Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra kybernetiky

DIPLOMOVÁ PRÁCE

PLZEŇ, 2014 **MILAN KLÁŠTERKA**

P R O H L Á Š E N Í

Předkládám tímto k posouzení a obhajobě diplomovou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž úplný seznam je její součástí.

v Plzni dne 21. května 2014 .....

vlastnoruční podpis

Poděkování
Děkuji především vedoucímu bakalářské práce panu Ing. Janu Vaňkovi, Ph.D. za významnou pomoc při řešení této práce a za cenné rady ohledně optimalizace algoritmu v CUDA C.

Anotace

Úkolem této práce je implementace trénování a klasifikace SVM klasifikátorů na GPU. Je zde nastíněn úvod do programování v jazyce CUDA C, který je také použit pro realizaci klasifikátoru se čtyřmi běžně používanými typy kernel funkcí, tedy lineární, polynomiální, RBF a tangenciální. Dále je popsán moderní a velmi rychlý algoritmus trénování SVM klasifikátorů - SMO. Dalším úkolem je porovnání výsledků a časů běhu s veřejně dostupnými implementacemi SVM na GPU a dále ještě s balíkem LIBSVM `\cite{LIBSVM}`. Důležitou součástí této práce je optimalizace implementovaného klasifikátoru v programovacím jazyce CUDA C.

Klíčová slova: SVM trénování, SVM klasifikace, lineární kernel funkce, polynomiální kernel funkce, RBF kernel funkce, tangenciální kernel funkce, NVIDIA CUDA C, paralelní programování na GPU.

Annotation

The goal of this work is an implementation of training and prediction of SVM classifiers on GPU. There is a brief introduction to the CUDA C programming language, which is also used for the implementation of a classifier with four commonly used types of kernel functions, linear, polynomial, RBF and tangential. Next goal is description of modern and very fast algorithm for training of SVM classifiers - SMO. Another challenge is the comparison of run times and the classification results with freely available libraries for SVM classifiers using GPU and LIBSVM toolbox. An integral part of this work is to optimize the classifier implemented in the CUDA C programming language.

Key words: SVM training, SVM classification, linear kernel function, polynomial kernel function, Gaussian RBF, hyperbolic tangent function, NVIDIA CUDA C, parallel programming on GPU.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zkratek

CPU	Control Processing Unit
CUBLAS	CUDA Basic Linear Algebra Subroutines
CUDA	Compute Unified Device Architecture
CUFFT	CUDA Fast Fourier Transform
GPU	Graphics Processor Unit
KKT	Karushovy–Kuhnovy–Tuckerovy podmínky
OpenCL	Open Computing Language
PCG	Projected Conjugate Gradient
RAM	Random Access Memory
RBF	Radial Basis Function
SMO	Sequential Minimal Optimization
SVM	Support Vector Machine

1 Úvod

Vývoj informačních technologií jde neustále kupředu a je zapotřebí adaptovat používané metody a algoritmy způsobem, který zajistí velmi žádoucí využití potenciálu nově vzniklých platforem.

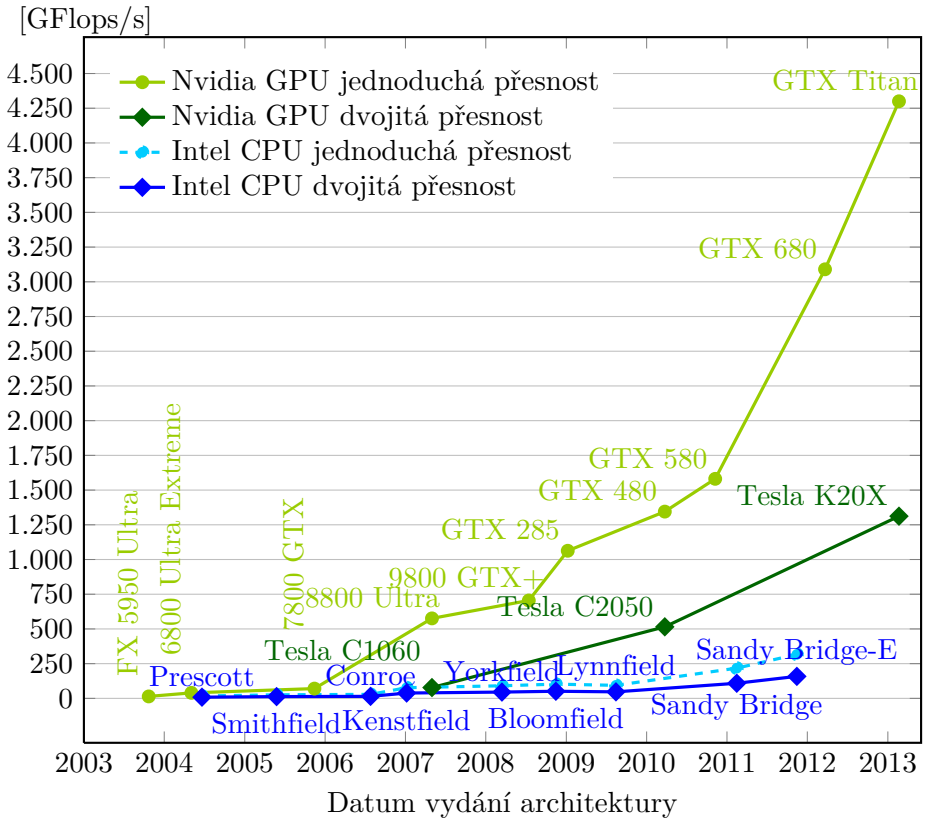
Převážná většina programů psaných v dnešní době je schopna využít pouze výpočetní sílu procesoru CPU a propustnost operační paměti RAM. Tyto prostředky nemusí být ve všech případech tou nejlepší volbou, např. lze-li metodu algoritmizovat způsobem, který umožňuje pracovat paralelně a to jak při výpočtech tak při práci s daty. Pro takové typy úloh je vhodnější použít výpočetní schopnosti GPU, která je zaměřena na paralelní zpracování a je větší propustností paměti v porovnání s RAM. Těchto charakteristik GPU lze využít u SVM klasifikátorů, kterým se věnuje tento dokument. Zkratka SVM je z anglického Support Vector Machine a jejím hlavním znakem je výběr podpůrných vektorů (angl. support vectors) z množiny trénovacích dat.

Původní verze trénování SVM klasifikátorů byla nejen velmi náročná z hlediska výpočetního výkonu, ale i paměti. Paměťová náročnost je řešena moderní metodou SMO [5], která umožňuje zredukovat množství uchovávaných dat na minimum. Díky tomu a důmyslně navržené heuristice dokáže SMO podstatným způsobem přispět k rychlosti.

Pro názornější zhodnocení výsledků této práce budou všechny implementované algoritmy porovnávány s jinými verzemi SVM klasifikátorů na stejném hardwaru, tedy na stejné grafické kartě, což bude obnášet důkladný rozbor daných programů. A dále bude předveden rozdíl v rychlosti a přesnosti (resp. korektnosti u trénování) implementací na GPU oproti implementaci na CPU.

2 GPGPU

Díky potřebě zpracovávat 3D grafiku s vysokým rozlišením v reálném čase se z GPU[1] staly vysoce paralelizované, vícevláknové a mnohajádrové procesory s ohromnou výpočetní silou a vysokou paměťovou propustností[2].
Grafické karty se postupně staly několikanásobně výkonnější než klasické procesory z hlediska výpočtů s čísly s plovoucí řádovou čárkou (tzv. `float`) a paměťové propustnosti, což znázorňují následující grafy.



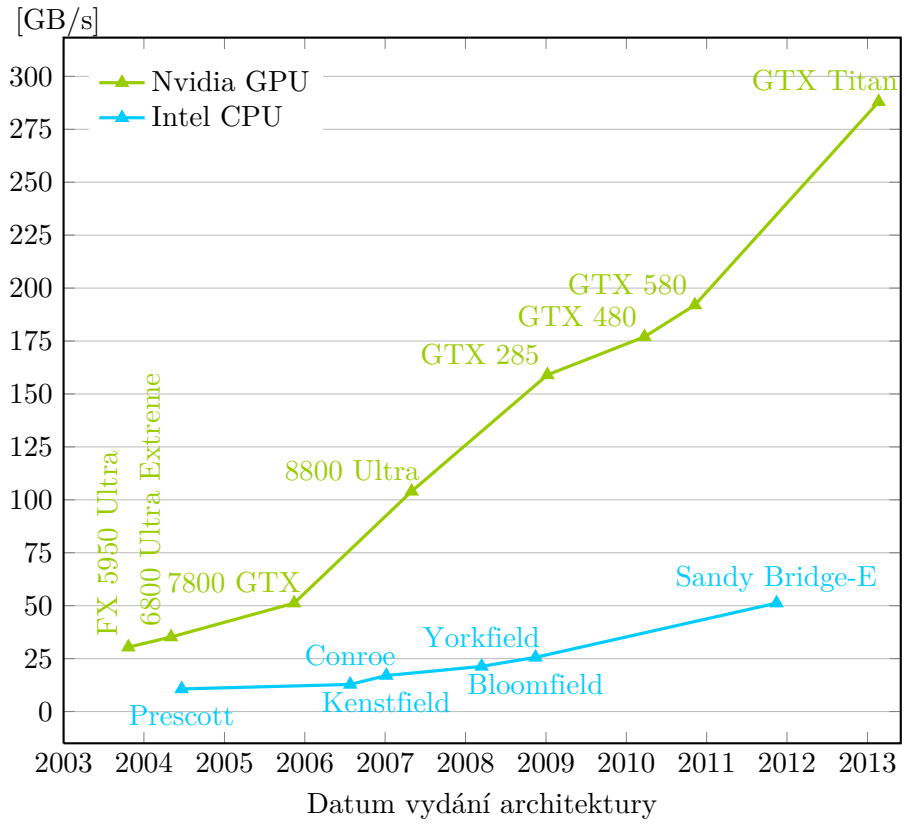
Obrázek 2.1: Vývoj počtu operací s čísly s plovoucí řádovou čárkou za sekundu pro CPU a GPU

Na obrázku 2.1 jsou zobrazeny průběhy výkonů grafických karet a procesorů z hlediska počtu aritmetických operací za sekundu. Čárkovaný průběh odpovídající dvojité přesnosti výpočtu za pomoci CPU je odvozen zdvojnásobením

[1] Graphics Processor Unit - grafická výpočetní jednotka; jádro grafické karty

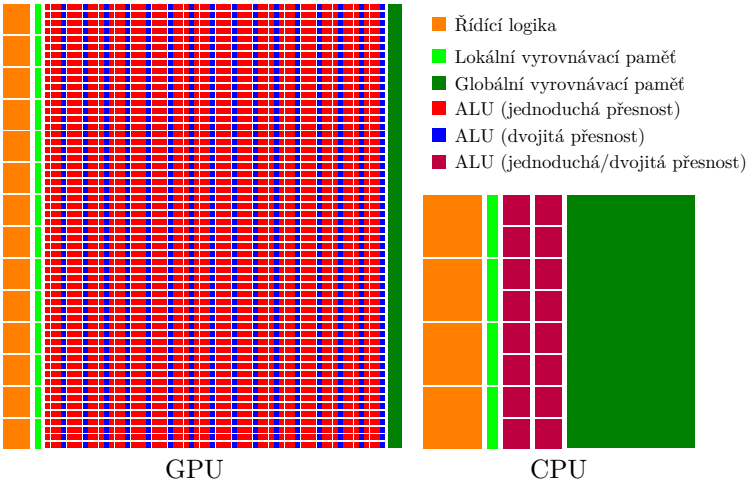
[2] Paměťová propustnost - počet Bytů, které lze uložit / načíst do paměti za jednotku času

hodnoty grafu pro jednoduchou přesnost CPU. To může být použito díky tomu, že při výpočtu ve dvojité přesnosti CPU zpracovává poloviční počet hodnot než při přesnosti dvojnásobné. To však platí pouze pro CPU, protože GPU využívá pro výpočty ve dvojnásobné přesnosti jiné výpočetní moduly, kterých je na čipu umístěno podstatně méně.



Obrázek 2.2: Vývoj paměťové propustnosti pro CPU a GPU

Důvodem výše zobrazených rozdílů mezi CPU a GPU je specializace GPU na výpočetně náročné vysoce paralelizované výpočty (grafické renderování), proto je více tranzistorů věnováno výpočetním jednotkám než na mezipamětem a řízení toku dat. To je schematicky znázorněno na následujícím obrázku.



Obrázek 2.3: Porovnání struktury CPU a GPU

Na obrázku výše jsou schematicky zobrazeny struktury CPU (Intel Sandy Bridge) a GPU (Nvidia Kepler). CPU je více zaměřeno na komplexní řízení toku dat za pomoci vyrovnávací paměti (která běžně zabírá polovinu velikosti čipu) a pro výpočty v dvojité i jednoduché přesnosti lze využít ty samé ALU[3]. Oproti tomu v GPU je snaha zjednodušit řízení toku dat na minimum, kdy se veškerá prováděná činnost rozloží mezi velké množství výpočetních jednotek (ALU), které jsou však rozděleny podle přesnosti, se kterou pracují (dvojitá nebo jednoduchá). Přičemž ALU pro výpočty v jednoduché přesnosti jsou využity jak jako výpočetní bloky, tak jako grafické jednotky („unified shader“), ale ALU pro výpočty ve dvojité přesnosti jsou využity jen při aritmeticko-logických výpočtech.

Grafické procesory jsou obzvláště dobře navrženy pro zpracování problémů, které lze vyjádřit jako *data-paralelní* výpočty (ten samý program se provádí na mnoha datech paralelně → menší nároky na řízení toku dat) s vysokým poměrem aritmetických operací k paměťovým operacím (→ čas potřebný pro přístup k paměti - čtení / zápis - je překryt výpočty namísto velké vyrovnávací paměti jako je tomu u CPU).

Při *data-paralelních* výpočtech jsou datové jednotky namapovány (přiřazeny) na paralelně zpracovávaná vlákna. U mnoha aplikací, které zpracovávají

[3]Arithmetic and Logic Unit = Aritmeticko-logická jednotka

velké množství dat, lze využít *data-paralelního* programovacího modelu pro zrychlení výpočtů. Příkladem je 3D renderování, kde jsou rozsáhlé množiny bodů a vrcholů, nebo zpracování obrazu a médií (kódování, dekódování, změna rozlišení, stereo vidění a rozpoznávání obrazů). Nicméně může být *data-paralelního* zpracování využito v polích mimo zpracování obrazu, od zpracování signálu nebo simulace fyziky po finanční nebo biologické výpočty.

2.1 OpenCL

OpenCL je standard umožňující jednotný přístup při programování na různorodým platformách (CPU, GPU, ...). Kdy při používání stejných funkcí, lze tentýž algoritmus napsaný v OpenCL spustit např. na CPU i na GPU. Aktuální verze obsahuje například knihovny pro základní maticové operace (ACML) a rychlý výpočet Fourierovy transformace.

2.2 CUDA C

Na konci roku 2006, Nvidia představila programovací model CUDA, který umožňuje využít výše popsané vlastnosti GPU pro numerické výpočty. Jako první mohli vývojáři použít CUDA jako knihovny programovacího jazyka C, avšak nyní je možné použít C++, Fortran nebo „wrapper“ do jiného programovacího jazyka (Java, Python). Dále CUDA obsahuje knihovny pro výpočet Fourierovy transformace (CUFFT) a pro podporu maticových operací (CUBLAS).

2.2.1 CUBLAS

CUBLAS je knihovna umožňující využívat CUDA pro běžné maticové a vektorové operace, bez nutnosti znát architekturu GPU a k ní se vážící optimalizace nutné pro požadované zrychlení výpočtů.

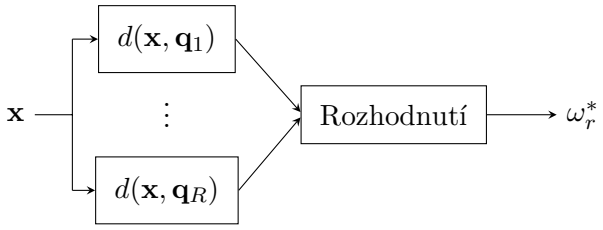
Knihovna CUBLAS používá, kvůli kompatibilitě s programovacím jazykem Fortran, sloupcovou reprezentaci matic. Oproti tomu programovací jazyk C používá řádkovou reprezentaci matic, proto je nutné dvou-dimenzionální data transponovat. Dále je nutné všechny vstupní i výstupní matice (vektory) předem alokovat (v případě vstupních dat vyplnit daty) v paměti GPU. Po těchto úpravách lze zavolat požadovanou CUBLAS funkci, a poté převést výsledná data z paměti GPU do RAM.

3 Klasifikace – rozpoznání obrazů

Rozpoznávání neboli klasifikace je považováno za základní projev člověka. Člověk je schopen rozpoznávat položky jak abstraktní (pojmy), tak i konkrétní (předměty a jevy), ty jsou klasifikovány s využitím smyslů (vizuální, zvukové a hmatové rozpoznávání).

Klasifikace objektu znamená, že je přiřazen do třídy, k níž má nějaký vztah. Avšak k tomu, aby klasifikace byla možná, musí být vybrán vhodný způsob reprezentace daného předmětu = **obraz**. Klasifikátor tedy nerozpoznává předměty samotné, ale jejich obrazy, ty jsou reprezentovány vektorem příznaků[4].

Správný popis klasifikovaného objektu však není jediným problémem. Dále je potřeba vyřešit samotnou klasifikaci, tedy práci klasifikátoru. Ten musí být nejprve nějakým způsobem naučen, jakým způsobem má zadávané obrazy třídit. Rozhodnutí, do které třídy obraz patří, je prováděno na základě hodnoty tzv. diskriminačních funkcí $d(\mathbf{x}, \mathbf{q}_r)$, kde $\mathbf{x}$ je klasifikovaný obraz a $\mathbf{q}_r$ je je vektor parametrů, který pro klasifikátor představuje vlastnosti r-té třídy.
Klasifikátor poté pracuje následovně:



Obrázek 3.1: Blokové schéma klasifikátoru

Než je možné začít obrazy klasifikovat, je nutné specifikovat diskriminační funkci $d(\mathbf{x}, \mathbf{q}_r)$, parametry $\mathbf{q}_r$ všech tříd a rozhodnutí, které provede samotné zařazení do třídy. Proto se činnost klasifikátoru dělí na dvě fáze:

1. **Fáze nastavení**
   Klasifikátoru se postupně předkládají **trénovací obrazy**, často spolu

[4]Příznak = numericky vyjádřené vlastnosti objektu.

s informací učitele o třídě, do níž daný obraz patří.
Daný algoritmus učení poté pro každou třídu určí parametr $\mathbf{q}_r$, reprezentující buď odhadovaný tvar rozložení obrazů v třídě nebo rozdělující nadplochu.

2. **Fáze klasifikace**
Jakmile je klasifikátor naučen, už není potřeba mu předávat informace o třídě, do níž patří, tudíž je schopen správně klasifikovat vstupní obraz do jedné ze tříd, jejichž seznam si vytvořil při učení.

Rozhodnutí a diskriminační funkce jsou dány algoritmem, který je zvolen pro klasifikaci.

3.1 Lineární klasifikátory

Lineární klasifikátory jsou často využívány, protože používají lineární rozdělující nadplochu, jejíž trénování i klasifikace podle ní je snadno realizovatelné.

Lineární klasifikátory rozhodují o zařazení vstupního obrazu do příslušné třídy podle lineární diskriminační funkce

$$g(\mathbf{x}) = \mathbf{q} \circ \mathbf{x}, \tag{3.1}$$

kde $\mathbf{q} = [q_0, \ldots, q_n]$ je vektor parametrů získaný při učení (reprezentuje kolmici s rozdělující nadrovině; q_0), $\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$ je testovaný obraz a operátor $\circ$ znamená skalární součin vektorů definovaný následovně

$$\mathbf{x} \circ \mathbf{y} = \sum_{i=0}^{n} x_i \cdot y_i, \tag{3.2}$$

kde n je dimenze obrazového prostoru.
Přičemž při klasifikaci do dvou tříd (dichotomii) se zařazení do příslušné třídy provádí takto:

$$\omega = \operatorname{sign}\left[g(\mathbf{x})\right], \tag{3.3}$$

kde ω je třída, do které klasifikátor vstupní obraz zařadí a může nabývat hodnoty z množiny $\{-1; 1\}$ (při $\omega = 0$ klasifikátor není schopen rozhodnout, avšak tato hodnota je velmi nepravděpodobná a v praxi se tento případ řeší přiřazením daného obrazu do libovolné ze tříd).

3.2 Nelineární klasifikátory

Používají se dva principy nelineárních klasifikátorů:

1. Nelineární diskriminační funkce.
   Používá se v případech, kdy lze analýzou problému zjistit vhodný tvar nebo je pro daný případ přímo známa vhodná nelineární diskriminační funkce.

2. Nelineární transformace Φ vstupního obrazového prostoru H_m (dimenze m) do prostoru H_p (dimenze $p > m$), kde jsou třídy lineárně separovatelné, a tudíž lze použít lineární klasifikátor.

$$\Phi : H_m \to H_p \tag{3.4}$$

   Transformační funkce $\Phi(\mathbf{x})$ pro obraz $\mathbf{x}$ má následující tvar

$$\Phi(\mathbf{x}) = \left[\Phi_1(\mathbf{x}), \Phi_2(\mathbf{x}), \ldots, \Phi_p(\mathbf{x})\right], \tag{3.5}$$

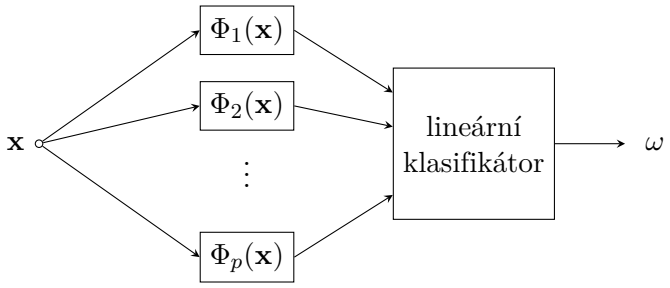
   kde $\Phi_1(\mathbf{x}), \Phi_2(\mathbf{x}), \ldots, \Phi_p(\mathbf{x})$ jsou jednotlivé transformační funkce nového prostoru H_p.

   Diskriminační funkce pro r-tou třídu:

$$g_r(\mathbf{x}) = q_{r,1}\Phi_1(\mathbf{x}) + q_{r,2}\Phi_2(\mathbf{x}) + \ldots + q_{r,p}\Phi_p(\mathbf{x}), \tag{3.6}$$

   kde $q_{r,1}, q_{r,2}, \ldots, q_{r,p}$ jsou parametry klasifikátoru získané při učení.

   Funkce klasifikátoru je znázorněna na následujícím obrázku.



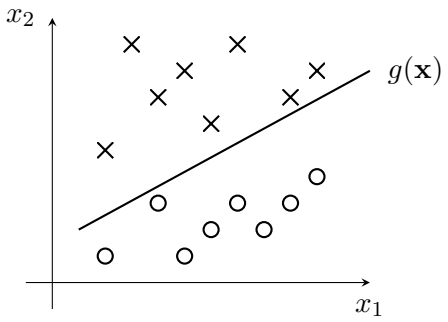
Obrázek 3.2: Blokové schéma nelineárního klasifikátoru

Ovšem při transformaci do vyšší dimenze se s disperzí mezi třídami, kterou se tímto způsobem snažíme zvětšit, zvýší i disperze mezi obrazy v jednotlivých třídách. Tím se může zvýšit i případný šum naměřených hodnot obrazů. Tento jev je nazýván „kletba dimenzionality“ (z anglického *curse of dimensionality*).

Ovšem při transformaci do vyšší dimenze, kdy se třídy stávají lineárně separovatelné, se od sebe jednotlivé obrazy vzdalují. Což znamená pokles vypovídací hodnoty jednotlivých obrazů, tj. relativní pokrytí prostoru dané třídy (Hughesův jev [12]).
S tím jsou spojeny i technické problémy s uchováváním vektorů vysoké dimenze a zvýšení výpočetních nároků na procesor.

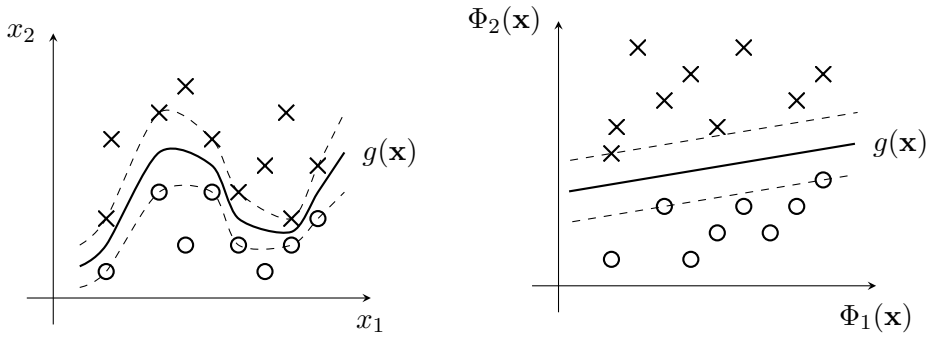
4 Support Vector Machine – SVM

Klasifikátory typu SVM vycházejí z lineárního klasifikátoru, který je u nich použit při klasifikaci, ale rozdíl je v principu trénování.
Na diskriminační funkci lineárního klasifikátoru je kladen jen požadavek, aby bezchybně rozdělovala obě třídy.



Obrázek 4.1: Lineární diskriminační funkce

Oproti tomu musí být diskriminační funkce SVM klasifikátoru taková, aby maximalizovala minimální vzdálenost od nejbližších obrazů (robustnost – margin):
Příklad pro optimální nastavení diskriminační funkce $g(\mathbf{x})$ při dimenzi obrazového prostoru $n = 2$ a počtu tříd $R = 2$:



(a) Nelineární diskriminační funkce v netransformovaném prostoru

(b) Lineární diskriminační funkce v transformovaném prostoru

Obrázek 4.2: Optimální nastavení lineární diskriminační funkce SVM klasifikátoru v obrazovém prostoru H_p

Čárkované přímky ohraničují oblast, která má při maximalizaci vzdáleností všech obrazů od diskriminační funkce $g(\mathbf{x})$ největší možný *margin*.

Obrazy ležící v nejbližším okolí rozdělující nadroviny (na obrázku 4.2b vyznačeno čárkovaně) jsou u SVM klasifikátorů označovány jako podpůrné vektory („support vectors“ odtud *Support vector machine*) a jsou pro učení tak důležité, že při odebrání všech ostatních obrazů by diskriminační funkce měla tentýž tvar. Natrénovaný model SVM obsahuje již pouze podpůrné vektory, ty jsou použity při následné klasifikaci.

Primární formulace SVM problému

$$\frac{1}{2} \cdot \|\mathbf{q}\| + C \sum_{i=1}^{l} \xi_i \longrightarrow \min_{\mathbf{q},\varrho,\xi}, \tag{4.1}$$

při splnění podmínek

$$y_i \cdot \left[\mathbf{q}^{\mathrm{T}} \cdot \Phi(\mathbf{x}_i) + \varrho\right] \geq 1 - \xi_i; \quad \xi_i \geq 0, \tag{4.2}$$

kde $\mathbf{q}$ je vektor parametrů klasifikátoru (vektor kolmý na rozdělující nadplochu; zajišťuje robustnost), C je horní mez pro hodnoty Lagrangeových multiplikátorů $\boldsymbol{\alpha}$ (dolní mez je vždy 0), ξ_i je chyba klasifikace i-tého trénovacího vektoru, y_i je i-tá informace o správné klasifikaci (informace od učitele), ϱ je práh, $\mathbf{x}_i$ je i-tý vstupní obraz a $\Phi(\mathbf{x})$ je nelineární transformační funkce (z principu tato funkce může být lineární, avšak nepoužívá se).

Pro SVM klasifikátory se s výhodou používá druhého přístupu (tzv. duální formulace) ve tvaru

$$L_D = \sum_i \alpha'_i - \frac{1}{2} \sum_{i,j} \alpha'_i \cdot \alpha'_j \cdot y_i \cdot y_j \cdot \Phi(\mathbf{x}_i) \circ \Phi(\mathbf{x}_j) \rightarrow \max_{\alpha_i,\alpha_j} \tag{4.3}$$

přičemž platí

$$\begin{gathered} 0 \leq \alpha'_i \leq C \\ \sum_i \alpha'_i \cdot y_i = 0, \end{gathered} \tag{4.4}$$

kde α'_i je i-tý Lagrangeův koeficient nabývající hodnot v intervalu $\langle 0; C \rangle$, y_i je informace od učitele pro zařazení do správné třídy pro i-tý obraz a $\Phi(\mathbf{x}_i)$

je i-tý obraz transformovaný do prostoru H_p.
Dále je předpokládáno, že margin je normalizován na hodnotu 1, jeho váha je poté nepřímo určena parametrem C, který navíc ovlivňuje úspěšnost klasifikace trénovacích dat.

Řešením rovnice je

$$\mathbf{q} = \sum_{i=1}^{N_s} \alpha_i' \cdot y_i \cdot \Phi(\mathbf{x}_i), \tag{4.5}$$

kde N_s je počet podpůrných vektorů.
Při algoritmizaci je vhodné použít nahrazení:

$$\alpha_i = \alpha_i' \cdot y_i, \tag{4.6}$$

za předpokladu, že „označení“ tříd y_i nabývá pouze hodnot -1 nebo 1, lze takto jednoduše sloučit 2 hodnoty do jedné.

Diskriminační funkce SVM klasifikátoru bude mít následující tvar

$$g(\mathbf{x}) = \sum_{i=1}^{N_s} \alpha_i \cdot \Phi(\mathbf{v}_i) \circ \Phi(\mathbf{x}) + \varrho, \tag{4.7}$$

kde $\mathbf{v}_i$ je podpůrný vektor a $\mathbf{x}$ je testovací vektor.

Vhodnější je však použít

$$g(\mathbf{x}) = \sum_{i=1}^{N_s} \alpha_i \cdot \mathrm{k}(\mathbf{v}_i, \mathbf{x}) + \varrho, \tag{4.8}$$

kde $\mathrm{k}(\mathbf{v}_i, \mathbf{x})$ je tzv. **kernel funkce** reprezentující skalární součin (viz vztah 3.2) mezi podpůrným vektorem $\mathbf{v}_i$ a testovacím vektorem $\mathbf{x}$.

Díky tomu, že není nutné používat nelineární funkce pro transformaci vstupních obrazů ($\Phi(\mathbf{x})$), jsou odstraněny problémy spojené s „kletbou dimenzionality“.

4.1 Fáze trénování

Fáze učení spočívá v natrénování parametrů regulátoru (vektory $\mathbf{q}_r$) pro všechny třídy pro zvolenou transformaci $\Phi(\mathbf{x})$. Na vstup klasifikátoru při tom přicházejí dvojice $(\mathbf{x}_i, y_i), i = 1, \ldots, l$, kde $x_i \in \mathbb{R}^n$ (n je dimenze obrazového prostoru) je vstupní obraz a $y_i \in \{-1, 1\}$ je informace od učitele neboli správná klasifikace.

Při trénování se získávají parametry α_i, které nabývají hodnoty > 0 pouze pro ty obrazy, které jsou nejblíže rozdělující nadploše. Tyto obrazy se po natrénování stávají podpůrnými vektory. Všechny obrazy mající po učení $\alpha_i \in \langle -\varepsilon; \varepsilon \rangle$ (ε je volitelná mez určující jak významný z hlediska marginu musí daný vektor být, aby byl do modelu přidán) se z modelu vypouští, protože by při klasifikaci nezměnily výsledné rozhodnutí.

Pro natrénování parametrů klasifikátoru se používají metody kvadratického programování (dále jen QP).
Protože QP má pro SVM speciální tvar, existuje řada metod určených přímo pro trénování SVM:

- Chunking [4],
- SMO – extrémní případ Chunking [5],
- Interior Point metoda [7],
- další.

Kde Chunking a SMO jsou používány výhradně pro trénování nelineárních verzí SVM. Metoda Interior Point se používá buď pro lineární klasifikátory nebo pro nelineární s malým počtem trénovacích obrazů (matice $\mathbf{Q}$ - obě její dimenze jsou druhou mocninou počtu vektorů - se musí vejít celá do paměti , kde bude zpracovávána).
V této práci bude použit algoritmus SMO, který je nejmodernější, nejobecnější a používá ho většina současných implementací pro nelineární kernel funkce. Bude mu věnována samostatná kapitola kapitola.

Pozn.: Balík LIBSVM [9], použitý pro porovnání výsledků klasifikace, využívá metodu SMO.

4.2 Fáze klasifikace

Klasifikace je prováděna tak, že se za pomoci SVM kernel funkcí porovnávají všechny podpůrné vektory (tedy vektory $\mathbf{x_i}$, u nichž byl natrénován parametr $\alpha_i \neq 0$) se zadaným vektorem, který má být zařazen do jedné ze tříd. Zařazení do příslušné třídy se provádí pomocí funkce sign(.) z diskriminační funkce

$$g(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i \cdot \mathrm{k}(\mathbf{v}_i, \mathbf{x}) - \varrho, \tag{4.9}$$

kde α_i je i-tý parametr udávající příslušnost k jedné ze dvou funkcí, $\mathbf{v}_i$ je i-tý podpůrný vektor, $\mathbf{x}$ je klasifikovaný vektor, $\mathrm{k}(\cdot)$ je použitá kernel funkce, ϱ je práh, ω je rozhodnutí klasifikátoru a N je počet všech podpůrných vektorů.

SVM kernel funkce používané v balíku LIBSVM, pro podpůrný vektor v_i a klasifikovaný vektor x o stejné dimenzi m, jsou popsány v následující kapitole, kde je podrobněji rozebrána jejich funkčnost.

5 Sequential Minimal Optimization – SMO

Základní metoda řešení trénování SVM klasifikátorů (viz kapitola 4) je pomalá, obzvláště pro rozsáhlé problémy, a trénovací algoritmus je složitý a obtížně implementovatelný. Proto byl vytvořen algoritmus, který je jednoduše implementovatelný, často rychlejší a s většími možnostmi nastavení než standardní „chunking“ algoritmus, který používá PGG (projected conjugate gradient).

Sequantial Minimal Optimization (dále jen SMO), jejímž autorem je Platt (viz [5]), používá ve vnitřní smyčce analytický krok QP (jeden krok kvadratického programování), oproti předchozím trénovacím algoritmům, využívajícím QP (kvadratické programování). Protože je většina času při aplikování SMO algoritmu strávena výpočtem rozhodovací funkce (a nikoliv prováděním smyčky kvadratického programování), může zpracovávat data s velkým množstvím nulových prvků (tzv. „řídká“ data).

Každý krok SMO algoritmu řeší nejmenší možný optimalizační problém. Pro standardní problém QP v SVM je tím nejmenším možným optimalizačním problémem výpočet dvou Lagrangeových multiplikátorů, protože Lagrangeovy multiplikátory musí řešit omezení na lineární rovnost. V každém kroku jsou voleny dva Lagrangeovy multiplikátory, které se společně optimalizují, nalezením optimálních hodnot pro tyto multiplikátory, a odpovídajícím způsobem se aktualizuje SVM model.

Výhoda SMO spočívá v tom, že řešení pro dva Lagrangeovy multiplikátory může být provedeno analyticky, což umožní vyhnout se výpočtu celé vnitřní smyčky QP.
Navíc, SMO nepotřebuje další paměť pro uchovávání výsledků QP (pokud pomineme minoritní množství paměti potřebné pro uchování matic o rozměru 2×2).

Algoritmus SMO se skládá ze tří komponent:

- analytické řešení dvou Lagrangeových multiplikátorů,
- heuristická volba volby multiplikátorů, které mají být optimalizovány,

- metoda pro výpočet prahu b.

5.1 Řešení pro Lagrangeovy multiplikátory

Nejprve je nutné spočítat meze, které budou následně aplikovány na nově vypočtenou hodnotu multiplikátoru α_j na jehož základě bude určena hodnota druhého multiplikátoru α_i, přičemž indexy i a j byly vybrány v předchozí iteraci algoritmu (libovolně zvoleny při inicializaci).

Pokud jsou třídy obou zvolených trénovacích vektorů různé, tedy $y_i \neq y_j$, pak budou pro výpočet horní (H) a dolní (L) meze použity vztahy:

$$\begin{aligned} L &= \max(0, \alpha_j^k - \alpha_i^k), \\ H &= \min(C, C + \alpha_j^k - \alpha_i^k), \end{aligned} \tag{5.1}$$

kde C je konstanta učení klasifikátoru, jejíž hodnota určuje rychlost a stabilitu konvergence (vyšší hodnota znamená vyšší rychlost za cenu nižší stability a naopak).
Pokud však $y_i = y_j$, bude pro meze platit následující:

$$\begin{aligned} L &= \max(0, \alpha_j^k + \alpha_i^k - C), \\ H &= \min(C, \alpha_j^k + \alpha_i^k). \end{aligned} \tag{5.2}$$

Druhá derivace diskriminační funkce může být vyjádřena jako:

$$\eta = 2 \cdot \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) - \mathrm{k}(\mathbf{x}_i, \mathbf{x}_i) - \mathrm{k}(\mathbf{x}_j, \mathbf{x}_j) \tag{5.3}$$

Dále je zapotřebí spočítat umístění omezeného maxima, na které nebyly aplikovány výše spočtené meze, za předpokladu, že se mohou změnit pouze dva Lagrangeovy multiplikátory.
Za normálních okolností je hodnota $\eta < 0$. V tomto případě lze novou (neomezenou) hodnotu vyčíslit takto:

$$\widehat{\alpha}_j^{k+1} = \alpha_j^k - \frac{y_j \cdot (E_i - E_j)}{\eta}, \tag{5.4}$$

kde $E_i = f^k(\mathbf{x}) - y_i$ je chyba pro i-tý trénovací vektor, obdobně pro E_j.
Omezené maximum určíme jako hodnotu z intervalu $\langle L; H \rangle$, která má k

neomezenému maximu nejblíže, tedy:

$$\alpha_j^{k+1} = \begin{cases} H & \text{pokud } \widehat{\alpha}_j^{k+1} > H, \\ L & \text{pokud } \widehat{\alpha}_j^{k+1} < L, \\ \widehat{\alpha}_j^{k+1} & \text{jinak.} \end{cases} \tag{5.5}$$

Hodnotu druhého Lagrangeova multiplikátoru určíme jednoduše pomocí následujícího vzorce:

$$\alpha_i^{k+1} = \alpha_i^k + y_i \cdot y_j \cdot (\alpha_j^k - \alpha_j^{k+1}). \tag{5.6}$$

V extrémním případě může hodnota η nabývat nezáporných hodnot. Nulová hodnota η vznikne v případě, kdy se alespoň jeden trénovací vektor vyskytuje v trénovací množině více než jednou. Algoritmus SMO pracuje správně i tehdy, kdy η je nezáporná, přičemž v tomto případě musí být diskriminační funkce vyčíslena na každém konci úsečky. Přičemž je nutné vyčíslit pouze ty výrazy, které závisí na α_j. SMO mění hodnoty Lagrangeových multiplikátorů tak, aby diskriminační funkce byla maximální. Pokud jsou hodnoty diskriminační funkce na obou koncích úsečky shodné a kernel funkce splňuje Mercerovu podmínku[5], nemůže maximalizace pokračovat. Tento případ je popsán dále v textu.

5.2 Heuristika pro výběr multiplikátorů

Algoritmus SMO optimalizuje v každém kroku dva Lagrangeovy multiplikátory, přičemž jeden z nich musel v předchozím kroku porušit KKT podmínky[6]. Proto SMO vždy pozmění dva Lagrangeovy multiplikátory tak, aby se posunuly po diskriminační funkci směrem nahoru. Díky tomu se hodnota diskriminační funkce zvýší v každém kroku a tím je zajištěna asymptotická konvergence. Za účelem zvýšení rychlosti konvergence používá SMO heuristiky pro výběr multiplikátorů, které mají být v dalším kroku optimalizovány.

Existují právě dvě heuristiky, přičemž jedna je použita pro výběr prvního Lagrangeova multiplikátoru a druhá pro výběr druhého. První heuristika je prováděna ve vnější smyčce SMO algoritmu, kdy se prochází všechny

[5] Mercerova podmínka pro funkci $\mathrm{k}(x, y)$ je splněna pokud $\iint \mathrm{k}(x, y) \cdot g(x) \cdot g(y) \mathrm{d}x\mathrm{d}y \leq 0$

[6] Karushovy–Kuhnovy–Tuckerovy podmínky jsou nutné podmínky pro zaručení optimality při řešení úloh nelineárního programování.

trénovací vektory za účelem nalezení těch, které porušují KKT podmínky. Pokud trénovací vektor poruší tyto podmínky je kandidátem na optimalizaci. Po volbě prvního multiplikátoru se použije druhá heuristika, která vybere druhý multiplikátor a následně jsou oba multiplikátory společně optimalizovány. Následně je SVM model aktualizován na základě optimalizovaných multiplikátorů a vnější smyčka pokračuje hledáním dalšího trénovacího vektoru porušujícího KKT.

Kvůli zrychlení trénování se ve vnější smyčce ne vždy prochází všechny trénovací vektory. Po jednom průchodu celou trénovací množinou se v dalších iteracích prochází pouze ty trénovací vzorky, jejichž Lagrangeovy multiplikátory nejsou rovny 0 ani C, tzn. neomezené vzorky. Ve všech následujících iteracích se projdou pouze tyto neomezené vzorky, které, pokud porušují KKT podmínky, vyvolají optimalizaci příslušných multiplikátorů, což pokračuje do té doby než není možné optimalizovat žádný ze zbylých vzorků ve vybrané podmnožině. Poté se opět jednou projdou všechny trénovací vektory. Vnější smyčka poté pracuje střídavě s celou trénovací množinou a nebo její podmnožinou, dokud všechny trénovací vzorky nesplní KKT podmínky (s předem danou povolenou odchylkou ε). V tomto momentě algoritmus končí.

První heuristika využívá výpočetní čas pro ty vzorky, které s velkou pravděpodobností poruší KKT podmínky - podmnožina neomezených vzorků. S postupem SMO algoritmu, Lagrangeovy multiplikátory, které jsou na mezích (0 nebo C) na těchto hodnotách zůstanou, zatímco multiplikátory, které jsou v intervalu $(0; C)$ změní svou hodnotu kvůli optimalizaci jiných multiplikátorů. SMO poté prochází podmnožinu neomezených vzorků dokud v ní existuje alespoň jeden prvek, který není optimální. Následně SMO pokračuje hledáním takových trénovacích vektorů v celé trénovací množině, které po optimalizaci neomezené podmnožiny nesplňují KKT podmínky.

Algoritmus SMO při každé kontrole KKT podmínek dovoluje odchylku od správného výsledku o velikosti hodnoty parametru ε, který je typicky volen v rozmezí 10^{-3} a 10^{-2}. Rozpoznávací systémy využívající SVM klasifikátoru běžně nevyžadují vysokou přesnost při kontrole splnění KKT podmínek. Při nastavení vysoké přesnosti (nízké hodnotě ε) nebude SMO algoritmus rychle konvergovat.

Po výběru prvního multiplikátoru, volí SMO druhý multiplikátor takovým způsobem, aby byla změna kriteriální funkce provedená společnou optimalizací obou multiplikátorů maximální. Vyčíslení kernel funkce je časově náročné, proto SMO nahrazuje čitatel rovnice 5.4 výrazem $|E_i - E_j|$. Algoritmus SMO uchovává hodnotu chybové proměnné E pro každý neomezený vektor z trénovací množiny, a poté vybere takový vzorek, který maximalizuje změnu chyby kriteriální funkce. Pokud je hodnota $E_i > 0$, tak je vybrán vzorek s minimální hodnotou E_j a pro $E_i \geq 0$ je vybrán vektor s maximální chybou E_j.

V některých, velmi neobvyklých, případech, nemůže SMO dosáhnout požadovaného snížení kriteriální funkce pomocí druhé heuristiky tak, jak je popsáno výše. Například, pokud první i druhá heuristika zvolí totožný trénovací vektor, což způsobí, že diskriminační funkce je konstantní v úseku optimalizace. K eliminaci tohoto problému používá SMO hierarchii pro volbu vzorku pomocí druhé heuristiky, kdy se hledá taková dvojice Lagrangeových multiplikátorů, která zajistí pokles kriteriální funkce. Hierarchie druhé heuristiky je složena z následujících kroků:

A) pokud druhá heuristika nenalezne takový trénovací vektor, který nezajistí konvergenci algoritmu, tak SMO prochází neomezené vzorky a hledá takový, který tuto podmínku splní;

B) pokud ani poté není nalezen žádný trénovací vektor zajišťující konvergenci, tak SMO při hledání vhodného vektoru prochází celou trénovací množinu.

Oba průchody, neomezenými vzorky A i celou trénovací množinou B, začínají na náhodně zvoleném vzorku, aby se předešlo řešením pouze vzorků na začátku trénovací množiny. V extrémně vzácných případech může nastat situace, kdy není ani po hierarchii druhé heuristiky nalezena vhodná dvojice vektorů pro optimalizaci. V tomto případě se vzorek zvolený v první heuristice přeskočí a algoritmus pokračuje dál.

5.3 Práh a mezipaměť chyb

Řešením rovnic kvadratického programování (4.3) pro určení hodnot vektoru $\boldsymbol{\alpha}$ nezískáme hodnotu prahu b pro SVM model, hodnota prahu musí být

vypočtena samostatně. Po každém kroku je hodnota b přepočtena tak, aby byly splněny KKT podmínky pro oba optimalizované vektory ($\mathbf{x}_i$ a $\mathbf{x}_j$). Následující práh b_1^{k+1} je platný, pokud nová hodnota α_i není na mezích, protože to na výstupu SVM klasifikátoru vyvolá hodnotu y_i při vstupu $\mathbf{x}_i$:

$$b_1^{k+1} = E_i + y_i \cdot (\alpha_i^{k+1} - \alpha_i^k) \cdot \mathrm{k}(\mathbf{x}_i, \mathbf{x}_i) + y_j \cdot (\alpha_j^{k+1} - \alpha_j^k) \cdot \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) + b^k. \quad (5.7)$$

Následující práh b_2^{k+1} je platný, pokud nová hodnota α_j není na mezích, protože to na výstupu SVM klasifikátoru vyvolá hodnotu y_j při vstupu $\mathbf{x}_j$:

$$b_2^{k+1} = E_i + y_i \cdot (\alpha_i^{k+1} - \alpha_i^k) \cdot \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) + y_j \cdot (\alpha_j^{k+1} - \alpha_j^k) \cdot \mathrm{k}(\mathbf{x}_j, \mathbf{x}_j) + b^k. \quad (5.8)$$

V případě, že jsou obě nové hodnoty prahu platné, mají stejnou hodnotu. Pokud jsou oba Lagrangeovy multiplikátory na mezích a $L \neq H$, pak všechny hodnoty v intervalu $\langle b_1^{k+1}; b_2^{k+1} \rangle$ zaručí splnění KKT podmínek. V tomto případě SMO volí práh jako střední hodnotu obou vypočtených.

Jak bylo probráno v sekci 5.2, chybová hodnota E je uchována v mezipaměti chyb pro každý Lagrangeův multiplikátor, který není roven nule ani C. Pokud je Lagrangeův multiplikátor neomezený a podílí se na optimalizaci, jeho chybová hodnota je nastavena na nulu. Po provedení optimalizace se všechny uložené chybové hodnoty neomezených multiplikátorů α_l, které nebyly využity při optimalizaci, aktualizují podle následujícího vzorce:

$$\begin{aligned} E_l^{k+1} = E_l^k &+ y_i \cdot (\alpha_i^{k+1} - \alpha_i^k) \cdot \mathrm{k}(\mathbf{x}_i, \mathbf{x}_l) \\ &+ y_j \cdot (\alpha_j^{k+1} - \alpha_j^k) \cdot \mathrm{k}(\mathbf{x}_j, \mathbf{x}_l) + b^k - b^{k+1}, \end{aligned} \quad (5.9)$$

kde k je předchozí krok algoritmu.
Pokud je chybová hodnota E vyžadována SMO algoritmem, bude se tato hodnota hledat v mezipaměti chyb, pokud se příslušný Lagrangeův multiplikátor nenachází na mezích. Jinak se bude diskriminační funkce SVM počítat na základě aktuálního vektoru $\boldsymbol{\alpha}$.

5.4 Problémy SMO algoritmu

SMO je důkladně organizovaný a výpočetně velice efektivní algoritmus, nicméně Keerthi ve své práci[6] popisuje, že díky způsobu provádění jednotlivých výpočtů a použití pouze jednoho prahu b nemusí korektně zkonvergovat

a tím se také stát neefektivním. To nastává v případě, že všechny hodnoty $\boldsymbol{\alpha}$ jsou po optimalizaci (v dané iteraci algoritmu) na mezích (rovny buď 0 nebo C) a výsledný práh b bude zvolen nesprávně. Následná kontrola ukončující podmínky bude díky špatnému prahu ovlivněna a může způsobit neukončení algoritmu a tím snížení jeho rychlosti, kvůli další (zbytečné) iteraci optimalizační smyčky. Naštěstí tento problém postupně mizí se zvyšujícím se počtem trénovacích obrazů a tedy i nižší pravděpodobností, že všechny hodnoty $\boldsymbol{\alpha}$ budou ležet na jedné z mezí.

Platt proto navrhl modifikaci, kdy nebude použit pouze jeden práh, nýbrž dva: b_{up} a b_{low}, pro které, za předpokladu známých hodnot vektoru $\mathbf{F}$, platí následující

$$\begin{aligned} b_{low} &= (F_{l_{low}} =) \max_{l \in I_0 \cup I_3 \cup I_4} \{F_l\}, \\ b_{up} &= \left(F_{l_{up}} =\right) \min_{l \in I_0 \cup I_1 \cup I_2} \{F_l\}, \end{aligned} \tag{5.10}$$

kde použité množiny indexů l jsou definovány následovně: $I_0 = \{l : 0 < \alpha_l < C\}$; $I_1 = \{l : y_l = 1, \alpha_l = 0\}$; $I_2 = \{l : y_l = -1, \alpha_l = C\}$; $I_3 = \{l : y_l = 1, \alpha_l = C\}$; $I_4 = \{l : y_l = -1, \alpha_l = 0\}$.

Díky této úpravě při hledání druhého indexu (v dané iteraci) algoritmus nejenže zvolí optimální druhý index, ale navíc ověří, kolik trénovacích obrazů ještě porušuje podmínky optimality a tím umožní snazší a korektnější (oproti originálnímu SMO) ukončení trénovacího algoritmu.

A protože chceme aby algoritmus zůstal efektivní, ale cache vektoru $\mathbf{F}$ uchovává jen hodnoty příslušné k indexům z množiny I_0, je optimálním řešením této situace pro určení b_{up} a b_{low}, použít jen tyto hodnoty.

V kroku SMO, kde se aktualizují hodnoty vektoru $\mathbf{F}$ je snadné do cache přidat i hodnoty pro nově určené indexy i a j, nechť $\tilde{I} = I_0 \cup \{i, j\}$. Následně se *částečně* spočtou dvojice (i_{low}, b_{low}) a (i_{up}, b_{up}) pouze nad indexy z množiny $\tilde{I}$.

Nově *částečně* vypočtené hodnoty b_{up} a b_{low} mohou být dále použity pro ověření ukončovací podmínky pro vnitřní optimalizační (vnitřní) smyčku,

pracující pouze nad množinou indexů I_0

$$b_{low} \leq b_{up} + 2 \cdot \tau, \tag{5.11}$$

kde τ je povolená odchylka.

Dále je možné zvolit takový index j, jehož obraz nejvíce porušuje podmínky optimality a na jeho základě dopočítat odpovídající obraz s indexem i. Nebo lze hledat takovou dvojici, která společně porušuje podmínky optimality nejvýrazněji.

Pokud je optimalizace nad množinou indexů I_0 hotova, pokračuje se optimalizací přes všechny trénovací obrazy (vnější smyčka). Takt se pokračuje dokud existuje nějaký obraz, který ještě porušuje podmínky optimality.

6 Analýza referenčních implementací SVM klasifikátoru

V této kapitole budou představeny a důkladně analyzovány algoritmy trénování a klasifikace SVM klasifikátorů implementovaných ať v klasifckém programovacím jazyce na x86 architektuře (C++) nebo v jazyce umožňujícím využít akcelerace paralelizovatelných částí výpočtů za pomoci grafické karty (Nvidia CUDA C).
Jedná se o algoritmy:

- libSVM - Chih-Chung Chang and Chih-Jen Lin [9]
- cuSVM - Austin Carpenter
- gpuSVM - Bryan Catanzaro
- GPU-accelerated LIBSVM - A. Athanasopoulos, A. Dimou, V. Mezaris, I. Kompatsiaris [10]

6.1 libSVM

Tuto knihovnu naprogramovanou v jazyce C++ lze v oblasti SVM klasifikátorů považovat za standard, na který se odkazuje při porovnávání výsledků a časů běhu většina aktuálních implementací. Z tohoto důvodu jsou velmi rozšířené implementace založené právě na tomto balíku, jedná se např. o Matlab, Python, Java a další.

Tato knihovna používá pro trénování algoritmus SMO s modifikacemi od Keerthi. Je tedy velmi přesná co se týče výsledků a proto budou její výsledky brány jako referenční v porovnání s ostatními.

6.2 cuSVM

Austin Carpenter vytvořil SVM klasifikátor s RBF kernelem, přitom využil jazyků C/C++ a CUDA C jak pro klasifikaci, tak pro trénování. Podstatnou součástí jeho kódu je tzv. *wrapper*, který umožňuje použít tohoto akcelerovaného algoritmu v prostředí Matlab. Výsledkem jeho práce jsou dvě Matlab funkce, které (zdokumentovány v . . .) lze jednoduše použít v daném prostředí (avšak vše nemusí fungovat tak, jak by se očekávalo, převážně kvůli změnám

v relativně mladém programovacím jazyce CUDA C).

Pro použití této implementace v benchmarku porovnávajícím jednotlivé implementace mezi sebou, bylo nutné doprogramovat rozhraní pro jazyk C a doprogramovat veškerou interakci se vstupními a výstupními soubory, protože originální implementace plně spoléhala na data z prostředí Matlabu.

Pro trénování SVM je použit algoritmus SMO, který výrazně redukuje paměťové nároky základního algoritmu chunking.
Algoritmus trénování je implementován ve dvou variantách:

1. Regresní trénování
2. Klasifikační trénování

6.2.1 Regresní trénování

Při regresním trénování se nepředpokládá apriorní znalost třídy do které daný vektor patří, je tudíž nutné předpokládat že daný trénovací vektor může patřit do libovolné z nich.

Nejprve se provádí transpozice matice trénovacích vektorů a výpočet skalárního součinu trénovacích vektorů se sebou samými.
Následně se inicializují hodnoty vektorů následovně:

- $\widetilde{\alpha}_i = 0 \quad \forall i = 1, \ldots, 2 \cdot n$, kde n je počet trénovacích vektorů
- $y_i = \begin{cases} 1 & \text{při } i \leq n \\ -1 & \text{jinak} \end{cases}$,
- $F_i = \begin{cases} -1 + \varepsilon & \text{při } i \leq n \\ 1 + \varepsilon & \text{jinak,} \end{cases}$
  kde ε je .

Dále algoritmus pokračuje smyčkou, která vypočte správné parametry $\widetilde{\alpha}_i^k$ pro jednotlivé trénovací vektory a práh ρ:

1. Hledání b_i a příslušného indexu i:

$$\begin{aligned} b_i &= \max_{k \in I}(-F_k \cdot y_k) \\ i &= \arg\max_{k \in I}(-F_k \cdot y_k), \end{aligned} \tag{6.1}$$

kde $I = \left\{i : (\widetilde{\alpha}_i^k < C \wedge y_i = 1) \vee (\widetilde{\alpha}_i^k > 0 \wedge y_i \neq 1)\right\}$, C je konstanta učení určující rychlost a stabilitu konvergence algoritmu, $\widetilde{\alpha}_i^k$ je parametr i-tého trénovacího vektoru v aktuálním kroku a y_i je požadovaný výstup pro i-tý trénovací vektor.

Kvůli určení maximální hodnoty jsou trénovací vektory rozděleny do 64 skupin (každá skupina má 256 vláken) a pro každou z nich je pomocí CUDA kernelu určeno lokální maximum. Globální maximum se poté vypočte na CPU.

2. Každých 256 iterací algoritmu se provede kontrola splnění ukončující podmínky. Pokud je celková chyba klasifikátoru J menší než zadaný práh, tak se algoritmus ukončí a hodnota prahu ρ se nastaví na hodnotu $\frac{b_i+b_j}{2}$, jinak se pokračuje dalším krokem.

$$J = \min_{j \in M}(-F_j \cdot y_j), \tag{6.2}$$

kde $M = \left\{j : (\widetilde{\alpha}_j^k < C \wedge y_j \neq 1) \vee (\widetilde{\alpha}_j^k > 0 \wedge y_j = 1)\right\}$

3. Následně se ověří existence výsledku RBF kernelu pro i-tý (výpočet i viz krok 1)trénovací vektor. Pokud je tato hodnota ještě v zásobníku (který má omezenou kapacitu v závislosti na velikosti paměti GPU), tak se pokračuje dalším krokem, jinak je nutné tuto hodnotu vypočítat a následně ji do zásobníku přidat (případně odebrat hodnotu, která je v zásobníku nejdéle).
   V tomto kroku je využito předem vypočítaného vektoru skalárních součinů všech trénovacích vektorů se sebou samými a upraveného vektorového tvaru pro výpočet hodnoty RBF kernelu:

$$\mathrm{k}(\mathbf{x}_a, \mathbf{x}_b) = \exp\left[\gamma \cdot (\mathbf{x}_a^{\mathrm{T}} \cdot \mathbf{x}_a + \mathbf{x}_b^{\mathrm{T}} \cdot \mathbf{x}_b - 2 \cdot \mathbf{x}_a^{\mathrm{T}} \cdot \mathbf{x}_b)\right], \tag{6.3}$$

   kde $\mathbf{x}$ je trénovací vektor (zde je brán předpoklad sloupcového vektoru). Přičemž skalární součiny $\mathbf{x}^{\mathrm{T}} \cdot \mathbf{x}$ jsou již vypočteny (bylo tak učiněno před započetím trénovacího algoritmu) pro všechny trénovací vektory $\mathbf{x}$.

Jedinou neznámou hodnotou zůstává skalární součin $\mathbf{x}_a^{\mathrm{T}} \cdot \mathbf{x}_b$. Protože

pro udržení výsledků skalárního součinu všech dvojic trénovacích vektorů by bylo zapotřebí extrémního množství paměti, je nutné tuto hodnotu pokaždé vypočítat. K tomu je používa funkce z knihovny CUBLAS, `cublasSgemv()`, která je schopna skalárního součinu matice (v tomto případě matice trénovacích vektorů) a dalšího vektoru (zvolený trénovací vektor).

Výsledkem tohoto kroku je, že v zásobníku existuje vektor $\mathrm{k}(\mathbf{x}_i, \mathbf{x}_k) \quad \forall k = 1, \ldots, n$, kde n je počet trénovacích vektorů.

4. Hledání hodnoty b_j pomocí druhé heuristiky SMO:

$$\begin{aligned} b_j &= \max_{k \in M} \left[\frac{(b_i + F_k \cdot y_k)^2}{den} \right] \\ j &= \arg\max_{k \in M} \left[\frac{(b_i + F_k \cdot y_k)^2}{den} \right], \end{aligned} \tag{6.4}$$

kde M je množina z kroku 2 a platí:

$$den = \begin{cases} 2 - 2 \cdot \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) & \text{při } (2 - 2 \cdot \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j)) < \tau_{min} \\ \tau_{min} & \text{jinak,} \end{cases} \tag{6.5}$$

5. Získání nových hodnot $\widetilde{\alpha}_i^{k+1}$ a $\widetilde{\alpha}_j^{k+1}$:
Pokud požadovaná odezva klasifikátoru na příslušný i-tý a j-tý trénovací vektor není stejná ($y_i \neq y_j$), tak jsou pro výpočet $\widetilde{\alpha}_i^k$ a $\widetilde{\alpha}_j^k$ použity následující vzorce:

$$\widetilde{\alpha}_i^{k+1} = \begin{cases} \Delta_\alpha & \text{při } (\Delta_\alpha > 0) \wedge (\widetilde{\alpha}_j^k + \lambda_d < 0) \\ C & \text{při } (\Delta_\alpha > 0) \wedge (\widetilde{\alpha}_i^k + \lambda_d > C) \\ 0 & \text{při } (\Delta_\alpha \leq 0) \wedge (\widetilde{\alpha}_i^k + \lambda_d < 0) \\ C + \Delta_\alpha & \text{při } (\Delta_\alpha \leq 0) \wedge (\widetilde{\alpha}_j^k + \lambda_d > C) \\ \widetilde{\alpha}_i^k + \lambda_d & \text{jinak;} \end{cases} \tag{6.6}$$

$$\widetilde{\alpha}_j^{k+1} = \begin{cases} 0 & \text{při } (\Delta_\alpha > 0) \wedge (\widetilde{\alpha}_j^k + \lambda_d < 0) \\ C - \Delta_\alpha & \text{při } (\Delta_\alpha > 0) \wedge (\widetilde{\alpha}_i^k + \lambda_d > C) \\ -\Delta_\alpha & \text{při } (\Delta_\alpha \leq 0) \wedge (\widetilde{\alpha}_i^k + \lambda_d < 0) \\ C & \text{při } (\Delta_\alpha \leq 0) \wedge (\widetilde{\alpha}_j^k + \lambda_d > C) \\ \widetilde{\alpha}_j^k + \lambda_d & \text{jinak,} \end{cases} \tag{6.7}$$

kde $\Delta_\alpha = \widetilde{\alpha}_i^k - \widetilde{\alpha}_j^k$, k je ředchozí krok algoritmu a platí:

$$\lambda_d = \frac{-F_i - F_j}{\lambda_{den}}, \tag{6.8}$$

přičemž:

$$\lambda_{den} = \begin{cases} 2 - 2 \cdot \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) & \text{při } (2 - 2 \cdot \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j)) < \tau_{min} \\ \tau_{min} & \text{jinak,} \end{cases} \tag{6.9}$$

V případě, že jsou obě požadované odezvy stejné ($y_i = y_j$), je nutné použít jiné vzorce:

$$\widetilde{\alpha}_i^{k+1} = \begin{cases} C & \text{při } (\sum_\alpha > C) \wedge (\widetilde{\alpha}_i^k - \lambda_s > C) \\ \sum_\alpha - C & \text{při } (\sum_\alpha > C) \wedge (\widetilde{\alpha}_j^k + \lambda_s > C) \\ 0 & \text{při } (\sum_\alpha \leq C) \wedge (\widetilde{\alpha}_i^k - \lambda_s < 0) \\ \sum_\alpha & \text{při } (\sum_\alpha \leq C) \wedge (\widetilde{\alpha}_j^k + \lambda_s < 0) \\ \widetilde{\alpha}_i^k - \lambda_s & \text{jinak;} \end{cases} \tag{6.10}$$

$$\widetilde{\alpha}_j^{k+1} = \begin{cases} \sum_\alpha - C & \text{při } (\sum_\alpha > C) \wedge (\widetilde{\alpha}_i^k - \lambda_s > C) \\ C & \text{při } (\sum_\alpha > C) \wedge (\widetilde{\alpha}_j^k + \lambda_s > C) \\ \sum_\alpha & \text{při } (\sum_\alpha \leq C) \wedge (\widetilde{\alpha}_i^k - \lambda_s < 0) \\ 0 & \text{při } (\sum_\alpha \leq C) \wedge (\widetilde{\alpha}_j^k + \lambda_s < 0) \\ \widetilde{\alpha}_j^k + \lambda_s & \text{jinak,} \end{cases} \tag{6.11}$$

kde $\sum_\alpha = \widetilde{\alpha}_i^k + \widetilde{\alpha}_j^k$ a platí:

$$\lambda_s = \frac{F_i - F_j}{\lambda_{den}} \tag{6.12}$$

Pozn.: Tento algoritmus je implementován v C a jedná se o upravený kód z knihovny libSVM.

6. Dále je nutné ověřit existenci výsledku j-tého kernelu v zásobníku podobně jako tomu bylo pro i-tý kernel v kroku 3.

7. Aktualizace hodnot vektoru $\mathbf{F}$:

$$F_l^{k+1} = F_l^k + y_l \cdot (y_i \cdot \Delta_{\alpha_i} \cdot \mathrm{k}(\mathbf{x}_i, \mathbf{x}_l) + y_j \cdot \Delta_{\alpha_i} \cdot \mathrm{k}(\mathbf{x}_j, \mathbf{x}_l)) \quad \forall l = 1, \ldots, n, \tag{6.13}$$

kde n je počet trénovacích vektorů, $\Delta_{\alpha i} = \widetilde{\alpha}_i^{k+1} - \widetilde{\alpha}_i^k$ ($\Delta_{\alpha j} = \widetilde{\alpha}_j^{k+1} - \widetilde{\alpha}_j^k$)

a k je předchozí krok algoritmu.

8. Návrat na krok 1.

Výše popsaný algoritmus vypočte hodnotu prahu ρ a vektor parametrů $\widetilde{\boldsymbol{\alpha}}$ (dimenze n - počet trénovacích vektorů).

Ve skutečnosti však algoritmus proběhne 2×. Důvodem je to, že výsledné hodnoty vektoru $\widetilde{\boldsymbol{\alpha}}$ jsou v rozmezí $\langle 0; 1\rangle$ a vyjadřují, zda byla konkrétní počáteční hodnota vektoru $\mathbf{y}$ (rozhodnutí klasifikátoru: -1 nebo 1) správně zvolena. Pro každý trénovací vektor se tedy zjišťuje správnost volby třídy. Kvůli tomu, aby algoritmus nebylo nutné spouštět 2× za sebou (nejprve s $y_i = -1 \forall i$ a poté $y_i = 1 \forall i$), tak se vše provádí najednou (paralelně), proto mají vektory $\widetilde{\boldsymbol{\alpha}}$, $\mathbf{y}$ a $\mathbf{F}$ dvojnásobnou velikost než je počet trénovacích vektorů.
Pokud je odezva klasifikátoru i-tého vektoru (y_i) správná, tak je příslušná hodnota α_i nenulová.

Dále je nutné vypočítat skutečný vektor $\boldsymbol{\alpha}$:

$$\alpha_i = \widetilde{\alpha}_i|_{y_i=+1} - \widetilde{\alpha}_i|_{y_i=-1} \quad \forall i = 1, \ldots, n \tag{6.14}$$

Na závěr se všechny trénovací vektory, které mají nenulovou hodnotu α_i, se prohlásí za podpůrné vektory.

Každé určení extrému (minimum nebo maximum a popřípadě i příslušný index) je prováděno obdobným způsobem jako v kroku 1 (64 bloků po 256 vláknech určí lokální extrém a pomocí CPU se zjistí globální extrém). Pro výpočet RBF kernelů a aktualizace hodnot vektoru $\mathbf{F}$ není pevně dán počet vláken ani bloků, tyto hodnoty jsou vypočteny na základě dimenze trénovacích vektorů.

Pozor! Při hledání hodnot i a j a při ověřování splnění ukončující podmínky je využito optimalizovaného algoritmu, který předpokládá, že posledních 32 vláken poběží vždy synchronně a nebude nutné je ručně synchronizovat. To platí **pouze** pro architektury steaming multiprocessor (SM) **1.3** a nižší. Na novějších GPU synchronizace nebude zaručena a s velkou pravděpodobností nebude výsledek správný.

6.2.2 Klasifikační trénování

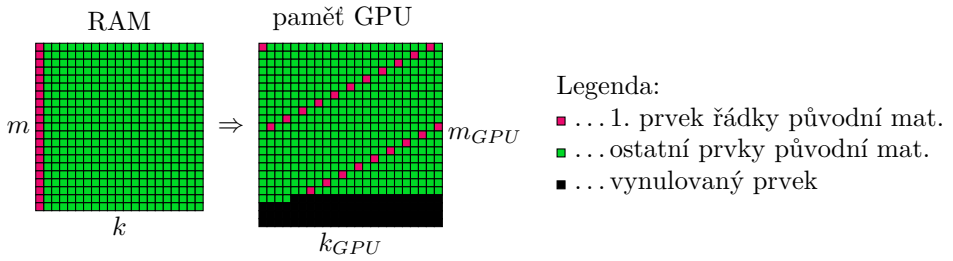
Při tomto trénování je známa informace „od učitele“, do které třídy daný trénovací vektor patří. Tato metoda je tedy oproti předchozí jednodušší a použitý algoritmus je rozdílný jen v následujících bodech:

- Algoritmus proběhne pouze jednou (není tedy zapotřebí dvojnásobné velikosti polí pro hodnoty $\widetilde{\boldsymbol{\alpha}}$ a $\mathbf{F}$).
- Hodnoty vektoru $\widetilde{\boldsymbol{\alpha}}$ jsou již konečné a není je nutné nijak upravovat.
- Hodnoty vektoru $\mathbf{F}$ se inicializují na -1 a vektor očekávané odezvy klasifikátoru $\mathbf{y}$ není potřebný.

6.2.3 Klasifikace

Úprava dimenze a počtu podpůrných vektorů na celočíselný násobek 32:

1. Nakopírování matice podpůrných vektorů z RAM na GPU



kde m je počet řádek původní matice, k je délka řádku původní matice, m_{GPU} a k_{GPU} jsou upravené rozměry matice na násobek 32. Austin Carpenter ve svém algoritmu používá pro zaokrouhlení na celočíselný násobek 32 následující vzorec:

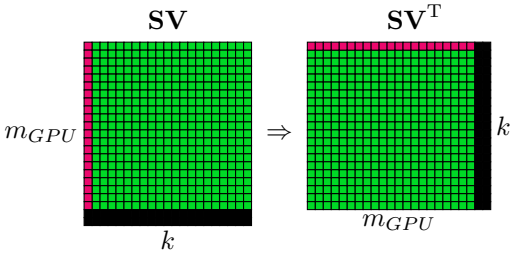
$$n_{GPU} = n + 32 - \left\lceil \frac{n}{32} - \left\lfloor \frac{n}{32} \right\rfloor \right\rceil \cdot 32, \tag{6.15}$$

kde n_{GPU} je nový rozměr (použitý pro výpočty a alokace na GPU), n je dimenze v RAM a $\lfloor x \rceil$ je zaokrouhlení nejbližší celé číslo, které není vyšší než x. Tento vzorec byl v jazyce C zapsán takto: `n_GPU = n + 32 - n % 32;`.

Na obrázku výše je znázorněn obsah paměti GPU po přenosu dat z RAM. Před samotným kopírováním byla daná část paměti GPU vyplněna nulami, což po nakopírování dat způsobilo, že v části, kam nebyla ukládána data jsou nuly.
Posunutí začátku řádku bylo způsobeno tím, že se při kopírování původních dat nerespektoval rozdíl rozměrů obou matic. To je řešeno v dalších krocích algoritmu.

2. Provedení první transpozice v paměti GPU

   Jelikož jsou všechny matice reprezentovány (jednorozměrnými) poli, je možné k nim přistupovat s jinými rozměry než jsou skutečné rozměry matice. Toho je využito při prvním transponování k tomu, aby ve výsledné matici v paměti GPU začínal řádek původní matice na začátku řádku matice nové.



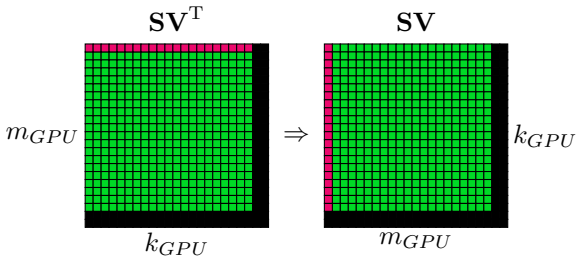
   Transpozive vstupní matice SVs je uložena do dočasné proměnné SVs^{T}, která byla opět předvyplněna nulami.

   Při této transpozici se nepracuje s celými maticemi ($m_{GPU} \times k_{GPU}$), ale pouze s jejich částí ($m_{GPU} \times k$, kde $k < k_{GPU}$).

3. Provedení druhé transpozice v paměti GPU

   Tato úprava převede transponovanou matici z kroku 2 do správného tvaru, kde bude možné použít rozměry s kterými byla matice vytvořena.

Výsledkem tohoto postupu je přidání nul na konec řádky (tak, aby na pozicích daného řádku původní matice a matice nové byly stejné hodnoty) a přidání nul na v původní matici neexistující řádky. Podpůrné vektory jsou



tedy připraveny k použití v paměti GPU tak, jak je tam bude následný algoritmus SVM klasifikátoru očekávat.

Samotná klasifikace je rozdělena do podúloh zpracovávaných v sérii. Počet těchto úloh závisí na počtu klasifikovaných vektorů tak, aby paměť pro mezivýpočty alokovaná na GPU nepřekročila určitou kapacitu (konkrétně byla zvolena hodnota 32 GB $\doteq$ 29,8 GiB).
Každá úloha zpracovává pouze část testovaných vektorů a je složena ze tří kroků:

- Odeslání určitého počtu testovacích vektorů do paměti GPU pomocí knihovny CUBLAS.
- Výpočet RBF kernelu.

  $$\mathrm{k}\left(\mathbf{sv}_i, \mathbf{v}\right) = \exp\left(-\gamma \cdot \left\|\mathbf{sv}_i - \mathbf{v}\right\|^2\right), \tag{6.16}$$

  kde $\mathbf{sv}_i$ je i-tý podpůrný vektor, $\mathbf{v}$ je klasifikovaný vektor a γ je parametr RBF kernelu získaný při trénování.

  Carpenter rozděluje výpočet kernelu na dvě části:

  - Výpočet záporné druhé mocniny Eukleidovské vzdálenosti ($-\left\|\mathbf{sv}_i - \mathbf{v}\right\|^2$).
    Zde je využito faktu, že druhou mocninu Eukleidovské vzdálenosti, lze rozepsat následovně:

    $$\begin{aligned}\left\|\mathbf{sv}_i - \mathbf{v}\right\|^2 &= (sv_{i,1} - v_1)^2 + (sv_{i,2} - v_2)^2 + \ldots + (sv_{i,n} - v_n)^2 = \\ &= \sum_{j=1}^{n}(sv_{i,j} - v_j)^2 = \sum_{j=1}^{d}\left(-2 \cdot sv_{i,j} \cdot v_j + sv_{i,j}^2 + v_j^2\right),\end{aligned}$$

kde $sv_{i,j}$ je j-tý prvek i-tého podpůrného vektoru, v_j je j-tý prvek klasifikovaného vektoru a d je dimenze podpůrných vektorů (musí být totožná s dimenzí testovaného vektoru).
Po vynásobení tohoto vzorce -1 vypadá výpočet jednoho kroku sumy následovně:

$$c_j = sv_{i,j} \cdot v_j - sv_{i,j}^2 - v_j^2, \tag{6.17}$$

kde c_j je j-tý prvek pole pro mezivýsledky.

– Po sečtení všech mezivýsledků se pro každý testovaný vektor použije výše uvedená RBF kernel funkce, tedy:

$$\mathrm{k}\left(\mathbf{sv}_i, \mathbf{v}\right) = \exp\left(\gamma \cdot \sum_{j=1}^{d} c_j\right), \tag{6.18}$$

kde d je dimenze testovaných vektorů.

- Výpočet neprahovaného výsledku klasifikace.
V tomto kroku se výsledek RBF kernel funkce vynásobí příslušným parametrem α_i^k příslušejícím k $\mathbf{sv}_i$:

$$r_k = \sum_{i=1}^{n} \alpha_i^k \cdot \mathrm{k}\left(\mathbf{sv}_i, \mathbf{v}_k\right), \tag{6.19}$$

kde r je výsledek klasifikace k-tého testovaného vektoru, který není porovnán s prahem.

Nakonec se k jednotlivým výsledkům přičte hodnota prahu. Tato hodnota se v případě, že se jedná o klasifikaci (a nikoliv o regresi), porovná s nulou a nahradí se označením příslušné třídy (pro hodnoty $< 0 \rightarrow -1$ a pro hodnoty $\geq 0 \rightarrow 1$).

6.3 gpuSVM

6.3.1 Trénování

Trénování gpuSVM klasifikátoru na vstupu vyžaduje transponované trénovací vektory uložené v matici, tedy každý sloupec matice je jeden trénovací vektor. Dobrovolným parametrem je matice trénovacích vektorů, které nebyly předem transponovány, přičemž pokud tato matice není zadána, tak ji

algoritmus určí z její transponované verze.

Před započetím samotného trénování je zvolena grafická karta, která má nejvyšší počet multi-procesorů, je zde brán předpoklad, že více multi-procesorů zaručí vyšší počet výpočetních jednotek. To ale nelze zaručit obecně, avšak v době programování této práce nebylo možné počet výpočetních jednotek jednoduše určit. K tomu by bylo zapotřebí znát všechny architektury (SM = 1.0, 1.1, 1.2, 1.3, 2.0, 2.1, ...) a jim odpovídající velikost multi-procesoru. Což by nefungovalo pro novější grafické karty, které mají architektury v té době neznámé.

Na začátku se provedou všechny alokace paměti na GPU. Přičemž jako poslední se alokuje paměť uchovávající maximální možný počet naposledy spočtených skalárních součinů trénovacích vektorů se všemi trénovacími vektory. A pro každý z implementovaných SVM kernelů jsou nastaveny parametry následovně:

Typ SVM kernelu	a	b	c
RBF	$-\gamma$	–	–
lineární	–	–	–
polynomiální	γ	$coef0$	$degree$
tangenciální	γ	$coef0$	–

Tabulka 6.1: Parametry trénovacího algoritmu SVM klasifikátoru gpuSVM

Kde znak „–“ značí nenastavený parametr, tento parametr tedy není při výpočtu čten.

Dále kód pokračuje podle zvoleného SVM kernelu.
Inicializace F_i a α_i:

$$F_i = -y_i \quad i = 1, \ldots, n, \tag{6.20}$$

kde **F** je pomocný vektor pro trénování, **y** je vektor s informací o zařazení do příslušné třídy a n je počet trénovacích vektorů.

$$\alpha_i = 0 \quad i = 1, \ldots, n, \tag{6.21}$$

kde $\boldsymbol{\alpha}$ je výstupní vektor.

Výpočet SVM kernel funkcí všech testovacích vektorů se sebou samými, tedy $\mathrm{k}(\mathbf{x}_i, \mathbf{x}_i)$, kde $i = 1, \ldots, n$.

Pozn.: Pro RBF kernel je hodnota $\mathrm{k}(\mathbf{x}_i, \mathbf{x}_i) = 1 \; \forall i$.

Kde $\mathbf{x}_i$ je i-tý trénovací vektor (v řádkové formě), $i = 1, \ldots, n$. První krok algoritmu, kde je využito pevně zvolených počátečních podmínek pro zjednodušení výpočtů:
Nastavení indexů pro výběr dvojice vhodných trénovacích vektorů ke zpracování v aktuálním kroku. Index i_L je zvolen podle první hodnoty $y_i < 0$, index i_H je zvolen podle první hodnoty $y_i > 0$ a jim příslušné hodnoty $b_L = 1$, $b_H = -1$.

Dále je spočtena SVM kernel funkce mezi $\mathbf{x}_{i_L}$ a $\mathbf{x}_{i_H}$ a následně je vypočten parametr η:

$$\eta = \mathrm{k}(\mathbf{x}_{i_L}, \mathbf{x}_{i_L}) + \mathrm{k}(\mathbf{x}_{i_H}, \mathbf{x}_{i_H}) - 2 \cdot \mathrm{k}(\mathbf{x}_{i_L}, \mathbf{x}_{i_H}) \tag{6.22}$$

V prvním kroku je zaručeno, že $\eta > 0$ a je definováno, že $\alpha_i^0 = 0 \; \forall i \in \langle 0; n \rangle$, lze tudíž určit nové hodnoty $\boldsymbol{\alpha}$ následovně:

$$\alpha_{i_L}^1 = \alpha_{i_H}^1 = \begin{cases} \frac{2}{\eta} & \text{při } \frac{2}{\eta} < C \\ C & \text{jinak,} \end{cases} \tag{6.23}$$

kde C je konstanta učení určující rychlost a stabilitu konvergence.

Následně je provedena trénovací smyčka, která postupně aktualizuje parametry, dokud maximální chyba klasifikace není menší než dvojnásobek zadané tolerance:

1. Na začátku se ověří, zda zásobník obsahuje výsledky příslušných SVM kernelů pro vektory $\mathbf{x}_{i_L}$ a $\mathbf{x}_{i_H}$. Této informace je využito k tomu, aby se v příslušné části algoritmu tyto kernely spočítaly. A dále je každý 128. krok provedena volba heuristiky, která bude k následnému výpočtu použita.
   Implementovány jsou čtyři metody volby heuristiky:

   - Heuristika prvního řádu - pomalu konvergující.

- Heuristika druhého řádu - rychle konvergující (nezaručuje konvergenci).
- Adaptivní heuristika - podle stavu a rychlosti konvergence se volí heuristika prvního nebo druhého řádu.
- Náhodná heuristika - náhodně se vybírá mezi heuristikou prvního a druhého řádu.

Jakoukoli z těchto možností může uživatel zvolit při spuštění funkce.

2. Heuristika prvního řádu:

   (a) Získání výsledku SVM kernel funkce ze zásobníku nebo jejich vypočtení a uložení na předem určené místo v zásobníku. Na základě čehož se aktualizují hodnoty vektoru **F**:

$$F_j^{k+1} = F_j^k + y_{i_L} \cdot \Delta_{\alpha_{i_L^k}} \cdot \mathrm{k}(\mathbf{x}_{i_L^k}, \mathbf{x}_j) + y_{i_H} \cdot \Delta_{\alpha_{i_H^k}} \cdot \mathrm{k}(\mathbf{x}_{i_H^k}, \mathbf{x}_j) \quad j = 1, \ldots, n, \tag{6.24}$$

   kde y_i je ohodnocení třídy pro i-tá trénovací vektor, $\Delta_{\alpha_i} = \alpha_i^{k+1} - \alpha_i^k$ a n je počet trénovacích vektorů.
   V tuto chvíli je možné určit nové indexy a jim příslušné hodnoty:

$$\begin{aligned} i_L &= \arg\max_{i \in I} F_i \\ b_L &= \max_{i \in I} F_i, \end{aligned} \tag{6.25}$$

   kde $I = \{i : [(\alpha_i > \varepsilon) \wedge (\alpha_i < C - \varepsilon)] \vee [(\alpha_i > \varepsilon) \wedge (\alpha_i \geq C - \varepsilon) \wedge y_i > 0] \vee [(\alpha_i \leq \varepsilon) \wedge y_i \leq 0]\}$.

$$\begin{aligned} i_H &= \arg\min_{i \in J} F_i \\ b_H &= \min_{i \in J} F_i, \end{aligned} \tag{6.26}$$

   kde $J = \{i : [(\alpha_i > \varepsilon) \wedge (\alpha_i < C - \varepsilon)] \vee [(\alpha_i > \varepsilon) \wedge (\alpha_i \geq C - \varepsilon) \wedge y_i \leq 0] \vee [(\alpha_i \leq \varepsilon) \wedge y_i > 0]\}$.

   (b) Dále jsou určeny horní a dolní meze pro hodnoty vektoru $\boldsymbol{\alpha}$ na

indexech i_H^{k+1} a i_L^{k+1}:

$$\alpha_{LB} = \begin{cases} 0 & \text{při } (y_{i_H} \cdot y_{i_L} < 0) \wedge (\Delta_\alpha < 0) \\ \Delta_\alpha & \text{při } (y_{i_H} \cdot y_{i_L} < 0) \wedge (\Delta_\alpha \geq 0) \\ 0 & \text{při } (y_{i_H} \cdot y_{i_L} \geq 0) \wedge (\sum_\alpha < C) \\ \sum_\alpha - C & \text{jinak,} \end{cases} \tag{6.27}$$

$$\alpha_{HB} = \begin{cases} \Delta_\alpha + C & \text{při } (y_{i_H} \cdot y_{i_L} < 0) \wedge (\Delta_\alpha < 0) \\ C & \text{při } (y_{i_H} \cdot y_{i_L} < 0) \wedge (\Delta_\alpha \geq 0) \\ \sum_\alpha & \text{při } (y_{i_H} \cdot y_{i_L} \geq 0) \wedge (\sum_\alpha < C) \\ C & \text{jinak,} \end{cases} \tag{6.28}$$

kde α_{LB} je dolní mez pro výpočet α_{i_H}, α_{HB} je horní mez pro výpočet α_{i_H}, $\Delta_\alpha = \alpha_{i_L} - \alpha_{i_H}$ a $\sum_\alpha = \alpha_{i_L} - \alpha_{i_H}$.

Aktualizace hodnoty α_{i_L} při splnění podmínky $\eta > 0$, kde $\eta = \mathrm{k}(\mathbf{x}_{i_H}, \mathbf{x}_{i_H}) + \mathrm{k}(\mathbf{x}_{i_L}, \mathbf{x}_{i_L}) - 2 \cdot \mathrm{k}(\mathbf{x}_{i_L}, \mathbf{x}_{i_H})$:

$$\alpha_{i_L}^{k+1} = \alpha_{i_L}^{k} + \frac{y_{i_L} \cdot (b_H - b_L)}{\eta} \tag{6.29}$$

Pokud je tato nová hodnota mimo interval $\langle \alpha_{LB}; \alpha_{LB} \rangle$, je nutné ji pozměnit následujícím způsbem:

$$\alpha_{i_L}^{k+1} = \begin{cases} \alpha_{LB} & \text{při } \alpha_{i_L}^{k+1} < \alpha_{LB} \\ \alpha_{HB} & \text{při } \alpha_{i_L}^{k+1} > \alpha_{HB} \end{cases} \tag{6.30}$$

kde $k+1$ je aktuální krok algoritmu.

Při nesplnění výše uvedené podmínky, tedy $\eta \leq 0$, je zapotřebí použít jiný vztah:

$$\alpha_{i_L}^{k+1} = \begin{cases} \alpha_{HB} & \text{při } [y_{i_L} \cdot (b_H - b_L) \cdot (\alpha_{HB} - \alpha_{LB}) > 0] \\ & \wedge [y_{i_L} \cdot (b_H - b_L) > 0] \\ \alpha_{LB} & \text{při } [y_{i_L} \cdot (b_H - b_L) \cdot (\alpha_{HB} - \alpha_{LB}) > 0] \\ & \wedge [y_{i_L} \cdot (b_H - b_L) \leq 0] \\ \alpha_{i_L}^{k} & \text{jinak.} \end{cases} \tag{6.31}$$

Pro aktualizaci hodnoty α_{i_H} lze poté použít následující vztah:

$$\alpha_{i_H}^{k+1} = \alpha_{i_H}^{k} - y_{i_H} \cdot y_{i_L} \cdot (\alpha_{i_L}^{k+1} - \alpha_{i_L}^{k}) \tag{6.32}$$

3. Heuristika druhého řádu:

   (a) Aktualizace hodnot vektoru $\mathbf{F}$:

$$\begin{aligned} F_j^{k+1} = F_j^k + y_{i_L} \cdot \Delta_{\alpha_{i_L^k}} \cdot \mathrm{k}(\mathbf{x}_{i_L^k}, \mathbf{x}_j) + y_{i_H} \cdot \Delta_{\alpha_{i_H^k}} \cdot \mathrm{k}(\mathbf{x}_{i_H^k}, \mathbf{x}_j) \\ j = 1, \ldots, n, \end{aligned} \tag{6.33}$$

   Dále se určí horní mez a jí příslušný index:

$$\begin{aligned} i_H &= \arg\min_{i \in J} F_i \\ b_H &= \min_{i \in J} F_i, \end{aligned} \tag{6.34}$$

   kde $J = \{i : [(y_i > 0) \wedge (\alpha_i < C - \varepsilon)] \vee [(y_i < 0) \wedge (\alpha_i > \varepsilon)]\}$.

   (b) Výpočet dolní meze:

$$b_L = \max_{i \in K} F_i \tag{6.35}$$

   kde $K = \{i : [(y_i > 0) \wedge (\alpha_i > \varepsilon)] \vee [(y_i < 0) \wedge (\alpha_i < C - \varepsilon)]\}$.
   Výpočet indexu dolní meze:

$$i_L = \arg\max_{i \in L} \frac{(b_H - F_i)^2}{\kappa} \tag{6.36}$$

   kde $L = \{i : (i \in K) \wedge [(b_H - F_i) \leq \varepsilon]\}$ a

$$\kappa = \begin{cases} \eta_i & \text{při } \eta_i > 0 \\ \varepsilon & \text{jinak.} \end{cases}, \tag{6.37}$$

   kde $\eta_i = \mathrm{k}(\mathbf{x_{i_H}}, \mathbf{x_{i_H}}) + \mathrm{k}(\mathbf{x_i}, \mathbf{x_i}) - 2 \cdot \mathrm{k}(\mathbf{x_{i_H}}, \mathbf{x_i})$

   (c) Aktualizace hodnot vektoru $\boldsymbol{\alpha}$.
   Nejprve je nutné určit horní a dolní mez:

$$\alpha_{LB} = \begin{cases} 0 & \text{při } (y_{i_H} \cdot y_{i_L} < 0) \wedge (\Delta_\alpha < 0) \\ \Delta_\alpha & \text{při } (y_{i_H} \cdot y_{i_L} < 0) \wedge (\Delta_\alpha \geq 0) \\ 0 & \text{při } (y_{i_H} \cdot y_{i_L} \geq 0) \wedge (\sum_\alpha < C) \\ \sum_\alpha - C & \text{jinak,} \end{cases} \tag{6.38}$$

$$\alpha_{HB} = \begin{cases} \Delta_\alpha + C & \text{při } (y_{i_H} \cdot y_{i_L} < 0) \wedge (\Delta_\alpha < 0) \\ C & \text{při } (y_{i_H} \cdot y_{i_L} < 0) \wedge (\Delta_\alpha \geq 0) \\ \sum_\alpha & \text{při } (y_{i_H} \cdot y_{i_L} \geq 0) \wedge (\sum_\alpha < C) \\ C & \text{jinak.} \end{cases} \tag{6.39}$$

Aktualizace hodnoty α_{i_L} při splnění podmínky $\eta > 0$, kde $\eta = \mathrm{k}(\mathbf{x}_{i_H}, \mathbf{x}_{i_H}) + \mathrm{k}(\mathbf{x}_{i_L}, \mathbf{x}_{i_L}) - 2 \cdot \mathrm{k}(\mathbf{x}_{i_L}, \mathbf{x}_{i_H})$:

$$\alpha_{i_L}^{k+1} = \begin{cases} \alpha_{LB} & \text{při } \alpha_{i_L}^k + \dfrac{y_{i_L} \cdot (F_{i_H} - F_{i_L})}{\eta} < \alpha_{LB} \\ \alpha_{HB} & \text{při } \alpha_{i_L}^k + \dfrac{y_{i_L} \cdot (F_{i_H} - F_{i_L})}{\eta} > \alpha_{HB} \\ \alpha_{i_L}^k + \dfrac{y_{i_L} \cdot (b_H - b_L)}{\eta} & \text{jinak.} \end{cases} \tag{6.40}$$

V tomto kroku je provedena nepatrná změna oproti heuristice prvního řádu. Byl zde nahrazen výraz $(b_H - b_L)$ za $(F_{i_H} - F_{i_L})$. Přičemž oba tyto výrazy jsi si v první heuristice rovny, avšak v tomto případě index i_L nepřísluší k hodnotě b_L, a proto musí být hodnota z tohoto indexu načtena přímo z vektoru $\mathbf{F}$.

Při nesplnění výše uvedené podmínky, tedy v případě, že $\eta \leq 0$, je zapotřebí použít vztahu:

$$\alpha_{i_L}^{k+1} = \begin{cases} \alpha_{HB} & \text{při } [y_{i_L} \cdot (b_H - b_L) \cdot (\alpha_{HB} - \alpha_{LB}) > 0] \\ & \wedge [y_{i_L} \cdot (b_H - b_L) > 0] \\ \alpha_{LB} & \text{při } [y_{i_L} \cdot (b_H - b_L) \cdot (\alpha_{HB} - \alpha_{LB}) > 0] \\ & \wedge [y_{i_L} \cdot (b_H - b_L) \geq 0] \\ \alpha_{i_L}^k & \text{jinak.} \end{cases} \tag{6.41}$$

Pro aktualizaci hodnoty α_{i_H} lze poté použít následující vztah:

$$\alpha_{i_H}^{k+1} = \alpha_{i_H}^k - y_{i_H} \cdot y_{i_L} \cdot (\alpha_{i_L}^{k+1} - \alpha_{i_L}^k) \tag{6.42}$$

Na závěr, po ukončení trénovací smyčky, se vypočte hodnota prahu:

$$\rho = \frac{b_L + b_H}{2} \tag{6.43}$$

6.3.2 Klasifikace

Zde jsou implementovány čtyři základní SVM kernely:

- RBF,
- polynomiální,
- lineární,
- tangenciální.

Před započetím samotné klasifikace je, stejně jako při trénování, zvolena grafická karta, která má nejvyšší počet multi-procesorů.

Jako první, avšak pouze pro RBF kernel, se vypočtou skalární součiny podpůrných vektorů se sebou samými. K tomu je využit CUDA kernel, kde každá výpočetní jednotka GPU vypočte skalární součet pro jeden podpůrný vektor.

Následně algoritmus pokračuje smyčkou, která provádí klasifikaci pro tak velkou skupinu testovacích vektorů, kolik se jich v danou chvíli vejde do paměti GPU:

1. První krok smyčky se týká pouze RBF kernelu a provádí výpočet skalárních součinů dané skupiny testovacích vektorů se sebou samými.
2. Následně se provede maticová operace pomocí funkce `cublasSgemm()` z knihovny CUBLAS, která provádí následující operaci:

$$\mathbf{C} = \alpha_{sgemm} \cdot \mathbf{SV} \cdot \mathbf{T}^{\mathrm{T}} + \beta_{sgemm} \cdot \mathbf{C}, \tag{6.44}$$

   kde **SV** je matce podpůrných vektorů, **T** je aktuálně zpracovávaná část matice testovacích vektorů, **C** je matice skalárních součinů všech podpůrných vektorů **SV** a aktuálních testovacích vektorů **T** a parametry α_{sgemm} a β_{sgemm} jsou nastaveny podle následující tabulky:

3. Získaná matice **C** se dále využije pro výpočet dílčích hodnot kernel funkcí pro všechny dvojice vektorů, kde jeden vektor je podpůrný z matice **SV** a druhý vektor je testovací z matice **T**. Provádí se tedy

Typ SVM kernelu	α_{sgemm}	β_{sgemm}
RBF	$2 \cdot \gamma$	$-\gamma$
lineární	1	0
polynomiální	γ	0
tangenciální	γ	0

Tabulka 6.2: Parametry funkce `cublasSgemm()` pro výpočet mezivýsledků pro jednotlivé SVM kernely

následující operace:

$$\begin{aligned} L_{i,j}^{RBF} &= \alpha_i \cdot \exp(C_{i,j}), \\ L_{i,j}^{lin} &= \alpha_i \cdot C_{i,j}, \\ L_{i,j}^{poly} &= \alpha_i \cdot (C_{i,j} + coef0)^{degree}, \\ L_{i,j}^{sigm} &= \alpha_i \cdot \tanh(C_{i,j} + coef0), \end{aligned} \tag{6.45}$$

kde $L_{i,j}^{x}$ je výsledek kernelu x ($x = RBF \rightarrow$ RBF kernel, $x = lin \rightarrow$ lineární kernel, $x = poly \rightarrow$ polynomiální kernel, $x = sigm \rightarrow$ tangenciální kernel) pro i-tý podpůrný vektor a j-tý testovaný vektor z matice $\mathbf{T}$, $\boldsymbol{\alpha}$ je vektor koeficientů modelu SVM klasifikátoru a $coef0$ a $degree$ jsou parametry příslušných SVM kernelů získané při trénování.

4. Nakonec se provede součet dílčích výsledků SVM kernelů pro každý testovací vektor (každému testovacímu vektoru přísluší sloupec v matici $\mathbf{L}$) a odečte se hodnota prahu ρ:

$$R_i = \rho + \sum_{j=1}^{m} L_{i,j} \tag{6.46}$$

kde R_i je výsledek klasifikace pro i-tý testovací vektor a m je počet testovaných vektorů.

6.4 GPU-accelerated LIBSVM

Jedná se o úpravu balíku libSVM, kde je pro trénování adaptována implementace využívající GPU (CUDA C), přičemž je zachováno rozhraní referenční knihovny. Akcelerovány pomocí CPU jsou pouze, výpočetně náročné, kernel funkce SVM; zbylá funkcionality Libsvm je zachována.
Dále v textu bude používáno zkráceně jako **gpu-libSVM**.

7 Implementace

Implementace je rozdělena do dvou nezávislých projektů/programů, kde oba jsou naprogramovány v programovacím jazyce C/C++ s využitím vývojového prostředí Visual Studio 2008.
Trénovací program obsahuje 4 implementace SVM:

- libSVM - využívá CPU, autoři: Chih-Chung Chang a Chih-Jen Lin;
- cudaSVM - využívá GPU, autor: Milan Klášterka;
- cuSVM - využívá GPU, autor: Austin Carpenter;
- gpuSVM - využívá GPU, autor: Bryan Catanzaro.

Klasifikační program obsahuje 4 implementace SVM:

- libSVM - využívá CPU, autoři: Chih-Chung Chang a Chih-Jen Lin;
- cudaSVM - využívá GPU, autor: Milan Klášterka;
- cuSVM - využívá GPU, autor: Austin Carpenter;
- gpuSVM - využívá GPU, autor: Bryan Catanzaro.

Oba programy (trénovací a klasifikační) totožně pracují se vstupními a výstupními soubory, což je popsáno v následující podkapitole.

7.1 Vstup

Všechny vstupní soubory jsou pouze textové a musí obsahovat správný formát všech parametrů a hodnot, jinak program navrátí chybovou hodnotu a ukončí svou činnost.

Formát souboru, obsahující trénovací nebo testovací vektory, bude mít následující strukturu:

```
<class_sv_1> <support_vector_1>
<class_sv_2> <support_vector_2>
...
```

Kde `<class_sv_n>` je označení třídy, ke které přísluší následný vektor `<support_vector_n>` zařazen. Při trénování slouží údaj `<class_sv_n>` jako

informace od učitele a při klasifikaci je použit pro kontrolu výstupního rozhodnutí klasifikátoru. Parametr `<support_vector_n>` je trénovací/testovaný vektor ve tvaru:

`<index_1>:<value_1>␣<index_2>:<value_2>␣...`

Kde `<index_n>` je index na jehož pozici má daný podpůrný vektor hodnota `<value_n>`. Díky tomuto zápisu lze velmi snadno uložit vektor tzv. řídkým způsobem, kdy jsou uloženy pouze jeho nenulové hodnoty. Za každou dvojicí (index:hodnota) je vyžadována mezera s výjimkou poslední dvojice, tedy na konci řádky (zde je mezera dobrovolná).
Příklad vstupního souboru s trénovacími/testovanými vektory.

```
+1␣3:1␣11:1␣14:1␣39:1␣42:1␣55:1␣67:1␣73:1␣75:1␣76:1␣83:1␣
-1␣5:1␣11:1␣15:1␣39:1␣40:1␣52:1␣63:1␣73:1␣74:1␣76:1␣83:1␣
```

Soubor s textovou reprezentací modelu obsahuje nejprve parametry modelu a následovány podpůrnými vektory ve stejném formátu jako je tomu u testovacích/trénovacích vektorů v jejich vstupních souborech.

Parametry modelu jsou uloženy v následujícím formátu, přičemž na pořadí řádků (kromě řádku obsahujícího `SV`, který se musí být posledním řádkem parametrů) nezáleží:

```
svm_type␣<svm>
kernel_type␣<kernel_function>
gamma␣<gamma_value>
nr_class␣<classes>
total_sv␣<support_vectors>
rho␣<rho_value>
degree␣<deg_value>
coef0␣<coef0_value>
label␣<class_1_label>␣<class_2_label>␣...
nr_sv␣<class_1_nr_sv>␣<class_2_nr_sv>␣...
SV
```

Následuje tabulka s popisem jednotlivých parametrů:
Parametry `gamma <gamma_value>`, `degree <deg_value>` a `coef0 <coef0_value>` jsou v souboru obsaženy v závislosti na použité kernel funkce při trénování modelu. Jejich výskyt popisuje následující tabulka:

`<svm>`	Typ SVM
`<kernel_function>`	Algoritmus použitý při trénování
`<gamma_value>`	Hodnota parametru γ (reálné číslo)
`<classes>`	Počet tříd podpůrných vektorů
`<support_vectors>`	Celkový počet podpůrných vektorů
`<rho_value>`	Hodnota parametru ϱ (reálné číslo)
`<deg_value>`	Hodnota parametru *degree* (celé číslo)
`<coef0_value>`	Hodnota parametru $coef0$ (reálné číslo)
`<class_n_label>`	Označení n-té třídy
`<class_n_nr_sv>`	Počet podpůrných vektorů v n-té třídě

Tabulka 7.1: Významy parametrů souboru s modelem klasifikátoru

Kernel funkce	Nutné parametry
Lineární	*žádný*
Polynomiální	`degree`, `gamma`, `coef0`
RBF	`gamma`
Tangenciální	`gamma`, `coef0`

Tabulka 7.2: Povinné parametry jednotlivých kernel funkcí

Knihovna libSVM používá pro uložení načtených souborů strukturu obsahující pouze nenulové prvky, proto zde není problém s načtení řídkých dat ze souboru. Oproti tomu všechny implementace v jazyce CUDA C požadují na vstupu hustá data a proto je zde nutné nejprve zjistit dimenze a počet všech vektorů před jejich samotným načtením do paměti, zde již v husté reprezentaci.

Knihovna libSVM přímo obsahuje funkce, které vstupní soubory s trénovacími/testovacími daty nebo natrénovaným modelem načtou a jediným úkolem je tuto funkce zavolat se správnými parametry.

Knihovna cuSVM neobsahuje žádné vstupně výstupní funkce z důvodu jejího účelu sloužit jako funkce Matlabu, kde byl brán předpoklad, že veškerá data jsou již načtena a budou sloužit jako vstupní parametry (vektory nebo matice). Kvůli tomu byl naprogramován algoritmus, který načte a připraví data do stejného formátu v jakém by je daná funkce dostala od Matlabu.

Knihovna gpuSVM obsahuje funkce pro načítání dat, avšak ty jsou schopny

správně načíst pouze soubory obsahující hustou reprezentaci všech vektorů. Je to dáno tím, že není brán ohled na index, ale pouze se počítá počet dvojic (index:hodnota) pro první vektor, který určí počet prvků vektoru. tato dimenze je pak použita jak pro alokaci, tak pro adresování jednotlivých vektorů a pokud první vektor nemá nejvyšší počet prvků ze všech vektorů, může se stát, že se program pokusí zapsat na místo, které není alokováno, což může vyústit v chybové ukončení celé aplikace. Z tohoto důvodu byla pro knihovnu gpuSVM použita stejná funkce pro načítání dat jako je tomu u cuSVM.

Načítání souborů sestává ze 4 částí:

- Načtení parametrů (pouze při načítání souboru s modelem)
  Po úspěšném otevření souboru se testují řetězce končící mezerou na začátku každé řádky a na základě tohoto řetězce se nastaví příslušný parametr(y) na hodnotu/y umístěné za mezerou (v případě více hodnot jsou ty odděleny mezerou).

- Načtení všech vektorů a jim příslušejícím hodnotám α ze souboru v textové formě do zásobníku
  Zde se alokuje zásobník, a postupně se do něj zapisují veškerá data příslušející (trénovacím, podpůrným nebo testovacím) vektorům a jim příslušné hodnoty α v textové podobě. Zásobník se vždy zaplní daty a pokud není uložen celý soubor, je jeho kapacita zdvojnásobena. V případě že se nepodaří velikost zdvojnásobit se dále pokračuje pouze s využitím zásobníku, který udržuje v paměti jeden řádek souboru. V opačném případě, kdy je soubor úspěšně načten do zásobníku se dále pracuje přímo se zásobníkem a odpadá nutnost číst data z pevného disku.

- Určení nejvyšší dimenze a počtu vektorů
  Zde se testuje pouze poslední dvojice na každém řádku, protože právě ta obsahuje nejvyšší index. Při sčítání počtu řádků se postupně hledá maximum ze všech přečtených indexů a to je prohlášeno za dimenzi vektorů. Celkový počet řádek je poté roven počtu vektorů a taktéž dimenzi vektoru $\boldsymbol{\alpha}$.

- Načtení nenulových hodnot vektorů a hodnot vektoru $\boldsymbol{\alpha}$
  Nakonec se opětovně projde celý zásobník (vstupní soubor) a postupně

se načítají hodnoty vektoru $\boldsymbol{\alpha}$ a hodnoty vektorů příslušející přiřazenému indexu.

Pro vytvářenou knihovnu cudaSVM jsou použity stejné funkce pro načítání dat ze souborů jako u gpuSVM resp. cuSVM.

7.2 Výstup

Pro trénovací algoritmus je výstupem natrénovaný model popsaný v předchozí kapitole.

Výstupem klasifikátoru je soubor obsahující výsledná zařazení do třídy pro testovací vektory, přičemž každý řádek výstupního souboru odpovídá testovacímu vektoru na stejném řádku v souboru, ve kterém je uložen.

Knihovna libSVM a knihovna gpuSVM obsahují jak funkci sloužící pro zapsání natrénovaného modelu do souboru, tak i funkci pro uložení výsledků klasifikace. Zde tedy není potřeba nic dalšího řešit.

Knihovna cuSVM neobsahuje, kvůli jejímu předpokládanému použití jako knihovny Matlabu, funkce pro uložení modelu ani výsledku klasifikace, proto je zde potřeba tyto funkcionality zajistit naprogramováním příslušných funkcí.
Pro vytvářenou knihovnu cudaSVM jsou použity stejné funkce pro ukládání výsledků do souborů jako u cuSVM.

7.3 Vlastní implementace – cudaSVM

7.3.1 Trénování

Vlastní implementace trénování SVM klasifikátoru (cudaSVM) využívá modifikovaný algoritmus SMO s Keerthi úpravami. Modifikace SMO spočívá v tom, že není použita vnitřní smyčka, která hledá pro jeden obraz zvolený ve vnější smyčce vhodný protějšek ke společné optimalizace. Vše je prováděno pouze ve vnější smyčce, kde se po volbě prvního obrazu zvolí nejvhodnější druhý obraz a provede se krok SMO. Dále se pokračuje opět volbou prvního obrazu. Tato úprava je použita kvůli implementaci na GPU, kde není problém (paralelně) porovnat daný vektor se všemi ostatními. Přičemž je

nutné zdůraznit, že tato úprava vede k problému s trénovacími množinami obsahující duplikáty obrazů. Při pokusu o nalezení optimálního protějšku pro optimalizaci s tímto obrazem dojde k volbě právě onoho duplikátu. Následný optimalizační krok však neprovede žádnou změnu, protože oba obrazy jsou shodné, a takto se pokračuje dokud se nedosáhne maximálního povoleného počtu iterací, kdy se algoritmus násilně ukončí. Výsledný model pak obsahuje pouze vektory, které byly zpracovány než došlo ke zmíněnému „zacyklení“ a není tedy validní.

Další významná změna oproti jiným implementacím je v inicializaci cache matice **Q**. Tato cache se naplní prvními n řádky matice **Q**, které se vejdou do paměti GPU. Pokud je tento krok proveden napřed, a nikoliv až ve chvíli, kdy je daný vektor potřeba (což vyžaduje přerušení trénovací smyčky a spuštění výpočtu daného řádku matice **Q**), lze při samotné optimalizace SMO ušetřit velké množství času. V případě rozsáhlé množiny trénovacích dat a / nebo malé paměti GPU tento způsob velmi pravděpodobně zhorší čas trénování tím, že většina z předpočtených řádek bude přepsána dřív než bude využita. Avšak u takových úloh bude čas provádění poměrně dlouhý a tudíž se tento problém nemusí znatelně projevit.

7.3.2 Klasifikace

Pro klasifikaci (predikci) je použita implementace, kdy se pomocí vlastní kernel funkce provede rozřazení testovacích vektorů. Vlastní implementace nespoléhá na knihovnu Cublas, používanou většinou ostatních autorů klasifikátorů (např. cuSVM, gpuSVM), což umožňuje naprogramovat právě jednu funkci, která provede přesně to co je potřeba. Cublas je schopen provádět velmi efektivně maticové operace (které lze při klasifikaci SVM klasifikátorem použít) a využít maximální potenciál dané GPU, avšak jeho inicializace může (v případě nízkého počtu testovacích vektorů a podpůrných vektorů v modelu) spotřebovat více času než samotný výpočet. Implementace cudaSVM však ani po významné optimalizaci není schopna dosáhnout na efektivitu knihovny Cublas, což sice není očekávaný výsledek, ale klasifikace je u SVM klasifikátorů jednodušší úloha a není potřeba na ni vynakládat přílišné úsilí.

8 Experimenty

Kvůli dříve zmíněným vlastnostem obou implementací byl pro experimenty zvolen následující hardware

Procesor	Intel Core 2 Duo E6850 @ 3.0 GHz
Základní deska	Asus P5K Pro
Operační paměť	6GB DDR2 800 (2×2GB + 2×1GB)
Grafické karty	NVIDIA GTX260 (216 CUDA jader, 896 MB GDDR3, PCI Express 1.1 x4) NVIDIA GTX480 (480 CUDA jader, 1536 MB GDDR3, PCI Express 1.1 x16)
Operační systém	Windows 7 64-bit SP1

Tabulka 8.1: Testovací sestava

Obě grafické karty NVIDIA GTX480 bude použita jak pro trénování, tak pro klasifikaci. Grafická karta NVIDIA GTX260 je používána pouze pro zobrazování. Aby byla při testování programů zaručena správná volba GPU, bude tak nastaveno v ovladači (verze 335.23). Budou porovnávány jak mplementace na GPU proti libSVM (CPU), tak implementace v CUDA C proti sobě na obou platformách.

U všech implementací bude měřen čas načtení dat ze souborů, doba potřebná pro trénování, doba potřebná pro uložení výsledků a celkový čas. Nejdůležitějším údajem je samozřejmě doba strávená při trénování, ale kvůli testování implementace GPU-Accelerated LIBSVM (kde není možnost tyto časy od sebe oddělit) bude zaznamenán i celkový čas běhu.

Některé části obou implementací musely být změněny z důvodu testování za odlišných podmínek než bylo plánováno autory.

Implementace	Změna kódu	Zdůvodnění
cuSVM	Nahrazení knihovny Matlabu `mex.h` za vlastní, kde jsou všechny použité *mex* funkce a parametry nahrazeny jejich verzemi v jazyce C.	Funkce Matlabu nejsou žádoucí z důvodu použití implementace pouze v C.
gpuSVM	Přejmenování objektu `Cache`.	Kolize s implementací objektu stejného jména v LibSVM.
	Modifikace programu tak, aby nepotřeboval knihovny specifické pro překladač GCC (konkrétně `sys/time.h`).	Implementace byla překládána pomocí Microsoft Visual Studio 2008 na platformě Windows
	Modifikace funkce `Cache::findData()`, kde některé parametry daného objektu nebyly správně inicializovány.	Překladač Microsoft Visual Studia 2008 neinicializuje objekty za programátora a při jejich použití kontroluje, zda byl daný objekt úspěšně inicializován.

Tabulka 8.2: Modifikace použitých implementací

Bohužel modifikace funkce `Cache::findData()` je velice nestabilní, a přestože implementace gpuSVM je nejrychlejší zde testovaná, budou experimenty spuštěny pouze pro klasifikační část této knihovny, protože tam není objekt `Cache` (ani jeho funkce) použit.

Dále bylo pro knihovnu cuSVM naprogramováno načítání a ukládání souborů s testovacími (popř. trénovacími) vektory a modelem. Pro cuSVM i gpuSVM byla dále přidána funkce, která ukládá natrénovaný model do textového souboru se stejným formátem, který používá LibSVM. Obě knihovny byly upraveny pro kompatibilitu s novějšími architekturami, což bylo nezbytné pro testování na GTX480.

8.1 Trénování

Trénování bylo prováděno s následujícími parametry:

Parametr	Hodnota	Popis
SVM kernel	RBF	
γ	$\frac{1}{m}$	Parametr SVM kernel funkce (m je počet trénovacích vektorů)
eps	0,001	Zastavovací podmínka (tolerance ztrátové funkce).

Tabulka 8.3: Hodnoty trénovacích parametrů

Výsledky pro trénování modelu z dat `alpha` (počet trénovacích obrazů: 1024, 8192 a 32768; $\gamma = 0{,}002$) při $C = 0{,}015625$:

Knihovna	Podpůrné vektory (třída$_{-1}$; třída$_{+1}$)	Hodnota prahu (ρ)	Čas [s]	
			Trénování	Celkem
LibSVM	1024 (508; 516)	0,984902	2,4830	3,3777
gpu-libSVM	1024 (508; 516)	0,984902	-	1,8341
cudaSVM	1024 (508; 516)	0,984501	0,0797	0,6301
cuSVM	1024 (508; 516)	0,984419	0,7060	1,2527
LibSVM	8192 (4083; 4109)	0,980778	133,4699	140,5803
gpu-libSVM	8192 (4083; 4109)	0,980791	-	78,6609
cudaSVM	8192 (4083; 4109)	0,980475	1,0096	5,3762
cuSVM	8192 (4083; 4109)	0,980595	4,7325	9,0934
LibSVM	32768 (16434; 16334)	−0,982719	2199,2856	2227,8748
gpu-libSVM	32768 (16434; 16334)	−0,982704	-	1201,7167
cudaSVM	32768 (16434; 16334)	−0,982415	38,5742	56,152
cuSVM	32768 (16434; 16334)	−0,982413	30,3581	48,0376

Tabulka 8.4: Výsledky trénování dat `epsilon_normalized` při $C = 0{,}015625$

Z výše uvedené tabulky plyne, že implementace, které využívají GPU (cudaSVM, cuSVM) na většinu výpočtů při trénování mají oproti CPU implementaci (LibSVM) i implementaci částečně akcelerovanou pomocí GPU (gpu-LibSVM) výrazný časový náskok. Knihovna gpu-LibSVM je zde přibližně 2x rychlejší než referenční LibSVM, avšak rozdíly ve výsledném prahu (ρ) jsou zanedbatelné. Práh vypočtený implementacemi na GPU je mírně

odlišný, což je způsobeno použitím jednoduché přesnosti při některých výpočtech, nicméně urychlení trénování je v řádech desítek.
Při trénování modelu z dat `alpha` (počet trénovacích obrazů: 1024; $\gamma = 0{,}002$) při $C = 1$ jsou výsledky následující:

Knihovna	Podpůrné vektory (třída$_{-1}$; třída$_{+1}$)	Hodnota prahu (ρ)	Čas [s]	
			Trénování	Celkem
LibSVM	1024 (508; 516)	0,00635	5,1594	6,0627
gpu-libSVM	1024 (508; 516)	0,00634941	-	1,8203
cudaSVM	1024 (508; 516)	0,00598214	0.134	0,6959
cuSVM	1024 (508; 516)	0,00594837	1,2377	1,7855

Tabulka 8.5: Výsledky trénování dat `epsilon_normalized` při $C = 0{,}015625$

Pro vyšší hodnotu parametru C provedl SMO algoritmus více iterací a tudíž je čas spotřebovaný inicializací nižší, což vede k nárůstu rychlosti gpu-libSVM implementace na 3× oproti LibSVM.

8.2 Klasifikace

Pro klasifikační experimenty byl použit model vytvořený LibSVM knihovnou a byl porovnán se všemi implementacemi klasifikátoru.

Při klasifikace LibSVM modelu (pro počet podpůrných vektorů 1024 a $C = 1$) oproti testovacím datům (počet testovacích obrazů: 100 000) jsou výsledky následující:

Knihovna	Úspěšnost klasifikace [%]	Čas klasifikace [s]
LibSVM	51,929	195,8202
cudaSVM	51,982	1,0929
cuSVM	51,929	0,8369
gpuSVM	51,929	0,4956

Tabulka 8.6: Výsledky trénování dat `epsilon_normalized`

Doba klasifikace se za pomoci GPU výrazně zkrátí, protože klasifikace je vpodstatě jen násobení matic, na které je GPU architektura velmi vhodná. Chyba vlastní implementace (cudaSVM) je způsobena zaokrouhlovacími chybami.

9 Zhodnocení výsledků

Trénování vlastní implementace (cudaSVM) bylo ve většině případů podstatně rychlejší. Výjimky tvořily množiny dat, které byly jen o málo větší než aby je bylo možné uložit do paměti GPU, protože v takových případech bylo do cache **Q** zbytečně uloženo velké množství dat. Tento problém postupně mizí s dále vzrůstající dimenzí, kdy už cache obsahuje pouze velmi malou část celé matice a tudíž není zpoždění při předvyplnění výrazné.

Zrychlení implementace GPU-Accelerated LIBSVM (týká se pouze trénování) není tak výrazné jako je tomu u ostatním implementací, které využívají GPU k většímu množství z potřebných výpočtů. Na druhou stranu, je toto řešení plně funkční a lze ho okamžitě nasadit bez nutnosti upravovat kód tak, aby vyhovoval novým architekturám (jako obě ostatní CUDA C implementace) nebo dokonce řešit chyby a nedokonalosti celého návrhu implementace klasifikátoru (gpuSVM).

10 Závěr

Úkolem této práce bylo implementovat SVM klasifikátor v programovacím jazyce CUDA C, který využívá paralelního výkonu grafických karet, a porovnat ho s jinými volně dostupnými implementacemi.

Podařilo se naprogramovat vlastní plně funkční klasifikátor, který má srovnatelné výsledky s balíkem LIBSVM, jenž je považován za standard. Dále byly výsledky a rychlost porovnány s třemi implementacemi využívající jazyk CUDA C, konkrétně se jednalo o cuSVM (Carpenter), gpuSVM (Catanzaro) a GPU-Accelerated LIBSVM (A. Athanasopoulos, A. Dimou, V. Mezaris, I. Kompatsiaris).

U implementace gpuSVM byly řešeny problémy se špatnou / chybějící inicializací proměnných a s rozhraním, kde bylo velmi nevhodně řešeno načítání dat ze souboru, což v některých případech způsobovalo načtení nevalidních dat nebo pád programu. Klasifikátor knihovny gpuSVM byl úspěšně zprovozněn, avšak trénovací algoritmus vyžaduje velkou míru nestability, jejíž příčina nebyla nalezena. Pro cuSVM implementaci bylo pouze nutné doprogramovat rozhraní pro načítání souborů, protože původní funkčnost byla jako zásuvný modul do Matlabu naprogramovaný v C.

Implementace v programovacím jazyce CUDA C umožňuje velmi výrazně snížit dobu trénování i následné klasifikace. Což je důležité pro tzv. „cross-validation“, při které se spouští trénování mnohokrát a různými parametry a zjišťuje se, které hodnoty jsou pro daná data nejvhodnější.

Literatura

[1] NVIDIA Corporation, *CUDA C Programming Guide* http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, NVIDIA Corporation, 2012

[2] Mark Harris, NVIDIA Developer Technology : *Optimizing Parallel Reduction in CUDA* http://developer.nvidia.com/sites/default/files/akamai/cuda/files/reduction.pdf

[3] Prof. Ing. Psutka Josef, CSc., *Učící se systémy a klasifikátory*. Plzeň: Západočeská univerziza v Plzni, 2011

[4] Taku Kudo and Yuji Matsumoto : *Chunking with Support Vector Machines* http://acl.ldc.upenn.edu/N/N01/N01-1025.pdf

[5] John C. Platt : *Fast Training of Support Vector Machines using Sequential Minimal Optimization*, Microsoft Research, 2000

[6] S.S. Keerthi : *Improvements to Platt's SMO Algorithm for SVM Classifier Design*, Singapore: National University of Singapore, 2002

[7] Wikipedia contributors, *Interior point method*, Wikipedia, The Free Encyclopedia, 12 May 2014, 13:15 UTC, http://en.wikipedia.org/w/index.php?title=Interior_point_method&oldid=608219819 [accessed 15 May 2014]

[8] Christopher J.C. Burges : *A Tutorial on Support Vector Machines for Pattern Recognition* http://research.microsoft.com/pubs/67119/svmtutorial.pdf

[9] Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1--27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[10] A. Athanasopoulos, A. Dimou, V. Mezaris, I. Kompatsiaris, *GPU Acceleration for Support Vector Machines*, Proc. 12th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, The Netherlands, April 2011. https://github.com/MKLab-ITI/CUDA

[11] Wikipedia contributors, *Comparison of Nvidia graphics processing units*, Wikipedia, The Free Encyclopedia, 1 July 2013, 20:46 UTC, http://en.wikipedia.org/w/index.php?title=Comparison_of_Nvidia_graphics_processing_units&oldid=562448052 [accessed 3 July 2013]

[12] Wikipedia contributors, *Curse of dimensionality*, Wikipedia, The Free Encyclopedia, 15 May 2013, 19:34 UTC, http://en.wikipedia.org/w/index.php?title=Curse_of_dimensionality&oldid=555258440 [accessed 3 July 2013]

Přílohy

A Obsah přiloženého média

Zdrojové soubory trénovacího a klasifikačního benchmarku jsou spolu se skriptem pro vygenerování projektu pomocí programu *CMake* uloženy na přiloženém médiu.
Dále médium obsahuje trénovací a testovací data a elektronickou verzi této dokumentace.