FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Vývoj paralelních algoritmů pro strojové učení na GPU
Student:	Šimon Bařinka
Vedoucí:	Ing. Tomáš Oberhuber, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

1. Seznamte se se základy strojového učení, zejména neuronových sítí a algoritmu backpropagation [2] pro učení neuronových sítí.
2. Seznamte se s návrhem knihovny TNL [1] a jejím využitím při vývoji paralelních algoritmů.
3. Nastudujte způsoby implementace algoritmu backpropagation na GPU a naprogramujte ho za pomocí základních algoritmů a datových struktur z knihovny TNL.
4. Otestujte implementovaný algoritmus pro jednoduchou klasifikaci bodů v prostoru.

Seznam odborné literatury

[1] www.tnp-project.org
[2] Charu C. Aggarwal, Neural Networks and Deep Learning, Springer, 2018.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 15. února 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Vývoj paralelních algoritmů pro strojové učení na GPU

Šimon Bařinka

Katedra teoretické informatiky
Vedoucí práce: Ing. Tomáš Oberhuber, Ph.D.

8. června 2020

Poděkování

Tímto bych rád poděkoval vedoucímu své práce Ing. Tomáši Oberhuberovi, Ph.D., za ochotu, trpělivost a cenné rady v průběhu realizace mé bakalářské práce. Také bych rád poděkoval své rodině a přítelkyni za plnou podporu během svého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 8. června 2020 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2020 Šimon Bařinka. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Bařinka, Šimon. *Vývoj paralelních algoritmů pro strojové učení na GPU*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

// Abstrakt

Tato práce se zaměřuje na vývoj paralelních algoritmů pro strojové učení na GPU. V práci je podrobně popsána architektura umělých neuronových sítí a vybrané algoritmy tradičního strojového učení. Dále je popsán vývoj knihovny pro snadnou konstrukci umělých neuronových sítí a jejich následné paralelní učení na GPU nebo CPU pomocí algoritmu backpropagation. Součástí je testování implementované knihovny na nejznámějších veřejných datasetech.

Klíčová slova strojové učení, neuronové sítě, backpropagation, CUDA, grafické karty, TNL

Abstract

This thesis focuses on a development of parallel algorithms of machine learning executing on a GPU. It describes neural network architecture in a great detail as well as selected models of the conventional machine learning. Furthermore, it describes development of a library for simple construction and consecutive parallel training on a GPU or CPU, using backpropagation algorithm. Eventually, the implemented library is tested on the most popular public datasets.

Keywords machine learning, neural networks, backpropagation, CUDA, graphics cards, TNL

Obsah

Úvod **1**
Motivace 1
Cíle práce 2
Členění práce 2
Notace 3

1 Architektura umělých neuronových sítí **5**
1.1 Biologický původ a historie 5
1.2 Perceptron 6
1.2.1 Gradientní sestup 8
1.3 Vícevrstvé neuronové sítě 10
1.3.1 Vhodné aktivační funkce 10
1.4 Proces učení 12

2 Úvod do strojového učení **15**
2.1 Lineární regrese 16
2.2 Logistická regrese 17
2.3 Metoda podpůrných vektorů 19

3 Backpropagation **21**
3.1 Výpočetní graf 21
3.1.1 Řetízkové pravidlo ve výpočetním grafu 22
3.1.2 Řetízkové pravidlo a dynamické programování 23
3.2 Průběh algoritmu 24

4 Implementace knihovny **25**
4.1 Použité technologie 25
4.2 Implementace backpropagation 25
4.2.1 Další možnosti paralelizace 28

4.3 Struktura knihovny 30

5 Testování **35**
5.1 Prostředí 35
5.2 Datasety 35
5.2.1 MNIST 36
5.2.2 CIFAR-10 36
5.3 Výsledky 36
5.4 Závěr testování 38

Závěr **39**
Možnosti budoucí práce 39

Bibliografie **41**

A Obsah přiloženého flash disku **43**

Seznam obrázků

1.1 Umělý neuron. 6
1.2 Model perceptronu. 6
1.3 Gradientní sestup funkce $f(x)$ začínající v bodě x_0. 9
1.4 Vícevrstvá neuronová síť. Převzato z [2]. Přeloženo autorem. . . . 10
1.5 Grafy pro různé aktivační funkce. Převzato z [2]. 11
1.6 Ukázka použití softmax funkce. Převzato z [2]. Přeloženo autorem. 12

2.1 Závislost neuronových sítí a tradičního strojového učení na množství dostupných dat. Převzato z [2]. Přeloženo autorem. 15
2.2 Neuronová síť reprezentující model lineární regrese. 17
2.3 Neuronová síť reprezentující model logistické regrese. 18

3.1 Výpočetní graf pro funkci $f(x, y)$. 21
3.2 Výpočetní graf pro funkci $f(g(z(x, y)))$. 22
3.3 Výpočetní graf pro funkci $f(z(g(x)), k(g(x)))$. 22
3.4 Znázornění řetízkového pravidla pro funkci $f(z(g(x)), k(g(x)))$ pomocí výpočetního grafu. 23
3.5 Znázornění výpočetní redundance, při naivním algoritmu. 23

4.1 Znázornění rozdělení neuronu na složku lineární a aktivační. 26
4.2 Způsob uložení vstupů, gradientů a parametrů uvnitř vrstvy. . . . 27
4.3 Sada abstraktních tříd sloužící pro implementaci vrstev. 34

Seznam tabulek

4.1 Definice chování pro jednotlivé vrstvy v dopředném chodu a zpětném chodu. 27

5.1 Výsledky testování MNIST 1. 37
5.2 Výsledky testování MNIST 2. 37
5.3 Výsledky testování MNIST 3. 37
5.4 Výsledky testování CIFAR-10 1. 38

Úvod

„Namísto vytváření programu, který simuluje mysl dospělého člověka, proč raději nezkusit vytvořit takový, který simuluje mysl dítěte? Pokud bychom takový program následně podrobili vhodnému vzdělání, výsledkem by byl mozek dospělého.“

—Alan Turing [1]

Motivace

Oblast strojového učení na sebe v posledních letech strhává nebývalé množství pozornosti. V první řadě je to důsledek neustále narůstající dostupnosti informací a dat, kterou s sebou přinesl rozmach osobních počítačů a Internetu. V dnešním světě jsou každé kliknutí, nákup i virtuální konverzace ukládány a archivovány. Takto shromažďovaná data slouží jako vstup pro algoritmy a modely strojového učení, což vytváří ideální podmínky pro jejich rozvoj.

Významnou podmnožinou strojového učení jsou *umělé neuronové sítě*. Jsou to struktury, které jsou lépe přizpůsobeny pro paralelní zpracování dat, než tradiční algoritmy strojového učení. Aktuální podmínky dovolují rozsáhlý rozvoj těchto sítí, které jsou v současnosti aktivní oblastí výzkumu. Neuronové sítě v dnešní době dokáží řídit auto nebo vést smysluplnou konverzaci v podobě chytrých společníků uvnitř mobilních zařízení, a dá se tak očekávat, že v budoucnosti najdou ještě širší uplatnění v každodenním životě.

Množství dostupných dat postupně překonává možnosti běžných centrálních procesorových jednotek (z anglického *central processing unit*, zkráceně CPU), což mělo za následek experimentování se specializovanými procesory, jakými jsou například grafické karty, neboli *grafické procesorové jednotky* (z anglického *graphics processing unit*, zkráceně GPU). Grafické karty jsou přizpůsobeny pro velké množství menších paralelních úkolů, které se opakují. Toho dokáže neuronová síť efektivně využít, ovšem samotné psaní programu

pro grafickou kartu je v mnoha směrech odlišné od programování pro klasický procesor a často je i časově náročnější.

Cíle práce

Hlavním cílem této práce je implementace knihovny, která poskytne jednoduché rozhraní pro konstrukci umělé neuronové sítě a její následné paralelní učení a predikci na CPU nebo GPU. Tato knihovna odstíní uživatele od komplikací spojených s psaním kódu pro GPU a algoritmů souvisejících s učením neuronových sítí. Zároveň umožní snadné rozšíření a dosazení prvků naprogramovaných uživatelem.

Implementace bude otestována na několika veřejně dostupných datasetech. Výsledky těchto testů budou diskutovány jak z pohledu rychlosti učení, tak z pohledu generalizace a přesnosti predikce na datech, které nebyly k učení použity.

Dalším úkolem práce je popis implementace učení neuronových sítí na GPU a diskuze nad možnostmi paralelizace učení.

V neposlední řadě tato práce shrne základy strojového učení, zejména pak neuronových sítí. Modely strojového učení budou popsány v souvislosti s modely neuronových sítí, neboť modely tradičního strojového učení lze často vyjádřit pomocí jednoduchého modelu neuronové sítě, čehož bude práce využívat.

Členění práce

V úvodů práce byl vysvětlen vztah mezi tradičním strojovým učením a neuronovými sítěmi. Byly nastíněny důvody nedávné zvýšené popularity neuronových sítí a téma využití grafických karet, které bude dále rozvinuto v jedné z následujících kapitol. Dále byl ujasněn výstup a cíl práce.

První kapitola je věnována detailnímu popisu neuronových sítí. Část této kapitoly zahrnuje úvod k samotnému algoritmu učení.

Druhá kapitola popisuje několik základních modelů tradičního strojového učení v kontextu neuronových sítí. Účelem práce není popsat tyto modely do detailů, nýbrž jen shrnout základní poznatky.

Třetí kapitola se podrobně věnuje algoritmu učení umělých neuronových sítí.

Čtvrtá kapitola zahrnuje způsob realizace knihovny včetně použitých technologií a návrhových rozhodnutí. V kapitole jsou diskutovány možnosti budoucího rozšíření knihovny. Čtvrtá kapitola je pro tuto práci stěžejní.

Pátá kapitola se věnuje testování funkční implementace. Práce je uzavřena diskuzí nad možnostmi budoucího rozšíření knihovny.

Notace

Notace v této práci je převzata z [2]. Vektor i matice jsou značeny velkými písmeny s šipkou nad nimi $\overrightarrow{X}$. Skalární součin je značen tečkou $\overrightarrow{X} \cdot \overrightarrow{Y}$. Matice s testovacími daty je vždy značena jako $\mathcal{D}$. Tato matice obsahuje n řádků, kde každý řádek obsahuje datový bod o d hodnotách.

Kapitola 1

Architektura umělých neuronových sítí

Tato kapitola objasňuje historii a biologický původ umělých neuronových sítí. Dále do hloubky popisuje architekturu, a také nastiňuje algoritmus učení.

1.1 Biologický původ a historie

Nervový systém člověka obsahuje buňky nazývající se *neurony*. Jednotlivé neurony se v lidském těle spojují pomocí tzv. *axonů* a *dendritů*. Tato spojení se nazývají *synapse*. Síla synaptických spojení se často mění jako reakce na vnější *stimuly*. Díky těmto změnám se člověk učí. [2]

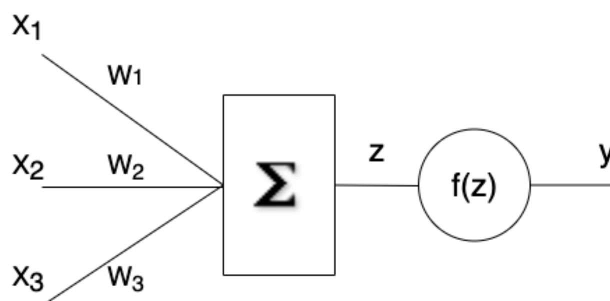
Stejně jako v lidském těle, základní jednotkou umělé neuronové sítě je *neuron*.

Každý vstup do umělého neuronu je nejdříve vynásoben *váhou* a následně předán vnitřní funkci, jejíž výstup je i výstupem samotného neuronu. Umělý neuron může mít více vstupů, v takovém případě jsou vstupy nejdříve vynásobeny jednotlivými váhami, následně sečteny a předány vnitřní funkci. [2] Tato myšlenka je znázorněna na obrázku 1.1.

Model neuronu byl poprvé publikován již roku 1943 v [3], autory byli Warren Mcculloch a Walter Pitts.

Použitím umělých neuronů je možné vyjádřit architekturu modelu neuronové sítě. Umělá neuronová síť počítá funkci několika vstupů. Vstupy jsou postupně propagovány až k výstupním neuronům. K učení dochází pomocí měnění jednotlivých vah patřících jednotlivým neuronům. [2]

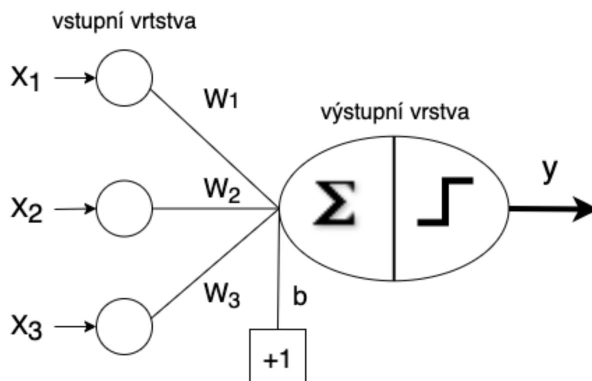
V roce 1958 psycholog Frank Rosenblatt publikoval model *perceptronu*. [4] Jedná se o nejjednodušší instanci umělé neuronové sítě. Avšak v roce 1969 Marvin Minsky a Seymour Papert publikovali odrazující výsledky týkající se reálných schopností perceptronu. [5] Tato publikace měla za následek utlumení výzkumu neuronových sítí v 80. letech. [6]



Obrázek 1.1: Umělý neuron.

1.2 Perceptron

Perceptron je nejjednodušší instancí umělé neuronové sítě. Obsahuje jednu vstupní a jednu výstupní vrstvu. Perceptron je příkladem modelu s jedinou výpočetní vrstvou. Vstupní vrstvu nepočítáme mezi výpočetní vrstvy. [2] Architektura perceptronu je ukázána na obrázku 1.2



Obrázek 1.2: Model perceptronu.

Označme $\mathcal{D}$ jako množinu dat, které slouží pro učení perceptronu. Každý člen této množiny je dvojice $(\overrightarrow{X}, y)$, kde $\overrightarrow{X} = [x_1 \dots x_d]$ obsahuje d *příznaků* a $y \in \{-1, +1\}$ obsahuje *pozorovanou hodnotu vysvětlované proměnné*, která může nabývat dvou hodnot. Pozorovanou hodnotou je myšlena hodnota, která je poskytnuta v rámci trénovacího datasetu. Vstupní vrstva obsahuje d neuronů, které předávají vstupní data $\overrightarrow{X}$ spolu s váhami $\overrightarrow{X} = [w_1 \dots w_d]$ výstupní vrstvě.

Výstupní vrstva vyhodnotí lineární funkci $\sum_{i=1}^{d} w_i x_i$. Výsledek této lineární funkce je předán tzv. *aktivační funkci*. V případě perceptronu se jedná

o *znaménkovou funkci*, jejíž předpis je definován výčtem případů 1.1.

$$sign(x) = \begin{cases} \text{-1} & \text{x} < 0, \\ 0 & \text{x} = 0, \\ 1 & jinak. \end{cases} \tag{1.1}$$

Hodnota aktivační funkce je následovně použita jako predikce vysvětlované proměnné.

Vysvětlovaná proměnná $y \in \{-1, +1\}$ může v případě perceptronu nabývat pouze dvou hodnot. Perceptron je tedy příkladem *binárního klasifikátoru*, který pomocí znaménkové funkce mapuje vstup $\overrightarrow{X}$ na predikci $\hat{y} \in \{-1, +1\}$ následovně:

$$\hat{y} = sign\{\sum_{i=1}^{d} w_i x_i\} \tag{1.2}$$

Ve většině případů je nutné, aby vstup znaménkové funkce v rovnici 1.2 obsahoval i konstantní neměnnou složku. Konstanta je potřeba, pokud je žádoucí, aby aktivační funkce byla *„posunuta"*.

Tato neměnná složka se nazývá *práh* a značí se b. Použitím prahu je například možné dosáhnout situace, kdy výstup znaménkové funkce bude 1 pokud $x > 2$. V takovém případě $b = -2$. [2]

Rovnici 1.2 je třeba změnit následovně:

$$\hat{y} = sign\{\sum_{i=1}^{d} w_i x_i + b\} \tag{1.3}$$

Práh je možné vyjádřit jako neuron, jehož výstup je vždy +1 a je násoben váhou w_0. Tento *„trik"* vede ke zjednodušení zápisu mnoha rovnic, a proto práh nebude v rámci práce explicitně uváděn. Práh je pomocí neuronu znázorněn na obrázku 1.2.

Úkolem umělé neuronové sítě, a tedy i perceptronu, je minimalizovat chybu při predikci. Toho učící algoritmus dosáhne definicí *ztrátové funkce*, kterou v průběhu učení minimalizuje. Ztrátová funkce slouží k měření míry chyby predikce vůči trénovacím datům.

Algoritmus perceptronu je možné definovat pomocí metody *nejmenších čtverců* v rámci všech datových bodů v množině $\mathcal{D}$. S využitím metody nejmenších čtverců je možné optimalizaci ztrátové funkce perceptronu zapsat následovně:

$$\arg\min_{\overrightarrow{W}} L = \sum_{(\overrightarrow{X}, y) \in \mathcal{D}} (y - \hat{y})^2 = \sum_{(\overrightarrow{X}, y) \in \mathcal{D}} (y - sign\{\overrightarrow{W} \cdot \overrightarrow{X}\})^2 \tag{1.4}$$

Volba ztrátové funkce je závislá na výběru aktivační funkce. Na druhou stranu, výběr aktivační funkce je závislý na povaze datasetu a vysvětlované

proměnné. Pokud je vysvětlovaná proměnná reálné číslo, je možné zvolit jako aktivační funkci *identitu* $I(x) = x$. Pokud je vysvětlovaná proměnná binární $y \in \{0, 1\}$, a je žádoucí, aby neuronová síť predikovala pravděpodobnost, že $y = 1$, tak je možné použít *sigmoidu* 1.5 jako aktivační funkci a *negativní logaritmus věrohodnostní funkce* 1.6 jako funkci ztrátovou. Pokud je použita sigmoida, perceptron klasifikuje instanci do třídy 1 pokud $\hat{y} > 0.5$. [2]

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{1.5}$$

$$L(y, \hat{y}) = -\log\left(|y/2 - 0.5 + \hat{y}|\right) \tag{1.6}$$

Požadovanou vlastností aktivační funkce je *nelinearita*, která umožňuje neuronové síti modelovat nelineární vztahy mezi vstupem $\overrightarrow{X}$ a predikcí $\hat{y}$, což dovoluje zachytit komplexnější vztahy. Je dokázáno, že každou vícevrstvou neuronovou síť, která používá pouze *lineární* aktivační funkce, lze vyjádřit pomocí sítě obsahující *jedinou* lineární vrstvu. [7]

Původní myšlenka perceptronu používá jako aktivační funkci znaménkovou funkci. Přestože znaménková funkce může být použita pro predikci, její použití ve fázi učení je značně nepraktické. Povrch ztrátové funkce popsané v 1.4 není kvůli znaménkové funkci hladký a má *„schody"*. Navíc, hodnota v diferencovatelných bodech znaménkové funkce je konstantní v dlouhých souvislých intervalech.

Kvůli těmto vlastnostem není ztrátová funkce 1.4 vhodná pro minimalizaci pomocí metody *gradientního sestupu*.

1.2.1 Gradientní sestup

Pro pochopení gradientního sestupu je nutné definovat *gradient* funkce. Gradient funkce $f : \mathbb{R}^n \mapsto \mathbb{R}$ je *vektorová funkce* $\nabla f : \mathbb{R}^n \mapsto \mathbb{R}^n$, jejíž hodnota v bodě $\overrightarrow{P}$ je vektor, jehož komponenty jsou parciální derivace f v $\overrightarrow{P} = [x_1 \dots x_n]$:

$$\nabla f(\overrightarrow{P}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\overrightarrow{P}) \\ \frac{\partial f}{\partial x_2}(\overrightarrow{P}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\overrightarrow{P}) \end{bmatrix} \tag{1.7}$$

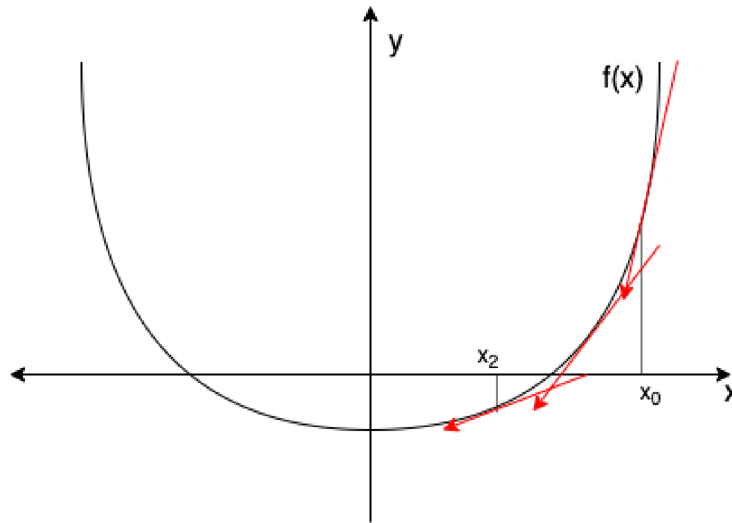
Gradient může být interpretován jako *„směr a velikost nejrychlejšího růstu funkce"*. [8]

Metoda gradientního sestupu je iterativní optimalizační algoritmus pro hledání *lokálního minima*. Využívá gradientu ztrátové funkce vůči parametrům

sítě $\overrightarrow{W}$. Tento algoritmus mění váhy neuronové sítě v jednotlivých iteracích následovně:

$$\overrightarrow{W}_{n+1} = \overrightarrow{W}_n - \alpha \nabla_{\overrightarrow{W}} L(\overrightarrow{W}_n) \tag{1.8}$$

Lze vidět, že sestup probíhá *proti* směru gradientu. Rychlost sestupu reguluje $\alpha \in \mathbb{R}_+$. $\overrightarrow{W}_0$ je nejčastěji inicializováno malými náhodnými hodnotami. [9] Gradientní sestup na jednoduché funkci je znázorněn na obrázku 1.3.



Obrázek 1.3: Gradientní sestup funkce $f(x)$ začínající v bodě x_0.

Ztrátová funkce 1.4 počítá chybu pomocí celého datasetu $\mathcal{D}$. Z praktického hlediska je toto počítání nevhodné, protože značně zpomaluje fázi učení. Proto se používá tzv. *stochastický gradientní sestup*, který postupně dosazuje dvojice $(\overrightarrow{X}, y) \in \mathcal{D}$ v náhodném pořadí a počítá predikci $\hat{y}$ v tzv. *dopředném chodu*. Argumentem ztrátové funkce je v případě stochastického gradientního sestupu pouze jediná dvojice $(\hat{y}, y)$. Váhy jsou následně změněny pomocí rekurentní rovnice 1.8 v tzv. *zpětném chodu*, tím je jedna iterace učícího algoritmu uzavřena. V případě, že každý datový bod z $\mathcal{D}$ projde neuronovou sítí v dopředném i zpětném chodu, mluvíme o jedné *epoše* učícího algoritmu. [2]

Další formou gradientního sestupu je *dávkový stochastický gradientní sestup* (z anglického *mini-batch stochastic gradient descent*). Jedná se o kompromis mezi gradientním sestupem a stochastickým gradientním sestupem. V takovém případě je pro dopředný i zpětný chod použita větší množina S datových bodů. Díky tomuto přístupu je často konvergence ztrátové funkce hladší, protože vypočítaný gradient v každé iteraci je průměrem gradientů jednotlivých členů množiny S, a tím pádem se jedná o přesnější odhad skutečného gradientu ztrátové funkce. [2] Z implementačního pohledu je možné

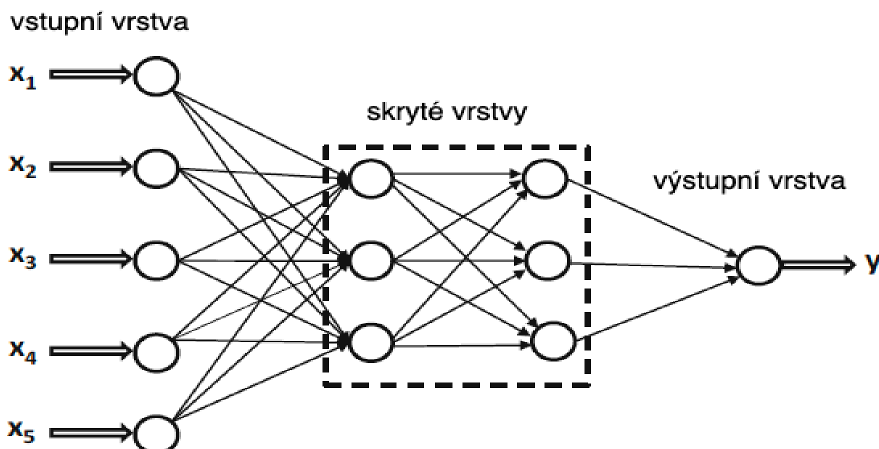
použít vektorové instrukce zpracovávající více dat najednou, což dále zrychluje dobu učení a predikce.

1.3 Vícevrstvé neuronové sítě

Vícevrstvým neuronovým sítím se také říká *vícevrstvé perceptrony*. Vícevrstvé sítě skutečně z perceptronu vychází. Zachovávají myšlenku vstupní a výstupní vrstvy a přidávají *skryté vrstvy*. [10]

Skryté vrstvy jsou vloženy mezi vrstvy vstupní a výstupní. Trénovací data jasně uvádí, jak se musí chovat výstupní vrstva. Výstupní vrstva musí produkovat predikce $\hat{y}$, které jsou blízko skutečné hodnotě vysvětlované proměnné y. Naopak trénovací dataset neuvádí žádaný výstup pro vrstvy skryté. Výstup skrytých vrstev tedy musí být řízen učícím algoritmem. [10]

Typická architektura vícevrstvých sítí je znázorněna na obrázku 1.4.



Obrázek 1.4: Vícevrstvá neuronová síť. Převzato z [2]. Přeloženo autorem.

1.3.1 Vhodné aktivační funkce

Správný výběr aktivačních funkcí je kritickou částí návrhu neuronové sítě. Největší síla neuronových sítí vychází z faktu, že opakovaná skladba určitých funkcí navyšuje reprezentační schopnost neuronové sítě, a tudíž zmenšuje prostor parametrů potřebných k úspěšnému učení. [2]

Ne všechny funkce jsou vhodné jako aktivační. Již bylo vysvětleno, proč výběr znaménkové aktivační funkce není příliš praktický. Naopak často používanými funkcemi jsou *sigmoida, ReLU (usměrněná lineární funkce)* nebo *hyberbolický tangens (zkráceně tanh)*.

Sigmoida jež byla definována v 1.5. Obor hodnot sigmoidy je otevřený interval $(0, 1)$ a je diferencovatelná na celém definičním oboru. Sigmoida může být použita jako aktivační funkce, pokud je například potřeba, aby výstup

neuronové sítě při binární klasifikaci určoval pravděpodobnost, že $y = 1$. Naopak hyperbolický tangens může být použit, pokud je nutné, aby výstup byl kladný i záporný, jelikož obor hodnot této funkce je interval $(-1, 1)$.

Sigmoida a hyperbolický tangens byly v minulosti preferovanou volbou pro aktivační funkce. V současné době jsou tyto funkce z velké části nahrazovány ReLU funkcí. Hlavními důvody je fakt, že ReLU se lépe trénuje, a také, že není shora omezená, což přináší větší flexibilitu pro výstup neuronové sítě. [2]

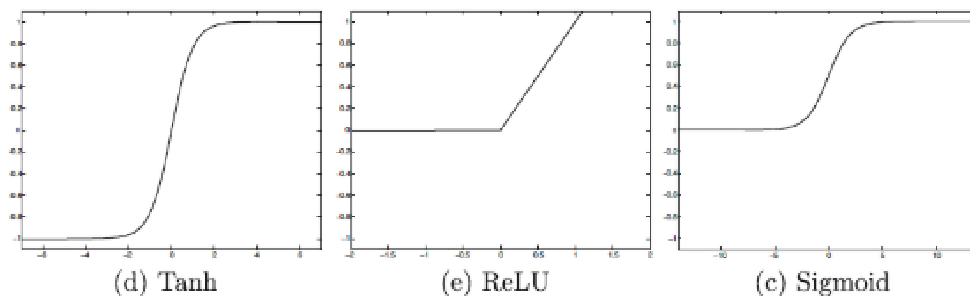
Předpisy těchto funkcí jsou následující:

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{1.9}$$

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{1.10}$$

$$relu(x) = \max\{x, 0\} \tag{1.11}$$

Sigmoidě a tanh se často přezdívá „*omezující*“ (z anglického *squashing*), jelikož výstupem těchto funkcí je vstup omezený určitým intervalem. [2] To je patrné i z grafů těchto funkcí na obrázku 1.5.



Obrázek 1.5: Grafy pro různé aktivační funkce. Převzato z [2].

Speciálním případem aktivační funkce je *softmax*:

$$softmax(\overrightarrow{X})_i = \frac{\exp x_i}{\sum_{j=0}^{k} \exp x_j} \qquad \forall i \in \{1 \dots k\} \tag{1.12}$$

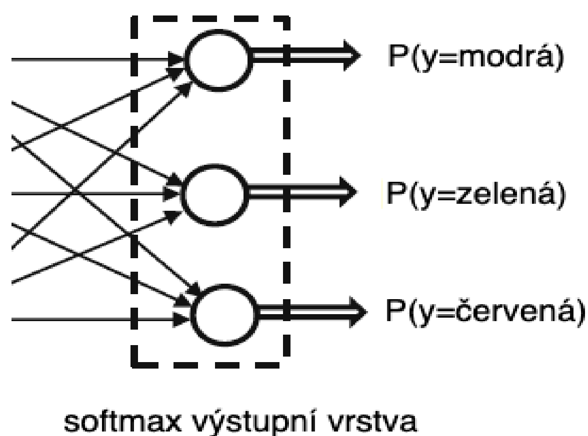
Softmax je možné použít, pokud je potřeba, aby výstup sítě reprezentoval rozložení pravděpodobnosti přes k diskrétních hodnot. Vstupem je vektor $\overrightarrow{X} = [x_1 \dots x_k]$ reálných čísel a výstupem je také vektor $\overrightarrow{Y} = [y_1 \dots y_k]$, kde pro každé y_i platí, že spadá do intervalu $(0, 1]$ a zároveň $\sum_{i=0}^{k} y_i = 1$ [10]. Díky těmto vlastnostem lze výstup $\overrightarrow{Y}$ interpretovat jako validní rozložení pravděpodobností.

Softmax se nejčastěji používá jako aktivační funkce pro výstupní vrstvu. V takovém případě neuronová síť minimalizuje *logaritmickou ztrátovou funkci* (z anglického *log loss*):

$$logloss = -\sum_{i=1}^{M} y_i \log(\hat{y}_i) \tag{1.13}$$

kde M je počet diskrétních hodnot, kterých může vysvětlovaná proměnná nabývat, y_1 vyjadřuje skutečnou pravděpodobnost pro danou hodnotu vysvětlované proměnné a $\hat{y}$ vyjadřuje predikci.

Pokud například neuronová síť rozpoznává barvu auta na obrázku, lze si výstupní vrstvu představit následovně:



Obrázek 1.6: Ukázka použití softmax funkce. Převzato z [2]. Přeloženo autorem.

1.4 Proces učení

Při učení vícevrstvých neuronových sítí je hlavním problémem fakt, že ztrátová funkce je *funkcí složenou, parametrizovanou váhami napříč všemi vrstvami uvnitř sítě.* [2] Výpočet gradientu ztrátové funkce vůči všem vahám tedy není tak přímočarý jako u jednovrstvé neuronové sítě.

Pro usnadnění procesu učení se využívá algoritmus *backpropagation*, který při zpětném chodu sítě využívá informaci o lokálním gradientu ztrátové funkce, kterou šíří zpět do dřívějších vrstev. Činí tak efektivně pomocí *řetězového pravidla* pro výpočet derivace složené funkce a pomocí techniky *dynamického programování.* [10]

Je důležité zdůraznit, že backpropagation řeší pouze problém výpočtu gradientu. Samotný „*update*“ vah řeší metoda gradientního sestupu, případně její varianta.

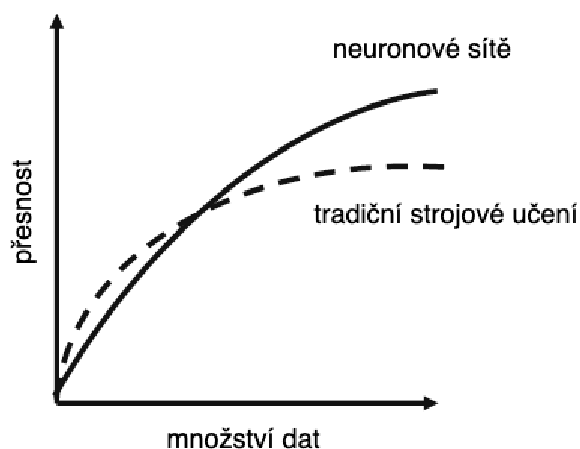
Backpropagation není komplikovaný algoritmus. Skrývá ale mnoho drobností, které nemusí být na první pohled zřejmé. Algoritmu backpropagation je v práci věnována celá kapitola, ve které bude algoritmus rozebrán podrobně. Dále bude uvedeno, jak lze při jeho běhu efektivně využít stovek vláken na grafickém čipu, a několikanásobně tak zrychlit proces učení umělé neuronové sítě.

Kapitola 2

Úvod do strojového učení

Hluboké neuronové sítě jsou často příliš složitými řešeními pro případy s omezeným množstvím dat. V takovém případě je výhodnější zvolit modely tradičního strojového učení, u kterých je optimalizace výrazně jednodušší, neboť tento proces je transparentnější.

Na druhou stranu, při stoupajícím množství dat, stoupá i potenciál neuronových sítí, které dosahují svého pomyslného maxima později než jiné modely strojového učení. [2] Nejlépe tuto skutečnost znázorňuje jednoduchý graf 2.1.



Obrázek 2.1: Závislost neuronových sítí a tradičního strojového učení na množství dostupných dat. Převzato z [2]. Přeloženo autorem.

V této kapitole bude představeno několik modelů strojového učení v kontextu neuronových sítí, neboť většinu modelů strojového učení lze vyjádřit pomocí jednoduché neuronové sítě o jedné nebo dvou výpočetních vrstvách. [2]

Modely strojového učení lze rozdělit do třech kategorií podle problému,

které řeší:

- **Učení s učitelem** – model se při učení s učitelem snaží aproximovat funkci, která mapuje vstup na výstup na základě dvojic, které se skládají ze vstupních dat a očekávaného výstupu. Neuronové sítě jsou příkladem modelu učeného s učitelem;
- **Učení bez učitele** – model se v datech snaží rozpoznat doposud nerozpoznané vzory. V trénovacích datech tedy neexistuje žádná vysvětlovaná proměnná;
- **Učení posilováním** (z anglického *reinforcement learning*) – proces poháněný odměnou za učiněná rozhodnutí. Model se v průběhu učení snaží maximalizovat získanou odměnu, ať už je definovaná jakkoliv. Používá se v případech, kdy je problém snadné vyhodnotit, ale těžké specifikovat. Například na konci šachové partie je snadné vyhodnotit, jestli hráč vyhrál, ale je těžké specifikovat ideální pohyb hráče při každém tahu. [11, 12, 2]

Učení s učitelem lze dále dělit následovně:

- **Klasifikace** – model rozděluje vstupní data do dvou nebo více tříd;
- **Regrese** – model predikuje reálné číslo na základě vstupních dat. [10]

Toto dělení není u neuronových sítí tak důležité, jelikož stačí vyměnit výstupní vrstvu a ztrátovou funkci. Zbytek sítě může ve většině případů zůstat stejný.

2.1 Lineární regrese

Lineární regrese je příkladem učení s učitelem. Trénovací datový bod má formu dvojice $(\overrightarrow{X}, y)$, kde $\overrightarrow{X} = [x_1 \dots x_d]$ je vektor vstupních hodnot a $y \in \mathbb{R}$ je hodnota vysvětlované proměnné. Predikci získáme následovně:

$$\hat{y} = \overrightarrow{W} \cdot \overrightarrow{X} \tag{2.1}$$

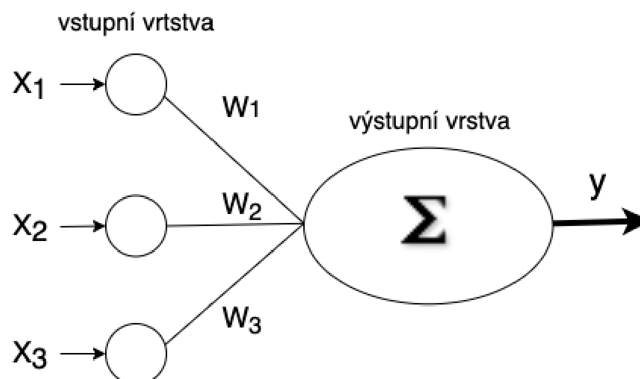
kde $\overrightarrow{W}$ je vektor vah, které jsou během během učení nastavovány tak, aby se minimalizovala chyba predikce. Chyba predikce modelu je vyjádřena pomocí *kvadratické chyby* napříč celým trénovacím datasetem $\mathcal{D}$. Tedy:

$$L = \sum_{(\overrightarrow{X}_i, y_i) \in \mathcal{D}} (y_i - \hat{y}_i)^2 \tag{2.2}$$

Stejně jako u neuronových sítí je vhodné použít v rovnici 2.1 *práh*, ale i v této kapitole bude použit trik, kdy každý vstup $\overrightarrow{X} = [x_1, \dots, x_d]$ převedeme

na $\overrightarrow{X} = [1, x_1, \ldots, x_d]$ a do $\overrightarrow{W}$ přidáme váhu w_0, která bude vyjadřovat práh. Tím pádem se práh nemusí v rovnicích explicitně uvádět.

Model logistické regrese lze jednoduše vyjádřit jako neuronovou síť. Taková síť je znázorněná na obrázku 2.2.



Obrázek 2.2: Neuronová síť reprezentující model lineární regrese.

Jedná se o síť minimalizující ztrátovou funkci $L(y, \hat{y}) = (y - \hat{y})^2$ s jediným výpočetním neuronem ve výstupní vrstvě, který používá identitu jako aktivační funkci.

Stejně jako kteroukoliv jinou umělou neuronovou síť, i tento model lze učit pomocí metody gradientního sestupu. Avšak pro tento speciální případ existuje uzavřená forma řešení, a není tudíž potřeba používat žádnou iterativní metodu.

Uzavřená forma využívá matici $\mathbb{A}$ definovanou pomocí matice trénovacích dat $\mathbb{D}$, jejíž řádky obsahují jednotlivé vektory vstupních trénovacích dat $\overrightarrow{X} = [1, x_1 \ldots x_d]$:

$$\mathbb{A} = (\mathbb{D}^T \mathbb{D})^{-1} \mathbb{D}^T \tag{2.3}$$

Vektor vah je poté definován následovně:

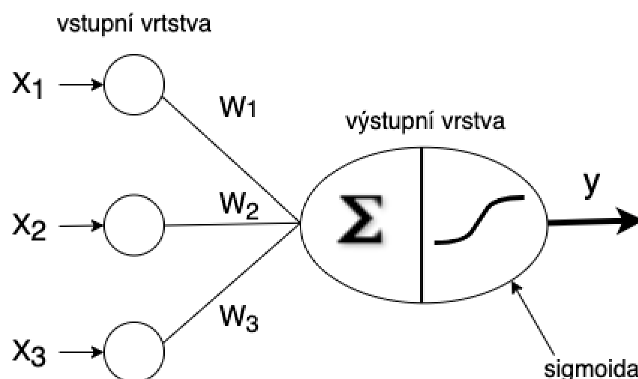
$$\overrightarrow{W} = \mathbb{A} \overrightarrow{Y} \tag{2.4}$$

kde $\overrightarrow{Y} = [y_1 \ldots y_n]$ je vektor hodnot vysvětlované proměnné. [2]

2.2 Logistická regrese

Logistickou regresi je možné považovat za formu lineární regrese pro binární klasifikaci.

Jedná se o pravděpodobnostní model, který klasifikuje na základě pravděpodobnosti. Výstup $\hat{y}$ je interpretován jako pravděpodobnost příslušnosti do třídy 1. Model klasifikuje datový bod do třídy 1, pokud $\hat{y} > 0.5$. Cílem učení



Obrázek 2.3: Neuronová síť reprezentující model logistické regrese.

modelu je maximalizovat $\hat{y}$ pokud $y \in \{-1, +1\} = +1$ a minimalizovat $\hat{y}$ pokud $y = -1$. [10] Toho lze dosáhnout minimalizací vhodné ztrátové funkce. Jednou z takových funkcí je *negativní logaritmus věrohodnostní funkce* pro všech n instancí trénovacích dat:

$$L = \sum_{i=1}^{n} -\log\left(|y_i/2 - 0.5 + \hat{y}_i|\right) \tag{2.5}$$

Predikce $\hat{y}$ pro vstupní data $\overrightarrow{X} = [1, x_1, \ldots, x_d]$ a parametry modelu $\overrightarrow{W} = [w_0, w_1, \ldots, w_d]$ je definována v 2.6 a její vyjádření pomocí sigmoidy v 2.7.

$$\hat{y} = \frac{1}{1 + \exp\{-\overrightarrow{W} \cdot \overrightarrow{X}\}} \tag{2.6}$$

$$\hat{y} = sigmoid(\overrightarrow{W} \cdot \overrightarrow{X}) \tag{2.7}$$

Model logistické regrese lze opět vyjádřit pomocí jednoduché umělé neuronové sítě. Jedná se o model perceptronu se sigmoidou jako aktivační funkcí. Jako ztrátová funkce je pochopitelně použita negativní věrohodnostní funkce. Tento model je znázorněn na obrázku 2.3.

Parametry logistické regrese jsou nastavovány pomocí stochastického gradientního sestupu. K tomu je zapotřebí definovat upravení ztrátové funkce 2.5 pro jedinou instanci testovacích dat $(\overrightarrow{X}_i, y_i)$:

$$L_i = -\log\left(|y_i/2 - 0.5 + \hat{y}_i|\right) \tag{2.8}$$

Gradient této ztrátové funkce vůči parametrům modelu lze vyjádřit takto:

$$\nabla_{\overrightarrow{W}} L_i = \frac{y_i \overrightarrow{W}}{1 + \exp\{y_i \overrightarrow{W} \cdot \overrightarrow{X}_i\}} \tag{2.9}$$

Tento gradient následně stačí dosadit do již známé rekurentní rovnice pro stochastický gradientní sestup:

$$\overrightarrow{W}_{i+1} = \overrightarrow{W}_i - \alpha \nabla_{\overrightarrow{W}} L_i \tag{2.10}$$

Model logistické regrese je určen pro klasifikaci do dvou *kategorických* tříd, tedy do dvou tříd, které mezi sebou nemají žádný vztah pořadí. Avšak existuje i varianta pro *ordinální* třídy, tedy třídy, které mezi sebou mají vztah pořadí (např: základní škola, střední škola, vysoká škola). [13]

Tato varianta se nazývá *ordinální regrese.* [14] Varianta pro více než dvě třídy se nazývá *multinominální logistická regrese.* [2]

2.3 Metoda podpůrných vektorů

Metoda podpůrných vektorů (z anglického *support vector machines*, zkráceně **SVM**) je dalším příkladem z modelů učených s učitelem.

Stejně jako pro model lineární regrese, vztah pro predikci $\hat{y}$ je definován následovně:

$$\hat{y} = \overrightarrow{W} \cdot \overrightarrow{X} \tag{2.11}$$

I přesto, že SVM predikuje $\hat{y}$ stejným způsobem jako regresní model, SVM je model určený pro binární klasifikaci. To je zásluhou použité ztrátové funkce definované pro jednu dvojici $(\overrightarrow{X}_i, y)$ testovacích dat:

$$L_i = \max\{0, 1 - y_i \hat{y}_i\} \tag{2.12}$$

Myšlenka této ztrátové funkce je, že trénovací instance $(\overrightarrow{X}_i, y)$, kde $y = +1$ bude penalizována pouze pokud predikce $\hat{y}$ je menší než 1, a pro $y = -1$ pouze pokud $\hat{y}$ je větší než -1. Samotná klasifikace probíhá použitím znaménkové funkce $sign(\hat{y})$. [2] Například pro $y = 1$ tedy nezáleží o kolik větší $\hat{y}$ je.

Z pohledu neuronových sítí se tedy dá říci, že neuronová síť vytvořená podle modelu SVM používá jinou aktivační funkci při učení a při samotné predikci bez trénovacích dat.

Gradient ztrátové funkce použitý pro stochastický gradientní sestup je následující:

$$\nabla_{\overrightarrow{W}} L_i = \begin{cases} -y_i \overrightarrow{X}_i & y_i \hat{y}_i < 1, \\ 0 & jinak. \end{cases} \tag{2.13}$$

Kapitola 3

Backpropagation

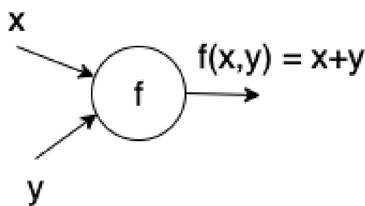
Metoda gradientního sestupu v každé iteraci upraví všechny parametry neuronové sítě. Činí tak pomocí vztahu popsaného v 1.3, tedy pomocí gradientu ztrátové funkce vůči vahám sítě.

V případě perceptronu je výpočet gradientu snadný, jelikož ztrátová funkce je přímou funkcí těchto vah. V případě vícevrstvé architektury je ztrátová funkce funkcí složenou z mnoha menších funkcí a výpočet gradientu není tak přímočarý. [10]

Algoritmus je vhodné vysvětlit z pohledu *výpočetního grafu*. Jedná se o abstrakci, která zjednoduší pohled na složené funkce více proměnných a samotný průchod backpropagation takovou funkcí.

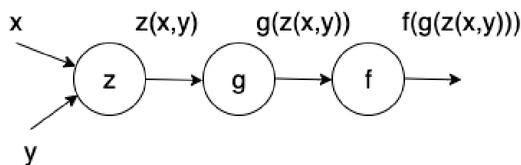
3.1 Výpočetní graf

Vrcholy ve výpočetním grafu reprezentují funkce a hrany „*tok*“ hodnot proměnných. Libovolná funkce $f : \mathbb{R}^n \mapsto \mathbb{R}$ bude v grafu reprezentována jako vrchol f, který má n příchozích hran a jedinou odchozí. Na obrázku 3.1 je znázorněn graf pro funkci $f(x, y) = x + y$.

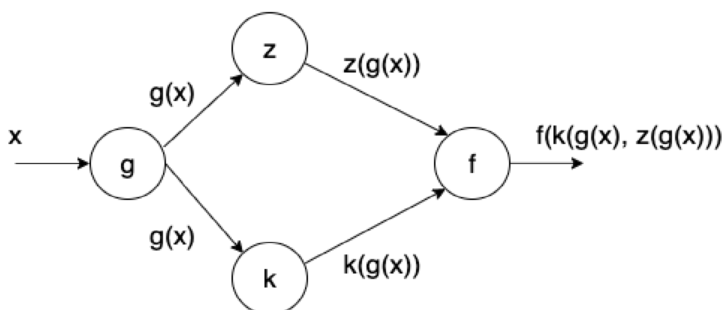


Obrázek 3.1: Výpočetní graf pro funkci $f(x, y)$.

Tímto způsobem je možné znázornit i funkce složené. Příklady takovýchto funkcí jsou na obrázcích 3.2 a 3.3



Obrázek 3.2: Výpočetní graf pro funkci $f(g(z(x,y)))$.



Obrázek 3.3: Výpočetní graf pro funkci $f(z(g(x)), k(g(x)))$.

Je snadné si představit, že takovým grafem je možné reprezentovat celou neuronovou síť.

3.1.1 Řetízkové pravidlo ve výpočetním grafu

Řetízkové pravidlo (z anglického *chain rule*) slouží pro výpočet derivace funkce, která je funkcí složenou z jednodušších funkcí, pro které jsou známy derivace. [10]

Nechť $y = g(x)$ a $z = f(g(x)) = f(y)$ jsou reálné funkce. Řetízkové pravidlo je definováno jako:

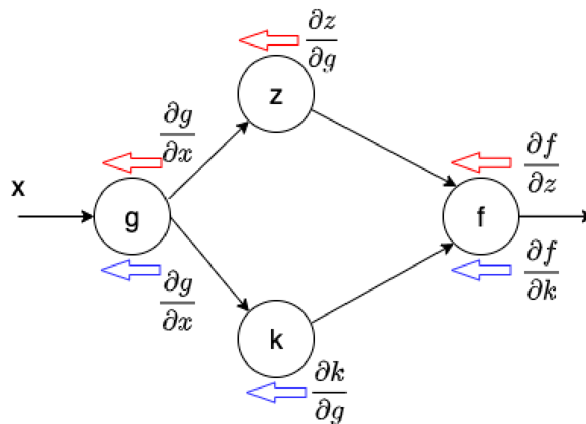
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \quad [10] \tag{3.1}$$

Variantu pro funkci více proměnných lze ukázat na funkci z grafu 3.3:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial k}\frac{\partial k}{\partial g}\frac{\partial g}{\partial x} \quad [2] \tag{3.2}$$

Pomocí výpočetního grafu si lze řetízkové pravidlo pro $\frac{\partial f}{\partial x}$ představit jako cestování po hranách proti jejich směru, po všech možných cestách z vrcholu f ke hraně x:

Výpočetní graf pro vícevrstvé neuronové sítě není ve většině případů tak jednoduchý jako graf 3.4. Propojení vrstev tvoří *úplný bipartitní graf*, což



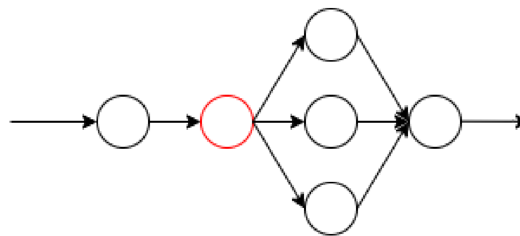
Obrázek 3.4: Znázornění řetízkového pravidla pro funkci $f(z(g(x)), k(g(x)))$ pomocí výpočetního grafu.

znamená, že mezi vrstvami o velikostech m a n existuje mn cest, a to je hlavní důvod proč je backpropagation potřeba. [15]

Počet cest ve výpočetním grafu reprezentujících neuronovou síť bude stoupat exponenciálně vůči hloubce sítě. Například pro síť se 100 vrcholy a třemi vrstvami bude existovat milion cest ze vstupní hrany po výstupní. Výpočet gradientu pomocí naivního použití řetízkového pravidla se stává výpočetně nesnesitelným i pro takto malé sítě. [2]

3.1.2 Řetízkové pravidlo a dynamické programování

Neuron předává svůj výstup všem neuronům ve vrstvě následující. Důležité je pozorování, že lokální gradient jednotlivých neuronů se počítá opakovaně. Situace je znázorněna na obrázku 3.5, kde se lokální gradient červeného neuronu bude počítat pro každou cestu, která skrz něj prochází z výstupní hrany do vstupní.



Obrázek 3.5: Znázornění výpočetní redundance, při naivním algoritmu.

Backpropagation tuto redundanci odstraňuje pomocí techniky dynamického programování, kdy si pamatuje již jednou vypočítaný lokální gradient, což je spolu s řetízkovým pravidlem hlavní myšlenkou backpropagation.

3.2 Průběh algoritmu

Backpropagation ovlivňuje hlavně zpětný chod neuronové sítě, při kterém dochází k výpočtu gradientů, ale drobné úpravy jsou třeba i při dopředném chodu. Backpropagation počítá s tím, že jednotlivé výstupy pro všechny neurony jsou již spočítány. To znamená, že tyto hodnoty je potřeba uchovat v paměti.

Zpětný chod začíná inicializací všech lokálních gradientů pro ztrátovou funkci L, tedy $\frac{\partial L}{\partial o}$, kde o je výstup výstupní vrstvy. Pokud je počet neuronů ve výstupní vrstvě větší než 1, tak je tento gradient spočítán pro každý výstup o_i.

Nechť existuje cesta z libovolného neuronu h_1 k výstupnímu neuronu o. Množina těchto cest je značená $\mathcal{P}$. Každá taková cesta je značena posloupností neuronů, skrz které prochází, tedy posloupností $h_1, h_2, h_3, \ldots, h_k, o$. Váha, která násobí výstup z neuronu h_r do neuronu h_{r+1} je značena $w_{(h_r, h_{r+1})}$. Parciální derivaci ztrátové funkce vůči váze pro výstup neuronu h_{r-1} lze spočítat následovně:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \underbrace{\frac{\partial L}{\partial o} \cdot \left[\sum_{\mathcal{P}} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]}_{\frac{\partial L}{\partial h_r} \text{Spočítáno pomocí backpropagation}} \cdot \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \quad [2] \tag{3.3}$$

Rekurze pro výpočet parciální derivace $\frac{\partial L}{\partial h_r}$ vůči libovolnému neuronu h_r je definovaná následovně:

$$\frac{\partial L}{\partial h_r} = \sum_{h: h_r \Rightarrow h} \frac{\partial h}{\partial h_r} \frac{\partial L}{\partial h} \quad [2] \tag{3.4}$$

kde značení $\sum_{h: h_r \Rightarrow h}$ značí sumu přes všechny neurony h, které jsou v o jednu dřívější vrstvě než neuron h_r. Jelikož tato rekurze začíná neurony ve výstupní vrstvě a lokální gradient pro ztrátovou funkci L je již spočítán během inicializace, tak je garantováno, že člen $\frac{\partial L}{\partial h}$ je již uložen v paměti.

Složitost backpropagation pro jeden průchod zpětným chodem sítě je lineární vůči počtu hran v sítí. [2]

Kapitola 4

Implementace knihovny

Tato kapitola popisuje realizaci a strukturu knihovny, umožňující snadnou konstrukci umělé neuronové sítě a její následné učení na GPU nebo CPU.

V úvodu jsou popsány použité technologie. Dále bude popsána zvolená implementace backpropagation a návrhová rozhodnutí, která umožňují paralelizaci některých částí algoritmu. Kapitolu uzavře stručný popis použitých nástrojů při samotném vývoji, a také podporované platformy.

4.1 Použité technologie

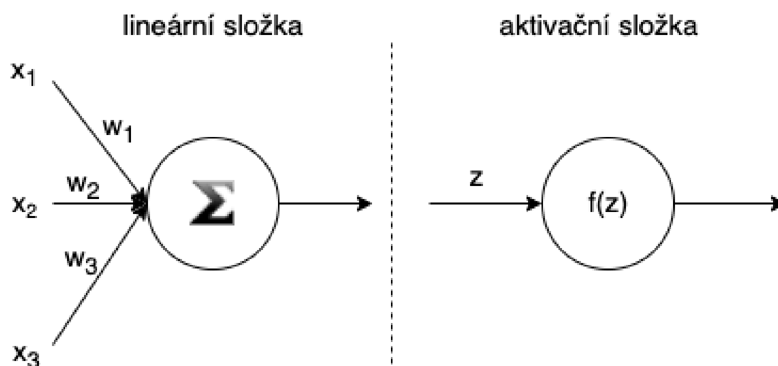
Implementovaná knihovna je založená na numerické knihovně *Template Numerical Library* (zkráceně **TNL**) [16]. TNL je vyvíjena na Katedře matematiky na Fakultě jaderné a fyzikálně inženýrské ČVUT v Praze. TNL je hlavičková knihovna poskytující unifikované rozhraní a správu paměti včetně rozhraní pro implementaci numerických metod na GPU i CPU. TNL poskytuje struktury lineární algebry jako jsou řídké matice, plné matice nebo vektory. Je založena na hardwarové a softwarové platformě společnosti *NVIDIA* [17] nazvané **CUDA**. CUDA umožňuje kompilovat kód napsaný v jazyce **C++**, který bude proveden na GPU. TNL je vyvíjena pod licencí *MIT* [18].

Knihovna bude implementována v jazyce C++. C++ je rychlý kompilovaný jazyk, který si i přesto ponechává dobrou čitelnost a podporu pro objektový návrh. Pro výpočetně náročné algoritmy, kterým je rozhodně i backpropagation, je ideální.

4.2 Implementace backpropagation

Algoritmus backpropagation je kritickou částí trénování neuronové sítě. Implementovaný algoritmus bude ve svém jádru stejný, jako ten který je vysvětlen v předchozí kapitole. Za účelem paralelizace je ale nutné mírně pozměnit strukturu algoritmu i reprezentaci samotné sítě.

Neuronová síť je rozdělena do nezávislých vrstev. A to do takové míry, kdy je i samotný neuron rozdělen na lineární složku a aktivační složku. Aktivační složkou může být například sigmoida. Neuron si lze tedy představit následovně:



Obrázek 4.1: Znázornění rozdělení neuronu na složku lineární a aktivační.

Jednotlivé výpočetní uzly v jedné vrstvě jsou sjednoceny do jedné velké struktury. Vrstvy se tak stávají nezávislými členy sítě, což značně zjednodušuje jejich implementaci, neboť nemají ponětí o přilehlých vrstvách, ani o struktuře neuronové sítě, ve které se nachází.

Tato myšlenka značně usnadňuje implementaci vlastních vrstev a samotnou konstrukci neuronové sítě. Konstrukce sítě tak spočívá pouze ve vkládání vrstev ve vhodném pořadí, což spolu se systémem šablon dovoluje maximální flexibilitu. Uživatel se tak může plně soustředit na maximalizaci přesnosti na daném datasetu, nikoliv na implementační detaily.

Sjednocení neuronů do vrstev navíc dovoluje uchovávání hodnot uvnitř matic. Je tedy možné dopředný i zpětný chod vyjádřit v kontextu násobení matic a aplikací některé z aktivačních funkcí po složkách. Obě tyto operace lze provést velice efektivně a paralelně, zejména na specializovaných zařízeních, kterými jsou i grafické karty. [17]

Vrstva si uchovává matici se svými vstupy a vypočítanými gradienty ztrátové funkce vůči vrstvě samotné. Pro případ lineární vrstvy je uchovávána i matice s nastavitelnými váhami.

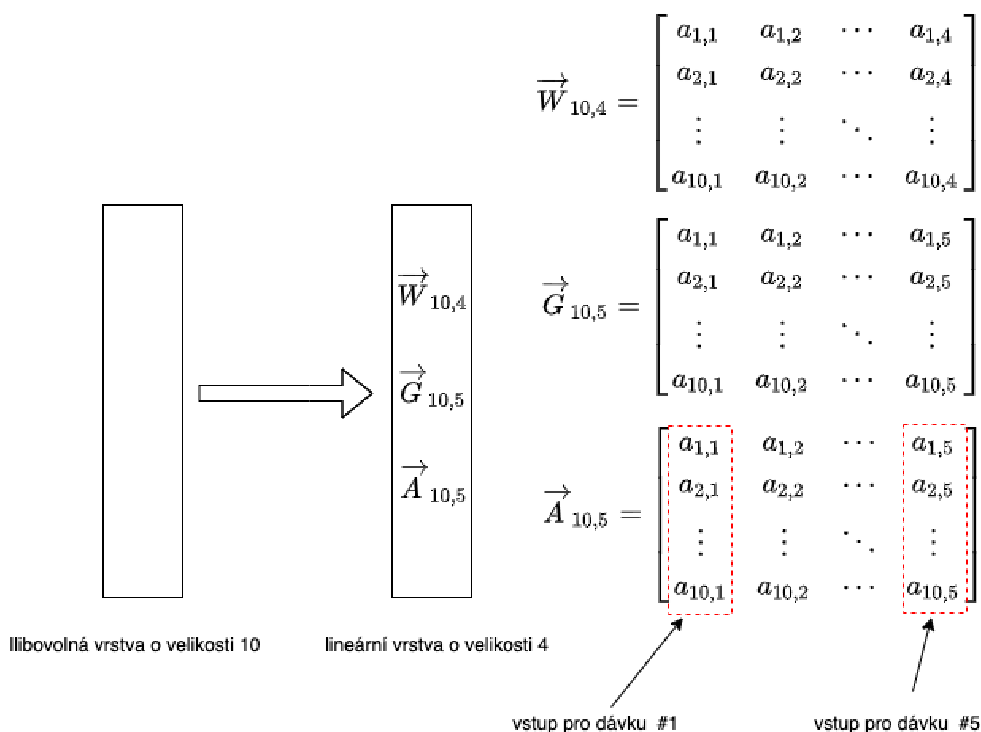
Knihovna implementuje *dávkový* stochastický gradientní sestup. Pro uložení vstupních hodnot a gradientů tedy nestačí pouze vektor, ale musí být použita matice. Příklad uložení hodnot uvnitř lineární vrstvy zasazené do sítě s dávkou o velikosti 5 je znázorněn na obrázku 4.2:

Pro implementaci vrstvy poté stačí jen definovat chování při dopředném chodu a chování při zpětném chodu. Tyto operace pro vrstvy zmíněné v této práci jsou následující:

Tabulka 4.1: Definice chování pro jednotlivé vrstvy v dopředném chodu a zpětném chodu.

Vrstva	Vpřed	Vzad
lineární	$\overrightarrow{A}_{i+1} = \overrightarrow{W}^T \overrightarrow{A}_i$	$\overrightarrow{G}_i = \overrightarrow{W} \overrightarrow{G}_{i+1}$
sigmoida	$\overrightarrow{A}_{i+1} = sigmoid(\overrightarrow{A}_i)$	$\overrightarrow{G}_i = \overrightarrow{G}_{i+1} \odot \overrightarrow{A}_{i+1} \odot (1 - \overrightarrow{A}_{i+1})$
relu	$\overrightarrow{A}_{i+1} = \overrightarrow{A}_i \odot I(\overrightarrow{A}_i > 0)$	$\overrightarrow{G}_i = \overrightarrow{G}_{i+1} \odot I(\overrightarrow{A}_i > 0)$
tanh	$\overrightarrow{A}_{i+1} = tanh(\overrightarrow{A}_i)$	$\overrightarrow{G}_i = \overrightarrow{G}_{i+1} \odot (1 - \overrightarrow{A}_{i+1} \odot \overrightarrow{A}_{i+1})$

kde $\odot$ značí násobení matic po složkách a indexy u matic $\overrightarrow{A}$ a $\overrightarrow{G}$ značí pozici vrstvy v dopředném chodu. Například tedy matice $\overrightarrow{A}_3$ značí vstup třetí vrstvy. Definice *relu* vrstvy používá funkci značenou $I(\cdot)$. Tato funkce je vyhodnocena jako 1 pokud podmínka uvnitř závorek je splněna, jinak 0.



Obrázek 4.2: Způsob uložení vstupů, gradientů a parametrů uvnitř vrstvy.

Pokud je možné brát ztrátovou funkci jako další vrstvu, nabízí se otázka, zda je možné vložit tuto vrstvu i doprostřed sítě a ne pouze na její konec.

Teoreticky to možné je a existují architektury, které toho využívají [2], ale v současné verzi této knihovny tato možnost zatím implementována není.

Celý průběh učení je nyní možné popsat následujícím kódem, který zachycuje hlavní myšlenku bez implementačních detailů:

```
void train(Matrix X, Matrix Y)
{
    Network.firstLayer.input = X;
    Network.lossLayer.targets = Y;

    for (int i = 0; i < network.layersSize; i++) {
        Network.layers[i].forward();
    }

    for (int i = network.layersSize - 1; i >= 0; i--) {
        Network.layers[i].backward();
    }

    Network.Optimizer.optimize(network.weights);
}
```

Je důležité povšimnout si volání třídy `optimizer`. Jak již bylo zmíněno, backpropagation pouze spočítá gradienty ztrátové funkce vůči parametrům sítě. O samotný „*update*“ se stará optimalizační algoritmus, kterým je například *stochastický gradientní sestup*.

V průběhu predikce již není potřeba optimalizovat parametry sítě. Implementace predikce tedy vypadá následovně:

```
void predict(Matrix X)
{
    Network.firstLayer.input = X;

    for (int i = 0; i < network.layersSize; i++) {
        Network.layers[i].forward();
    }
}
```

4.2.1 Další možnosti paralelizace

V práci je popsána a implementována paralelní forma backpropagation, při které dochází k paralelizaci uvnitř výpočetních vrstev, tedy při provádění maticových operací. Algoritmus pro paralelizaci napříč vrstvami není známý a v popsané formě zřejmě i nemožný, neboť výpočet gradientu uvnitř jedné z vrstev je závislý na výsledcích výpočtu ve všech vrstvách dřívějších. [19, 20]

Současné publikované paralelní verze backpropagation se zabývají otázkou distribuce pro více výpočetních uzlů. Tyto algoritmy pracují s myšlenkou rozdělení trénovacího datasetu na menší části a následné spuštení backpropagation na více menších neuronových sítí. Tyto neuronové sítě jsou spolu s částí datasetu trénovány na rozdílných výpočetních uzlech. [19, 20]

Následná predikce probíhá některou z forem většinového hlasování. Predikce dílčích neuronových sítí se spojí v jedinou predikci pomocí určitého algoritmu. V případě klasifikace může být tímto algoritmem predikce pomocí nejčastěji zastoupené třídy. V případě regrese se může jednat o průměr ze všech predikcí. Tento druh predikce pomocí více modelů se obecně nazývá *Ensemble metoda*. [21]

Konkrétní implementace tohoto druhu paralelizace jsou popsány například v [19] a [20]

4.3 Struktura knihovny

TNL neposkytuje konkrétní implementace tříd, nýbrž pouze C++ šablony, do kterých uživatel dosadí požadované datové typy. Knihovna tak poskytuje maximální flexibilitu pro programátora. Hlavička typické datové struktury poté vypadá následovně:

```
TNL::Matrices::DenseMatrix<Real, Device, Index>
```

kde `Real` značí datový typ pro uložené hodnoty uvnitř struktury, `Device` značí zařízení, na kterém budou hodnoty alokovány, a také kde budou provedeny metody třídy. `Index` značí datový typ určen pro indexaci uvnitř struktury.

Implementovaná knihovna tento systém použití šablon zachovává, čímž lépe zapadá do celkové architektury TNL, a také umožňuje stejnou flexibilitu rozhraní. Z toho vyplývá, že implementovaná knihovna je knihovnou hlavičkovou kvůli limitacím C++ šablon. Je tedy implementována v sadě hlavičkových souborů. Všechny implementované struktury jsou tedy C++ šablony, ale ve zbývající části práce budou šablony také nazývány třídami.

Hlavní třídou uchovávající vrstvy sítě je

```
Network<Task, Real, Device, Index>
```

Oproti strukturám TNL je přidán parametr `Task`, kterým síť parametrizuje typ problému, který bude řešit. Typicky jsou jimi klasifikace nebo regrese. Pokud je například definována síť pro klasifikaci do dvou tříd, tak očekávaný výstup predikce je $y \in \{0, 1\}$. Pokud je ale klasifikace rozhodována na základě pravděpodobnosti a výstupní vrstva je definována jako sigmoida, tak je výstupem $\hat{y} \in (0, 1)$. Dle parametru `Task` tedy síť pozná, že má reálný výstup převést na celé číslo. Naopak například při dosazení typu `Tasks::Regression`, se ponechá výstup jako reálný.

Metody `Network<>::addLayer` a `Network<>::addTrainableLayer` slouží k přidávání výpočetních vrstev, respektive výpočetních vrstev obsahujících váhy, které jsou nastavovány při optimalizaci. Hlavičky těchto metod jsou definovány následovně:

```
template <typename Task
          typename Real
          typename Device
          typename Index>
template <template<typename, typename, typename> class T
          typename... Args>
void
Network<Task, Real, Device, Index>::
addLayer(Args... args)
```

```
template <typename Task,
          typename Real,
          typename Device,
          typename Index>
template <template<typename, typename,typename> class T,
          typename... Args>
void
Network<Task, Real, Device, Index>::
addTrainableLayer(Args... args)
```

Tyto deklarace mohou na uživatele působit odpudivě, ale náročnosti jejich implementace jsou vyváženy jednoduchostí při samotném použití. Tyto metody, kromě přidání vrstvy, řeší několik dalších problémů. To je ukázáno na příkladu konstrukce sítě o dvou vrstvách:

```
int main()
{
    Network<Tasks::Classification,
            double,
            TNL::Devices::Cuda
            int> network;

    // velikost vrstvy = 10
    network.addTrainableLayer<LinearLayer>(10);
    network.addLayer<ReLuLayer>();
}
```

Důležité pozorování je, že jednotlivé vrstvy nemusí být před přidáním konstruovány. Neexistuje tedy problém *vlastnictví*, tedy problém, kdy uživateli není zřejmé, zda je jeho úkolem dealokovat alokovanou paměť, nebo to za něj provede vnitřní implementace knihovny. Tento problém lze řešit pomocí tzv. „smart pointerů“, které implicitně definují, komu alokovaná paměť patří, a kdo jí musí dealokovat. Ovšem to do rozhraní vnáší další datové typy, se kterými musí být uživatel seznámen, a to může působit komplikovaně.

Další problém, který uvedené rozhraní řeší, je explicitní uvádění šablonových parametrů. Z důvodů uvedených dříve v této sekci má každá implementovaná vrstva tři šablonové parametry. Tedy `Real`, `Device` a `Index`. Ovšem při použití metod pro přidávání vrstev není potřeba tyto parametry specifikovat. Parametry uvedené při konstrukci třídy `Network<>` se dosadí automaticky. V uvedeném příkladu to budou `double`, `TNL::Devices::Cuda` a `int`.

Veškeré rozhraní implementované knihovny je navrženo obdobným způsobem, jakým jsou navrženy metody `addLayer` a `addTrainableLayer`.

Jakožto struktury pro ukládání dat v souvislé paměti jsou použity implementace plné matice a vektoru od TNL:

```
TNL::Matrices::DenseMatrix<Real, Device, Index>
```

```
TNL::Containers::Vector<Real, Device, Index>
```

Tyto třídy poskytují unifikované rozhraní pro alokovanou paměť na CPU nebo GPU, v závislosti na parametru `Device`. Dále poskytují rozhraní pro základní operace lineární algebry.

Základní použití knihovny pro konstrukci, trénování a predikci vypadá následovně:

```
using DEVICE = TNL::Device::Cuda;
using MatrixType =
TNL::Matrices::DenseMatrix<float, DEVICE, int>;

float execute(MatrixType trainX, MatrixType trainY,
              MatrixType testX, MatrixType testY)
{
    const int numberOfEpochs = 3;
    const int batchSize = 4;

    Network<Tasks::Classification, float, DEVICE, int>
    network(numberOfEpochs, batchSize);

    network.setOptimizer<SGDOptimizer>(0.001);
    network.setInitializer<RandomInitializer>();

    network.addTrainableLayer<LinearLayer>(100);
    network.addLayer<ReLuLayer>();

    network.addTrainableLayer<LinearLayer>(300);
    network.addLayer<SigmoidLayer>();

    network.addTrainableLayer<LinearLayer>(10);
    network.addLayer<SoftMaxLayer>();

    network.setLossLayer<CrossEntropyLoss>();
    network.fit(trainY, trainX);
    auto predicted = network.predict(testX);
    float acc = Metrics::Accuracy::exec(predicted, testY);
    return acc;
}
```

V tomto příkladu jsou do funkce již dodána připravená trénovací i testovací data v podporovaných strukturách. Do konstruktoru třídy `Network` lze dosadit dva argumenty. Argumenty `batchSize` a `numberOfEpochs` určující velikost dávky, respektive počet epoch použitých při trénování. Poslední přidaná vrstva pomocí `addLayer` nebo `addTrainableLayer` je implicitně považována za výstupní. Neuronová síť z příkladu tedy bude mít 10 výstupních neuronů, jakožto výstup vrstvy implementující *softmax*. Vrstva ztrátové funkce je dodána pomocí metody `setLossLayer`. Metody `fit` a `predict` slouží pro spuštění trénování a predikce. Na závěr je spočítána přesnost klasifikace na testovacích datech.

Dosazený parametr `TNL::Devices::Cuda` určuje zařízení, na kterém je síť alokována a spuštěna. V tomto případě je to tedy GPU využívající platformu Cuda. Pokud bychom chtěli zařízení přepnout na CPU, stačí pouze vyměnit `TNL::Devices::Cuda` za `TNL::Devices::Host`. O zbytek se postará knihovna a TNL.

Dále jsou v příkladu použity metody `setInitializer` a `setOptimizer`. Tyto metody slouží pro dosazení předpřipravených nebo uživatelem implementovaných algoritmů pro počáteční inicializaci vah a samotnou optimalizaci ztrátové funkce pomocí gradientů. V příkladu je dosazen pro optimalizaci algoritmus pro *stochastický gradientní sestup* s parametrem $\alpha = 0.001$ a jako inicializátor je dosazen algoritmus, který inicializuje váhy na náhodné hodnoty v intervalu $(-1, 1)$;

Snadná rozšiřitelnost knihovny je zajištěna čistým rozhraním, které poskytuje sada abstraktních tříd. Tyto třídy mají v názvu předponu `Abstract` a jejich použití je popsáno v přiložené dokumentaci. Například pro algoritmus optimalizace je definována abstraktní třída `AbstractOptimizer`, definována následovně:

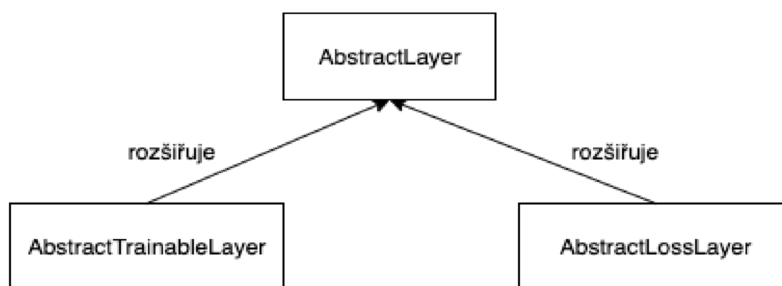
```
template <typename Real, typename Device, typename Index>
class AbstractOptimizer
{
public:
    using VectorType =
    TNL::Containers::Vector<Real, Device, Index>;
    using VectorViewType =
    typename VectorType::ViewType;
    using ConstVectorViewType =
    typename VectorType::ConstViewType;
public:
    virtual ~AbstractOptimizer() = default;

    virtual void exec(VectorViewType parameters,
    const ConstVectorViewType parametersGradients) = 0;
```

```
};
```

Stačí pouze přetížit metodu `exec`, která má za argumenty parametry všech vrstev uložené v souvislé paměti včetně jejich gradientů.

Sada abstraktních tříd je poskytnuta i pro implementaci vrstev:



Obrázek 4.3: Sada abstraktních tříd sloužící pro implementaci vrstev.

Kapitola 5

Testování

V této kapitole je popsáno testování implementace knihovny. Knihovna je otestována na dvou datasetech obrázků a na několika architekturách neuronových sítí. Jednotlivé sítě jsou spouštěny na CPU i GPU. Testování neslouží k dosažení nejlepší přesnosti, nýbrž jako důkaz funkčnosti implementace. Důraz je kladen na ujištění, že přesnost je v „*rozumných*“ mezích, a že testy běžící na GPU jsou výrazně rychlejší než testy běžící na CPU.

5.1 Prostředí

Implementace je testována na operačním systému **Ubuntu 16.04**. S následujícími komponenty:

- **GPU** – Nvidia Geforce GTX 1080Ti;
- **CPU** – Intel Core i3-8100, 3,6GHz;
- **Hlavní páměť** – 12GiB;
- **Frekvence hlavní paměti** – 3200MHz.

Kompilátor použitý pro testy spouštěné na CPU je GNU GCC 7.4.0 s přepínači -O3 -DNDEBUG.

Kompilátor použitý pro testy spouštěné na GPU je CUDA NVCC 10.2 s přepínači -O3 -DNDEBUG -Xptxas -O3.

5.2 Datasety

Pro testy budou použity dva datasety s obrázky.

5.2.1 MNIST

MNIST [22] je populární dataset, často používaný k otestování modelů strojového učení. Dataset se skládá ze 70 000 černobílých obrázků ručně psaných číslic. Jedná se o celá čísla od 0 do 9. Každá číslice je zastoupena stejným počtem reprezentantů. Obrázky mají rozměr 28×28 pixelů.

5.2.2 CIFAR-10

Cifar-10 [23] se skládá z 60 000 barevných obrázků o rozměrech 32×32 pixelů. Dataset obsahuje reprezentanty z deseti kategorií. Kategoriemi jsou například pes, žába, loď nebo auto. Každá kategorie je zastoupena stejným počtem obrázků.

5.3 Výsledky

Pro všechny uvedené testy byl použit stochastický gradientní sestup s parametrem $\alpha = 0.001$. Inicializace parametrů je náhodná v intervalu $(-1, 1)$. Zbylé parametry jsou uvedeny výčtem, následované tabulkou s výsledky experimentu. Data v tabulce jsou vždy průměrem tří testů se stejnými parametry.

- **Dataset:** MNIST;
- **Velikost dávky:** 16;
- **Počet epoch:** 20;
- **Datový typ Real**: `float`;
- **Architektura:**
    1. Lineární vrstva – velikost = 100;
    2. ReLU;
    3. Lineární vrstva – velikost = 100;
    4. ReLU;
    5. Lineární vrstva – velikost = 100;
    6. Sigmoida;
    7. Lineární vrstva – velikost = 10;
    8. Softmax;
    9. Negativní logaritmus věrohodnostní funkce.

Tabulka 5.1: Výsledky testování MNIST 1.

	CPU	**GPU**
přesnost na testovacích datech	0.959	0.965
přesnost na trénovacích datech	0.971	0.978
doba běhu	562.1s	163.2s

- **Datový typ Real:** `double`;

Ostatní parametry zůstávají stejné jako u předešlého testu.

Tabulka 5.2: Výsledky testování MNIST 2.

	CPU	**GPU**
přesnost na testovacích datech	0.969	0.961
přesnost na trénovacích datech	0.977	0.971
doba běhu	747.2s	325.3s

- **Dataset:** MNIST;
- **Velikost dávky:** 32;
- **Počet epoch:** 30;
- **Datový typ Real**: `float`;
- **Architektura:**
  1. Lineární vrstva – velikost = 200;
  2. ReLU;
  3. Lineární vrstva – velikost = 250;
  4. ReLU;
  5. Lineární vrstva – velikost = 200;
  6. Sigmoida;
  7. Lineární vrstva – velikost = 10;
  8. Softmax;
  9. Negativní logaritmus věrohodnostní funkce.

Tabulka 5.3: Výsledky testování MNIST 3.

	CPU	**GPU**
přesnost na testovacích datech	0.950	0.948
přesnost na trénovacích datech	0.965	0.961
doba běhu	**1741.0s**	204.3s

- **Dataset:** CIFAR-10;
- **Velikost dávky:** 128;
- **Počet epoch:** 20;
- **Datový typ Real**: `float`;
- **Architektura:**
  1. Lineární vrstva – velikost = 1500;
  2. ReLU;
  3. Lineární vrstva – velikost = 1000;
  4. ReLU;
  5. Lineární vrstva – velikost = 10;
  6. Softmax;
  7. Negativní logaritmus věrohodnostní funkce.

Tabulka 5.4: Výsledky testování CIFAR-10 1.

	CPU	**GPU**
přesnost na testovacích datech	0.430	0.402
přesnost na trénovacích datech	0.427	0.403
doba běhu	3201.2s	840.1s

5.4 Závěr testování

Dle dosažených výsledků se dá předpokládat, že algoritmus backpropagation i algoritmus predikce jsou implementovány správně.

Výsledky predikce se od sebe napříč zařízeními neliší a jsou v mezích možností pro základní architekturu umělých neuronových sítí.

Doba běhu se při stejných testovacích podmínkách liší výrazně, v jednom z testovacích scénářů byla na GPU stabilně osmkrát rychlejší, což poukazuje na správnost implementace napříč zařízeními.

Závěr

Hlavním cílem práce byla implementace knihovny pro snadnou práci s umělými neuronovými sítěmi. Výsledná implementace poskytuje jednoduché rozhraní včetně možnosti snadného rozšíření. Zkonstruovanou síť lze jednoduše spustit na CPU i GPU, čehož bylo dosaženo integrací s knihovnou TNL. Provedené testy indikovaly správnost implementace napříč zařízeními.

Dále byla podrobně rozebrána architektura umělých sítí včetně detailního vysvětlení algoritmu backpropagation. V neposlední řadě byly představeny některé modely tradičního strojového učení.

Možnosti budoucí práce

V knihovně byla implementována pouze část z algoritmů, které oblast neuronových sítí přináší. Možnosti rozšíření knihovny jsou velmi rozsáhlé. V této práci nebylo provedeno porovnání s jinými knihovnami zaměřující se na neuronové sítě. Detailní testování a porovnání s jinými knihovnami by jistě poodhalilo, kam by se vývoj knihovny měl ubírat.

Bibliografie

1. TURING, A. M. Computing Machinery and Intelligence. *Mind* [online]. 1950, roč. 59, č. 236, s. 433–460 [cit. 2020-05-30]. ISSN 00264423. Dostupné z: `http://www.jstor.org/stable/2251299`.
2. AGGARWAL, Charu C. *Neural networks and deep learning: a textbook*. Springer, 2018. ISBN 978-3-319-94462-3.
3. MCCULLOCH, Warren; PITTS, Walter. A Logical Calculus of Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*. 1943, roč. 5.
4. ROSENBLATT, Frank. The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. *Psychological Review*. 1958.
5. MINSKY, Marvin; PAPERT, Seymour. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.
6. WILAMOWSKI, Bogdan M.; IRWIN, J. David. *Intelligent Systems*. CRC Press, 2018.
7. AGGARWAL, Charu C. *Machine learning for text*. Springer, 2018. ISBN 3319735306.
8. BACHMAN, David. *Advanced Calculus Demystified*. New York, USA: McGraw-Hill. ISBN 978-0-07-148121-2.
9. LEMARÉCHAL, Claude. Cauchy and the Gradient Method. *Doc Math Extra*. 2010. Dostupné také z: `https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal%20claude.pdf`.
10. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.
11. RUSSELL, Stuart J.; NORVIG, Peter. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010. ISBN 9780136042594.

12. HINTON, Geoffrey; SEJNOWSKI, Terrence. *Unsupervised Learning: Foundations of Neural Computation*. MIT Press, 1999. ISBN 978-0262581684.

13. *What is the differene between categorical, ordinal and numerical variables?* [online] [cit. 2020-05-30]. Dostupné z: https://stats.idre.ucla.edu/other/mult-pkg/whatstat/what-is-the-difference-between-categorical-ordinal-and-numerical-variables/.

14. *Ordinal Regression using SPSS Statistics* [online] [cit. 2020-05-30]. Dostupné z: https://statistics.laerd.com/spss-tutorials/ordinal-regression-using-spss-statistics.php.

15. BONDY, John Adrian; MURTY, U. S. R. *Graph Theory with Applications*. North-Holland, 1976. ISBN ISBN 0-444-19451-7.

16. NUCLEAR SCIENCES, Faculty of; PHYSICAL ENGINEERING, CTU Prague Mathematical Modelling Group. *Template Numerical Library* [online] [cit. 2020-05-30]. Dostupné z: https://tnl-project.org.

17. *NVIDIA* [online] [cit. 2020-05-30]. Dostupné z: www.nvidia.com.

18. *The MIT License* [online] [cit. 2020-05-30]. Dostupné z: https://opensource.org/licenses/MIT.

19. RUBIO, Jose de Jesus. Parallelizing Backpropagation Neural Network Using MapReduce and Cascading Model. *Computational Intelligence and Neuroscience* [online]. 2016 [cit. 2020-05-30]. Dostupné z: https://doi.org/10.1155/2016/2842780.

20. PETHICK, Mark. Parallelization of a Backpropagation Neural Network on a Cluster Computer. *Computational Intelligence and Neuroscience* [online] [cit. 2020-05-30]. Dostupné z: http://www.cs.otago.ac.nz/staffpriv/hzy/papers/pdcs03.pdf.

21. POLIKAR, R. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*. 2006.

22. QIAO, Yu. *Modified National Institute of Standards and Technology database* [online] [cit. 2020-05-30]. Dostupné z: https://www.gavo.t.u-tokyo.ac.jp/~qiao/database.html.

23. KRIZHEVSKY, Alex. *Learning Multiple Layers of Features from Tiny Images* [online] [cit. 2020-05-30]. Dostupné z: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Příloha A

Obsah přiloženého flash disku

```
│ readme.txt ............................ stručný popis obsahu flash disku
├── src
│   ├── impl .................................... zdrojové kódy implementace
│   └── thesis ........................ zdrojová forma práce ve formátu LaTeX
└── text
    └── BP_Barinka_Simon_2020.pdf ............ text práce ve formátu PDF
```