Perceptron na GPU

B4M39GPU 2019/2020 zimní semestr

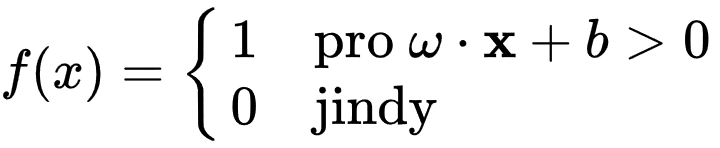
Jan Skala

# Abstrakt

Semestrální práce se zaměřuje na paralelní implementaci Perceptronu. Zahrnuje implementaci trénování a klasifikace dat. Obsahuje naivní CPU implementaci a paralelní GPU implementaci. Cílem je zjistit, zda se dosáhne, a pokud ano tak jakého, zrychlení výpočtu při použití grafické karty a paralelní implementace.

# Popis algoritmu

Perceptron je nejjednodušší model dopředné neuronové sítě. Sestává se pouze z jednoho neuronu a byl vynalezen Frankem Rosenblattem v roce 1957. Jedná se o binární klasifikátor, který mapuje vektor sloupců x na výstupní hodnoty f(x).



Rovnice Klasifikace vektoru vstupů x

Kde **ω** je vektor vah a **b** je konstanta(někdy se jí také říká bias nebo práh) a **x** je vektor vstupních dat. Pro jednodušší implementaci se místo 0 vrací -1.

Hlavní nedostatek perceptronu spočívá v tom, že aby správně fungoval musí být data lineární separovatelná. Někdy lze tento nedostatek řešit částečnou úpravou vstupních dat. Máme-li dvourozměrná data a je patrná že existuje kružnice, která by data oddělila, můžeme data převést do polárních souřadnic. Dalším rozšířením je vícevrstevný perceptron, který má již mnohem širší možnosti použití. (Mařík, 1993)

## Trénování

V této práci bylo využito učení s učitelem. Síti byla nejdříve prezentována tréninková množina, která odpovídá správnému chování. Trénování probíhá v iteracích. Jejich počet můžeme stanovit. V každé iteraci je aktualizován vektor vah a bias. Zároveň pracujeme s konstantou **r**,která značí learning rate (rychlost učení, ale typicky se tento pojem nepřekládá).Dosazením menších hodnot docílíme pomalejšího učení, tzv fine tuning. Dosazením větších hodnot zrychlíme proces učení, ale riskujeme, že optimum nespočítáme správně. Hodnoty jsou v rozmezí od 0 do 1.

Nejprve je zvolen náhodný vektor vah (může se jednat o nulový vektor). Každý vektor vstupních dat klasifikujeme a porovnáme s očekávaným výsledkem. Pokud se výstupy neshodují provedeme upravíme váhy a bias. Učící algoritmus u perceptronu je konečný, pouze pokud řešení existuje, a navíc pokud je učící množina lineárně separovatelná.

Váhy perceptronu a bias jsou adaptovány dle následujícího pravidla:

Skoková přenosová funkce

Skoková přenosová funkce vrací pro vstup menší než daná mez -1, pro větší vrací 1.

## Klasifikace

Klasifikace spočívá ve vynásobení vektoru s daty (D) vektorem vah(W) skalárně. Následně se přičte bias (b). Výsledek se použije jako argument skokové funkce (f).

# Struktura projektu

* Data – složka obsahuje trénovací a testovací Datasety
* CL11.hpp – Header only knihovna pro zpracovávání vstupních argumentů
* CSV.cpp/h – pomocná třída, slouží k načítání CSV dat do struktury std::vector
* Debug.h – obsahuje skupinu funkcí pro debugování – vypisování dat z GPU do konzole
* default.h – obsahuje definici assert makra pro spouštění CUDA funkcí
* kernels.cu – implementace všech kernelů
* kernels.cuh – hlavičky a implementace spouštěcí funkcí pro kernely
* main.cpp – hlavní vstup programu – načítání argumentů a spuštění perceptronu
* Perceptron.cpp/h – Hlavní třída implementující funkcionalitu perceptronu

## Datasety

Všechna data byla převedena do CSV pro snazší načítání. K trénování a testování byly použity následující Datasety:

* Iris
* Minst číslice 1
* Mnist číslice 2
* Mnist číslice 3

# CPU Implementace

CPU implementace je realizovaná ve třídě Perceptron. Konstruktor přijímá na vstupu 3 parametry:

* Počet iterací trénování
* Learning Rate
* Slovní výstup – true pokud požadujeme logování zpět do konzole

## Trénování

Metoda **fit** ve třídě Perceptron. Jako parametry přijímá dva vektory jeden s daty a druhý s očekávanými klasifikacemi. Začíná s nulovými váhami. Celá metoda je implementována sekvenčně není použita žádná paralelizace, ať už v podobě OpenMP nebo SIMD. Vypočítané váhy a bias si uloží do privátních polí.

## Klasifikace

Metoda **predict** ve třídě Perceptron. Skalárně vynásobí vektor vstupních dat s vektorem vah, následné přičte bias a na výsledek aplikuje skokovou přenosovou funkci.

# GPU Implementace

GPU implementace je realizovaná ve třídě Perceptron. Konstruktor přijímá na vstupu 3 parametry:

* Počet iterací trénování
* Learning Rate
* Slovní výstup – true pokud požadujeme logování zpět do konzole

Zároveň využívá CUDA kernely a jejich pomocné a spouštěcí funkce.

# Použité Kernely

Ke každému kernelu je definovaná pomocná funkce, která provádí alokaci a volá kernel. Třída Perceptron tak nemusí volat kernel přímo, nemusí provádět alokaci na GPU a vývojové prostředí Visual Studio nehází chybová hlášení o nevalidní syntaxi při používání streamové notace <<<,>>>.

## Skalární součin

Skalární součin je jedna z esenciálních operací, která je použita jak při trénování, tak při klasifikaci. Implementoval jsem jej jako device funkci, kterou mohu volat z více kernelů. Nejprve vynásobím příslušné složky obou vektorů a uložím data do sdílené paměti. Následně data se sdílené paměti sečtu pomocí paralelní redukční sumy. Tuto funkci volají dva kernely. K\_dot, který dělá obyčejný skalární součin. A k\_update, který po vektorovém součinu, ještě počítá chybu klasifikace a úpravu hodnotu proměnné bias.

## Update

Nejprve skalárně vynásobí vstupní data a váhy (device funkce d\_dot). K výsledku přičte bias a aplikuje skokovou přenosovou funkci. Výstupem je proměnná **u** (update), která je použita v dalším kernelu. Bohužel nelze paralelizovat na úrovni dat, protože pro trénování na datech di se využívají výsledky z trénování dat di-1.

Kde u značí proměnnou **u** (update)  
**p** značí learning rate  
**a** značí očekávanou klasifikaci  
**b** značí **bias\_gpu  
Dj**značí řádek trénovacích dat  
**w** značí vektor vah  
**f(x)** značí skokovou přenosovou funkci

## Násobení skalárem

Nyní zbývá korekce vah. Vektor s daty vynásobím skalárem **u**, který jsem spočítal v předchozím kernelu. Výsledek uložím jako nový vektor vah. Následně překopíruji skalár **u** do proměnné **bias\_gpu**.

Kde Wi+1 značí nový vektor vah  
Dj značí řádek s daty, pomocí kterých se provádí trénink

## Klasifikace

Vzhledem k tomu, že dat ke klasifikaci je typicky více než rozměr vah. Například dataset mnist potřebuje 784 vah, zatímco dat je více než 6 tisíc. V tomto kernelu dává smysl paralelizovat na úrovni dat. Každé vlákno tedy provede skalární součin řádku dat a vektoru vah. Následně přičte bias a spočítá hodnotu skokové přenosové funkce.

# Naměřené výsledky

CPU

GPU 1

GPU 2

# Závěr

# Reference

Kelleher, C. (nedatováno). *curran.* Načteno z Github: https://gist.github.com/curran/a08a1080b88344b0c8a7

Mařík, V. (1993). *Umělá inteligence I.* Praha: Academia.

Redmon, J. (nedatováno). *projects/mnist-in-csv*. Načteno z Mnist dataset CSV: https://pjreddie.com/projects/mnist-in-csv/