VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

APROXIMACE HLUBOKÝCH NEURONOVÝCH SÍTÍ

DEEP NEURAL NETWORKS APPROXIMATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MARTIN STODŮLKA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. VAVERKA FILIP,**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Stodůlka Martin**
Program: Informační technologie
Název: **Aproximace hlubokých neuronových sítí**
Deep Neural Networks Approximation
Kategorie: Umělá inteligence
Zadání:

1. Seznamte se s frameworky pro implementaci modelů založených na hlubokých neuronových sítích (DNN), zaměřte se na sítě pro práci s obrazem.
2. Nastudujte problematiku návrhu aproximovaných hardwarových jednotek (celočíselné sčítačky a násobičky) a seznamte se s volně dostupnými knihovnami aproximovaných komponent. Zaměřte se na oblast efektivního vyhodnocení jejich parametrů (kvality) za pomoci konvenčního hardware.
3. Zvolte jeden z frameworků a otestujte jeho použití na vhodné klasifikační úloze.
4. Zvolený framework rozšiřte o možnost použití aproximovaných funkčních jednotek s cílem maximalizovat výkonnost.
5. Porovnejte chování (přesnost, rychlost, ...) několika vhodně zvolených DNN modelů na původním a rozšířeném frameworku.
6. Vyhodnoťte dosažené výsledky, zaměřte se především na kvalitu (přesnost) testovaných DNN modelů.

Literatura:

- Dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů zadání 1-3

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Vaverka Filip, Ing.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 9. května 2019

Abstrakt

Cílem mé práce je zjistit vliv a dopad aproximovaného počítání na přesnost hluboké neuronové sítě, konkrétně neuronové sítě pro klasifikaci obrazu. Pro implementaci neuronové sítě byla použita varianta frameworku Caffe zvaná Ristretto-caffe, která byla rozšířena o možnost použití aproximovaných operací v konvolučních vrstvách. pro používání aproximovaných komponent. Aproximované počítání bylo použito na násobení v dopředné propagaci při konvoluci. Jako aproximované komponenty byly zvoleny komponenty z knihovny Evoapproxlib.

Abstract

The goal of this work is to find out the impact of approximated computing on accuracy of deep neural network, specifically neural networks for image classification. A version of framework Caffe called Ristretto-caffe was chosen for neural network implementation, which was extended for the use of approximated operations. Approximated computing was used for multiplication in forward pass for convolution. Approximated components from Evoapproxlib were chosen for this work.

Klíčová slova

Hluboké neuronové sítě, klasifikace obrazu, aproximované počítání, aproximované komponenty, Evoapproxlib, C++, CUDA, Caffe, Ristretto-caffe

Keywords

Deep neural networks, image classification, approximated computing, approximated circuits, Evoapproxlib, C++, CUDA, Caffe, Ristretto-caffe

Citace

STODŮLKA, Martin. *APROXIMACE HLUBOKÝCH NEURONOVÝCH SÍTÍ.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vaverka Filip,

APROXIMACE HLUBOKÝCH NEURONOVÝCH SÍTÍ

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Filipa Vaverky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Martin Stodůlka
14. května 2019

Poděkování

Chtěl bych poděkovat hlavně vedoucímu mé práce Ing. Filipu Vaverkovi, za trpělivost a ochotu pro rychlou a stálou komunikaci.

Obsah

1 Úvod **3**

2 Neuronové sítě **4**
2.1 Základní princip 5
2.1.1 Neuron 6
2.1.2 Model sítě 7
2.2 Učení 8
2.2.1 Zpětné šíření chyby 8
2.2.2 Přeučení 10
2.3 Typy neuronových sítí 10
2.4 Konvoluční sítě 11
2.4.1 Architektura 11
2.4.2 Vrstvy 12

3 Urychlení operací neuronové sítě **14**
3.1 TPU 14
3.2 Aproximované počítání 18
3.2.1 Aproximované komponenty 18
3.2.2 Knihovna EvoApprox 19

4 Caffe **20**
4.1 Typy souborů pro práci s Caffe 21
4.1.1 Trénovací a testovací sada 21
4.1.2 Popis modelu sítě 21
4.1.3 Solver 23
4.1.4 Váhy a biasy 23
4.2 Vrstvy 23
4.2.1 Vstupní vrstva 23
4.2.2 Konvoluční vrstva 24
4.2.3 Pooling vrstva 24
4.2.4 ReLU 25
4.2.5 Plně propojená vrstva 25
4.2.6 Vrstvy pro měření a výpis informací 26
4.2.7 Vrstva chybové funkce 26
4.3 Konzolové prostředí a základní operace 27
4.3.1 Trénování 27
4.3.2 Finetuning 27
4.3.3 Testování 28

4.4 Rozšíření Ristretto . . . 28
4.4.1 Dynamic fixed point . . . 29
4.4.2 Minifloat . . . 29

5 Implementace a výsledky **30**
5.1 Implementace . . . 30
5.1.1 Kvantování modelu sítě . . . 30
5.1.2 Specifikování použití aproximované komponenty . . . 30
5.1.3 Realizace aproximovaných komponent . . . 31
5.1.4 Výpočet konvoluce . . . 31
5.2 Testovaný model . . . 32
5.2.1 Postup . . . 33
5.3 Výsledky . . . 34

6 Závěr **37**

Literatura **38**

A Obsah přiloženého paměťového média **41**

B Manuál **42**
B.1 Přeložení frameworku . . . 42
B.2 Použití skriptu pro generování vyhledávacích tabulek . . . 42

Kapitola 1

Úvod

Tato práce se zabývá využitím aproximovaného počítání v hlubokých neuronových sítí, konkrétně neuronovými sítěmi pro klasifikaci obrazu. Neuronové sítě jsou výpočetně náročné a nevyžadují velkou přesnost pro jejich výpočty. Většinou je snaha provádět výpočty nad čísly nižší bitové šířky jak typický `float` (32 bitů). Cílem této práce je zjistit dopad aproximovaného počítání na tyto neuronové sítě a implementovat výsledné řešení na GPU kvůli časové náročnosti.

V prvních dvou kapitolách popisuji funkce a principy neuronových sítí a aproximovaného počítání. U neuronových sítí je popsán jejich základní princip, z čeho vycházejí a jejich podoba biologickým strukturám. Dále je zde vysvětleno jejich rozšíření v podobě konvolučních neuronových sítí a jejich význam pro klasifikaci obrazu. Aproximované počítání je zde popsáno pro aproximované komponenty a knihovnu EvoApprox[10].

V posledních dvou kapitolách je popsána implementace a zobrazeny výsledky. V kapitole Caffe[4] je popsán celý framework i s jeho rozšířením Caffe-Ristretto[2] pro kvantizaci neuronových sítí. Zde je popsána i implementace rozšíření pro používání aproximovaných komponent a pomocných skriptů pro používání daného rozšíření. V poslední kapitole jsou uvedeny a popsány výsledky z experimentů provedených na různých modelech sítí s různými podmínkami.

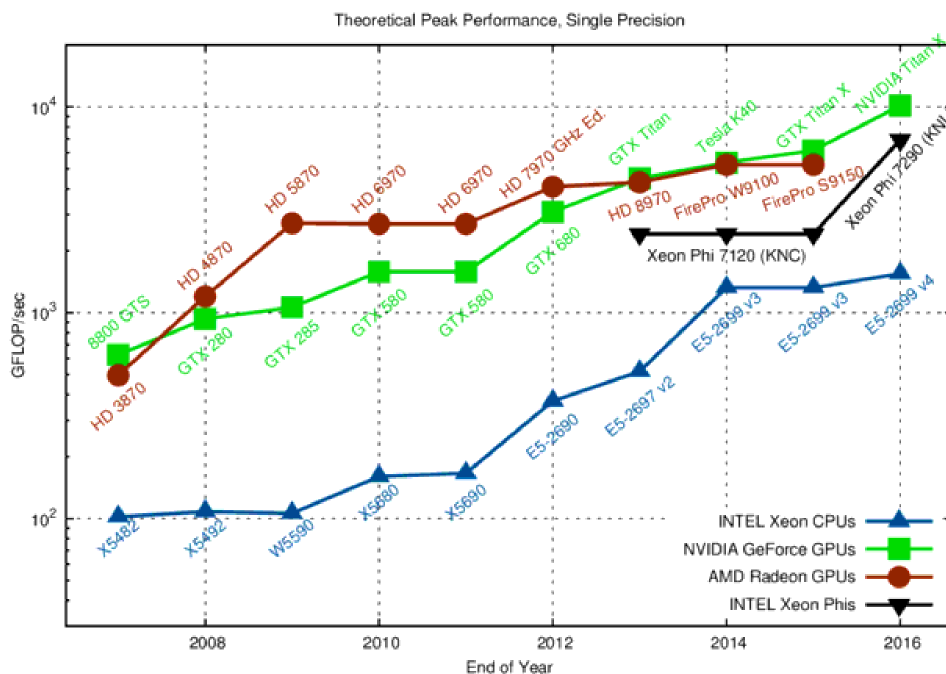
V závěru této práce jsou zhodnoceny její výsledky, přínos a doporučení na případný další vývoj.

Kapitola 2

Neuronové sítě

Neuronové sítě jsou používány v oblasti umělé inteligence pro řešení různých úloh, např. simulace nervových soustav, klasifikace a rozpoznávání obrazu a řeči, filtrování spamu a další. Jsou to sítě neuronů které jsou navzájem propojeny, přičemž neuron představuje základní jednotku pro stavbu neuronové sítě. Tento koncept vznikl z inspirace biologických neuronových sítí.

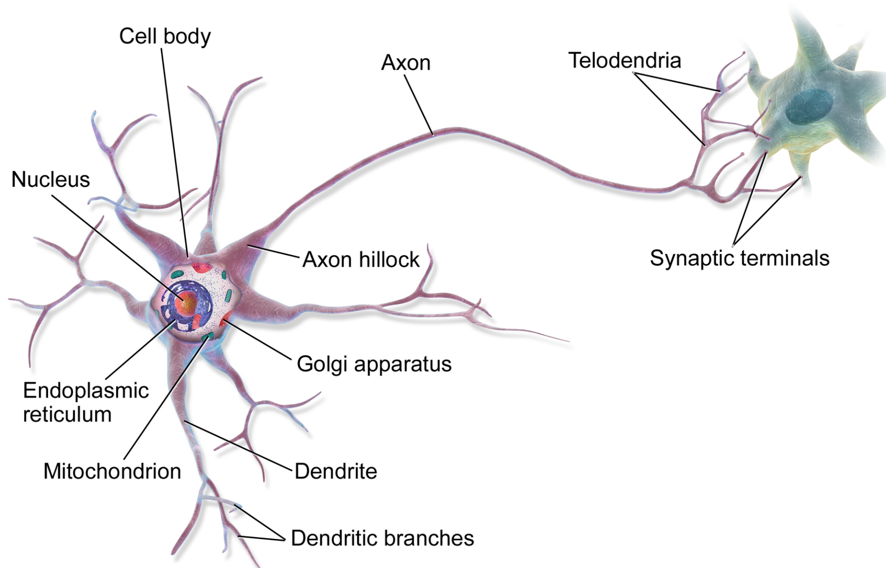
První výpočetní model neuronových sítí, vyvinutý Warren McCullochem a Walter Pittsem, vznikl již v roce 1943[7]. Vývoj neuronových pokračoval do roku 1969, kdy Marvin Minsky a Seymour Papert zveřejnili svou práci[8] ve které kritizovali a poukázali na nedostatky tehdejších neuronových sítí. Zájem o neuronové sítě byl obnoven v roce 1975, objevem využití algoritmu zpětného šíření chyby v neuronových sítích[17]. Od této doby vývoj neuronových sítí pokračoval. V posledním desetiletí byla neuronovým sítím věnována velká pozornost, obzvlášť díky vzrůstu výpočetní síly a použití grafických akcelerátorů.



Obrázek 2.1: Růst výkonu CPU a GPU v 32 bitových floating point oeprací v posledním desetiletí. Převzato z [13]

2.1 Základní princip

Umělé neuronové sítě vycházejí z biologických neuronových sítí vyskytujících se v mozcích obratlých a některých bezobratlých živočichů. Biologické neuronové sítě jsou složeny z neuronů, které jsou navzájem propojeny synapsemi které vstupují do dendritů. Každý neuron může být propojen s více neurony najednou a zároveň může přijímat informace od více neuronů ze svých dendritů.



Obrázek 2.2: Stavba biologického neuronu. Převzato z [22]

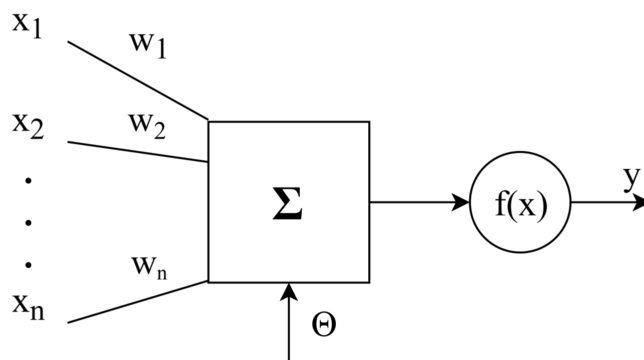
Každý neuron v reakci na vzruchy z jeho dendritů šíří vzruchy přes jeho axon do synapsí propojených s ostatními neurony. Synapse můžou mít excitační nebo inhibiční charakter. Excitační posilují aktivitu napojeného neuronu a inhibiční provádějí přesný opak. Aktivita v biologických neuronových sítích je vyjádřena pomocí napětí. Aktivita neuronu je vyhodnocena výslednou sumou napětí z jeho dendritů. Pro aktivaci neuronu musí tato suma překročit tzv. prahový potenciál. Tímto šířením se postupně dojde k poslednímu neuronu(ům), který ovládá určitou funkci organismu na základě jeho výstupu.

Umělé neuronové sítě jsou složeny z umělých neuronů, jejich vah, aktivačních funkcí a propojení. V podstatě jsou to simulace biologických neuronových sítí, avšak není cílem u většiny sítí podrobně simulovat biologickou neuronovou síť, ale její mechanismus zpracovávání informací. Charakter sítě je dán propojením neuronů, jejich vah a volby aktivačních funkcí.

Propojení a umístění neuronů a volba aktivačních funkcí je otázka návrhu sítě. Váhy jsou získány trénováním sítě. Trénování má za úkol nalézt takové váhy sítě, aby síť převáděla vstup na požadovaný výstup. Na příklad u rozpoznávání obrazu očekáváme na vstupu obrázky a na výstupu jejich kategorii např. kočka, pes atd. Každá neuronová síť požaduje nějaká trénovací data, aby bylo možno ji natrénovat.

2.1.1 Neuron

Neuron[15] představuje základní jednotku neuronové sítě. Existuje více modelů neuronu v neuronových sítích. Zde je popsán McCulloch-Pitts model[7], který je nejpoužívanější. Každý neuron má své vstupy $(x_1, x_2, x_3, ..., x_n)$ s příslušnými synaptickými váhami $(w_1, w_2, w_3, ..., w_n)$, práh Θ který slouží jako parametr pro aktivační funkci $f(x)$ a jeden výstup y, který může být napojený na více neuronů. Data na vstupech a výstupech neuronů můžou být spojitá nebo binární.



Obrázek 2.3: Schéma neuronu

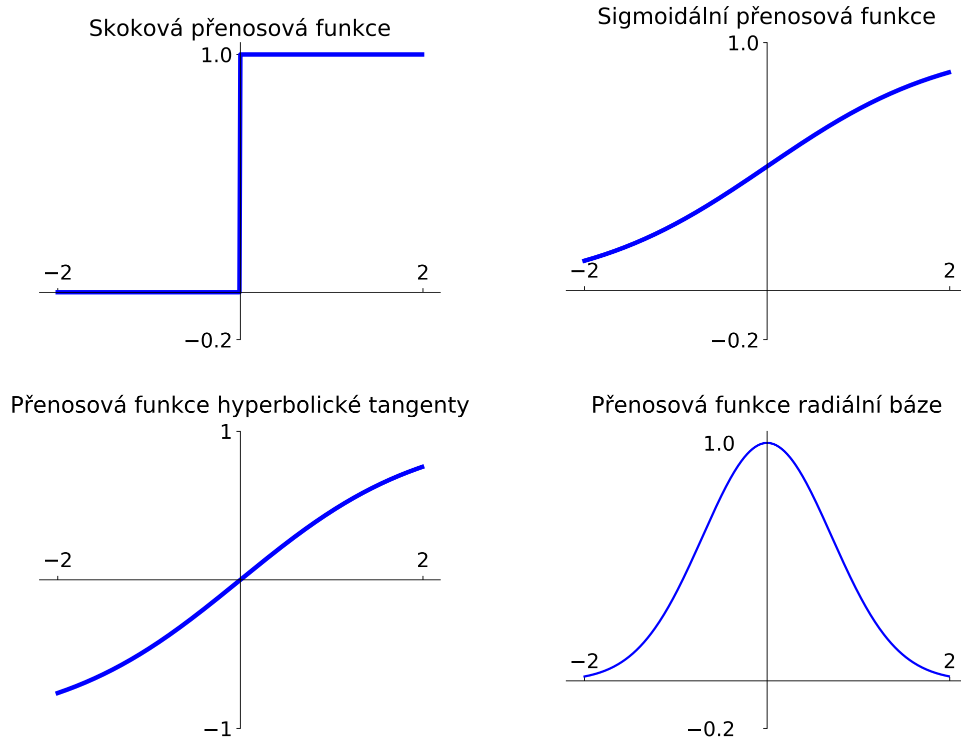
Výstup neuronu se vypočítá jako:

$$y = f(\sum_{i=0}^{n}(x_i w_i + \theta)), \tag{2.1}$$

kde n je počet vstupů(vah).

Existuje více druhů aktivačních funkcí. Aktivační funkce se volí na základě typů neuronů a typu neuronové sítě. Funkce můžou být spojité i nespojité. Příklady aktivačních funkcí[18]:

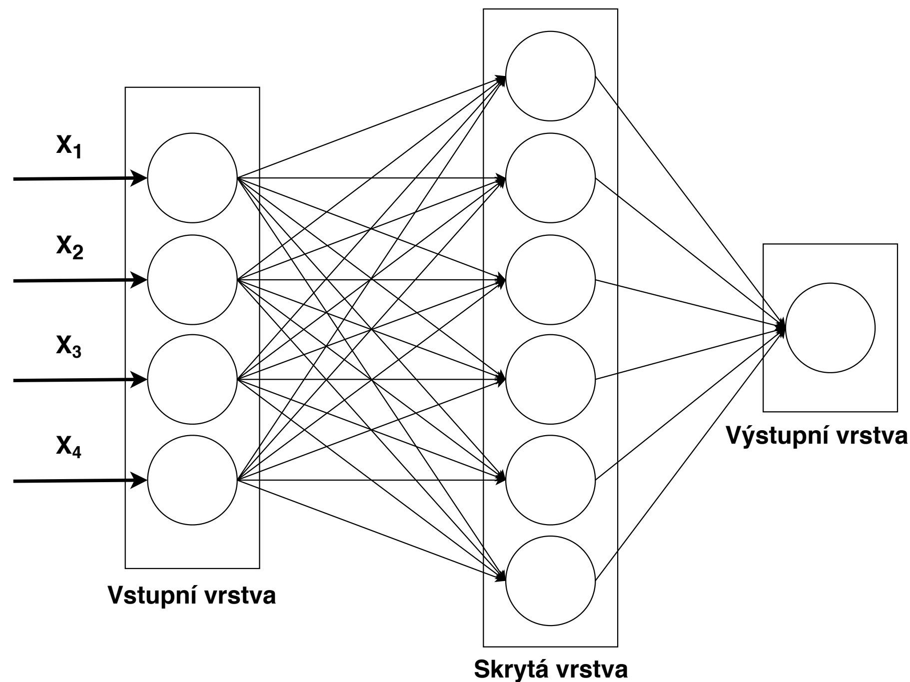
- **Skoková aktivační funkce** – Jednoduchá funkce která pro $x < \theta$ má hodnotu $f(x) = 0$ a pro $x \geq \theta \rightarrow f(x) = 1$. Využívá se u binárních neuronů.
- **Sigmoidální aktivační funkce** – Funkce ve tvaru: $f(x) = \frac{1}{1+e^{-x}}$. Oproti skokové aktivační funkci je spojitá na celém svém definičním oboru.
- **aktivační funkce hyperbolické tangenty** – Funkce ve tvaru $f(x) = \frac{2}{1+e^{-x}} - 1$. Její hodnoty se blíží -1 v $-\infty$ a jedničce v ∞. Pro $f(0)$ je hodnota 0.
- **aktivační funkce radiální báze** – Funkce ve tvaru $f(x) = e^{-x^2}$. Podobná Gaussově rozdělení, avšak nejedná se o rozdělení pravděpodobnosti.



Obrázek 2.4: Grafy různých aktivačních funkcí

2.1.2 Model sítě

Neuronová síť se skládá z neuronů a jejich parametrů (váhové koeficienty, prahové hodnoty, volba a parametry aktivační funkce). Sítě můžeme třídit do vícevrstvých a jednovrstvých. Do jednovrstvých sítí patří např. Hopfieldova síť a perceptron. Vícevrstvých sítí je mnohem více např. síť perceptronů, konvoluční sítě. Každá neuronová síť může být popsána pomocí grafu. Grafy neuronových sítí se většinou používají jako vizuální reprezentace stavby sítě (bez synaptických vah a prahových hodnot). V složitějších modelech jsou neurony členěny do vrstev, které jsou v grafu pak znázorněny jako bloky místo jednotlivých neuronů.



Obrázek 2.5: Příklad grafu neuronové sítě

2.2 Učení

Učení (trénování) je jednou z nejdůležitějších operací neuronové sítě. Bez tohoto konceptu by neuronové sítě ztráceli význam v řešení složitějších úloh. Veškeré informace neuronové sítě jsou uloženy v jejích synaptických váhách a prahových hodnotách, což určuje její chování na výstupu při určitých vstupních hodnotách. Cílem učení je najít takové parametry sítě, aby byla co nejvíc minimalizovaná chybová funkce, která nám hodnotí výpočetní chybu neuronové sítě v jedné skalární hodnotě. Učení může probíhat sekvenčně nebo dávkově. Při dávkovém zpracování jsou váhy změněny až po zpracování celé dávky vstupů. Jsou tři typy učení:

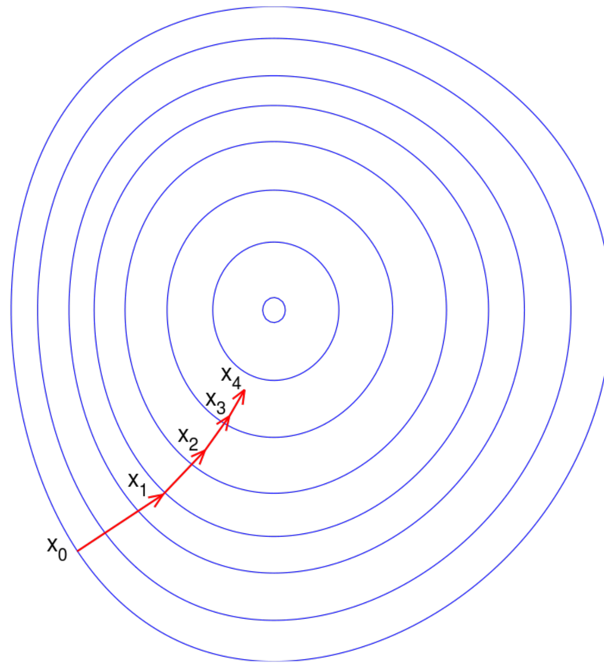
- **Učení bez učitele** – Při tomto typu učení nám není znám výsledek naší sítě. Síť se sama snaží nastavit své váhy a prahy tak, aby na podobné vstupy reagovala stejným výstupem.
- **Učení s učitelem** – Učení s učitelem je metoda, která používá trénovací data a požadovaný výstup pro trénování sítě. Tento typ učení se používá nejvíce v oblastech rozpoznávání řeči a obrazu.
- **Zpětnovazební učení** – Síť nepracuje se žádnou sadou dat. Místo toho je použit agent, který interaguje se svým okolím a tyto akce jsou sledovány a ohodnoceny. Cílem je natrénovat danou síť, aby co nejlépe reagovala na určité podněty z okolí.

2.2.1 Zpětné šíření chyby

Jde o nejvíce používaný algoritmus pro učení vícevrstvých neuronových sítí. Tato metoda počítá gradient chybové funkce, na základě kterého počítá změnu parametrů sítě, tudíž

spadá pod kategorii učení s učitelem. Volba chybové funkce závisí na typu a parametrech neuronové sítě.

Algoritmus zpětné šíření chyby je součástí algoritmu gradient descent. Zpětné šíření chyby počítá pouze gradient chybové funkce. Iterativní přístup, kde se snažíme dosáhnout minima chybové funkce postupným měněním parametrů se nazývá gradient descent. Specificky se používá stochastický gradient descent, který je aproximací normálního gradient descentu. Jeho výhodou je, že potřebuje pouze jeden či více trénovacích vzorků. Gradient descent požaduje celou trénovací množinu pro spočítání jedné iterace.



Obrázek 2.6: Ilustrace gradient descent. Převzato z [21]

Učení probíhá následovně. Před začátkem učení je potřeba nastavit počáteční hodnoty všech vah a prahových hodnot a zvolit počáteční koeficient učení. Počáteční hodnoty vah a prahových hodnot jsou většinou náhodně generovány. Trénovací data jsou postupně přivedeny na vstup sítě a propagovány až na výstup. Z výstupu je zjištěn výsledek, který je porovnán s očekávaným výsledkem. Na konec je použit algoritmus zpětného šíření chyby, který postupně změní váhy všech neuronů tak, aby se sít přizpůsobila vstupu. Tento proces se opakuje dokud nedosáhne jistého kritéria (dosáhnutí počtu iterací, požadované přesnosti, požadované hodnoty chybové funkce). U algoritmu zpětného šíření chyby je vysoká přesnost důležitá, a proto se tato práce nezajímá její aproximací. S každou iterací se koeficient učení η snižuje.

Rovnice pro změnu parametrů sítě:

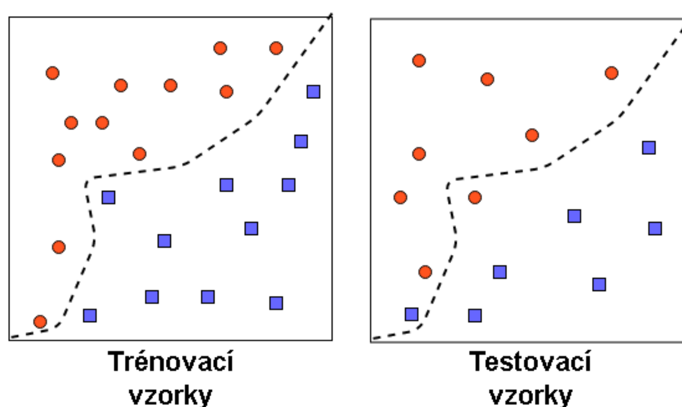
$$
\begin{aligned}
w_{ij}(t+1) &= w_{ij}(t) - \eta \frac{\partial C}{\partial w_{ij}}, \\
\tau_j(t+1) &= \tau_j(t) - \eta \frac{\partial C}{\partial \tau_j},
\end{aligned}
\tag{2.2}
$$

Kde w_{ij} je váha neuronu j z výstupu neuronu i, τ_j je prahová hodnota neuronu j, t je diskrétní časový parametr (číslo iterace) a C je chybová funkce.

Důležitým faktem je, že vstupní data se na výstup šíří dopředně (dopředná propagace), kdežto algoritmus zpětného šíření chyby mění parametry sítě zpětně (zpětná propagace) na základě výsledku dopředné propagace.

2.2.2 Přeučení

Při nevhodném zvolených parametrech pro učení se neuronová síť může dostat do stavu přeučení. Přeučení může být způsobeno trénováním sítě na malém vzorku dat s vysokým počtem iterací. V takové situaci začne síť rozpoznávat ve vstupních datech i šum a jiné detaily, což zapříčiní, že nová vstupní data budou rozpoznána s menší přesností.



Obrázek 2.7: Příklad přeučení, napravo síť natrénovaná podle určitých vzorků, nalevo použití stejné natrénované sítě na nové testovací vzorky

Přeučení není jediný efekt který zhorší přesnost neuronové sítě, např. špatně zvolený počáteční koeficient učení může zapříčinit uváznutí chybové funkce v lokálním minimu.

2.3 Typy neuronových sítí

Existuje mnoho typů neuronových sítí. Každý typ sítě je přizpůsoben jiné třídě problémů. Zde je uvedeno několik typů[19]:

- **Dopředné** – Jeden z prvních typů neuronových sítí. Data se šíří od vstupní vrstvy přes skrytou vrstvu až do výstupní vrstvy. Sítě tohoto typu jsou trénovány pomocí algoritmu zpětného šíření chyby.
- **Rekurentní** – Rekurentní sítě jsou podobné dopředným s tím rozdílem, že přijímají jako vstup také předchozí trénovací data. Tato vlastnost jim umožňuje vnímat kontext mezi daty. Jsou používané obzvláště v oblasti strojového překladu.

- **RBF sítě** – RBF (Radial basis function) sítě používají RBF jako aktivační funkci, které počítají výstup na základě vzdálenosti od zvolených středů. Jedná se o jednovrstvou síť, kde jediná skrytá vrstva je složena z neuronů používajících RBF. Tento typ sítí se používá např. pro aproximace funkcí a klasifikace.
- **Modulární** – Síť která se skládá z několika podsítí. Každá síť operuje nezávisle na ostatních (moduly), přičemž všechny tyto sítě jsou spravovány pod jedinou sítí. Tato síť přijímá pouze výstupy svých modulů a rozhoduje o konečném výstupu celé sítě.

2.4 Konvoluční sítě

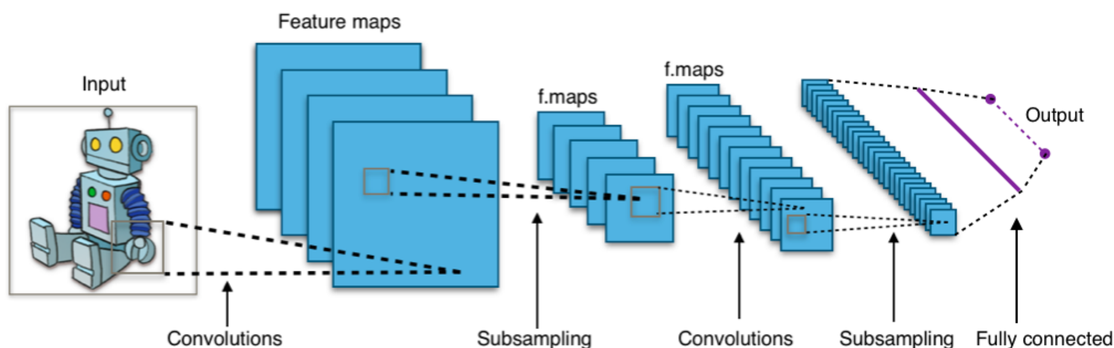
Konvoluční síť je typ hluboké neuronové sítě používané pro rozpoznávání obrazu. Skládá se ze vstupní vrstvy, skrytých vrstev a výstupní vrstvy. Jde o typ vícevrstvých perceptronů, avšak oproti nim jsou konvoluční sítě méně komplexnější (každý neuron není propojen se všemi ostatními). Oproti ostatním typům sítí jsou rozměry jejich vrstev uspořádány do tří dimenzí, protože obrázky sestávají z více kanálů (RGB). Každá vrstva musí transformovat svůj výstup tak, aby odpovídal rozměrům vstupu následující vrstvy.

Tento typ sítí vznikl z pozorování zrakové části mozku živočichů. Signály ze zrakového nervu jdou přímo do první vrstvy neuronů zrakové části mozku. Zde každý nerv je napojen na určitou oblast našeho zorného pole. Oblasti těchto nervů se můžou překrývat. Výstupy těchto nervů jsou dále šířeny do dalších vrstev nervového pole. Každý neuron v těchto vrstvách je schopen rozpoznat jistý příznak a šířit tuto informaci dál. První vrstvy rozpoznávají jednoduché příznaky např. vodorovná a svislá čára. Z jednoduchý příznaků se skládají složitější, až nakonec v poslední vrstvě je objekt klasifikován.

Toto získávání příznaků odpovídá konvoluci obrazu s konvolučním jádrem. Konvoluční neuronové sítě obsahují konvoluční vrstvy, které pomocí konvoluce získávají příznaky z obrazu. Hodnoty konvolučních jader jsou nalezeny trénováním sítě. Každá konvoluční vrstva zpracovává jiné úrovně příznaků. První vrstvy zpracovávají jednoduché příznaky, které se pak skládají do složitějších.

2.4.1 Architektura

Jako každá neuronová síť, konvoluční síť obsahuje vstupní a výstupní vrstvu. Skryté vrstvy se skládají z konvolučních, pooling a aktivačních vrstev. Po každé konvoluční vrstvě obvykle následuje pooling vrstva, která redukuje rozměry předchozí vrstvy pro zjednodušení výpočtů. Před výstupní vrstvou je plně propojená vrstva, která slouží ke klasifikaci příznaků z předchozí vrstvy. Poslední výstupní vrstva používá softmax funkci pro převedení vstupu předchozí vrstvy do pravděpodobnostního rozložení (součet všech prvků výstupního vektoru musí být roven jedné).

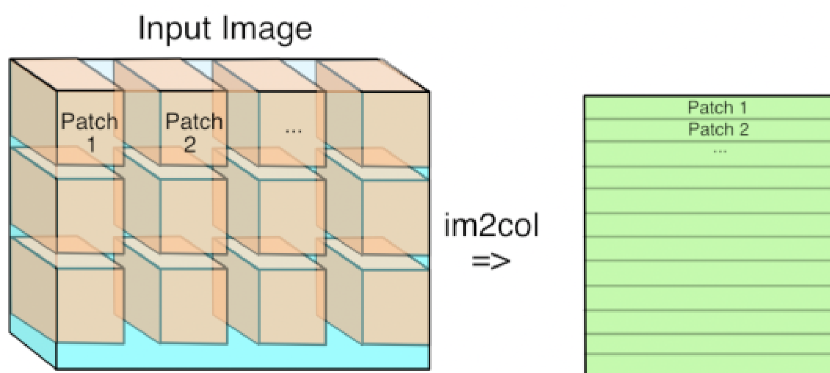


Obrázek 2.8: Architektura konvoluční neuronové sítě. Převzato z [20]

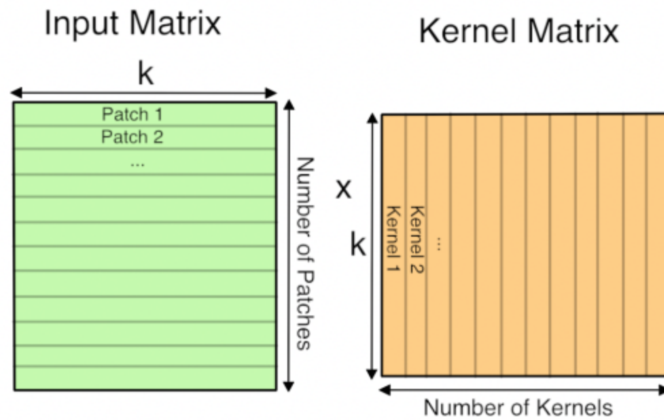
2.4.2 Vrstvy

Konvoluční síť má dvě specialní vrstvy[14] oproti ostatním sítím. Konvoluční a pooling vrstvy jsou na začátku každé konvoluční sítě. Dohromady slouží k extrakci příznaků ze vstupu. Pooling a konvoluční vrstvy se většinou nachází po dvojicích, přičemž pooling vrstva následuje po konvoluční.

- **Konvoluční vrstva** – Cílem této vrstvy je získat příznaky ze vstupního obrázku. Extrakce příznaků je provedena pomocí konvoluce, které je parametrizována rozměry filtru, krokem a odsazení okrajů (padding). Každá vrstva obsahuje svůj vlastní filtr, který je inicializován náhodnými hodnotami. Optimální hodnoty tohoto filtru jsou v průběhu trénování nalezeny. Konvoluce je počítána pomocí maticového násobení[16], jelikož maticové operace jsou vysoce optimalizované (BLAS, cuDNN) a umožňují efektivnější přístup k paměti. Pro maticové násobení je třeba oblasti obrázku a filtry transformovat. Každá podoblast obrázku je transformována jako jeden řádek matice a každý filtr je transformován jako sloupec druhé matice. Násobením těchto matic je provedena konvoluce.

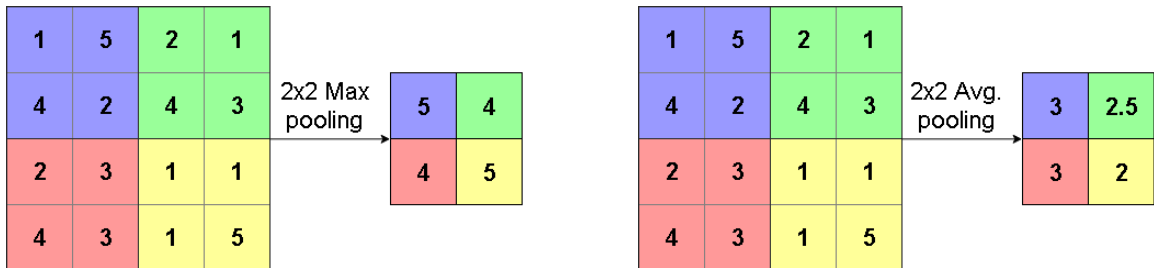


Obrázek 2.9: Transformace podoblastí obrazu do matice. Převzato z [16]



Obrázek 2.10: Konvoluce jako maticové násobení. Převzato z [16]

- **Pooling vrstva** – Tato vrstva snižuje rozměry předchozí vrstvy (obvykle konvoluční). Snížením rozměrů se snižuje potřebná výpočetní síla. Používají se dva typy a to max pooling a average pooling. Max pooling vybírá maximum z podoblastí svého vstupu. Touto operací je také vybrán nejsilnější výsledek, čímž potlačuje šum. Average pooling počítá průměr každé podoblastí svého vstupu. Tímto vyhlazením je také schopen potlačit šum.



Obrázek 2.11: Nalevo ukázka max pooling, napravo average pooling ve 2D

Kapitola 3

Urychlení operací neuronové sítě

3.1 TPU

Neuronové sítě vyžadují velký výpočetní výkon a to nejen na trénování, ale i při nasazení (počítání inference). S rostoucími nároky na neuronové sítě byla snaha vyvíjet specializované akcelerátory, které by urychlily tyto výpočty a zároveň snížily příkon relativně k výkonu. Dosud se jako akcelerátory používaly grafické akcelerátory, které umožnili značné urychlení výpočtů oproti CPU.

Grafické akcelerátory jsou specializované na úlohy, která se dají rozdělit na spoustu nezávislých stejných částí. Hlavní výhoda GPU oproti CPU je v počtu jader a vysoké paměťová propustnost, GPU obsahuje až stovky jader. Navíc každý Streaming Multiprocesor (SM/jádro) obsahuje 32 ALU jednotek (CUDA jader). Instrukce jsou prováděny v tzv. warpech, kde každý warp spouští 32 vláken (SIMT - single instruction multiple threads).

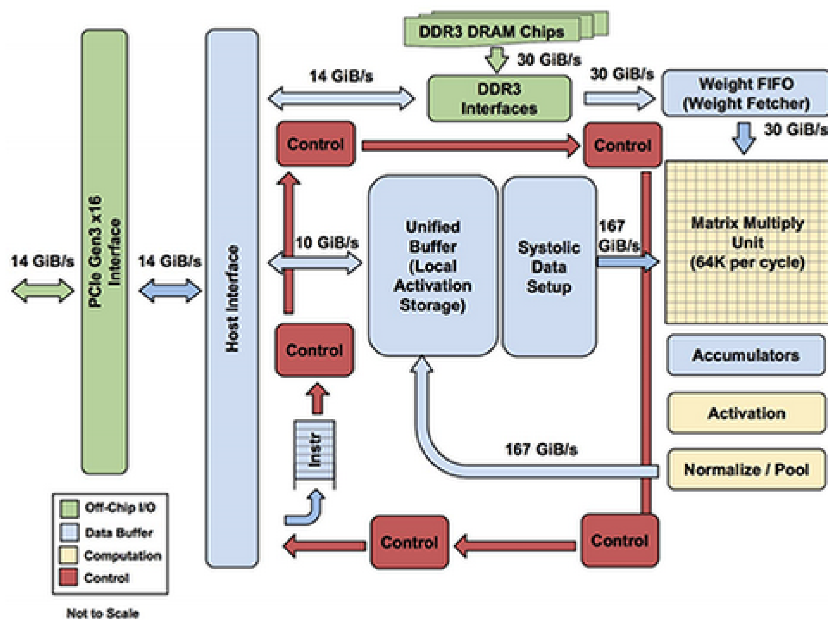
I přes značně zlepšený výpočetní výkon díky grafickým akcelerátorům se inženýři dívali po lepším řešení, které by také přinášelo lepší efektivitu (výkon na watt). Pro počítání inference jsou potřebné následující operace:

- **Násobení** – Násobení vstupů neuronu x_i a jeho vah w_i.
- **Sčítání** – Suma všech vynásobených vah a vstupů neuronu a přičtení prahové hodnoty θ: $\sum_{i=0}^{n}(x_i w_i + \theta)$.
- **Aktivační funkce** – Funkce aplikovaná na výslednou sumu: $f(\sum_{i=0}^{n} x_i w_i + \theta)$

Protože neuronové sítě provádějí tyto operace mnohokrát pro výpočet inference, bylo by vhodné je hardwarově implementovat jako ASIC. Integrované obvody typu ASIC jsou speciálně navržené obvody pro konkrétní aplikace. Jejich výhodou je, že jsou nejvýkonnější a nejefektivnější pro aplikace na kterou byly navrženy. Jejich nevýhodou jsou vysoké náklady na vývoj a výrobu (pokud se jich vyrábí malé množství).

Jejich efektivita je dána nízkoúrovňovou implementací a jejich specializací. CPU a grafické akcelerátory jsou navrženy pro více účelů a obsahují nepotřebné části pro tyto výpočty nebo příliš obecné komponenty. Například suma se dá spočítat sekvenčně pomocí jedné sčítačky s lineární složitostí nebo pomocí stromu sčítaček s logaritmickou časovou složitostí.

Společnost Google navrhla svoji vlastní implementaci TPU[5] (Tensor processing unit). Jedná se o CISC jednotku která je schopna vykonávat instrukce maticového násobení a určitých typů aktivačních funkcí. TPU a jeho instrukční sada byla navržena tak, aby se dala používat s jakoukoliv aplikací a pro jakoukoliv síť. Používaná není jen pro inferenci, ale i



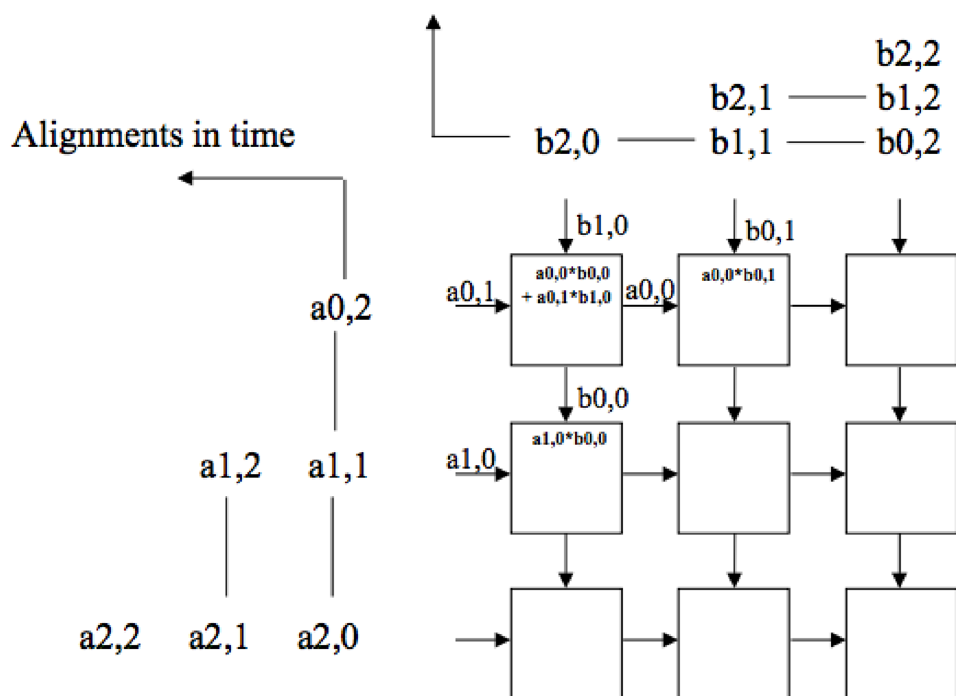
Obrázek 3.1: Blokové schéma TPU. Převzato z [5]

pro trénování.

Důležité části TPU jsou:

- **Matrix Multiplier Unit (MXU)** – Jednotka realizující maticové násobení. Obsahuje velké množství jednotek uspořádaných v matici, které násobí a zároveň sčítají výsledky tohoto násobení. Přijímá osmi bitové vstupy.
- **Unified Buffer (UB)** – Rychlá paměť typu SRAM s kapacitou 24 MB.
- **Activation Unit (AU)** – Jednotka schopna provádět určité aktivační funkce.

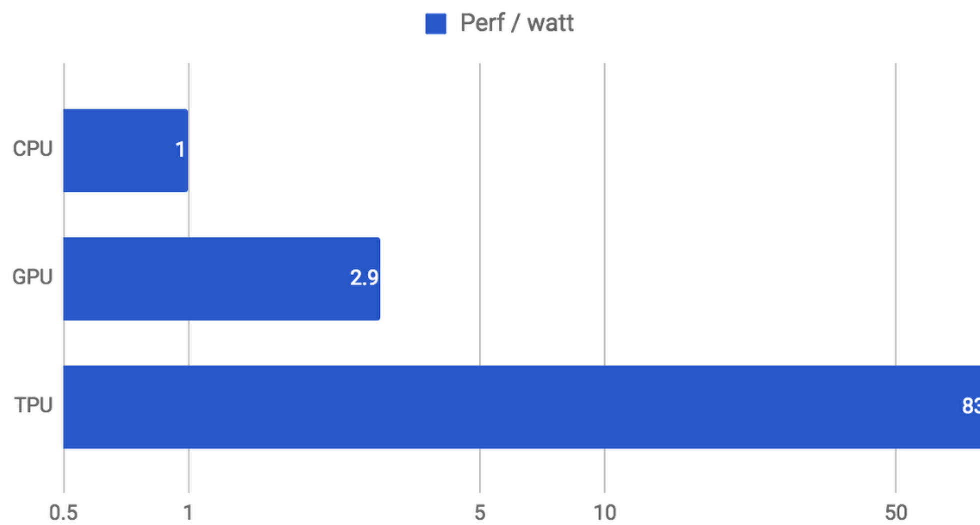
MXU je masivně paralelní jednotka schopná zpracovávat tisíce operací v každém taktu. Na rozdíl od CPU a grafických akcelerátorů pracuje na principu systolického pole (MISD podle Flynnovi klasifikace paralelních systémů). Název systolického pole pochází ze slova systola, které v latině znamená srdeční kontrakce. V této architektuře se data postupně propagují po jednotlivých jednotkách, které zároveň nad nimi provádějí určitou operaci.



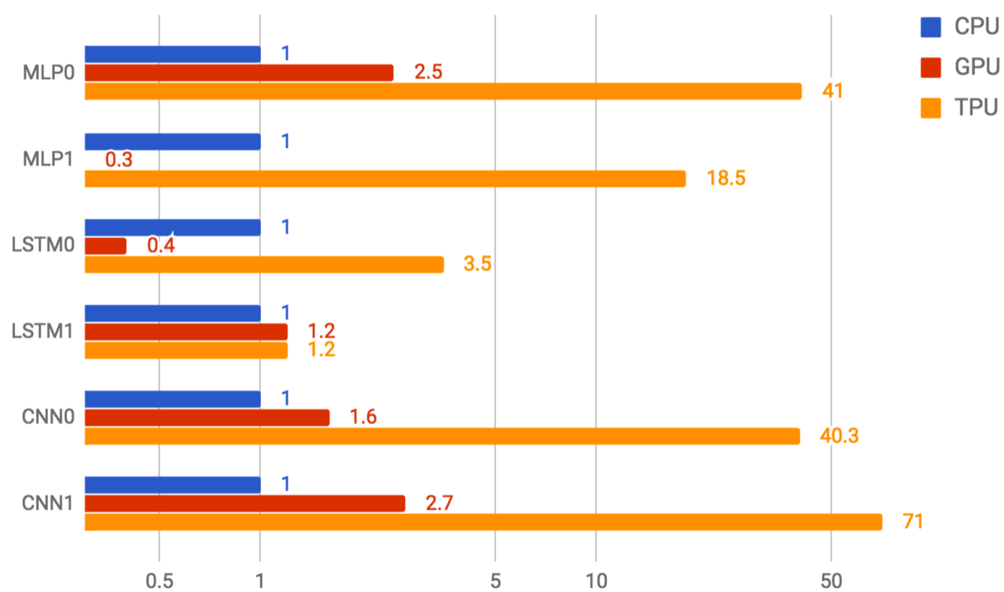
Obrázek 3.2: Příklad systolického pole (3x3). Převzato z [12]

TPU operuje na frekvenci 700 MHz, ovšem oproti CPU a grafickým akcelerátorům TPU zpracovává mnohem více operací každý takt. Díky tomuto faktu je TPU mnohem rychlejší a efektivnější než CPU nebo grafický akcelerátor.

Zařízení	Počet operací v cyklu
CPU	jednotky
CPU (s AVX/SSE)	desítky
GPU	desítky tisíc
TPU	stovky tisíc



Obrázek 3.3: Srovnání výkonu na watt. Převzato z [5]



Obrázek 3.4: Srovnání rychlosti CPU, GPU a TPU na různých sítích. Převzato z [5]

3.2 Aproximované počítání

Běžně v různých procesorech a vysokoúrovňových obvodech obecně používáme standardní realizace obvodů pro různé aritmeticko-logické operace. Je snaha vytvářet nové typy těchto obvodů, které mají lepší vlastnosti než ostatní (menší počet hradel, vyšší rychlost). Každý typ obvodu má své pro a proti, neexistuje nejlepší řešení, navíc některé aritmeticko-logické operace jsou náročnější než jiné. Tento trend nemůže pokračovat do nekonečna.

Smysl aproximovaného počítání[9] spočívá v tom, že ne každá aplikace vyžaduje přesné výsledky. V některých případech je chyba zavedena už při získávání dat (např. ze senzorů), jinde nám chyba z principu nevadí (např. ztrátová komprese - JPEG, MP3). Při zavedení aproximovaných počítání (realizované aproximovanými komponenty) můžeme snížit plochu na čipu, snížit zpoždění nebo snížit spotřebu.

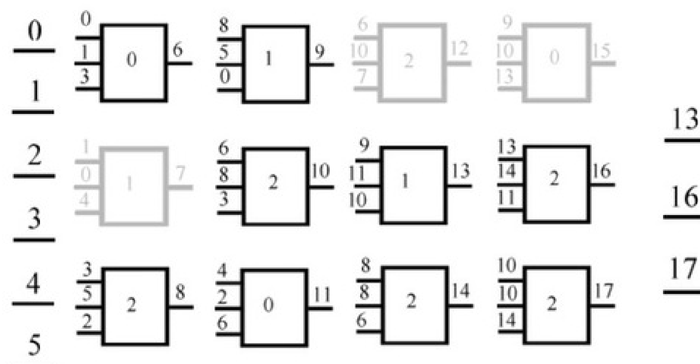
Neuronové sítě jsou také aproximace funkcí.

3.2.1 Aproximované komponenty

Aproximované komponenty vykonávají aritmeticko-logické operace, ovšem s odchylkou od skutečného výsledku. Jejich výhoda oproti konvenčním komponentám je jejich nižší spotřeba a menší plocha na čipu nebo vyšší rychlost. Pro jednotlivé typy komponent (sčítačka, násobička, atd.) je několik komponent které mají různé vlastnosti.

Tyto komponenty byly vytvořeny pomocí genetických algoritmů, specificky pomocí CGP (Cartesian Genetic Programming), kde se do mřížky postupně dosazují nebo modifikují hradla a specifikují se vstupy těchto hradel.

0 1 3 *0* 1 0 4 *1* 3 5 2 *2* 8 5 0 *1* 6 8 3 *2* 4 2 6 *0*
6 10 7 *2* 9 11 10 *1* 8 8 6 *2* 9 10 13 *15* 13 14 11 *16* 10 10 14 *2* 13 16 17



Obrázek 3.5: CGP - Cartesian Genetic Programming Na levé straně jsou počáteční vstupy a na pravé straně výstupy. Převzato z [11]

3.2.2 Knihovna EvoApprox

EvoApprox[10] je knihovna aproximovaných n bitových komponent. Konkrétně se jedná o násobičky a sčítačky. Veškeré komponenty byly vytvořeny pomocí CGP (Cartesian Genetic Programming). Zdrojové kódy pro tyto komponenty jsou k dispozici v jazyce C, Verilog a Matlabu. Pro účely této práce byly použity osmi bitové násobičky z této knihovny, specificky jejich implementace v jazyce C. Navíc je poskytnut skript pro přeložení EvoApprox do python modulu. Pro každou komponentu jsou poskytnuty následující statistiky:

- **HD** – Hammingova vzdálenost
- **EP** – Pravděpodobnost výskytu chyby
- **MAE** – Střední absolutní chyba
- **MSE** – Střední kvadratická chyba
- **MRE** – Střední relativní chyba
- **WCE** – Nejhorší chyba
- **WCRE** – Nejhorší relativní chyba

Kapitola 4

Caffe

Caffe[4] je framework pro hluboké neuronové sítě. Framework Byl vyvinut v roce 2014 Yangqingem a Berkeley Artificial Intelligence Research (BIAR) a dále volně rozvíjeno různými přispěvateli. Celé prostředí je open source pod BSD licencí a je volně přístupné. Repozitář frameworku je na githubu[1].

Caffe je napsáno v jazyce C++ a CUDA. Celý projekt je multiplatformní, přeložitelný na různých linuxových distribucích, OS X a Windows. Používá mnoho závislostí

- Potřebné závislosti
  - **CUDA** – platforma pro grafické akcelerátory od firmy Nvidia pro akcelerované počítání
  - **BLAS** – knihovna pro vektorové a maticové operace. Existuje více implemtací této knihovny např. ATLAS, MKL a OpenBLAS.
  - **Boost** – balík knihoven pro C++ které implementující různé pomocné funkce v oblasti lineární algebry, generace náhodných čísel, multithreadingu, zpracování obrazů a mnoho dalších.
  - Protobuf – obecně mechanismus pro serializaci dat. V případě téhle závislosti její C++ implementace.
  - Glog – knihovna pro usnadnění vypisování informací na výstup (logování)
  - Gflags – knihovna pro zpracování argumentů příkazové řádky
  - Hdf5 – knihovna pro práci s formátem HDF. HDF umožňuje pracovat s objemnými a rozmanitými daty.
- Volitelné závislosti
  - **OpenCV** – otevřená multiplatformní knihovna pro práci s obrazem.
  - **CuDNN** – knihovna implementující základní optimalizované funkce pro hluboké neuronové sítě pro grafické akcelerátory od firmy Nvidia.
  - Lmdb a Leveldb – knihovny pro práci s databázemi typu klíč-hodnota.
  - **Python** – skriptovací programovací jazyk
  - Numpy – modul pythonu pro náročné výpočetní operace

[1] https://github.com/BVLC/caffe

- **MATLAB** – Programové prostředí a skriptovací programovací jazyk určený pro vědeckotechnické účely, simulace, paralelní výpočty apod.

Caffe umožňuje definice hlubokých neuronových sítí pomocí "proto" souborů, trénovaní sítí, nasazení natrénovaných sítí. Také disponuje značnou rychlostí a umožňuje akceleraci veškerých výpočtů pomocí grafických akcelerátorů. Framework je možné používat jako konzolovou aplikaci, modul pro Python (Pycaffe) nebo MATLAB (Matcaffe).

4.1 Typy souborů pro práci s Caffe

Celý framework pracuje s různými typy souborů, kde každý typ souboru slouží pro specifickou funkci. V souborech je uložena trénovací a testovací sada, popis modelu sítě, popis postupu a parametrů trénování sítě (solver) apod. Některé typy souborů jsou definovány pomocí jiných knihoven. V následující částí bude každý typ souboru popsán.

4.1.1 Trénovací a testovací sada

Jsou podporovány tři formáty pro vstupní trénovací a testovací data neuronové sítě. Mezi tyto formáty patří Leveldb, Lmdb a HDF5. Obecně jsou to databáze typu klíč-hodnota, kde klíč je číslo popisující do které kategorie obrázek (hodnota) patří.

Tyto databáze lze vytvořit pomocí tohoto frameworku, pokud byl kompilován s OpenCV. V repozitáři je obsažen skript (caffe/examples/imagenet/create_imagenet.sh), který využívá nástroj vytvořený kompilací Caffe pod názvem convert_imageset. Tento nástroj očekává cestu k adresáři obsahující trénovací data, cestu k adresáři obsahující testovací data a metadata k databázi obrázků. Samotné obrázky jsou podporované v různých formátech (konkrétně formáty které podporuje OpenCV). Tool je taky schopen každý obrázek škálovat (změnit velikost) podle zadaných parametrů. Jeho výstupem jsou dvě Lmdb databáze, jedna obsahující testovací sadu, druhá trénovací sadu. Metadata k databázi obrázku je jednoduchý textový soubor ve formátu:

```
<Nazev souboru> <Cislo kategorie>
kocka1.jpg 1
pes1.jpg 2
pes2.jpg 2
kocka2.jpg 1
.
.
.
```

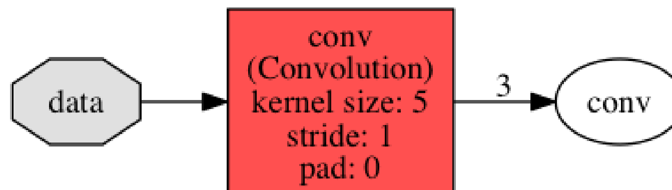
Výpis 4.1: Příklad metadat testovací/trénovací databáze

4.1.2 Popis modelu sítě

Model sítě je popsán v souborech typicky s příponou ".prototxt". Jedná se o prostý textový soubor napsaný pomocí syntaktických pravidel Proto. Definice jednotlivých částí které lze použít při popisu modelu jsou definovány v souboru caffe/src/caffe/proto/caffe.proto. Tento soubor slouží také jako definice syntaxe pro parser a automatické vytvoření definice tříd ve zdrojovém kódu z definovaných částí. Díky tomuto mechanismu lze snadno přidat vlastní definice částí sítě bez nutnosti upravovat kód parseru.

```
name: "convolution"
input: "data"
input_dim: 1
input_dim: 1
input_dim: 100
input_dim: 100
layer {
  name: "conv"
  type: "Convolution"
  bottom: "data"
  top: "conv"
  convolution_param {
    num_output: 3
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
```

Výpis 4.2: Textová definice modelu sítě použitá v obrázku 4.1



Obrázek 4.1: Příklad vygenerovaného grafu z textové definice modelu pomocí Pycaffe

4.1.3 Solver

Solver je popsán v souborech typicky s příponou ".prototxt". Jedná se o prostý textový soubor napsaný pomocí syntaktických pravidel Proto. Definice jednotlivých parametrů které lze použít jsou definovány v souboru caffe/src/caffe/proto/caffe.proto. V solveru jsou popsány popsány jednotlivé parametry pro trénování neuronové sítě jako např. počáteční koeficient učení, cesta k souboru s popisem modelu sítě apod.

```
net: "models/bvlc_googlenet/train_val.prototxt"
test_iter: 1000
test_interval: 4000
test_initialization: false
display: 40
average_loss: 40
base_lr: 0.01
.
.
.
```

Výpis 4.3: Část solveru

4.1.4 Váhy a biasy

Váhy a biasy neuronových sítí jsou obsažené v binárních souborech s příponou ".caffemodel". Tyto soubory jsou výstupem trénování pomocí frameworku Caffe. Můžou být použity pro pokračování trénování sítě nebo pro nasazení sítě společně s jejím popisem.

4.2 Vrstvy

Caffe disponuje velkým množstvím předdefinovaných vrstev. Definice jednotlivých vrstev které lze použít při popisu modelu jsou definovány v souboru caffe/src/caffe/proto/caffe.proto. Každá vrstva je implementovaná jako třída v /src/caffe/layers/ a /include/caffe/layers/. Každá třída zdědí virtuální metody dopředné a zpětné propagace pro různá zařízení jako např. `forward_cpu` a `forward_gpu` z třídy Layers, které musí sama implementovat.

Všechny vrstvy mají společné povinné parametry jako jméno, typ, vstupní vrstva atd. Určité vrstvy obsahují své vlastní parametry např. konvoluční vrstva. V následující části jsou uvedeny důležité vrstvy pro konvoluční neuronové sítě.

4.2.1 Vstupní vrstva

Vstupní vrstva přijímá jako svůj vstup databázi dvojic obrázek-klíč ve formátu Leveldb, Lmdb a HDF5 v poli `data_param`, kde je zároveň specifikována velikost dávky. Na rozdíl od jiných vrstev tato vrstva nemá specifikovanou vstupní vrstvu, jelikož žádnou nepřijímá. Vrstva může přijímat více vstupů zároveň. Pomocí pole `include` lze specifikovat zda-li se vstup bude používat při trénování nebo testování.

V uvedeném příkladu vstupní vrstva "data" přijímá jako vstup databázi typu LMBD uloženou v examples/imagenet/ilsvrc12_train_lmdb. Data tohoto vstupu se zpracovávají po 256 vzorcích najednou. Tato vstupní vrstva je použita pouze při trénování.

```
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_train_lmdb"
    batch_size: 256
    backend: LMDB
  }
}
```

Výpis 4.4: Vstupní vrstva

4.2.2 Konvoluční vrstva

Konvoluční vrstva přijímá vstup z předchozí vrstvy a provádí nad ním konvoluci. Obsahuje speciální pole `convolution_param`, kde jsou specifikovány parametry pro konvoluční vrstvu. Lze nastavit rozměry filtru, krok filtru, odsazení (padding). Kromě samotných vlastností filtru lze nastavit pravidla pro inicializaci počátečních hodnot filtru.

V uvedeném příkladu konvoluční vrstva "conv1" přijímá jako svůj vstup výstup vrstvy "data". Obsahuje celkem 96 různých filtrů pro učení, přičemž každý filtr má velikost 11x11. Filtr se posouvá po vstupu s krokem délky 4.

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4
        .
        .
        .
```

Výpis 4.5: Část konvoluční vrstvy

4.2.3 Pooling vrstva

Pooling vrstva snižuje rozměry předchozí vrstvy (obvykle konvoluční). V poli `pooling_param` je možné specifikovat parametry, které určují typ použité funkce (max nebo mean) a její parametry. Vrstva která následuje po pooling vrstvě musí počítat s rozdílnými rozměry.

V uvedeném příkladu konvoluční vrstva "pool1" přijímá jako svůj vstup výstup vrstvy "conv1". Jako typ funkce používá max pooling, jejíž filtr má rozměry 3x3 s krokem o velikosti 2.

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
```

Výpis 4.6: Pooling vrstva

4.2.4 ReLU

ReLU je vrstva realizující ReLU aktivační aktivační funkci ve formě $f(x) = max(x, 0)$, která filtruje záporné výstupy předchozí vrstvy. Parametrem `negative_slope` lze implementovat tzv. Leaky-ReLU, které zmenší záporná čísla na základě číselného parametru. Její definice je následovná:

$$f(x) = \begin{cases} \text{x}, & \text{pro x} > 0 \\ \text{ax}, & \text{jinak} \end{cases} \tag{4.1}$$

Kde a je parametr Leaky-ReLU.

V uvedeném příkladu ReLU vrstva "relu1" přijímá jako svůj vstup výstup vrstvy "conv1". ReLU vrstva neobsahuje žádné další parametry.

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
```

Výpis 4.7: ReLU

4.2.5 Plně propojená vrstva

Jednoduchá vrstva, která převede vstup do jednoduchého jednorozměrného vektoru pomocí maticového násobení. Většinou se používá na konci konvoluční sítě pro klasifikaci příznaků a získání výsledné klasifikace.

V uvedeném příkladu plně propojená vrstva "fc8" přijímá jako svůj vstup výstup vrstvy "fc7". Jejím výstupem je vektor 1000 čísel. Její váhy jsou inicializovány podle normálního rozložení pravděpodobnosti s $\sigma = 0.01$. Biasy jsou inicializovány na hodnotu 0.

```
layer {
  name: "fc8"
  type: "InnerProduct"
  inner_product_param {
```

```
    num_output: 1000
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
  bottom: "fc7"
  top: "fc8"
}
```

Výpis 4.8: Plně propojená vrstva

4.2.6 Vrstvy pro měření a výpis informací

Vrstvy, které slouží pro měření a výpis informací během trénování a testování sítě.

V uvedeném příkladu vrstva pro měření přesnosti "loss3/top-1" přijímá jako svůj vstup výstup vrstev "loss3/classifier" a "label". Vrstva je inicializována při testovací fázi.

```
layer {
  name: "loss3/top-1"
  type: "Accuracy"
  bottom: "loss3/classifier"
  bottom: "label"
  top: "loss3/top-1"
  include {
    phase: TEST
  }
}
```

Výpis 4.9: Vrstva pro měření přesnosti

4.2.7 Vrstva chybové funkce

Vrstva počítající chybovou funkci neuronové sítě. Trénování sítí v Caffe je řízené touto funkcí, proto je klíčové. Často se výpočet chybové funkce spojuje s výpočtem výsledné vrstvy pomocí Softmax (SoftmaxWithLoss).

V uvedeném příkladu vrstva chybové funkce "loss" přijímá jako svůj vstup výstup vrstev "pred" a "label". Vrstva chybové funkce neobsahuje žádné další parametry.

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
```

```
}
```

Výpis 4.10: Vrstva chybové funkce

4.3 Konzolové prostředí a základní operace

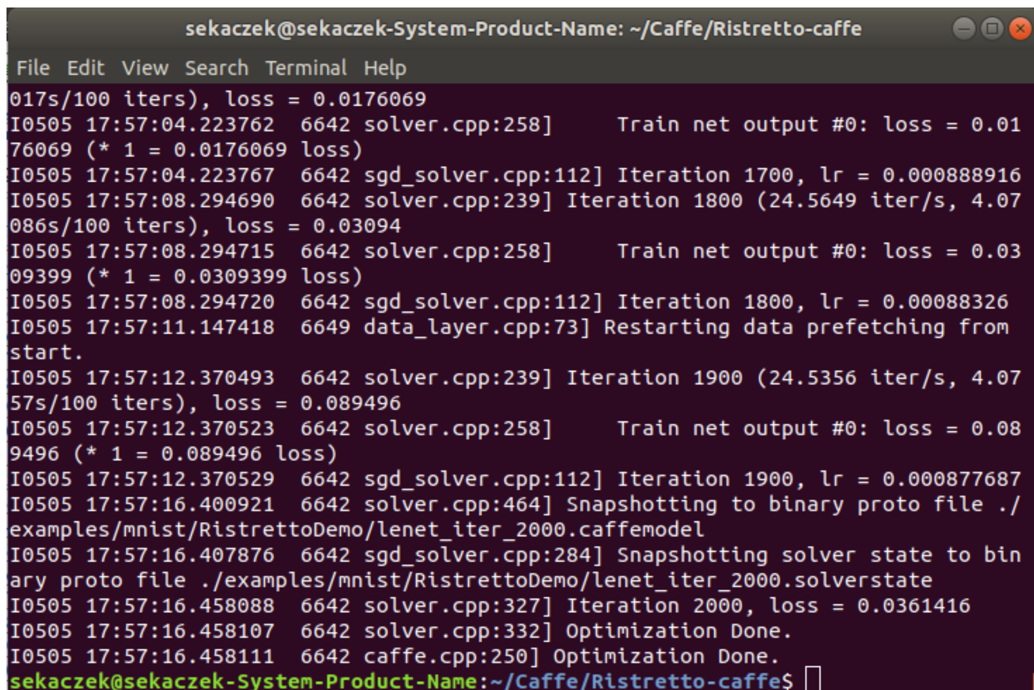
Framework je možno ovládat z příkazové řádky pomocí spustitelné formy cmdcaffe. Při spuštění bez jakýchkoliv argumentů je zobrazena nápověda. Mezi základní příkazy této konzolové apliakce patří:

4.3.1 Trénování

Trénování je základní operace pro natrénování určitého popisu modelu neuronové sítě. Pro trénování je potřeba definici modelu sítě, solver a případně již natrénovaný model sítě, pokud je cílem doučit již natrénovanou síť. Výsledkem trénování je soubor obsahující natrénovaný model neuronové sítě. Trénování může být kdykoliv ukončeno signálem `SIGINT` a současný stav sítě bude uložen.

```
caffe train -solver cesta_k_solveru -gpu=0
```

Výpis 4.11: Příkaz pro spuštění trénování



Obrázek 4.2: Ukázka trénování

4.3.2 Finetuning

Z pohledu Caffe je finetuning varianta trénování. Pro finetuning je potřeba stejných věcí jako pro trénování a binární soubor který obsahuje natrénovaný model neuronové sítě.

```
caffe train -solver cesta_k_solveru -weights cesta_k_vaham -gpu=0
```

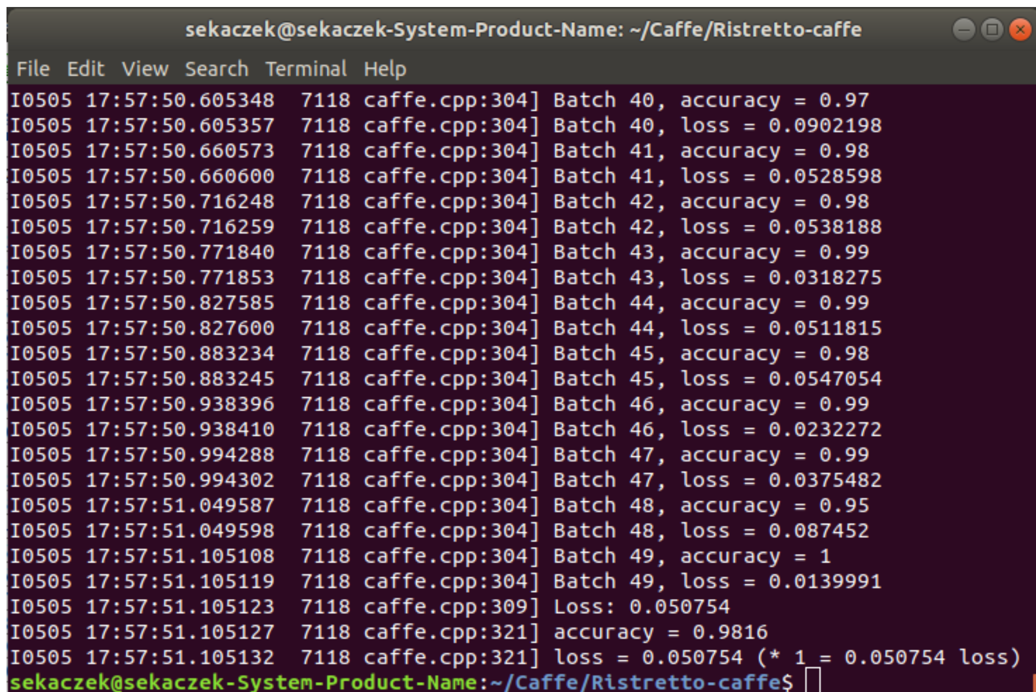
Výpis 4.12: Příkaz pro spuštění finetuningu

4.3.3 Testování

Testování slouží k změření přesnosti již natrénovaného modelu sítě. Pro testování je potřeba natrénovaný model sítě a popis modelu sítě.

```
caffe test -model cesta_k_modelu -weights cesta_k_vaham -gpu=0
```

Výpis 4.13: Příkaz pro spuštění testování



Obrázek 4.3: Ukázka trénování

4.4 Rozšíření Ristretto

Ristretto je rozšíření frameworku Caffe, které umožňuje provádět kvantování nad již natrénovaným modelem sítě. Rozšíření za tímto účelem definuje svoje vlastní vrstvy, které obsahují parametry kvantování.

```
quantization_param {
bw_layer_in: 8
bw_layer_out: 8
bw_params: 8
fl_layer_in: 0
fl_layer_out: -3
fl_params: 7
```

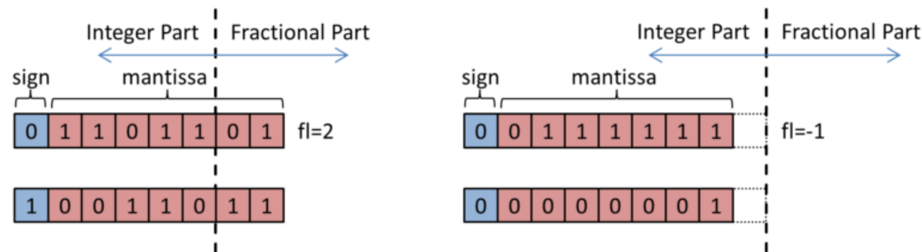
```
}
```
Výpis 4.14: Příklad parametrů pro kvantování

Pro kvantování modelu sítě je potřeba popis jeho modelu a jeho natrénovaný model. Ristretto umožňuje tři typy kvantování: dynamic fixed point, minifloat a Power-of-two parameters. Po kvantování sítě je zpravidla dobrým postupem provést finetuning, aby se síť lépe přizpůsobila novým parametrům.

```
ristretto quantize --model=cesta_k_modelu \
  --weights=cesta_k_vaham \
  --model_quantized=cesta_k_vyslednemu_kvantizovanemu_modelu \
  --iterations=100 --gpu=0 --trimming_mode=dynamic_fixed_point --error_margin=1
```
Výpis 4.15: Příklad spuštění kvantizace používající dynamic fixed point

4.4.1 Dynamic fixed point

V sítích s vrstvami větších rozměrů není potřeba vysokého rozpětí hodnot které nabízí `floating point`. Parametry v těchto vrstvách jsou většinou malá čísla na které by nám stačil datový typ s menší bitovou šířkou. Fixed point je limitovaný malým rozsahem hodnot, proto se používá dynamic fixed point[3]. Dynamic fixed point je vypočítán jako $n = (-1)^s \cdot 2^{-fl} \cdot \sum_{i=0}^{B-2} 2^i \cdot x_i$, kde B je bitová šířka, s znaménkový bit, a fl je délka desetinné části.



Obrázek 4.4: Příklady dynamic fixed point čísel. Převzato z [2]

4.4.2 Minifloat

Minifloat funguje na stejným principu jako normální float, akorát se liší od IEEE standartu svou bitovou šířkou a jinými parametry. Části vymezené pro mantisu a exponent jsou volně zvoleny ristrettem. Implementace minifloatu v Ristrettu se liší od IEEE standartu tím, že denormalizovaná čísla jako *NaN* a *INF* jsou nahrazeny nulou.

Obrázek 4.5: Mini floating point o bitové šířce 16. Převzato z [2]

Kapitola 5

Implementace a výsledky

5.1 Implementace

Záměrem této práce bylo výsledné řešení implementovat pomocí grafických akcelerátorů, i přesto že CPU by bylo lepší pro simulaci aproximovaných jednotek. Důvodem je fakt, že zpětné šíření chyby je výpočetně náročné. Aproximována byla pouze dopředná propagace, jelikož aproximace zpětného šíření chyby by pouze zpomalila nebo dokonce znemožnila trénování. Aproximované jsou pouze operace násobení v konvolučních vrstvách, protože násobičky mají největší dopad na příkon, plochu na čipu a zpoždění. V následující části jsou popsány řešení jednotlivých podproblémů, které vedli k výslednému řešení.

5.1.1 Kvantování modelu sítě

Jako typ kvantování byl použit dynamic fixed point, protože byly použity komponenty z knihovny EvoApprox, které pracují s celými čísly. Ristretto volí různé bitové šířky (16, 8, 4, 2 bity) které otestuje a změří jejich přesnost, na základě které zvolí nejefektivnější řešení. Kvantování modelu sítě bylo potřeba upravit, tak aby nedocházelo k přetečení a zároveň byla využita celá komponenta. Pro jednoduchost byly zvolené pouze 8 bitové komponenty z knihovny EvoApprox.

Potřeba pro změnu kvantovacího algoritmu byla vyřešena přidáním argumentu příkazové řádky `--bitwidth=[8|any]`. Pokud je kvantování voláno s argumentem `--bitwidth=8`, všechny vygenerované vrstvy mají svoji bitovou šířku nastavenou na 8 a vyhodnocení ostatních bitových šířek je přeskočeno. Zároveň je zajištěno, aby nejvýznamnější bit celé části měl maximálně hodnotu $2^7 = 128$, kvůli přetečení při použití 8 bitových aproximovaných komponent.

5.1.2 Specifikování použití aproximované komponenty

Je potřeba nějakým způsobem zadat zda-li daná konvoluční vrstva používá komponentu a o jakou konkrétní komponentu se jedná. Parsování v Caffe je vyřešeno pomocí knihovny Protobuf, kde definice jednotlivých vrstev a jejich parametrů jsou uloženy v "proto" souboru, díky tomu je možné jednoduše upravit nebo přidat novou definici vrstvy.

Místo definice zcela nové vrstvy byla rozšířena již existující vrstva `conv_ristretto_layer`. Specificky pole `ConvolutionParameter`, které se nachází ve vrstvě `conv_ristretto_layer`. Do pole byly přidány parametry `approx_circ_enable` a `approx_circ_path`.

`Approx_circ_enable` je hodnota booleovského typu která specifikuje zda-li se má použít v dané vrstvě aproximovaná jednotka. `approx_circ_path` je textový řetězec který udává cestu k souboru, kde jsou uloženy data konkrétní aproximované jednotky.

5.1.3 Realizace aproximovaných komponent

Knihovna EvoApprox nabízí implementace svých komponent v jazyce C nebo verilogu. Verilog je jazyk sloužící pro hardwarový popis (HDL – Hardware Description Language). Pro účely této práce byla zvolena implementace v jazyce C.

Simulace aproximované komponenty, kde jsou vyhodnoceny jednotlivé hradla pomocí logických funkcí je příliš náročná. Místo zavedení implementací všech aproximovaných jednotek byla zvolena vyhledávací tabulka. Výhodou je, že časová složitost takové vyhledávací tabulky je konstantní. U 8 bitových jednotek je počet všech možných kombinací pro dva operandy $256^2 = 65536$, přičemž každý výsledek je typu `short`. To znamená, že každá vrstva která používá aproximovanou jednotku potřebuje 128KiB paměti, což je zanedbatelné vzhledem k velikosti dnešních pamětí. Každá vrstva si při své inicializaci načte svou vyhledávací tabulku podle cesty uvedené jako parametr `approx_circ_path`. Tato tabulka je následně zkopírovaná do paměti GPU.

Vyhledávací tabulky jsou uložené v binárních souborech. Tyto binární soubory jsou generované pomocí skriptu "LUT_gen" napsaného v jazyce python. Tento skript vyžaduje složku s aproximovanými komponenty a textový soubor "template.txt" kde je uložen předpis funkce `main`. Funkce `main` postupně volá funkci dané aproximované jednotky a generuje binární soubor.

Skript prochází všechny podsložky ve složce "8x8_unsigned" a hledá soubory s příponou ".c". Jakmile najde takový soubor vygeneruje nový C zdrojový soubor, který vznikne spojením funkce jednotky a funkce `main` z "template.txt". Tento vygenerovaný soubor je přeložen překladačem GCC a spuštěn, čímž vygeneruje binární soubor ve kterém je uložena vyhledávací tabulka dané jednotky.

5.1.4 Výpočet konvoluce

Samotná konvoluce je implementována jako maticové násobení. Tato operace je implementována pomocí CUDY na grafické kartě jako kernel `gpu_gemm_approx`. Tento kernel je spuštěn z funkce `caffe_gpu_gemm_approx`, která se stará o inicializaci parametrů kernelu a jeho spuštění.

Samotné maticové násobení je implementováno jako `for` cyklus, který postupně sčítá násobky dvou matic do výsledné matice. Každé jádro GPU počítá výsledek jednoho prvku výsledné matice. Vyhledávací tabulka je implementovaná jako texturová paměť pro větší paměťovou propustnost. Jelikož vybrané komponenty z EvoApprox počítají bez znamének je potřeba dodatečně ošetřit znaménko pomocí porovnání znamének obou operandů.

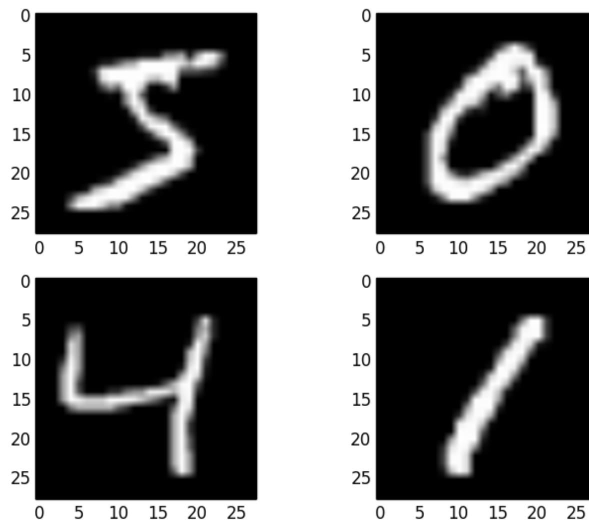
Všechny hodnoty vrstev jsou definovány jako floating point hodnoty i přes kvantování. Kvantování je simulováno zaokrouhlením mantisy floating pointu na určitý bit. Při výpočtech je potřeba převést floating point číslo x s desetinou části o délce fl bitů na celé číslo pomocí vzorce: $x \cdot 2^{fl}$. Při zpětném převodu je použit vzorec $x \cdot 2^{-fl}$.

```
if(x < N && y < M){
        C[x + y*N] = 0;
        for(int i = 0; i < K; i++){
                Aval = A[i + y*K] * powf(2, flA);
                Bval = B[x + i*N] * powf(2, flB);
                C[x + y*N] +=
                //nasobeni dvou cisel pomoci LUT
                tex1Dfetch(approx_LUT_tex, ((Aval << 8) | Bval))
                * powf(2, -(flA+flB))      //prevod zpet do floating point
                * ((float(0) < (A[i + y*K]*B[x + i*N])) //osetreni znamenka
                - ((A[i + y*K]*B[x + i*N]) < float(0)));
        }
}
```

Výpis 5.1: Část kernelu realizující maticové násobení

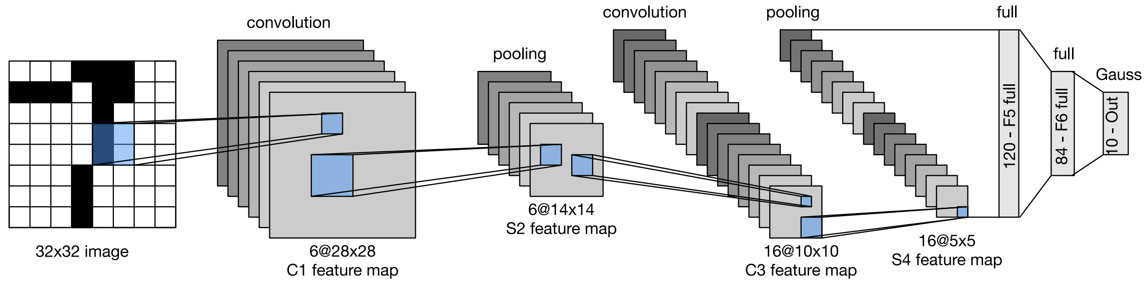
5.2 Testovaný model

Jako model na kterém bylo výsledné řešení testováno byl zvolen "LeNet-5"[6]. Jedná se o jednoduchý model navrhnutý pro rozpoznání ručně napsaných číslic. Jako trénovací a testovací sada byla zvolena databáze "MNIST"[1]. MNIST databáze se skládá ze 70 000 černobílých obrázků ručně napsaných číslic s rozlišením 28x28.



Obrázek 5.1: Příklad obrázků z databáze MNIST. Převzato z [1]

[1] http://yann.lecun.com/exdb/mnist/



Obrázek 5.2: Architektura modelu LeNet. Převzato z [23]

5.2.1 Postup

Síť byla nejprve normálně natrénována na FP32 s 10 000 iteracemi a počátečním koeficientem učení $\eta = 0.01$. Nálsedně byla kvantována pomocí Ristretta. Výsledný kvantovaný model byl použit pro otestování aproximovaných komponent. Pro testování bylo zvoleno několik aproximovaných komponent. Každý typ aproximované komponenty byl použit v obou konvolučních vrstvách. Pro každou komponentu byl proveden desetkrát finetuning. Po každém finetuningu byl výsledný model otestován, čímž se také změřila jeho přesnost. Pro testování byly použity následující komponenty:

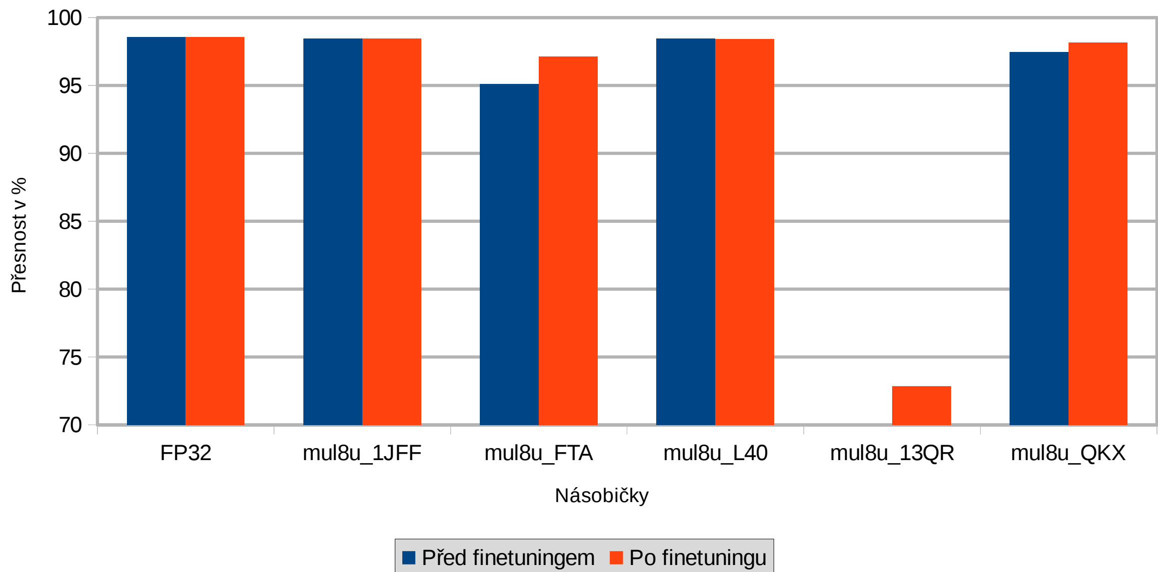
Aprox. komponenta	MAE(%)	WCE(%)	EP(%)	MRE(%)	MSE
mul8u_1JFF	0.00	0.00	0.00	0.00	0
mul8u_FTA	0.89	4.29	98.74	13.96	543210
mul8u_L40	1.54	13.92	74.91	7.46	36892.825e2
mul8u_13QR	4.83	19.46	99.20	44.00	15608.397e3
mul8u_QKX	5.09	49.23	97.47	21.95	34405.106e3

5.3 Výsledky

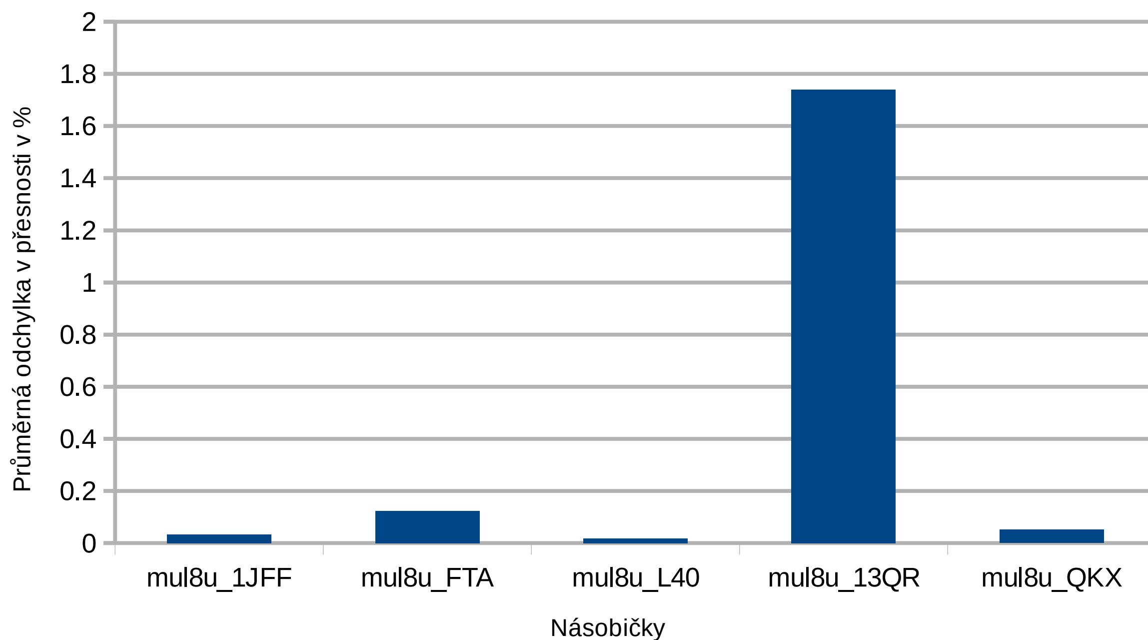
Z naměřených výsledků lze vidět, že se přesnost drasticky nesnížila. Jedinou výjimkou byla komponenta mul8u_13QR, která dokonce před finetuningem dosahovala přesnosti 0% a i po finetuningu dosahovala přesnosti pouze 72.851%. Taky to je jediná komponenta, jejíž přesnost se výrazně lišila v jednotlivých měření.

U ostatních komponent je snížení v přesnosti podstatně menší. Důležité je si všimnout, že samotná průměrná chyba ve výsledku (MAE) neurčuje nutně kvalitu komponenty v síti co se týče přesnosti. Důležité je také brát ohled na pravděpodobnost výskytu chyby (EP) a relativní chybu (MRE). Například mul8u_L40 má pravděpodobnost výskytu chyby pouze 74.91%, což může značit že je přesnější v jistém rozsahu hodnot, který může být klíčový pro danou neuronovou síť.

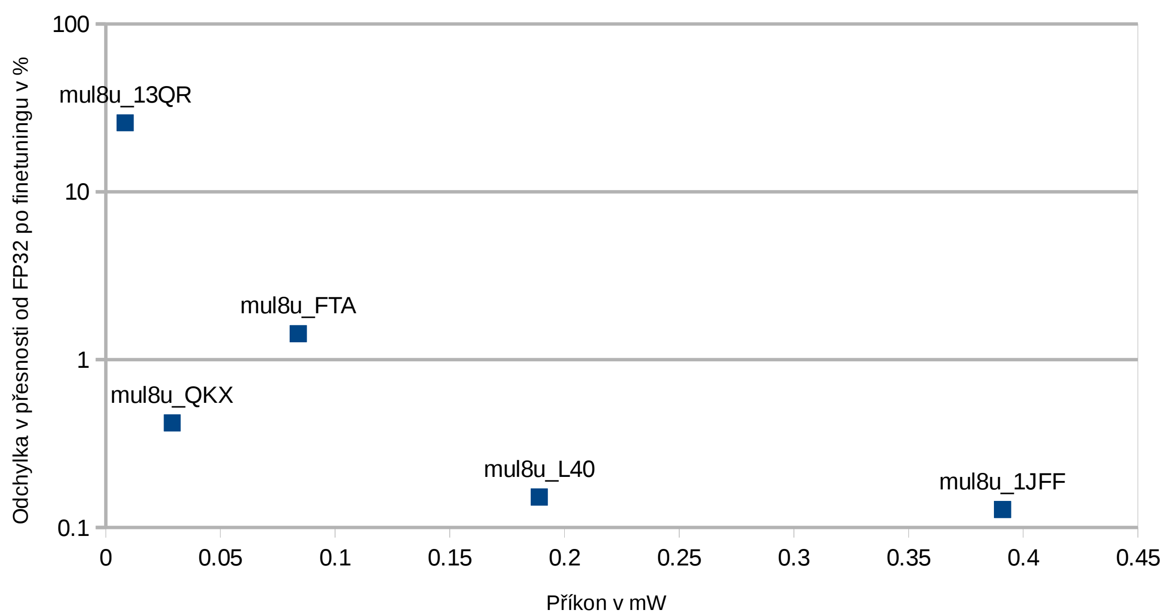
V následující části jsou uvedeny veškeré výsledky měření, včetně srovnání dosažených přesností s technickými parametry jednotlivých aproximovaných komponent, které byly získány z dokumentace EvoApprox.



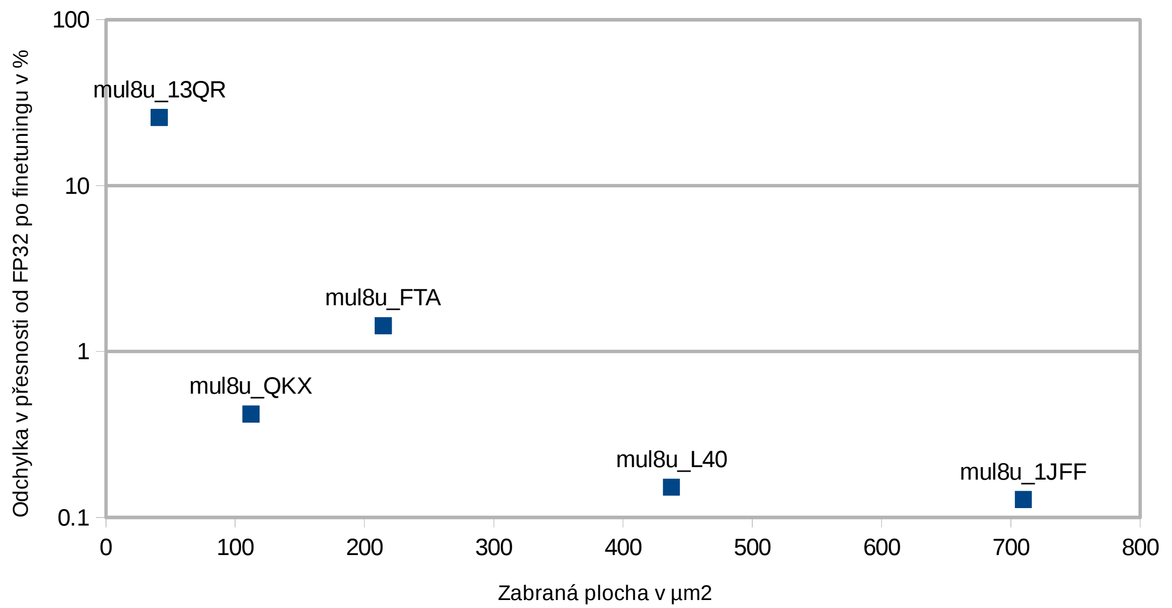
Obrázek 5.3: Dosažená přesnost různých aproximačních jednotek



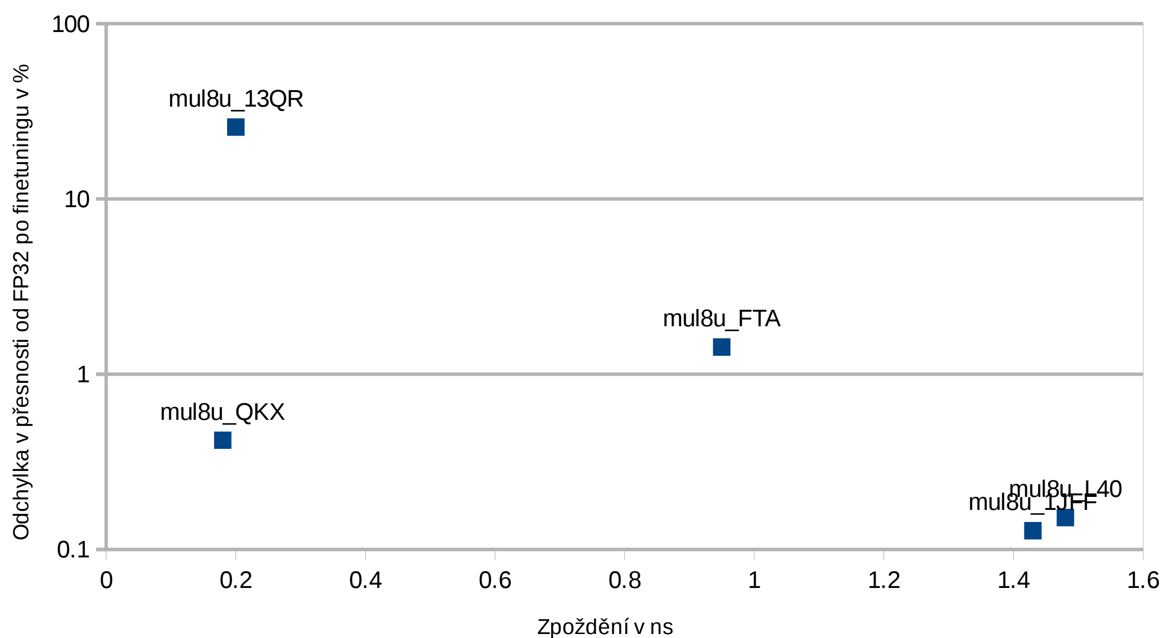
Obrázek 5.4: Průměrná odchylka v přesnosti u jednotlivcích měření



Obrázek 5.5: Srovnání příkonu a odchylky v přesnosti



Obrázek 5.6: Srovnání plochy na čipu a odchylky v přesnosti



Obrázek 5.7: Srovnání zpoždění a odchylky v přesnosti

Kapitola 6

Závěr

Cílem této práce bylo zjistit přínos aproximování výpočtů v neuronových sítích. Hlavní motivací pro tuto práci byla optimalizace hardwarových implementací akcelerátorů pro výpočty operací neuronových sítích. Nejprve bylo potřeba vybrat framework, který by byl rozšířen o použití aproximovaných výpočtu. V tomto případě jsem si zvolil Ristretto-caffe, jelikož jde o open source framework který má již v sobě implementované kvantování a podporuje akceleraci pomocí GPU. Pro aproximované komponenty jsem si zvolil knihovnu EvoApprox, protože se jedná o open source knihovnu, která obsahuje velké množství komponent s přehlednými statistikami a implementací v jazyce C. Při implementaci výsledného řešení jsem se snažil brát ohled na rychlost, ovšem nebyl to hlavní záměr. V budoucnu by mohl někdo další zkusit optimalizovat mé řešení, nebo navrhnout jiný postup.

Výsledné řešení jsem implementoval pomocí CUDY na GPU. Aproximoval jsem pouze násobení, jelikož násobičky mají vysoký dopad na spotřebu, zabranou plochu a rychlost. Možná by bylo vhodné zkusit aproximovat i sčítání a zjistit dopad této aproximace na efektivitu výpočtu. Řešení jsem testoval na jednoduchém modelu, kvůli časovým nárokům. Určitě by bylo vhodné do budoucna otestovat výsledné řešení na složitějších modelech jako např. GoogleNet.

Z výsledků testování na modelu LeNet-5 bylo vidět že se neuronová síť po finetuningu uměla přizpůsobit použitým aproximovaným komponentám. Ztráta na přesnosti byla u většiny z nich minimální, zatímco přínos z hlediska příkonu, zabrané plochy a zpoždění byl velmi značný. Tento výsledek naznačuje že by aproximované komponenty bylo možno použít pro optimalizaci výpočtu inference v hardwarových akcelerátorech. Podle mého názoru se cíle projektu podařilo splnit. Dalším postupem by mohlo být navržení programu, který se automaticky snaží najít optimální model sítě s použitím aproximovaných komponent.

Literatura

[1] Brownlee, J.: *Handwritten Digit Recognition using Convolutional Neural Networks in Python with Keras*. [Online; navštíveno 07.05.2019].
URL https://machinelearningmastery.com/handwritten-digit-recognition-using-convolutional-neural-networks-python-keras/

[2] Gysel, P.; Pimentel, J.; Motamedi, M.; aj.: Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2018, doi:10.1109/TNNLS.2018.2808319.
URL https://ieeexplore.ieee.org/document/8318896

[3] Hubara, I.; Courbariaux, M.; Soudry, D.; aj.: Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research*, ročník 18, 2017: s. 187:1–187:30.
URL http://jmlr.org/papers/v18/16-456.html

[4] Jia, Y.; Shelhamer, E.; Donahue, J.; aj.: Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.
URL https://arxiv.org/abs/1408.5093

[5] Kaz Sato, D. P., Cliff Young: *An in-depth look at Google's first Tensor Processing Unit (TPU)*. [Online; navštíveno 30.04.2019].
URL https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[6] Lecun, Y.; Bottou, L.; Bengio, Y.; aj.: Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998, s. 2278–2324.

[7] Mcculloch, W.; Pitts, W.: A Logical Calculus of Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, ročník 5, 1943: s. 127–147.

[8] Minsky, M.; Papert, S.: *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[9] Mittal, S.: A Survey of Techniques for Approximate Computing. *ACM Computing Surveys*, 2016.
URL https://www.osti.gov/pages/biblio/1286958

[10] Mrázek, V.; Hrbáček, R.; Vašíček, Z.; aj.: EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods. In *Proc. of the 2017 Design, Automation & Test in Europe Conference &*

Exhibition (DATE), European Design and Automation Association, 2017, ISBN 978-3-9815370-9-3, s. 258–261, doi:10.23919/DATE.2017.7926993.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=11262

[11] Oranchak, D.: *Cartesian Genetic Programming for the Java Evolutionary Computing Toolkit (CGP for ECJ)*. [Online; navštíveno 1.05.2019].
URL http://www.oranchak.com/cgp/doc/

[12] Perez, C. E.: *Google's AI Processor's (TPU) Heart Throbbing Inspiration*. [Online; navštíveno 02.05.2019].
URL https://medium.com/intuitionmachine/googles-ai-processor-is-inspired-by-the-heart-d0f01b72defe

[13] Rupp, K.: *CPU, GPU and MIC Hardware Characteristics over Time*. [Online; navštíveno 06.05.2019].
URL https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

[14] Saha, S.: *A Comprehensive Guide to Convolutional Neural Networks*. [Online; navštíveno 29.04.2019].
URL https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[15] Volná, E.: *NEURONOVÉ SÍTĚ 1*. Ostravská univerzita v Ostravě, 2008.
URL http://www1.osu.cz/~volna/Neuronove_site_skripta.pdf

[16] Warden, P.: *Why GEMM is at the heart of deep learning*. [Online; navštíveno 08.05.2019].
URL https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

[17] Werbos, P. J.: *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Dizertační práce, Harvard University, 1974.

[18] *Umělá neuronová síť*. Wikipedia: the free encyclopedia. [Online; navštíveno 27.04.2019].
URL https://cs.wikipedia.org/wiki/Um%C4%9Bl%C3%A1_neuronov%C3%A1_s%C3%AD%C5%A5

[19] *Types of artificial neural networks*. Wikipedia: the free encyclopedia. [Online; navštíveno 27.04.2019].
URL https://en.wikipedia.org/wiki/Types_of_artificial_neural_networks

[20] *Convolutional neural network*. Wikipedia: the free encyclopedia. [Online; navštíveno 29.04.2019].
URL https://en.wikipedia.org/wiki/Convolutional_neural_network

[21] *Gradient descent*. Wikipedia: the free encyclopedia. [Online; navštíveno 07.05.2019].
URL https://en.wikipedia.org/wiki/Gradient_descent

[22] *Neural circuit*. Wikipedia: the free encyclopedia. [Online; navštíveno 26.04.2019].
URL https://en.wikipedia.org/wiki/Neural_circuit

[23] Zhang, A.; Lipton, Z. C.; Li, M.; aj.: *Dive into Deep Learning*. 2019, http://www.d2l.ai.

Příloha A

Obsah přiloženého paměťového média

Na přiloženém DVD jsou uloženy veškeré zdrojové soubory, dokumentace k instalaci potřebných modulů a manuál pro přeložení ukázkové aplikace. Zde je uveden přesný popis adresářové struktury, která je uložena na DVD disku:

- **caffe/** – Upravené zdrojové soubory frameworku Ristretto–caffe
- **caffe_deps/** – Moduly potřebné pro sestavení Ristretto–caffe
- **doc/IBT** – Zdrojové LATEX soubory pro sestavení PDF souboru této práce, včetně veškerých zdrojů použitých v této práci
- **doc/** – Výsledný (tento) PDF soubor
- **script/** – Pomocné skripty téhle práce

Příloha B

Manuál

B.1 Přeložení frameworku

- Rozšíření byla implementována pouze pro Nvidia grafické karty. Pro správnou funkci frameworku je potřeba mít Nvidia grafickou kartu a nainstalované drivery společně s CUDOU verze 7 a výš.
- Přeložení frameworku bylo testováno na operačním systému Ubuntu 14 společně s grafickou kartou architektury Maxwell.
- Veškeré instrukce jsou v adresáři caffe_deps.

B.2 Použití skriptu pro generování vyhledávacích tabulek

- Ve složce script/ je umístěn python skript "LUT_gen" a další pomocné soubory pro generování binárních vyhledávacích tabulek.
- Skript stačí spustit pomocí pythonu ve stejném adresáři kde je umístěn bez jakýchkoliv argumentů příkazové řádky.
- Vygenerované vyhledávacích tabulky jsou ve složce "LUTs" nacházející se ve složce "LUT_gen".