoritmus se skládá ze dvou významných částí:

- Optimalizace dvou lagrangeových násobitelů
- Heuristika k výběru těchto násobitelů

### 4.1.1 Optimalizace dvou koeficientů

Jak již bylo zmíněno, nejmenší možné množství lagrangeových koeficientů, které je možno v jednom kroku upravovat, jsou dva. Dále je v kapitole 3 uvedeno, jaké podmínky musejí lagrangeovi násobitele splňovat.

- Hraniční podmínky
    - $0 \le \alpha_i \le C$
- Podmínka linearity
    - $\sum_i \alpha_i y_i = 0$




*Obrázek 10: řešení optimalizace dvou koeficientů (převzato z [2])*

Z hraniční podmínky je zřejmé, že hodnoty koeficientů se nacházejí uvnitř čtverce o velikosti hrany $C$ a z podmínky linearity lze najít uvnitř tohoto čtverce úsečku, na které se musejí dané dva koeficienty nacházet. Tuto situaci ukazuje obrázek 10. Vpravo je případ, že se jedná o koeficienty dvou vektorů stejné třídy a vlevo je případ, že se jedná o koeficienty vektorů opačných tříd. Jeden krok algoritmu SMO tedy musí najít optimální pozici koeficientů na dané úseče z obrázku 10.

V následující části o algoritmu SMO bude dolních indexů 1 a 2 použito k rozlišení jednoho a druhého optimalizovaného koeficientu a dalších proměnných, spojených s těmito koeficienty. Algoritmus jako první řeší koeficient $\alpha_2$.

Nejdříve se stavoví hranice minimální a maximální možné hodnoty koeficientu $\alpha_2$ podle následující tabulky.

| | Spodní hranice | Horní hranice |
|---|---|---|
| $y_1 \neq y_2$ | $L = \max(0, \alpha_2 - \alpha_1)$ | $H = \min(C, C + \alpha_2 - \alpha_1)$ |
| $y_1 = y_2$ | $L = \max(0, \alpha_2 + \alpha_1 - C)$ | $H = \min(C, \alpha_2 + \alpha_1)$ |

*Tabulka 2: hraniční podmínky pro novou hodnotu $\alpha_2$*

Následně se spočte nová hodnota koeficientu $\alpha_2$:

$$\alpha_2^{new} = \alpha_2 + \frac{y_2(E_1 - E_2)}{K(\vec{x_1}, \vec{x_1}) + K(\vec{x_2}, \vec{x_2}) - 2K(\vec{x_1}, \vec{x_2})} \tag{31}$$

kde $E_i = f(\vec{x_i}) - y_i$ je aktuální klasifikační chyba pro trénovací vektor $\vec{x_i}$. Spočtenou hodnotu $\alpha_2^{new}$ je třeba oříznout, aby splňovala omezení stanovené tabulkou 2.

$$\alpha_2^{new,clipped} = \begin{cases} H & \text{pokud} \quad \alpha_2^{new} \geq H \\ \alpha_2^{new} & \text{pokud} \quad L < \alpha_2^{new} < H \\ L & \text{pokud} \quad \alpha_2^{new} \leq L \end{cases} \tag{32}$$

Výpočtem rovnice (32) dostaneme novou hodnotu koeficientu $\alpha_2$ a následně můžeme analyticky opravit hodnotu koeficientu $\alpha_1$:

$$\alpha_1^{new} = \alpha_1 + y_1 y_2 (\alpha_2 - \alpha_2^{new,clipped}) \tag{33}$$

### 4.1.2 Heuristika pro výběr koeficientů

V předešlé části je popsáno, jak se v jednom kroku optimalizace upravují hodnoty dvou koeficientů. Nejdříve je ale nutné určit, se kterými dvěma koeficienty se bude pracovat. Z informací obsažených v kapitole 3 je možné sestavit následující podmínky, které se označují jako KKT podmínky (*Karush-Kuhn-Tucker*)

$$\begin{aligned} &\alpha_i = 0 \Leftrightarrow y_i f(\vec{x_i}) \geq 1 \\ &0 < \alpha_i < C \Leftrightarrow y_i f(\vec{x_i}) = 1 \\ &\alpha_i = C \Leftrightarrow y_i f(\vec{x_i}) \leq 1 \end{aligned} \tag{34}$$

Výběr prvního koeficientu pro optimalizaci je dán vnější smyčkou algoritmu SMO, který prochází přes všechny trénovací data. Vhodným kandidátem na optimalizaci je první trénovací vzorek, který podmínky z (34) porušuje. Jakmile tato smyčka projde všechny trénovací vzorky, začne nový cyklus, kde se zkontroluje, zda trénovací vzorky, jejichž koeficienty nejsou ani $0$ ani $C$, porušují podmínky z (34). Pokud porušují, budou se nadále optimalizovat. Vykonávání těchto smyček se zastaví, jakmile všechny trénovací vektory vyhovují (s jistou přesností $\varepsilon$) KTT podmínkám.

Výběr druhého koeficientu pro optimalizaci je určen tak, aby během jednoho kroku došlo k co největšímu posunu v učení. SMO algoritmus si v rámci urychlení udržuje v paměti klasifikační chyby $E_i$ pro trénovací vektory. Pokud je tato chyba $E_1$ pro první zvolený

koeficient kladná, pak je jako koeficient $\alpha_2$ vybrán takový, který má chybu $E_2$ minimální. Pokud je chyba $E_1$ záporná, pak je vybrán koeficient $\alpha_2$ s maximální chybou $E_2$.

### 4.1.3 Výpočet posunutí

Po každém kroku optimalizace je zároveň aktualizována hodnota posunutí $b$. Ta je dána průměrem $b = (b_1 + b_2)/2$, kde:

$$\begin{aligned} b_1 &= E_1 + y_1(\alpha_1^{new} - \alpha_1)K(\vec{x_1}, \vec{x_1}) + y_2(\alpha_2^{new,clipped} - \alpha_2)K(\vec{x_1}, \vec{x_2}) + b \\ b_2 &= E_2 + y_1(\alpha_1^{new} - \alpha_1)K(\vec{x_1}, \vec{x_2}) + y_2(\alpha_2^{new,clipped} - \alpha_2)K(\vec{x_2}, \vec{x_2}) + b \end{aligned} \tag{35}$$

### 4.1.4 Shrnutí

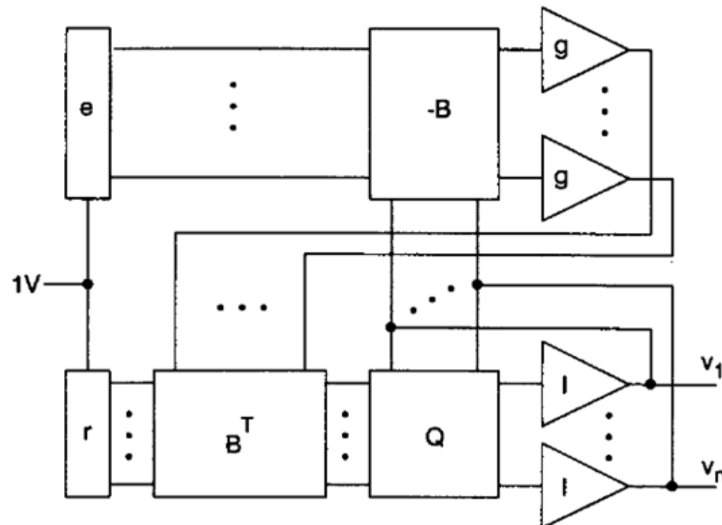
SMO algoritmus je jednoduchou a poměrně rychlou metodou [2], jak řešit trénování SVM. Na rozdíl od jiných metod řeší nejmenší možný QP problém dvou koeficientů analyticky. Nevýhodou s ohledem na implementaci trénování ve vestavěných systémech je, že během jednotlivých cyklů trénování je třeba často počítat klasifikační funkci (odezvu klasifikátoru v aktuálním stavu trénování) na trénovací vektory. Pokud by byla zvolena nějaká jádrová funkce, nemusí být triviální ji počítat s omezenými výpočetními prostředky. Implementace trénování SMO algoritmem byla provedena softwarově, v počáteční fázi řešení tohoto projektu k bližšímu seznámení s funkcí a vlastnostmi SVM.

Pseudokód algoritmu SMO, převzatý z [2] je uveden v příloze této práce.

## 4.2 HW-friendly trénování

Trénovací algoritmy SVM které byly koncem 90.let minulého století známy, nebyly vhodné pro trénování SVM v hardwaru [5]. Neexistovala metodika, jak implementovat trénování ve vestavěných systémech s omezenými zdroji, nebo v masivně se rozvíjejících hradlových polích.

V roce 1998 představil D. Anguita s kolegy rekurentní neuronovou síť obsahující integrátory, která konverguje ke stabilnímu bodu splňujícímu KTT podmínky a je řešením daného optimalizačního problému – trénování SVM [5]. To byl první krok k realizaci hardwarového trénování. Pro ilustraci je zmíněná rekurentní síť na obrázku níže.




*Obrázek 11: obvodová realizace trénování (převzato z [5])*

### 4.2.1 VLSI algoritmy

Algoritmy, které jsou dále prezentovány, jsou v literatuře označeny jako VLSI algoritmy. Nejedná se o název jednoho konkrétního algoritmu, nýbrž o souhrnné označení pro metody, vhodné pro obvodovou realizaci.

#### 4.2.1.1 Optimalizace lagrangeových koeficientů

Duální lagrangian k minimalizaci (13), uvedený v kapitole 3.1, je možné přepsat do následujícího výrazu:

$$L_D\left(\vec{\alpha}\right)=\frac{1}{2}\vec{\alpha}'\vec{Q}\vec{\alpha}+\vec{r}'\vec{\alpha} \tag{36}$$

kde $\vec{\alpha}$ je vektor lagrangeových koeficientů, $\vec{r}, r_i=-1\ \forall i$ je vektor samých $-1$, $\vec{Q}$ je matice hodnot $q_{i,j}=y_i y_j \vec{x_i}\cdot\vec{x_j}$ a $i, j$ indexují trénovací data. Podmínky řešení zůstávají stejné, jako jsou uvedeny v kapitole 3.1. Z vlastností matice $\vec{Q}$ a ze zmíněných podmínek vyplívá, že tento problém nemá lokální minimum [7], takže při hledání řešení nehrozí uváznutí právě v tomto lokálním minimu.

Na základě spojité rekurentní výpočetní sítě, popsané v roce 1988 L. Chuaem, která byla určena k řešení nelineárního optimalizačního problému, sestavil D. Anguita s kolegy podobnou síť pracující ve spojitém čase, určenou k trénování SVM [7] - viz. obrázek 11. Z diferenciální rovnice, která reprezentuje funkci této sítě, je pak v [7] integrací Eulerovou metodou získána rovnice rekurentní sítě, pracující v diskrétním čase:

$$\vec{z}^{k} = \vec{\alpha}^{k} - \eta \nabla \vec{E} \quad (37)$$

$$\alpha_i^{k+1} = u(z_i^k) \quad (38)$$

kde

$$\nabla \vec{E} = \vec{Q}\vec{\alpha} + \vec{r} \quad (39)$$

$$u(v) = \begin{cases} C & v > C \\ v & 0 \le v \le C \\ 0 & v \le 0 \end{cases} \quad (40)$$

Horní indexy ve výše uvedených rovnicích udávají diskrétní výpočetní krok a funkce $u()$ slouží k ořezání nových hodnot koeficientů $\alpha$ na hodnoty z intervalu $\langle 0, C \rangle$. Počáteční hodnoty koeficientů $\alpha$ jsou nulové, tedy $\vec{\alpha}^{0} = (0,0,0,...,0)$.

Diskrétní diferenciální rovnice (37) a (38) je zajímavá z pohledu hardwarového řešení, protože není složitá. Jde o násobení vektorů a matic. Navíc v [7] je uvedeno, že toto řešení konverguje ke globálnímu minimu pravděpodobně v exponenciálním čase, pokud

$$\eta \le \frac{2}{n q_{max}} \quad (41)$$

kde $n$ je počet trénovacích dat a $q_{max} = \max\{|q_{i,j}|\}$ je maximum z absolutních hodnot prvků matice $\vec{Q}$. Iterační proces výpočtu je pak ukončen typicky v situaci, kdy změna velikosti koeficientů $\alpha$ bude menší, než stanovená hodnota $\varepsilon$: $|\alpha_i^k - \alpha_i^{k+1}| < \varepsilon, \forall i$.

Problém při použití tohoto samotného řešení je, že nepočítá s parametrem posunutí $b$ klasifikátoru SVM, čímž neumožňuje použití pro lineární úlohy. Pokud však použijeme jádrovou funkci RBF, čímž transformujeme problém do vysokodimenzionálního prostoru příznaků, je možné najít takové řešení, kdy oddělující hyperplocha prochází počátkem tohoto prostoru příznaků ($b = 0$). Jak již bylo zmíněno, tak použití jádrových funkcí nemá vliv na trénovací algoritmus, takže samotnou transformaci provedeme změnou způsobu výpočtu vstupní matice $\vec{Q}$:

$$q_{i,j} = q_{j,i} = y_i y_j K(\vec{x_i}, \vec{x_j}) \quad (42)$$

#### 4.2.1.2 Optimalizace prahu klasifikátoru

VLSI algoritmus popsaný v předešlé kapitole žádným způsobem neupravuje práh klasifikátoru $b$. Iteračním procesem pouze hledá koeficienty $\alpha$ a předpokládá, že $b=0$. Tento předpoklad omezuje použití tohoto algoritmu na lineární data a je nutné použít transformaci do prostoru příznaků, kde bude možno najít oddělující hyperplochu procházející počátkem souřadného systému v tomto prostoru příznaků. Dále je v [8] uvedeno, že v reálných problémech pak může mít natrénovaný klasifikátor problém s teoreticky deklarovanou dobrou generalizací SVM.

Metoda, jak hledat vhodnou hodnotu posunutí $b$ během trénování je uvedena v [8]. Z hodnoty výrazu $s=\operatorname{sgn}(\vec{y}\cdot\vec{\alpha})$ lze stanovit interval, ve kterém se nachází optimální hodnota $b_{opt}$.

| | |
|---|---|
| $s<0$ | $b_{opt}<b$ |
| $s>0$ | $b_{opt}>b$ |
| $s=0$ | $b_{opt}=b$ |

Algoritmus hledání optimálního řešení pracuje s intervalem hodnot posunutí $b$ - $\langle b_l,b_h\rangle$. Hledá tedy nejdříve takový interval, kde $b_l>0$ a $b_h<0$. Pak optimální hodnota prahu klasifikátoru leží v intervalu $\langle b_l,b_h\rangle$ a pomocí bisekce se hledá na tomto intervalu bod, ve kterém $s=0 \Leftrightarrow b=b_{opt}$.

Počáteční hodnota $b$ není známa, ale podle [8] je možné začít s intervalem $\langle -1,1\rangle$. Nejdříve se tedy hledají takové hranice tohoto intervalu tak, že $b_l<b_{opt}<b_h$ a následně se na tomto intervalu bisekcí hledá $b_{opt}$. Hledání je ukončeno, jakmile bude interval $\langle b_l,b_h\rangle$ užší, než stanovená konstanta $\varepsilon_b$.

Pro zohlednění úpravy prahu $b$ je v iteračním algoritmu optimalizace $\vec{\alpha}$ nahrazen v rovnici (39) parametr $\vec{r}$ parametrem $\vec{r}\,'=\vec{r}-\vec{y}b$.

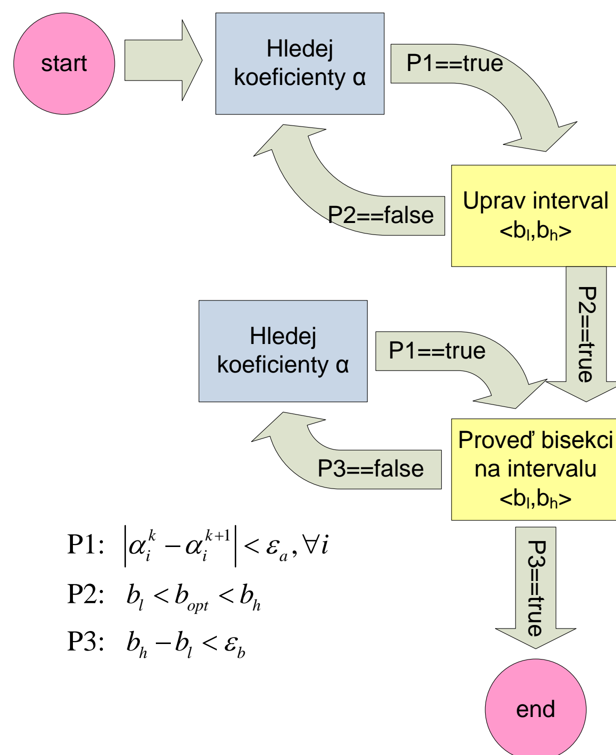
### 4.2.2 Shrnutí

VLSI algoritmy byly navrženy s ohledem na jejich práci v hardwaru. Jelikož obsahují pouze jednoduché aritmetické operace, lze je snadno implementovat i v hradlových polích FPGA a jsou potencionálně vhodné k velké paralelizaci. Během trénování není třeba vyhodnocovat

klasifikační funkci, což znamená, že v případě nelineárních SVM s jádrem není třeba během trénování počítat exponenciální funkce (pro RBF jádro jako např. u algoritmu SMO), což by způsobovalo zejména pro implementaci v FPGA komplikace. Dále je podle [7] a [8] možno tyto algoritmy efektivně implementovat na architekturách, používajících aritmetiku s pevnou řádovou čárkou – *fixed point*.

Vstupem trénování je matice hodnot $\vec{Q}$, vypočtená z trénovacích dat. Tato matice je během trénování nadále konstantní – nedochází k její modifikaci. Výstupem trénování je pak vektor koeficientů $\vec{\alpha}$, alternativně doplněný o hodnotu posunutí $b$. Nevýhodou tohoto přístupu k trénování je, že matice $\vec{Q}$ může dosahovat značné velikosti. Pro $n$ trénovacích vektorů má matice $n^2$ prvků.

Pseudokód algoritmů VLSI, převzatý z [8] je v příloze této práce. Jednoduché schéma výpočtu je na obrázku níže.




*Obrázek 12: schéma algoritmu s výpočtem prahu klasifikátoru*

# 5 Implementace trénování

Tento projekt má za cíl implementovat algoritmus trénování SVM v FPGA hradlovém poli. Z různých způsobů trénování, kde dva z nich jsou popsány v předchozí kapitole, byl s ohledem na cílovou architekturu vybrán VLSI algoritmus z kapitoly 4.2 . Důvody, vedoucí k tomuto rozhodnutí jsou uvedeny v poslední části předešlé kapitoly.

## 5.1 Aritmetika

Vstupem trénovacího procesu je matice reálných čísel $\vec{Q}$ spočítaná ze sady trénovacích dat. Výpočet této matice byl prozatím ponechán na uživateli, jelikož při použití jádrové funkce RBF by tento výpočet v FPGA přinášel komplikace. Dostupná literatura se také nezmiňuje o výpočtech jádrové funkce přímo v hradlovém poli. Pokud se bude realizovat algoritmus, který navíc provádí i optimalizaci prahu klasifikátoru $b$ , pak je třeba k trénování ještě vektor $\vec{y}$ , obsahující označení tříd trénovacích vektorů $y_i \in \{-1,1\}$ .

Výstupem trénování je vektor $\vec{\alpha}$ s lagrangeovými koeficienty a případně upravený práh $b$ . Paměťové nároky na velikost vstupních a výstupních dat je uvedena v následující tabulce.

| Počet trénovacích vektorů | $n$ |
|---|---|
| Velikost matice $\vec{Q}$ | $n \cdot n$ |
| Velikosti vektorů $\vec{\alpha}$ a $\vec{y}$ | $n$ |

*Tabulka 3*

### 5.1.1 Reprezentace parametrů

Hradlové pole FPGA a nástroje pro syntézu primárně nepodporují výpočty s čísly v plovoucí desetinné čárce – *floating point*. Pokud by bylo třeba použít těchto výpočtů, pak by bylo třeba sáhnout po nějaké knihovně, která by realizovala tyto výpočty. Nicméně by se tím zvýšilo množství potřebné logiky na čipu a pravděpodobně i snížila rychlost výpočtu.

Dle experimentů, uvedených v [7], je možné realizovat výpočty nutné k trénování SVM i s čísly s pevnou desetinnou čárkou – *fexed point*. Uvedené experimenty uvádějí, že postačí 12 bitů k zakódování členů matice $\vec{Q}$ a 16 bitů k zakódování ostatních parametrů - $\vec{\alpha}$ , $b$ . Pro jednoduchost používá realizace tohoto projektu všechny parametry jako 16-ti

bitové, což odpovídá šířce jednoho slova v cílovém systému, kam bude trénování SVM umístěno.

| Parametr | Znaménkové | Celá část | Desetinná část | Rozsah hodnot |
|---|---|---|---|---|
| $q_{i,j}$ | Ano | 1 | 15 | $-1 \le q_{i,j} \le 1-2^{-15}$ |
| $\vec{\alpha}$ | Ano | $r$ | $s \qquad (s=16-r)$ | $-2^{r-1} \le b \le 2^{r-1}-2^{-s} \triangleq C$ |
| $b$ | Ano | $r$ | $s \qquad (s=16-r)$ | $-2^{r-1} \le b \le 2^{r-1}-2^{-s}$ |

*Tabulka 4*

### 5.1.2 Základní výpočet

Zvolená 16-ti bitová reprezentace parametrů koresponduje i se snadno použitelnými komponentami z FPGA, jako jsou blokové paměti a vestavěné hardwarové násobičky.

Parametry $q_{i,j}$, $\alpha_i$ a $y_i$ jsou jako 16-ti bitové uloženy v blokové paměti [9] FPGA, ostatní parametry a pomocné proměnné (akumulátory) jsou v dostatečné datové šířce v distribuované paměti na čipu FPGA.

Dle rovnic z kapitoly 4.2.1.1 je pro výpočet každého nového koeficientu $\alpha_i^{k+1}$ nutné provést $n$ operací MAC (násob a sčítej) mezi prvky matice a aktuálními hodnotami koeficientů $\alpha_i^k$, kde $n$ je počet trénovacích vektorů. Nový koeficient $\alpha_i^{k+1}$ se tedy spočte následujícím způsobem:

$$\alpha_i^{k+1} = \alpha_i^k - \eta \sum_{j=1}^{n} (q_{i,j}\alpha_j^k - y_j b + 1) \tag{43}$$

Pro násobení dvou 16-ti bitových čísel je třeba výsledek reprezentovat na 32 bitech. Jelikož se toto násobení provádí $n$-krát a přičítá do akumulačního registru, je třeba zajistit, aby během akumulace součinů nedošlo k přetečení výsledku. To zajistíme tak, že k akumulátoru přidáme potřebný počet doplňkových bitů. Pro $n$ sčítání postačí rozšířit akumulátor o $\lceil \log_2 n \rceil$ bitů. Během akumulace je dále přičítána hodnota $-y_j b + 1$, kterou je třeba reprezentovat na korektní datové šířce a posunout tak, aby odpovídala pozici desetinné čárky v součinu $q_{i,j}\alpha_j^k$.

Po dokončení výpočtu sumy z rovnice (43) je třeba výsledek násobit konstantou $\eta$ tak, aby byla splněna podmínka konvergence (41). Toto násobení může být nahrazeno

bitovým posunem vpravo (násobí se číslem menším než 1) o příslušný počet míst $\lceil \log_2 1/\eta \rceil$, který bude zaokrouhlen na nejbližší vyšší celé číslo.

Výsledek je pak odečten od původní hodnoty $\alpha_i^k$, která je též reprezentována na rozšířené bitové šířce akumulátoru. Nově spočtená hodnota $\alpha_i^{k+1}$ je pak oříznuta na intervalu $\langle 0, C \rangle$ a uložena zpět do paměti.

V jednom kroku je třeba spočítat všech $n$ nových parametrů $\alpha_i^{k+1}$. Jelikož parametry $\alpha_i^{k+1}$ nejsou mezi sebou nijak závislé, lze je počítat paralelně, např. $x$ jednotkami a ukončit tak výpočet jednoho kompletního iteračního kroku až $x$-krát rychleji.

Po dokončení jednoho iteračního kroku je rozhodnuto, zda se bude provádět další krok k výpočtu vektoru $\vec{\alpha}^{k+1}$. Podmínkou pro ukončení výpočtů nových kroků je:

$$\left| \alpha_i^k - \alpha_i^{k+1} \right| < \varepsilon_\alpha, \forall i \tag{44}$$

### 5.1.3 Kompletní architektura

Realizované SVM trénování je včetně úpravy parametru prahu $b$ klasifikátoru. To znamená, že je na počátku inicializována hodnota parametrů $b$, $b_l$, $b_h$ a je spuštěn iterační výpočet z 0. Po dosažení ukončující podmínky (44) je na základě $s = \mathrm{sgn}(\vec{y} \cdot \vec{\alpha})$ určen další postup, dle pseudokódu v příloze 3. Příslušným způsobem se upraví parametry $b$, $b_l$, $b_h$ a opět se spustí základní výpočet z kapitoly 0. To se opakuje, dokud není nalezena optimální hodnota prahu $b$.

## 5.2 Dekompozice trénování

Problematika trénování v FPGA je rozdělena do čtyř základních komponent, které jsou uvedeny v následující tabulce.

| **`update_b`** | Základní komponenta, která obsahuje všechny tři další komponenty, uvedené v tabulce. Řeší optimalizaci prahu klasifikátoru. |
|---|---|
| `mm_Qa` | Jednotka, která provádí optimalizaci lagrangeových násobitelů. |
| `sum_ay` | Jednotka, která počítá parametr $s$ pro použití hlavní komponentou. |

| RAM_Q_a | Paměť dat. |
|---|---|

*Tabulka 5: komponenty algoritmu*

### 5.2.1 Komponenta `update_b`

Toto je základní komponenta řešení SVM trénování. Poskytuje rozhraní pro zápis vstupních parametrů trénování do vnitřní paměti a pro čtení výsledku trénování ven. Uvnitř jsou obsaženy sub-komponenty pro `mm_Qa`, `sum_ay` a `RAM_Q_a`.

**Funkce:**

- Adresní dekodér pro čtení/zápis vstupu/výstupu
- Stavový automat řešící optimalizaci prahu $b$ a spouštění optimalizace koeficientů $\alpha$

**Rozhraní:**

| | |
|---|---|
| `reset : in std_logic;` | Reset |
| `clk : in std_logic;` | Hodinový signál |
| `data_in : in std_logic_vector(W_ABQ-1 downto 0);` | Datový vstup |
| `data_out : out std_logic_vector(W_ABQ-1 downto 0);` | Datový výstup |
| `write_q : in std_logic;` | zápis do paměti pro $\vec{Q}$ |
| `write_y : in std_logic;` | Zápis do paměti pro $\vec{y}$ |
| `write_n : in std_logic;` | Zápis počtu trénovacích vektorů |
| `read_a : in std_logic;` | Čtení z paměti pro $\vec{\alpha}$ |
| `read_b : in std_logic;` | Čtení prahu $b$ klasifikátoru |
| `addr : in integer := 0;` | Adresa pro zápis/čtení |
| `ub_start : in std_logic;` | Spuštění trénování nastavením na 1 |
| `ub_clr : in std_logic;` | Uvedení do výchozího stavu (zatím nevyužito) |
| `ub_end : out std_logic` | Trénování ukončeno, pokud je signál 1 |

**Použití:**

Nejprve je třeba komponentu naplnit parametry pro trénování. Jedná se o zápis prvků matice $\vec{Q}$, při aktivním signálu `write_q`. Zápis je adresován signálem `addr`. Koeficienty matice jsou v paměti umístěny po řádcích za sebou. Stejným způsobem se do paměti pro zapíší hodnoty vektoru $\vec{y}$. Dále je třeba pomocí signálu `write_n` uložit do komponenty počet trénovacích dat.

Po naplnění komponenty vstupními daty je možno spustit trénování nastavením logické 1 na vstup `ub_start`. Jakmile je trénování dokončeno, je nastaven signál `ub_start` na 1.

Natrénované hodnoty je třeba vyčíst pomocí signálů `read_a, read_b`. Data se čtou z datového výstupu a čtení vektoru $\vec{\alpha}$ je opět adresováno signálem `addr`.

### 5.2.2 Komponenta `mm_Qa`

Tato komponenta realizuje optimalizaci koeficientů $\alpha$. Generuje signály pro čtení a zápis hodnot z a do paměti. Jednotka čte z paměti hodnoty vektoru $\vec{\alpha}^k$ a do jiné paměti zapisuje nově spočtené hodnoty $\vec{\alpha}^{k+1}$. Dokud není ukončen jeden kompletní krok výpočtu, je třeba udržovat hodnoty původního vektoru $\vec{\alpha}^k$, proto jsou pro tyto koeficienty vyhrazeny dvě paměti. Jejich přepínání se pak realizuje signálem `sel_aact` po ukončení každého kroku výpočtu.

**Rozhraní:**

| | |
|---|---|
| `reset : in std_logic;` | Reset |
| `clk : in std_logic;` | Hodinový signál |
| `BLOCK_START : in std_logic;` | Signál pro spuštění výpočtu |
| `BLOCK_END : out std_logic;` | Běh výpočtu |
| `BLOCK_RUN : out std_logic;` | Oznámení ukončení výpočtu |
| `n : in integer;` | Počet trénovacích vektorů |
| `Q_ij : in std_logic_vector(W_ABQ-1 downto 0);` | Vstup pro členy matice $\vec{Q}$ |
| `a0_i : in std_logic_vector(W_ABQ-1 downto 0);` | Vstup pro koeficienty $\alpha_i^k$ |
| `b : in std_logic_vector(W_ABQ-1 downto 0);` | Vstup pro aktuální velikost prahu |
| `y_sgn : in std_logic;` | Vstup pro hodnoty $y_i$ |
| `a1_i : out std_logic_vector(W_ABQ-1 downto 0);` | Výstup nové hodnoty $a_i^{k+1}$ |
| `a1_we : out std_logic;` | Zápis nové hodnoty |
| `addr_a0 : out integer := 0;` | Adresa pro čtení $\alpha_i^k$ |
| `addr_a1 : out integer := 0;` | Adresa pro zápis $a_i^{k+1}$ |
| `addr_Q : out integer := 0;` | Adresa pro čtení $q_{i,j}$ |
| `sel_aact : out std_logic;` | Výběr mezi pamětí pro $\vec{\alpha}^k$ a $\vec{\alpha}^{k+1}$ |

**Použití:**

Spuštění jedné kompletní optimalizace koeficientů $\alpha$ je zahájeno aktivací signálu BLOCK_START. Jakmile je dosaženo ukončující podmínky (44) je výpočet ukončen a generován signál BLOCK_END. Hodnota $\varepsilon_\alpha$, která má vliv na ukončení trénování je uložena jako konstanta v této komponentě.

### 5.2.3 Komponenta `sum_ay`

Komponenta `sum_ay` realizuje výpočet funkce $s = \mathrm{sgn}(\vec{y} \cdot \vec{\alpha})$, která slouží k rozhodování o způsobu optimalizace prahu $b$ klasifikátoru.

**Rozhraní:**

| | |
|---|---|
| `reset : in std_logic;` | Reset |
| `clk : in std_logic;` | Hodinový signál |
| `BLOCK_START : in std_logic;` | Signál pro spuštění výpočtu |
| `BLOCK_END : out std_logic;` | Běh výpočtu |
| `BLOCK_RUN : out std_logic;` | Oznámení ukončení výpočtu |
| `n : in integer;` | Počet trénovacích vektorů |
| `y_i : in std_logic_vector(W_ABQ-1 downto 0);` | Vstup pro hodnoty $y_i$ |
| `a_i : in std_logic_vector(W_ABQ-1 downto 0);` | Vstup pro koeficienty $\alpha_i$ |
| `res : out std_logic_vector(W_ACC-1 downto 0);` | Výsledek výpočtu |

**Použití:**

Výpočet je zahájen signálem `BLOCK_START` ihned po dokončení výpočtu jednotky `mm_Qa` a výsledek je použit pro nadřazenou jednotku `b_update` k rozhodnutí, jak optimalizovat práh $b$ klasifikátoru.

### 5.2.4 Komponenta `RAM_Q_a`

Tato komponenta obaluje použité blokové paměti pro uchování pracovních dat.

**Rozhraní:**

| | |
|---|---|
| `reset : in std_logic;` | Reset |
| `clk : in std_logic;` | Hodinový signál |
| `Q_in : in std_logic_vector(W_ABQ-1 downto 0);` | Zapisovaná hodnota $q_{i,j}$ |
| `Q_out : out std_logic_vector(W_ABQ-1 downto 0);` | Čtená hodnota $q_{i,j}$ |

| | |
|---|---|
| `a_in :`<br>` in std_logic_vector(W_ABQ-1 downto 0);` | Zapisovaná hodnota $\alpha_i$ |
| `a_out :`<br>` out std_logic_vector(W_ABQ-1 downto 0);` | Čtená hodnota $\alpha_i$ |
| `y_in :`<br>` in std_logic_vector(W_ABQ-1 downto 0);` | Zapisovaná hodnota $y_i$ |
| `y_out :`<br>` out std_logic_vector(W_ABQ-1 downto 0);` | Čtená hodnota $y_i$ |
| `addr_r_Q : in integer;`<br>`addr_r_a : in integer;`<br>`addr_r_y : in integer;` | Adresy pro čtení z pamětí |
| `addr_w_Q : in integer;`<br>`addr_w_a : in integer;`<br>`addr_w_y : in integer;` | Adresy pro zápis do pamětí |
| `we_a : in std_logic;`<br>`we_y : in std_logic;`<br>`we_Q : in std_logic;` | Signály pro povolení zápisů do pamětí |

### 5.2.5 Testování

Pro simulaci trénování SVM v programu ModelSim (nebo ISE) je k dispozici testovací komponenta (testbench) tb_update_b. Tato komponenta nejdříve zapíše počet trénovacích vektorů. Následně ze souboru načte matici $\vec{Q}$ a vektor $\vec{y}$. Poté je trénování připraveno a spuštěno. Jakmile je trénování dokončeno, jsou výsledky zapsány do výstupního souboru.

| | |
|---|---|
| `Q_FILE` | Vstupní soubor s maticí $\vec{Q}$. Tento soubor musí obsahovat $n \cdot n$ čísel, každé na jednom řádku. Povoleny jsou znaménková 16-ti bitová celá čísla. |
| `Y_FILE` | Vstupní soubor s vektorem $\vec{y}$. Tento soubor musí obsahovat $n$ čísel, každé na jednom řádku. Povolené hodnoty pro vektor jsou +1 a -1. |
| `A_FILE` | Výstupní soubor. Na prvním řádku obsahuje `b číslo`, a na dalších řádcích hodnoty natrénovaného vektoru $\vec{\alpha}$. Všechny hodnoty jsou znaménková 16-ti bitová čísla celá čísla. |

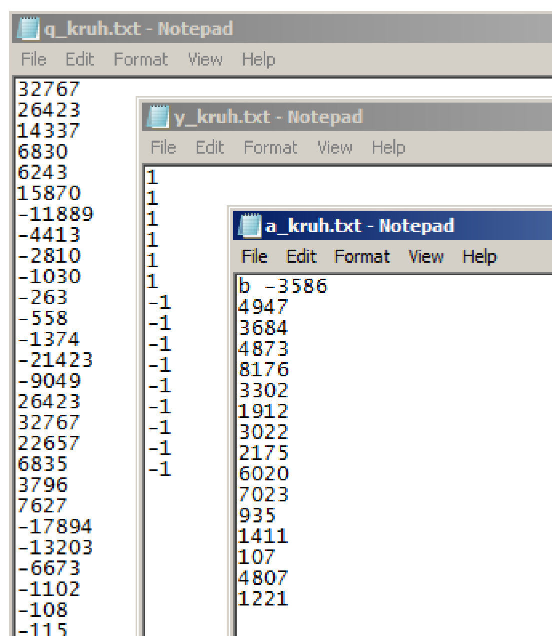
*Tabulka 6: vstupní/výstupní soubory*

#### 5.2.5.1 Formát čísel v testovacích souborech

Prvky matice $\vec{Q}$ se vypočítají aplikací jádrové funkce. V případě použití RBF jádrové funkce je zajištěno, že hodnoty matice jsou v intervalu $\langle -1,1 \rangle$. Pro interpretaci jako 16-ti bitového znaménkového čísla je třeba hodnoty násobit číslem $2^{15}$ a zaokrouhlit na celé číslo. Musí se

ale dát pozor, aby v případě hodnoty původní +1 nedošlo k přetečení do záporných čísel, jelikož maximální povolená kladná hodnota je $1-2^{-15}$. Přepočet lze tedy doplnit jednoduchou podmínkou, že pokud převedená celočíselná hodnota je $2^{15}$, pak se do souboru Q_FILE s prvky matice $\vec{Q}$ zapíše hodnota $2^{15}-1$.

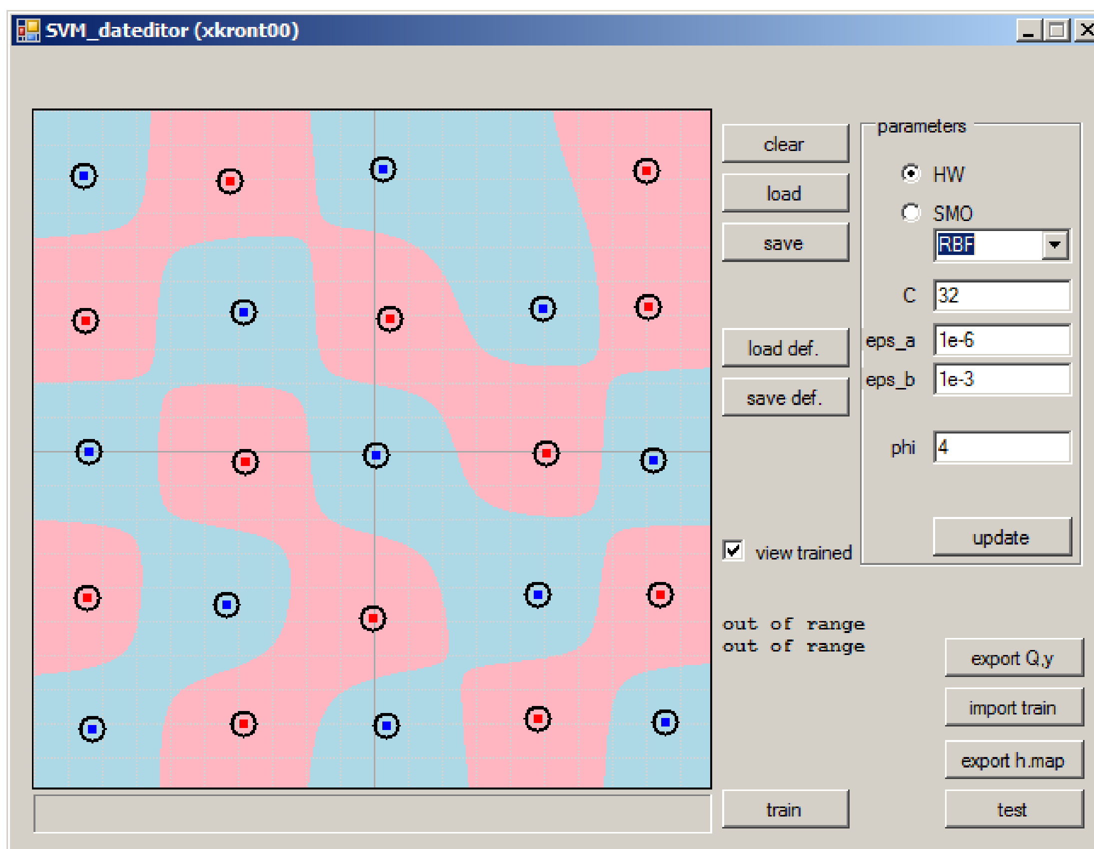
Formát výstupního souboru A_FILE je naznačen v předchozí tabulce. Obsahuje $n$ prvků natrénovaného vektoru $\alpha_i$ a jednu hodnotu prahu $b$. Jedná se opět o 16-ti bitová znaménková čísla, jedno na každém řádku. Řádek s hodnotou prahu je uvozen znakem b. Tato čísla pak lze interpretovat jako desetinná čísla po jejich vydělení hodnotou $2^s$, kde $s$ je stanovený počet bitů, k reprezentaci desetinné části čísla.




*Obrázek 13: ukázka formátu vstupních/výstupních souborů*

## 5.2.6 Vizualizační SW

K prvotnímu seznámení s procesem trénování SVM byl vytvořen program, který realizoval trénování a zobrazení výsledků trénování. Později byl tento program doplněn o další užitečné funkce, které umožňují generovat soubory pro simulaci trénování a přečíst a zobrazit výsledky trénování.



*Obrázek 14: podpůrný software*

Program `SVM_dateditor` je omezen na problémy dimenze 2, což znamená, že pracuje s daty, kde vektor příznaků je složen ze dvou reálných čísel. Pracovní plocha programu zobrazuje část plochy, kde spodní levý roh odpovídá souřadnici [-1,-1] a horní pravý roh odpovídá souřadnici [1,1]. Kliknutím levým nebo pravým tlačítkem myši do pracovní oblasti lze přidat pozitivní nebo negativní (modrý nebo červený) bod na dané umístění.

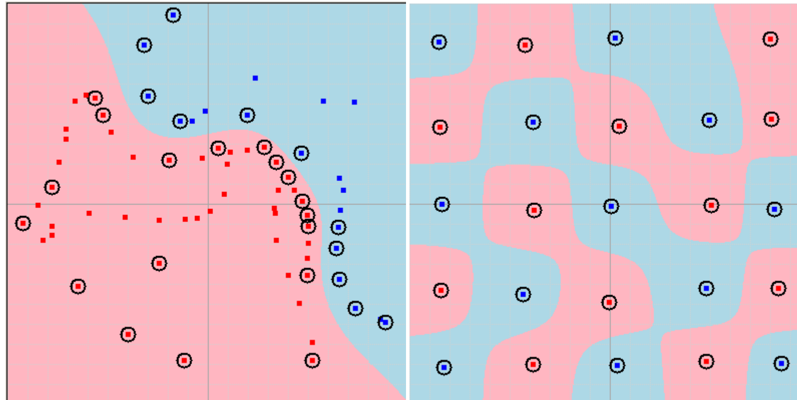
Program je napsán v jazyce C# v prostředí .NET 3.5 a více informací o jeho ovládání je uvedeno v příloze této práce.

## 5.2.7 Test

Kvalita trénování byla testována v simulaci, za podpory programu představeného v předešlé části.

Pro problémy do velikosti přibližně 50ti trénovacích bodů bylo možno úpravou trénovacích parametrů dosáhnout situace, kde všechny trénovací body byly po ukončení trénování správně klasifikovány – jak při trénování v software, tak i při trénování v simulátoru. Rozdíl mezi trénováním v software (*floating-point*) a v hardware (*fixed point*), při vhodném nastavení parametrů trénování, není znatelný. Ve většině případů měla řešení, trénovaná simulátorem o jeden až tři podpůrné vektory více. Vzhledem k tomu, že doba

klasifikace je přímo úměrná počtu podpůrných vektorů, byla by pak klasifikace výpočetně náročnější.



*Obrázek 15: ukázka natrénovaného klasifikátoru (trénováno v simulátoru)*

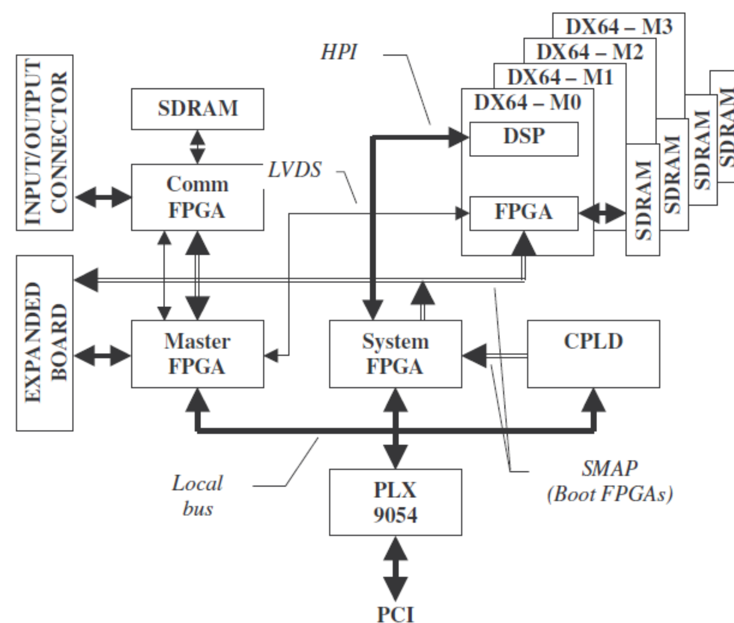
Další testy byly provedeny s databází *Banana*, dostupné na internetu [12]. Tato databáze obsahuje trénovací sady a testovací sady údajů. Trénovací sady obsahují 400 vzorků, testovací sady pak obsahují 4900 vzorků. Experimentováním s různým nastavením parametrů učení bylo dosaženo chyby klasifikátoru na trénovacích datech asi 1% a chyby na testovacích datech asi 9% a 7% pro případy trénování v simulátoru a v software. Časová náročnost trénování pro 400 trénovacích vzorků byla přibližně 325e6 taktů, což by bylo při pracovní frekvenci 100MHz 3,25 sekund.

# 6 Platforma Uni1P/DX64

Realizovaná jednotka trénování SVM byla je dále upravena k provozu na desce DX64. Jedná se o speciální výpočetní systém, skládající se ze signálového procesoru DSP a hradlového pole FPGA.

## 6.1 Popis

Programovatelné výpočetní moduly DX64 se osazují do desky Uni1P. Tato deska je pak pomocí sběrnice PCI připojena k osobnímu počítači a tvoří rozhraní mezi osobním počítačem a až 4-mi moduly DX64. Blokové schéma této desky je na obrázku 16.




*Obrázek 16: blokové schéma desky Uni1P, převzato z [10]*

Systémové FPGA slouží ke konfiguraci FPGA čipů na deskách DX64. Dále je systémové FPGA připojeno ke sběrnici HPI, která slouží ke komunikaci s modulem DX64. Konfigurace FPGA čipu na modulu DX64 je pak složena ze dvou částí.

1. Standardní komunikační rozhraní s okolím
2. Uživatelsky definované funkční jednotky

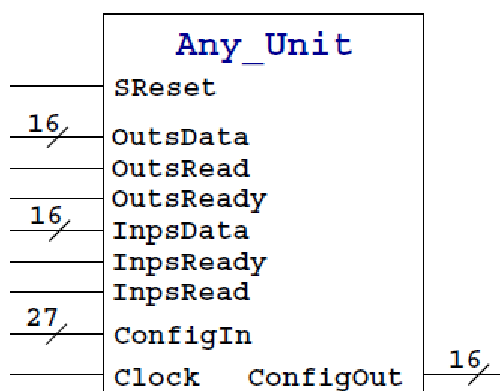
Právě pro uživatelské funkční jednotky je komponenta `update_b` trénování SVM () obalena do komponenty `TOP_FU`, která přizpůsobuje jednotlivá komunikační rozhraní.

Deska DX64 obsahuje signálový procesor TI řady C6416 a FPGA čip Xilinx Virtex II 250, nebo Xilinx Virtex II 1000.

### 6.1.1 CICB

CICB tvoří architekturu kruhové propojovací sítě (odlehčenou variantu křížového přepínače) v FPGA na desce DX64 [11]. Propojovací síť poskytuje rozhraní pro komunikaci s uživatelsky definovanými funkčními jednotkami v FPGA. Blokové schéma této jednotky je přiloženo na datovém nosiči, který je součástí této práce.

Funkční jednotka, která je připojena do této propojovací sítě má následující rozhraní:




*Obrázek 17: funkční jednotka v propojovací síti*

Datové vstupy a výstupy slouží k připojení na propojovací síť a k masivní komunikaci mezi jednotlivými funkčními jednotkami a externí pamětí. Dále jednotka obsahuje konfigurační vstup a výstup, které lze použít ke komunikaci s funkční jednotkou. Bohužel neexistuje obsáhlejší dokumentace o funkci a práci s touto sítí, tak je její použití odvozeno na základě zdrojových kódů a konzultace s Ing. Martinem Žádníkem.

| | |
|---|---|
| `ConfigIn.`<br>`.Id`<br>`.Node`<br>`.Reg`<br>`.DataIn`<br>`.Write`<br>`.Read` | Parametr `Id` udává identifikátor funkční jednotky, pro kterou je tato konfigurační informace určena. Signály `Read` a `Write` udávají, zda se bude jednat o zápis nebo čtení informací z funkční jednotky. V případě zápisu se zapíše hodnota z `DataIn` na adresu `Reg` ve funkční jednotce. V případě čtení se bude číst z adresy `Reg` tolik slov, kolik udává hodnota v `DataIn`. |
| `ConfigOut` | Datový výstup určený pro čtení dat z funkční jednotky |

*Tabulka 7: konfigurační signály*

### 6.1.2 Funkční jednotka `TOP_FU`

Implementovaná funkční jednotka `TOP_FU` má rozhraní uvedené na obrázku 16. Tato jednotka pak obaluje implementované trénování SVM - `update_b`. Konfiguračním vstupem `ConfigIn` se do jednotky nahrávají data potřebná k trénování a zapisuje se do konfiguračního registru této jednotky. Přes výstup `ConfigOut` se čte výsledek trénování. Při čtení nebo zápisu dat do funkční jednotky se používá adresace signálem `ConfigIn.Reg`. Používané adresy jsou uvedeny v tabulce níže.

| adresa | r/w | funkce |
|---|---|---|
| 0 | r/w | Čtení nebo zápis do konfiguračního registru `TOP_FU`. Slouží k ovládání komponenty `update_b`. |
| 2 | w | Zápis hodnoty matice $\vec{Q}$ do `update_b` |
| 4 | w | Zápis hodnoty vektoru $\vec{y}$ do `update_b` |
| 6 | w | Zápis hodnoty $n$ do `update_b` |
| 8 | r | Čtení hodnot vektoru $\vec{\alpha}$ z `update_b` |
| 10 | r | Čtení hodnoty $b$ z `update_b` |

*Tabulka 8: adresní prostor*

Vkládání hodnot pro trénování a čtení natrénovaných údajů se tedy děje zápisem na konfigurační vstup jednotky `TOP_FU`. Na základě adresy zápisu/čtení `ConfigIn.Reg` jsou pomocí čítačů automaticky generovány adresy a další signály pro zápis a čtení hodnot do jednotky trénování SVM - `update_b`.

**Konfigurační registr (adresa 0):**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 15 | - | - | - | - | - | - | - | - | 8 |
| 7 | - | - | - | - | - | - | b1 | b0 | 0 |

- b0: lze zapisovat i číst. Zápisem hodnoty 1 se spustí trénování. Po dobu trénování je bit nastaven na hodnotě 1.
- b1: lze pouze číst. Hodnota tohoto bitu je nastavena na 1 po ukončení trénování. Spuštěním nového trénování se bit nastaví na 0.

### 6.1.3 Syntéza

Syntéza jednotky `TOP_FU` byla provedena v nástroji Xilinx ISE Design Suite 11. Bohužel tato verze softwaru již neumožňuje pracovat s čipy Virtex II, které jsou umístěny na modulu DX64 a starší verze nebyla momentálně k dispozici. Pro otestování parametrů syntézy byl tedy vybrán jiný čip, podobné velikosti, konkrétně Xilinx Spartan-3E 250.

Všechny kroky syntézy proběhly úspěšně a následující tabulky shrnují dosažené parametry.

**Komponenta TOP_FU:**

```
Device utilization summary:
---------------------------
Selected Device : 3s250evq100-5
Number of Slices: 586 out of 2448 23%
Number of Slice Flip Flops: 555 out of 4896 11%
Number of 4 input LUTs: 1094 out of 4896 22%
Number of IOs: 82
Number of BRAMs: 3 out of 12 25%
Number of MULT18X18SIOs: 1 out of 12 8%
Number of GCLKs: 1 out of 24 4%

Minimum period: 11.267ns (Maximum Frequency: 88.757MHz)
```

**TOP_FU uvnitř propojovací sítě CICB:**

```
Device utilization summary:
---------------------------
Selected Device : 3s250evq100-5
Number of Slices: 1103 out of 2448 45%
Number of Slice Flip Flops: 1132 out of 4896 23%
Number of 4 input LUTs: 1934 out of 4896 39%
Number used as logic: 1932
Number used as Shift registers: 2
Number of IOs: 59
Number of bonded IOBs: 48 out of 66 72%
IOB Flip Flops: 60
Number of BRAMs: 9 out of 12 75%
Number of MULT18X18SIOs: 1 out of 12 8%
Number of GCLKs: 4 out of 24 16%
Number of DCMs: 3 out of 4 75%

Minimum period: 11.267ns (Maximum Frequency: 88.757MHz)
```

# 7 Závěr

Cílem této diplomové práce bylo implementovat trénování klasifikátoru Support Vector Machine v hradlovém poli FPGA. Po prostudování dostupné literatury, zabývající se problematikou SVM byla vytvořena softwarová aplikace, realizující trénování klasifikátoru a jednoduché zobrazení výsledku. Poté byla vytvořena v jazyce VHDL komponenta, realizující zvolený algoritmus trénování. Správná funkce komponenty byla ověřena simulací programem ModelSim. Softwarová aplikace pak byla doplněna o možnost exportu a importu dat z/do souborů, které je možné použít k simulaci trénování programem ModelSim.

Dále byly prostudovány dostupné materiály k desce Uni1P, která na sobě může mít až čtyři moduly DX64 s FPGA čipem. Poté byl navržen způsob, jak zabudovat vytvořenou a otestovanou komponentu trénování SVM do FPGA čipu na desce DX64 a navržen způsob komunikace s komponentou. Následně byla provedena syntéza kompletní konfigurace FPGA a rozmístění na čipu. Obě tyto akce proběhly bez chyb. Syntéza a rozmístění ale proběhly na modernější typ FPGA čipu, jelikož dostupná verze syntetizátoru již nepodporovala FPGA čipy použité na modulu DX64. Testování v reálném systému zatím nebylo provedeno.

Další práce na tomto projektu by měla nejdříve otestovat správnou funkci navrženého systému v reálném zařízení. Následně by bylo vhodné implementovat paralelní algoritmus pro řešení násobení matice vektorem, aby se dosáhlo snížení doby jednoho trénovacího cyklu. Pokud by bylo třeba trénovat problémy velkým počtem trénovacích vektorů, je možnost využití externích dynamických pamětí, dostupných na platformě Uni1P/DX64.

# Literatura

[1] Burges Ch., *A Tutorilal on Support Vector Machines for Pattern Recognition*, 1998. Dokument dostupný na URL http://www.umiacs.umd.edu/~joseph/support-vector-machines4.pdf (prosinec 2008).

[2] Platt J., Microsoft Research. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Dokument dostupný na URL http://research.microsoft.com/pubs/69644/tr-98-14.pdf (prosinec 2008).

[3] Theodoridis S., Koutroumbas K., *Pattern Recognition, second edition*. San Diego, ACADEMIC PRESS 2003.

[4] Duda R., Hart P., Stork D., *Pattern Classification, second edition*. New York, WILEY-INTERSCIENCE 2003.

[5] Anguita D., Ridella S:, Rovetta S., *Circuital implementation of support vector machines*, 1998. Dokument dostupný na URL http://www.smartlab.dibe.unige.it/Files/publication/pdf/R009.PDF (duben 2009).

[6] Anguita D., Boni A., Ridella S., *Learning algorithm for nonlinear support vector machines for digital VLSI.* ELECTRONICS LETTERS, vol.35, 1999, 1349-1350 s.

[7] Anguita D., Boni A., Ridella S., *Digital VLSI algorithms and architectures for support vector machines*. International Journal of Neural Systems, Vol. 10, No. 3, 2000, 159-170 s.

[8] Anguita D., Boni A., Ridella S., *A Digital Architecture for Support Vector Machines: Theory, Algorithm and FPGA Implementation*. IEEE Transactions on Neural Networks, Vol. 14, No. 5, 2003, 993-1009 s.

[9] *Using Block RAM in Spartan-3 Generation FPGAs*. Dokument dostupný na URL http://www.xilinx.com/support/documentation/application_notes/xapp463.pdf (květen 2009).

[10] Hegar A., *Uni1P – Registry*. Camea s.r.o., 200?.

[11] Hegar A., *Kruhová propojovací síť*. Camea s.r.o., 200?.

[12] *UCI Machine Learning Repository.* Internetová stránka, dostupná na URL http://archive.ics.uci.edu/ml/index.html (duben 2009).

# Seznam příloh

Příloha 1: datový nosič DVD

Příloha 2: pseudokód SMO algoritmu

Příloha 3: pseudokód VLSI algoritmu

Příloha 4: informace o ovládání vizualizačního programu

# Příloha 2:

```
target = desired output vector
point = training point matrix

procedure takeStep(i1,i2)
  if (i1 == i2) return 0
  alph1 = Lagrange multiplier for i1
  y1 = target[i1]
  E1 = SVM output on point[i1] - y1 (check in error cache)
  s = y1*y2
  Compute L, H via equations (13) and (14)
  if (L == H)
    return 0
  k11 = kernel(point[i1],point[i1])
  k12 = kernel(point[i1],point[i2])
  k22 = kernel(point[i2],point[i2])
  eta = k11+k22-2*k12
  if (eta > 0)
  {
     a2 = alph2 + y2*(E1-E2)/eta
     if (a2 < L) a2 = L
     else if (a2 > H) a2 = H
  }
  else
  {
     Lobj = objective function at a2=L
     Hobj = objective function at a2=H
     if (Lobj < Hobj-eps)
        a2 = L
     else if (Lobj > Hobj+eps)
        a2 = H
     else
        a2 = alph2
  }
  if (|a2-alph2| < eps*(a2+alph2+eps))
     return 0
  a1 = alph1+s*(alph2-a2)
  Update threshold to reflect change in Lagrange multipliers
  Update weight vector to reflect change in a1 & a2, if SVM is linear
  Update error cache using new Lagrange multipliers
  Store a1 in the alpha array
  Store a2 in the alpha array
  return 1
endprocedure

procedure examineExample(i2)
  y2 = target[i2]
  alph2 = Lagrange multiplier for i2
  E2 = SVM output on point[i2] - y2 (check in error cache)
  r2 = E2*y2
  if ((r2 < -tol && alph2 < C) || (r2 > tol && alph2 > 0))
  {
    if (number of non-zero & non-C alpha > 1)
    {
      i1 = result of second choice heuristic (section 2.2)
      if takeStep(i1,i2)
        return 1
    }
```

```
      loop over all non-zero and non-C alpha, starting at a random point
      {
         i1 = identity of current alpha
         if takeStep(i1,i2)
           return 1
      }
      loop over all possible i1, starting at a random point
      {
         i1 = loop variable
         if (takeStep(i1,i2)
           return 1
      }
   }
   return 0
endprocedure

main routine:
    numChanged = 0;
    examineAll = 1;
    while (numChanged > 0 | examineAll)
    {
       numChanged = 0;
       if (examineAll)
          loop I over all training examples
             numChanged += examineExample(I)
       else
          loop I over examples where alpha is not 0 & not C
             numChanged += examineExample(I)
       if (examineAll == 1)
          examineAll = 0
       else if (numChanged == 0)
          examineAll = 1
   }
```

# Příloha 3:

| Algoritmus výpočtu nových koeficientů $\alpha_i$ | Algoritmus úpravy prahu $b$ |
|---|---|

ALGORITHM 1: DSVM WITH FIXED BIAS

| Step | Description |
|---|---|
| 1. | input: $Q, b, \boldsymbol{\alpha}^0$; |
| 2. | set $\tilde{\boldsymbol{r}} = (\boldsymbol{r} - \boldsymbol{y}b)$; $\eta = \frac{2}{m \max_{i,j}(q_{ij})}$ |
| 3. | set endrun=false; $k = 0$ |
| 4. | while not endrun do |
| 5. | $\boldsymbol{g} = \left(-Q\boldsymbol{\alpha}^k + \tilde{\boldsymbol{r}}\right)$ |
| 6. | $\boldsymbol{z}^{k+1} = \boldsymbol{\alpha}^k + \eta \boldsymbol{g}$ |
| 7. | $\alpha_i^{k+1} = \max\left(0, \min\left(z_i^{k+1}, C\right)\right), \forall i = 1, \ldots, m$ |
| 8. | if $\forall i \left(\alpha_i^{k+1} - \alpha_i^k\right) \leq \varepsilon$ then |
| 8. | endrun=true |
| 10. | else k=k+1 |
| 11. | enddo |
| 12. | output: $\boldsymbol{\alpha}' = \boldsymbol{\alpha}^{k+1}$ |

ALGORITHM 2: FIBS

| Step | Description |
|---|---|
| 1. | $b_{low} = b = -1$; $\boldsymbol{\alpha}^0 = 0$; $b_{up} = +1$; endFibs=false |
| 2. | $\boldsymbol{\alpha}' = \mathcal{A}_b(Q, b, \boldsymbol{\alpha}^0)$; $s = \operatorname{sgn}\left(y^T \boldsymbol{\alpha}'\right)$; $\boldsymbol{\alpha}^0 = \boldsymbol{\alpha}'$ |
| 3. | if $s < 0$ then |
| 4. | while $s < 0$ do |
| 5. | $b_{up} = b$; $b = 2b$; $b_{low} = 2b_{low}$ |
| 6. | $\boldsymbol{\alpha}' = \mathcal{A}_b(Q, b, \boldsymbol{\alpha}^0)$; $s = \operatorname{sgn}\left(y^T \boldsymbol{\alpha}'\right)$; $\boldsymbol{\alpha}^0 = \boldsymbol{\alpha}'$ |
| 7. | enddo |
| 8. | else if $s > 0$ then |
| 9. | $b = b_{up}$ |
| 10. | $\boldsymbol{\alpha}' = \mathcal{A}_b(Q, b, \boldsymbol{\alpha}^0)$; $s = \operatorname{sgn}\left(y^T \boldsymbol{\alpha}'\right)$; $\boldsymbol{\alpha}^0 = \boldsymbol{\alpha}'$ |
| 11. | if $s > 0$ then |
| 12. | while $s > 0$ do |
| 13. | $b_{low} = b$; $b = 2b$; $b_{up} = 2b_{up}$ |
| 14. | $\boldsymbol{\alpha}' = \mathcal{A}_b(Q, b, \boldsymbol{\alpha}^0)$; $s = \operatorname{sgn}\left(y^T \boldsymbol{\alpha}'\right)$; $\boldsymbol{\alpha}^0 = \boldsymbol{\alpha}'$ |
| 15. | enddo |
| 16. | else endFibs = true |
| 17. | while not endFibs do |
| 18. | $b = (b_{low} + b_{up})/2$ |
| 19. | $\boldsymbol{\alpha}' = \mathcal{A}_b(Q, b, \boldsymbol{\alpha}^0)$; $s = \operatorname{sgn}\left(y^T \boldsymbol{\alpha}'\right)$; $\boldsymbol{\alpha}^0 = \boldsymbol{\alpha}'$ |
| 20. | if $s = 0$ or $(b_{low} - b_{up}) \leq \varepsilon_b$ then endFibs=true |
| 21. | else if $s < 0$ then $b_{up} = b$ |
| 22. | else $b_{low} = b$ |
| 23. | enddo |

# Příloha 4:

| Přidávání bodů | Kliknutí levým nebo pravým tlačítkem do pracovní plochy |
|---|---|
| tl. *clear* | Smaže všechny trénovací body |
| tl. *load* | Načte uložené body ze souboru |
| tl. *save* | Uloží body do souboru. Každý bod je uložen na samostatném řádku textového souboru. Na řádku jsou za sebou hodnoty prvků vektoru příznaků a řádek je ukončen řetězcem `+1` nebo `-1`, dle třídy daného vektoru (bodu) |
| tl. *load def.* | Program načte body, uložené v souboru `default.svm`. Pokud tento soubor existuje při spuštění programu, je automaticky načten a zobrazen jeho obsah. |
| tl. *save daf.* | Program uloží body, uložené v souboru `default.svm` |
| tl. *train* | Program provede trénování na dané sadě trénovacích bodů. Způsob trénování je závislý na parametrech, nastavených v panelu *parameters* |
| *view trained* | Pokud je tento checkbox zakřížkován a SVM je natrénováno, nebo je načten výsledek trénování ze souboru (např. z výstupního souboru simulace v hw), je pracovní panel vybarven červenou nebo modrou barvou, která ukazuje prostory, kde je klasifikována červená a kde modrá třída. Zároveň jsou všechny nalezené podpůrné vektory ($\alpha_i \neq 0$) zakroužkovány. |
| tl. *export Q,y* | Ze zobrazených dat generuje soubory pro testování v simulátoru hw – soubor `Q_FILE` a `Y_FILE`. |
| tl. *import a* | Pro načtený a zobrazený problém importuje řešení – např. soubor `A_FILE`, který je výsledkem simulace v hw. Jakmile je soubor načten, jsou zvýrazněny podpůrné vektory a pokud je zakřížkováno *view trained*, jsou barevně odlišeny části plochy, klasifikované jednotlivými třídami. |
| tl. *export h.map* | Exportuje natrénovanou klasifikační funkci – matici čísel, jejíž hodnoty jsou reálná čísla, která vrací klasifikátor na daných pozicích v pracovním prostoru. Tento soubor je např. možné |

| | |
|---|---|
| | načíst do programu typu Matlab a zobrazit klasifikační funkci jako na obrázku 9. |
| tl. *test* | Testování klasifikátoru – pro natrénovaný problém lze načíst testovací data a provést vyhodnocení počtu špatně klasifikovaných dat. Vstupní soubor má stejný formát, jako soubor s uloženými body (vzniklý tlačítkem *save*) |
| panel *parameters* | V tomto panelu lze modifikovat různé parametry, které ovlivňují jak samotné trénování v softwaru, tak i export a import údajů do/z souboru (simulátoru) |